EIR

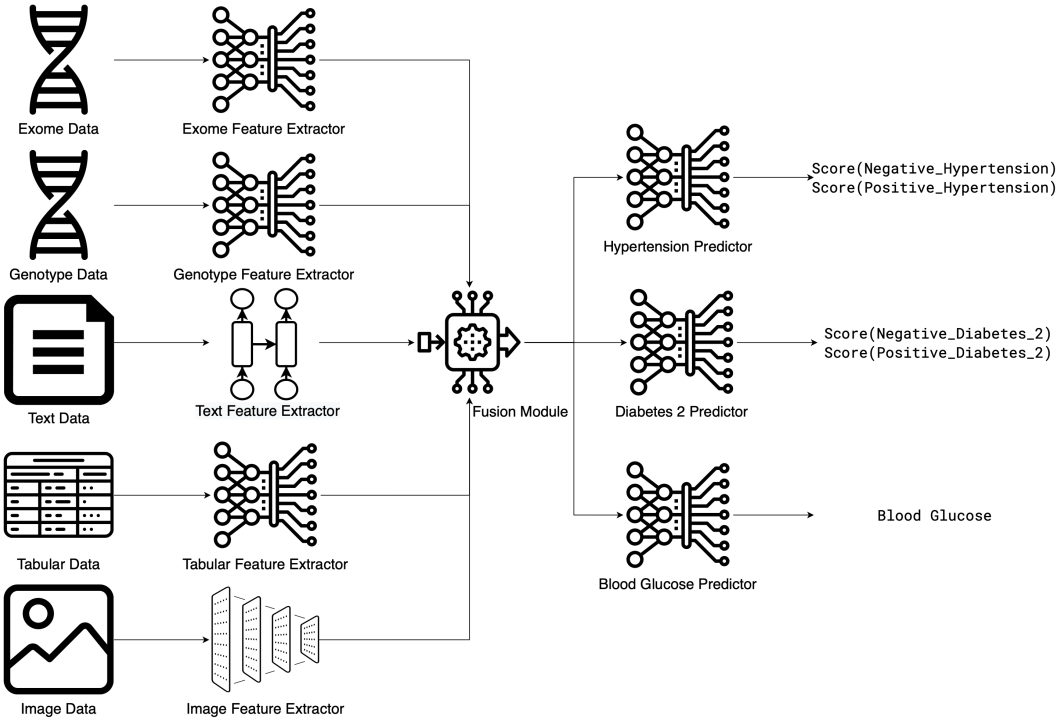
Arnor Sigurdsson

Apr 25, 2024

CONTENTS

EIR is a framework for supervised modelling, sequence generation and array generation on genotype, tabular, sequence, image, array, and binary input data. It is designed to provide a high-level, yet modular API that reduces the amount of boilerplate code and pre-processing required to train a model.

Warning: This project is in alpha phase. Expect backwards incompatible changes and API changes.



CHAPTER

ONE

INSTALLATION

1.1 Installing EIR via pip

```
$ pip install eir-dl
```

Important: The latest version of EIR supports Python 3.11. Using an older version of Python will install an outdated version of EIR, which is likely to be incompatible with the current documentation and may contain bugs. Please make sure that you are installing EIR in a Python 3.11 environment.

1.2 Installing EIR via Container Engine

Here's an example with Docker:

```
$ docker build -t eir:latest https://raw.githubusercontent.com/arnor-sigurdsson/EIR/
↪master/Dockerfile
$ docker run -d --name eir_container eir:latest
$ docker exec -it eir_container bash
```

CHAPTER

TWO

DOCUMENTATION

To get started, please read *01 – Genotype Tutorial: Ancestry Prediction*.

2.1 Supervised Learning

2.1.1 01 – Genotype Tutorial: Ancestry Prediction

A - Setup

In this tutorial, we will be using genotype data to train deep learning models for ancestry prediction.

Note: This tutorial goes into some detail about how `EIR` works, and how to use it. If you are more interested in quickly training the deep learning models for genomic prediction, the EIR-auto-GP project might be of use to you.

To start, please download processed sample data (or process your own *.bed*, *.bim*, *.fam* files with e.g. plink pipelines). The sample data we are using here for predicting ancestry is the public Human Origins dataset, but the same approach can just as well be used for e.g. disease predictions in other cohorts (for example the UK Biobank).

Examining the sample data, we can see the following structure:

```
processed_sample_data
├── arrays                      # Genotype data as NumPy arrays
├── data_final_gen.bim          # Variant information file accompanying the genotype␣
↪arrays
└── human_origins_labels.csv    # Contains the target labels (what we want to predict␣
↪from the genotype data)
```

Important: The label file ID column must be called "ID" (uppercase).

For this tutorial, we are going to use the data above to models to predict ancestry, of which there are 6 classes (Asia, Eastern Asia, Europe, Latin America and the Caribbean, Middle East and Sub-Saharan Africa). Before diving into the model training, we first have to configure our experiments.

To configure the experiments we want to run, we will use `.yaml` configurations. Running `eirtrain --help`, we can see the configurations needed:

```
usage: eirtrain [-h] --global_configs GLOBAL_CONFIGS [GLOBAL_CONFIGS ...]
                [--input_configs [INPUT_CONFIGS ...]]
                [--fusion_configs [FUSION_CONFIGS ...]] --output_configs
                OUTPUT_CONFIGS [OUTPUT_CONFIGS ...]

options:
  -h, --help            show this help message and exit
  --global_configs GLOBAL_CONFIGS [GLOBAL_CONFIGS ...]
                        Global .yaml configurations for the experiment.
  --input_configs [INPUT_CONFIGS ...]
                        Input feature extraction .yaml configurations. Each
                        configuration represents one input.
  --fusion_configs [FUSION_CONFIGS ...]
                        Fusion .yaml configurations.
  --output_configs OUTPUT_CONFIGS [OUTPUT_CONFIGS ...]
                        Output .yaml configurations.
```

Above we can see that there are four types of configurations we can use: *global*, *inputs*, *fusion* and *outputs*. To see more details about what should be in these configuration files, we can check the *Configuration API* reference.

Note: Instead of having to type out the configuration files below manually, you can download them from the `docs/tutorials/tutorial_files/01_basic_tutorial` directory in the project repository

While the **global** configuration has a lot of options, the only one we really need to fill in now is `output_folder` and evaluation interval (in batch iterations), so we have the following `tutorial_01_globals.yaml` file:

Listing 1: tutorial_01_globals.yaml

```yaml
output_folder: eir_tutorials/tutorial_runs/a_using_eir/tutorial_01_run
checkpoint_interval: 200
sample_interval: 200
```

We also need to tell the framework where to load **inputs** from, and some information about the input, for that we use an input `.yaml` configuration called `tutorial_01_inputs.yaml`:

Listing 2: tutorial_01_input.yaml

```yaml
input_info:
  input_source: eir_tutorials/a_using_eir/01_basic_tutorial/data/processed_sample_data/
↪arrays
  input_name: genotype
  input_type: omics

input_type_info:
  snp_file: eir_tutorials/a_using_eir/01_basic_tutorial/data/processed_sample_data/data_
↪final_gen.bim

model_config:
  model_type: genome-local-net
```

Above we can see that the input needs 3 fields: `input_info`, `input_type_info` and `model_config`. The `input_info` contains basic information about the input. The `input_type_info` contains information specific to the input type (in this case *omics*). Finally, the `model_config` contains configuration for the model that should be

trained with the input data. For more information about the configurations, e.g. which parameters are relevant for the chosen models and what they do, head over to the *Configuration API* reference.

Finally, we need to specify what **outputs** to predict during training. For that we will use the `tutorial_01_outputs.yaml` file with the following content:

Listing 3: tutorial_01_outputs.yaml

```yaml
output_info:
  output_name: ancestry_output
  output_source: eir_tutorials/a_using_eir/01_basic_tutorial/data/processed_sample_data/
↪human_origins_labels.csv
  output_type: tabular
output_type_info:
  target_cat_columns:
    - Origin
```

Note: You might notice that we have not written any fusion config so far. The fusion configuration controls how different modalities (i.e. input data types, for example genotype and clinical data) are combined using a neural network. While we indeed *can* configure the fusion, we will leave use the defaults for now. The default fusion model is a fully connected neural network.

With all this, we should have our project directory looking something like this:

```
eir_tutorials/a_using_eir/01_basic_tutorial/
├── conf
│   ├── large_scale_fusion.yaml
│   ├── large_scale_globals.yaml
│   ├── large_scale_input_gln.yaml
│   ├── large_scale_input_tabular.yaml
│   ├── large_scale_output.yaml
│   ├── tutorial_01_globals.yaml
│   ├── tutorial_01_input.yaml
│   ├── tutorial_01_outputs.yaml
│   └── tutorial_01_outputs_unknown.yaml
└── data
    ├── processed_sample_data
    │   ├── arrays
    │   ├── data_final_gen.bim
    │   └── human_origins_labels.csv
    └── processed_sample_data.zip
```

B - Training

Training a GLN model

Now that we have our configurations set up, training is simply passing them to the framework, like so:

```
eirtrain \
--global_configs eir_tutorials/a_using_eir/01_basic_tutorial/conf/tutorial_01_globals.
↪yaml \
```

(continues on next page)

(continued from previous page)

```
--input_configs eir_tutorials/a_using_eir/01_basic_tutorial/conf/tutorial_01_input.yaml \
--output_configs eir_tutorials/a_using_eir/01_basic_tutorial/conf/tutorial_01_outputs.
↪yaml
```

This will generate a folder in the current directory called `eir_tutorials`, and `eir_tutorials/tutorial_runs/a_using_eir/tutorial_01_run` (note that the inner run name comes from the value in `global_config` we set before) will contain the results from our experiment.

Tip: You might try running the command above again after it partially/completely finishes, and most likely you will encounter a `FileExistsError`. This is to avoid accidentally overwriting previous experiments. When performing another run, we will have to delete/rename the experiment, or change it in the configuration (see below).

Examining the directory, we see the following structure (some files have been excluded here for brevity):

```
eir_tutorials/tutorial_runs/a_using_eir/tutorial_01_run/
├── configs
├── meta
│   └── eir_version.txt
├── model_info.txt
├── results
│   └── ancestry_output
│       └── Origin
│           ├── samples
│           │   ├── 200
│           │   │   ├── confusion_matrix.pdf
│           │   │   ├── mc_pr_curve.pdf
│           │   │   ├── mc_roc_curve.pdf
│           │   │   └── predictions.csv
│           │   ├── 400
│           │   │   ├── confusion_matrix.pdf
│           │   │   ├── mc_pr_curve.pdf
│           │   │   ├── mc_roc_curve.pdf
│           │   │   └── predictions.csv
│           │   └── 600
│           │       ├── confusion_matrix.pdf
│           │       ├── mc_pr_curve.pdf
│           │       ├── mc_roc_curve.pdf
│           │       └── predictions.csv
│           ├── training_curve_ACC.pdf
│           ├── training_curve_AP-MACRO.pdf
│           ├── training_curve_LOSS.pdf
│           ├── training_curve_MCC.pdf
│           └── training_curve_ROC-AUC-MACRO.pdf
├── saved_models
├── test_predictions
│   ├── known_outputs
│   │   ├── ancestry_output
│   │   │   └── Origin
│   │   │       ├── confusion_matrix.pdf
│   │   │       ├── mc_pr_curve.pdf
│   │   │       ├── mc_roc_curve.pdf
```

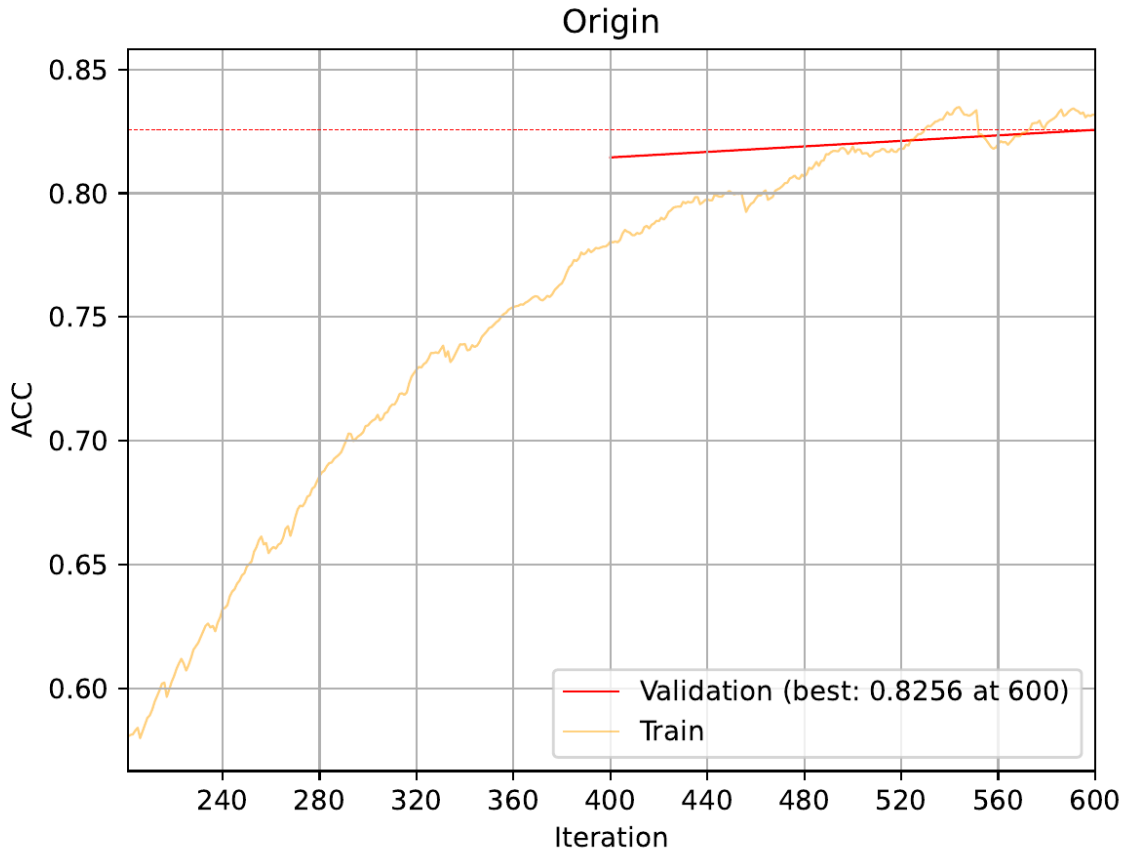
(continues on next page)

(continued from previous page)

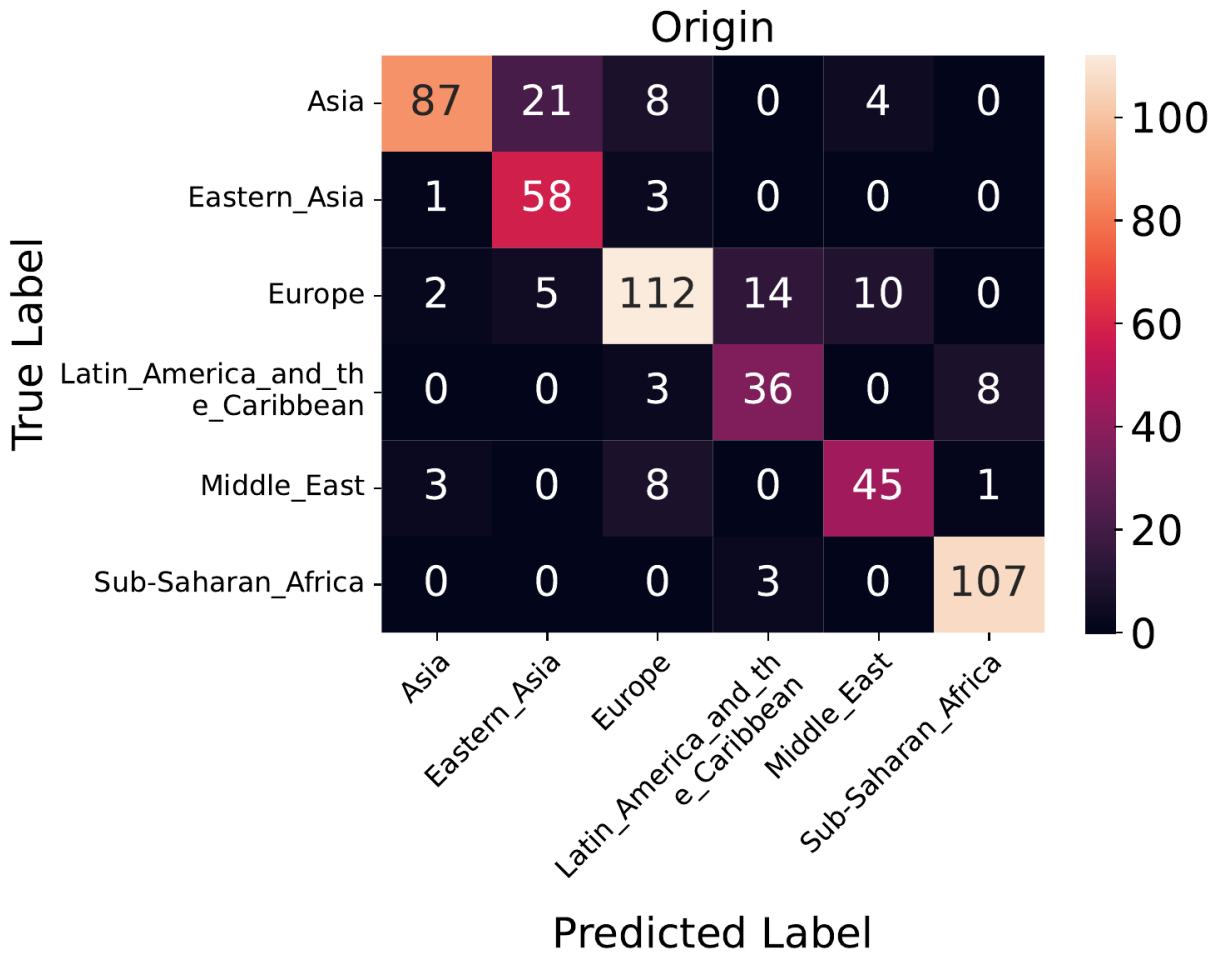
```
│   │   │       └── predictions.csv
│   │   └── calculated_metrics.json
│   └── unknown_outputs
│       └── ancestry_output
│           └── Origin
│               └── predictions.csv
├── training_curve_LOSS-AVERAGE.pdf
└── training_curve_PERF-AVERAGE.pdf
```

In the results folder for a given output, the [200, 400, 600] folders contain our validation results according to our `sample_interval` configuration in the global config.

We can examine how our model did with respect to accuracy (let's assume our targets are fairly balanced in this case) by checking the *training_curve_ACC.png* file:



Examining the actual predictions and how they matched the target labels, we can look at the confusion matrix in one of the evaluation folders of `results/Origin/samples`. When I ran this, I got the following at iteration 600:

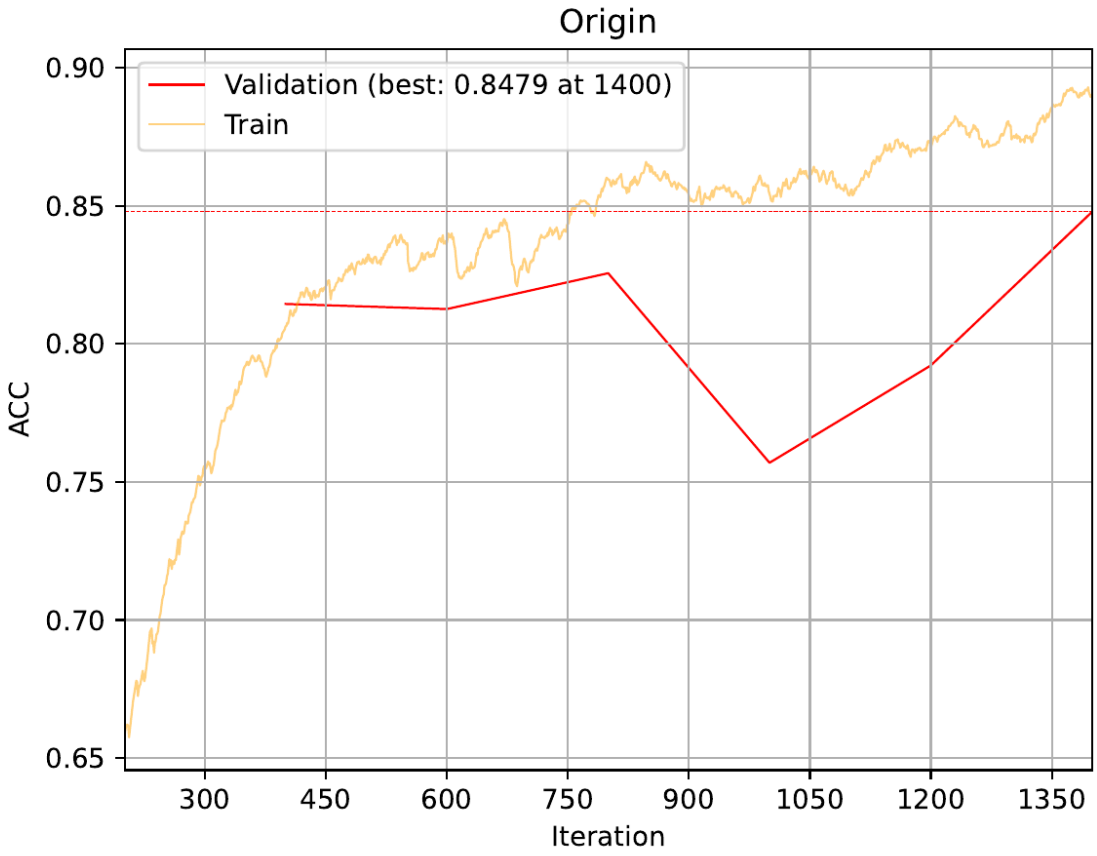


In the training curve above, we can see that our model barely got going before the run finished! Let's try another experiment. We can change the `output_folder` value in `01_basic_tutorial/tutorial_01_globals.yaml`, but the framework also supports rudimentary injection of values from the command line. Let's try that, setting a new run name, increasing the number of epochs and changing the learning rate:

```
eirtrain \
--global_configs eir_tutorials/a_using_eir/01_basic_tutorial/conf/tutorial_01_globals.
↪yaml \
--input_configs eir_tutorials/a_using_eir/01_basic_tutorial/conf/tutorial_01_input.yaml \
--output_configs eir_tutorials/a_using_eir/01_basic_tutorial/conf/tutorial_01_outputs.
↪yaml \
--tutorial_01_globals.output_folder=eir_tutorials/tutorial_runs/a_using_eir/tutorial_01_
↪run_lr-0.002_epochs-20 \
--tutorial_01_globals.lr=0.002 \
--tutorial_01_globals.n_epochs=20
```

Note: The injected values are according to the configuration filenames.

Looking at the training curve from that run, we can see we did a bit better:



We also notice that there is a gap between the training and evaluation performances, indicating that the model is starting to overfit on the training data. There are a bunch of regularisation settings we could try, such as increasing dropout in the input, fusion and output modules. Check the *Configuration API* reference for a full overview.

C - Predicting on external samples

Predicting on samples with known labels

To predict on external samples, we run `eirpredict`. As we can see when running `eirpredict --help`, it looks quite similar to `eirtrain`:

```
usage: eirpredict [-h] [--global_configs [GLOBAL_CONFIGS ...]]
                  [--input_configs [INPUT_CONFIGS ...]]
                  [--fusion_configs [FUSION_CONFIGS ...]]
                  [--output_configs [OUTPUT_CONFIGS ...]] --model_path
                  MODEL_PATH [--evaluate] --output_folder OUTPUT_FOLDER
                  [--attribution_background_source {train,predict}]

options:
  -h, --help            show this help message and exit
  --global_configs [GLOBAL_CONFIGS ...]
                        Global .yaml configurations for the experiment.
```

(continues on next page)

(continued from previous page)

```
  --input_configs [INPUT_CONFIGS ...]
                        Input feature extraction .yaml configurations. Each
                        configuration represents one input.
  --fusion_configs [FUSION_CONFIGS ...]
                        Fusion .yaml configurations.
  --output_configs [OUTPUT_CONFIGS ...]
                        Output .yaml configurations.
  --model_path MODEL_PATH
                        Path to model to use for predictions.
  --evaluate
  --output_folder OUTPUT_FOLDER
                        Where to save prediction results.
  --attribution_background_source {train,predict}
                        For attribution analysis, whether to load backgrounds
                        from the data used for training or to use the current
                        data passed to the predict module.
```

Generally we do not change much of the configs when predicting, with the exception of the input configs (and then mainly setting the `input_source`, i.e. where to load our samples to predict/test on from) and perhaps the global config (e.g. we might not compute attributions during training, but compute them on our test set by activating `compute_attributions` in the global config when predicting). Specific to `eirpredict`, we have to choose a saved model (`--model_path`), whether we want to evaluate the performance on the test set (`--evaluate` this means that the respective labels must be present in the `--output_configs`) and where to save the prediction results (`--output_folder`).

For the sake of this tutorial, we use one of the saved models from our previous training run and use it for inference using `eirpredict` module. Here, we will simply use it to predict on the same data as before.

Warning: We are only predicting on the same data we trained on in this tutorial to show how to use the `eirpredict` module. Always take care in separating what data you use for training and to evaluate generalization performance of your models!

Run the commands below, making sure you add the correct path of a saved model to the `--model_path` argument.

To test, we can run the following command (note that you will have to add the path to your saved model for the `--model_path` parameter below).

```
eirpredict \
--global_configs eir_tutorials/a_using_eir/01_basic_tutorial/conf/tutorial_01_globals.
↪yaml \
--input_configs eir_tutorials/a_using_eir/01_basic_tutorial/conf/tutorial_01_input.yaml \
--output_configs eir_tutorials/a_using_eir/01_basic_tutorial/conf/tutorial_01_outputs.
↪yaml \
--model_path eir_tutorials/tutorial_runs/a_using_eir/tutorial_01_run/saved_models/
↪tutorial_01_run_model_600_perf-average=0.8764.pt \
--evaluate \
--output_folder eir_tutorials/tutorial_runs/a_using_eir/tutorial_01_run/test_predictions/
↪known_outputs
```

This will generate a file called `calculated_metrics.json` in the supplied `output_folder` as well as a folder for each output (in this case called `ancestry_output` containing the actual predictions and plots. Of course the metrics are quite nonsensical here, as we are predicting on the same data we trained on.

One of the files generated are the actual predictions, found in the `predictions.csv` file:

The `True Label Untransformed` column contains the actual labels, as they were in the raw data. The `True Label` column contains the labels after they have been numerically encoded / normalized in `EIR`. The other columns represent the raw network outputs for each of the classes.

Predicting on samples with unknown labels

Notice that when running the command above, we knew the labels of the samples we were predicting on. In practice, we are often predicting on samples we have no clue about the labels of. In this case, we can again use the `eirpredict` with slightly modified arguments:

```
eirpredict \
--global_configs eir_tutorials/a_using_eir/01_basic_tutorial/conf/tutorial_01_globals.
↪yaml \
--input_configs eir_tutorials/a_using_eir/01_basic_tutorial/conf/tutorial_01_input.yaml \
--output_configs eir_tutorials/a_using_eir/01_basic_tutorial/conf/tutorial_01_outputs_
↪unknown.yaml \
--model_path eir_tutorials/tutorial_runs/a_using_eir/tutorial_01_run/saved_models/
↪tutorial_01_run_model_600_perf-average=0.8764.pt \
--output_folder eir_tutorials/tutorial_runs/a_using_eir/tutorial_01_run/test_predictions/
↪unknown_outputs
```

We can notice a couple of changes here compared to the previous command:

1. We have removed the `--evaluate` flag, as we do not have the labels for the samples we are predicting on.
2. We have a different output configuation file, `tutorial_01_outputs_unknown.yaml`.
3. We have a different output folder, `tutorial_01_unknown`.

If we take a look at the `tutorial_01_outputs_unknown.yaml` file, we can see that it contains the following:

Listing 4: tutorial_01_outputs_unknown.yaml

```yaml
output_info:
  output_name: ancestry_output
  output_source: null
  output_type: tabular
output_type_info:
  target_cat_columns:
    - Origin
```

Notice that everything is the same as before, but for `output_source` we have `null` instead of the *.csv* label file we had before.

Taking a look at the produced `predictions.csv` file, we can see that we only have the actual predictions, and no true labels:

D - Applying to your own data (e.g. UK Biobank)

Thank you for reading this far! Hopefully this tutorial introduced you well enough to the framework so you can apply it to your own data. For that, you will have to process it first (see: plink pipelines). Then you will have to set the relevant paths for the inputs (e.g. `input_source`, `snp_file`) and outputs (e.g. `output_source`, `target_cat_columns` or `target_con_columns` if you have continuous targets).

Important: If you are interested in quickly training deep learning models for genomic prediction, the EIR-auto-GP project might be of use to you.

When moving to large scale data such as the UK Biobank, the configurations we used on the ancestry toy data in this tutorial will likely not be sufficient. For example, the learning rate is likely too high. For this, here are some baseline configurations that we have found to work well as a starting point:

Listing 5: globals.yaml

```yaml
output_folder: "FILL"
sample_interval: 500
checkpoint_interval: 500
batch_size: "FILL"
lr: 0.0002
lr_plateau_patience: 5
gradient_clipping: 1.0
valid_size: "FILL"
n_epochs: 50
dataloader_workers: "FILL"
device: "FILL"
early_stopping_buffer: 2000
early_stopping_patience: 10
mixing_alpha: 0.2
optimizer: "adabelief"
weighted_sampling_columns: # for categorical targets, remove if only doing regression
  - "all"
log_level: "debug"
```

Listing 6: input_genotype.yaml

```yaml
input_info:
  input_source: "FILL"
  input_name: "genotype"
  input_type: "omics"
input_type_info:
  mixing_subtype: "cutmix-block"
  na_augment_alpha: 1.0
  na_augment_beta: 2.0
  snp_file: "FILL" # can delete if not computing attributions
model_config:
  model_type: "genome-local-net"
  model_init_config:
    rb_do: 0.1
    channel_exp_base: 2
    kernel_width: 16
    first_kernel_expansion: -4
    l1: 0.0
    cutoff: 4096
```

Listing 7: input_tabular.yaml

```yaml
input_info:
  input_source: "FILL"
  input_name: "tabular_input"
  input_type: "tabular"
input_type_info:
  input_cat_columns:
    - "FILL"
  input_con_columns:
    - "FILL"
model_config:
  model_type: "tabular"
  model_init_config:
    fc_layer: true
```

Listing 8: fusion.yaml

```yaml
model_config:
  fc_do: 0.1
  fc_task_dim: 512
  layers:
    - 2
  rb_do: 0.1
  stochastic_depth_p: 0.1
model_type: "default"
```

Listing 9: output.yaml

```yaml
output_info:
  output_name: "FILL"
```

(continues on next page)

(continued from previous page)

```
  output_source: "FILL"
  output_type: "tabular"
output_type_info:
  target_con_columns:
    - "FILL"
  target_cat_columns:
    - "FILL"
model_config:
  model_type: "mlp_residual"
  model_init_config:
    rb_do: 0.2
    fc_do: 0.2
    fc_task_dim: 512
    layers:
      - 2
    stochastic_depth_p: 0.2
    final_layer_type: "linear"
```

E - Serving

In this final section, we demonstrate serving our trained model as a web service and interacting with it using HTTP requests.

Starting the Web Service

To serve the model, use the following command:

```
eirserve --model-path [MODEL_PATH]
```

Replace *[MODEL_PATH]* with the actual path to your trained model. This command initiates a web service that listens for incoming requests.

Here is an example of the command:

```
eirserve \
--model-path eir_tutorials/tutorial_runs/a_using_eir/tutorial_01_run/saved_models/
↪tutorial_01_run_model_600_perf-average=0.8764.pt
```

Sending Requests

With the server running, we can now send requests. The requests are prepared by loading numpy array data, converting it to base64 encoded strings, and then constructing a JSON payload.

Here's an example Python function demonstrating this process:

```
import numpy as np
import base64
import requests

def encode_numpy_array(file_path: str) -> str:
```

(continues on next page)

(continued from previous page)

```
    array = np.load(file_path)
    encoded = base64.b64encode(array.tobytes()).decode('utf-8')
    return encoded

def send_request(url: str, payload: dict):
    response = requests.post(url, json=payload)
    return response.json()

encoded_data = encode_numpy_array('path_to_your_numpy_array.npy')
response = send_request('http://localhost:8000/predict', {'genotype': encoded_data})
print(response)
```

Analyzing Responses

Here are some examples of responses from the server for a set of inputs:

Listing 10: predictions.json

```
[
    {
        "request": {
            "genotype": "eir_tutorials/a_using_eir/01_basic_tutorial/data/processed_
↪sample_data/arrays/A374.npy"
        },
        "response": {
            "result": {
                "ancestry_output": {
                    "Origin": {
                        "Asia": 0.010410779155790806,
                        "Eastern_Asia": 0.0011356589384377003,
                        "Europe": 0.854654848575592,
                        "Latin_America_and_the_Caribbean": 0.008827924728393555,
                        "Middle_East": 0.1237422451376915,
                        "Sub-Saharan_Africa": 0.00122847652528435
                    }
                }
            }
        }
    },
    {
        "request": {
            "genotype": "eir_tutorials/a_using_eir/01_basic_tutorial/data/processed_
↪sample_data/arrays/Ayodo_468C.npy"
        },
        "response": {
            "result": {
                "ancestry_output": {
                    "Origin": {
                        "Asia": 0.0017986423335969448,
                        "Eastern_Asia": 0.0030721763614565134,
                        "Europe": 0.0034481489565223455,
```

(continues on next page)

(continued from previous page)


```
                        "Latin_America_and_the_Caribbean": 0.026503251865506172,
                        "Middle_East": 0.1034306138753891,
                        "Sub-Saharan_Africa": 0.861747145652771
                    }
                }
            }
        }
    },
    {
        "request": {
            "genotype": "eir_tutorials/a_using_eir/01_basic_tutorial/data/processed_
↪sample_data/arrays/NOR146.npy"
        },
        "response": {
            "result": {
                "ancestry_output": {
                    "Origin": {
                        "Asia": 0.008015047758817673,
                        "Eastern_Asia": 0.0006639149505645037,
                        "Europe": 0.9414306282997131,
                        "Latin_America_and_the_Caribbean": 0.03938961401581764,
                        "Middle_East": 0.009529098868370056,
                        "Sub-Saharan_Africa": 0.0009716283529996872
                    }
                }
            }
        }
    }
]
```


2.1.2 02 – Tabular Tutorial: Nonlinear Poker Hands

A - Setup

In this tutorial, we will be training a model using only tabular data as input. The task is to predict poker hands from the suit an rank of cards. See here for more information about the dataset.

Note that this tutorial assumes that you are already familiar with the basic functionality of the framework (see *01 – Genotype Tutorial: Ancestry Prediction*).

To download the data for for this tutorial, use this link.

Having a quick look at the data, we can see it consists of 10 categorical inputs columns and 1 categorical output column (which has 10 classes).

```
$ head -n 3 poker_hands_data/poker_hands_train.csv

ID,S1,C1,S2,C2,S3,C3,S4,C4,S5,C5,CLASS
0,2,11,2,13,2,10,2,12,2,1,9
1,3,12,3,11,3,13,3,10,3,1,9
```

To start with, we can use the following configurations for the global, input, target and predictor parts respectively:

Listing 11: 02_poker_hands_globals.yaml

```
output_folder: eir_tutorials/tutorial_runs/a_using_eir/tutorial_02_run
manual_valid_ids_file: eir_tutorials/a_using_eir/02_tabular_tutorial/data/poker_hands_
↪data/pre_split_valid_ids.txt
n_saved_models: 1
checkpoint_interval: 1000
sample_interval: 1000
n_epochs: 50
```

Note: You might notice the perhaps new `manual_valid_ids_file` argument in the global configuration. This is because the data is quite imbalanced, so we provide a pre-computed validation set to ensure that all classes are present in both the training and validation set. Be aware that currently the framework does not handle having a mismatch in which classes are present in the training and validation sets.

Listing 12: 02_poker_hands_input.yaml

```
input_info:
  input_source: eir_tutorials/a_using_eir/02_tabular_tutorial/data/poker_hands_data/
↪poker_hands_train.csv
  input_name: poker_hands
  input_type: tabular

input_type_info:
  input_cat_columns:
          - S1
          - C1
          - S2
          - C2
          - S3
          - C3
          - S4
          - C4
          - S5
          - C5

model_config:
  model_type: tabular
```

Listing 13: 02_poker_hands_fusion.yaml

```
model_type: mlp-residual
model_config:
  rb_do: 0.20
  fc_do: 0.20
```

Listing 14: 02_poker_hands_output.yaml

```
output_info:
  output_source: eir_tutorials/a_using_eir/02_tabular_tutorial/data/poker_hands_data/
↪poker_hands_train.csv
```

(continues on next page)

(continued from previous page)

```
  output_name: poker_prediction
  output_type: tabular
output_type_info:
  target_cat_columns:
    - CLASS
```

So, after setting up, our folder structure should look something like this:

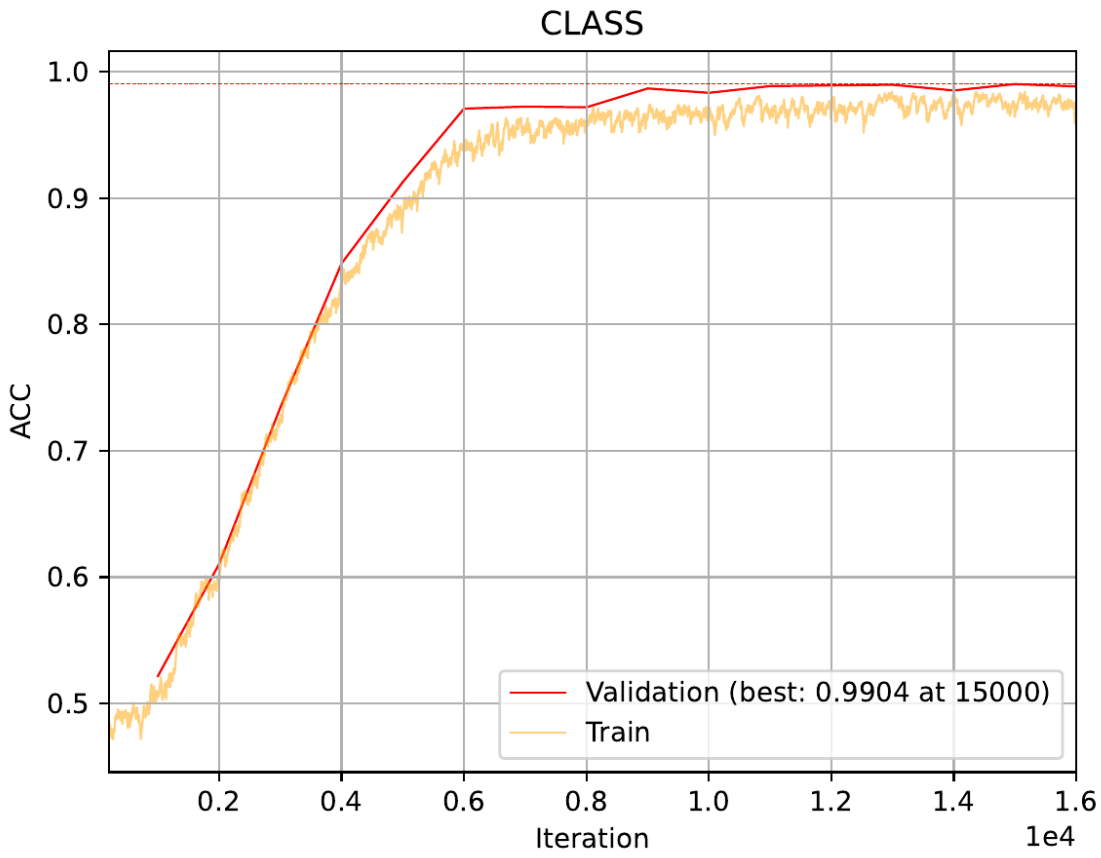
```
eir_tutorials/a_using_eir/02_tabular_tutorial/
├── conf
│   ├── 02_poker_hands_fusion.yaml
│   ├── 02_poker_hands_globals.yaml
│   ├── 02_poker_hands_input.yaml
│   └── 02_poker_hands_output.yaml
└── data
    └── poker_hands_data
        ├── poker_hands_test.csv
        ├── poker_hands_train.csv
        └── pre_split_valid_ids.txt
```

B - Training

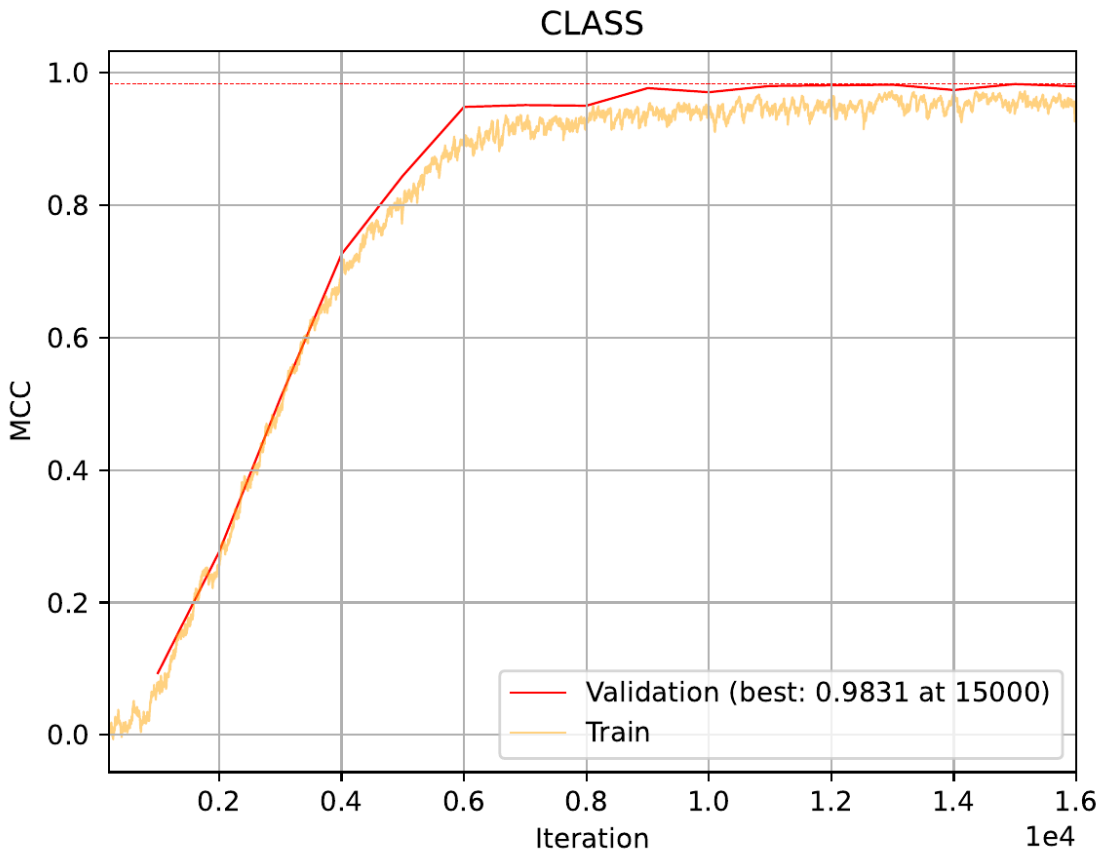
Now we are ready to train our first model! We can use the command below, which feeds the configs we defined above to the framework (fully running this should take around 10 minutes, so now is a good time to stretch your legs or grab a cup of coffee!):

```
eirtrain \
--global_configs eir_tutorials/a_using_eir/02_tabular_tutorial/conf/02_poker_hands_
↪globals.yaml \
--input_configs eir_tutorials/a_using_eir/02_tabular_tutorial/conf/02_poker_hands_input.
↪yaml \
--fusion_configs eir_tutorials/a_using_eir/02_tabular_tutorial/conf/02_poker_hands_
↪fusion.yaml \
--output_configs eir_tutorials/a_using_eir/02_tabular_tutorial/conf/02_poker_hands_
↪output.yaml
```

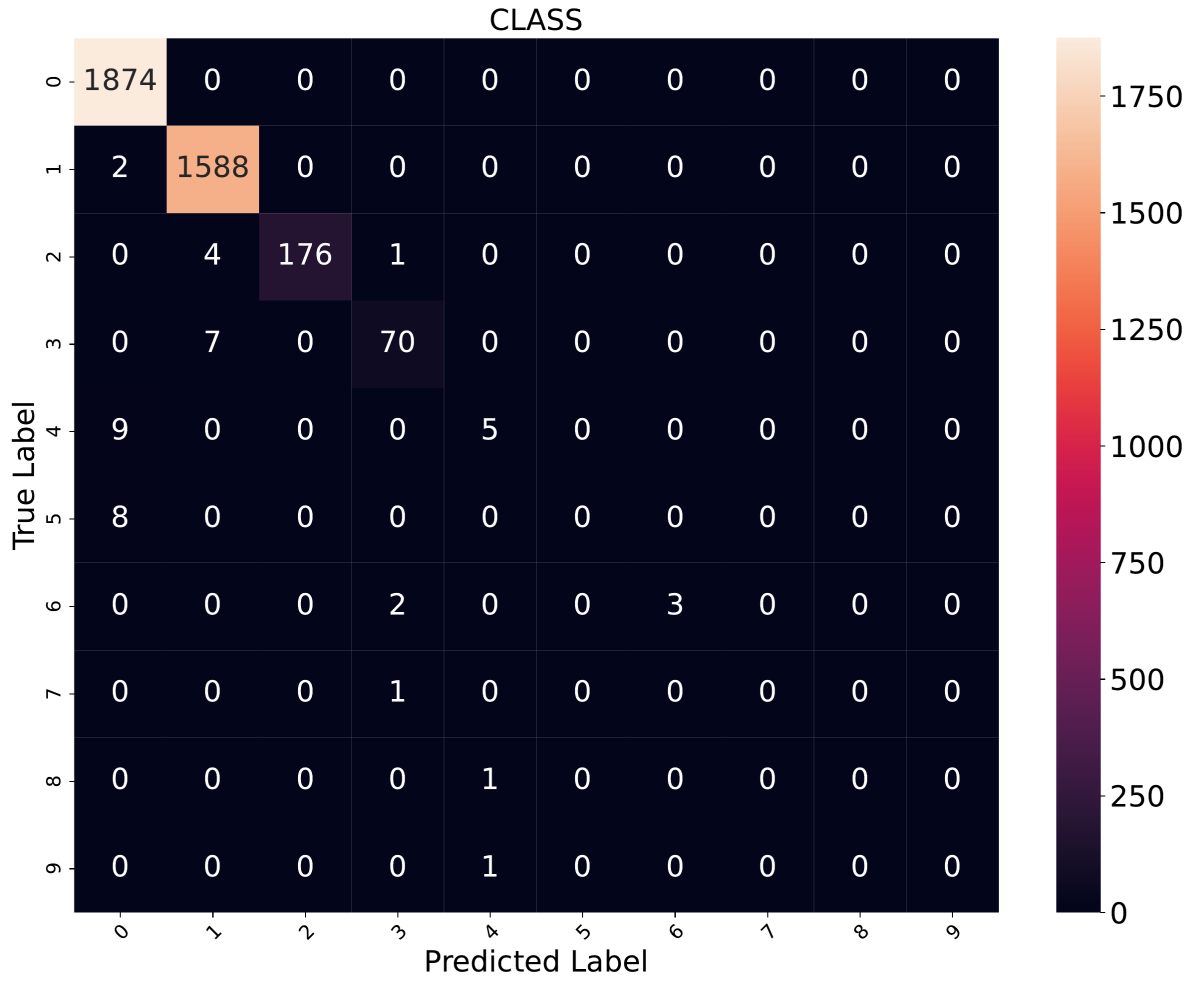
We can examine how our model did with respect to accuracy by checking the *training_curve_ACC.png* file:



However, we do know that the data is very imbalanced, so a better idea might be checking the MCC:



Both look fairly good, but how are we really doing? Let's check the confusion matrix for our predictions at iteration 15000:



So there it is – we are performing quite well for classes 0-3, but (perhaps as expected), we perform very poorly on the rare classes.

In any case, let's have a look at how well we do on the test set!

C - Predicting on test set

To test, we can run the following command (note that you will have to add the path to your saved model for the `--model_path` parameter below).

```
eirpredict \
--global_configs eir_tutorials/a_using_eir/02_tabular_tutorial/conf/02_poker_hands_
↪globals.yaml \
--input_configs eir_tutorials/a_using_eir/02_tabular_tutorial/conf/02_poker_hands_input_
↪test.yaml \
--fusion_configs eir_tutorials/a_using_eir/02_tabular_tutorial/conf/02_poker_hands_
↪fusion.yaml \
--output_configs eir_tutorials/a_using_eir/02_tabular_tutorial/conf/02_poker_hands_
↪output_test.yaml \
--model_path eir_tutorials/tutorial_runs/a_using_eir/tutorial_02_run/saved_models/
↪tutorial_02_run_model_15000_perf-average=0.8400.pt \
```

(continues on next page)

(continued from previous page)

```
--evaluate \
--output_folder eir_tutorials/tutorial_runs/a_using_eir/tutorial_02_run/
```

This will create the following extra files in the `eir_tutorials/tutorial_runs/a_using_eir/tutorial_02_run` directory

```
├── CLASS
│   ├── confusion_matrix.png
│   ├── mc_pr_curve.png
│   ├── mc_roc_curve.png
│   └── predictions.csv
├── calculated_metrics.json
```

The `calculated_metrics.json` file can be quite useful, as it contains the performance of our model on the test set.

Listing 15: calculated_metrics.json


```
{"poker_prediction": {"CLASS": {"poker_prediction_CLASS_mcc": 0.981885141539836, "poker_
↪prediction_CLASS_acc": 0.9897459897459897, "poker_prediction_CLASS_roc-auc-macro": 0.
↪9290761712622213, "poker_prediction_CLASS_ap-macro": 0.5669042208423734, "poker_
↪prediction_CLASS_loss": 0.04536255821585655}}, "average": {"average": {"loss-average":␣
↪0.04536255821585655, "perf-average": 0.8259551778814769}}}
```


This seems pretty good, but we don't really have any baseline to compare it to. Luckily, there is an great paper titled TabNet: Attentive Interpretable Tabular Learning, which is also using NNs on tabular data, and they even use the Poker Hand dataset as well!

Table 1: TabNet paper performances for Poker Hand induction dataset.

Model	Test accuracy (%)
DT	50.0
MLP	50.0
Deep neural DT	65.1
XGBoost	71.1
LightGBM	70.0
CatBoost	66.6
TabNet	99.2
Rule-based	100.0

So using our humble model before we saw an accuracy of 99.1%. Of course, since the dataset is highly imbalanced, it can be difficult to compare with the numbers in the table above. For example it can be that TabNet is performing very well on the rare classes, which will not have a large effect on the total test accuracy. However, our performance is perhaps a nice baseline, especially since TabNet is a much more complex model, and we did not do extensive hyper-parameter tuning!

E - Serving

In this final section, we demonstrate serving our trained model as a web service and interacting with it using HTTP requests.

Starting the Web Service

To serve the model, use the following command:

```
eirserve --model-path [MODEL_PATH]
```

Replace *[MODEL_PATH]* with the actual path to your trained model. This command initiates a web service that listens for incoming requests.

Here is an example of the command:

```
eirserve \
--model-path eir_tutorials/tutorial_runs/a_using_eir/tutorial_02_run/saved_models/
↪tutorial_02_run_model_15000_perf-average=0.8400.pt
```

Sending Requests

With the server running, we can now send requests. For tabular data, we send the payload directly as a Python dictionary.

Here's an example Python function demonstrating this process:

```
import requests

def send_request(url: str, payload: dict):
    response = requests.post(url, json=payload)
    return response.json()

payload = {
    "poker_hands": {
        "S1": "3", "C1": "12",
        "S2": "3", "C2": "2",
        "S3": "3", "C3": "11",
        "S4": "4", "C4": "5",
        "S5": "2", "C5": "5"
    }
}

response = send_request('http://localhost:8000/predict', payload)
print(response)
```

Additionally, you can send requests using *bash*:

```
curl -X 'POST' \\
  'http://localhost:8000/predict' \\
  -H 'accept: application/json' \\
  -H 'Content-Type: application/json' \\
  -d '{
```

(continues on next page)

(continued from previous page)

```
    "poker_hands": {
        "S1": "3", "C1": "12",
        "S2": "3", "C2": "2",
        "S3": "3", "C3": "11",
        "S4": "4", "C4": "5",
        "S5": "2", "C5": "5"
    }
}'
```

Analyzing Responses

After sending requests to the served model, the responses can be analyzed. These responses provide insights into the model's predictions based on the input data.

Listing 16: predictions.json


```
[
    {
        "request": {
            "poker_hands": {
                "S1": "3",
                "C1": "12",
                "S2": "3",
                "C2": "2",
                "S3": "3",
                "C3": "11",
                "S4": "4",
                "C4": "5",
                "S5": "2",
                "C5": "5"
            }
        },
        "response": {
            "result": {
                "poker_prediction": {
                    "CLASS": {
                        "0": 1.188167789223371e-05,
                        "1": 0.9977349042892456,
                        "2": 0.002234159503132105,
                        "3": 7.436228770529851e-06,
                        "4": 1.001353666651994e-07,
                        "5": 4.333995548222447e-06,
                        "6": 8.68369767204058e-08,
                        "7": 8.973422893632232e-08,
                        "8": 4.198012902634218e-06,
                        "9": 2.7543637770577334e-06
                    }
                }
            }
        }
    }
```


(continues on next page)

(continued from previous page)

```
]
```

If you made it this far, I want to thank you for reading. I hope this tutorial was useful / interesting to you!

2.1.3 03 – Sequence Tutorial: Movie Reviews and Peptides

In this tutorial, we will be training models using discrete sequences as inputs. Here, we will be doing two tasks. Firstly, we train a model to classify positive vs. negative sentiment in the IMDB reviews dataset. Secondly, we will train another model to detect anticancer properties in peptides using the anticancer peptides dataset.

Note that this tutorial assumes that you are already familiar with the basic functionality of the framework (see *01 – Genotype Tutorial: Ancestry Prediction*).

A - IMDB Reviews

A1 - IMDB Setup

For this first task, we will do a relatively classic NLP task, where we train a model to predict sentiment from IMDB reviews, see here for more information about the data. To download the data and configurations for this part of the tutorial, use this link.

Here we can see an example of one review from the dataset.

```
$ cat IMDB/IMDB_Reviews/3314_1.txt

Reading through all these positive reviews I find myself baffled.
How is it that so many enjoyed what I consider to be a woefully bad adaptation
of my second favourite Jane Austen novel? There are many problems with the film,
already mentioned in a few reviews; simply put it is a hammed-up, over-acted,
chintzy mess from opening credits to butchered ending.<br /><br />While many
characters are mis-cast and neither Ewan McGregor nor Toni Collette puts in a
performance that is worthy of them, the worst by far is Paltrow. \
I have very much enjoyed her performance in some roles, but here she is
abominable - she is self-conscious, nasal, slouching and entirely disconnected
from her characters and those around her. An extremely disappointing effort -
though even a perfect Emma could not have saved this film.
```

Whatever movie this review is from, it seems that the person certainly did not enjoy it! This is fairly obvious for us to see, now the question is if we train a model to do the same.

As in previous tutorials, we will start by defining our configurations.

Listing 17: 03a_imdb_globals.yaml

```
output_folder: eir_tutorials/tutorial_runs/a_using_eir/tutorial_03_imdb_run
valid_size: 0.10
n_saved_models: 1
checkpoint_interval: 500
sample_interval: 500
memory_dataset: true
n_epochs: 25
compute_attributions: true
```

(continues on next page)

(continued from previous page)

```
max_attributions_per_class: 512
attributions_every_sample_factor: 4
```

Note: You might notice that in the global configuration in this tutorial, we have a couple of new parameters going on. Namely the `compute_attributions`, `max_attributions_per_class` and `attributions_every_sample_factor`. These are settings related to computing attributions so we can interpret/explain how our inputs influence the model outputs. For more information, check out the *Configuration API* reference.

Listing 18: 03a_imdb_input.yaml

```
input_info:
  input_source: eir_tutorials/a_using_eir/03_sequence_tutorial/data/IMDB/IMDB_Reviews
  input_name: imdb_reviews
  input_type: sequence

input_type_info:
        sampling_strategy_if_longer: "uniform"
        max_length: 64
        split_on: " "
        min_freq: 10
        tokenizer: "basic_english"
        tokenizer_language: "en"

model_config:
        model_type: sequence-default
        embedding_dim: 32
        position: embed
        pool: avg
        model_init_config:
          num_heads: 2
          dropout: 0.2
```

Listing 19: 03a_imdb_output.yaml

```
output_info:
  output_source: eir_tutorials/a_using_eir/03_sequence_tutorial/data/IMDB/imdb_labels.csv
  output_name: imdb_output
  output_type: tabular

output_type_info:
  target_cat_columns:
    - Sentiment
```

Tip: There are a lot of new configuration options going on here, head over to the *Configuration API* reference for more details.

Now with the configurations set up, our folder structure should look like this:

Listing 20: Folder structure after setting up the configurations.

```
eir_tutorials/a_using_eir/03_sequence_tutorial/
├── a_IMDB
│   └── conf
│       ├── 03a_imdb_globals.yaml
│       ├── 03a_imdb_input.yaml
│       └── 03a_imdb_output.yaml
└── data
    └── IMDB
        ├── IMDB_Reviews
        ├── conf
        ├── imdb.vocab
        └── imdb_labels.csv
```

A2 - IMDB Training

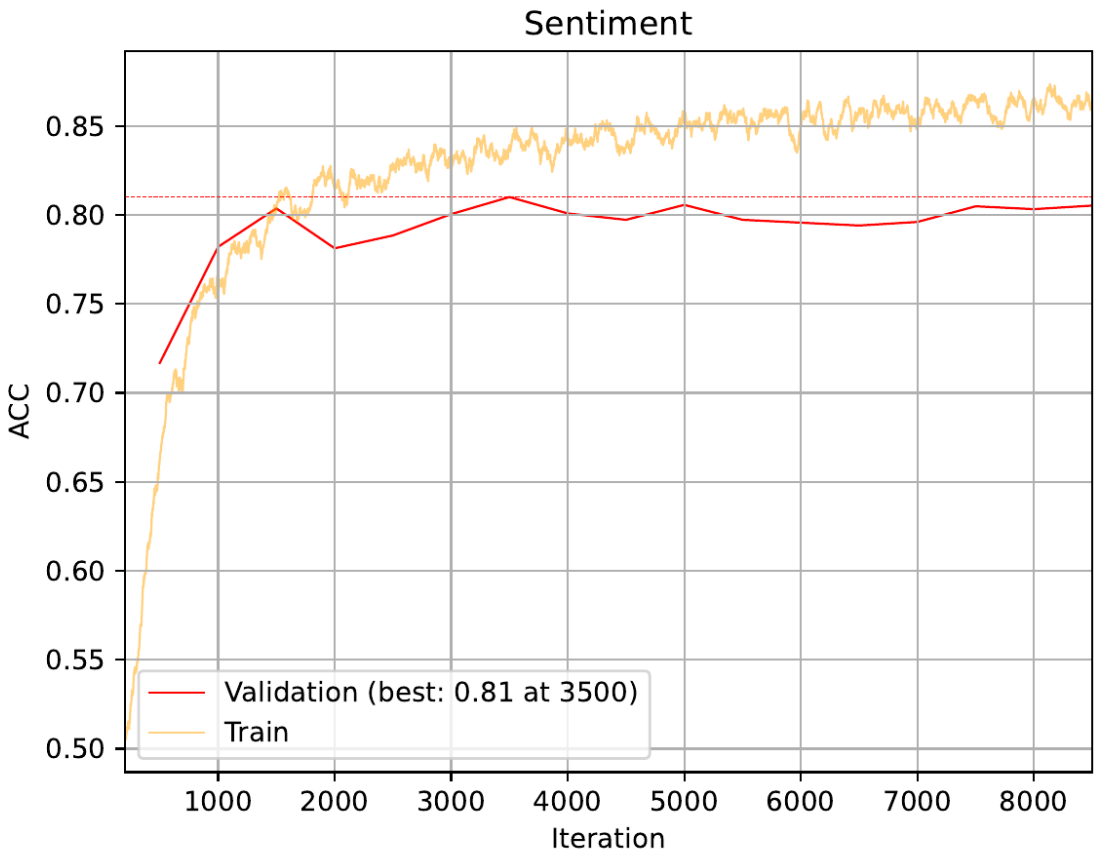
As before, we can train a model using `eirtrain`:

Listing 21: Training a model to predict sentiment from IMDB reviews.

```
eirtrain \
--global_configs eir_tutorials/a_using_eir/03_sequence_tutorial/a_IMDB/conf/03a_imdb_
↪globals.yaml \
--input_configs eir_tutorials/a_using_eir/03_sequence_tutorial/a_IMDB/conf/03a_imdb_
↪input.yaml \
--output_configs eir_tutorials/a_using_eir/03_sequence_tutorial/a_IMDB/conf/03a_imdb_
↪output.yaml
```

This took around 20 minutes to run on my laptop, so this is a good chance to take a nap or do something else for a while!

Looking at the accuracy, I got the following training/validation results:



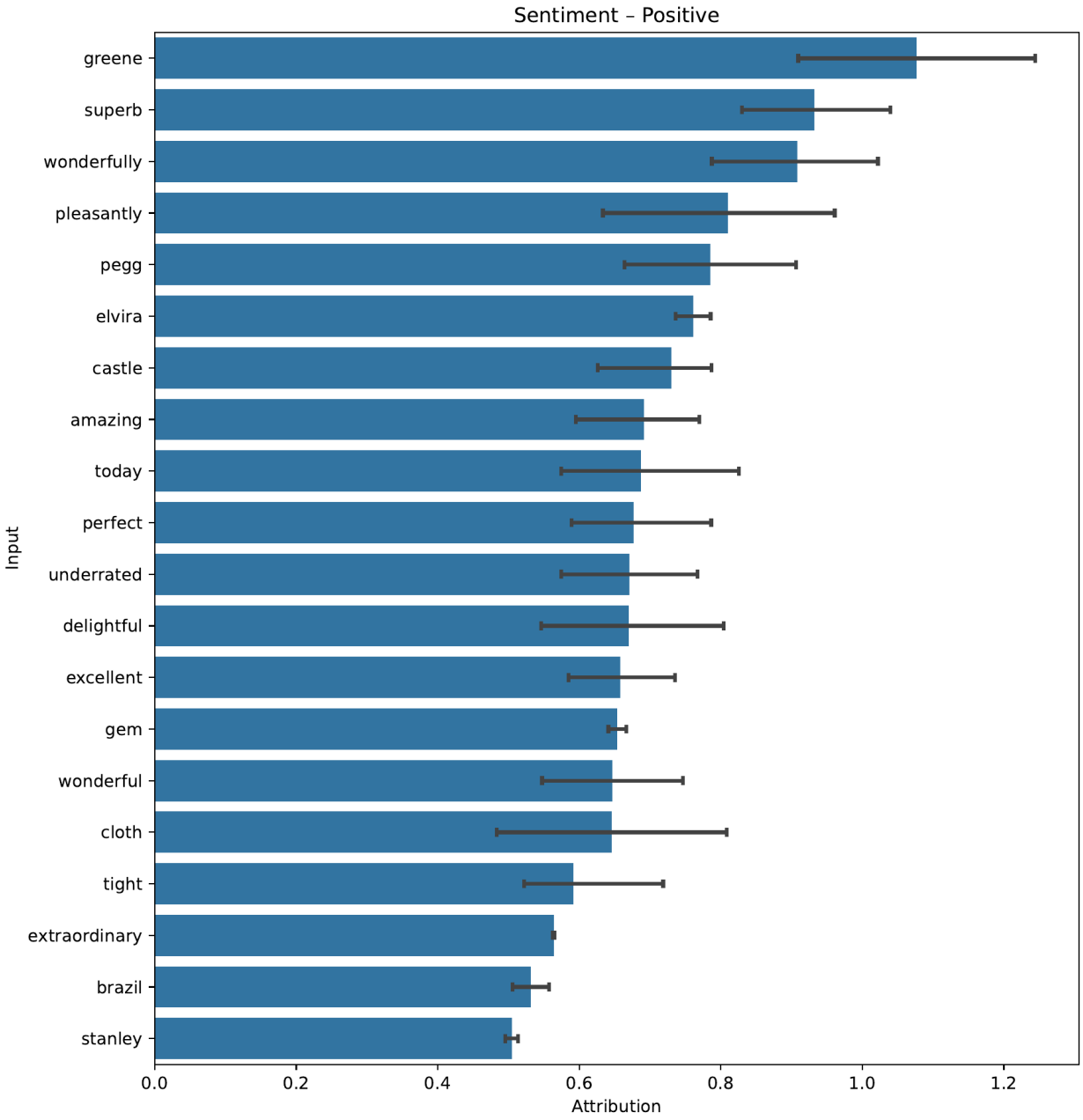
Perhaps not great, but not too bad either! Especially since we are using a relatively short sequence length.

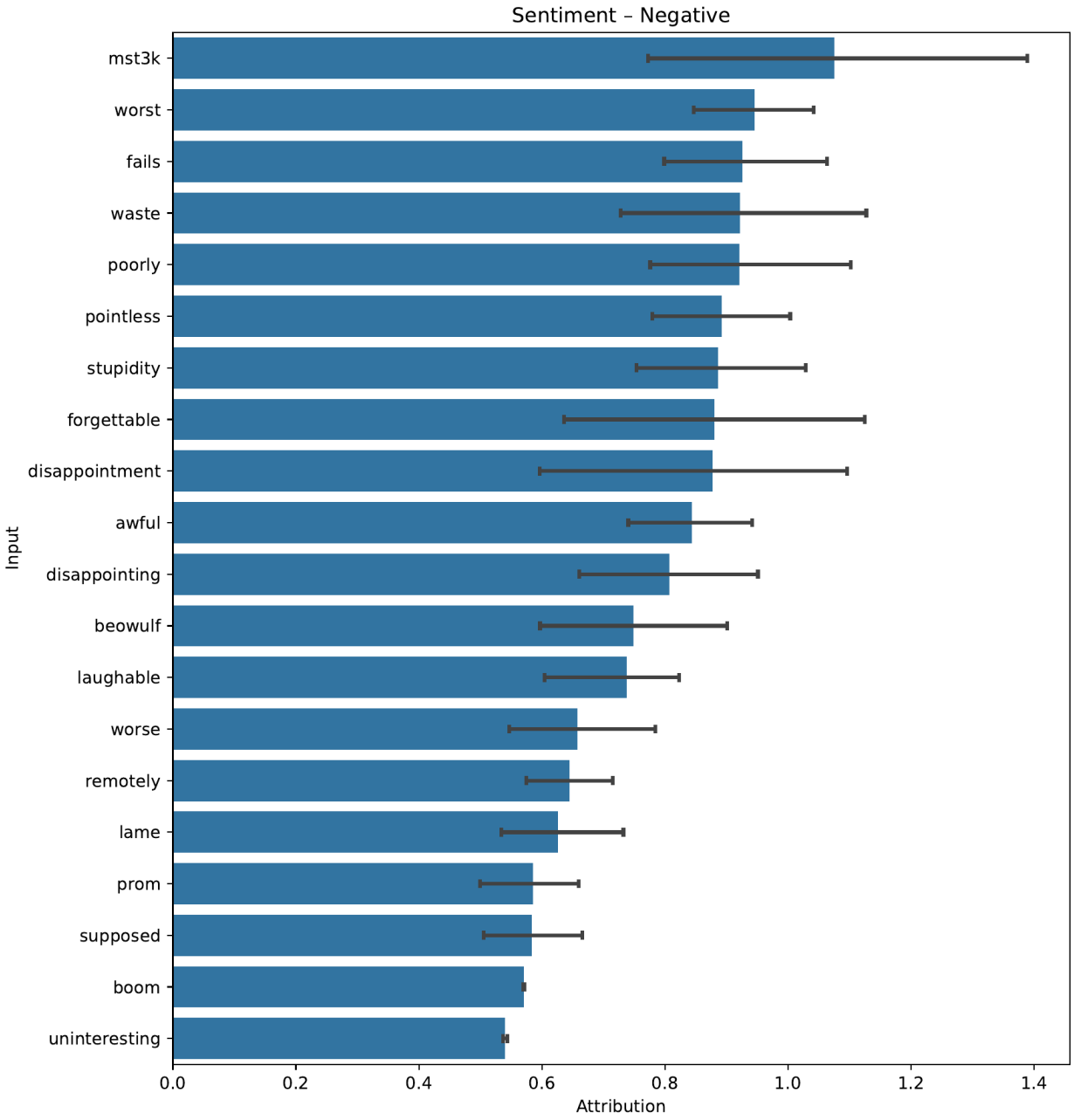
Note: Here we are using a transformer based neural network for the training, however do not underestimate the power of classical, more established methods. In fact, simpler, non neural-network based methods have attained better accuracy that what we see above! If you have some time to kill, try playing with the hyper parameters a bit to see how they affect the performance.

A3 - IMDB Interpretation

Now remember those new flags we used in the global configuration, `compute_attributions` and friends? Setting those will instruct the framework to compute and analyze how the inputs influence the model towards a certain output. In this case, the attributions can be found in the `imdb_sentiment/results/Sentiment/samples/<every_2000_iterations>/attributions` folders. Behind the scenes, the framework uses integrated gradients, implemented in the fantastic the Captum library, to compute the attributions.

Firstly, let's have a look at the words that had the biggest influence towards a Positive and Negative sentiment.


Sentiment – Positive
greene
superb
wonderfully
pleasantly
pegg
elvira
castle
amazing
today
perfect
underrated
delightful
excellent
gem
wonderful
cloth
tight
extraordinary
brazil
stanley
Input
0.0
0.2
0.4
0.6
0.8
1.0
1.2
Attribution



Note: Which tokens are included in this plot and how they are sorted is based both on the average and 95% confidence interval of the attribution. The raw values are also stored, in case you want to do your own analysis. The CIs represent the 95% confidence interval after 1,000 bootstrap samples.

So fortunately, it seems indeed that our model learned some relevant things! When training on sequences, the framework will also by default save attributions towards the relevant label for 10 single samples, here is one such example, where we look at the attributions towards a positive sentiment.

That concludes the NLP specific part of this tutorial, next we will apply the same approach but for biological data!

B - Anticancer Peptides

B1 - Anticancer Peptides Setup

Modelling on language like we did above is both fun and relatable, but now we try something a bit more niche. For this second part of the tutorial, we will use the framework to predict anti breast cancer properties of peptides (a peptide is basically a short protein sequence). See here for more information about the dataset. To download the data and configurations for this part of the tutorial, use this link.

Again, let's take a quick look at one sample we are going to be modelling on:

Here we can see an example of one review from the dataset.

```
$ cat Anticancer_Peptides/breast_cancer_train/1.txt

AAWKWAWAKKWAKAKKWAKAA
```

So immediately we can see that this is fairly different from our movie reviews, let's see how it goes with the modelling part. As always, we start with the configurations. You might notice a new option in the global configuration, `weighted_sampling_columns`. This setting controls which target column to use for weighted sampling, and the special keyword `all` will take an average across all target columns. In this case we have only one ("class"), so it just accounts for that one. This can be useful for this dataset as it is quite imbalanced w.r.t. target labels, as you will see momentarily.

Listing 22: 03b_peptides_globals.yaml

```
output_folder: eir_tutorials/tutorial_runs/a_using_eir/tutorial_03_anti_breast_cancer_
↪peptides_run
valid_size: 0.25
n_saved_models: 1
checkpoint_interval: 200
sample_interval: 200
n_epochs: 500
memory_dataset: True
batch_size: 32
early_stopping_buffer: 2000
compute_attributions: True
attributions_every_sample_factor: 3
max_attributions_per_class: 512
weighted_sampling_columns:
  - all
```

Note: You might notice that we use a large validation set here. This a similar situation as in *02 – Tabular Tutorial: Nonlinear Poker Hands*, where we used a manual validation set to ensure that we have all classes present in the validation set. Here, we take the lazier approach and just make the validation set larger. Currently the framework does not handle having a mismatch in which classes are present in the training and validation sets.

Notice that the input configuration is slightly different. For example, as we are not dealing with natural language, we do not split on whitespace anymore, but rather on "".

Listing 23: 03b_peptides_input.yaml

```yaml
input_info:
  input_source: eir_tutorials/a_using_eir/03_sequence_tutorial/data/Anticancer_Peptides/
↪breast_cancer_train
  input_name: peptide_sequences
  input_type: sequence

input_type_info:
        max_length: "max"
        split_on: ""
        min_freq: 1

model_config:
        model_type: sequence-default
        position: embed
        embedding_dim: 32
        pool: avg
        model_init_config:
          num_heads: 8
          dropout: 0.2

interpretation_config:
  num_samples_to_interpret: 30
  interpretation_sampling_strategy: random_sample
```

Listing 24: 03b_peptides_output.yaml

```yaml
output_info:
  output_source: eir_tutorials/a_using_eir/03_sequence_tutorial/data/Anticancer_Peptides/
↪breast_cancer_labels.csv
  output_name: peptides_output
  output_type: tabular

output_type_info:
  target_cat_columns:
    - class
```

B1 - Anticancer Peptides Training

For the peptide data, the folder structure should look something like this:

```
eir_tutorials/a_using_eir/03_sequence_tutorial/
├── b_Anticancer_peptides
│   └── conf
│       ├── 03b_peptides_globals.yaml
│       ├── 03b_peptides_input.yaml
│       └── 03b_peptides_output.yaml
└── data
    └── Anticancer_Peptides
        ├── breast_cancer_labels.csv
```

(continues on next page)

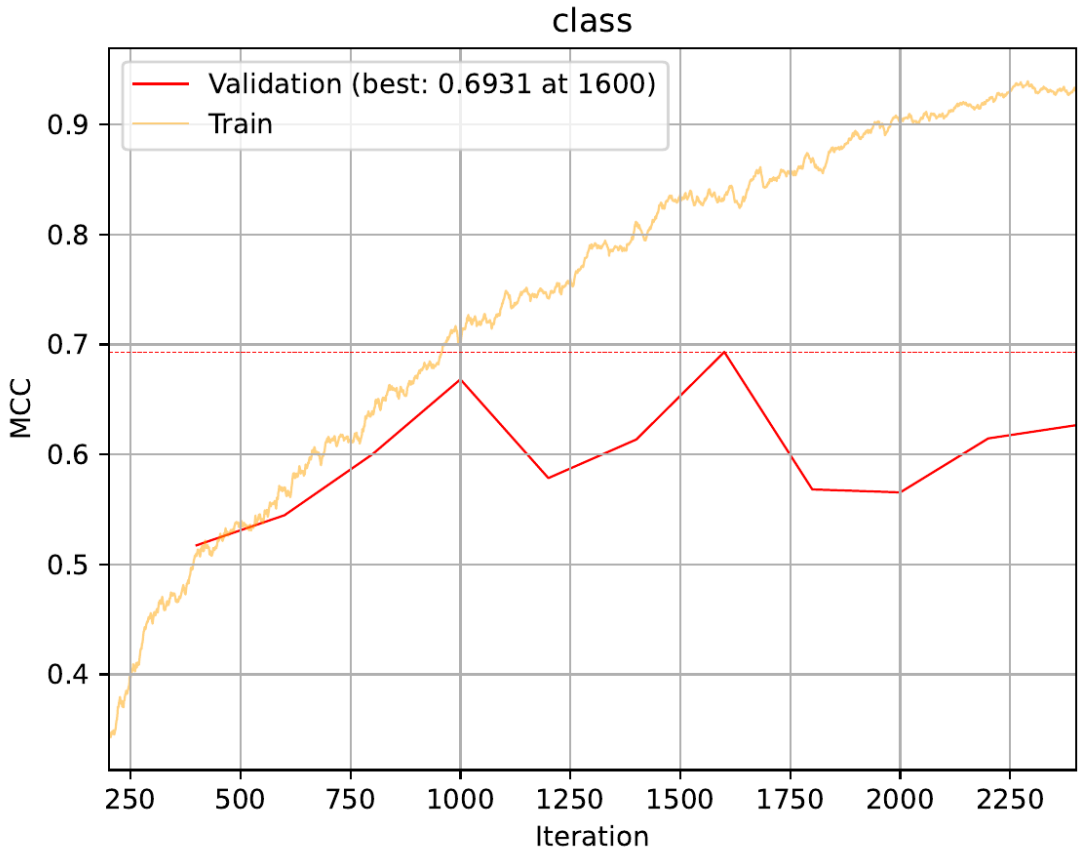
(continued from previous page)

```
        └── breast_cancer_train
```

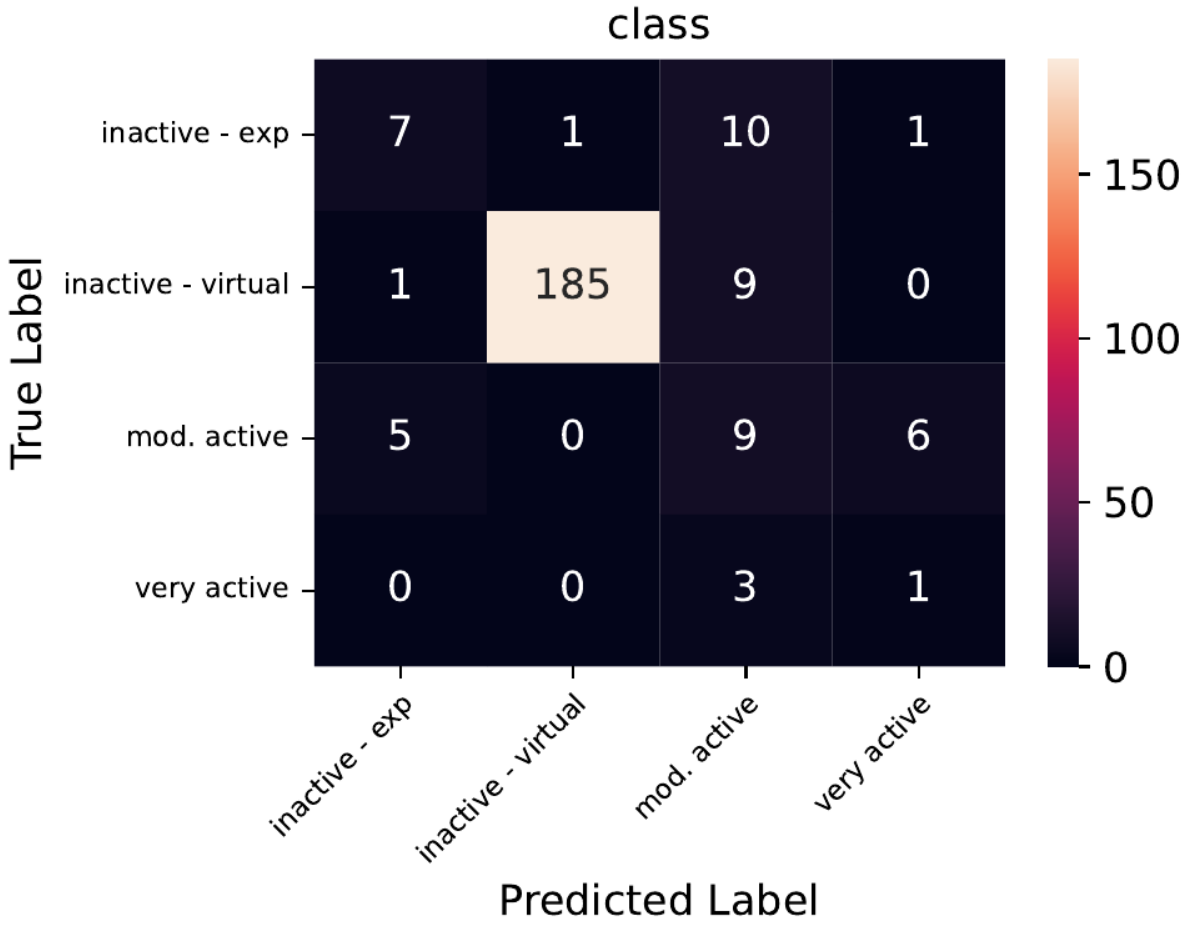
As before, we run:

```
eirtrain \
--global_configs eir_tutorials/a_using_eir/03_sequence_tutorial/b_Anticancer_peptides/
↪conf/03b_peptides_globals.yaml \
--input_configs eir_tutorials/a_using_eir/03_sequence_tutorial/b_Anticancer_peptides/
↪conf/03b_peptides_input.yaml \
--output_configs eir_tutorials/a_using_eir/03_sequence_tutorial/b_Anticancer_peptides/
↪conf/03b_peptides_output.yaml
```

As the data is imbalanced, we will look at the MCC training curve:

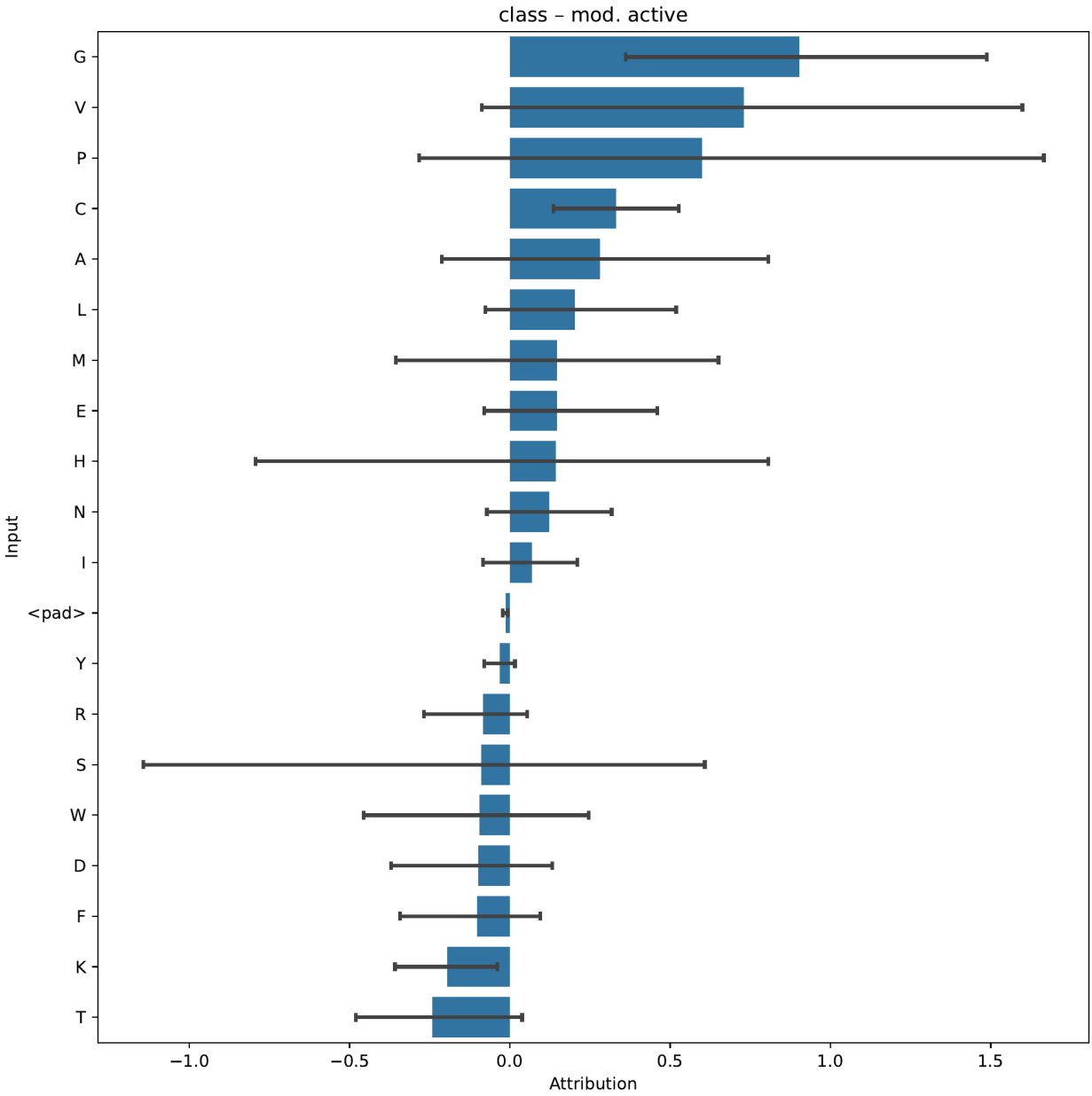


Checking the confusion matrix at iteration 2000, we see:



Looking at the training curve, we see that we are definitely overfitting quite a bit! We could probably squeeze out a better performance by playing with the hyperparameters a bit, but for now we will keep going!

As before, let's have a look at the attributions. In this case we will check attributions towards the moderately active class:



In this case, it seems that there is a high degree of uncertainty in the attributions, as the confidence intervals are quite large. This is likely due to the fact that the dataset is quite imbalanced, and there are few samples of moderately active peptides in the validation set.

Looking at an example of single moderately active sample and how its inputs influence the model towards a prediction of the moderately active class, we see:

Warning: Remember that this does not necessarily tell us anything about actual biological causality!

E - Serving

In this final section, we demonstrate serving our trained model as a web service and interacting with it using HTTP requests.

Starting the Web Service

To serve the model, use the following command:

```
eirserve --model-path [MODEL_PATH]
```

Replace *[MODEL_PATH]* with the actual path to your trained model. This command initiates a web service that listens for incoming requests.

Here is an example of the command:

```
eirserve \
--model-path eir_tutorials/tutorial_runs/a_using_eir/tutorial_03_imdb_run/saved_models/
↪tutorial_03_imdb_run_model_3500_perf-average=0.7969.pt
```

Sending Requests

With the server running, we can now send requests. For sequence data like IMDb reviews, we send the payload as a simple JSON object.

Here's an example Python function demonstrating this process:

```
import requests

def send_request(url: str, payload: dict):
    response = requests.post(url, json=payload)
    return response.json()

payload = {
    "imdb_reviews": "This movie was great! I loved it!"
}

response = send_request('http://localhost:8000/predict', payload)
print(response)
```

Additionally, you can send requests using *bash*:

```
curl -X 'POST' \\
  'http://localhost:8000/predict' \\
  -H 'accept: application/json' \\
  -H 'Content-Type: application/json' \\
  -d '{
      "imdb_reviews": "This movie was great! I loved it!"
  }'
```

Analyzing Responses

After sending requests to the served model, the responses can be analyzed. These responses provide insights into the model's predictions based on the input data.

Listing 25: predictions.json


```
[
    {
        "request": {
            "imdb_reviews": "This move was great! I loved it!"
        },
        "response": {
            "result": {
                "imdb_output": {
                    "Sentiment": {
                        "Negative": 0.10254316031932831,
                        "Positive": 0.8974568247795105
                    }
                }
            }
        }
    },
    {
        "request": {
            "imdb_reviews": "This move was terrible! I hated it!"
        },
        "response": {
            "result": {
                "imdb_output": {
                    "Sentiment": {
                        "Negative": 0.862532913684845,
                        "Positive": 0.13746710121631622
                    }
                }
            }
        }
    },
    {
        "request": {
            "imdb_reviews": "You'll have to have your wits about you and your brain␣
↪fully switched on watching Oppenheimer as it could easily get away from a nonattentive␣
↪viewer. This is intelligent filmmaking which shows it's audience great respect. It␣
↪fires dialogue packed with information at a relentless pace and jumps to very␣
↪different times in Oppenheimer's life continuously through it's 3 hour runtime. There␣
↪are visual clues to guide the viewer through these times but again you'll have to get␣
↪to grips with these quite quickly. This relentlessness helps to express the urgency␣
↪with which the US attacked it's chase for the atomic bomb before Germany could do the␣
↪same. An absolute career best performance from (the consistenly brilliant) Cillian␣
↪Murphy anchors the film. "
        },
        "response": {
            "result": {
```


(continues on next page)

(continued from previous page)

```
                "imdb_output": {
                    "Sentiment": {
                        "Negative": 0.017507053911685944,
                        "Positive": 0.9824929237365723
                    }
                }
            }
        }
    }
]
```

This concludes the sequence tutorial, thank you for making it this far. I hope you enjoyed it and it was useful to you. Feel free to try this out on your own data, I would love to hear about it!

2.1.4 04 – Established Architectures and Pretrained Models

In this tutorial, we will be seeing, how we can use local transformers, state-of-the-art, NLP architectures, and pretrained NLP models with `EIR` in order to predict sentiment from text. We will be using the IMDB reviews dataset, see here for more information about the data. To download the data and configurations for this part of the tutorial, use this link.

Note that this tutorial assumes that you are already familiar with the basic functionality of the framework (see *01 – Genotype Tutorial: Ancestry Prediction*). If you have not already, it can also be useful to go over the sequence tutorial (see *03 – Sequence Tutorial: Movie Reviews and Peptides*).

A - Baseline

After downloading the data, the folder structure should look something like this (note that at this point, the `yaml` configuration files are probably not present, but we will make them during this tutorial, alternatively you can download them from the project repository):

```
eir_tutorials/a_using_eir/04_pretrained_sequence_tutorial/
├── conf
│   ├── 04_imdb_globals.yaml
│   ├── 04_imdb_input.yaml
│   ├── 04_imdb_input_longformer.yaml
│   ├── 04_imdb_input_tiny-bert.yaml
│   ├── 04_imdb_input_windowed.yaml
│   └── 04_imdb_output.yaml
└── data
    └── IMDB
        ├── IMDB_Reviews
        ├── conf
        ├── imdb.vocab
        └── imdb_labels.csv
```

First we will use the built-in transformer model in `EIR`, just to establish a baseline.

As always, configurations first!

Listing 26: 04_imdb_globals.yaml

```yaml
output_folder: eir_tutorials/tutorial_runs/a_using_eir/tutorial_04_imdb_run
valid_size: 0.10
n_saved_models: 1
checkpoint_interval: 500
sample_interval: 500
early_stopping_patience: 5
memory_dataset: true
n_epochs: 25
mixing_alpha: 0.2
device: "mps"
dataloader_workers: 0
```

Note: Training these sequence models can take quite some time if one is using a laptop. If possible, try using a system with a GPU available! If not, set the device setting to 'cpu'.

Note: You might notice that we have a new configuration in our global config, `mixing_alpha`. The parameter controls the level of Mixup, a really cool data augmentation which is included in the framework, and is automatically applied to all input modalities (genotype, tabular, sequence, images, binary data) when set in the global configuration.

Listing 27: 04_imdb_input.yaml

```yaml
input_info:
  input_source: eir_tutorials/a_using_eir/04_pretrained_sequence_tutorial/data/IMDB/IMDB_
↪Reviews
  input_name: imdb_reviews
  input_type: sequence

input_type_info:
        sampling_strategy_if_longer: "uniform"
        max_length: 128
        split_on: " "
        min_freq: 10
        tokenizer: "basic_english"
        tokenizer_language: "en"

model_config:
        model_type: sequence-default
        embedding_dim: 32
        position: embed
        pool: avg
        model_init_config:
          num_heads: 2
          dropout: 0.2
```

Listing 28: 04_imdb_output.yaml

```yaml
output_info:
```

(continues on next page)

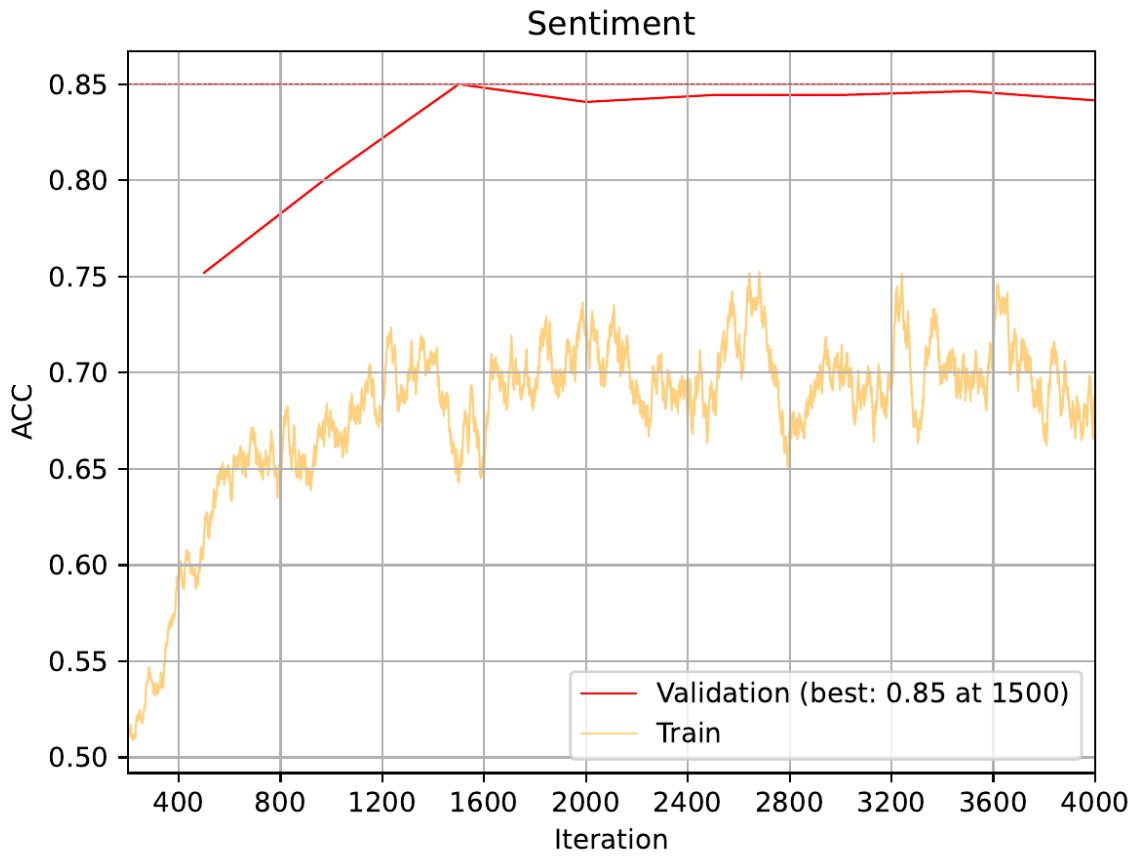
(continued from previous page)

```
  output_source: eir_tutorials/a_using_eir/03_sequence_tutorial/data/IMDB/imdb_labels.csv
  output_name: imdb_output
  output_type: tabular

output_type_info:
  target_cat_columns:
    - Sentiment
```

As before, we do our training with the following command:

```
eirtrain \
--global_configs eir_tutorials/a_using_eir/04_pretrained_sequence_tutorial/conf/04_imdb_
↪globals.yaml \
--input_configs eir_tutorials/a_using_eir/04_pretrained_sequence_tutorial/conf/04_imdb_
↪input.yaml \
--output_configs eir_tutorials/a_using_eir/04_pretrained_sequence_tutorial/conf/04_imdb_
↪output.yaml
```

Checking the accuracy, we see:



A little better than what we saw in the *03 – Sequence Tutorial: Movie Reviews and Peptides*, which makes sense as here we are using longer sequences and more data augmentation. In any case, now we have a nice little baseline to compare to!

B - Local Transformer

Transformer models are notorious for being quite expensive to train computationally, both when it comes to memory and raw compute. The main culprit is the quadratic increase w.r.t. input length. One relatively straightforward way to get around this is not looking at the full sequence at once, but rather in parts (kind of like a convolution). This functionality is included by default and can be controlled with the `window_size` parameter of the `input_type_info` field when training sequence models.

Now, let's try training one such model, using a window size of 64 and increasing the maximum sequence length to 512:

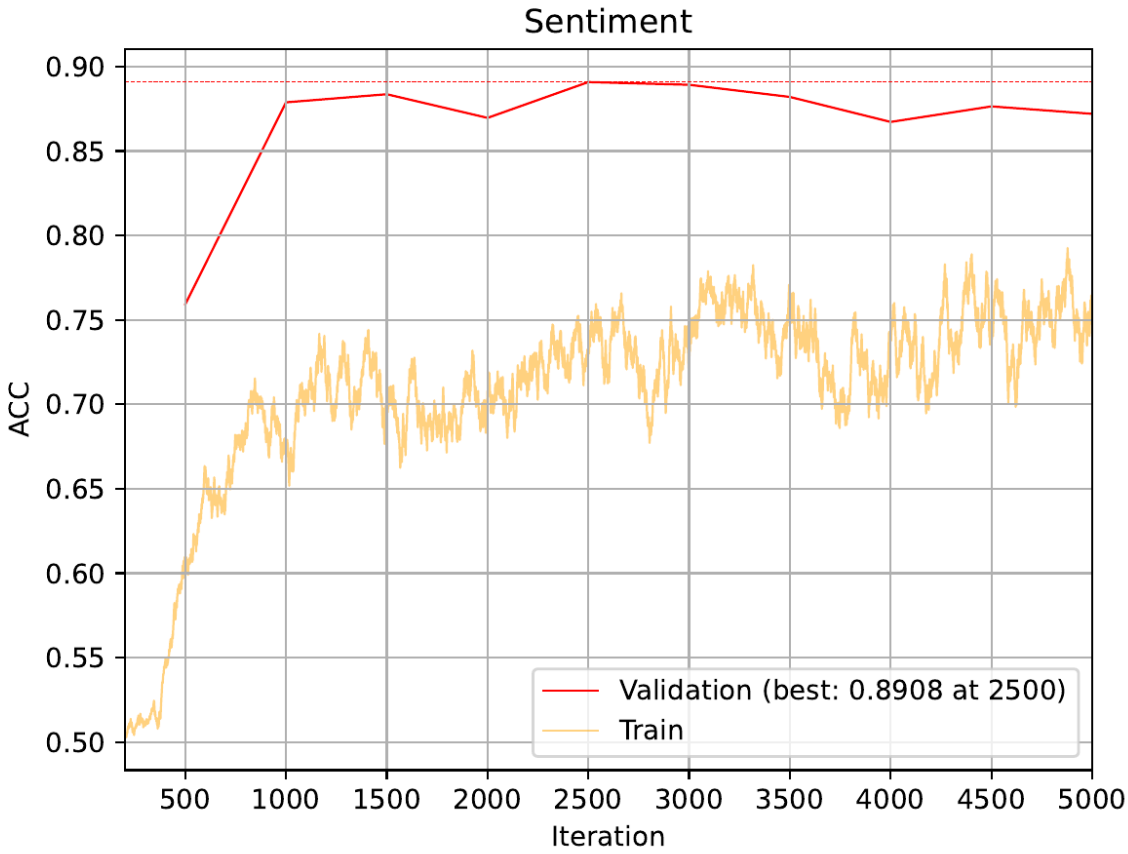
Listing 29: 04_imdb_input_windowed.yaml

```yaml
input_info:
  input_source: eir_tutorials/a_using_eir/04_pretrained_sequence_tutorial/data/IMDB/IMDB_
↪Reviews
  input_name: imdb_reviews_windowed
  input_type: sequence

input_type_info:
        sampling_strategy_if_longer: "uniform"
        max_length: 512
        split_on: " "
        min_freq: 10
        tokenizer: "basic_english"
        tokenizer_language: "en"

model_config:
  model_type: sequence-default
  window_size: 64
  position: embed
  pool: avg
  embedding_dim: 32
  model_init_config:
    num_heads: 2
    dropout: 0.2
```

To train, we just swap out the input configuration from the command above:

```
eirtrain \
--global_configs eir_tutorials/a_using_eir/04_pretrained_sequence_tutorial/conf/04_imdb_
↪globals.yaml \
--input_configs eir_tutorials/a_using_eir/04_pretrained_sequence_tutorial/conf/04_imdb_
↪input_windowed.yaml \
--output_configs eir_tutorials/a_using_eir/04_pretrained_sequence_tutorial/conf/04_imdb_
↪output.yaml \
--04_imdb_globals.output_folder=eir_tutorials/tutorial_runs/a_using_eir/tutorial_04_imdb_
↪run_local
```

Training this model gave the following training curve:



Indeed, increasing the sequence length does seem to help, and using a window size of 64 seems to work fairly well.

C - Established architecture: Longformer

Now, the windowed approach above is perhaps a quick win to tackle the scaling problems of transformers when it comes to input length. In fact, this is such a notorious problem that people have done a lot of work in finding cool architectures and methods to get around it. By taking advantage of the excellent work Hugging Face has done, we can use these established architectures within `EIR` (big thanks to them by the way!). The architecture we will be using is called Longformer, and as mentioned it tries to approximate full self-attention in order to scale linearly w.r.t input size.

Tip: Hugging Face has implemented a bunch of other pretrained models and architectures, check this link for an exhaustive list.

To use the Longformer model, we use the following configuration, notice that in the model configuration we are now passing in flags *specifically* to the LongFormer model:

Listing 30: 04_imdb_input_longformer.yaml

```
input_info:
  input_source: eir_tutorials/a_using_eir/04_pretrained_sequence_tutorial/data/IMDB/IMDB_
↪Reviews
  input_name: imdb_reviews_longformer
```

(continues on next page)

(continued from previous page)

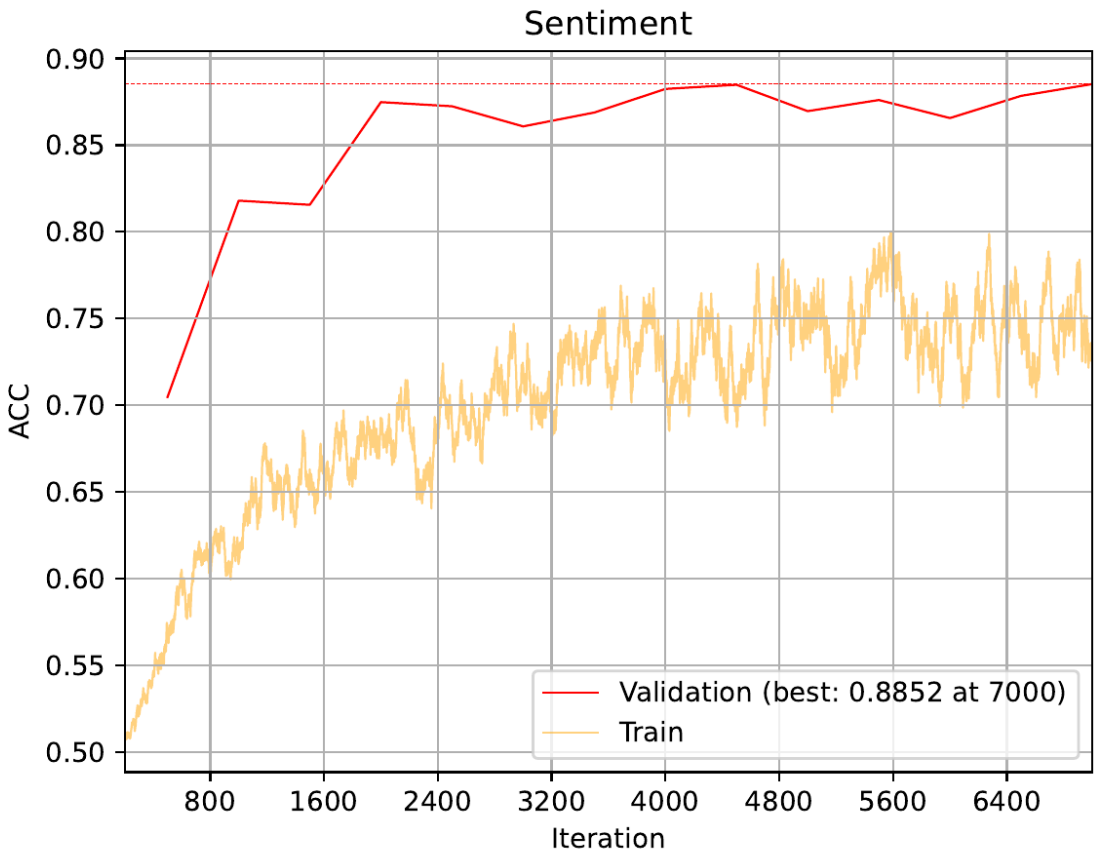
```
  input_type: sequence

input_type_info:
        sampling_strategy_if_longer: "uniform"
        max_length: 512
        split_on: " "
        min_freq: 10
        tokenizer: "basic_english"
        tokenizer_language: "en"

model_config:
      model_type: longformer
      pretrained_model: false
      position: embed
      pool: avg
      model_init_config:
            num_hidden_layers: 2
            hidden_size: 32
            num_attention_heads: 2
            intermediate_size: 32
            attention_window: 64
            max_position_embeddings: 1024
```

Note: The established architectures can have a bunch of different configurations available. Head over to the Hugging Face docs to see which flags they accept and what they do. For example, the LongFormer docs can be found here.

We train with the following command:

```
eirtrain \
--global_configs eir_tutorials/a_using_eir/04_pretrained_sequence_tutorial/conf/04_imdb_
↪globals.yaml \
--input_configs eir_tutorials/a_using_eir/04_pretrained_sequence_tutorial/conf/04_imdb_
↪input_longformer.yaml \
--output_configs eir_tutorials/a_using_eir/04_pretrained_sequence_tutorial/conf/04_imdb_
↪output.yaml \
--04_imdb_globals.output_folder=eir_tutorials/tutorial_runs/a_using_eir/tutorial_04_imdb_
↪run_longformer
```

And get the following training curve:



Indeed, we see an improvement on the validation set when using the the Longformer model compared to the first run. There does not seem to be a big difference compared to our local transformer run, Of course, we would have to evaluate on a test set to get the final performance, but this is looking pretty good!

D - Pretrained Model: Tiny BERT

Now, we have seen how we can use cool architectures to train our models. However, we can take this one step further and use a pretrained model as well, taking advantage of the fact that they have already been trained on a bunch of data.

In this case, we will use a little BERT model called Tiny BERT. The approach is almost the same as we saw above with the Longformer, here is the configuration:

Listing 31: 04_imdb_input_tiny-bert.yaml

```
input_info:
  input_source: eir_tutorials/a_using_eir/04_pretrained_sequence_tutorial/data/IMDB/IMDB_
↪Reviews
  input_name: imdb_reviews_tiny_bert
  input_type: sequence

input_type_info:
  sampling_strategy_if_longer: "uniform"
  max_length: 512
```

(continues on next page)

(continued from previous page)

```
  split_on: " "
  min_freq: 10

model_config:
    model_type: "prajjwal1/bert-tiny"
    pretrained_model: true
    freeze_pretrained_model: false
    position: embed
    pool: avg
```

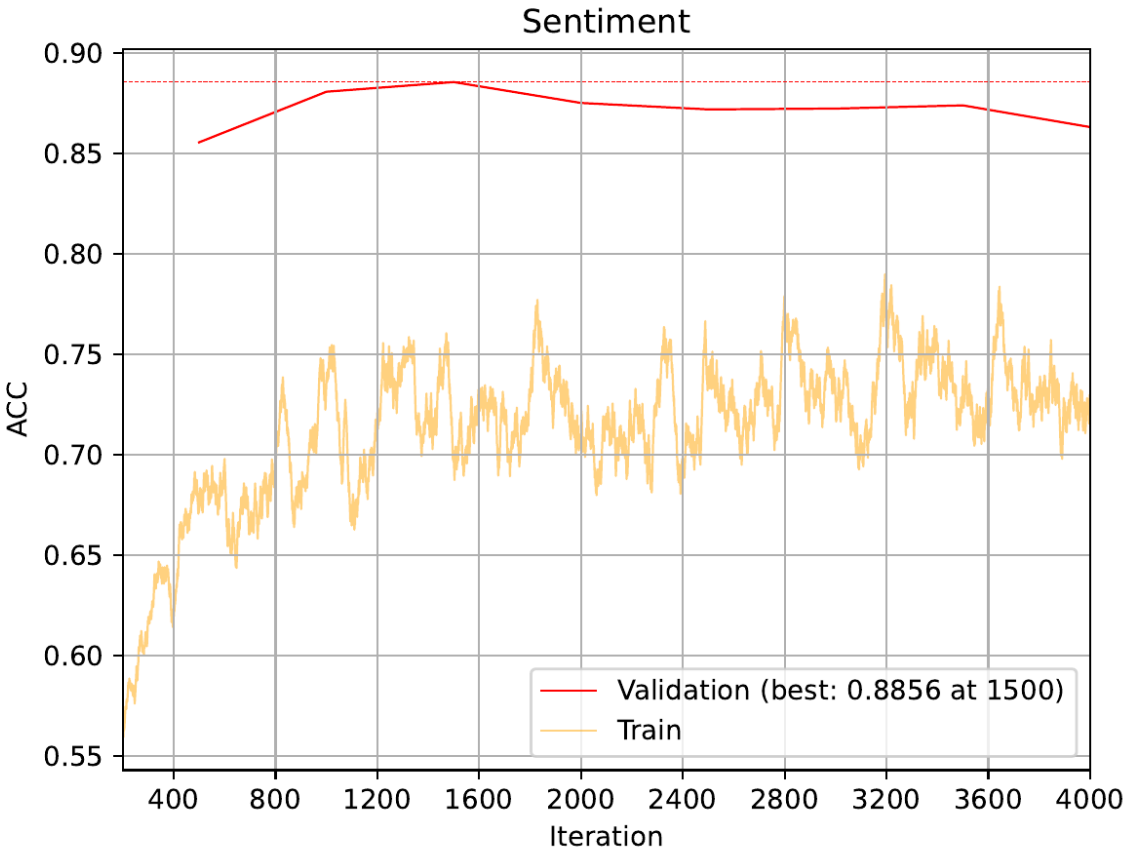
Note that when using these pretrained models, we are generally not configuring things like tokenizers and `model_config`, as we use the default tokenizers and configurations used to train the model. `EIR` will do this automatically when you leave the fields blank like above. Also notice the flag, `freeze_pretrained_model`, if set to `False`, we will not train the weights of the pretrained model but rather leave them as they are. This can greatly speed up training, but can come a cost of performance as we are not fine tuning the this part of our model for our task.

Note: For the pretrained models, we again take advantage of the excellent work from Hugging Face. In this case, the have a hub with a bunch of pretrained models, which we can use with `EIR`.

This model is quite a bit larger than the nones we have used so far so here it helps to have a powerful computer. We run this as always with:

```
eirtrain \
--global_configs eir_tutorials/a_using_eir/04_pretrained_sequence_tutorial/conf/04_imdb_
↪globals.yaml \
--input_configs eir_tutorials/a_using_eir/04_pretrained_sequence_tutorial/conf/04_imdb_
↪input_tiny-bert.yaml \
--output_configs eir_tutorials/a_using_eir/04_pretrained_sequence_tutorial/conf/04_imdb_
↪output.yaml \
--04_imdb_globals.output_folder=eir_tutorials/tutorial_runs/a_using_eir/tutorial_04_imdb_
↪run_tiny-bert
```

The training curve looks like so:



The pre-trained model performs quite similarly to our other long context models. However, notice how quickly it reached it top validation performance compared to the other models. Therefore, even though we are using a much bigger model than before, this kind of fine tuning can save us a lot of time!

Note: Many of these pretrained architectures are trained on data that is automatically crawled from the web. Therefore in this case, there might be possibility they have seen our reviews before as part of their training! Of course we are not too concerned for the sake of this tutorial.

E - Combining Models

So far we have seen how can can train bunch of cool models by themselves, but now we will be a bit cheeky and combined them into one big model.

Warning: Make sure that the `input_name` under the `input_info` field is unique for each input when doing combining models.

In this case, we will freeze the weights of the pretrained Tiny BERT part of our model.

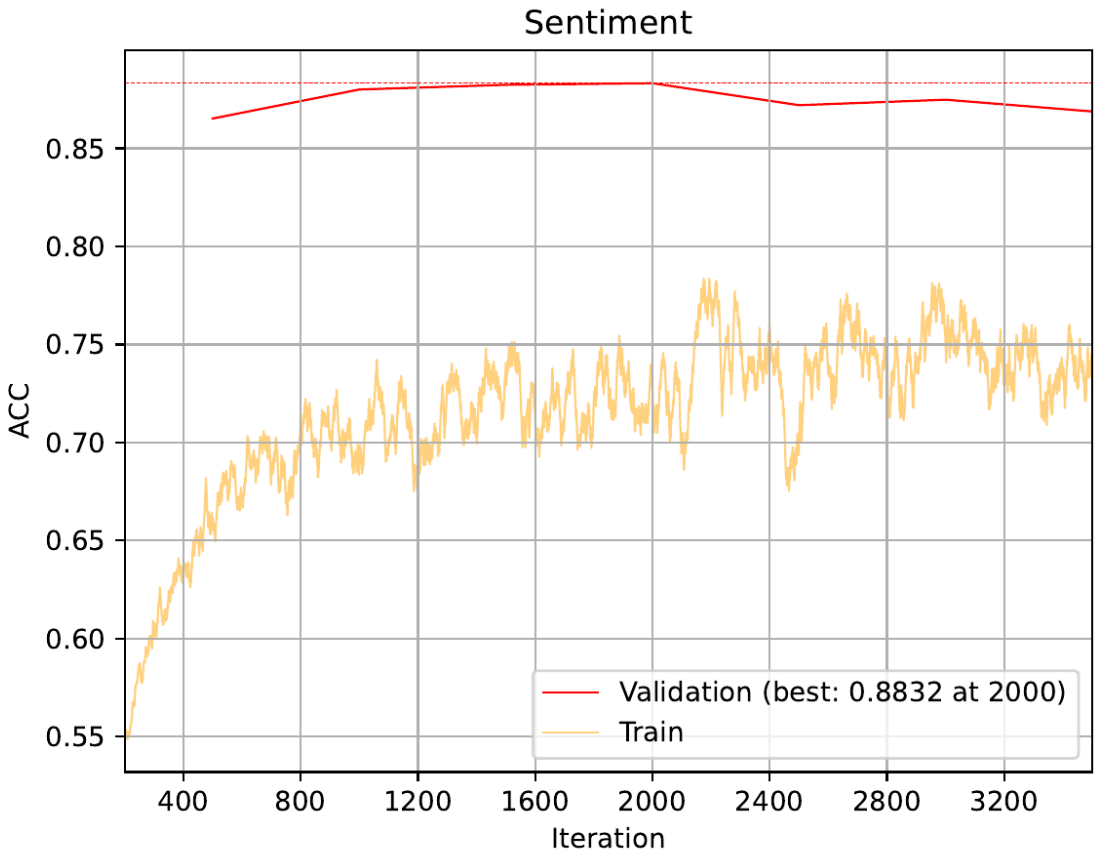
```
eirtrain \
--global_configs eir_tutorials/a_using_eir/04_pretrained_sequence_tutorial/conf/04_imdb_
```

(continues on next page)

(continued from previous page)

```
↪globals.yaml \
--input_configs eir_tutorials/a_using_eir/04_pretrained_sequence_tutorial/conf/04_imdb_
↪input_windowed.yaml eir_tutorials/a_using_eir/04_pretrained_sequence_tutorial/conf/04_
↪imdb_input_longformer.yaml eir_tutorials/a_using_eir/04_pretrained_sequence_tutorial/
↪conf/04_imdb_input_tiny-bert.yaml \
--output_configs eir_tutorials/a_using_eir/04_pretrained_sequence_tutorial/conf/04_imdb_
↪output.yaml \
--04_imdb_globals.output_folder=eir_tutorials/tutorial_runs/a_using_eir/tutorial_04_imdb_
↪run_combined \
--04_imdb_globals.device='cpu'
```

And our performance:



So in this case, we do not see a huge improvement when combining our models. However when relevant, it can greatly boost performance especially in those cases where the different input configurations refer to different modalities, i.e. do not just act on the same input like we did above.

Tip: Combining input configs is not only confined to sequence models or even the same modalities. For example, to train a model that uses genotype, sequence and tabular data, just pass the relevant configurations to the `--input_configs` flag!

F - Serving

In this final section, we demonstrate serving our trained model as a web service and interacting with it using HTTP requests.

Starting the Web Service

To serve the model, use the following command:

```
eirserve --model-path [MODEL_PATH]
```

Replace *[MODEL_PATH]* with the actual path to your trained model. This command initiates a web service that listens for incoming requests.

Here is an example of the command:

```
eirserve \
--model-path eir_tutorials/tutorial_runs/a_using_eir/tutorial_04_imdb_run_combined/saved_
↪models/tutorial_04_imdb_run_combined_model_1000_perf-average=0.8883.pt
```

Sending Requests

With the server running, we can now send requests. For this model, we send different features extracted from the same input text.

Here's an example Python function demonstrating this process:

```
import requests

def send_request(url: str, payload: dict):
    response = requests.post(url, json=payload)
    return response.json()

payload = {
    "imdb_reviews_windowed": "This movie was great! I loved it!",
    "imdb_reviews_longformer": "This movie was great! I loved it!",
    "imdb_reviews_tiny_bert": "This movie was great! I loved it!"
}

response = send_request('http://localhost:8000/predict', payload)
print(response)
```

Additionally, you can send requests using *bash*:

```
curl -X 'POST' \\
  'http://localhost:8000/predict' \\
  -H 'accept: application/json' \\
  -H 'Content-Type: application/json' \\
  -d '{
      "imdb_reviews_windowed": "This movie was great! I loved it!",
      "imdb_reviews_longformer": "This movie was great! I loved it!",
      "imdb_reviews_tiny_bert": "This movie was great! I loved it!"
  }'
```

Analyzing Responses

After sending requests to the served model, the responses can be analyzed. These responses provide insights into the model's predictions based on the input data.

Listing 32: predictions.json


```
[
    {
        "request": {
            "imdb_reviews_windowed": "This move was great! I loved it!",
            "imdb_reviews_longformer": "This move was great! I loved it!",
            "imdb_reviews_tiny_bert": "This move was great! I loved it!"
        },
        "response": {
            "result": {
                "imdb_output": {
                    "Sentiment": {
                        "Negative": 0.03049383871257305,
                        "Positive": 0.9695060849189758
                    }
                }
            }
        }
    },
    {
        "request": {
            "imdb_reviews_windowed": "This move was terrible! I hated it!",
            "imdb_reviews_longformer": "This move was terrible! I hated it!",
            "imdb_reviews_tiny_bert": "This move was terrible! I hated it!"
        },
        "response": {
            "result": {
                "imdb_output": {
                    "Sentiment": {
                        "Negative": 0.9445462822914124,
                        "Positive": 0.05545369163155556
                    }
                }
            }
        }
    },
    {
        "request": {
            "imdb_reviews_windowed": "You'll have to have your wits about you and your␣
↪brain fully switched on watching Oppenheimer as it could easily get away from a␣
↪nonattentive viewer. This is intelligent filmmaking which shows it's audience great␣
↪respect. It fires dialogue packed with information at a relentless pace and jumps to␣
↪very different times in Oppenheimer's life continuously through it's 3 hour runtime.␣
↪There are visual clues to guide the viewer through these times but again you'll have␣
↪to get to grips with these quite quickly. This relentlessness helps to express the␣
↪urgency with which the US attacked it's chase for the atomic bomb before Germany could␣
↪do the same. An absolute career best performance from (the consistenly brilliant)␣
```


(continues on next page)

(continued from previous page)


```
↪Cillian Murphy anchors the film. ",
            "imdb_reviews_longformer": "You'll have to have your wits about you and your␣
↪brain fully switched on watching Oppenheimer as it could easily get away from a␣
↪nonattentive viewer. This is intelligent filmmaking which shows it's audience great␣
↪respect. It fires dialogue packed with information at a relentless pace and jumps to␣
↪very different times in Oppenheimer's life continuously through it's 3 hour runtime.␣
↪There are visual clues to guide the viewer through these times but again you'll have␣
↪to get to grips with these quite quickly. This relentlessness helps to express the␣
↪urgency with which the US attacked it's chase for the atomic bomb before Germany could␣
↪do the same. An absolute career best performance from (the consistenly brilliant)␣
↪Cillian Murphy anchors the film. ",
            "imdb_reviews_tiny_bert": "You'll have to have your wits about you and your␣
↪brain fully switched on watching Oppenheimer as it could easily get away from a␣
↪nonattentive viewer. This is intelligent filmmaking which shows it's audience great␣
↪respect. It fires dialogue packed with information at a relentless pace and jumps to␣
↪very different times in Oppenheimer's life continuously through it's 3 hour runtime.␣
↪There are visual clues to guide the viewer through these times but again you'll have␣
↪to get to grips with these quite quickly. This relentlessness helps to express the␣
↪urgency with which the US attacked it's chase for the atomic bomb before Germany could␣
↪do the same. An absolute career best performance from (the consistenly brilliant)␣
↪Cillian Murphy anchors the film. "
        },
        "response": {
            "result": {
                "imdb_output": {
                    "Sentiment": {
                        "Negative": 0.031759195029735565,
                        "Positive": 0.9682407975196838
                    }
                }
            }
        }
    }
]
```


If you made it this far, I want to thank you for reading!

2.1.5 05 – Image Tutorial: Hot Dog or Not?

In this tutorial, we will be using `EIR` to train deep learning models for image classification. Specifically, we will be training our models in the important task of classifying whether an image contains a hot dog or not We will be using a subset of the Food-101 dataset, originally introduced here To download the data and configurations for this part of the tutorial, use this link.

Note that this tutorial assumes that you are already familiar with the basic functionality of the framework (see *01 – Genotype Tutorial: Ancestry Prediction*). If you have not already, it can also be useful to go over the sequence tutorial (see *03 – Sequence Tutorial: Movie Reviews and Peptides*).

A - Baseline

```
eir_tutorials/a_using_eir/05_image_tutorial/
├── conf
│   ├── globals.yaml
│   ├── inputs.yaml
│   ├── inputs_efficientnet_b0.yaml
│   ├── inputs_resnet18.yaml
│   └── output.yaml
└── data
    └── hot_dog_not_hot_dog
        ├── food_images
        └── labels.csv
```

Looking at the data we are working with, we can indeed see that it contains images of hot dogs and all kinds of other food:

I did not know drinking coffee/cacao with hot dogs was a thing. Anyway, now we will train a simple residual network from scratch to get a little baseline. The image models we be using come from the excellent timm library, which includes those used in this tutorial and many more!

To the configuration!

Listing 33: globals.yaml

```yaml
output_folder: eir_tutorials/tutorial_runs/a_using_eir/tutorial_05_is_it_a_hot_dog
valid_size: 0.10
device: "mps"
batch_size: 32
n_saved_models: 1
```

(continues on next page)

(continued from previous page)

```
dataloader_workers: 0
checkpoint_interval: 100
sample_interval: 100
n_epochs: 200
memory_dataset: True
max_attributions_per_class: 10
compute_attributions: True
mixing_alpha: 0.5
plot_skip_steps: 0
```

Listing 34: inputs.yaml

```
input_info:
  input_source: eir_tutorials/a_using_eir/05_image_tutorial/data/hot_dog_not_hot_dog/
↪food_images
  input_name: hot_dog
  input_type: image

input_type_info:
  mixing_subtype: "cutmix"
  size:
    - 64

model_config:
  model_type: "ResNet"
  model_init_config:
    layers: [1, 1, 1, 1]
    block: "BasicBlock"

interpretation_config:
    num_samples_to_interpret: 30
```

Listing 35: output.yaml

```
output_info:
        output_source: eir_tutorials/a_using_eir/05_image_tutorial/data/hot_dog_not_hot_
↪dog/labels.csv
        output_name: hot_dog_output
        output_type: tabular

output_type_info:
        target_cat_columns:
                - CLASS
```

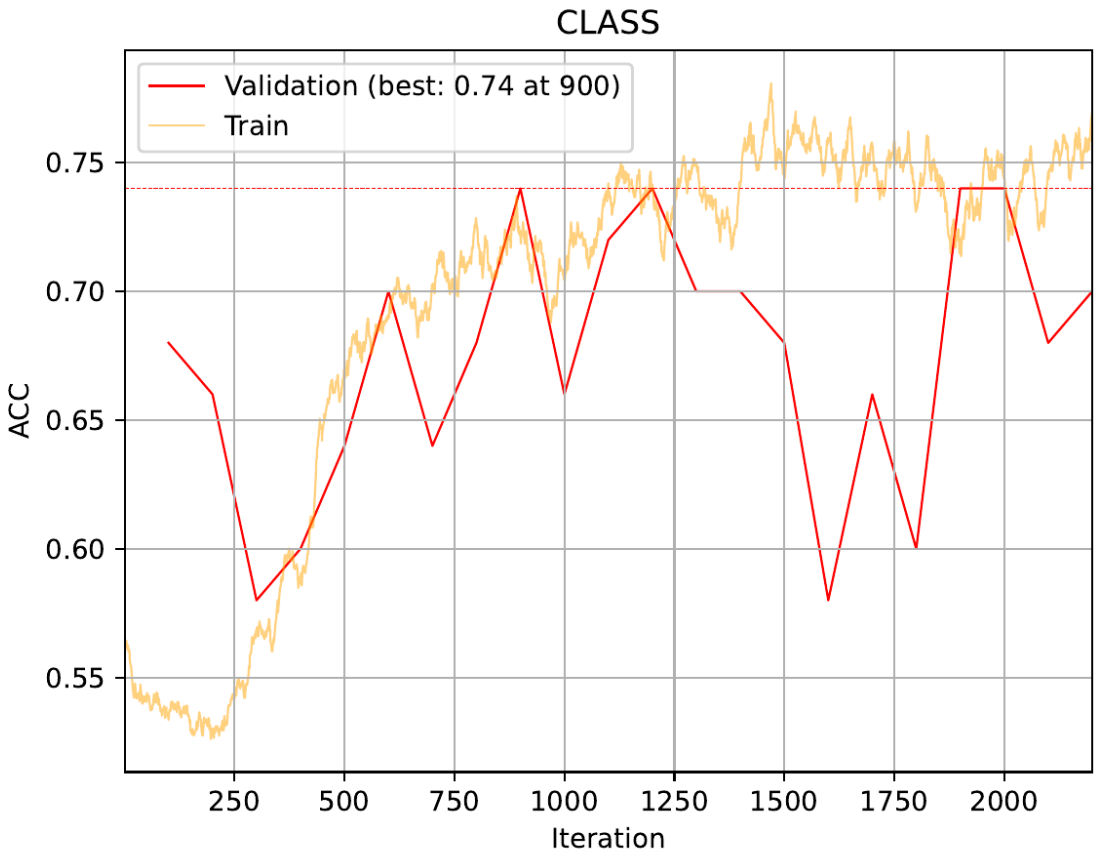
As usually, we do our training with the following command:

```
eirtrain \
--global_configs eir_tutorials/a_using_eir/05_image_tutorial/conf/globals.yaml \
--input_configs eir_tutorials/a_using_eir/05_image_tutorial/conf/inputs.yaml \
--output_configs eir_tutorials/a_using_eir/05_image_tutorial/conf/output.yaml
```

Note: Training these deep image models can take quite some time if one is using a laptop. If possible, try using a

system with a GPU available!

Now for the results, we see the following:



That looks *kind of* ok, but far from great. Our validation performance is all over the place (a contributing factor could be that our validation set here is very small), and we don't get a better performance than around 76%. Certainly not good enough for an actual app!

B - Pretrained Image Model

Now we will take advantage of the fact that there exist pretrained models that have been trained on a bunch of data (not just a few pictures of hot dogs and other food) and see whether that helps our performance.

Now our input configuration looks like this:

Listing 36: inputs_resnet18.yaml

```
input_info:
  input_source: eir_tutorials/a_using_eir/05_image_tutorial/data/hot_dog_not_hot_dog/
↪food_images
  input_name: hot_dog_resnet18
  input_type: image
```

(continues on next page)

(continued from previous page)

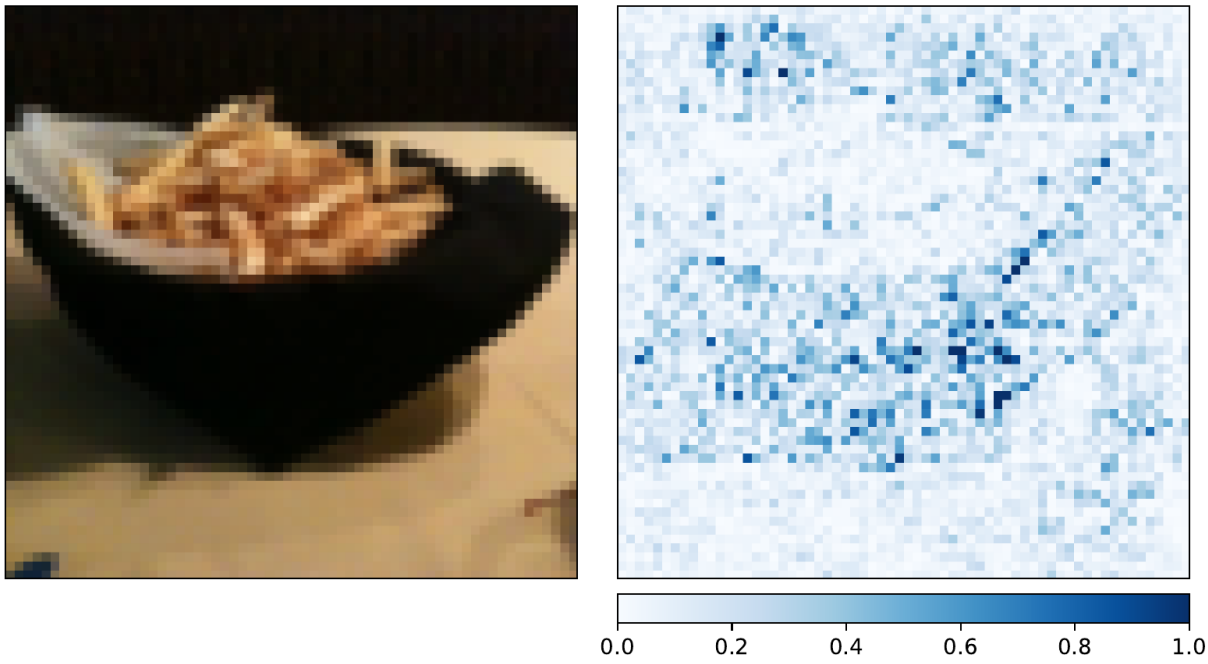
```yaml
input_type_info:
  mixing_subtype: "cutmix"
  size:
    - 64

model_config:
  model_type: "resnet18"
  pretrained_model: True

interpretation_config:
    num_samples_to_interpret: 30
```

To train, we run:

```
eirtrain \
--global_configs eir_tutorials/a_using_eir/05_image_tutorial/conf/globals.yaml \
--input_configs eir_tutorials/a_using_eir/05_image_tutorial/conf/inputs_resnet18.yaml \
--output_configs eir_tutorials/a_using_eir/05_image_tutorial/conf/output.yaml \
--globals.output_folder=eir_tutorials/tutorial_runs/a_using_eir/tutorial_05_is_it_a_hot_
↪dog_pretrained_resnet
```

Looking at our performance, we see:

Definitely better! One factor here could be that we are training on different image sizes than the original model was trained on. In any case, let's have a look at what our models are focusing on when deciding something is *not* a hot dog. (perhaps you already noticed we set the `compute_attributions` value to `True` in the global configuration):



That is not a hot dog alright, and our model seems to agree.

C - Combining pretrained image models

For the last part of this tutorial, we will be combining two pretrained models. We will keep the ResNet18 models as it is, feeding it 64 pixel images. We will also add a EfficientNet-B0 feature extractor, but feed it 224 pixel images.

The configuration for the EfficientNet part looks like this:

Listing 37: inputs_efficientnet_b0.yaml

```
input_info:
  input_source: eir_tutorials/a_using_eir/05_image_tutorial/data/hot_dog_not_hot_dog/
↪food_images
  input_name: hot_dog_efficientnet
  input_type: image

input_type_info:
  mixing_subtype: "cutmix"
  size:
```

(continues on next page)

(continued from previous page)

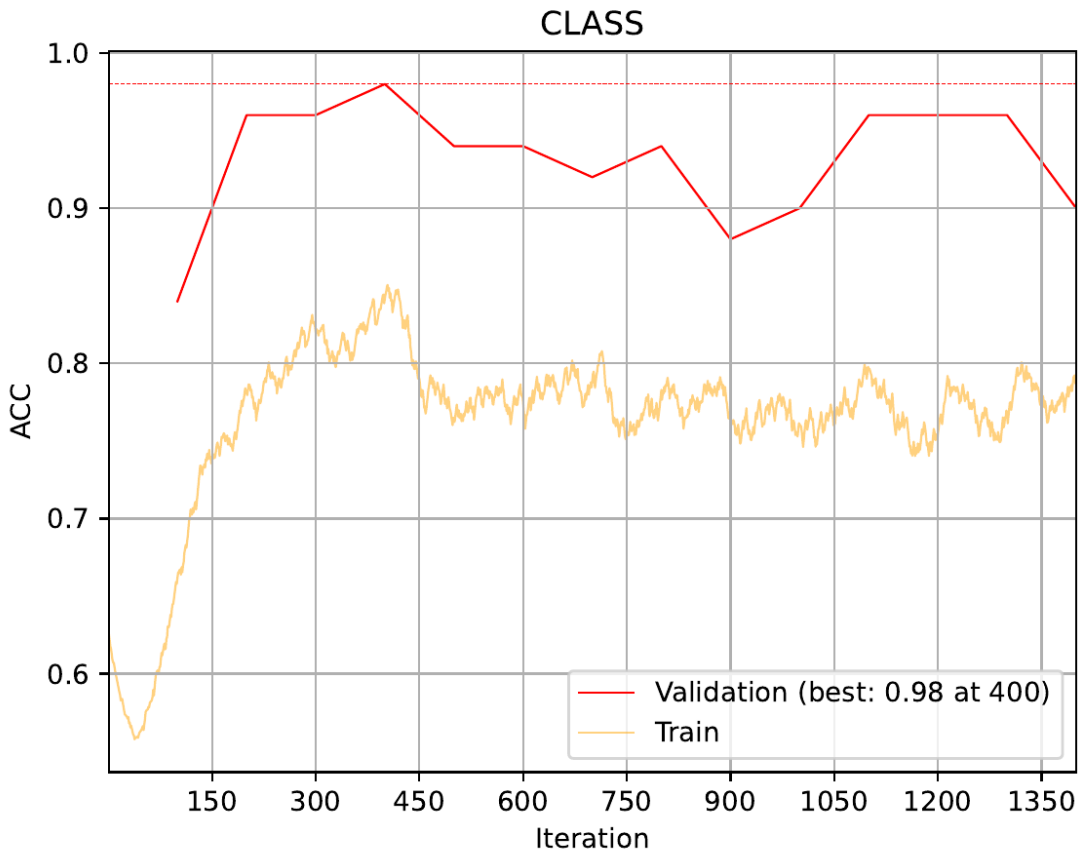
```
    - 224

model_config:
  model_type: "efficientnet_b0"
  pretrained_model: True

interpretation_config:
    num_samples_to_interpret: 30
```

Training as usual, notice that we are now passing in both input configurations to the `--input_configs` flag.

```
eirtrain \
--global_configs eir_tutorials/a_using_eir/05_image_tutorial/conf/globals.yaml \
--input_configs eir_tutorials/a_using_eir/05_image_tutorial/conf/inputs_efficientnet_b0.
↪yaml eir_tutorials/a_using_eir/05_image_tutorial/conf/inputs_resnet18.yaml \
--output_configs eir_tutorials/a_using_eir/05_image_tutorial/conf/output.yaml \
--globals.output_folder=eir_tutorials/tutorial_runs/a_using_eir/tutorial_05_is_it_a_hot_
↪dog_pretrained_combined
```

Note: Here we are maybe getting ahead of ourselves a little and going straight into combining models. Perhaps only using EfficientNet performs even better. I will leave that task to you, dear reader.

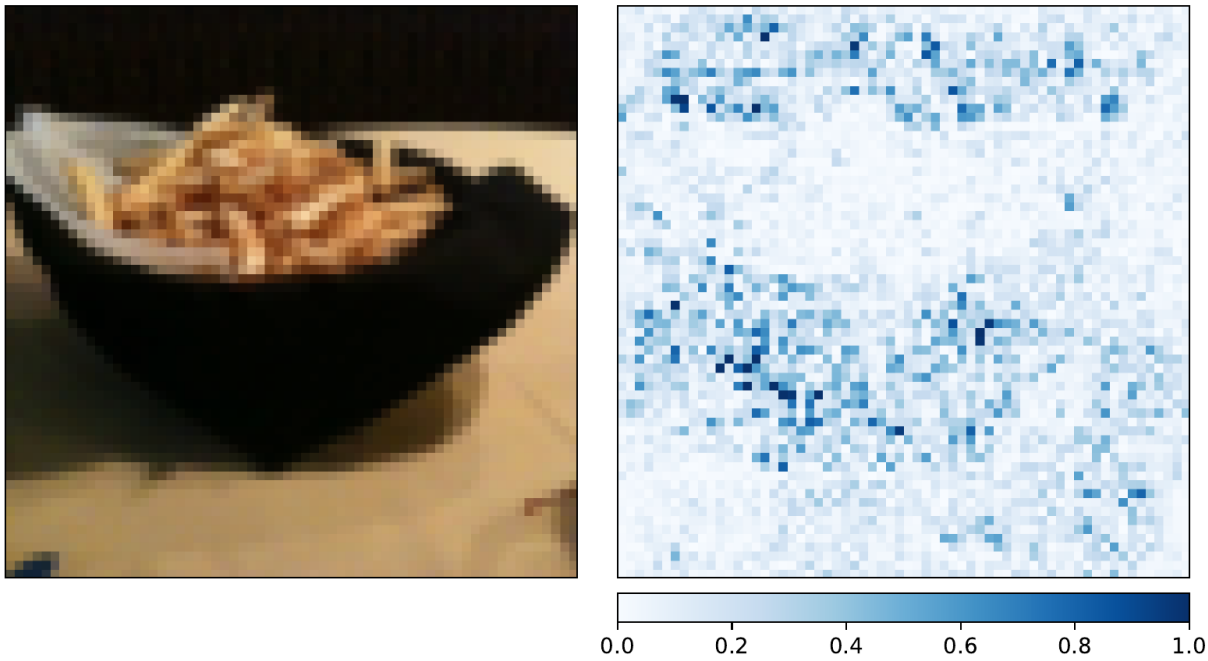
The training and validation curves I got look like so (I got a bit impatient and stopped the run early):



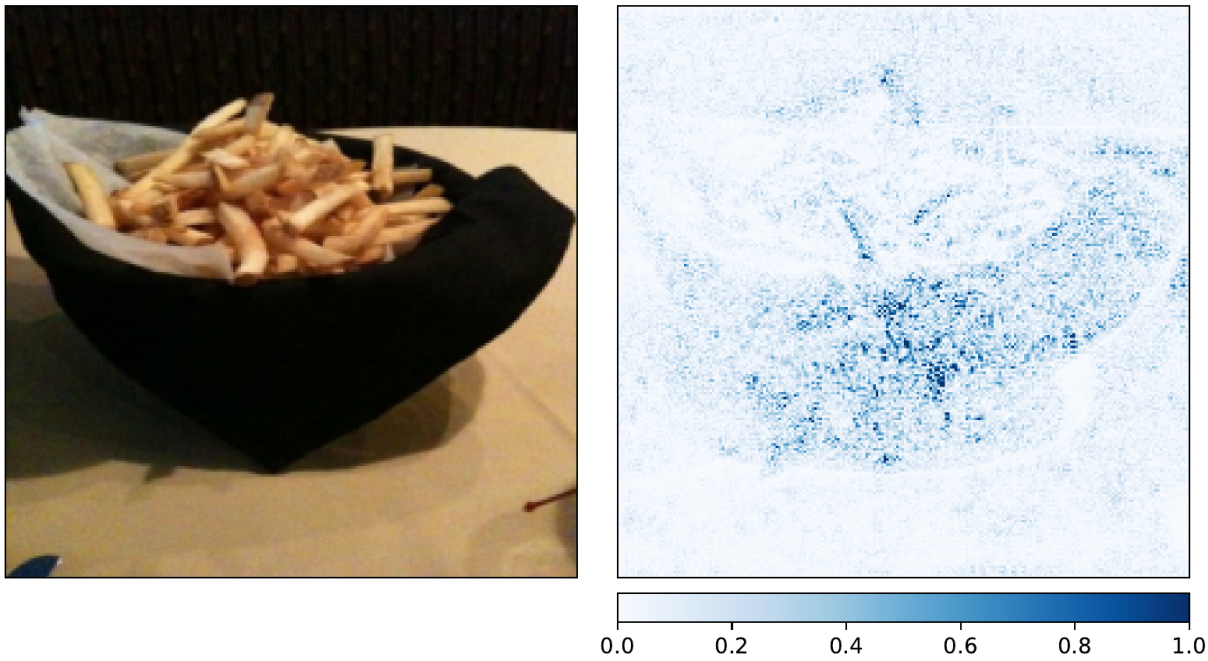
Definitely looks more stable, and better performance than before. As mentioned earlier, we should be careful about trusting these results too much as we have a tiny validation set, but since we are doing a tutorial, we'll allow it!

For the last part of this tutorial, let's have a look at what the our features extractors focus on for an example image.

First the ResNet18 feature extractor:



And then the EfficientNet-B0 feature extractor:



While it's definitely more clear to the human eye in the ResNet18 case, both feature extractors seem to be focusing on the french fries when deciding that this is indeed, not a hot dog.

D - Serving

In this final section, we demonstrate serving our trained image classification model as a web service and interacting with it using HTTP requests.

Starting the Web Service

To serve the model, use the following command:

```
eirserve --model-path [MODEL_PATH]
```

Replace *[MODEL_PATH]* with the actual path to your trained model. This command initiates a web service that listens for incoming requests.

Here is an example of the command:

```
eirserve \
--model-path eir_tutorials/tutorial_runs/a_using_eir/tutorial_05_is_it_a_hot_dog_
↪pretrained_combined/saved_models/tutorial_05_is_it_a_hot_dog_pretrained_combined_model_
↪400_perf-average=0.9857.pt
```

Sending Requests

With the server running, we can now send image-based requests. For this model, we send encoded images to different feature extraction endpoints.

Here's an example Python function demonstrating this process:

```
import requests
import base64
from PIL import Image
from io import BytesIO

def encode_image_to_base64(file_path: str) -> str:
    with Image.open(file_path) as image:
        buffered = BytesIO()
        image.save(buffered, format="JPEG")
        return base64.b64encode(buffered.getvalue()).decode("utf-8")

def send_request(url: str, payload: dict):
    response = requests.post(url, json=payload)
    return response.json()

payload = {
    "hot_dog_efficientnet": encode_image_to_base64("path/to/image1.jpg"),
    "hot_dog_resnet18": encode_image_to_base64("path/to/image1.jpg")
}

response = send_request('http://localhost:8000/predict', payload)
print(response)
```

Additionally, you can send requests using *bash*. Note that this requires preparing the base64-encoded image content in advance:

```
curl -X 'POST' \\
  'http://localhost:8000/predict' \\
  -H 'accept: application/json' \\
  -H 'Content-Type: application/json' \\
  -d '{
      "hot_dog_efficientnet": "[BASE64_ENCODED_IMAGE]",
      "hot_dog_resnet18": "[BASE64_ENCODED_IMAGE]"
  }'
```

Analyzing Responses

Before we going into the responses, let's view the images that were used for predictions:

1040579.jpg

108743.jpg

After sending requests to the served model, the responses can be analyzed. These responses provide insights into the model's predictions based on the input images.

Listing 38: predictions.json


```
[
    {
        "request": {
            "hot_dog_efficientnet": "eir_tutorials/a_using_eir/05_image_tutorial/data/
↪hot_dog_not_hot_dog/food_images/1040579.jpg",
            "hot_dog_resnet18": "eir_tutorials/a_using_eir/05_image_tutorial/data/hot_
↪dog_not_hot_dog/food_images/1040579.jpg"
        },
        "response": {
            "result": {
                "hot_dog_output": {
                    "CLASS": {
                        "Hot Dog": 0.8565942049026489,
                        "Not Hot Dog": 0.14340578019618988
                    }
                }
            }
        }
    },
    {
        "request": {
            "hot_dog_efficientnet": "eir_tutorials/a_using_eir/05_image_tutorial/data/
↪hot_dog_not_hot_dog/food_images/108743.jpg",
            "hot_dog_resnet18": "eir_tutorials/a_using_eir/05_image_tutorial/data/hot_
↪dog_not_hot_dog/food_images/108743.jpg"
        },
        "response": {
            "result": {
                "hot_dog_output": {
                    "CLASS": {
                        "Hot Dog": 0.07436760514974594,
                        "Not Hot Dog": 0.9256323575973511
                    }
                }
            }
        }
    }
]
```


With that, we conclude this image tutorial. Thank you for reading!

2.1.6 06 – Training on binary data

Today, for this tutorial, we will be training deep learning models on raw binary data. In general, it is a good approach to use inductive bias and domain expertise when training our models, but sometimes we might not have a good idea of how to present our data, or we simply want to turn off our brains for a bit and throw raw compute at our problem. We will be using the familiar IMDB reviews dataset, see here for more information about the data. To download the data and configurations for this part of the tutorial, use this link.

A - Local Transformer

After downloading the data, the folder structure should look like this:

```
eir_tutorials/a_using_eir/06_raw_bytes_tutorial/
├── conf
│   ├── globals.yaml
│   ├── input.yaml
│   └── output.yaml
└── data
    └── IMDB
        ├── IMDB_Reviews
        ├── conf
        ├── imdb.vocab
        └── imdb_labels.csv
```

We will use the built-in local transformer model in `EIR` for this tutorial.

If you have done the previous tutorials you might be used to this, but the configurations are here:

Listing 39: globals.yaml

```yaml
output_folder: eir_tutorials/tutorial_runs/a_using_eir/tutorial_06_imdb_sentiment_binary
valid_size: 0.10
n_saved_models: 1
device: "mps"
checkpoint_interval: 1000
sample_interval: 1000
dataloader_workers: 0
memory_dataset: true
n_epochs: 50
mixing_alpha: 0.5
```

Listing 40: input.yaml

```yaml
input_info:
  input_source: eir_tutorials/a_using_eir/03_sequence_tutorial/data/IMDB/IMDB_Reviews
  input_name: imdb_reviews_bytes_base_transformer
  input_type: bytes

input_type_info:
        sampling_strategy_if_longer: "uniform"
        max_length: 1024

model_config:
```

(continues on next page)

(continued from previous page)

```
        model_type: sequence-default
        window_size: 128
        embedding_dim: 64
        pool: avg
        position: "embed"
        model_init_config:
          num_layers: 4
          num_heads: 8
```

Listing 41: output.yaml

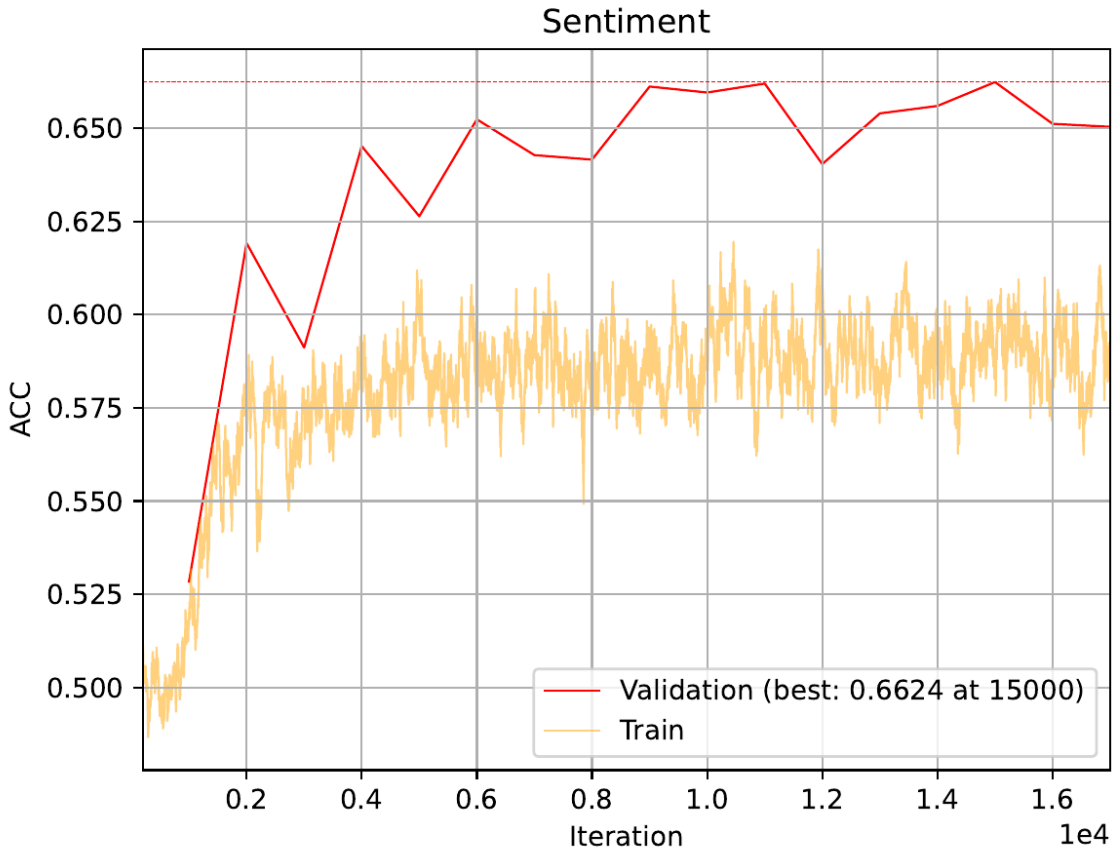
```
output_info:
        output_source: eir_tutorials/a_using_eir/03_sequence_tutorial/data/IMDB/imdb_
↪labels.csv
        output_name: imdb_output
        output_type: tabular

output_type_info:
        target_cat_columns:
                - Sentiment
```

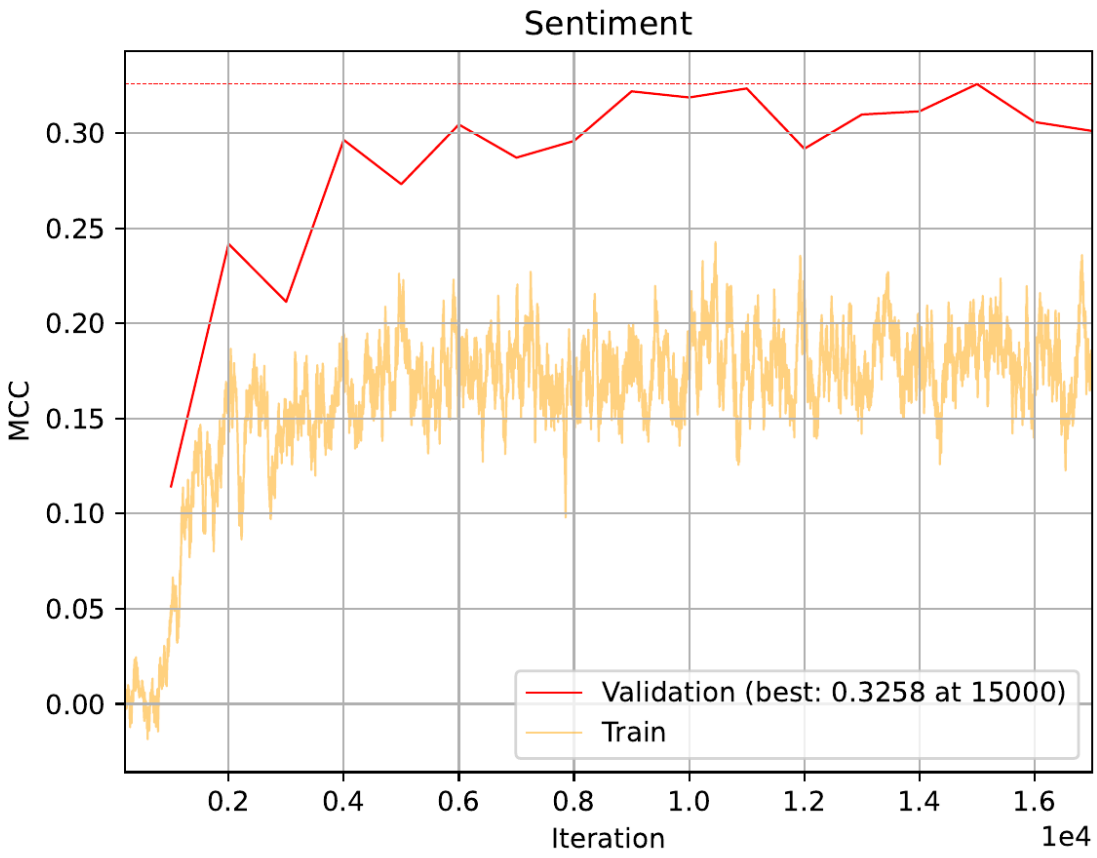
Note: The model we are training here is relatively deep, so you probably need a GPU to train it in a reasonable amount of time. If you do not have access to a GPU, try reducing the number of layers and the sequence length.

As usual, we can run the following command to train:

```
eirtrain \
--global_configs eir_tutorials/a_using_eir/06_raw_bytes_tutorial/conf/globals.yaml \
--input_configs eir_tutorials/a_using_eir/06_raw_bytes_tutorial/conf/input.yaml \
--output_configs eir_tutorials/a_using_eir/06_raw_bytes_tutorial/conf/output.yaml
```

When training, I got the following training curves:


Sentiment
ACC
0.650
0.625
0.600
0.575
0.550
0.525
0.500
0.2
0.4
0.6
0.8
1.0
1.2
1.4
1.6
Iteration
1e4
Validation (best: 0.6624 at 15000)
Train



Not so great, but not a complete failure either! When comparing with our previous modelling on this task (see *03 – Sequence Tutorial: Movie Reviews and Peptides*), we definitely performed better when doing word level modelling compared to running on the raw bytes like we are doing here. It can well be we need to configure our model better, or train it on more data, but for now we will say that adapting the training to the task (in this case NLP) seems to perform better than training on raw binary data.

Tip: Here we are training on natural language data, but the approach here can in theory be applied to any type of file on a disk (e.g. images, videos, or other more obscure formats). As we saw above however, good results not guaranteed!

B - Serving

In this section, we'll guide you through serving our t rained IMDB Reviews Bytes Classification model as a web service and show you how to interact with it using HTTP requests.

Starting the Web Service

To serve the model, execute the following command:

```
eirserve --model-path [MODEL_PATH]
```

Replace *[MODEL_PATH]* with the actual path to your trained model. This command initiates a web service that listens for incoming HTTP requests.

Here is an example of the command used:

```
eirdeploy \
--model-path eir_tutorials/tutorial_runs/a_using_eir/tutorial_06_imdb_sentiment_binary/
↪saved_models/tutorial_06_imdb_sentiment_binary_model_15000_perf-average=0.5741.pt
```

Sending Requests

Once the server is up and running, you can send requests to it. For this binary model, we send text data in byte format to the model's endpoint.

Here's an example Python function to demonstrate how to send a request:

```
import requests
import numpy as np
import base64

def load_and_encode_data(data_pointer: str) -> str:
    arr = np.fromfile(data_pointer, dtype="uint8")
    arr_bytes = arr.tobytes()
    return base64.b64encode(arr_bytes).decode("utf-8")

def send_request(url: str, encoded_data: str):
    payload = {"data": encoded_data}
    response = requests.post(url, json=payload)
    return response.json()

encoded_data = load_and_encode_data('path/to/textfile.txt')
response = send_request('http://localhost:8000/predict', encoded_data)
print(response)
```

Analyzing Responses

After sending requests to the served model, you will receive responses that provide insights into the model's predictions based on the input text data.

Let's take a look at some of the text data used for predictions:

Listing 42: 10021_2.txt

```
The worst movie I have seen since Tera Jadoo Chal Gaya. There is no story, no humor, no␣
↪nothing! The action sequences seem more like a series of haphazard Akshay Kumar Thumbs-
↪Up advertisements stitched together. Heavily influenced from The Matrix and Kung-Fu␣
↪Hustle but very poorly executed.<br /><br />I did not go a lot of expectations, but␣
↪watching this movie is an exasperating experience which makes you wonder "What were␣
↪these guys thinking??!!".<br /><br />The only thing you might remember after watching␣
↪it is an anorexic Kareena in a bikini.<br /><br />The reason why I did not give a␣
↪rating of '1' is that every time I think I have seen the worst, Bollywood proves me␣
↪wrong.
```

Listing 43: 10132_9.txt

```
In this first episode of Friends, we are introduced to the 6 main characters of the␣
↪series: Monica Geller,Phoebe Buffay,Chandler Bing,Ross Geller, Joey Tribbiani and␣
↪eventually Rachel Green .<br /><br />We discover that Rachel, a rich girl that is␣
↪Monica's friend from high school times, left her fiancé, Barry, at the altar, since␣
↪she discovered she didn't love him. She also decides to live with Monica and become␣
↪independent from her father,getting a new job as a waitress in Central Perk.<br /><br /
↪>Ross, for the other hand,discovered his wife is a lesbian and lost her for Susan, her␣
↪partner. (We see him moving to a new apartment during the episode)<br /><br />Monica,␣
↪in this episode, makes out (and eventually sleeps) with Paul "the wine guy", who gave␣
↪her the excuse of being impotent since he divorced his wife. But in reality, he was␣
↪just deceiving her.<br /><br />Ps: I just loooove Joey's and Chandler's haircuts in␣
↪this first season! =)
```

Here are examples of the model's predictions:

Listing 44: predictions.json

```
[
    {
        "request": {
            "imdb_reviews_bytes_base_transformer": "eir_tutorials/a_using_eir/03_
↪sequence_tutorial/data/IMDB/IMDB_Reviews/10021_2.txt"
        },
        "response": {
            "result": {
                "imdb_output": {
                    "Sentiment": {
                        "Negative": 0.7403308749198914,
                        "Positive": 0.25966906547546387
                    }
                }
            }
        }
```

(continues on next page)

(continued from previous page)

```
    },
    {
        "request": {
            "imdb_reviews_bytes_base_transformer": "eir_tutorials/a_using_eir/03_
↪sequence_tutorial/data/IMDB/IMDB_Reviews/10132_9.txt"
        },
        "response": {
            "result": {
                "imdb_output": {
                    "Sentiment": {
                        "Negative": 0.22369135916233063,
                        "Positive": 0.7763086557388306
                    }
                }
            }
        }
    }
]
```

This concludes our tutorial, thank you for following along!

2.1.7 07 – Multimodal Training: Combining Tabular, Text, and Image

Here, we will be exploring multi-modal training. That is, training a model on multiple different types of data. For example, we can train a model to predict some output based on both text and images. We will be using a subset of a dataset from PetFinder.my a Malaysian website that matches adopters with homeless pets. The dataset contains images of pets, as well as some text-based description of the pets, and finally some tabular data.

So here, the task will be to predict the speed at which a pet will be adopted. This is formed here as a classification task with 4 different classes, where the classes are the number of days it took for the pet to be adopted.

To download the data for this part of the tutorial, use this link.

Note: Here we have combined the 5 classes from the original dataset into 4 classes for this tutorial, as one of the classes was very small compared to the others. However, the original classes are still available in the main tabular file.

After downloading the data, the folder structure should look like this (note that we will create the configuration files ourselves in the tutorial as we go along):

```
eir_tutorials/a_using_eir/07_multimodal_tutorial/
├── conf
│   ├── 07_apx-a_input_description_pretrained.yaml
│   ├── 07_apx-b_mt_input_tabular.yaml
│   ├── 07_apx-b_mt_output.yaml
│   ├── 07_fusion.yaml
│   ├── 07_globals.yaml
│   ├── 07_input_description.yaml
│   ├── 07_input_image.yaml
│   ├── 07_input_tabular.yaml
│   └── 07_output.yaml
└── data
```

(continues on next page)

(continued from previous page)

```
    ├── descriptions.csv
    ├── images
    └── tabular.csv
```

We are in for a relatively long tutorial, so I'll try to keep it concise. Let's get started!

A - Tabular Data

First, we will start training only on the tabular data, which is stored in a CSV file. Note that here the tabular data has been transposed, for visual purposes.

Here are the configurations files for the tabular data:

Listing 45: 07_globals.yaml

```yaml
output_folder: eir_tutorials/tutorial_runs/a_using_eir/tutorial_07_multimodal_run
valid_size: 0.10
memory_dataset: true
checkpoint_interval: 200
sample_interval: 200
n_epochs: 25
device: "cpu"
lr: 5.0e-04
optimizer: adamw
gradient_clipping: 1.0
early_stopping_patience: 5
early_stopping_buffer: 2000
compute_attributions: false
attributions_every_sample_factor: 10
max_attributions_per_class: 512
mixing_alpha: 0.2
```

Listing 46: 07_input_tabular.yaml

```yaml
input_info:
  input_source: eir_tutorials/a_using_eir/07_multimodal_tutorial/data/tabular.csv
  input_name: pets_tabular
  input_type: tabular

input_type_info:
  input_cat_columns:
          - Type
          - Breed1
          - Breed2
          - Gender
          - Color1
          - Color2
          - Color3
          - MaturitySize
          - State
          - FurLength
          - Vaccinated
```

(continues on next page)

(continued from previous page)

```
          - Dewormed
          - Sterilized
          - Health
          - Fee

  input_con_columns:
          - Age
          - Quantity
          - VideoAmt
          - PhotoAmt

model_config:
  model_type: tabular
```

Listing 47: 07_fusion.yaml

```
model_config:
  layers:
  - 2
  rb_do: 0.25
model_type: mlp-residual
```

Listing 48: 07_output.yaml

```
output_info:
  output_source: eir_tutorials/a_using_eir/07_multimodal_tutorial/data/tabular.csv
  output_name: pet_adoption
  output_type: tabular

output_type_info:
  target_cat_columns:
    - AdoptionSpeed
  cat_label_smoothing: 0.1

model_config:
  model_init_config:
    layers:
      - 2
    fc_do: 0.25
    rb_do: 0.25
    stochastic_depth_p: 0.25
```

As usual, we can run the following command to train:

```
eirtrain \
--global_configs eir_tutorials/a_using_eir/07_multimodal_tutorial/conf/07_globals.yaml \
--input_configs eir_tutorials/a_using_eir/07_multimodal_tutorial/conf/07_input_tabular.
↪yaml \
--fusion_configs eir_tutorials/a_using_eir/07_multimodal_tutorial/conf/07_fusion.yaml \
--output_configs eir_tutorials/a_using_eir/07_multimodal_tutorial/conf/07_output.yaml \
--07_globals.output_folder=eir_tutorials/tutorial_runs/a_using_eir/tutorial_07a_
↪multimodal_tabular \
```

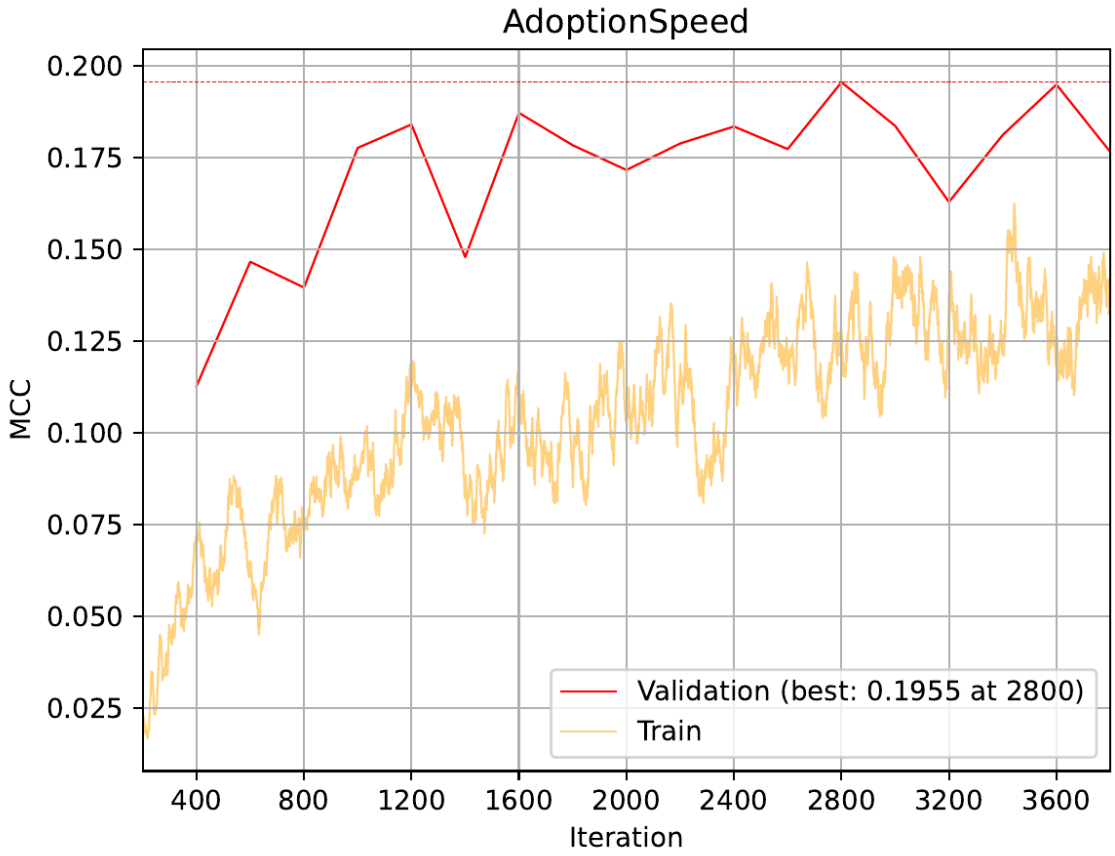
(continues on next page)

(continued from previous page)

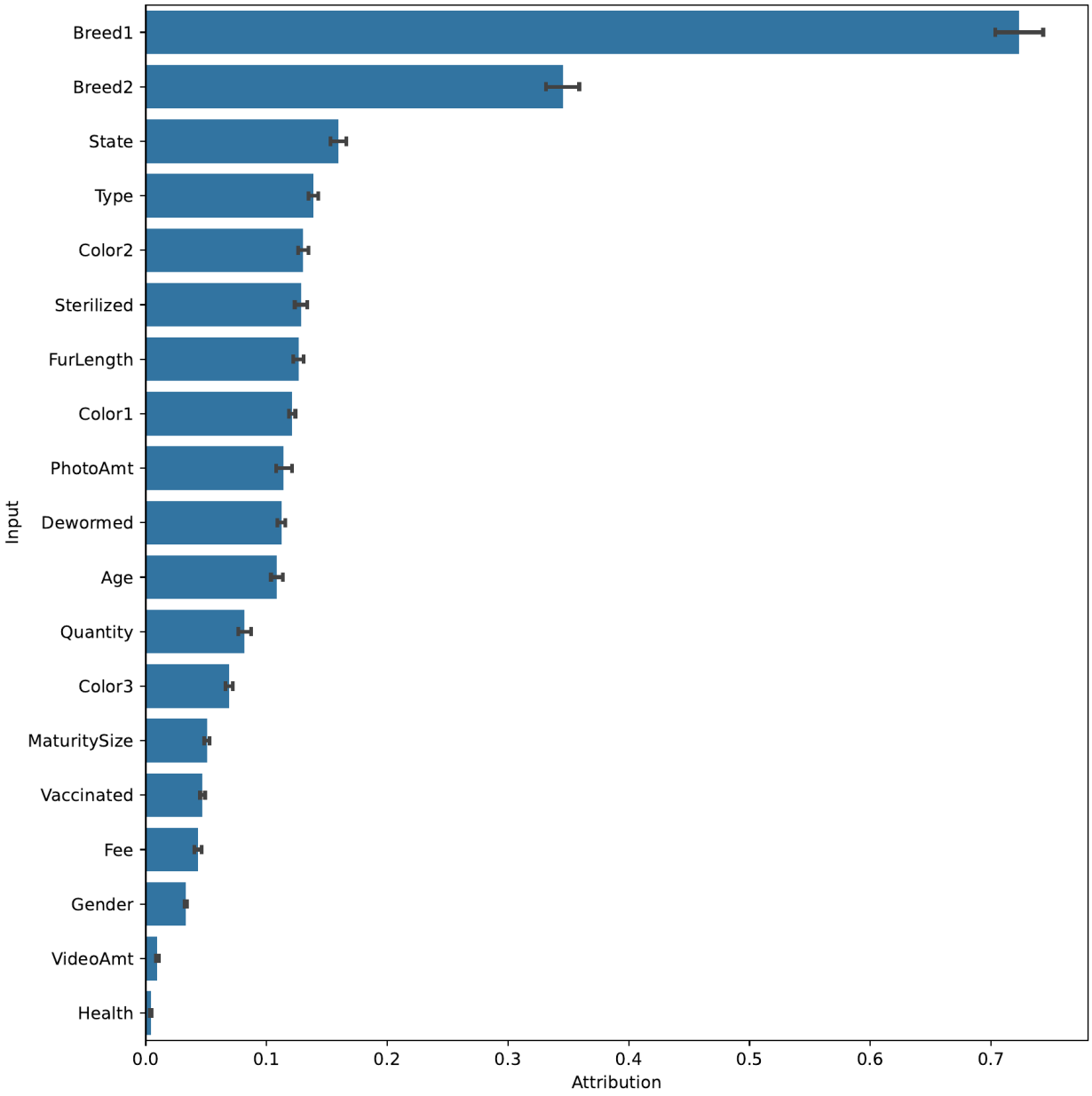
```
--07_globals.compute_attributions=true
```

Note: Here we are setting the `--compute_attributions=true` parameter, from the command line, to get the integrated gradients attributions of the model w.r.t. the tabular input data.

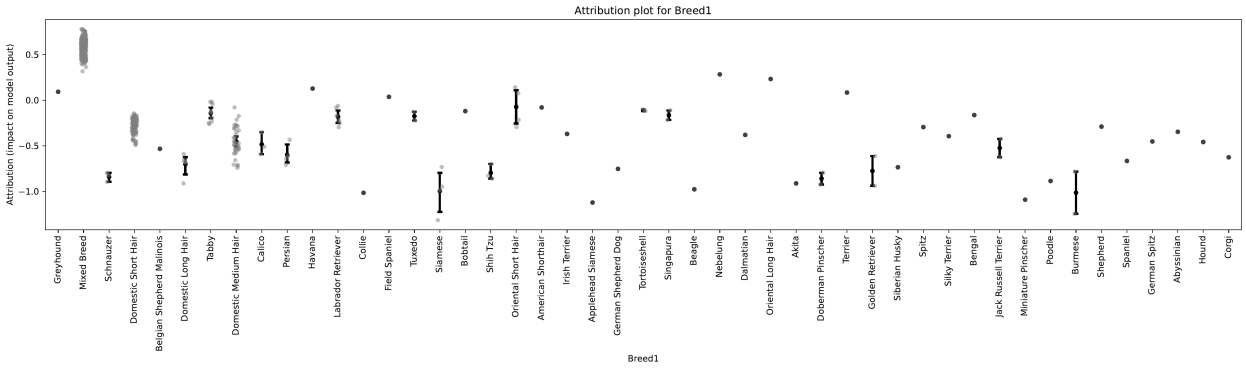
When training, I got the following training curve:



Now, since we set the `--compute_attributions=true` parameter, we can have a look at the attributions (notice in the global configuration, we set `compute_attributions_ever_sample_factor=10`, which means they are computed every 10 sampling iterations, i.e. 200 * 10 = 2000 training iterations). Specifically, we check the file under `samples/4000/attributions/` in the `results` folder. First, we can have a look at the feature importance for the tabular data.



Here we can see that `Breed1` is the feature that most strongly influenced the model's prediction. In the `attributions` folder, we can also see how the inputs influence the model towards a specific class. Here, we will look at how the `Breed1` input values influence the model towards the class "D: 100+ Days", meaning the pet was adopted after 100 days:



So from this it seems that, unfortunately, mixed breed pets are less likely to be adopted (that is, the value "Mixed Breed" pushes the model towards making the "D: 100+ Days" prediction). This does perhaps make intuitive sense, but keep in mind that this is specifically analyzing the behavior of the model, and not guaranteed to be true, causal relationships. Additionally, this is something that could likely be discovered with simpler methods, such as a logistic regression model. However, this is just an example of how to use the integrated gradients attributions to analyze the deep-learning model.

B - Tabular + Text Data

Now, we will train the model on both tabular and text data. The text data in question are descriptions of the cute pets, which are stored in a CSV file.

Note: When reading sequence data from a CSV file, the file must follow the specification of having two columns, one containing the sequence IDs ("ID"), and the other containing the sequence data ("Sequence"). Note that the names of these columns are strictly enforced.

First, let's take a look at an example from the text data:

```
Nibble is a 3+ month old ball of cuteness. He is energetic and playful. I rescued a␣
↪couple of cats a few months ago but could not get them neutered in time as the clinic␣
↪was fully scheduled. The result was this little kitty. I do not have enough space and␣
↪funds to care for more cats in my household. Looking for responsible people to take␣
↪over Nibble's care.
```

So to train on both tabular and text data, we will need to specify a configuration for the text data as well:

```
input_info:
  input_source: eir_tutorials/a_using_eir/07_multimodal_tutorial/data/descriptions.csv
  input_name: pet_descriptions
  input_type: sequence

input_type_info:
        sampling_strategy_if_longer: "uniform"
        max_length: "average"
        split_on: " "
        min_freq: 2
        tokenizer: "basic_english"
        tokenizer_language: "en"

model_config:
```

(continues on next page)

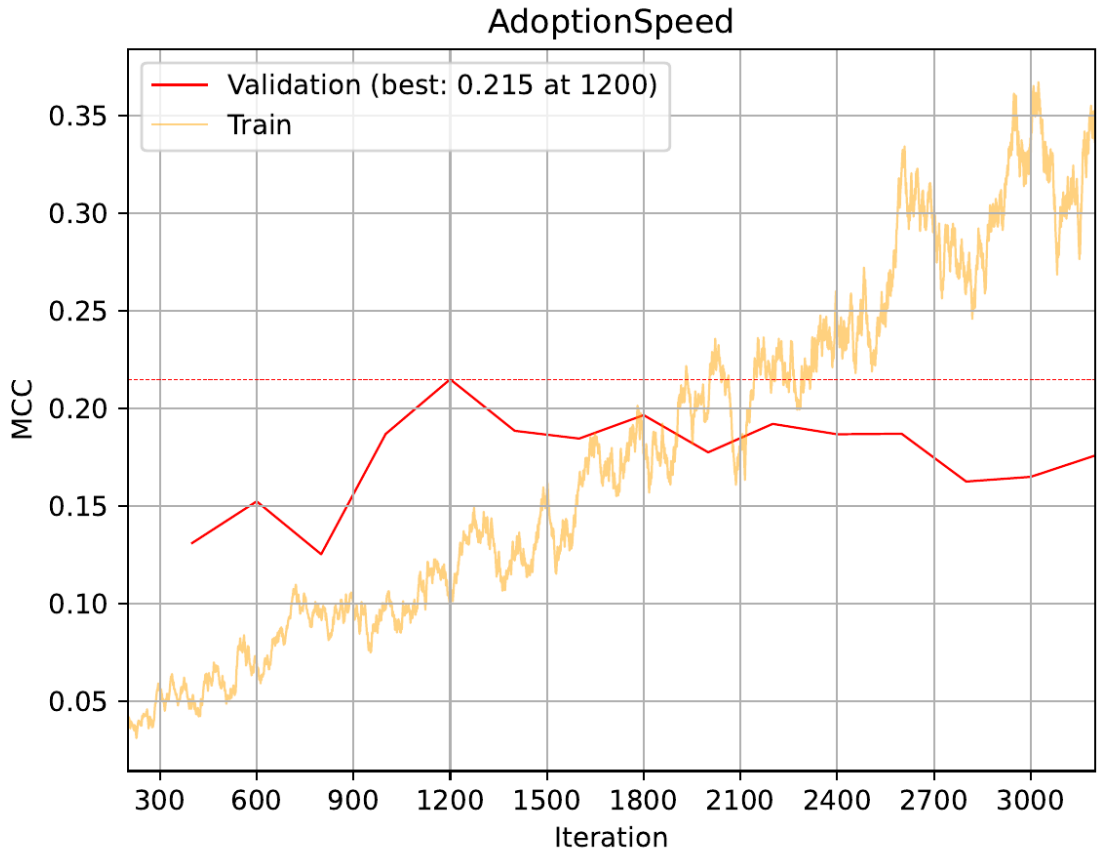
(continued from previous page)

```
      model_type: sequence-default
      embedding_dim: 64
      position: embed
      pool: avg
      model_init_config:
        num_heads: 4
        dropout: 0.2
```

Then to train, we simply include that configuration file under the `--input_configs` parameter:

```
eirtrain \
--global_configs eir_tutorials/a_using_eir/07_multimodal_tutorial/conf/07_globals.yaml \
--input_configs eir_tutorials/a_using_eir/07_multimodal_tutorial/conf/07_input_tabular.
↪yaml eir_tutorials/a_using_eir/07_multimodal_tutorial/conf/07_input_description.yaml \
--fusion_configs eir_tutorials/a_using_eir/07_multimodal_tutorial/conf/07_fusion.yaml \
--output_configs eir_tutorials/a_using_eir/07_multimodal_tutorial/conf/07_output.yaml \
--07_globals.output_folder=eir_tutorials/tutorial_runs/a_using_eir/tutorial_07b_
↪multimodal_tabular_description
```

Now, when training, we get the following training curve:

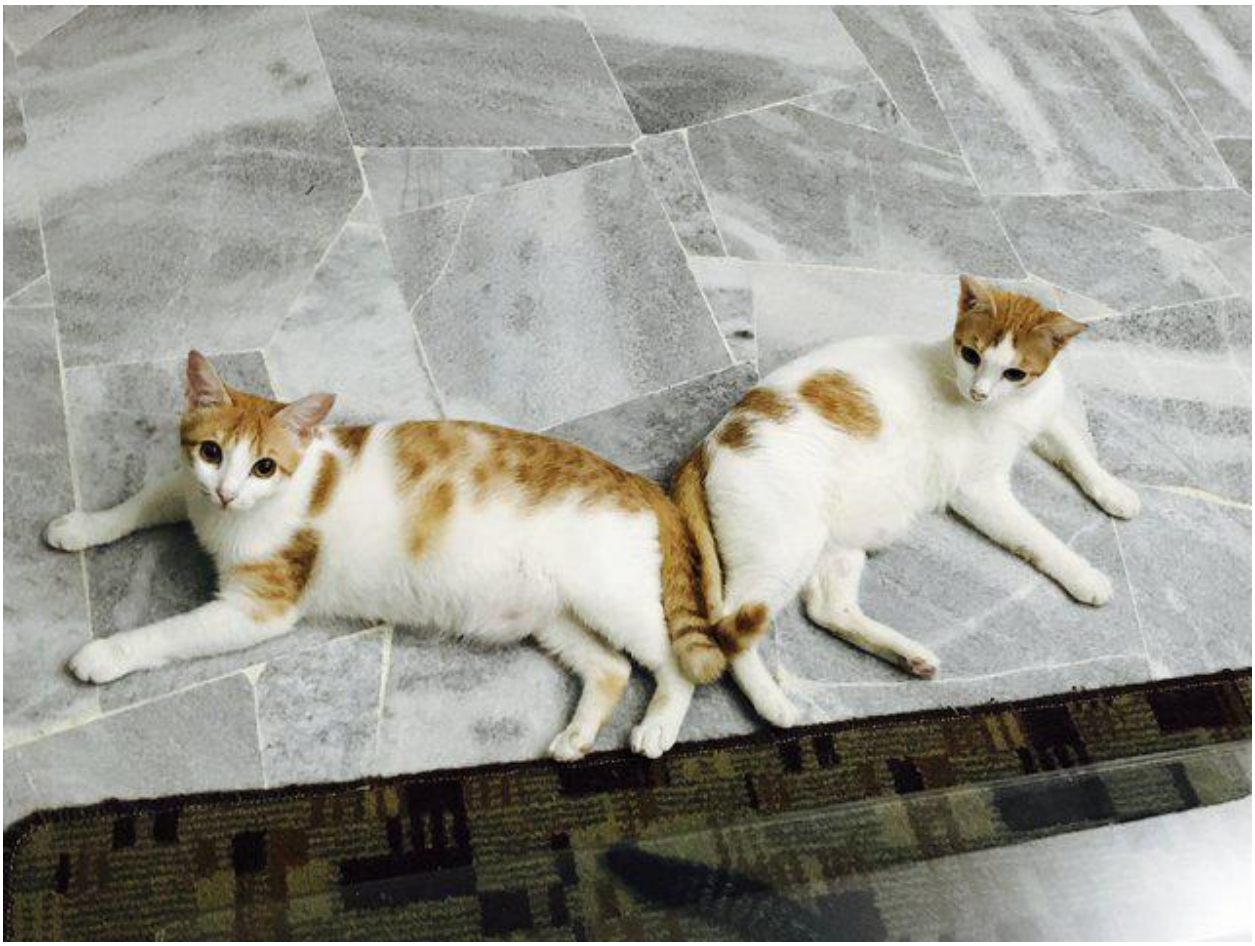


So here we can see that the model seems to perform slightly better when trained on both tabular and text data. We also start to see possible signs of overfitting, as the training curve starts to diverge from the validation curve.

C - Tabular + Text + Image Data

Now, we will train the model on all three types of data: tabular, text, and image. The image data is stored in a folder, where each image is stored in a separate file.

As before, let's have a quick look at an example image:



Configuration file for the image data:

```yaml
input_info:
  input_source: eir_tutorials/a_using_eir/07_multimodal_tutorial/data/images
  input_name: cute_pet_images
  input_type: image

input_type_info:
  mixing_subtype: "cutmix"
  size:
    - 128

model_config:
  model_type: "resnet18"
  pretrained_model: True
  freeze_pretrained_model: True
```

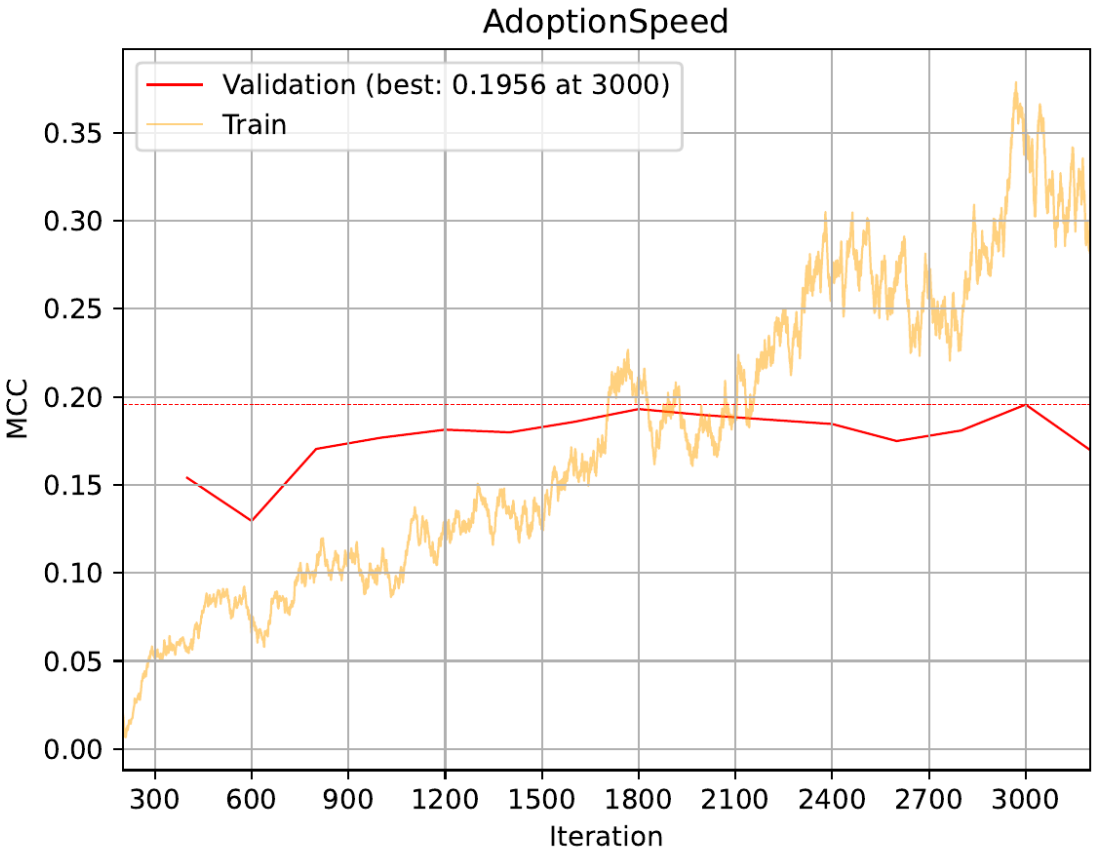
Note: Here we are using a pre-trained ResNet-18 model to extract the image features. We are using the `--pretrained_model` parameter to specify the that we want to use pre-trained weights. We are also using the `--freeze_pretrained_model` parameter to freeze the weights of the pre-trained model, so that they are not updated during training.

And then we can train the model on all three types of data:

```
eirtrain \
--global_configs eir_tutorials/a_using_eir/07_multimodal_tutorial/conf/07_globals.yaml \
--input_configs eir_tutorials/a_using_eir/07_multimodal_tutorial/conf/07_input_tabular.
↪yaml eir_tutorials/a_using_eir/07_multimodal_tutorial/conf/07_input_description.yaml␣
↪eir_tutorials/a_using_eir/07_multimodal_tutorial/conf/07_input_image.yaml \
--fusion_configs eir_tutorials/a_using_eir/07_multimodal_tutorial/conf/07_fusion.yaml \
--output_configs eir_tutorials/a_using_eir/07_multimodal_tutorial/conf/07_output.yaml \
--07_globals.output_folder=eir_tutorials/tutorial_runs/a_using_eir/tutorial_07c_
↪multimodal_tabular_description_image
```

Note: Here we are setting the `device` parameter to `cuda:0`, to train on the GPU. If you do not have a GPU, you can skip this parameter, or set it to `cpu`. Note that training on the CPU will likely be much slower, especially now that we are training on images as well.

When training, we get the following training curve:



So in this case, including the image data does not seem to improve the performance of the model further compared to the model trained on the tabular and text data. However, it does seem that the validation performance more quickly reaches peak performance when including the image data. It might be interesting to try training on the image data alone, to how much performance we can get from that. Furthermore, one could try unfreezing the pre-trained model, and see if that improves the performance. However, this tutorial is getting long enough already, so we will leave that as an exercise for those interested.

D - Serving

This section guides you through serving a multimodal model that combines tabular data, text descriptions, and images. We'll demonstrate how to interact with this served model using HTTP requests.

Starting the Web Service

To serve the multimodal model, use the following command:

```
eirserve --model-path [MODEL_PATH]
```

Replace *[MODEL_PATH]* with the actual path to your trained multimodal model. This command starts a web service that listens for incoming HTTP requests.

Example of the serving command:

```
eirserve \
--model-path eir_tutorials/tutorial_runs/a_using_eir/tutorial_07c_multimodal_tabular_
↪description_image/saved_models/tutorial_07c_multimodal_tabular_description_image_model_
↪2200_perf-average=0.4133.pt
```

Preparing and Sending Requests

Once the server is running, you can send requests containing tabular data, text descriptions, and image paths. Here's an example Python function to demonstrate this process:

```python
import requests
import json

def send_request(url: str, request_data: dict):
    response = requests.post(url, json=request_data)
    return response.json()

request_data = {
    "pets_tabular": {
        "Type": "Cat",
        "Name": "Nibble",
        "Age": 1.0,
        "Breed1": "Tabby",
        ...
    },
    "pet_descriptions": "A super cute tabby cat!!!",
    "cute_pet_images": "path/to/image.jpg"
}

response = send_request('http://localhost:8000/predict', request_data)
print(response)
```

Analyzing Responses

After sending requests to the served model, you will receive responses that provide a prediction based on the combined data (tabular, description, and image).

Let's take a look at some example predictions made by the model:

Listing 49: predictions.json

```
[
    {
        "request": {
            "pets_tabular": {
                "Type": "Cat",
                "Name": "Nibble",
                "Age": 1.0,
                "Breed1": "Tabby",
                "Breed2": "0",
                "Gender": "Male",
```

(continues on next page)

(continued from previous page)

```
                "Color1": "Black",
                "Color2": "White",
                "Color3": "0",
                "MaturitySize": "Small",
                "FurLength": "Short",
                "Vaccinated": "No",
                "Dewormed": "No",
                "Sterilized": "No",
                "Health": "Healthy",
                "Quantity": 1.0,
                "Fee": "Free",
                "State": "Selangor",
                "VideoAmt": 0.0,
                "PhotoAmt": 1.0
            },
            "pet_descriptions": "A super cute tabby cat!!!",
            "cute_pet_images": "eir_tutorials/a_using_eir/07_multimodal_tutorial/data/
↪images/86e1089a3.jpg"
        },
        "response": {
            "result": {
                "pet_adoption": {
                    "AdoptionSpeed": {
                        "A: 0-7 Days": 0.5612660050392151,
                        "B: 8-30 Days": 0.2135147899389267,
                        "C: 31-90 Days": 0.10258349031209946,
                        "D: 100+ Days": 0.12263575941324234
                    }
                }
            }
        }
    },
    {
        "request": {
            "pets_tabular": {
                "Type": "Cat",
                "Name": "Nibble",
                "Age": 5.0,
                "Breed1": "Tabby",
                "Breed2": "0",
                "Gender": "Male",
                "Color1": "Black",
                "Color2": "White",
                "Color3": "0",
                "MaturitySize": "Small",
                "FurLength": "Short",
                "Vaccinated": "No",
                "Dewormed": "No",
                "Sterilized": "No",
                "Health": "Healthy",
                "Quantity": 1.0,
                "Fee": "Free",
```

(continues on next page)

(continued from previous page)

```
                "State": "Selangor",
                "VideoAmt": 0.0,
                "PhotoAmt": 1.0
            },
            "pet_descriptions": "A super cute tabby cat!!!",
            "cute_pet_images": "eir_tutorials/a_using_eir/07_multimodal_tutorial/data/
↪images/86e1089a3.jpg"
        },
        "response": {
            "result": {
                "pet_adoption": {
                    "AdoptionSpeed": {
                        "A: 0-7 Days": 0.5546148419380188,
                        "B: 8-30 Days": 0.21321046352386475,
                        "C: 31-90 Days": 0.10370028018951416,
                        "D: 100+ Days": 0.12847435474395752
                    }
                }
            }
        }
    },
    {
        "request": {
            "pets_tabular": {
                "Type": "Cat",
                "Name": "Nibble",
                "Age": 10.0,
                "Breed1": "Tabby",
                "Breed2": "0",
                "Gender": "Male",
                "Color1": "Black",
                "Color2": "White",
                "Color3": "0",
                "MaturitySize": "Small",
                "FurLength": "Short",
                "Vaccinated": "No",
                "Dewormed": "No",
                "Sterilized": "No",
                "Health": "Healthy",
                "Quantity": 1.0,
                "Fee": "Free",
                "State": "Selangor",
                "VideoAmt": 0.0,
                "PhotoAmt": 1.0
            },
            "pet_descriptions": "A super cute tabby cat!!!",
            "cute_pet_images": "eir_tutorials/a_using_eir/07_multimodal_tutorial/data/
↪images/86e1089a3.jpg"
        },
        "response": {
            "result": {
                "pet_adoption": {
```

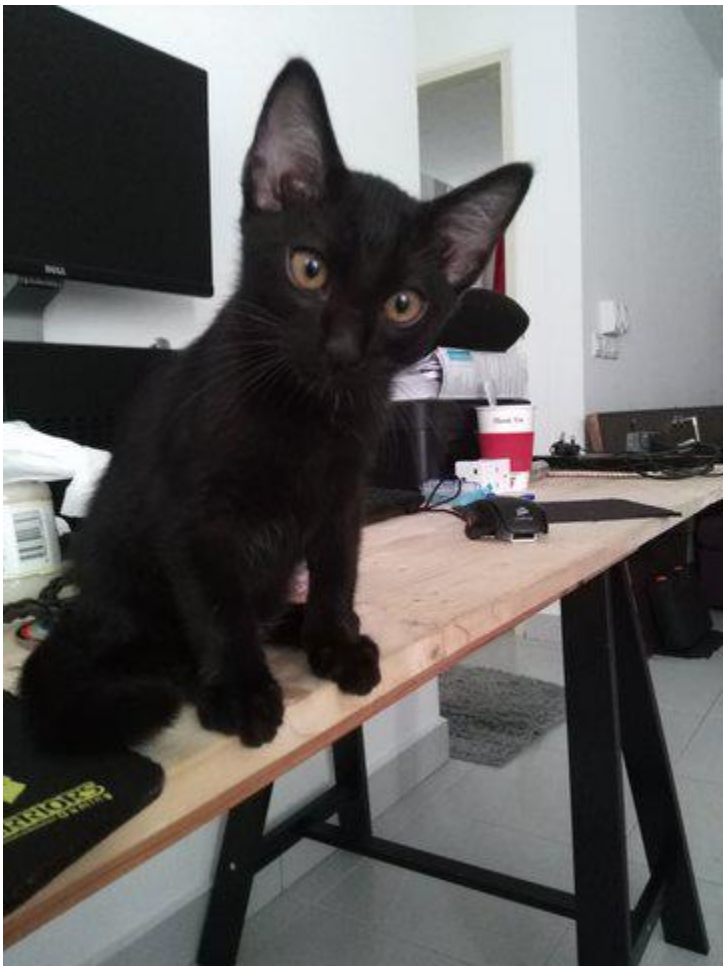
(continues on next page)

(continued from previous page)


```
                    "AdoptionSpeed": {
                        "A: 0-7 Days": 0.5458986759185791,
                        "B: 8-30 Days": 0.2128952294588089,
                        "C: 31-90 Days": 0.10505900532007217,
                        "D: 100+ Days": 0.13614711165428162
                    }
                }
            }
        }
    },
    {
        "request": {
            "pets_tabular": {
                "Type": "Cat",
                "Name": "Nibble",
                "Age": 3000.0,
                "Breed1": "Tabby",
                "Breed2": "0",
                "Gender": "Male",
                "Color1": "Black",
                "Color2": "White",
                "Color3": "0",
                "MaturitySize": "Small",
                "FurLength": "Short",
                "Vaccinated": "No",
                "Dewormed": "No",
                "Sterilized": "No",
                "Health": "Healthy",
                "Quantity": 1.0,
                "Fee": "Free",
                "State": "Selangor",
                "VideoAmt": 0.0,
                "PhotoAmt": 1.0
            },
            "pet_descriptions": "A super cute tabby cat!!!",
            "cute_pet_images": "eir_tutorials/a_using_eir/07_multimodal_tutorial/data/
↪images/86e1089a3.jpg"
        },
        "response": {
            "result": {
                "pet_adoption": {
                    "AdoptionSpeed": {
                        "A: 0-7 Days": 0.08935897797346115,
                        "B: 8-30 Days": 0.12761463224887848,
                        "C: 31-90 Days": 0.17728523910045624,
                        "D: 100+ Days": 0.6057411432266235
                    }
                }
            }
        }
    }
]
```

You can see that the inputs to the models are basically identical, except that we are varying the age of the pet. The general trend is that the older the pet, the longer it takes to be adopted, according to the model. This, unfortunately, is perhaps not surprising and is particularly visible when we increase the age to the extreme of 3000 months (250 years) – I mean, who would not want to adopt a 250 year old sage cat? :)

While not visible in the JSON above, here is the image used:



86e1089a3.jpg

That is it for the main part of the tutorial. I hope you enjoyed it! Below are a couple of appendixes with some additional experiments that might be interesting.

Appendix A - Adding a pre-trained text feature extractor

In this appendix, we will add a pre-trained text feature extractor in addition to the one we already have. We will be using a pre-trained Tiny BERT model (see (see *04 – Established Architectures and Pretrained Models* for more information). It is not certain whether this will improve the performance of the model, but the idea is more to showcase some functionalities of the framework.

Here is the configuration file for the pre-trained text feature extractor:

```
input_info:
  input_source: eir_tutorials/a_using_eir/07_multimodal_tutorial/data/descriptions.csv
```

(continues on next page)

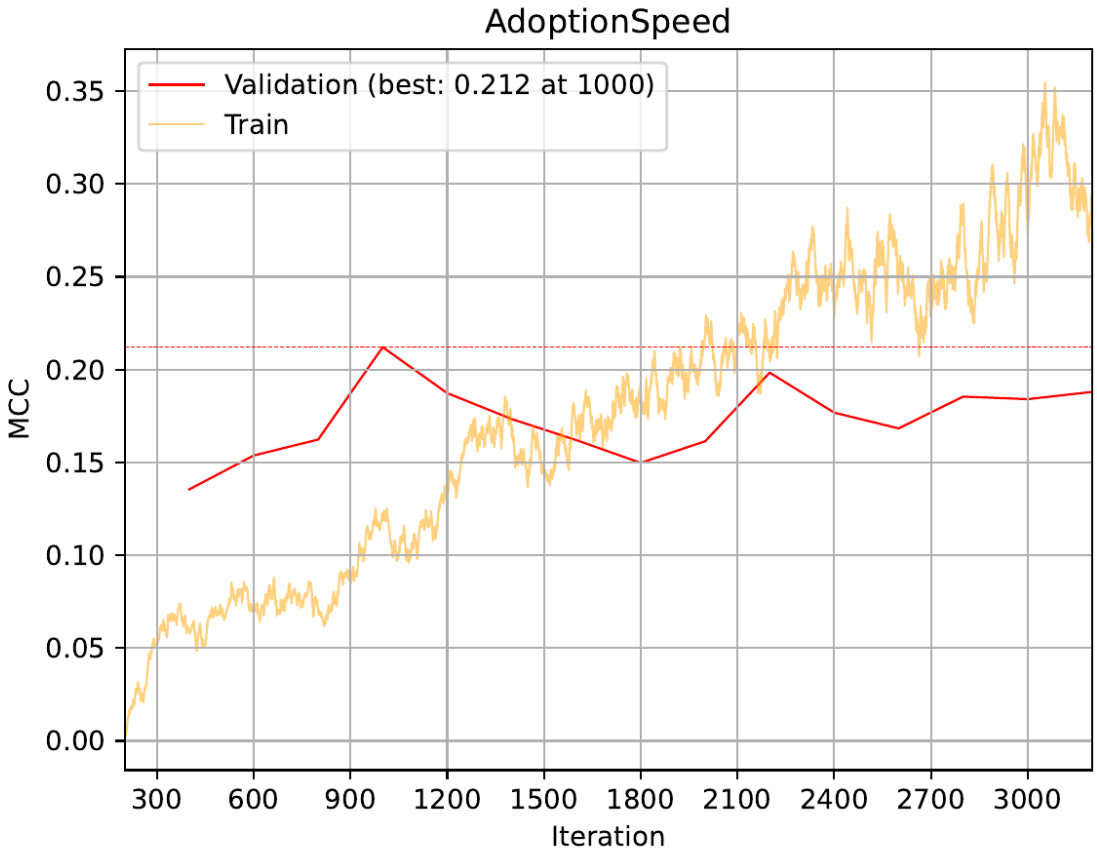
(continued from previous page)

```
  input_name: pet_descriptions_pretrained
  input_type: sequence

input_type_info:
  sampling_strategy_if_longer: "uniform"
  max_length: 64
  split_on: " "
  min_freq: 10

model_config:
    model_type: "prajjwal1/bert-tiny"
    pretrained_model: true
    freeze_pretrained_model: true
    position: embed
    pool: avg
```

The command:

```
eirtrain \
--global_configs eir_tutorials/a_using_eir/07_multimodal_tutorial/conf/07_globals.yaml \
--input_configs eir_tutorials/a_using_eir/07_multimodal_tutorial/conf/07_input_tabular.
↪yaml eir_tutorials/a_using_eir/07_multimodal_tutorial/conf/07_input_description.yaml␣
↪eir_tutorials/a_using_eir/07_multimodal_tutorial/conf/07_apx-a_input_description_
↪pretrained.yaml eir_tutorials/a_using_eir/07_multimodal_tutorial/conf/07_input_image.
↪yaml \
--fusion_configs eir_tutorials/a_using_eir/07_multimodal_tutorial/conf/07_fusion.yaml \
--output_configs eir_tutorials/a_using_eir/07_multimodal_tutorial/conf/07_output.yaml \
--07_globals.output_folder=eir_tutorials/tutorial_runs/a_using_eir/tutorial_07-apx-a_
↪multimodal_tabular_description_pretrained
```

The training curve:



So it seems that the pre-trained text feature extractor does not help, and likely we are even overfitting a bit more!

Appendix B - Multi-modal, multi-task learning

In this part, we will train the model to not only predict the adoption speed, but also the pet's age and number of pets in the image. For this, we have to modify the tabular input and output configurations:

```
input_info:
  input_source: eir_tutorials/a_using_eir/07_multimodal_tutorial/data/tabular.csv
  input_name: pets_tabular
  input_type: tabular

input_type_info:
  input_cat_columns:
          - Type
          - Breed1
          - Breed2
          - Gender
          - Color1
          - Color2
          - Color3
          - MaturitySize
          - State
```

(continues on next page)

(continued from previous page)

```
          - FurLength
          - Vaccinated
          - Dewormed
          - Sterilized
          - Health
          - Fee

  input_con_columns:
          - VideoAmt
          - PhotoAmt

model_config:
  model_type: tabular
```

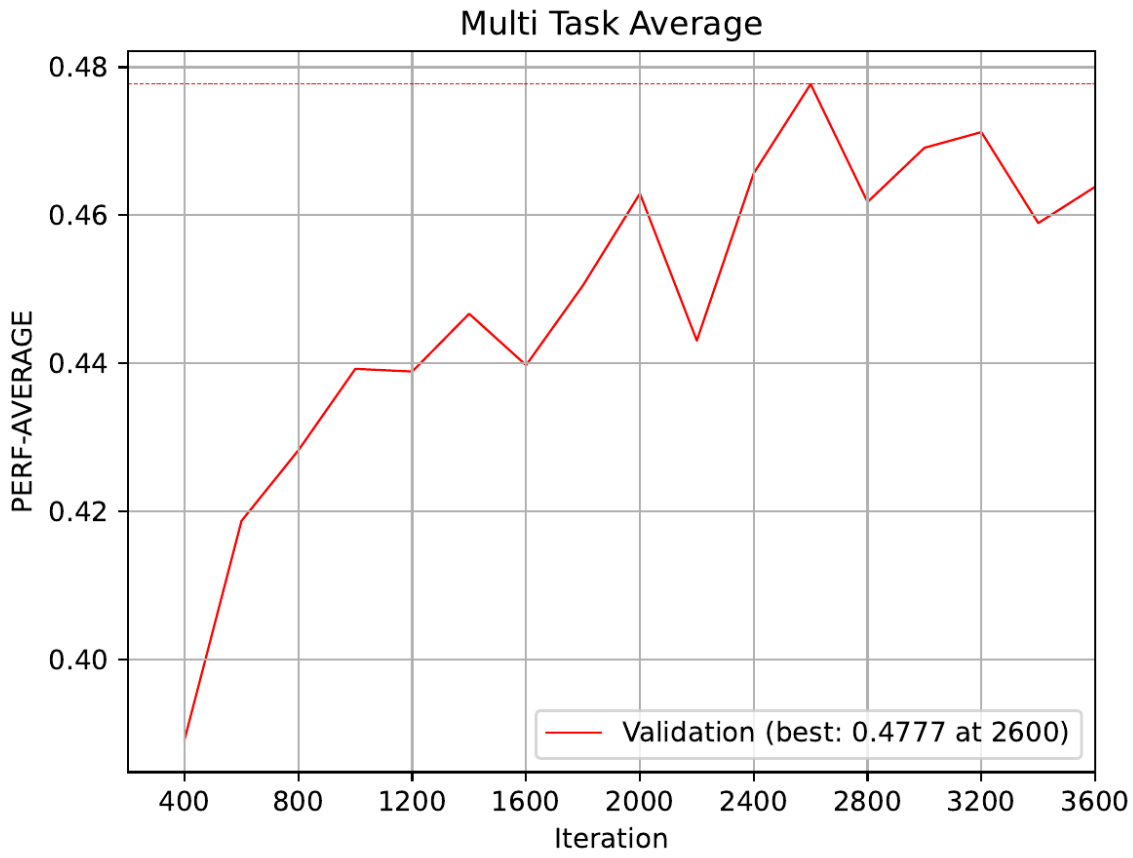
```
output_info:
  output_source: eir_tutorials/a_using_eir/07_multimodal_tutorial/data/tabular.csv
  output_name: pet_adoption
  output_type: tabular

output_type_info:
  target_cat_columns:
    - AdoptionSpeed
  target_con_columns:
    - Age
    - Quantity
  cat_label_smoothing: 0.1
```

Note that we have moved the features that we want to predict from the input configuration to the output configuration.

The command:

```
eirtrain \
--global_configs eir_tutorials/a_using_eir/07_multimodal_tutorial/conf/07_globals.yaml \
--input_configs eir_tutorials/a_using_eir/07_multimodal_tutorial/conf/07_apx-b_mt_input_
↪tabular.yaml eir_tutorials/a_using_eir/07_multimodal_tutorial/conf/07_input_
↪description.yaml eir_tutorials/a_using_eir/07_multimodal_tutorial/conf/07_apx-a_input_
↪description_pretrained.yaml eir_tutorials/a_using_eir/07_multimodal_tutorial/conf/07_
↪input_image.yaml \
--fusion_configs eir_tutorials/a_using_eir/07_multimodal_tutorial/conf/07_fusion.yaml \
--output_configs eir_tutorials/a_using_eir/07_multimodal_tutorial/conf/07_apx-b_mt_
↪output.yaml \
--07_globals.output_folder=eir_tutorials/tutorial_runs/a_using_eir/tutorial_07-apx-b_
↪multimodal_tabular_description_multi_task
```

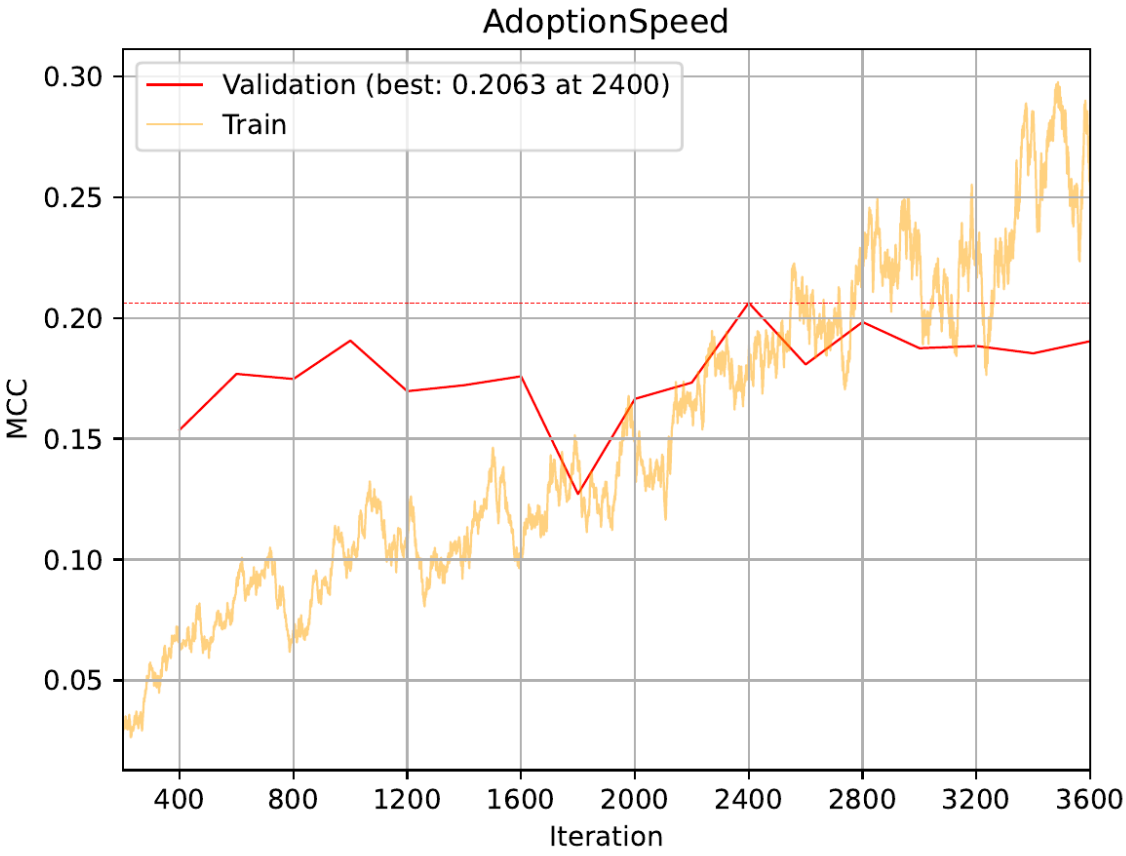
First we can have a look at the average performance:

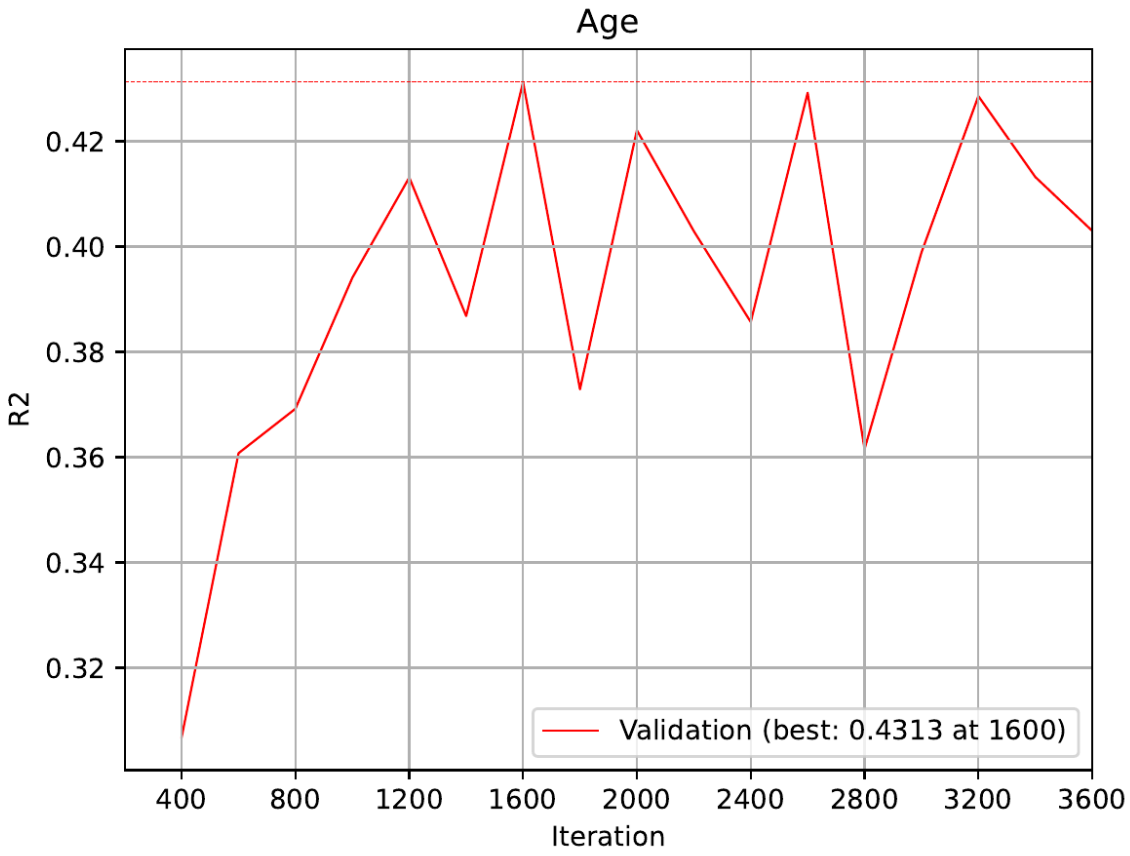


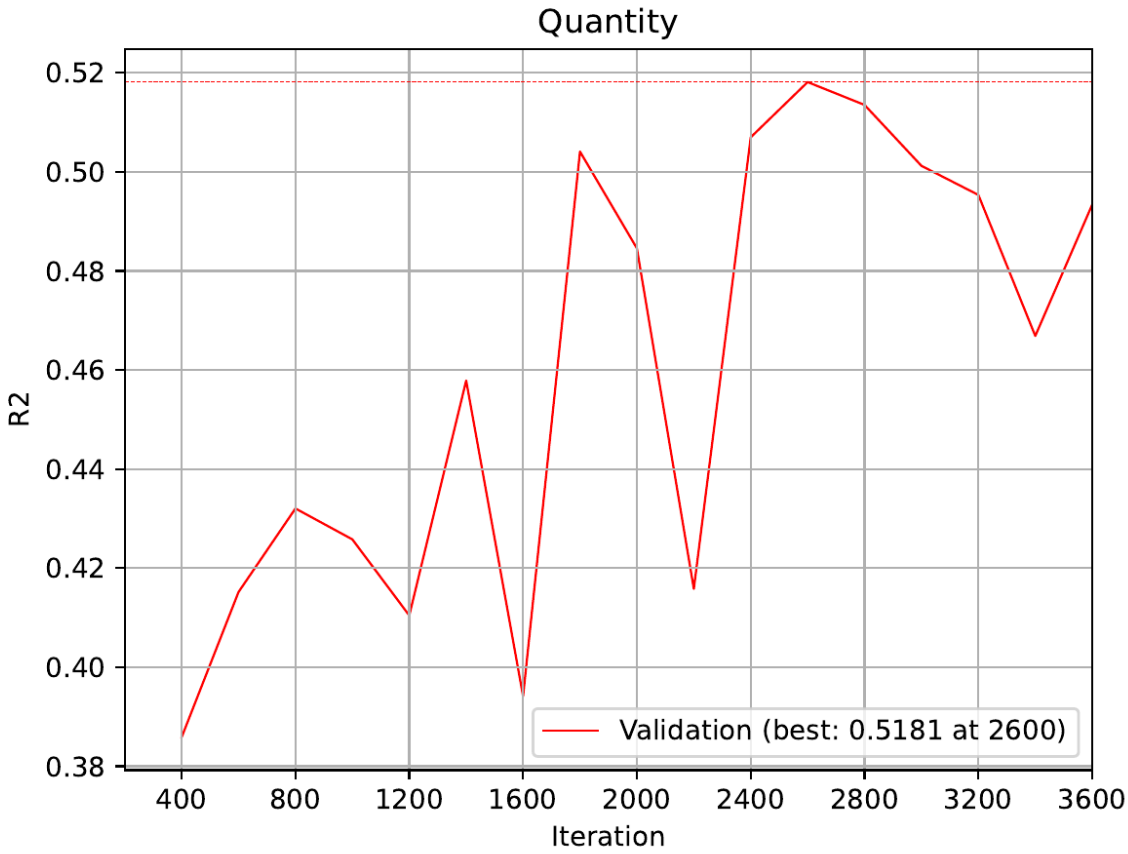
Note: The average performance by default is the average of the MCC, ROC-AUC and average precision (AP) for categorical targets and 1.0-LOSS, PCC, R2 for continuous targets.

So, since we are using different inputs and outputs in this task, we cannot compare directly to the previous results. However, we can see that the model seems to be able to learn to predict the 3 different targets fairly well.

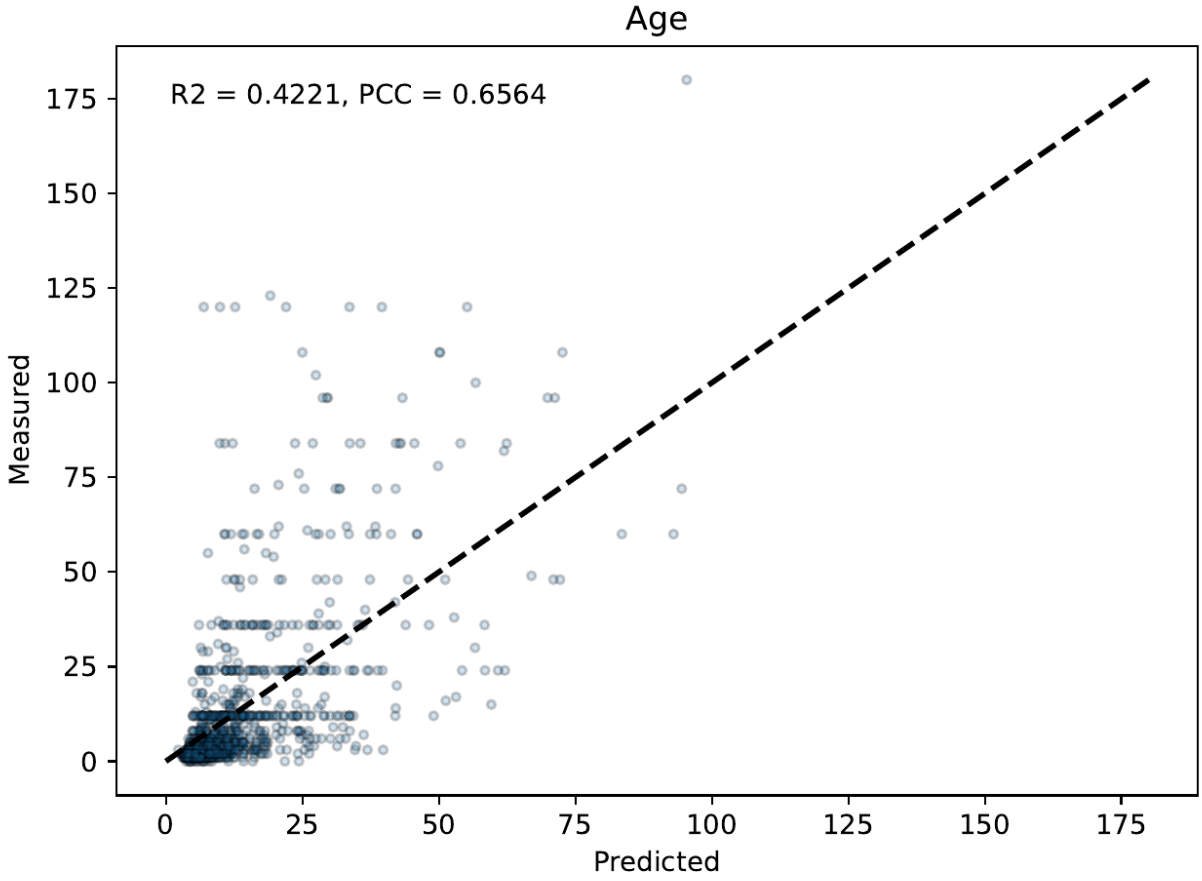
The training curves for the adoption speed, age and quantity:

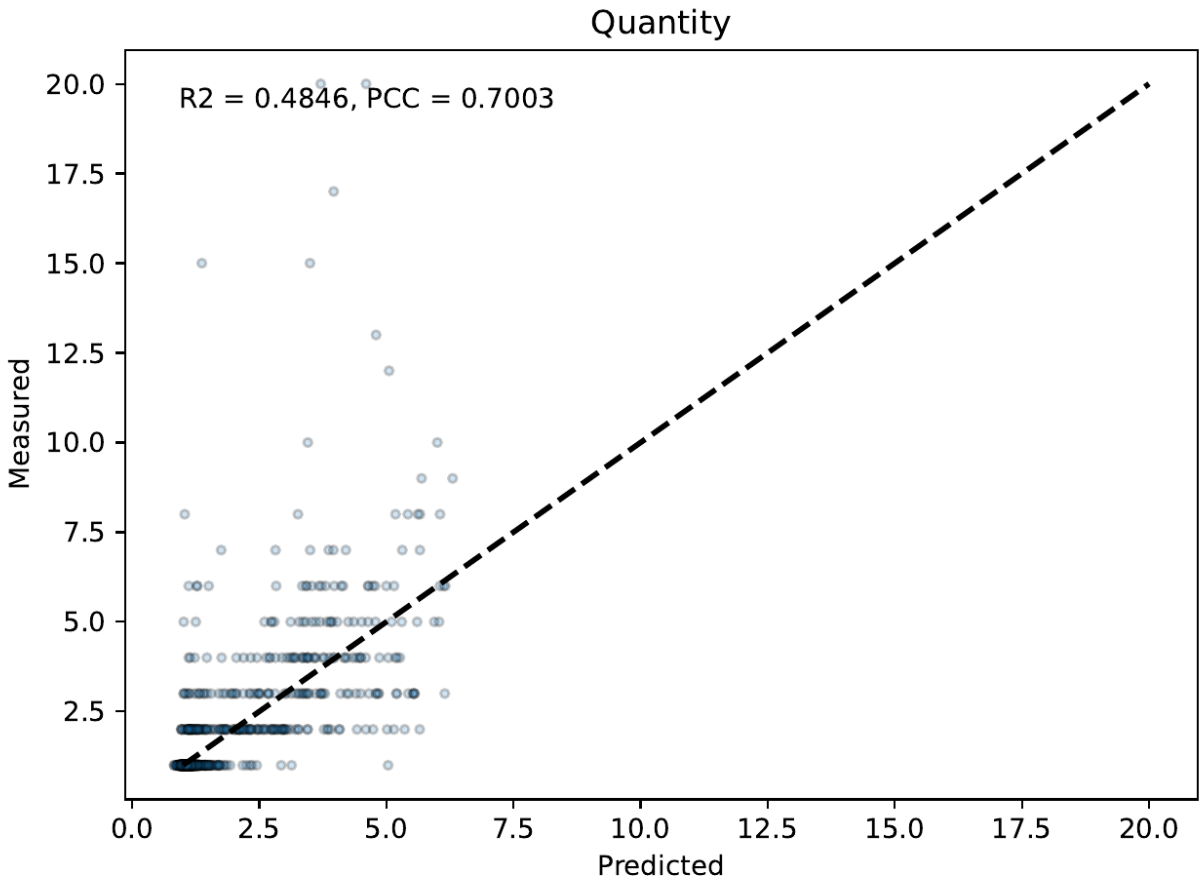

AdoptionSpeed
Validation (best: 0.2063 at 2400)
Train
MCC
0.30
0.25
0.20
0.15
0.10
0.05
400
800
1200
1600
2000
2400
2800
3200
3600
Iteration


Age
R2
0.42
0.40
0.38
0.36
0.34
0.32
400
800
1200
1600
2000
2400
2800
3200
3600
Iteration
Validation (best: 0.4313 at 1600)



Finally, we can also look at the regression predictions by checking the `samples` folder for the `Age` and `Quantity` targets. Here are a couple of examples:


Age
R2 = 0.4221, PCC = 0.6564
Measured
Predicted
0
25
50
75
100
125
150
175



So in the case of quantity, it is expected that the model gets some of the predictions wrong, since in our parsed data we are only using randomly chosen one image, but the original data includes multiple images (it can also be that it can learn some of this from the descriptions). However, the model seems to be able to predict the quantity fairly well, and same for the age.

2.1.8 08 – Training on arrays with CNN, LCL, and Transformer Models

In this tutorial, we will be looking at the built in support for training models on structured arrays in `EIR`. Here, structured refers to the arrays all having the same shape, and arrays refers to the fact that the data is stored in a numpy array. We will be using the same data as we did in *01 – Genotype Tutorial: Ancestry Prediction*, but instead treating them as general arrays instead of genotypes. Currently, the array functionality in `EIR` is built to handle 1, 2 and 3 dimensional arrays. As in the genotype tutorial, we will be using data processed from the Human Origins dataset. To download the data and configurations for this part of the tutorial, use this link.

A - Data

After downloading the data, the folder structure should look like this:

```
eir_tutorials/a_using_eir/08_array_tutorial/
├── conf
│   ├── globals.yaml
│   ├── input_1d_cnn.yaml
│   ├── input_1d_lcl.yaml
│   ├── input_1d_transformer.yaml
│   ├── input_2d_cnn.yaml
│   ├── input_2d_lcl.yaml
│   ├── input_2d_transformer.yaml
│   ├── input_3d_cnn.yaml
│   ├── input_3d_lcl.yaml
│   ├── input_3d_transformer.yaml
│   └── outputs.yaml
└── data
    ├── processed_sample_data
    │   ├── arrays_1d
    │   ├── arrays_2d
    │   ├── arrays_3d
    │   └── human_origins_labels.csv
    └── processed_sample_data.zip
```

Besides the configurations, there are 3 folders there storing the genotype arrays, with each folder corresponding to a different dimensionality (although all the versions are generated from the same base data). The arrays in the 1D folder encodes the reference, heterozygous, alternative and missing genotypes as 0, 1, 2 and 3 respectively. The 2D arrays encode the same information, as a one-hot encoded array. Finally, the 3D arrays contain the same one-hot encoding as the 2D case, but with a flipped copy of the array as the second channel. This is all perhaps a bit redundant, but it's just for this tutorial.

B - Training

Here are the configurations for the 1D case:

Listing 50: globals.yaml

```yaml
output_folder: eir_tutorials/tutorial_runs/a_using_eir/tutorial_08_run
checkpoint_interval: 200
sample_interval: 200
n_epochs: 20
memory_dataset: True
device: "mps"
```

Listing 51: input_1d_cnn.yaml

```yaml
input_info:
  input_source: eir_tutorials/a_using_eir/08_array_tutorial/data/processed_sample_data/
↪arrays_1d
  input_name: genotype_as_array
  input_type: array
```

(continues on next page)

(continued from previous page)

```
model_config:
  model_type: cnn
  model_init_config:
    kernel_height: 1
    kernel_width: 4
```

Listing 52: outputs.yaml

```
output_info:
  output_name: ancestry_output
  output_source: eir_tutorials/a_using_eir/08_array_tutorial/data/processed_sample_data/
↪human_origins_labels.csv
  output_type: tabular
output_type_info:
  target_cat_columns:
    - Origin
```

Important: The CNN functionality for arrays is currently experimental, and might change in later versions of `EIR`.

We will be training both the CNN, LCL (locally-connected-layers) and transformer models, here is an example configuration for the LCL model:

Listing 53: input_1d_lcl.yaml

```
input_info:
  input_source: eir_tutorials/a_using_eir/08_array_tutorial/data/processed_sample_data/
↪arrays_1d
  input_name: genotype_as_array
  input_type: array

model_config:
  model_type: lcl
  model_init_config:
    kernel_width: 4
    first_kernel_expansion: 1
```

Important: While there is a lot of similarity between training the LCL models here and the genotype models in *01 – Genotype Tutorial: Ancestry Prediction*, there are some important differences. The most important is how the LC layers are applied over the input dimensions. Considering the 2D case, where we have one-hot encoded arrays with shape `(4, n_SNPs)`. In the genotype case, the `kernel_width` parameter in the LC layer will be applied in colum-order, meaning a width of 8 will cover the first 2 SNPs. In the array case, the `kernel_width` parameter is applied in row-order, meaning a width of 8 will cover the first row of the first 8 SNPs.

Here is an example configuration for the transformer model:

Listing 54: input_1d_transformer.yaml

```
input_info:
  input_source: eir_tutorials/a_using_eir/08_array_tutorial/data/processed_sample_data/
↪arrays_1d
```

(continues on next page)

(continued from previous page)

```
  input_name: genotype_as_array
  input_type: array

model_config:
  model_type: transformer
  model_init_config:
    embedding_dim: 32
    patch_size:
      - 1
      - 1
      - 4
```

Important: For the transformer models, the `patch_size` parameter is used to determine the size of the patches that are fed into the transformer. The total number of input elements must be divisible by the patch size. The order follows the same convention as PyTorch, meaning CxHxW. For 1D and 2D inputs, use a size of 1 for the redundant dimensions when specifying the patch size.

As usual, we can run the following command to train for the CNN, LCL and Tranformer cases:

```
eirtrain \
--global_configs eir_tutorials/a_using_eir/08_array_tutorial/conf/globals.yaml \
--input_configs eir_tutorials/a_using_eir/08_array_tutorial/conf/input_1d_cnn.yaml \
--output_configs eir_tutorials/a_using_eir/08_array_tutorial/conf/outputs.yaml \
--globals.output_folder=eir_tutorials/tutorial_runs/a_using_eir/tutorial_08_run_cnn-1d
```

```
eirtrain \
--global_configs eir_tutorials/a_using_eir/08_array_tutorial/conf/globals.yaml \
--input_configs eir_tutorials/a_using_eir/08_array_tutorial/conf/input_1d_lcl.yaml \
--output_configs eir_tutorials/a_using_eir/08_array_tutorial/conf/outputs.yaml \
--globals.output_folder=eir_tutorials/tutorial_runs/a_using_eir/tutorial_08_run_lcl-1d
```

```
eirtrain \
--global_configs eir_tutorials/a_using_eir/08_array_tutorial/conf/globals.yaml \
--input_configs eir_tutorials/a_using_eir/08_array_tutorial/conf/input_1d_transformer.
↪yaml \
--output_configs eir_tutorials/a_using_eir/08_array_tutorial/conf/outputs.yaml \
--globals.output_folder=eir_tutorials/tutorial_runs/a_using_eir/tutorial_08_run_
↪transformer-1d
```

For the 2D and 3D cases, here are the configurations:

Listing 55: input_2d_cnn.yaml

```
input_info:
  input_source: eir_tutorials/a_using_eir/08_array_tutorial/data/processed_sample_data/
↪arrays_2d
  input_name: genotype_as_array
  input_type: array

model_config:
  model_type: cnn
```

(continues on next page)

(continued from previous page)

```
  model_init_config:
    kernel_height: 1
    first_kernel_expansion_height: 4
    kernel_width: 4
```

Listing 56: input_2d_lcl.yaml

```
input_info:
  input_source: eir_tutorials/a_using_eir/08_array_tutorial/data/processed_sample_data/
↪arrays_2d
  input_name: genotype_as_array
  input_type: array

model_config:
  model_type: lcl
  model_init_config:
    kernel_width: 8
    first_kernel_expansion: 1
```

Listing 57: input_2d_transformer.yaml

```
input_info:
  input_source: eir_tutorials/a_using_eir/08_array_tutorial/data/processed_sample_data/
↪arrays_2d
  input_name: genotype_as_array
  input_type: array

model_config:
  model_type: transformer
  model_init_config:
    embedding_dim: 32
    patch_size:
      - 1
      - 4
      - 4
```

Listing 58: input_3d_cnn.yaml

```
input_info:
  input_source: eir_tutorials/a_using_eir/08_array_tutorial/data/processed_sample_data/
↪arrays_3d
  input_name: genotype_as_array
  input_type: array

model_config:
  model_type: cnn
  model_init_config:
    kernel_height: 1
    first_kernel_expansion_height: 4
    kernel_width: 4
```

Listing 59: input_3d_lcl.yaml

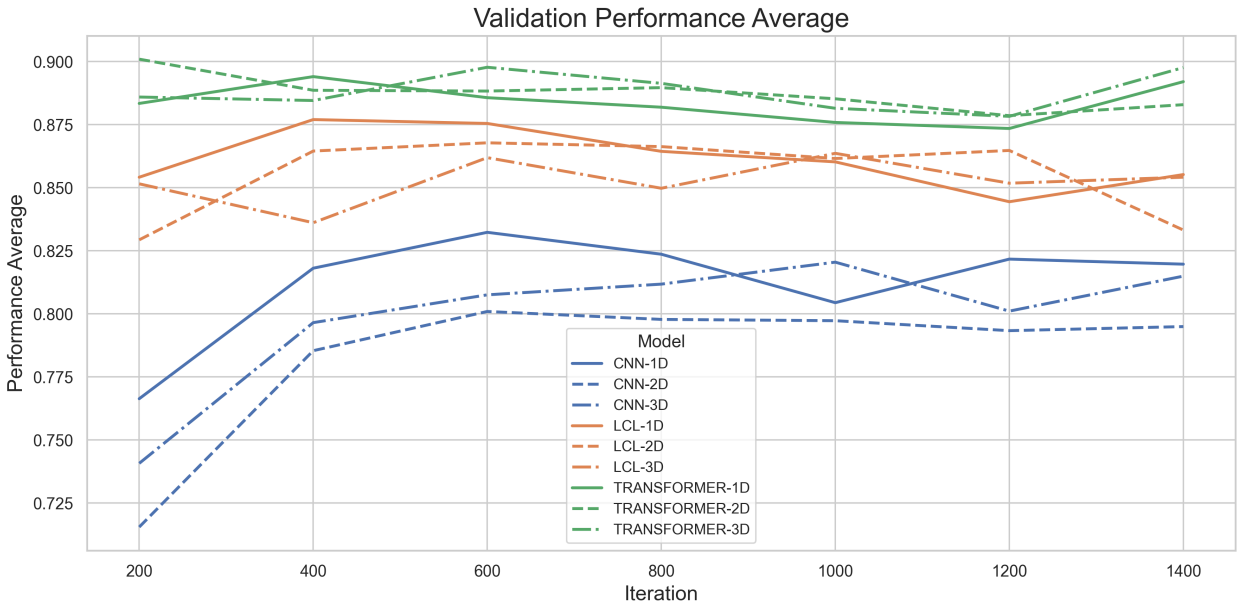
```
input_info:
  input_source: eir_tutorials/a_using_eir/08_array_tutorial/data/processed_sample_data/
↪arrays_3d
  input_name: genotype_as_array
  input_type: array

model_config:
  model_type: lcl
  model_init_config:
    kernel_width: 16
    first_kernel_expansion: 1
```

Listing 60: input_3d_transformer.yaml

```
input_info:
  input_source: eir_tutorials/a_using_eir/08_array_tutorial/data/processed_sample_data/
↪arrays_3d
  input_name: genotype_as_array
  input_type: array

model_config:
  model_type: transformer
  model_init_config:
    embedding_dim: 32
    patch_size:
      - 2
      - 4
      - 4
```

Note: For the CNN model, you might be wondering about the `kernel_height` and `first_kernel_expansion_height` parameters. The `kernel_height` parameter refers to the "base" kernel height that is used throughout the model. In the 2D case, we are working with 4xN arrays, and want the kernels in the first layer to be able to cover the entire height of the array. Successive kernels will then operate on a height of 1. Coming back to the parameters, the `first_kernel_expansion_height=4` is indicating that the first layer should have a kernel height of 4, and the `kernel_height=1` is indicating that the successive layers should have a kernel height of 1.

After training, I got the following validation results:



So, here it seems that the transformer models and LCL models are performing a bit better than the CNN models, with the transformers being the best. However, we are training for a relatively short time, and one might get better results by e.g. increasing the number of filters in the CNN case.

C - Serving

In this final section, we demonstrate serving our trained model for 3D array data as a web service and interacting with it using HTTP requests.

Starting the Web Service

To serve the model, use the following command:

```
eirserve --model-path [MODEL_PATH]
```

Replace *[MODEL_PATH]* with the actual path to your trained model. This command initiates a web service that listens for incoming requests.

Here is an example of the command:

```
eirserve \
--model-path eir_tutorials/tutorial_runs/a_using_eir/tutorial_08_run_transformer-3d/
↪saved_models/tutorial_08_run_transformer-3d_model_600_perf-average=0.8977.pt
```

Sending Requests

With the server running, we can now send requests for 3D array data. The data is encoded in base64 before sending.

Here's an example Python function demonstrating this process:

```python
import requests
import numpy as np
import base64

def encode_array_to_base64(file_path: str) -> str:
    array_np = np.load(file_path)
    array_bytes = array_np.tobytes()
    return base64.b64encode(array_bytes).decode('utf-8')

def send_request(url: str, payload: dict):
    response = requests.post(url, json=payload)
    return response.json()

payload = {
    "genotype_as_array": encode_array_to_base64("path/to/array_file.npy")
}

response = send_request('http://localhost:8000/predict', payload)
print(response)
```

Analyzing Responses

After sending requests to the served model, the responses might look something like this:

Listing 61: predictions.json

```
[
    {
        "request": {
            "genotype_as_array": "eir_tutorials/a_using_eir/08_array_tutorial/data/
↪processed_sample_data/arrays_3d/A374.npy"
        },
        "response": {}
    },
    {
        "request": {
            "genotype_as_array": "eir_tutorials/a_using_eir/08_array_tutorial/data/
↪processed_sample_data/arrays_3d/Ayodo_468C.npy"
        },
        "response": {}
    },
    {
        "request": {
            "genotype_as_array": "eir_tutorials/a_using_eir/08_array_tutorial/data/
↪processed_sample_data/arrays_3d/NOR146.npy"
        },
```

(continues on next page)

(continued from previous page)

```
        "response": {}
    }
]
```

If you made it this far, thanks for reading! I hope you found this tutorial useful.

2.2 Sequence Generation

2.2.1 01 – Sequence Generation: Generating Movie Reviews

In this tutorial, we will look into the built-in support in *EIR* for sequence generation tasks (similar to what GPT does). Sequences can represent various types of data such as time series, sentences, genetic information, and more. This technique allows us to generate new, meaningful sequences based on patterns learned from the training data.

We will be using the same dataset we used in the *03 – Sequence Tutorial: Movie Reviews and Peptides*: the IMDB reviews dataset. However, instead of classifying the reviews, our goal this time will be to generate new movie reviews.

Note: This tutorial assumes you are familiar with the basics of *EIR*, and have gone through the *01 – Genotype Tutorial: Ancestry Prediction* and the *03 – Sequence Tutorial: Movie Reviews and Peptides*. Not required, but recommended.

A - Data

As in the *03 – Sequence Tutorial: Movie Reviews and Peptides*, we will be using the IMDB reviews dataset. See here for more information about the data. To download the data, use this link.

After downloading the data, the folder structure should look like this (we will look at the configs in a bit):

```
eir_tutorials/c_sequence_output/01_sequence_generation
├── conf
│   ├── fusion.yaml
│   ├── globals.yaml
│   ├── output.yaml
│   ├── output_bpe.yaml
│   └── output_test.yaml
└── data
    └── IMDB
        ├── IMDB_Reviews
        ├── conf
        ├── imdb.vocab
        └── imdb_labels.csv
```

B - Training

Training is almost the same as when doing supervised learning, with a couple of changes in our configurations. The biggest difference is perhaps that when doing pure sequence generation tasks (i.e., there are no auxiliary inputs), we do not need to specify an input configuration, we only have a global, fusion and output config:

The global config is does not introduce any new parameters:

Listing 62: globals.yaml

```yaml
output_folder: eir_tutorials/tutorial_runs/c_sequence_output/01_sequence_generation
valid_size: 500
n_saved_models: 1
checkpoint_interval: 500
sample_interval: 500
memory_dataset: true
n_epochs: 100
batch_size: 256
device: "mps"
```

Note: Above I am using the `mps` device for training, which is in some Macs. If you are using a different device, you can change it to `cpu` or e.g., `cuda:0`.

When we are doing only sequence generation (i.e., that is the only task), the only supported fusion module is "pass-through" currently, this is because each sequence generation head performs its own fusion. Therefore, customizing the fusion module with settings we have seen before (e.g., setting the model type to "mlp-residual") would not have any effect. However, if you are doing sequence generation as one of multiple tasks, where at least one of the tasks is a supervised prediction, you can customize the fusion module. However, it will only be used for the supervised task, the sequence generation task will still use the "pass-through" fusion, which is automatically added.

Listing 63: fusion.yaml

```yaml
model_type: "pass-through"
```

Now for the output, the structure is very similar to what we have seen before, but with a couple of changes. The first difference is the `output_type`, here instead of `tabular`, we set it to `sequence`. The other difference is that we now have a `sampling_config`, specific to sequence generation. This allows us to configure various parameters related to the sampling process during training, where sequences are generated every `sample_interval`.

Another thing of note is that here we are training a character-level model, as `split_on` is set to `""`.

Listing 64: output.yaml

```yaml
output_info:
  output_source: eir_tutorials/c_sequence_output/01_sequence_generation/data/IMDB/IMDB_
↪Reviews
  output_name: imdb_output
  output_type: sequence

output_type_info:
  max_length: 64
  split_on: ""
  sampling_strategy_if_longer: "uniform"
  min_freq: 1
```

(continues on next page)

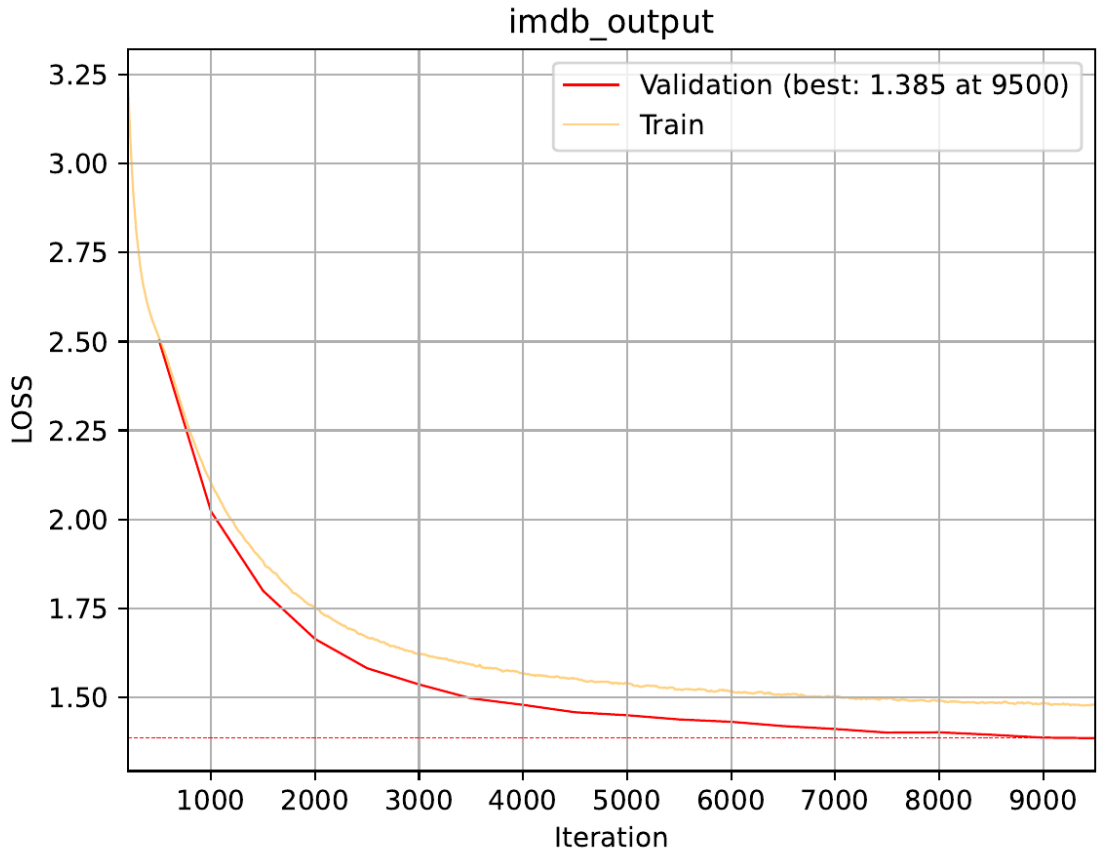
(continued from previous page)

```
model_config:
  embedding_dim: 64
  model_init_config:
    num_layers: 6

sampling_config:
  generated_sequence_length: 128
  n_eval_inputs: 1

  manual_inputs:
    - imdb_output: "This movie is the most"

    - imdb_output: "Steven"
```

After setting up the configs, training is similar to what we have seen before:

```
eirtrain \
--global_configs eir_tutorials/c_sequence_output/01_sequence_generation/conf/globals.
↪yaml \
--fusion_configs eir_tutorials/c_sequence_output/01_sequence_generation/conf/fusion.yaml␣
↪\
--output_configs eir_tutorials/c_sequence_output/01_sequence_generation/conf/output_bpe.
↪yaml \
--globals.output_folder=eir_tutorials/tutorial_runs/c_sequence_output/01_sequence_
↪generation_bpe
```

I got the following results:



However, the most interesting part is not the training curve, but the generated sequences. If we look in the familiar `samples` folder, we can see the generated sequences. At iteration 500, they are mostly gibberish:

Listing 65: Auto-generated sequence at iteration 500

```
he anos e atth sthas singulit, tre is ame wo heth chesolowre ad isse woffoutrtong sond␣
↪ton ifieers ant ar d whery, chid e e her
```

Listing 66: Manually sequence at iteration 500 with custom prompt

```
This movie is the mostove t ove arovetally ar of wolid t aso s malotrindis, mans d,␣
↪cthak. gecthestin Alesean once avectiet trth
```

However, at iteration 9500, we can see that the model is starting to generate more meaningful sequences:

Listing 67: Auto-generated sequence at iteration 9500

```
ng happening out the film is not the class of the acting of the film like this film,␣
↪everything for my favourite effects and if
```

Listing 68: Manually sequence at iteration 9500 with custom prompt

```
This movie is the most action cast who did watch a great dialogue, she gets a poor story␣
↪better movie into like this comprofessi
```

C - Prediction: Creating new sequences with a trained model

Now that we have trained our model, we can use it to generate new sequences. Similarly to the process when we are doing supervised prediction, we use the `eirpredict` command, with a couple of minor changes now that we are doing sequence generation.

The first change can be seen in the output configuration. Here we have a file called `output_test.yaml`, which is similar to the `output.yaml` we used for training, but notice the change in `output_source`:

Listing 69: output_test.yaml

```yaml
output_info:
  output_source: null
  output_name: imdb_output
  output_type: sequence

output_type_info:
  max_length: 64
  split_on: ""
  sampling_strategy_if_longer: "uniform"
  min_freq: 1

model_config:
  embedding_dim: 64
  model_init_config:
    num_layers: 6

sampling_config:
  generated_sequence_length: 64
  n_eval_inputs: 10

  manual_inputs:
    - imdb_output: "This movie is the most"

    - imdb_output: "Steven"
```

Here we have `null` for the `output_source`, which is because we do not have any concrete inputs for the sequence generation task. Now, to control the sequence generation prediction functionality, we are using the `sampling_config` in the configuration above, which allows to e.g. specify the generated sequence length, now many sequences to generate from an empty prompt (`n_eval_inputs`) and finally generate sequences from custom prompts (`manual_inputs`).

Now we execute our `eirpredict` command:

```
eirpredict \
--global_configs eir_tutorials/c_sequence_output/01_sequence_generation/conf/globals.
↪yaml \
--fusion_configs eir_tutorials/c_sequence_output/01_sequence_generation/conf/fusion.yaml␣
↪\
--output_configs eir_tutorials/c_sequence_output/01_sequence_generation/conf/output_test.
↪yaml \
--model_path eir_tutorials/tutorial_runs/c_sequence_output/01_sequence_generation/saved_
↪models/01_sequence_generation_model_9500_perf-average=-0.3847.pt \
--output_folder eir_tutorials/tutorial_runs/c_sequence_output/01_sequence_generation/
↪test_results
```

This will save our results under the paths specified in the `output_folder` parameter, containing both the auto-generated and manually generated sequences.

Here is an example of an auto-generated sequence:

Listing 70: Prediction auto-generated sequence 1

```
done at a hold feeling that doesn't love awful. But it was so a
```

And here are the manually generated sequences with our custom prompts:

Listing 71: Prediction manually generated sequence 1

```
This movie is the most camera character with an acting fun thing
```

Listing 72: Prediction manually generated sequence 2

```
Steven Alimon from this about the world of emotions and they bot
```

While our generated reviews are far from realistic, they do show that the model is learning to generate sequences that are somewhat meaningful.

E - Sequence Generation with BPE Tokenization

Now that we have seen how to do sequence generation with a character-level model, let's see how we can do it with a token-level model. This time, we will use the IMDB dataset, but with an implementation of BPE (Byte Pair Encoding) tokenization.

BPE, as detailed in this paper, is a sub-word tokenization method that progressively learns the most common sequences of characters (or bytes) to form an efficient set of tokens.

As we'll see, using BPE tokenization allows us to generate longer sequences than with the character model.

To use it, a couple of changes are needed in the output configuration:

Listing 73: output_bpe.yaml

```yaml
output_info:
  output_source: eir_tutorials/c_sequence_output/01_sequence_generation/data/IMDB/IMDB_
↪Reviews
  output_name: imdb_output
  output_type: sequence

output_type_info:
  max_length: 32
  split_on: null
  tokenizer: "bpe"
  adaptive_tokenizer_max_vocab_size: 1024
  sampling_strategy_if_longer: "uniform"
  min_freq: 1

model_config:
  embedding_dim: 64
  model_init_config:
    num_layers: 2
```

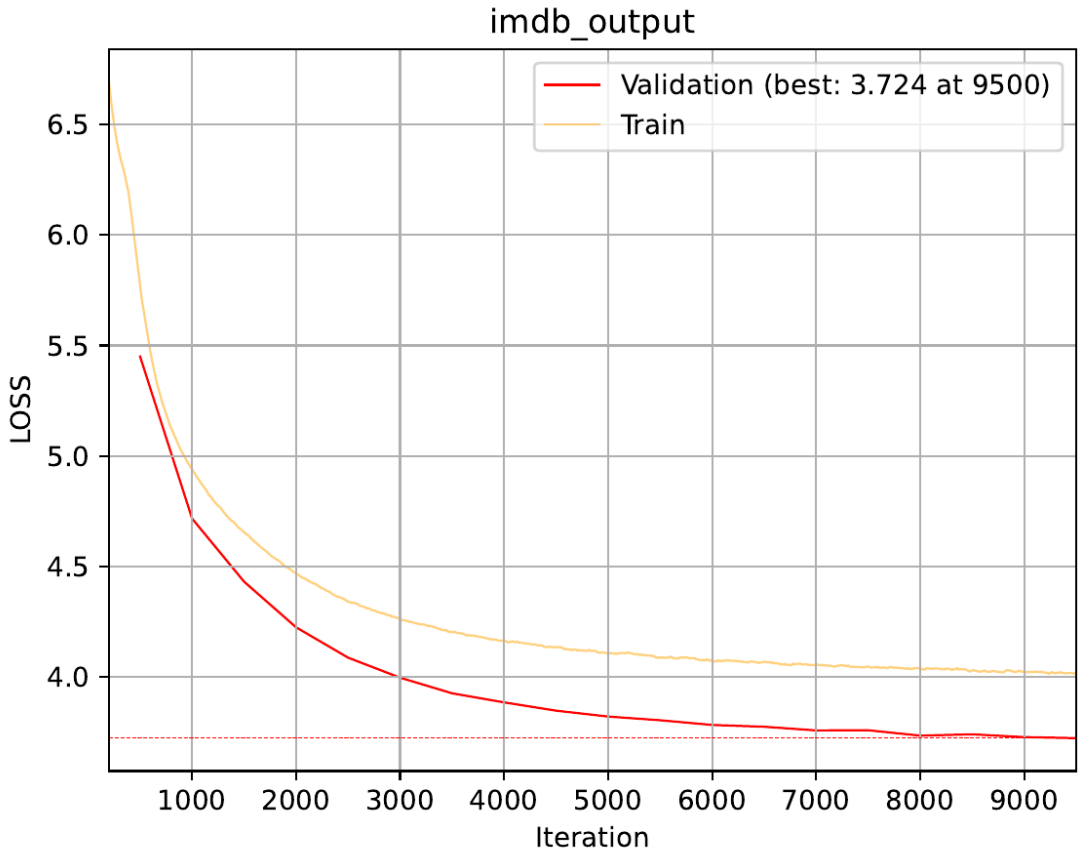
(continues on next page)

(continued from previous page)

```
sampling_config:
  generated_sequence_length: 64
  n_eval_inputs: 1

  manual_inputs:
    - imdb_output: "This movie is the most"

    - imdb_output: "Steven"
```

Since the tokenizer can operate on the raw text, we set `split_on` to `null`, and we can also control the maximum vocabulary size with `adaptive_tokenizer_max_vocab_size` parameter.

Here is the training curve I got for this model:



Here are the auto-generated and manually generated sequences at iteration 500:

Listing 74: Auto-generated sequence at iteration 500

```
als sing d. Cals to making of it to sandly pic. The mapical nos that the cursing in I don
↪'t bave this film is fen the ters to then of the lobangiting is bri
```

Listing 75: Manually sequence at iteration 500 with custom prompt

```
This movie is the mostitob Lredy in cy is fes the movie a drie it that the donly a movie␣
↪was pole a ceing of hy the movie a shiilors of s, the bothed that I don't wark
```

And as before, at iteration 9500, we can see that the model is starting to generate more meaningful sequences:

Listing 76: Auto-generated sequence at iteration 9500

```
push is also a pretty humanizing job (I remembered to do anyone who are the same guy who␣
↪would get a home to do not sit up to call themselves for an exception of TV. and this␣
↪is one of the
```

Listing 77: Manually sequence at iteration 9500 with custom prompt

```
This movie is the mostly gone and power of Sarton (Dean Shouse Farts), Marton Rairedons,␣
↪that she gets inside a classic nonsense and teacher both
```

Hopefully this tutorial has given you a good overview of how to use the sequence generation functionality in `EIR`. Thank you for reading!

F - Serving

In this final section, we demonstrate serving our trained model for sequence generation as a web service and interacting with it using HTTP requests.

Starting the Web Service

To serve the model, use the following command:

```
eirserve --model-path [MODEL_PATH]
```

Replace *[MODEL_PATH]* with the actual path to your trained model. This command initiates a web service that listens for incoming requests.

Here is an example of the command:

```
eirserve \
--model-path eir_tutorials/tutorial_runs/c_sequence_output/01_sequence_generation/saved_
↪models/01_sequence_generation_model_9500_perf-average=-0.3847.pt
```

Important: Currently neither serving nor predicting works with the "bpe" tokenizer due to a bug / design decision in the library that implements it, see here for more information.

Sending Requests

With the server running, we can now send requests for generating sequences based on initial text prompts.

Here's an example Python function demonstrating this process:

```
import requests

def send_request(url: str, payload: dict):
    response = requests.post(url, json=payload)
    return response.json()

example_requests = [
    {"imdb_output": "This movie was great, I have to say "},
    {"imdb_output": "This movie was terrible, I "},
]

for payload in example_requests:
    response = send_request('http://localhost:8000/predict', payload)
    print(f"Prompt: {payload['imdb_output']}")
    print(f"Generated text: {response}\n")
```

Additionally, you can send requests using *bash*:

```
curl -X 'POST' \\
  'http://localhost:8000/predict' \\
  -H 'accept: application/json' \\
  -H 'Content-Type: application/json' \\
  -d '{
      "imdb_output": "This movie was great, I have to say "
  }'
```

Analyzing Responses

After sending requests to the served model, the responses can be analyzed. These responses demonstrate the model's ability to generate text sequences based on the provided prompts.

Listing 78: predictions.json

```
[
    {
        "request": {
            "imdb_output": "This movie was great, I have to say "
        },
        "response": {
            "result": {
                "imdb_output": "This movie was great, I have to say it can have been␣
↪funny, scared after watching a better film about any trying to be a real ca"
            }
        }
    },
    {
        "request": {
```

(continues on next page)

(continued from previous page)

```
            "imdb_output": "This movie was terrible, I "
        },
        "response": {
            "result": {
                "imdb_output": "This movie was terrible, I won't see the rest of the the-
↪written with some way, the worst movies are beautiful. A funny of the f"
            }
        }
    },
    {
        "request": {
            "imdb_output": "This movie was so "
        },
        "response": {
            "result": {
                "imdb_output": "This movie was so hide about shows to watch about her␣
↪rating a good family, his point in figure. The day characters were forgett"
            }
        }
    },
    {
        "request": {
            "imdb_output": "This movi"
        },
        "response": {
            "result": {
                "imdb_output": "This movie can be go even have to make you something␣
↪hopefully force to say how it's setting it and to see this so to so miss th"
            }
        }
    },
    {
        "request": {
            "imdb_output": "Toda"
        },
        "response": {
            "result": {
                "imdb_output": "Today. And falls about some like such the point and␣
↪still with sci-single. A dark and the dark danger and had ending more cast a"
            }
        }
    },
    {
        "request": {
            "imdb_output": ""
        },
        "response": {
            "result": {
                "imdb_output": "nch the genre, it should like watch a brain for each␣
↪other plays to mean so what it destroyed he has off. You spent failing to t"
            }
        }
```

(continues on next page)

(continued from previous page)

```
    }
]
```

If you made it this far, I want to thank you for reading!

2.2.2 02 - Sequence to Sequence: Spanish to English Translation

In this tutorial, we will use `EIR` for sequence-to-sequence tasks. Sequence to Sequence (seq-to-seq) models are a type of models that transform an input sequence into an output sequence, a task relevant for numerous applications like machine translation, summarization, and more.

For this tutorial, our task will be translating Spanish sentences into English, using a dataset from Tatoeba.

A - Data

You can download the data for this tutorial here.

After downloading the data, the folder structure should look like this (we will look at the configs in a bit):

```
eir_tutorials/c_sequence_output/02_sequence_to_sequence
├── conf
│   ├── fusion.yaml
│   ├── globals.yaml
│   ├── input_spanish.yaml
│   └── output.yaml
└── data
    └── eng-spanish
        ├── english.csv
        └── spanish.csv
```

B - Training

Training follows a similar approach as we saw in the previous tutorial, *01 – Sequence Generation: Generating Movie Reviews*.

First, we will train on only the English data, without any Spanish data to establish a baseline.

For reference, here are the configurations:

Listing 79: globals.yaml

```yaml
output_folder: eir_tutorials/tutorial_runs/c_sequence_output/02_seq_to_seq
valid_size: 500
n_saved_models: 1
checkpoint_interval: 500
sample_interval: 500
memory_dataset: true
n_epochs: 10
batch_size: 256
lr: 0.0005
optimizer: "adabelief"
device: "mps"
```

Listing 80: fusion.yaml

```yaml
model_type: "pass-through"
```

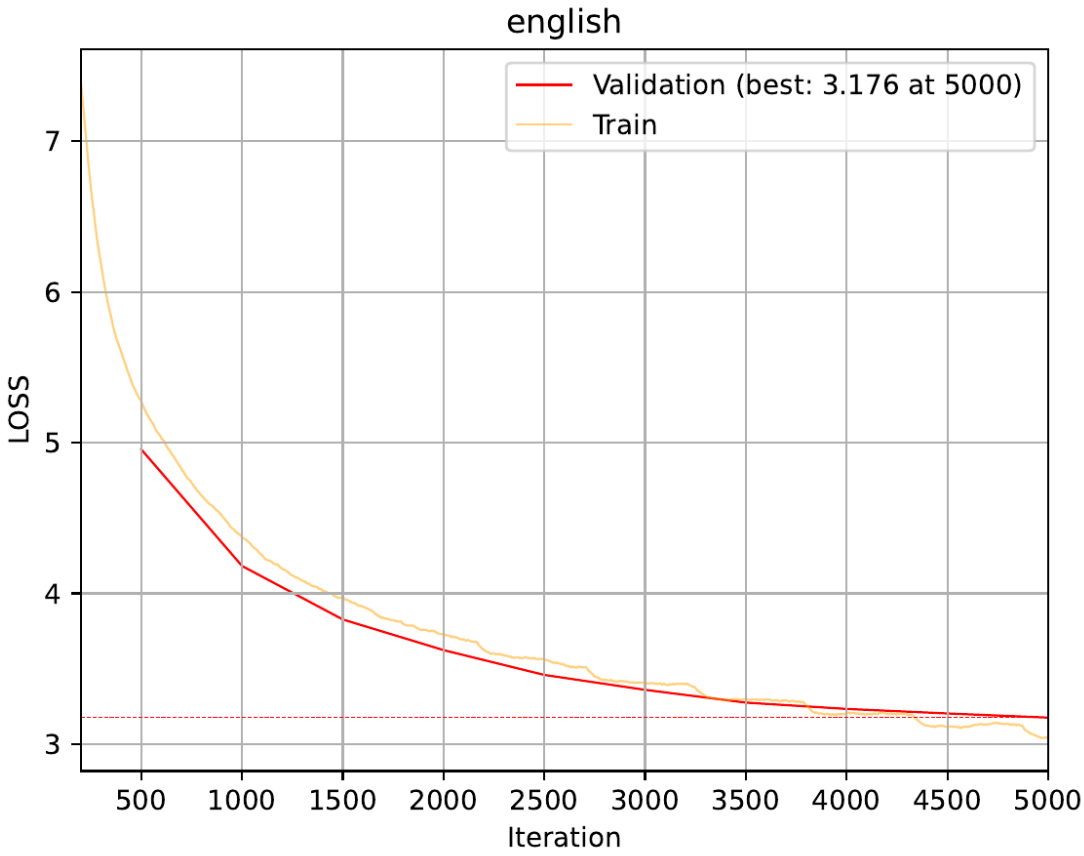
Listing 81: output.yaml

```yaml
output_info:
  output_source: eir_tutorials/c_sequence_output/02_sequence_to_sequence/data/eng-
↪spanish/english.csv
  output_name: english
  output_type: sequence

output_type_info:
  max_length: 32
  split_on: " "
  sampling_strategy_if_longer: "uniform"
  min_freq: 10

model_config:
  embedding_dim: 128
  model_init_config:
    num_layers: 6

sampling_config:
  generated_sequence_length: 64
  n_eval_inputs: 10
```

With these configurations, we can train with the following command:

```
eirtrain \
--global_configs eir_tutorials/c_sequence_output/02_sequence_to_sequence/conf/globals.
↪yaml \
--fusion_configs eir_tutorials/c_sequence_output/02_sequence_to_sequence/conf/fusion.
↪yaml \
--output_configs eir_tutorials/c_sequence_output/02_sequence_to_sequence/conf/output.
↪yaml \
--globals.output_folder=eir_tutorials/tutorial_runs/c_sequence_output/02_seq_to_seq_eng_
↪only
```

When running the command above, I got the following training curve:



Here are a couple of example of the generated sentences using only English data:

Listing 82: Generated English caption using only English data 1

```
Tom
```

Listing 83: Generated English caption using only English data 2

```
I don't have time to do this.
```

While the captions above are make some sense, a more interesting task is actually using the Spanish data as input, and generate the respective English translation. For this, we will include an input configuration for the Spanish data:

Listing 84: input_spanish.yaml

```
input_info:
  input_source: eir_tutorials/c_sequence_output/02_sequence_to_sequence/data/eng-spanish/
↪spanish.csv
  input_name: spanish
  input_type: sequence

input_type_info:
  max_length: 32
  split_on: " "
```

(continues on next page)

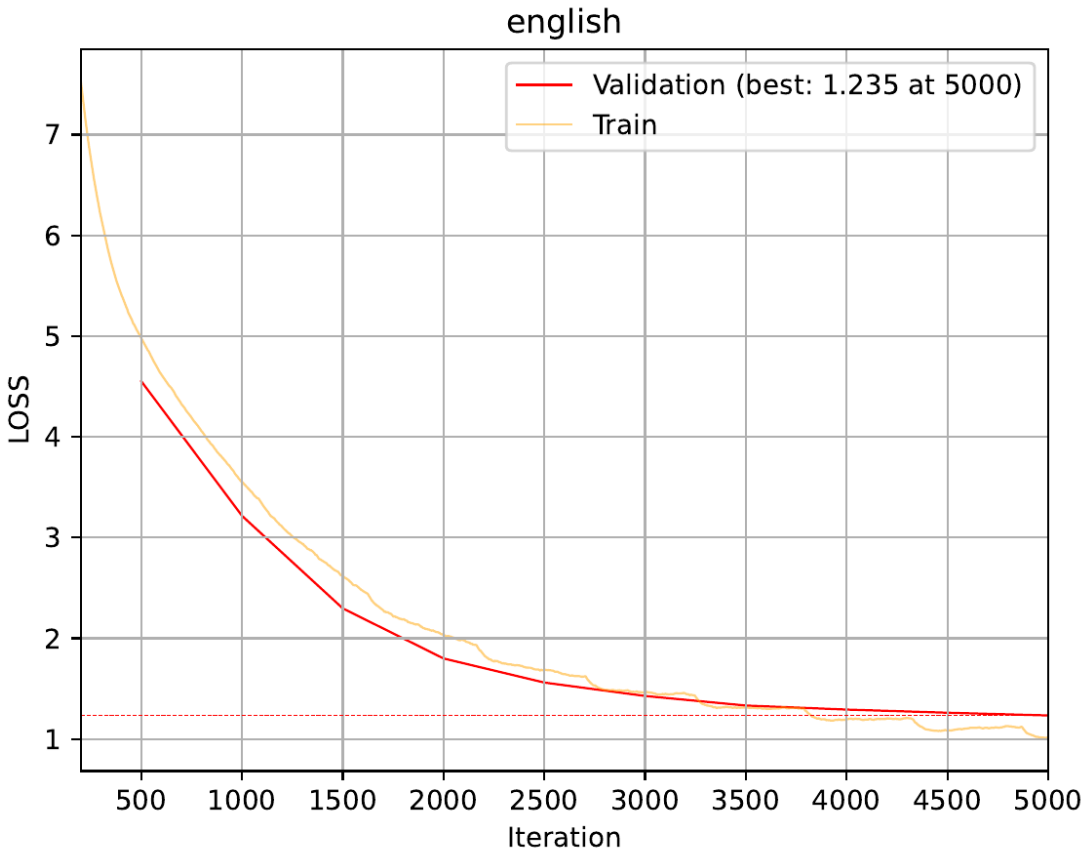
(continued from previous page)

```
  sampling_strategy_if_longer: "uniform"
  min_freq: 10

model_config:
  embedding_dim: 128
  model_init_config:
    num_layers: 6
```

To train, we will use the following command:

```
eirtrain \
--global_configs eir_tutorials/c_sequence_output/02_sequence_to_sequence/conf/globals.
↪yaml \
--input_configs eir_tutorials/c_sequence_output/02_sequence_to_sequence/conf/input_
↪spanish.yaml \
--fusion_configs eir_tutorials/c_sequence_output/02_sequence_to_sequence/conf/fusion.
↪yaml \
--output_configs eir_tutorials/c_sequence_output/02_sequence_to_sequence/conf/output.yaml
```

When running the command above, I got the following training curve:



We can see that the training curve is better than when we only used English data, indicating that the model can utilize the Spanish data to generate the English sentences.

Now, we can look at some of the generated sentences:

While these are not perfect translations, they are maybe not too bad considering a simple model trained for around an hour on a laptop.

C - Serving

In this final section, we demonstrate serving our trained model for sequence-to-sequence translation as a web service and interacting with it using HTTP requests.

Starting the Web Service

To serve the model, use the following command:

```
eirserve --model-path [MODEL_PATH]
```

Replace *[MODEL_PATH]* with the actual path to your trained model. This command initiates a web service that listens for incoming requests.

Here is an example of the command:

```
eirserve \
--model-path eir_tutorials/tutorial_runs/c_sequence_output/02_seq_to_seq/saved_models/02_
↪seq_to_seq_model_5000_perf-average=-0.2346.pt
```

Sending Requests

With the server running, we can now send requests for translating text from Spanish to English.

Here's an example Python function demonstrating this process:

```
import requests

def send_request(url: str, payload: dict):
    response = requests.post(url, json=payload)
    return response.json()

example_requests = [
    {"english": "", "spanish": "Tengo mucho hambre"},
    {"english": "", "spanish": "¿Por qué Tomás sigue en Boston?"},
]

for payload in example_requests:
    response = send_request('http://localhost:8000/predict', payload)
    print(f"Spanish: {payload['spanish']}")
    print(f"Translated to English: {response['english']}\n")
```

Additionally, you can send requests using *bash*:

```
curl -X 'POST' \\
  'http://localhost:8000/predict' \\
  -H 'accept: application/json' \\
```

(continues on next page)

(continued from previous page)

```
  -H 'Content-Type: application/json' \\
  -d '{
      "english": "", "spanish": "Tengo mucho hambre"
  }'
```

Analyzing Responses

After sending requests to the served model, the responses can be analyzed. These responses provide insights into the model's ability to translate from Spanish to English.

Listing 85: predictions.json

```
[
    {
        "request": {
            "english": "",
            "spanish": "Tengo mucho hambre"
        },
        "response": {
            "result": {
                "english": "I'm very hungry and"
            }
        }
    },
    {
        "request": {
            "english": "",
            "spanish": "¿Por qué Tomás sigue en Boston?"
        },
        "response": {
            "result": {
                "english": "Why is Tom still in Boston?"
            }
        }
    },
    {
        "request": {
            "english": "Why",
            "spanish": "¿Por qué Tomás sigue en Boston?"
        },
        "response": {
            "result": {
                "english": "Why is Tom still Boston?"
            }
        }
    },
    {
        "request": {
            "english": "",
            "spanish": "Un gato muy alto"
        },
```

(continues on next page)

(continued from previous page)

```
        "response": {
            "result": {
                "english": "A cat was very high."
            }
        }
    }
]
```

Thanks for reading!

2.2.3 03 - Image to Sequence: Image Captioning

In this tutorial, we will utilize `EIR` for image-to-sequence tasks. Image to Sequence (img-to-seq) models are a type of models that convert an input image into a sequence of words. This could be useful for tasks like image captioning, where the model generates a description of the contents of an image.

In this tutorial, we will be generating captions for images using the COCO 2017 dataset.

A - Data

You can download the data for this tutorial here.

After downloading the data, the folder structure should look like this (we will look at the configs in a bit):

```
eir_tutorials/c_sequence_output/03_image_captioning
├── conf
│   ├── fusion.yaml
│   ├── globals.yaml
│   ├── inputs_resnet18.yaml
│   └── output.yaml
└── data
    └── image_captioning
        ├── captions.csv
        └── images
```

B - Training

Training follows a similar approach as we saw in the previous tutorial, *01 – Sequence Generation: Generating Movie Reviews*.

For reference, here are the configurations:

Listing 86: globals.yaml

```
output_folder: eir_tutorials/tutorial_runs/c_sequence_output/03_image_captioning
valid_size: 500
n_saved_models: 1
checkpoint_interval: 500
sample_interval: 500
memory_dataset: false
n_epochs: 3
```

(continues on next page)

(continued from previous page)

```
batch_size: 64
lr: 0.0005
optimizer: "adabelief"
device: "mps"
```

Listing 87: fusion.yaml

```
model_type: "pass-through"
```

Listing 88: inputs_resnet18.yaml

```
input_info:
  input_source: eir_tutorials/c_sequence_output/03_image_captioning/data/image_
↪captioning/images
  input_name: image_captioning
  input_type: image

input_type_info:
  size:
    - 64
  auto_augment: true

model_config:
  model_type: "resnet18"
  pretrained_model: True
```

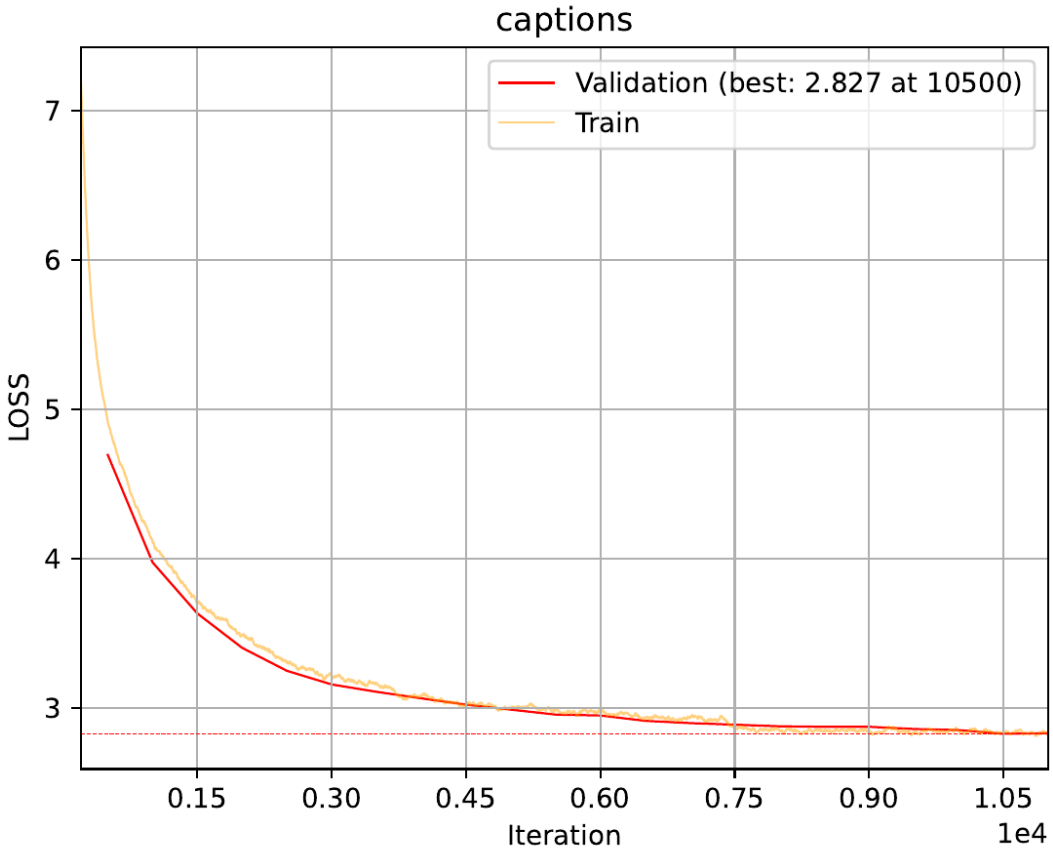
Listing 89: output.yaml

```
output_info:
  output_source: eir_tutorials/c_sequence_output/03_image_captioning/data/image_
↪captioning/captions.csv
  output_name: captions
  output_type: sequence

output_type_info:
  max_length: 32
  split_on: " "
  sampling_strategy_if_longer: "uniform"
  min_freq: 20

model_config:
  embedding_dim: 128
  model_init_config:
    num_layers: 6

sampling_config:
  generated_sequence_length: 64
  n_eval_inputs: 10
```

Like previously, we will start by training a model only on the text to establish as baseline:

```
eirtrain \
--global_configs eir_tutorials/c_sequence_output/03_image_captioning/conf/globals.yaml \
--fusion_configs eir_tutorials/c_sequence_output/03_image_captioning/conf/fusion.yaml \
--output_configs eir_tutorials/c_sequence_output/03_image_captioning/conf/output.yaml \
--globals.output_folder=eir_tutorials/tutorial_runs/c_sequence_output/03_image_
↪captioning_text_only
```

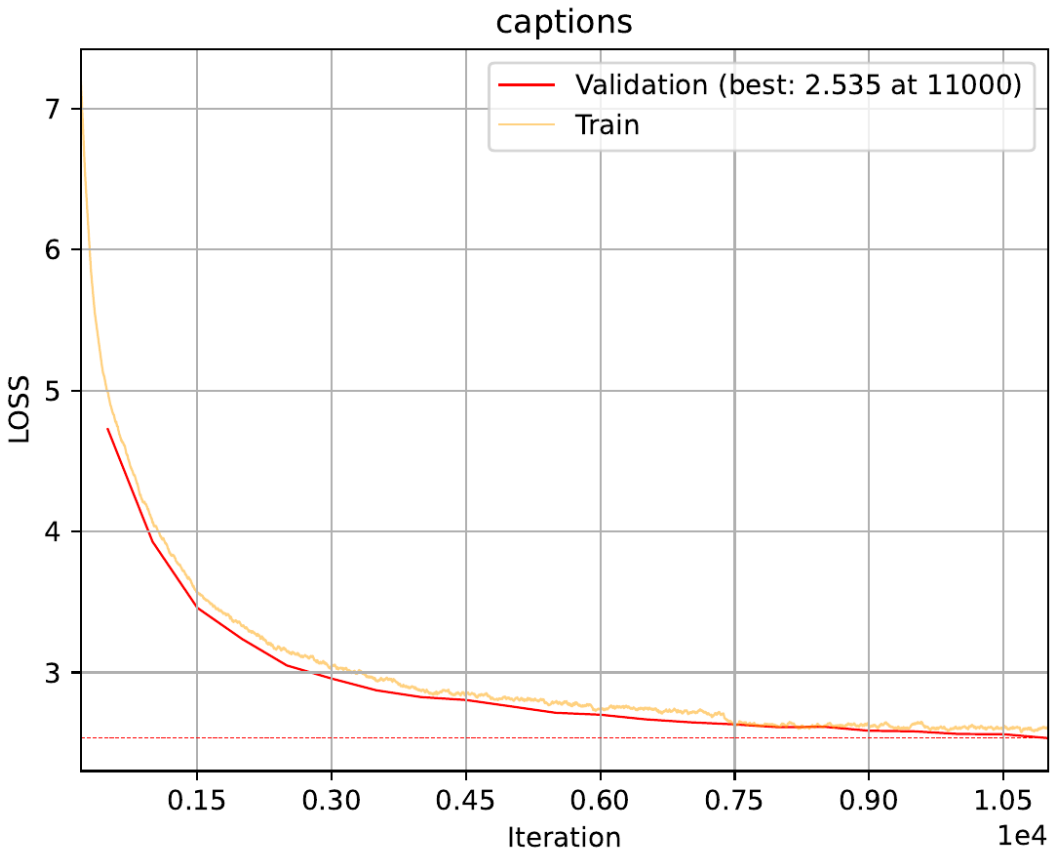
When running the command above, I got the following training curve:



Now, we will train a model that uses both the image and the text:

```
eirtrain \
--global_configs eir_tutorials/c_sequence_output/03_image_captioning/conf/globals.yaml \
--input_configs eir_tutorials/c_sequence_output/03_image_captioning/conf/inputs_resnet18.
↪yaml \
--fusion_configs eir_tutorials/c_sequence_output/03_image_captioning/conf/fusion.yaml \
--output_configs eir_tutorials/c_sequence_output/03_image_captioning/conf/output.yaml
```

When running the command above, I got the following training curve:



The fact that the validation loss is lower indicates that the model is likely able to use the image to improve the quality of the captions.

After training, we can look at some of the generated captions:

A group of people are riding on the back of a

A street filled with lots of cars and a sign that says

A small brown dog standing on a grass covered field.

A man is standing in a rocky beach.

Three giraffes stand in a grassy field near some rocks.

A horse standing next to a fence on a brick building

A kitchen with a stove and microwave on top of it.

A man in a suit and tie wearing a hat

The people are standing and on a hill on their skis.

A large brick building with a clock on top of it.

While the captions seem to be somewhat related to the images, they are far from perfect. As the validation loss is still decreasing, we could train the model for longer, try a larger model, use larger images, or use a larger dataset.

D - Serving

In this final section, we demonstrate serving our trained image captioning model as a web service and interacting with it using HTTP requests.

Starting the Web Service

To serve the model, use the following command:

```
eirserve --model-path [MODEL_PATH]
```

Replace *[MODEL_PATH]* with the actual path to your trained model. This command initiates a web service that listens for incoming requests.

Here is an example of the command:

```
eirserve \
--model-path eir_tutorials/tutorial_runs/c_sequence_output/03_image_captioning/saved_
↪models/03_image_captioning_model_11000_perf-average=-1.5346.pt
```

Sending Requests

With the server running, we can now send image-based requests for caption generation. For this model, we send images and receive their captions.

Here's an example Python function demonstrating this process:

```
import requests
import base64
from PIL import Image
from io import BytesIO

def encode_image_to_base64(file_path: str) -> str:
    with Image.open(file_path) as image:
        buffered = BytesIO()
        image.save(buffered, format="JPEG")
        return base64.b64encode(buffered.getvalue()).decode("utf-8")

def send_request(url: str, payload: dict):
    response = requests.post(url, json=payload)
    return response.json()

payload = {
    "image_captioning": encode_image_to_base64("path/to/image.jpg"),
    "captions": ""
}

response = send_request('http://localhost:8000/predict', payload)
print(response)
```

Additionally, you can send requests using *bash*. Note that this requires preparing the base64-encoded image content in advance:

```
curl -X 'POST' \\
  'http://localhost:8000/predict' \\
  -H 'accept: application/json' \\
  -H 'Content-Type: application/json' \\
  -d '{
      "image_captioning": "[BASE64_ENCODED_IMAGE]",
      "captions": ""
  }'
```

Analyzing Responses

Before analyzing the responses, let's view the images that were used for generating captions:

000000000009.jpg

000000000034.jpg

000000581929.jpg

After sending requests to the served model, the responses can be analyzed. These responses provide insights into the model's capability to generate captions for the input images.

Listing 90: predictions.json

```
[
    {
        "request": {
            "image_captioning": "eir_tutorials/c_sequence_output/03_image_captioning/
↪data/image_captioning/images/000000000009.jpg",
            "captions": ""
        },
        "response": {
            "result": {
                "captions": "A bowl of broccoli and a is on a plate."
            }
```

(continues on next page)

(continued from previous page)

```
        }
    },
    {
        "request": {
            "image_captioning": "eir_tutorials/c_sequence_output/03_image_captioning/
↪data/image_captioning/images/000000000034.jpg",
            "captions": ""
        },
        "response": {
            "result": {
                "captions": "Two zebras standing side by side in a grassy field."
            }
        }
    },
    {
        "request": {
            "image_captioning": "eir_tutorials/c_sequence_output/03_image_captioning/
↪data/image_captioning/images/000000581929.jpg",
            "captions": "A horse"
        },
        "response": {
            "result": {
                "captions": "A horse and a goat grazing in the grass"
            }
        }
    }
]
```

Thank you for reading!

2.2.4 04 - Tabular to Sequence: Protein Sequence Generation

In this tutorial, we'll employ `EIR` for sequence generation conditioned on tabular data. Specifically, we will be generating protein sequences conditioned on their classification.

A - Data

The dataset for this tutorial can be downloaded from here.

This dataset is processed from a Kaggle dataset available here. The original data, in turn, originates from the RCSB Protein Data Bank.

After downloading the data, your folder structure should look something like this (we will add the configuration files as we progress):

```
eir_tutorials/c_sequence_output/04_protein_sequence_generation
├── conf
│   ├── fusion.yaml
│   ├── globals.yaml
│   ├── inputs_tabular.yaml
│   ├── inputs_tabular_test.yaml
│   ├── output.yaml
```

(continues on next page)

(continued from previous page)

```
│   ├── output_conditioned.yaml
│   └── output_conditioned_test.yaml
└── data
    ├── test_protein_sequences.csv
    ├── test_tabular_info.csv
    ├── train_protein_sequences.csv
    └── train_tabular_info.csv
```

B - Unconditional Protein Sequence Generation

Training will be similar to what we did in a previous tutorial, *01 – Sequence Generation: Generating Movie Reviews*. First, we will start by establishing a baseline by training a model on the protein sequences only:

Below are the relevant configurations:

Listing 91: globals.yaml

```yaml
output_folder: eir_tutorials/tutorial_runs/c_sequence_output/04_protein_sequences
valid_size: 512
n_saved_models: 1
checkpoint_interval: 500
sample_interval: 500
memory_dataset: false
n_epochs: 20
batch_size: 256
lr: 0.0005
optimizer: "adabelief"
device: "mps"
latent_sampling:
  layers_to_sample:
    - "output_modules.protein_sequence.output_transformer.layers.1"
```

Listing 92: fusion.yaml

```yaml
model_type: "pass-through"
```

Listing 93: output.yaml

```yaml
output_info:
  output_source: eir_tutorials/c_sequence_output/04_protein_sequence_generation/data/
↪train_protein_sequences.csv
  output_name: protein_sequence
  output_type: sequence

output_type_info:
  max_length: 128
  split_on: ""
  sampling_strategy_if_longer: "uniform"
  min_freq: 1

model_config:
```

(continues on next page)

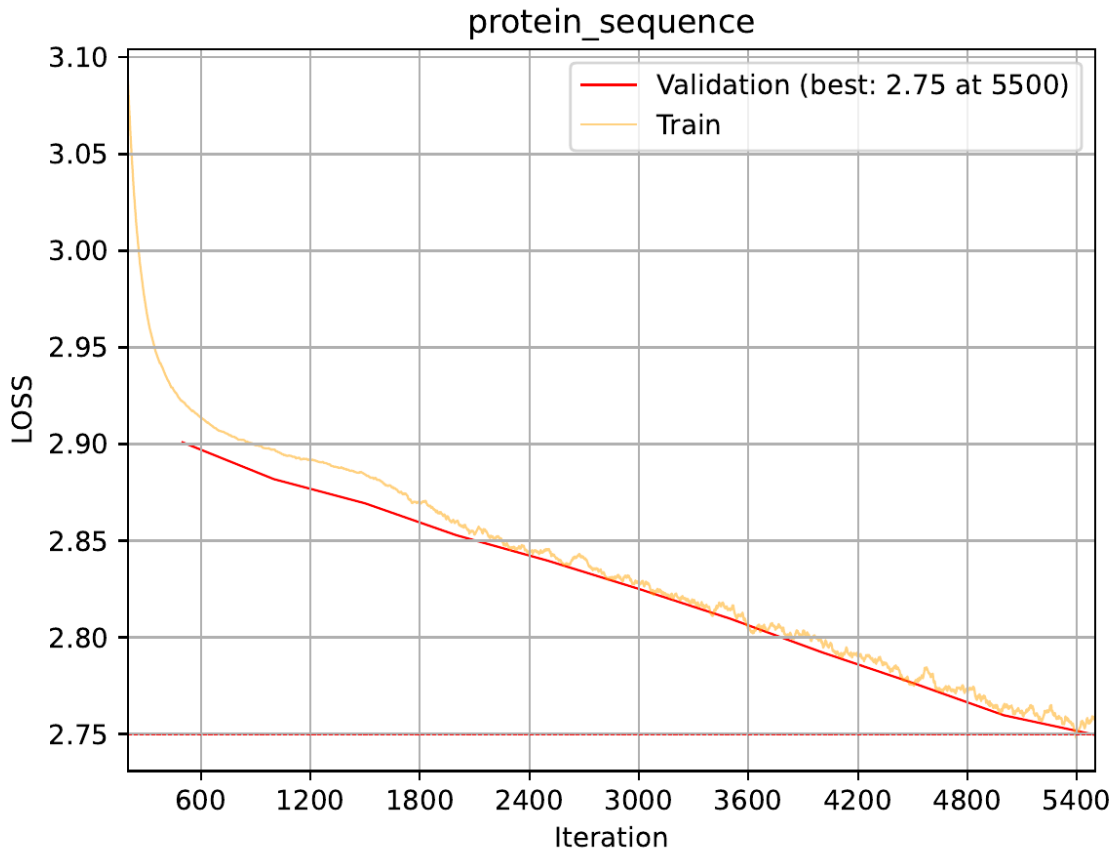
(continued from previous page)

```
  embedding_dim: 64

sampling_config:
  generated_sequence_length: 128
  n_eval_inputs: 10
```

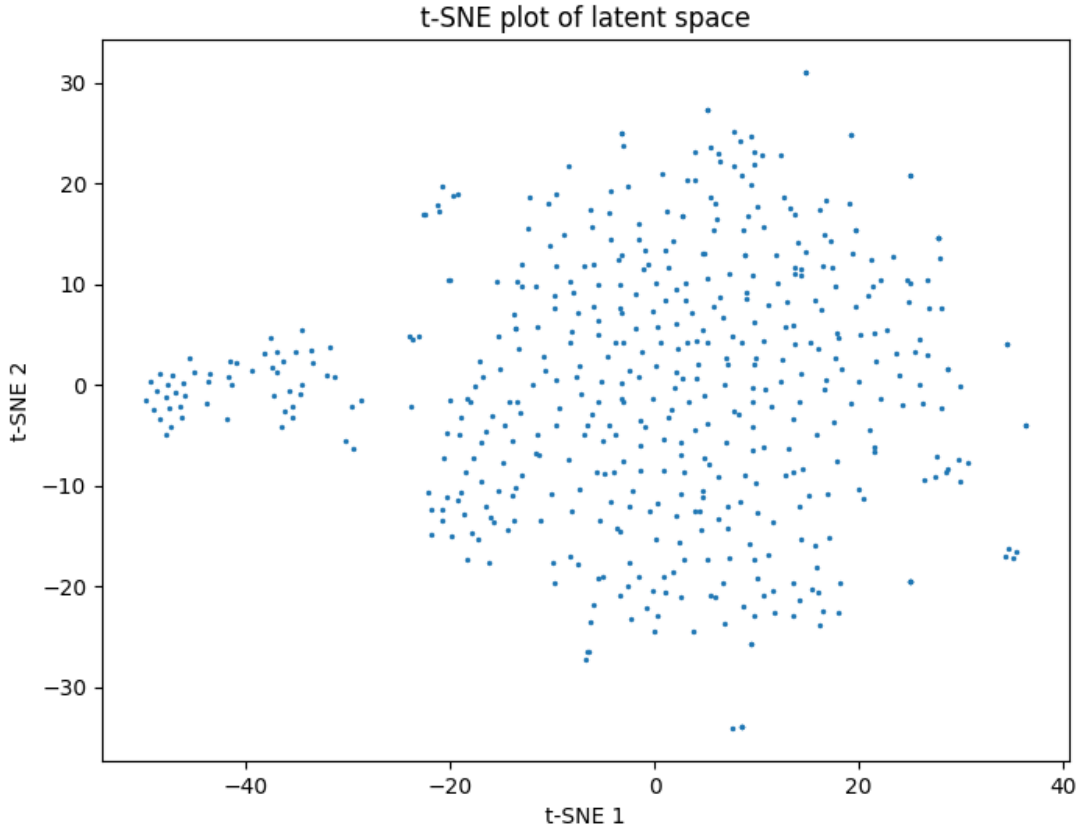
Training the model:

```
eirtrain \
--global_configs eir_tutorials/c_sequence_output/04_protein_sequence_generation/conf/
↪globals.yaml \
--fusion_configs eir_tutorials/c_sequence_output/04_protein_sequence_generation/conf/
↪fusion.yaml \
--output_configs eir_tutorials/c_sequence_output/04_protein_sequence_generation/conf/
↪output.yaml \
--globals.output_folder=eir_tutorials/tutorial_runs/c_sequence_output/04_protein_
↪sequence_generation_sequence_only
```

Executing the command above resulted in the following training curve:



You might have noticed the `latent_sampling` parameter in the global configuration, which allows us to extract a representation from a specified the model. In a addition to saving the validation set representations, we also get a couple of visualizations. For example, here is a t-SNE plot of the validation set representations at iteration 5000:



C - Conditional Protein Sequence Generation

Next, we'll train a model incorporating both tabular data, which contains the protein type classification and the protein sequences.

For this, we add the input configuration containing the tabular data:

Listing 94: input.yaml

```
input_info:
  input_source: eir_tutorials/c_sequence_output/04_protein_sequence_generation/data/
↪train_tabular_info.csv
  input_name: proteins_tabular
  input_type: tabular

input_type_info:
  input_cat_columns:
    - classification
```

Additionally, we can update our output configuration to generate sequences based on manually specified tabular input values:

Listing 95: output.yaml

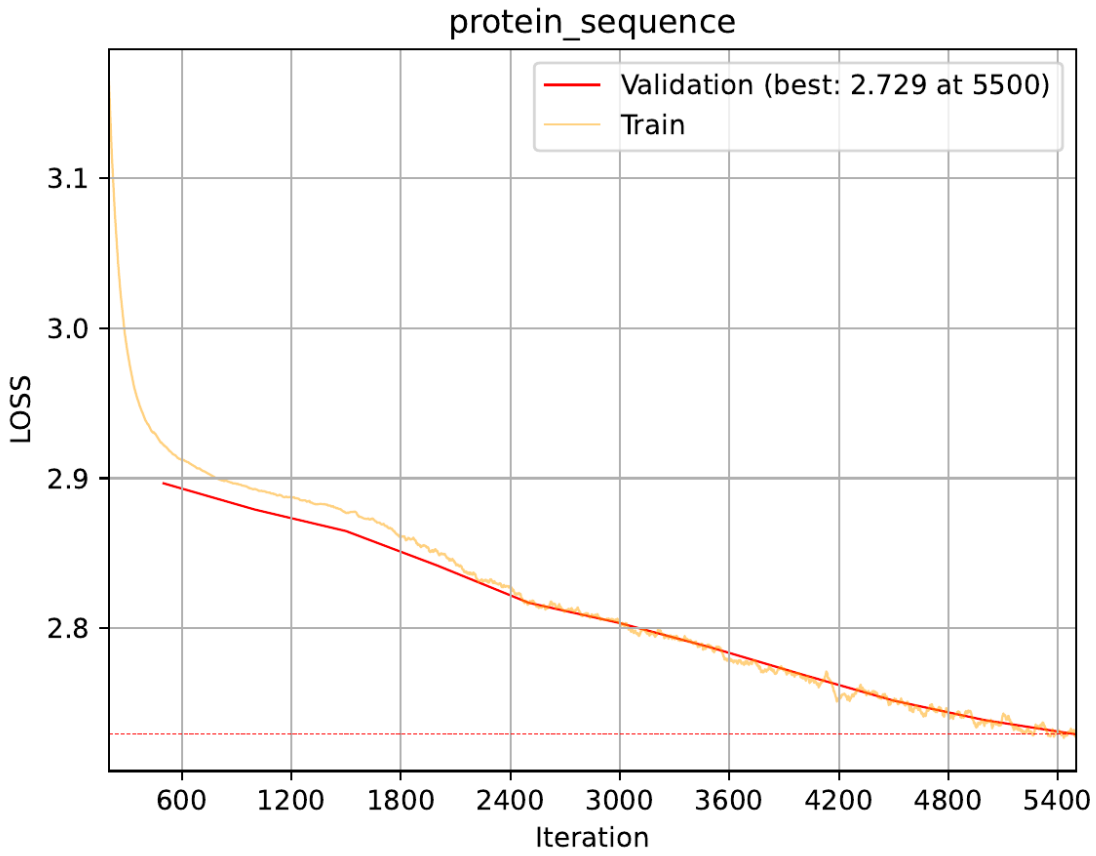
```yaml
output_info:
  output_source: eir_tutorials/c_sequence_output/04_protein_sequence_generation/data/
↪train_protein_sequences.csv
  output_name: protein_sequence
  output_type: sequence

output_type_info:
  max_length: 128
  split_on: ""
  sampling_strategy_if_longer: "uniform"
  min_freq: 1

model_config:
  embedding_dim: 64

sampling_config:
  generated_sequence_length: 128
  n_eval_inputs: 0

  manual_inputs:
    - proteins_tabular:
        classification: "HYDROLASE"
      protein_sequence: ""

    - proteins_tabular:
        classification: "TRANSFERASE"
      protein_sequence: ""

    - proteins_tabular:
        classification: "OXIDOREDUCTASE"
      protein_sequence: ""
```

Note: While not shown here, you can view the generated sequences in the `samples/<iteration>/manual` folder during/after training.

Training the conditional model:

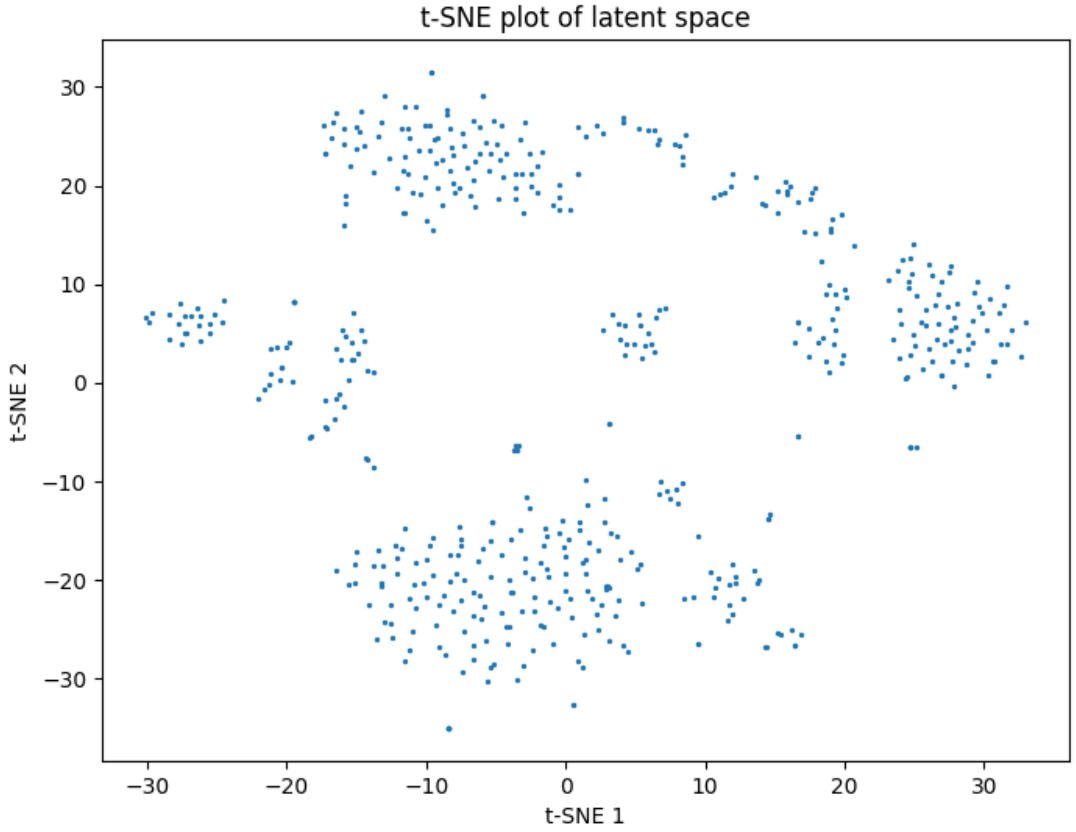
```
eirtrain \
--global_configs eir_tutorials/c_sequence_output/04_protein_sequence_generation/conf/
↪globals.yaml \
--input_configs eir_tutorials/c_sequence_output/04_protein_sequence_generation/conf/
↪inputs_tabular.yaml \
--fusion_configs eir_tutorials/c_sequence_output/04_protein_sequence_generation/conf/
↪fusion.yaml \
--output_configs eir_tutorials/c_sequence_output/04_protein_sequence_generation/conf/
↪output_conditioned.yaml \
--globals.output_folder=eir_tutorials/tutorial_runs/c_sequence_output/04_protein_
↪sequence_generation_tabular
```

When executing the above command, the following training curve was obtained:



The (albeit slightly) lowered validation loss suggests the model effectively uses tabular data to improve sequence quality.

Similarly to before, we can visualize the validation set representations at iteration 5000, now for the conditional model:



The separation does seem to be slightly better than before, which could be due to the model given the additional information from the tabular data.

D - Generating New Sequences of a Specific Protein Type

Finally, we will take a quick look at how we can use a trained model to generate new sequences of a specific protein type. For this, we will use configuration files similar to the ones used for training, but now pointing to the test set data:

Listing 96: input.yaml

```
input_info:
  input_source: eir_tutorials/c_sequence_output/04_protein_sequence_generation/data/test_
↪tabular_info.csv
  input_name: proteins_tabular
  input_type: tabular

input_type_info:
  input_cat_columns:
    - classification
```

Listing 97: output.yaml

```yaml
output_info:
  output_source: eir_tutorials/c_sequence_output/04_protein_sequence_generation/data/
↪test_protein_sequences.csv
  output_name: protein_sequence
  output_type: sequence

output_type_info:
  max_length: 128
  split_on: ""
  sampling_strategy_if_longer: "uniform"
  min_freq: 1

model_config:
  embedding_dim: 64

sampling_config:
  generated_sequence_length: 512
  n_eval_inputs: 0

  manual_inputs:
    - proteins_tabular:
        classification: "HYDROLASE"
      protein_sequence: ""

    - proteins_tabular:
        classification: "TRANSFERASE"
      protein_sequence: ""

    - proteins_tabular:
        classification: "OXIDOREDUCTASE"
      protein_sequence: ""
```

Now, we can use the `eirpredict` command as follows:

```
eirpredict \
--global_configs eir_tutorials/c_sequence_output/04_protein_sequence_generation/conf/
↪globals.yaml \
--input_configs eir_tutorials/c_sequence_output/04_protein_sequence_generation/conf/
↪inputs_tabular_test.yaml \
--fusion_configs eir_tutorials/c_sequence_output/04_protein_sequence_generation/conf/
↪fusion.yaml \
--output_configs eir_tutorials/c_sequence_output/04_protein_sequence_generation/conf/
↪output_conditioned_test.yaml \
--model_path eir_tutorials/tutorial_runs/c_sequence_output/04_protein_sequence_
↪generation_tabular/saved_models/04_protein_sequence_generation_tabular_model_5500_perf-
↪average=-1.7293.pt \
--output_folder eir_tutorials/tutorial_runs/c_sequence_output/04_protein_sequence_
↪generation_tabular/test_results \
--evaluate
```

This will save the results in the specified `--output_folder`. While we do evaluate the loss, it's perhaps more inter-

esting to look at the generated sequences as well as the latent sampling, available in the `results` and `latents` folders, respectively.

F - Serving

In this final section, we demonstrate serving our trained model for protein sequence generation with tabular inputs as a web service and interacting with it using HTTP requests.

Starting the Web Service

To serve the model, use the following command:

```
eirserve --model-path [MODEL_PATH]
```

Replace *[MODEL_PATH]* with the actual path to your trained model. This command initiates a web service that listens for incoming requests.

Here is an example of the command:

```
eirserve \
--model-path eir_tutorials/tutorial_runs/c_sequence_output/04_protein_sequence_
↪generation_tabular/saved_models/04_protein_sequence_generation_tabular_model_5500_perf-
↪average=-1.7293.pt
```

Sending Requests

With the server running, we can now send requests that include tabular data to generate protein sequences.

Here's an example Python function demonstrating this process:

```
import requests

def send_request(url: str, payload: dict):
    response = requests.post(url, json=payload)
    return response.json()

example_requests = [
    {"proteins_tabular": {"classification": "HYDROLASE"}, "protein_sequence": ""},
    {"proteins_tabular": {"classification": "TRANSFERASE"}, "protein_sequence": ""},
]

for payload in example_requests:
    response = send_request('http://localhost:8000/predict', payload)
    print(f"Classification: {payload['proteins_tabular']['classification']}")
    print(f"Generated protein sequence: {response['protein_sequence']}\n")
```

Additionally, you can send requests using *bash*:

```
curl -X 'POST' \\
  'http://localhost:8000/predict' \\
  -H 'accept: application/json' \\
  -H 'Content-Type: application/json' \\
```

(continues on next page)

(continued from previous page)

```
  -d '{
      "proteins_tabular": {"classification": "HYDROLASE"},
      "protein_sequence": ""
  }'
```

Analyzing Responses

After sending requests to the served model, the responses can be analyzed. These responses provide insights into the model's ability to generate protein sequences based on the tabular input.

Listing 98: predictions.json

```
[
    {
        "request": {
            "proteins_tabular": {
                "classification": "HYDROLASE"
            },
            "protein_sequence": ""
        },
        "response": {
            "result": {
                "protein_sequence":
↪"EILYEGKLLSGGVDAVFLPVRRDIKSVSALGYQSVDEDRILQSGDTIIVRDGPKIIGGLRAHAVHESIGLTLEGPAEFGVGSPEARFDETVRRTGVLVDH
↪"
            }
        }
    },
    {
        "request": {
            "proteins_tabular": {
                "classification": "TRANSFERASE"
            },
            "protein_sequence": ""
        },
        "response": {
            "result": {
                "protein_sequence":
↪"KEIYLNGAVNKYIYNVTNLSSGKEATKDIKKASKVTGQAAIREVKGDKIIKAYARKEDKLSKDPIIKDNLIVGIKELISFEYVTGNPDFVSLRLKGVLGG
↪"
            }
        }
    },
    {
        "request": {
            "proteins_tabular": {
                "classification": "OXIDOREDUCTASE"
            },
            "protein_sequence": "AAA"
        },
        "response": {
```

(continues on next page)

(continued from previous page)

```
            "result": {
                "protein_sequence":
↪"AAALLKLKKAVVLTGSQAILALGAVGAGASLRGGSADFQPVVAPGTASGIPTASVTFVKEAAQVLAENAATAVFGRDGDALRLTVTDAELDRTVETRVSP
↪"
            }
        }
    }
]
```

If you made it this far, I want to thank you for reading!

Thank you for reading!

2.3 Array Generation

2.3.1 01 – Array Output: Building a Simple Autoencoder for MNIST Digit Generation

In this tutorial, we will explore the capabilities of *EIR* for array output tasks, specifically focusing on MNIST digit generation using a simple autoencoder. Arrays can represent various types of data, including images, time series, and more. This technique allows us to generate new, meaningful arrays based on patterns learned from the training data.

Note: This tutorial assumes you are familiar with the basics of *EIR*, and have gone through previous tutorials. Not required, but recommended.

A - Data

Here, we will be using the well known MNIST dataset. The dataset here consists of preprocessed NumPy arrays containing the MNIST handwritten digit images. To download the data, use this link.

After downloading the data, the folder structure should look like this:

```
eir_tutorials/d_array_output/01_array_mnist_generation
├── conf
│   ├── globals.yaml
│   ├── input_mnist_array.yaml
│   ├── input_mnist_array_with_label.yaml
│   ├── input_mnist_label.yaml
│   ├── output.yaml
│   └── output_with_label.yaml
└── data
    ├── __MACOSX
    ├── mnist_labels.csv
    └── mnist_npy
```

B - Training A Simple Autoencoder

Training an autoencoder for MNIST digit generation with *EIR* involves the familiar configuration files and follows a process similar to supervised learning. We'll discuss the key configurations and visualize the training process, including the training curve and generated images at different iterations.

The global config provides standard parameters for training:

Listing 99: globals.yaml

```yaml
output_folder: eir_tutorials/tutorial_runs/d_array_output/01_array_mnist_generation
checkpoint_interval: 1000
sample_interval: 1000
valid_size: 1000
batch_size: 64
n_epochs: 10
device: "cpu"
optimizer: adabelief
lr: 0.001
memory_dataset: true
latent_sampling:
  layers_to_sample:
    - "fusion_modules.computed.fusion_modules.fusion.1.0"
```

Note: One new thing you might notice here is the `latent_sampling` configuration in the global configuration, which let's you extract and visualize the latent space of chosen layers during training (computed on the validation set).

The input configuration specifies the structure of the MNIST array input:

Listing 100: input_mnist_array.yaml

```yaml
input_info:
  input_source: "eir_tutorials/d_array_output/01_array_mnist_generation/data/mnist_npy"
  input_name: mnist
  input_type: array

input_type_info:
  normalization: channel
  adaptive_normalization_max_samples: 10000


model_config:
  model_type: lcl
  model_init_config:
    kernel_width: 8
    attention_inclusion_cutoff: 128
```

The output configuration defines the structure and settings for the generated images:

Listing 101: output.yaml

```yaml
output_info:
  output_source: "eir_tutorials/d_array_output/01_array_mnist_generation/data/mnist_npy"
```

(continues on next page)

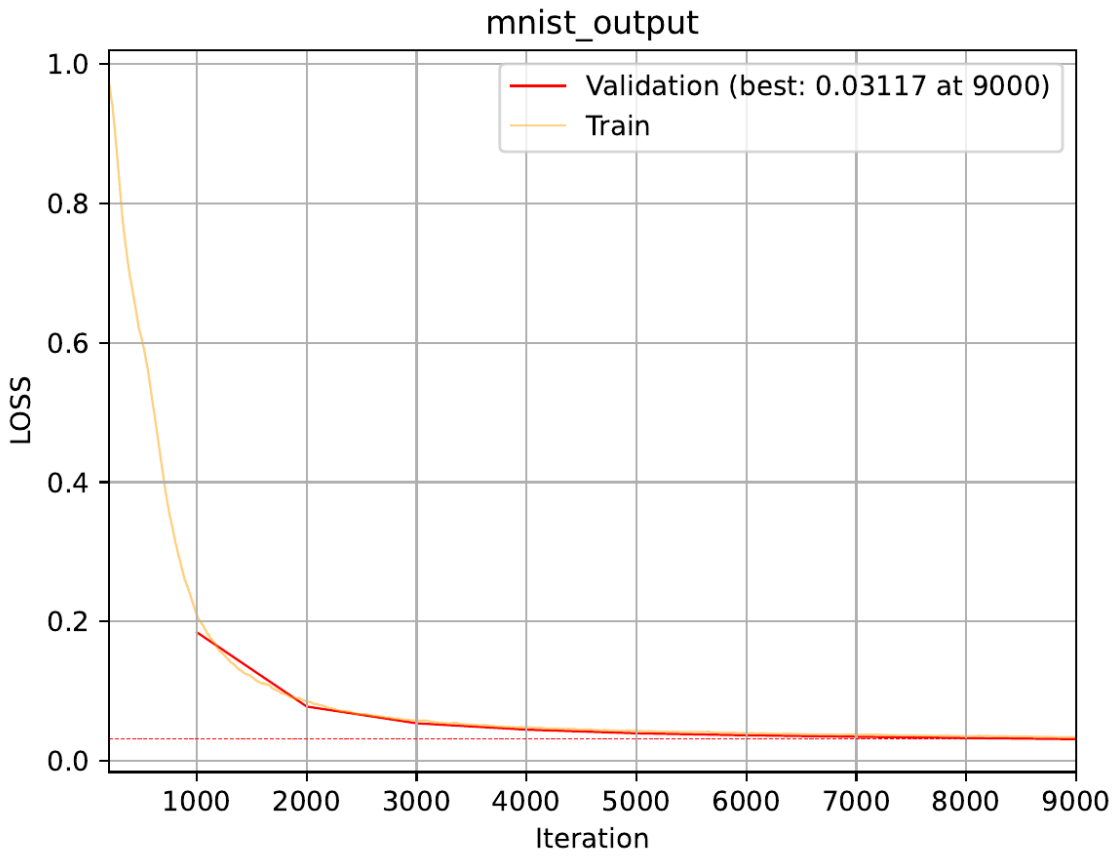
(continued from previous page)

```
  output_name: mnist_output
  output_type: array

output_type_info:
  normalization: channel
  adaptive_normalization_max_samples: 10000

model_config:
  model_type: lcl
  model_init_config:
    channel_exp_base: 4
```

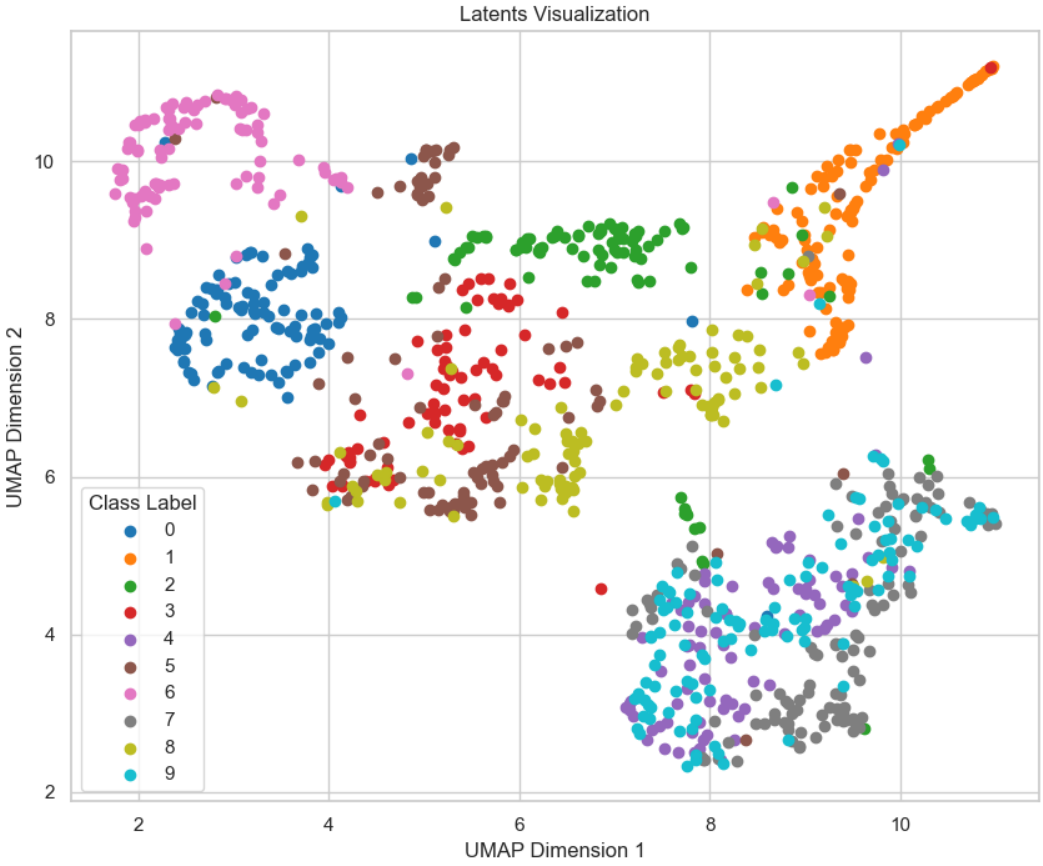
With the configurations in place, we can run the following command to start the training process:

```
eirtrain \
--global_configs eir_tutorials/d_array_output/01_array_mnist_generation/conf/globals.
↪yaml \
--input_configs eir_tutorials/d_array_output/01_array_mnist_generation/conf/input_mnist_
↪array.yaml \
--output_configs eir_tutorials/d_array_output/01_array_mnist_generation/conf/output.yaml
```

I got the following results:



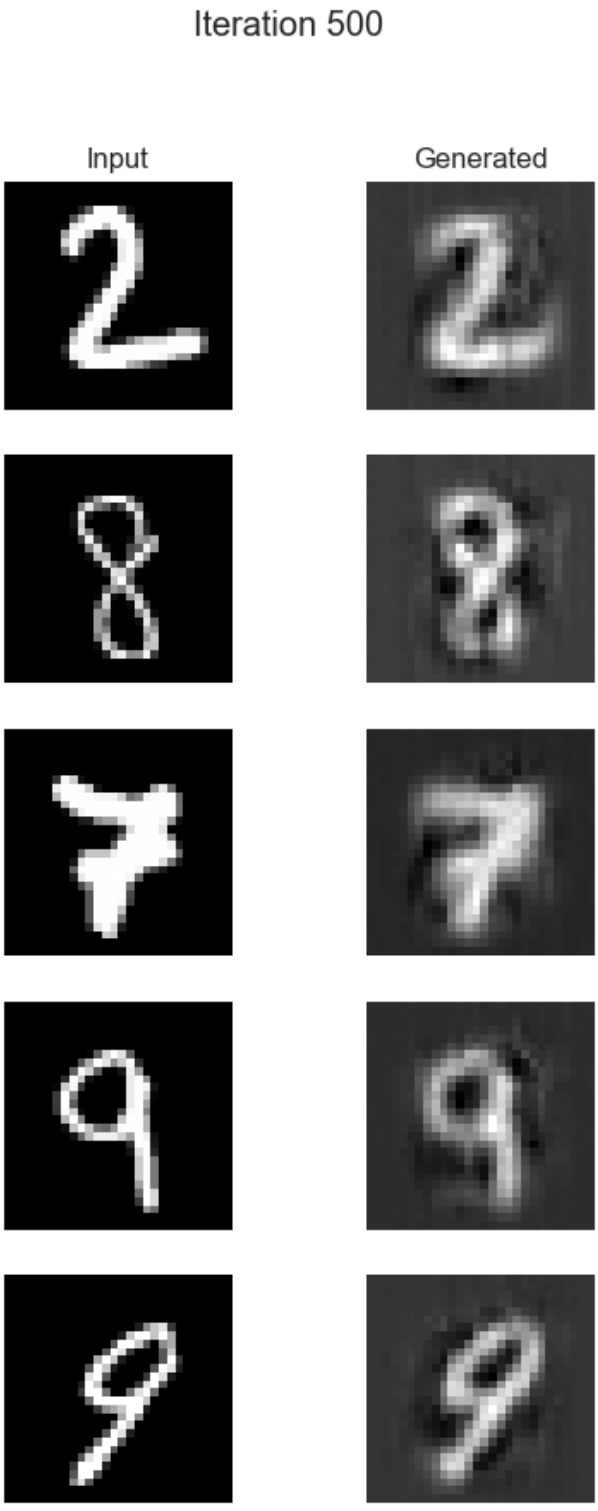
Since we had that latent space sampling configuration in the global config, the latents are saved and a couple of visualizations are generated, here is one with the t-SNE visualization of the latents at iteration 9000:



Here we have colored the latent space by the digit label, and we can see which labels are close to each other in the latent space. For example, it seems that 4, 7 and 9 are close to each other.

Now, when we are generating arrays, `EIR` will save some of the generated arrays (as well as the corresponding inputs) during training under the `results/samples/<iteration>` folders (the sampling is configurable by the sampling configuration in the output config). We can load these numpy arrays and visualize them.

Here is a comparison of generated images at iteration 500:



And at iteration 9000, we can observe the improvements in generation:


Iteration 9000
Input
Generated

C - Augmenting Our Autoencoder With More Data

In this section, we will explore how to augment our MNIST digit-generating autoencoder with additional data. Specifically, we will add the MNIST labels to the autoencoder, which will allow us to conditionally generate images of specific digits.

The global config remains the same as in the previous section:

Listing 102: globals.yaml

```yaml
output_folder: eir_tutorials/tutorial_runs/d_array_output/01_array_mnist_generation
checkpoint_interval: 1000
sample_interval: 1000
valid_size: 1000
batch_size: 64
n_epochs: 10
device: "cpu"
optimizer: adabelief
lr: 0.001
memory_dataset: true
latent_sampling:
  layers_to_sample:
    - "fusion_modules.computed.fusion_modules.fusion.1.0"
```

The input configuration now includes additional files to represent the augmented data:

Listing 103: input_mnist_array_with_label.yaml

```yaml
input_info:
  input_source: "eir_tutorials/d_array_output/01_array_mnist_generation/data/mnist_npy"
  input_name: mnist
  input_type: array

input_type_info:
  normalization: channel
  adaptive_normalization_max_samples: 10000
  modality_dropout_rate: 0.2

model_config:
  model_type: lcl
  model_init_config:
    kernel_width: 8
    attention_inclusion_cutoff: 128
```

Note: Here we see another new option, `modality_dropout_rate`, this will randomly drop out modalities during training, which can be useful for training models that can handle missing modalities.

Listing 104: input_mnist_label.yaml

```yaml
input_info:
  input_source: "eir_tutorials/d_array_output/01_array_mnist_generation/data/mnist_
↪labels.csv"
  input_name: mnist_label
```

(continues on next page)

(continued from previous page)

```
  input_type: tabular

input_type_info:
  input_cat_columns:
    - "CLASS"
```

The output configuration has also been modified to accommodate the augmented data:

Listing 105: output_with_label.yaml

```
output_info:
  output_source: "eir_tutorials/d_array_output/01_array_mnist_generation/data/mnist_npy"
  output_name: mnist_output
  output_type: array

output_type_info:
  normalization: channel
  adaptive_normalization_max_samples: 10000

model_config:
  model_type: lcl
  model_init_config:
    channel_exp_base: 4

sampling_config:
  manual_inputs:
    - "mnist_label":
        "CLASS": "0"

    - "mnist_label":
        "CLASS": "0"

    - "mnist_label":
        "CLASS": "5"

    - "mnist_label":
        "CLASS": "5"
```

Note: Notice here we are using some manual inputs in the sampling configuration, which will allow us to generate images of specific digits.

We can run the following command to start training the augmented autoencoder:

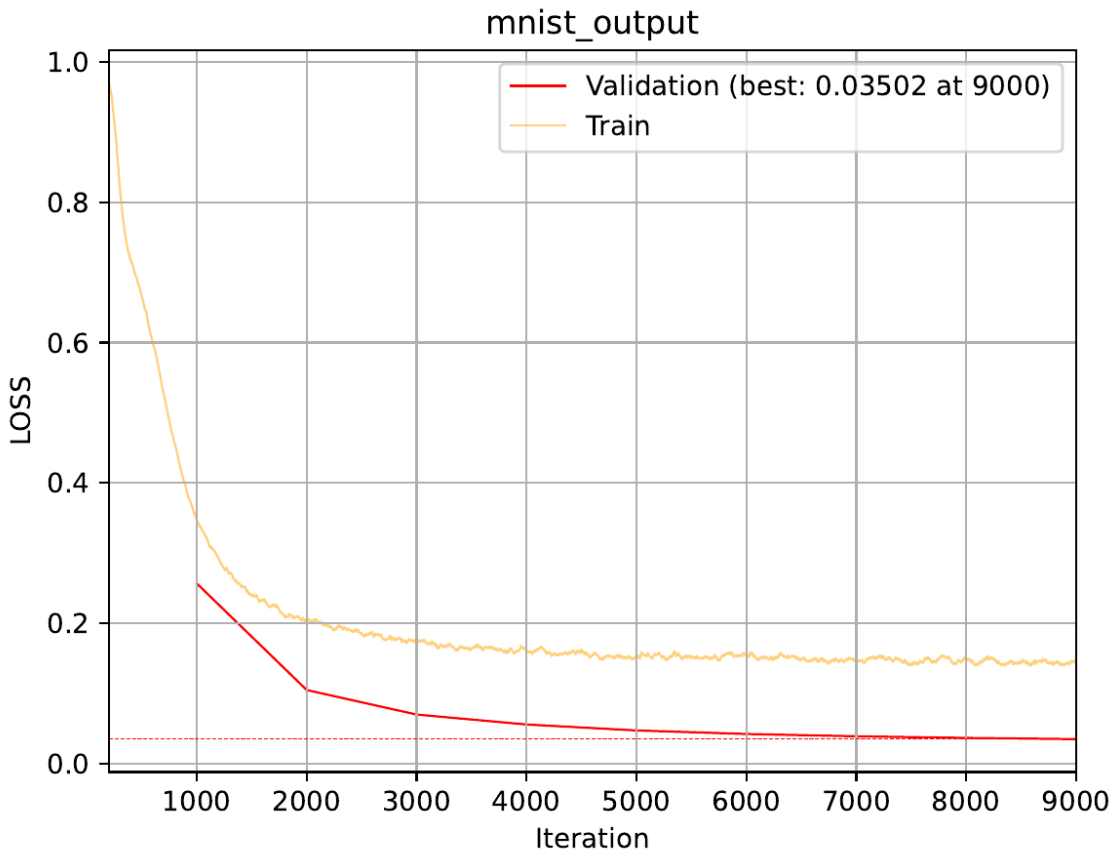
```
eirtrain \
--global_configs eir_tutorials/d_array_output/01_array_mnist_generation/conf/globals.
↪yaml \
--input_configs eir_tutorials/d_array_output/01_array_mnist_generation/conf/input_mnist_
↪array_with_label.yaml eir_tutorials/d_array_output/01_array_mnist_generation/conf/
↪input_mnist_label.yaml \
--output_configs eir_tutorials/d_array_output/01_array_mnist_generation/conf/output_with_
↪label.yaml \
```

(continues on next page)

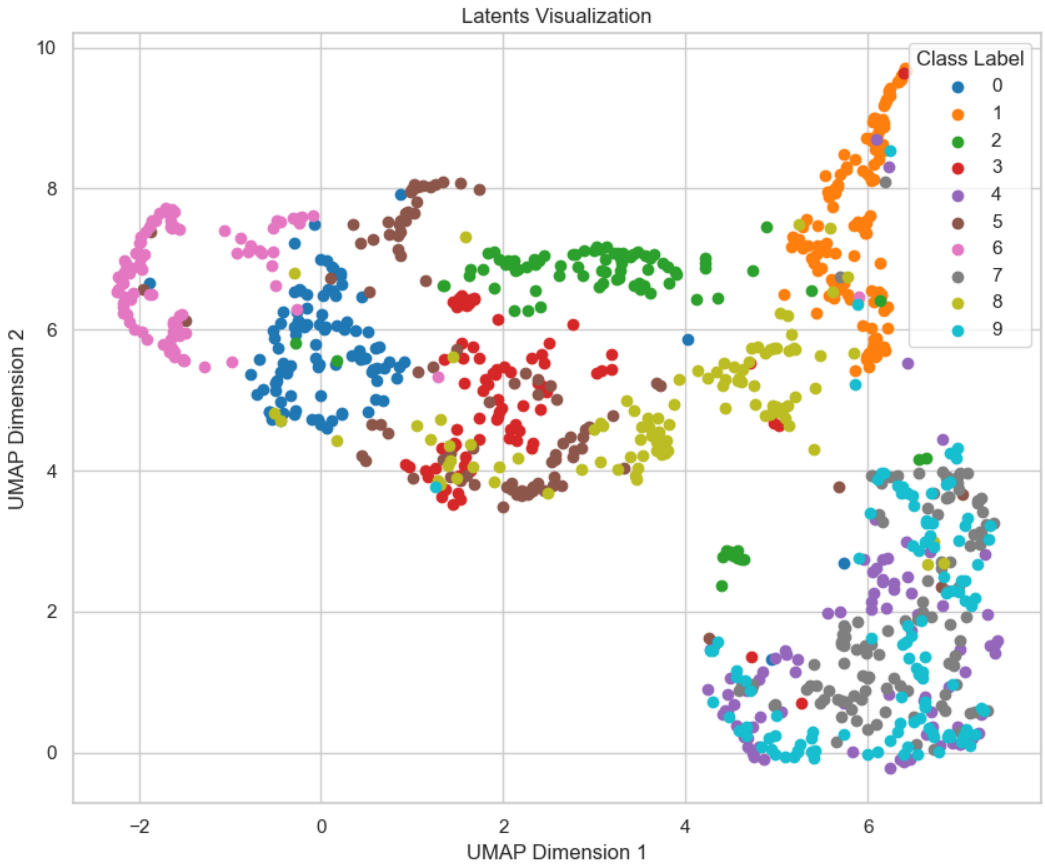
(continued from previous page)

```
--globals.output_folder=eir_tutorials/tutorial_runs/d_array_output/02_array_mnist_
↪generation_with_labels
```

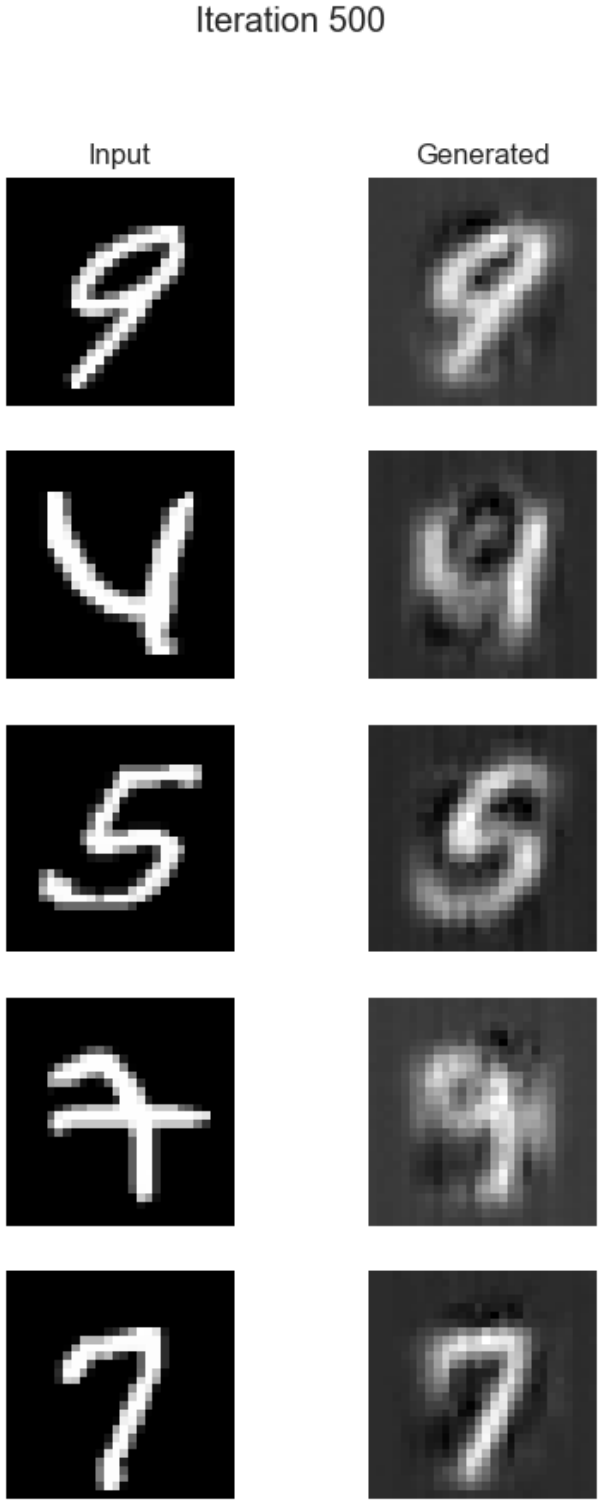
I got the following results:



Here is a visualization of the latent space:



Here is a comparison of generated images at iteration 500 and 9000:

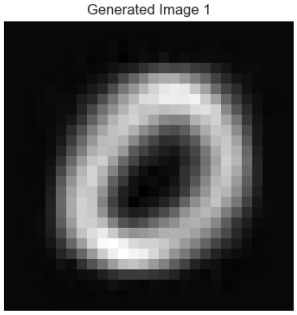

Iteration 500
Input
Generated

Now, since we added those manual inputs earlier, they are also saved in the `sample` folders (under `manual`), and we can visualize them:

Input for Image 1

{'CLASS': '0'}

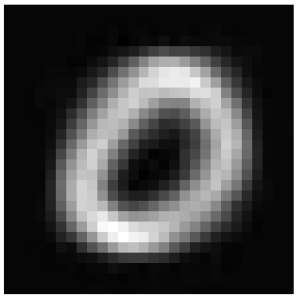
Generated Image 1



Input for Image 2

{'CLASS': '0'}

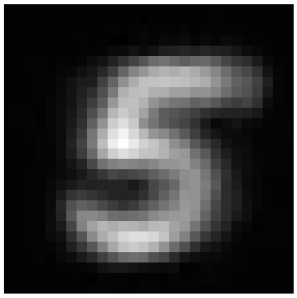
Generated Image 2



Input for Image 3

{'CLASS': '5'}

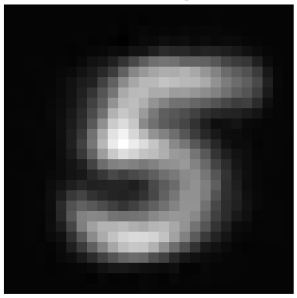
Generated Image 3



Input for Image 4

{'CLASS': '5'}

Generated Image 4



So indeed we can see, in the absence of the actual image to encode, the model uses the class label to generate the respective digit. While not immediately obvious, the generated images of the same class are not completely identical (although they are extremely similar), due to some stochasticity injected into the model.

D - Serving

In this final section, we demonstrate serving our trained model for MNIST array generation as a web service and interacting with it using HTTP requests.

Starting the Web Service

To serve the model, use the following command:

```
eirserve --model-path [MODEL_PATH]
```

Replace *[MODEL_PATH]* with the actual path to your trained model. This command initiates a web service that listens for incoming requests.

Here is an example of the command:

```
eirserve \
--model-path eir_tutorials/tutorial_runs/d_array_output/01_array_mnist_generation/saved_
↪models/01_array_mnist_generation_model_9000_perf-average=0.9688.pt
```

Sending Requests

With the server running, we can now send requests with MNIST data arrays. The data arrays are encoded in base64 before sending.

Here's an example Python function demonstrating this process:

```
import requests
import numpy as np
import base64

def encode_array_to_base64(file_path: str) -> str:
    array_np = np.load(file_path)
    array_bytes = array_np.tobytes()
    return base64.b64encode(array_bytes).decode('utf-8')

def send_request(url: str, payload: dict):
    response = requests.post(url, json=payload)
    return response.json()

payload = {
    "mnist": encode_array_to_base64("path/to/mnist_array.npy")
}

response = send_request('http://localhost:8000/predict', payload)
print(response)
```

Retrieving Array Information

You can get information about the array type and shape by sending a GET request to the */info* endpoint:

```
curl -X 'GET' \\
  'http://localhost:8000/info' \\
  -H 'accept: application/json'
```

This request will return details about the expected array input and output formats, such as type, shape, and data type.

Decoding and Processing the Response

After receiving a response, you can decode the base64 encoded array, reshape it, and cast it to the appropriate dtype using the information obtained from the `/info` endpoint:

```
def decode_array_from_base64(encoded_array: str, shape: tuple):
    array_bytes = base64.b64decode(encoded_array)
    return np.frombuffer(array_bytes, dtype=np.float32).reshape(shape)

array_np = decode_array_from_base64(
    response['mnist_output'], shape=(28, 28)
)
```

Important: While the original output arrays can be of any dtype, and that information is provided in the `/info` endpoint, the response output arrays are always of dtype `float32`, which is the output dtype of the model itself. The model output is then un-normalized using the training set statistics (assuming normalization was used during training).

For example, since these are images originally in `uint8` format, we can process the response arrays as follows:

```
from PIL import Image

array_np = (array_np - array_np.min()) / (array_np.max() - array_np.min())
array_np = (array_np * 255).astype(np.uint8)

image = Image.fromarray(array_np)
image.show()
```

Analyzing Responses

After sending requests to the served model, the responses can be analyzed.

Listing 106: predictions.json

```
[
    {
        "request": {
            "mnist": "eir_tutorials/d_array_output/01_array_mnist_generation/data/mnist_
↪npy/10001.npy"
        },
        "response": {
```

(continues on next page)

(continued from previous page)

```
            "result": {
                "mnist_output":
↪"AJATvgCEPj0A8EO8AADVugAYmjwAADS5AGDWuwAAcDsAqJK8AHCQPQBUIr0AkGy8AEBCvACA9roAgFM7AIDaOgBATLsAsMU8AGCM
↪ADPyvmAJAcCAqyI/AB44PiDpzz8AfVc/QNyeP0DXhj+Ahyg/
↪gCrbPgDMgr0AQKk7AFB1vADAczsAwCk8AChnPQAgOz0AvBK+AMxYPQCcmT0AAIG6wKs/
↪PwCygb0AkUk+QENjvwB5G7+Aoqu+8L7eP6DY6r9A/lbAoPOXP8Bo27/g3WvASOG1wBi2IsFAn/
↪nAUA7VwHAlv8Cwok3AAFBNvQAoyD4AgFQ9AFTivQAcxD0AKAQ9AHcTvgCERz0AcO28AIiovABzYj6AMCa/
↪wDUFP4DGhz6Qn27AAICNvzQw80CA05G/RM4xwSCBacBgUrm/
↪EuQBQQCe5r34XJDAgFIZwABOgT2A9LzAwISVwIAjDT+Adre+AGjSPcAFXz8AvuI9AChnPQAuFL4ACP88AMjVPQCMqb0A9ZC+QHWdvu
↪Er+ozk5AAJ11QKCrKkHMYiZBAMVgwAB8db+A5wbAQCRzQAD+370crrLBS2OUQtJBLkNOQDtD+mvHQYBzxj4A/
↪Go+oHhEwACizb+4ugFAAAk+PgAqsD0Apxi+APAPPADTcj4ABe8+ANfgPxClB8BA8BrAsFIvQOAlnEAQ2BVAuHjNQCBynb+AOxo/
↪cPwGwTxQPcFA92vA1j9fQl1cKkMlxJFD9vhoQ3pUZUJwR33AkK6sP2DhW8CAhcq/
↪sAauPwCA4rsATCs9ACgbvgAMPD0AIsw9wOtnP8B/oj/gboi/UHVMwPCePcA4kjpAgjwCQUCmDMHg8nnAiNE/
↪QIiRW8EMZdpAnEqmQglYPUMmbIFDPz9zQyeiFEP3nDBCRAm2QXCfcUBAVN0/
↪AFDFvABtAL6AkaC+AIAQPQDFNr4AIpU9gKKnvvA9A0CAZoo/
↪QLg3vwAa9T0AiCg9QNYBP4xOBUG0DxPB+EPtQHQcpkCqvNTBLEp7QjGlM0Ocbn9D/
↪dWBQ3EII0MiNVpCUHidQmcICUPtM4RCACTKPWCB3T8gjui/
↪gFWbvgD0ST0ANya+QDM5vwDs1L1oemlAEMTlP4AWNcDAWBs/gCc0v6QTAEH45QtAMAPhwIAzTj8ArpDBk63CQU/
↪EOEPgoIVDDpBzQ4qaC0N4J9ZAcPJgQr5dU0MYFIRDRtUZQ9BOKsAArWU/
↪ADtLPgBalr4AiNY9AEUOvoDjFL8AkA0/qH5CQABazT5A/
↪FXAYGm6P4BdLMCwdgVA4IK7v3DsmcD0GgbB2EWtwF4Hu0LGJWpD/xB9QyZL2UJAhFG/
↪ij34QSe5IENSmoFDz2qJQ4VGEEO8IpxAANBSPADLgT7Alg8/
↪AF6cvgAHDr4AILU7AI0PPoAlsT7g+zTAIKjRvwCERj2QTwRB4McywKBd0MDQgUPAWrmBwcpamkLZwk1D676DQ+PcOUOotYVBs0QgQ
↪wDf0vwAA6j1Q0WfAiGP4QIAKZb9AEkG/
↪5KoWwYwBFUEA5gNDtV96Q9oubUOTtgtDU1+dQipuMEMjWoZDd8uOQ2wEJ0NeJotC+Lq9wCD5rr8ATqq/
↪4PKjv+CIgz+A0O8+AKUYvgCgaDwA+6K+AC7vvQC7ZD5gjqu/
↪4KzOv9AzBUAszNdAUPMRwLwJWcHAQnlBMxERQ5NabENe7TFDYIIXQyWNakPINYtD2BVvQ9J0NkNUpA5CfjhTQThv3sCAEs2+4APMvu
↪AI64PQAvE74AoPM8AJYwvoCuuT6AEAxAoDzrvwBdGj/goPy/
↪PHu+QNCOnUByQ4XBIIX8QcyZQkM4zYVDDEloQyPCXkN8kYdD1VmBQ6JhJENxDWFCiKStwCByob9wwEBAwJ8IvwD82L0A3ZY/
↪APCxPQCisD0AIyC+APRMPQA3Fr4A5EQ9WNYsQIAOdL8A5JW9+PNXQIAAzUC+mNJBxlS4wTC840EAxT9D6zqBQxeHgUNAqoVD5hJnQ
↪AIxXQAAihr6Au/U+OOUFQACTqz4AzEE9AMQUvgBUiz0A4HW+AMwvv4CIkD4wgcg/IM2cvwD6Jz9o25fA/
↪FDFQOAMRcE/
↪mINCsbpaQ9WXiEMSkYRDnUxrQzVSAkM6JeRBLFFswSr1tMHAUhXAAAqMP2D2AsCAVY4+4LKdPwgDFEAAaLI+AP68PQDtFL4AkA49g
↪hQCEOAwmNATP4GwfKigkHAGJjAmKzUwNqpKEHQUak/
↪AOwxP4hkBkBgmypAQEJVPwBEgD0ACya+AED2PIAFQr9AiDfA8FGWwHCPsD8AKMy+cMrpP5RANEIA8P5C3yxcQ+TtjkOwRm1DmhdqQ
↪g+nVBGCYhQADHnr7wqrE/8N/PP4CavT4A+Go9ACYavgD48jyA5ce/ILTbv7iLjcAAmM4/AN/
↪WvrRwQEEHRQhDYu9gQ+2xhkMGt2ND9L76QsgA3ULJHTdD8FhlQySNu0Jcr1PBnNcCwYjgz0CwG9rAQPCWPwAKST+AWIM+wJ6FP8D1
↪IKTJP5BaZMCkxgdCOHxaQ1dIgkMxg1VDxRKDQgiwzMF0IGjB+qBmQoJvg0ORSzxDdz2MQeCJW8AQWZPAADVjvySZCsFQXQHAAJ6/
↪PZB24T9ALlM/ABqcPQD+gD0AdRe+ABzFPcBn0L8A0CA/AC4bPgCaiz7AJqK/
↪UCS6Ql6gg0NtxoFDfAUOQ2AUr78AcmvAZL5EQr3/
↪GENvH5BDG0tUQ8plAEKAgpC+iMK2QAqCekHAugDAgHCOPoD2KkAoBkFAAAFuvwAJXz4AsNc9AIoivgC3Cz4ASAS9AJ0EvgDjFj+oay
↪APoA6yEIX4oZD1N2BQxQnQkOstjJDvjhXQ/
↪WHaENyX4VDb2iHQzEvSUPohptBcFpmwMBZrUDo5QpAYB4VwMD7W0CITp5AoHvXP4Bh0r8A1Gs+ABCFPQAwKr4A8B+9YDiLPwBxsL5g
↪TCgFCrRVQQ4m0g0NUFntDgqmDQ+5BgUPpyIFD6ex3Q9jsPUOgXtpCQFBuP0BeHz8A830+IMhKwHDGbMCAInA/
↪HEerQAA6mD/ADVa/AMrRPQBcCT0AJBK+AOwMPYCeyT7ADhU/
↪cCZZwNj2M0DQ4E1ASDXKwDTW0ELhbktDBIZfQ9ltcEOdj1ZD+Bg7Q4aQxkK+cfVBcJ5HwIC/58BAK46/
↪AK+uvuioPEDAsxo/gGYqP8iYMEAAB1Y/ADgOvYAz1r4AqDs9AJkQvgAQ/
↪jwAqyc+QBsGPwCYFL8AlAc+YCcvQAAdoj84BdlAQCc3P0aKEEGgmg7BpIYaQdAOJ0Ak15hAeAzkwCz+g0CUkw9BuF3YwABEuD4Q3U
↪AOJTvgAQWz0AAQA+AGQaPQArFb4A0Ow8AGi4PADoGz0AexQ/wA80v8ANGT8AoLA/
↪sJ0JQKCuVsCA+gjA4CI6wEAxV78AvQg/QKyePwTigEDYWdTAAIFwQHDPvz9AZRE/YNzHv6Cb9T8AmAI9AC/
↪OPgC0MD0AsM48AHIGPgBsDD0Auxi+AIgTPQAIn7wAAGQ6AHBfvADvGr+AAoa+IO2NP4DZyz6A9VfAsC/
↪aP8Bhh0Cg+/G/cP++wEBzWUCAc4NA8BGzwOB+ir+wQAVAADx9vgCXIj8AdfQ/
```

(continues on next page)

(continued from previous page)


```
↪ACCpuwBwljwAAAO6AMABPQCGiD0AEOM8AIgVvgAsHj0AoNu7AICuugBAODsAeIC8ALAovACMEz4ARWo+QJYnwACm7j2o4jVAYPjyvu
↪4MyqP4BB/D4AOMW+ALSgvQDmOD+A2w5AAOCSuwCYZL0AYJQ7AIA2OwC4xjwAVJY9AAj0PA=="
            }
        }
    },
    {
        "request": {
            "mnist": "eir_tutorials/d_array_output/01_array_mnist_generation/data/mnist_
↪npy/50496.npy"
        },
        "response": {
            "result": {
                "mnist_output":
↪"AH6AvQCIt7wANjm+AEAvOwBwyDwAADa6APAuvABAcjsAKI28AHjFPABQWL0AsLW8AAAOOwCAwDoAWK08AACROwCAQ7sACtO9AMDc
↪IOzjPyA5/z8AanU/AGvKPgC6YL4gk4I/AKSTvQBw/
↪DwA4Lk7AMBoOwBAYDsAPAc+AM6bPQCQED4A7IS9ADDTvAADFL4AoOA7AE0mPwD6gb2AhaC+IKi3PxAVR0GI+Y9BxtjnQTLrIEJmGT
↪tQcwbEEL75/
↪pByfiVQYiNk8DIMZPAQBbQvwDwqLwAu52+kCyzP8AQJz8AdP09AIjWvABE1z0ADQc+ABiRvQDwFrwABje+AEgYPYCLqz6AAnS/
↪wBUTP/zeMEH8xERClt+dQgy0+UK/QB1DXsMZQyjO1EIdAihDNCYOQ9pKiEK8IyxBQGMkv9Q0+0AAMfk/
↪YH9XwKB53T8AtQo/
↪AFE+v8BpPj8ALPA9ADwaPgCQW70ASIa8ACMLvgAE+70AVtu9gMonwCB+NcA920hCol0aQ9vqREPpTW5DrZVpQze7SUNENFhDhOleQ
↪AORzPQBRLT4ASGi9AAhxvQDUHb0AaJG+CIYWQKjqxMCMqd1ApHADQ2AhQENn0SxDZw8DQ/
↪DUGUJkizlBoLBZwELxhUHO3/lCpVxuQ96VeUPi+/
↪xCgPFdQDANpkCwahVAIH8XQMBkjr8A1pu9wHtMP4D+yb4AvxA+AJqfvQDAY7wACRO+QH8Xv7CNH0CQ09rAglcKQvz/
↪JENQAAFDy8kdQs7NrcG8CIHBwGbRwBDDUcBI4MDAzDonQZlXVEKJVERD4qZfQ+jlUkIQmaFAdOqVQCwaikQAAMWc+sK1WQADHnT5AJv
↪APR7vQD4Vb0AmIU8QM8UvyD+hr8APdm+0NmswJDYGkLSMMZCA9FsQuosPkEo65DBZujAQSBsyT/
↪IGM7AVBauQYBkBz+06EBB4ClEQ/79hUNSsulCIGP+P8Bl1L9ANbo/gGC8PoDhbL/AbS+/
↪gI7ivwC6sz0AjJW9AFD5vYAmg74AhJc/
↪gKVhv4Drqb625jpBkATOQXBb4EHMU8dAGH1oQADAwrwA138+mHviwAA4qUGYqOLAMGSAwXzwHUPySGZDDIEKQxAgI0FAx1vBAHixv
↪QK8HPzAmCsAAKO08AGxkvQCpF75A+R8/
↪sArcP6Bfnz+wqR9AAFO1vuDWmr8QetnACBL8QAjZJ0FoLpnAcNgLwMS/LMFIxjlB/
↪ksIQdAy7MDrYyxDrSJ7Q2iQy0JAeWDAgEhXwOA8k0CAngw/kBCgwIBU0L6Ac/W/AM5vvgDcbL1AsAE/
↪0FHOPwDkST9wvNo/gJtlPwD5cD6AC3a/8CSjwMDq8b+Ig7DAWK3KwHZ/IkHgr9+/
↪+K4sQHC4XcE+TZ1BB5NQQ1neakOgvYdCLlUhQQDUOb3AAtS/gPG+vgAPPj4AGD4/
↪AOolvoAMuD4AKpW9AGBovCA9hD8AkrY9AH4SvhAZBsAgnxXAdKEewUrTF0FYNXtAGLr0wOC2nD/A7bq/
↪UBYPwE4BOUHwiu7AgP3AQkXkckOo77lC8GgAQQC2Cj8MFwzBQCIHP8Cf87/
↪Qc+VAoIvnP4Du7j4Aybo+AAh5vQCiDL4AoAQ/gH5iPwCwTz2IJNXAgK5dwBz/
↪CcH42o3AWPWUQBAasMCwnr5AgMOQP0hdEEEy8MLBKVvTQe4iaEMG8WlDyxE6Qtg3+sBIbQRA6HOVwKAOikCYrURA4Nnsv2D4wz8AO
↪YApZwACRYT8IgQZA0DP/P/
↪DSs8CgixPAmBGswfCQkMC4k1VAGLZDwfK3vEI9WndDyygWQ84DC0GwGXtBBmVjQaykjEDQ2dZAmG0BQFwmUsFgzqg/
↪ACEUvgAY2T0AYG29AOxZvQCC/b2A56G/QInZP6A3xD/gXoA/
↪Liw0QXypuUD8wQ3BSE63wFgFwsB8gqhAODPDwHOfFkJ7RjRDBkt4QyuVrUKIEZHALuU+Qegq5cDAwuq/
↪AJRUP8AZfb98H3XBwIjSPwCzcz4ATOo9AJh1vQCymL2A6Za+QIr0v8CIKL/
↪oWTVAgKxtv3CYNsBAtSfAKqA4QbYPhkK4aAFDRL0PQwBGCkNCuztD+7doQwzmGEN6TNFB4OCJv1xKdMEepafBIAOIwECTyr8A0EW9U
↪wEh5v4BztL5soLbBCjYAQpAB/
↪EJPu2pD+M14QyjHg0MvZ2xDPZeKQw6YYUOUjydC8CBJwI04h0H2qgZBiM+bQIMfhEEsAkVBkL7WQHqB00FEPZFAQNVZPwCofj0AQp
↪Q/
↪BPAILTHvyC+v8C4LC7BbC2UQaT4DUPeF1lDC+8KQ0Uiu0LObfdCERglQ9Dhh0OSiXpDepC0QggyVMH0kmLBALwjPfipMME8u0LBIG
↪4sOhwcw28kKy5FhD8hISQyxxLEGQOF1BuuKSQjGgN0MLz29D4YSEQ3XCKUOHbMZB9No5wUNuRsKG9ajCiJuVwhBPnMJlUADCuR4aw
↪AFOVPiYHAkKfe05D+cR0Q8z4VEJIkcHB8HFQQh42GUP2pF5Dik79QurRXUOmqU5DlQ42QxiH9EKUaTtBAFBnPxOphEEWh7xBrCgqQ
↪tCbf0IQ0TyN0PxoGRDT8Q0QwDo2bycVUVCr4Y5Q/
↪PqikNq6nBD2q44Q8IIWUPjF2ND8HpVQ4SaKUPzlSdDpcSbQQCWrD0AIUM+AOKfvQDnCD6Aupg+OFEkQGDKkD/
```


(continues on next page)

(continued from previous page)

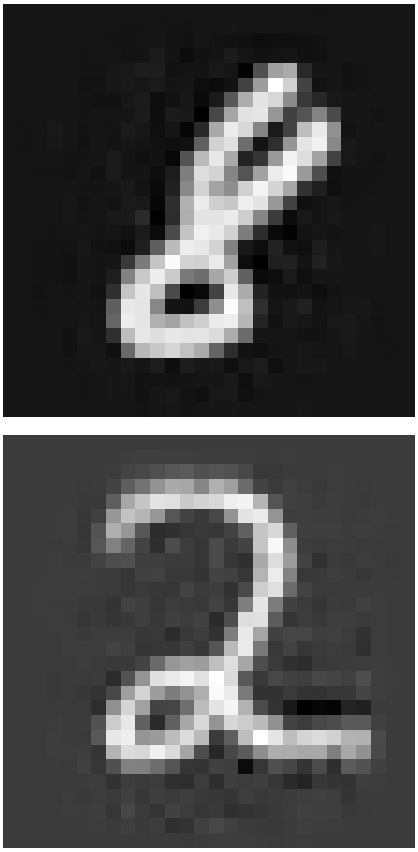

```
↪YfptAmIT8wPo0K0PBFWRDvktdQ+FkgEM5c2JD+7gVQysry0H6Q97BI7+GQV3EOEJsyepCSVkiQwcBIEM/
↪IRpDiuoaQ9AQwUIlRxtDRtT0Qr52lUEAcCM+AKUsPgBilr0Aqu29AKoNPgCqRb4A4EW/
↪hBPyQAjL8cC6FPBBfqTGQjr/xEK//
↪41C7pAOQpjqzMDAQxrBuOUIQSCIykAyVoTCQGXnwESO60EvwoBBBIcTwXHUsUFiV5jBNNw0QjMoBELkcPBAABjhPAB+Aj4A/
↪oC9ALjuvIBp575gPOS/wL+IvwBet73wMLY//NxHwfCgJ0Dg1J6/
↪YMP1P76DhsHA7I3BwMDIP0ZAksEwNmhAAChQP/
↪wFvkCwDrXAav+7wZnOA8LoNbzASBwcwTD9KsGoKp5AgEhhP0DXBL8Adgg+ALyEvQCA1rwA1OK9QP4+v0C5VD+w5gXAkH6oP6BZjb8
↪gEWtvhh0I0DA68zAWxWEQYDfr79UQgLBXmcKQcA72z/wTd7AgDMTwDitcMFkqD7BGNX9wHDkGMAgjNM/
↪7NMOQYCrwUCgWZZAsLogwOBqh78ATAA+AFDSPQApHT4A9H29ALi3vAAVHL4AAII7AKAMvIDf5r4gl5U/
↪YOjavwhgkEDgM1bAiHklQExci8FcvyvBFA/
↪PQKOihUGcAeNA8JcDwNCsVsCgJpzAAESyvmj7p0Bg80FAgIZdPwBw1T0AQDQ7AG8UPgAAqz0AmxE+AJxovQAcAr0Arg6+AACsOgCAr
↪Le+MF4JwAC5qT5A2eY/2DdWQBA13D8ACtw9QBl/v4CjkD7gBfg/
↪AExFPoAauD4AABC4ALBHPQDvDj4AQpM9AGcPPg=="
            }
        }
    },
    {
        "request": {
            "mnist": "eir_tutorials/d_array_output/01_array_mnist_generation/data/mnist_
↪npy/25640.npy"
        },
        "response": {
            "result": {
                "mnist_output":
↪"ADKzvQBYbT0A+U8+AIDUOgDgjTwAAI+6AAAbOgCARLsA8NW8AKTDPQAQTbwAULu8AIAWuwDAersAKOI8AABXOgCAMbsAvKK9AEBjI
↪wANA/AEhSPQBZ+T6AkU4/YMvPP8Aotj9A7Ck/
↪gO9OPwDQLb4AoHK8ALBgPAAADjoA4HK9ACYAvgBACzwA6Ju9ABCCPQCeHT4AwHq7ANONPgCYmrywLMg/
↪IO+yQJS8wkCAvGc/QK5eP/BMVMBU74NAWNjoQHCkVMBgFDVBqFXhwGieAkAi/ChBwHaOPyBK1r8gfaS/
↪AJ1SvjDp6z8Adsi9gKCgvgAoob0AwCW8AACdvQBWhD0A8Fw+AJMuPgCgeT3AaUG/
↪oOWBP9bbLUGA7htAABILPzD0zT/E9VPBINkuQCA4k78AgUi+aLwGQKCUgz+g1a0/AMOpvkAVOj/
↪gzSzAIJdOwABbhj8g4pg/gM+CPoAIDD8AKwO+AOAUPAAYkb0AbFk9gCSbPgCQ3r3AhiS/
↪AMC7v9DTJMAYyoNBpACVQBo+QkHAhrdA4I6WQFDGO8BgbPjAUFIWwQC60L9keYtASEvvwMzgL8GcL8JAGFGCwMDisb8Q+xZAKJoqQ
↪CCsuQJgNusBIMOHA2E1LQThjmcAgifI/
↪cOOpwLhOk8CIyuDAcjhPQfBeysDEk5tAQlMuQdQmMsHu31JBQHRfP76GCkGAJLe+APnpvmg7S0BAAo8/
↪gO3BPkCHPL8AkH48AOKyvQBARj0ASeA+AOUlP7BRIUDwS3jAIA+Lv0CWtcCAlug+QPBJQAhnykAWT6xBNCKpQnMzREMlCINDiYRoQ
↪AIAGvgD+rb0AYKM9AG0xvgCwHj0APjo/
↪ALMbPlVfgEEbji5C4qWIQixf9ULhpDpD+eRXQwwaaUMZ33lDlUWEQ+hPhEPRDmxDBc9vQ9ghOkPW801C4LiRP6ApjL8A3b6/
↪wD8Qv5BwR8AYeBlAQJkQPwBGib0AXPu9ALyQvgDuAb8AsjC+gFa1PpALZsAQGXpCfog0Q420TkOrvXVDPwGAQ+0Ja0MCUIND349jQ
↪0FwUtPB0KBHwFCzmMCY0jFAEJY+QAASAD8AEB48AH7dvUDlBb/gf5s/wNkbP7j/
↪m8CUfCZBs+AWQwP7kUM1uXtDVORsQ6WmZkMBm19DQYo7Q7a73ELCkHZC5qGpQitFGUPuzzFDq591Q9R0gkNkGNlCyMzUQOD5McAANV
↪oNiYP6CJOkDABoY/AIzpvgCUub0AJHm+mHkNQEhSAkDgXjfBWmWSQtXGf0NZ/4hDei4pQ/
↪Y+0EI2MuZCjG6vQgaiTELYvbjAhFZMwXjWJEDAL9JATp6UQtNkV0MCF5hDNspMQ7CCREJAqjXAUO25P5DMrMBQXrE/
↪AEJ2PgBVlD4AVNW9AHcWvnCSJUAAIK68OAYTQL4IDUPL7YhD3BpBQ8FrhUIYwwhBENWfQah3JsHQHmPBUPhvwKB1qD9xgJ9BqHRLw
↪8h0BQYSd/kDgiYw/
↪oDYewHSBHcF4fgdCqmcdQzJjbkO76UdD604fQvjOikDAv0XAgHzXv0Dzlz+Ahdy+ACDeOwCgmL0A+Bs90Pa4P2A4979chMVBFAT4Q
↪vPw+Qd4IHkEoOlhAAHznvaQTMEHMJGvB/PKLQlo7UkMeAHpDQGZDQ8g6SUEwigJAoGJFwAAIXD5QRL0/
↪wJVXvwDAubwAiNC9AIjePEDnIT+gzua/SCv1QMqntUK+OH5D5QBvQzDu/kIsa/
↪hAHoC6wfC1TMAgfjnAAE9NvvoGqUEorejAQBQiwYD1aUKBYFxDDWyGQxBIZEMbzxBC0FV6wAA6GT/A8ts/
↪kLgGQACQDj0A0A68AP6lvQDI+DzA+wA/
↪4COsv8xKB8GHYm9Ca3VpQ+J8lkOF8mNDAhDEQih8uEAMDiLBQNNBwMAeM0AkZz5BPExjwQR2hcEqD4xCUjNaQw5UhUOjoXpDAe2bQ
↪PPzAPDkAACkO+ABQMPQDamL0AgAQ7AHDIPmDnxr+EHQXBuB3lQdzD00KvC3lDmGGXQ261bUNh3wtDmsmJQqzvCcHY947AyMbrQEin
↪gCGiPgAQU7wAOKi9AOjJPACd0T5AouO/AG8MPnaNHkHglow/
```

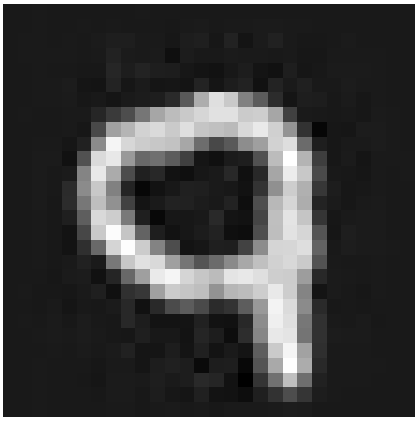

(continues on next page)

(continued from previous page)

```
↪ZL37Qiv7ZkOx0IRD2AR+Q0jSMkNw4fFCmsacQtb710JgWxVD2tQ9Q95WXUMrNHNDfap8Q4XLakPqnZZCAK9jPwBY9bwA+rW9AJosP
↪IHorwBA63T+w21lAoHoswURNf0HI5NBCzCoxQ9ZoikPp/
↪ZFDMjt5Q7xBZUOV9lVDoTOGQ6XdiEMY4YJDxrBvQ0f6bEOcKy5DtDecQahyJUAAsQc+oEaCvwDd+T6AgPq+AMDDPACYlb0AoAc+MI
↪gLplv7BCRcDAFFY/AO1LvgAo0TwA9KG9ABB6PPC6GMAA5ME+KO/
↪FwIAatb4wZxrAsDfXP4KbnUGg2DrBDkSoQSXagkKE/
↪q9CbK6BQs2Ju0EtvgdCsJe0QnsDKkM5lJBDS6yAQ+U4F0ObNbJBoJD7v8CUhr8AkKy/
↪gPLqvgD2N74AqGQ9ADS4vQC2ij3AtZa/
↪qA8pQGACz794JDRAgJHOvsCF1r80rJ5BFFiCQEQaDcFs0wbBwKndwPY3HEGUPyDBNE0vwTDemEFIkBhDXsmNQ5Ftg0OQxflCJgj/
↪QcDBaz8AYoW9MKb4P0BEpL8AXkS+ANibPADk0L0A8pm9ACiKPUgLAkBADq4/OIZIQABbGD5gqb6/wFtbwKBu/
↪kDgtv2/0HciwLSBA8F0LWtBlIuPQGqJg8FQZNNBfqgQQ6XeiENByIZDxtnwQnaR+UEI31lAoNK9P8DKxT8A8xK/
↪AOyUvgDAQ7wAbpa9AAAsPQCC4L2AcrA+gJFDv/Dz/T+AQAu/
↪oAbDP3zyG0GINMPAiC63wEB6hUDgtOTAepYGQaSYC8Fgqo8/
↪UPrfwEqFtkJau2pDtdCRQ2lbNUPcgOdBwBOoP4C1ZD9A0Vc/
↪ALqYvYAjbr8AgN+6AMKXvQDoKT0Aebc+AEdtvkBMYb9AwHm/
↪QJxJwGCLgD8AfD4+eKqzwETPhEBI2QNBCNqoQOB8ocFEPaNAhFq+QELnucGc85tCnhQ/
↪Q8ATmEPUJ0JDDmipQQCgSD4AwVo+AN1JPgAM5T0AdO69AHALvAAap70APB49AJBnPgCIhzwAaRS+IJGnvyhUhsCIRARA6KkqQACoq
↪ADQzA0DEGQECECUBQ7gLABMamQPCbKMCAwcrAdvgLQRxKE8GsaBnBThCEwdIjAEHJTARC7BGIQgu3E0L0rJdAQCwFPwBw6jwAAKC6/
↪IJXSvwD/dT6Ad9o/gJCRv+C4rz8QDA3AQJKuPwB0pT5AtFs/
↪gHkgQPQuiEAw4nRAADBbvAAsL70AAE48AICTOgDoX70A/QS+AACZug=="
            }
        }
    }
]
```

For example, using the approach described above, we can visualize the generated images from the responses:





If you made it this far, thank you for reading! I hope this tutorial was interesting and useful to you!

2.4 Pretraining

2.4.1 01 – Pretraining, Checkpointing and Continued Training

In this tutorial, we will be looking at how to use EIR to create pretrained models, and successively use them for continued training on the same data, as well as partially loading matching layers when changing the model architecture.

Note: This tutorial assumes you are familiar with the basics of EIR, and have gone through previous tutorials. Not required, but recommended.

A - Data

We will be using the same dataset we used in the *03 – Sequence Tutorial: Movie Reviews and Peptides*: the IMDB reviews dataset, and we will be repeating the same task as before, i.e., sentiment classification.

See here for more information about the data. To download the data, use this link.

After downloading the data, the folder structure should look like this:

```
eir_tutorials/e_pretraining/01_checkpointing
├── conf
│   ├── imdb_fusion.yaml
│   ├── imdb_globals.yaml
│   ├── imdb_input.yaml
│   └── imdb_output.yaml
└── data
    └── IMDB
        ├── IMDB_Reviews
        ├── conf
        ├── imdb.vocab
        └── imdb_labels.csv
```

B - Training a Model From Scratch

Training follows the same approach as we have seen on other tutorials, starting with the configurations.

The global config sets the universal parameters for training:

Listing 107: imdb_globals.yaml

```yaml
output_folder: eir_tutorials/tutorial_runs/e_pretraining/01_checkpointing
valid_size: 1024
n_saved_models: 1
checkpoint_interval: 200
plot_skip_steps: 0
sample_interval: 200
memory_dataset: true
dataloader_workers: 0
n_epochs: 5
batch_size: 64
lr: 0.0005
optimizer: "adabelief"
device: "cpu"
```

The input config outlines the IMDB dataset's specific structure:

Listing 108: imdb_input.yaml

```yaml
input_info:
  input_source: eir_tutorials/e_pretraining/01_checkpointing/data/IMDB/IMDB_Reviews
  input_name: captions
  input_type: sequence

input_type_info:
  max_length: 64
  split_on: " "
  tokenizer: null
  sampling_strategy_if_longer: "uniform"

model_config:
  embedding_dim: 64
```

For the output configurations:

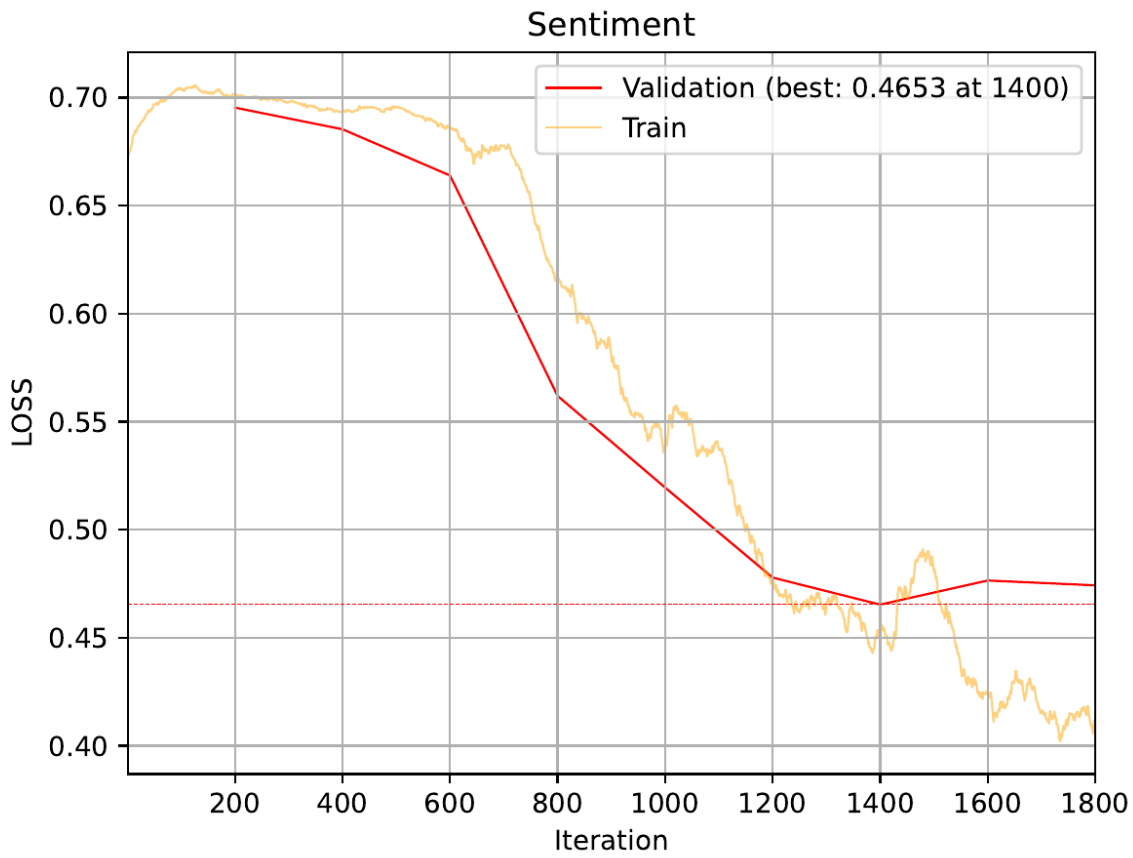
Listing 109: imdb_output.yaml

```yaml
output_info:
  output_source: eir_tutorials/e_pretraining/01_checkpointing/data/IMDB/imdb_labels.csv
  output_name: imdb_output
  output_type: tabular

output_type_info:
  target_cat_columns:
    - Sentiment
```

Here is the command for training:

```
eirtrain \
--global_configs eir_tutorials/e_pretraining/01_checkpointing/conf/imdb_globals.yaml \
--input_configs eir_tutorials/e_pretraining/01_checkpointing/conf/imdb_input.yaml \
--fusion_configs eir_tutorials/e_pretraining/01_checkpointing/conf/imdb_fusion.yaml \
--output_configs eir_tutorials/e_pretraining/01_checkpointing/conf/imdb_output.yaml \
--imdb_globals.output_folder=eir_tutorials/tutorial_runs/e_pretraining/01_checkpointing/
```

Training Results:



So, these training results are nothing too much out of the ordinary, with the training and validation loss both decreasing as training goes on.

C - Continuing Training from a Checkpoint

Often, you might want to resume training from a previously saved checkpoint. This can be especially useful for reasons such as fine-tuning the model on a different dataset, or resuming a long-running training process after interruption. For this, we can use the `pretrained_checkpoint` argument in the global config.

Here is how we can do that:

```
eirtrain \
--global_configs eir_tutorials/e_pretraining/01_checkpointing/conf/imdb_globals.yaml \
--input_configs eir_tutorials/e_pretraining/01_checkpointing/conf/imdb_input.yaml \
```

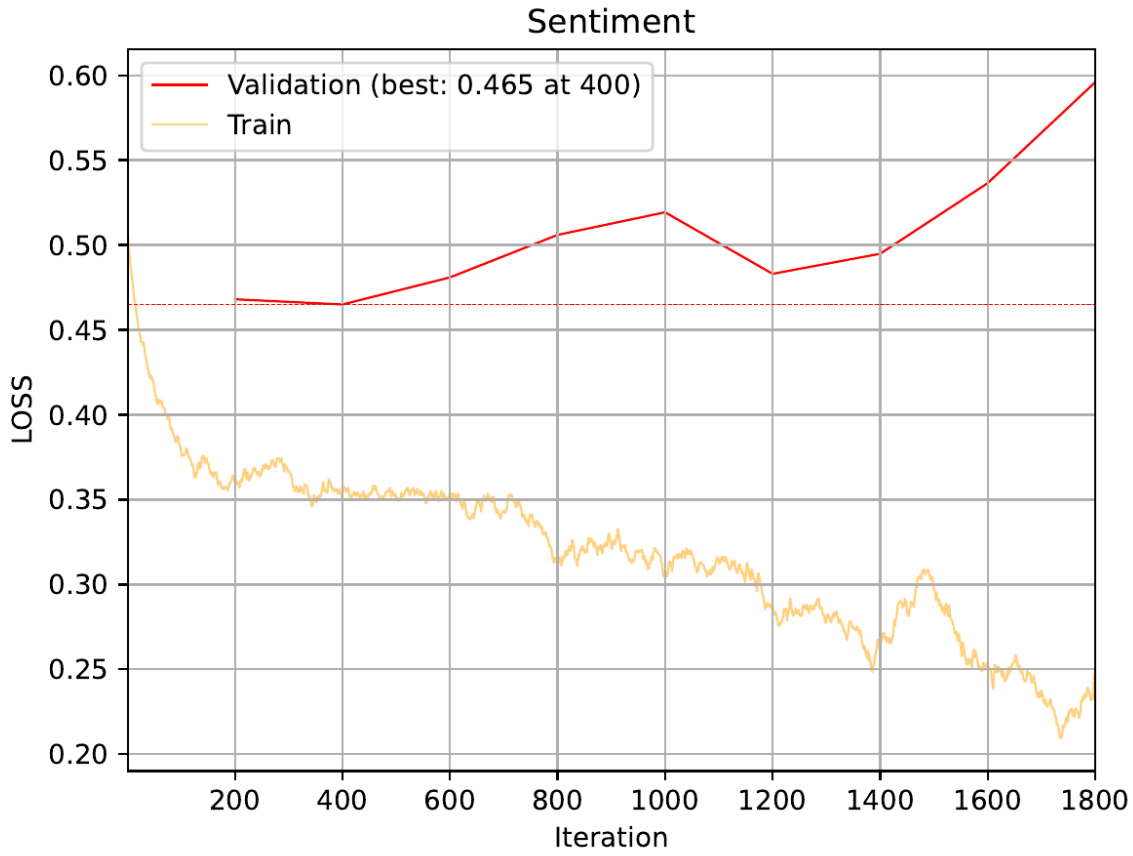
(continues on next page)

(continued from previous page)

```
--fusion_configs eir_tutorials/e_pretraining/01_checkpointing/conf/imdb_fusion.yaml \
--output_configs eir_tutorials/e_pretraining/01_checkpointing/conf/imdb_output.yaml \
--imdb_globals.output_folder=eir_tutorials/tutorial_runs/e_pretraining/01_checkpointing_
↪imdb_from_pretrained_global \
--imdb_globals.pretrained_checkpoint=eir_tutorials/tutorial_runs/e_pretraining/01_
↪checkpointing/saved_models/01_checkpointing_model_1800_perf-average=0.7765.pt
```

Important: The argument points towards a saved model file from a previous experiment, and the loading process relies on some saved data from the previous experiment. Therefore, it will likely not work if you try to load a checkpoint that has been moved from the relative path it was saved in.

Training Results After Continued Training:



From the training curve, it's evident how the model essentially picks up from where it left off as the training loss is already quite low from the start, compared to the previous training from scratch.

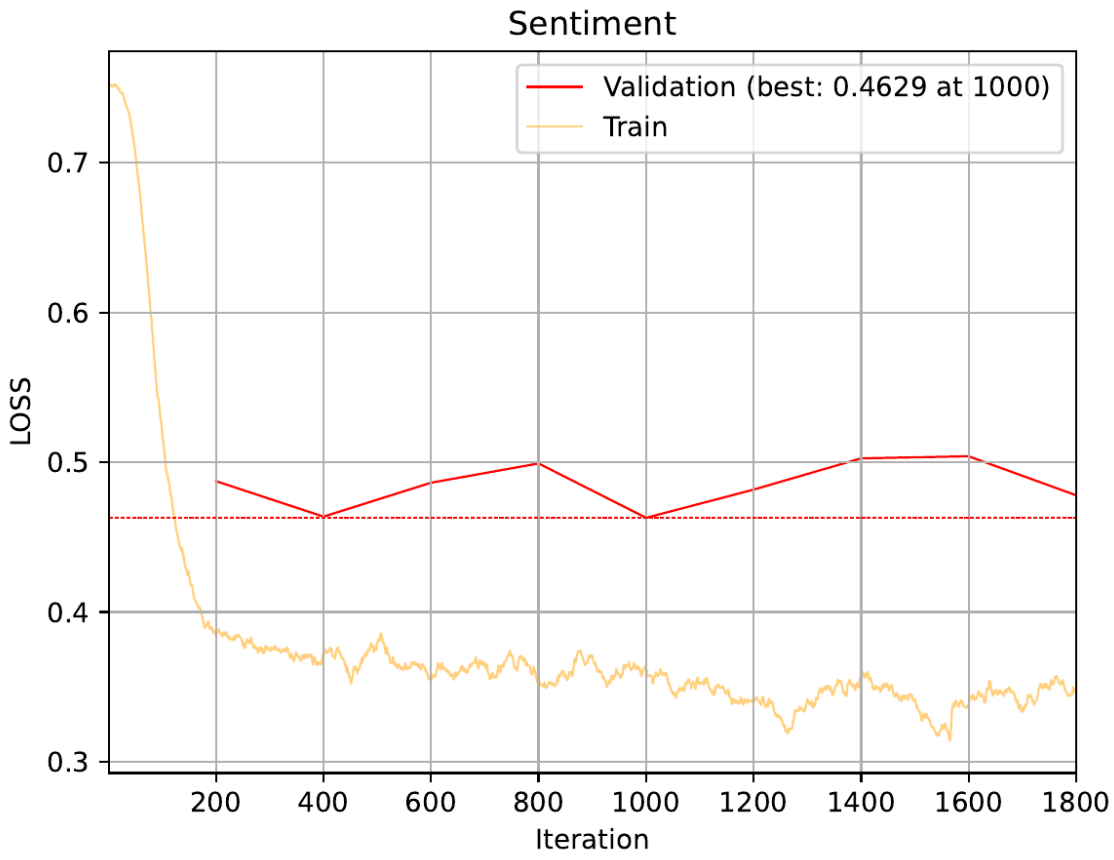
D - Partial Loading of Matching Layers

There are scenarios where you might change the architecture of your model but still want to use the pretrained weights for the layers that match. This can be achieved by setting the `strict_pretrained_loading` argument to `False` in the global config.

Below, we will change the dimension of the fully connected layers in the fusion module, but keep the rest of the model the same.

```
eirtrain \
--global_configs eir_tutorials/e_pretraining/01_checkpointing/conf/imdb_globals.yaml \
--input_configs eir_tutorials/e_pretraining/01_checkpointing/conf/imdb_input.yaml \
--fusion_configs eir_tutorials/e_pretraining/01_checkpointing/conf/imdb_fusion.yaml \
--output_configs eir_tutorials/e_pretraining/01_checkpointing/conf/imdb_output.yaml \
--imdb_globals.output_folder=eir_tutorials/tutorial_runs/e_pretraining/01_checkpointing_
↪imdb_from_pretrained_global_non_strict \
--imdb_fusion.model_config.fc_task_dim=64 \
--imdb_globals.pretrained_checkpoint=eir_tutorials/tutorial_runs/e_pretraining/01_
↪checkpointing/saved_models/01_checkpointing_model_1800_perf-average=0.7765.pt \
--imdb_globals.strict_pretrained_loading=False
```

Results After Partial Loading and Continued Training:



Notice how the training loss starts at a similar value as when training from scratch, but then more quickly decreases to a lower value, indicating that the model can still benefit from the pretrained weights in the unchanged layers.

Thank you for reading!

2.4.2 02 - Creating and Using a Mini Foundation Model

In this tutorial, we will explore how to create custom foundation models using `EIR`. Here we use the term "foundation model" as a fancy way of saying we pretrain a model for one task, and then use it or parts of it as a building block for other tasks.

We'll be working with three different datasets —IMDB reviews, COCO 2017 images, and CIFAR-10 images.

The overall goal is as follows:

1. Train a mini-foundation model for image captioning, which includes an image and text encoder (feature extractors), and a text decoder (output module).
2. Use the text encoder part from the mini-foundation model to train a sentiment analysis model on IMDB reviews.
3. Use the image encoder part from the mini-foundation model to train an image classification model on CIFAR-10.

A - Data

For this tutorial, we will use datasets from three different domains:

1. Text Data: IMDB Reviews - More information can be found here.
2. Image Data: COCO 2017 - Used mainly for image-to-text tasks like image captioning. More details can be found at the COCO 2017 dataset.
3. Image Data: CIFAR-10 - A dataset of 60,000 32x32 color images in 10 different classes. Useful for object recognition tasks. Learn more here.

You can download all datasets for this tutorial from the following link.

After downloading the data, your folder structure should be organized similarly to the following (the config files we will create as we go along the tutorial):

```
eir_tutorials/e_pretraining/02_mini_foundation
├── conf
│   ├── cifar
│   │   ├── cifar_fusion.yaml
│   │   ├── cifar_globals.yaml
│   │   ├── cifar_input.yaml
│   │   └── cifar_output.yaml
│   ├── fusion.yaml
│   ├── globals.yaml
│   ├── imdb
│   │   ├── imdb_fusion.yaml
│   │   ├── imdb_globals.yaml
│   │   ├── imdb_input.yaml
│   │   └── imdb_output.yaml
│   ├── inputs_image_array_cnn.yaml
│   ├── inputs_sequence.yaml
│   └── output_sequence.yaml
└── data
    ├── 02_mini_foundation
    │   ├── configs
    │   ├── logging_history.log
```

(continues on next page)

(continued from previous page)

```
    │   ├── meta
    │   ├── model_info.txt
    │   ├── results
    │   ├── saved_models
    │   ├── serializations
    │   ├── tensorboard_logs
    │   ├── train_average_history.log
    │   ├── training_curve_LOSS-AVERAGE.pdf
    │   ├── training_curve_PERF-AVERAGE.pdf
    │   └── validation_average_history.log
    ├── CIFAR10
    │   ├── images
    │   └── images_classes.csv
    ├── IMDB
    │   ├── imdb_labels.csv
    │   └── imdb_reviews.csv
    ├── image_captioning
    │   ├── captions.csv
    │   └── images
    └── vocab.txt
```

Notice how in the downloaded data, we actually include a `02_mini_foundation` experiment. This is so that you do not have to train the entire model from scratch, and also shows how one can share pre-trained models with others.

B - Training a Mini Foundation Model

Important: As mentioned above, you can download the pre-trained model for this tutorial and skip this section. However, if you want to train the model yourself, you can follow the steps below.

Here, we will show the training of a model for image captioning, similar to what we did in *03 - Image to Sequence: Image Captioning*, where the model uses both an image and text input to generate a caption for the image.

The global configuration establishes the foundational settings for training:

Listing 110: globals.yaml

```yaml
output_folder: eir_tutorials/tutorial_runs/e_pretraining/02_mini_foundation
valid_size: 1024
n_saved_models: 1
checkpoint_interval: 500
plot_skip_steps: 200
sample_interval: 500
memory_dataset: true
dataloader_workers: 0
n_epochs: 20
batch_size: 256
lr: 0.0005
optimizer: "adabelief"
device: "mps"
```

Listing 111: inputs_sequence.yaml

```yaml
input_info:
  input_source: eir_tutorials/e_pretraining/02_mini_foundation/data/image_captioning/
↪captions.csv
  input_name: text
  input_type: sequence

input_type_info:
  max_length: 128
  split_on: ""
  sampling_strategy_if_longer: "uniform"
  vocab_file: eir_tutorials/e_pretraining/02_mini_foundation/data/vocab.txt
  modality_dropout_rate: 0.1

model_config:
  embedding_dim: 64
```

Listing 112: inputs_image_array_cnn.yaml

```yaml
input_info:
  input_source: eir_tutorials/e_pretraining/02_mini_foundation/data/image_captioning/
↪images
  input_name: image_input
  input_type: image

model_config:
  model_type: cnn
  model_init_config:
    channel_exp_base: 5
    kernel_width: 2
    down_stride_width: 2
    kernel_height: 2
    down_stride_height: 2
```

Listing 113: fusion.yaml

```yaml
model_type: "pass-through"
```

Listing 114: outputs.yaml

```yaml
output_info:
  output_source: eir_tutorials/e_pretraining/02_mini_foundation/data/image_captioning/
↪captions.csv
  output_name: text
  output_type: sequence

output_type_info:
  max_length: 128
  split_on: ""
  sampling_strategy_if_longer: "uniform"
  vocab_file: eir_tutorials/e_pretraining/02_mini_foundation/data/vocab.txt
```

(continues on next page)

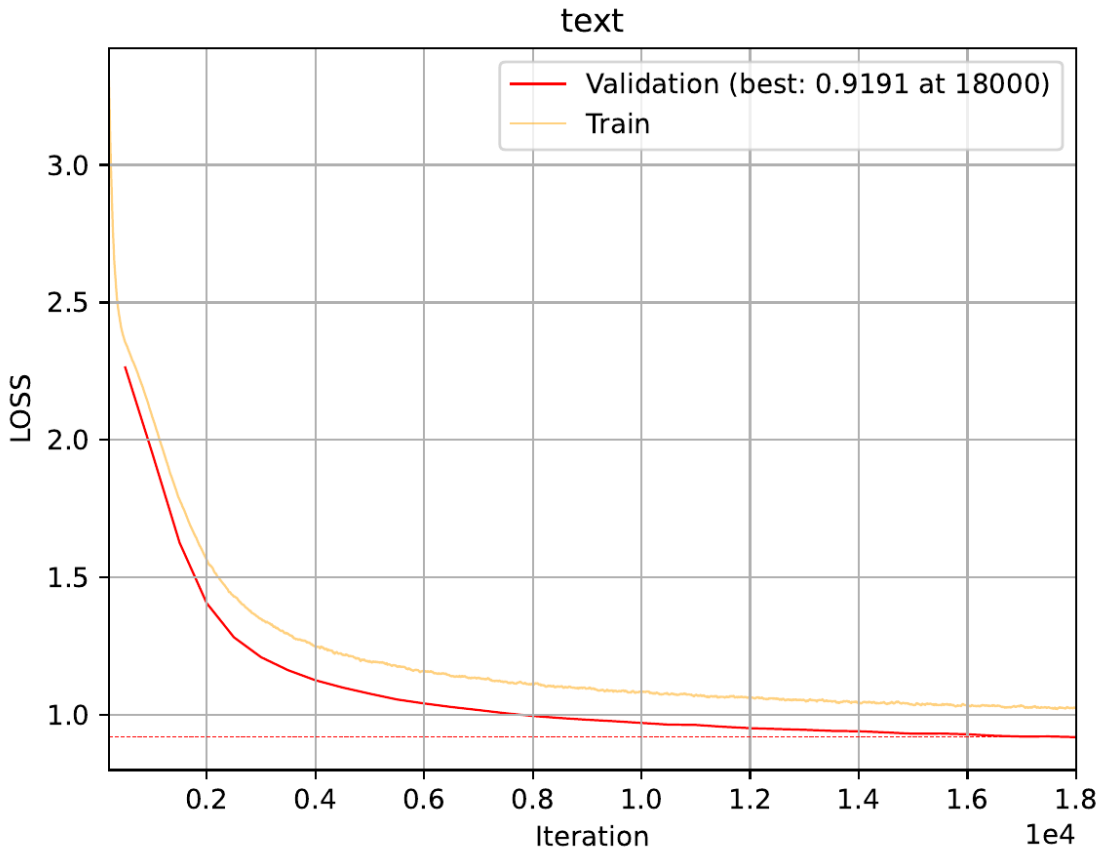
(continued from previous page)

```
sampling_config:
  generated_sequence_length: 64
  n_eval_inputs: 10
```

To train, we use the following command:

```
eirtrain \
--global_configs eir_tutorials/e_pretraining/02_mini_foundation/conf/globals.yaml \
--input_configs eir_tutorials/e_pretraining/02_mini_foundation/conf/inputs_image_array_
↪cnn.yaml eir_tutorials/e_pretraining/02_mini_foundation/conf/inputs_sequence.yaml \
--fusion_configs eir_tutorials/e_pretraining/02_mini_foundation/conf/fusion.yaml \
--output_configs eir_tutorials/e_pretraining/02_mini_foundation/conf/output_sequence.
↪yaml \
--globals.output_folder=eir_tutorials/tutorial_runs/e_pretraining/02_mini_foundation
```

Here we can see the training curve for the mini foundation model:



Now, given that we have either downloaded or trained the mini foundation model, we can use it to train other models.

C - Establishing an IMDB Baseline

Before using the mini foundation model, let's first establish a baseline by training a model from scratch to perform sentiment analysis on IMDB reviews.

Here are the configurations:

Listing 115: imdb_globals.yaml

```yaml
output_folder: eir_tutorials/tutorial_runs/e_pretraining/02_mini_foundation
valid_size: 1024
n_saved_models: 1
checkpoint_interval: 100
plot_skip_steps: 0
sample_interval: 100
memory_dataset: true
dataloader_workers: 0
n_epochs: 20
batch_size: 64
lr: 0.0005
optimizer: "adabelief"
device: "cpu"
```

Listing 116: imdb_input.yaml

```yaml
input_info:
  input_source: eir_tutorials/e_pretraining/02_mini_foundation/data/IMDB/imdb_reviews.csv
  input_name: text
  input_type: sequence

input_type_info:
  max_length: 128
  split_on: ""
  sampling_strategy_if_longer: "uniform"
  vocab_file: eir_tutorials/e_pretraining/02_mini_foundation/data/vocab.txt

model_config:
  embedding_dim: 64
```

Listing 117: imdb_output.yaml

```yaml
output_info:
  output_source: eir_tutorials/e_pretraining/02_mini_foundation/data/IMDB/imdb_labels.csv
  output_name: imdb_output
  output_type: tabular

output_type_info:
  target_cat_columns:
    - Sentiment
```

To kick off the training for IMDB from scratch, run the following command:

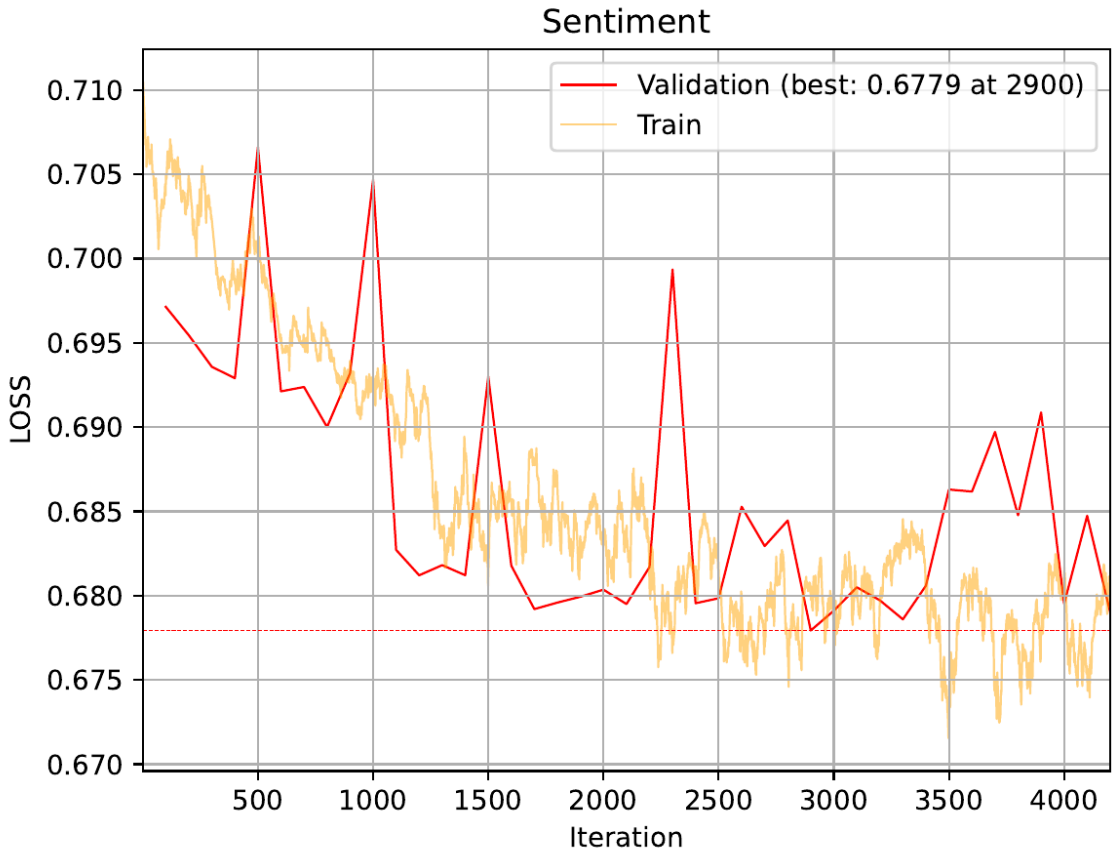
```
eirtrain \
--global_configs eir_tutorials/e_pretraining/02_mini_foundation/conf/imdb/imdb_globals.
```

(continues on next page)

(continued from previous page)

```
↪yaml \
--input_configs eir_tutorials/e_pretraining/02_mini_foundation/conf/imdb/imdb_input.yaml␣
↪\
--fusion_configs eir_tutorials/e_pretraining/02_mini_foundation/conf/imdb/imdb_fusion.
↪yaml \
--output_configs eir_tutorials/e_pretraining/02_mini_foundation/conf/imdb/imdb_output.
↪yaml \
--imdb_globals.output_folder=eir_tutorials/tutorial_runs/e_pretraining/02_mini_
↪foundation_imdb_from_scratch
```

The performance can be evaluated through these generated plots:



This serves as our baseline, which we'll aim to improve in the next section by using the mini foundation model.

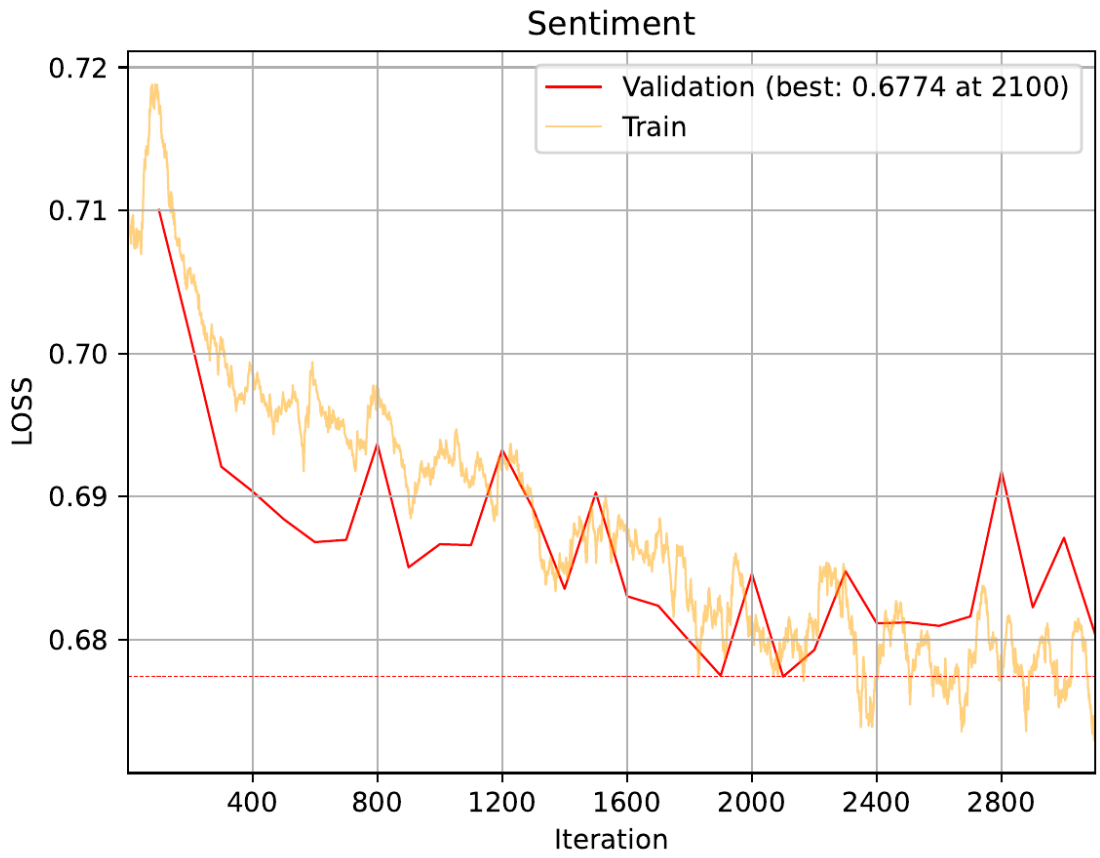
D - Using the Mini Foundation Model for IMDB

In this section, we'll use the pre-trained mini foundation model as a starting point for training our IMDB sentiment analysis model. Specifically, we will only load the text encoder part of the mini foundation model while other parts of the IMDB model will be trained from scratch.

While the configuration files remain the same, there is a slight change in the training command:

```
eirtrain \
--global_configs eir_tutorials/e_pretraining/02_mini_foundation/conf/imdb/imdb_globals.
↪yaml \
--input_configs eir_tutorials/e_pretraining/02_mini_foundation/conf/imdb/imdb_input.yaml␣
↪\
--fusion_configs eir_tutorials/e_pretraining/02_mini_foundation/conf/imdb/imdb_fusion.
↪yaml \
--output_configs eir_tutorials/e_pretraining/02_mini_foundation/conf/imdb/imdb_output.
↪yaml \
--imdb_globals.output_folder=eir_tutorials/tutorial_runs/e_pretraining/02_mini_
↪foundation_imdb_from_pretrained \
--imdb_input.pretrained_config.model_path=eir_tutorials/e_pretraining/02_mini_foundation/
↪data/02_mini_foundation/saved_models/02_mini_foundation_model_18000_perf-average=0.
↪0809.pt \
--imdb_input.pretrained_config.load_module_name=text
```

Let's examine the performance improvements, if any:



In this specific case, the training and validation losses are very marginally lower compared to the baseline. This indicates that the mini foundation model didn't contribute significantly to enhancing the model's performance for IMDB sentiment analysis. One reason could be that the text data each model is trained on is very different, with the mini foundation model being trained on somewhat robotic image captions, while the IMDB model is trained on various movie reviews.

Note: You might notice that the the pre-trained model was trained for more iterations, this was due to early stopping being activated earlier in the model trained from scratch, which might simply be due to randomness. Hence, the fact that the pre-trained model performs slightly better might be due to the fact that it was trained for more iterations, not necessarily because of the pre-training.

While the performance improvements are not significant in the text case, we will not give up on our mini foundation model just yet. Let's see how well the image encoder part of the mini foundation model performs when used for image classification.

E - Establishing a CIFAR10 Baseline

Just like for the IMDB case, we will first establish a baseline.

Here are the configurations for the CIFAR10 baseline:

Listing 118: cifar_globals.yaml

```yaml
output_folder: eir_tutorials/tutorial_runs/e_pretraining/02_mini_foundation
valid_size: 1024
n_saved_models: 1
checkpoint_interval: 100
plot_skip_steps: 0
sample_interval: 100
memory_dataset: true
dataloader_workers: 0
n_epochs: 20
batch_size: 64
lr: 0.0005
optimizer: "adabelief"
device: "mps"
```

Listing 119: cifar_input.yaml

```yaml
input_info:
  input_source: eir_tutorials/e_pretraining/02_mini_foundation/data/CIFAR10/images
  input_name: image_input
  input_type: image

model_config:
  model_type: cnn
  model_init_config:
    channel_exp_base: 5
    kernel_width: 2
    down_stride_width: 2
    kernel_height: 2
    down_stride_height: 2
```

Listing 120: cifar_output.yaml

```yaml
output_info:
  output_source: eir_tutorials/e_pretraining/02_mini_foundation/data/CIFAR10/images_
↪classes.csv
  output_name: cifar_output
  output_type: tabular

output_type_info:
  target_cat_columns:
    - Class
```

To initiate the training for CIFAR10 from scratch, execute the following command:

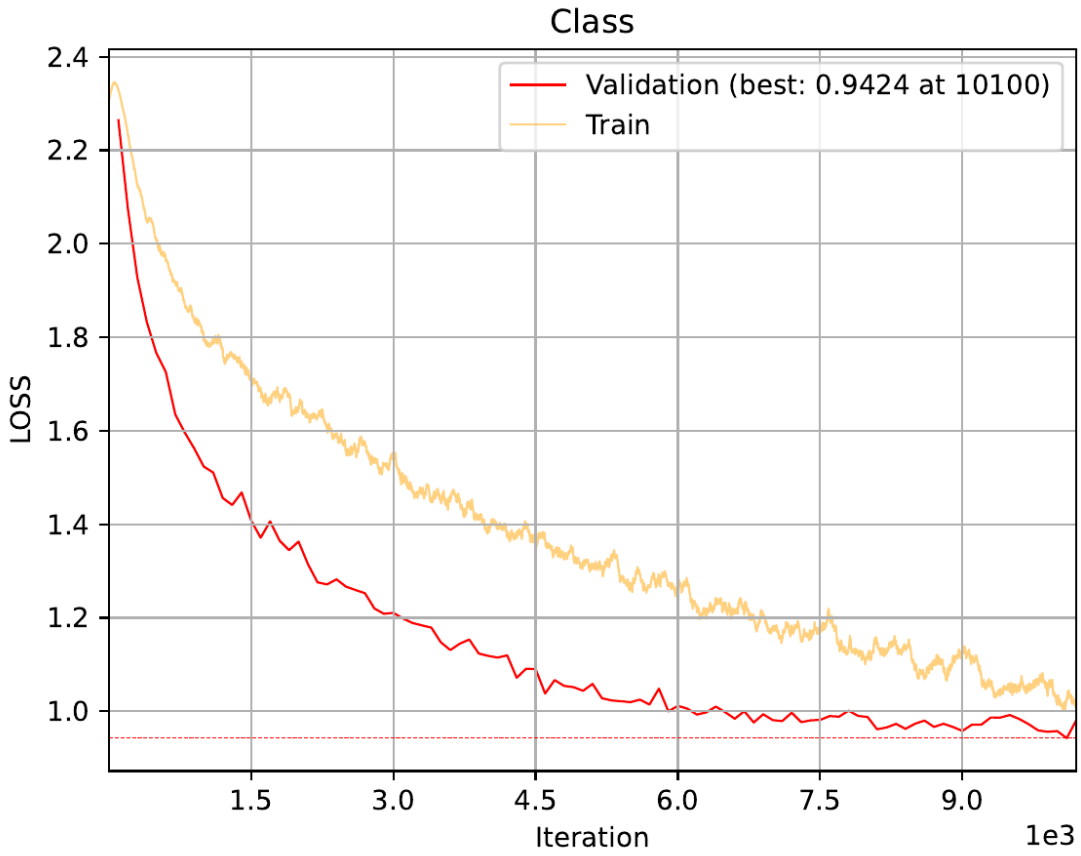
```
eirtrain \
--global_configs eir_tutorials/e_pretraining/02_mini_foundation/conf/cifar/cifar_globals.
```

(continues on next page)

(continued from previous page)

```
↪yaml \
--input_configs eir_tutorials/e_pretraining/02_mini_foundation/conf/cifar/cifar_input.
↪yaml \
--fusion_configs eir_tutorials/e_pretraining/02_mini_foundation/conf/cifar/cifar_fusion.
↪yaml \
--output_configs eir_tutorials/e_pretraining/02_mini_foundation/conf/cifar/cifar_output.
↪yaml \
--cifar_globals.output_folder=eir_tutorials/tutorial_runs/e_pretraining/02_mini_
↪foundation_cifar_from_scratch
```

Training curve:



This will serve as our baseline for CIFAR10, which we will compare against the model that uses the image encoder from the mini foundation model in the next section.

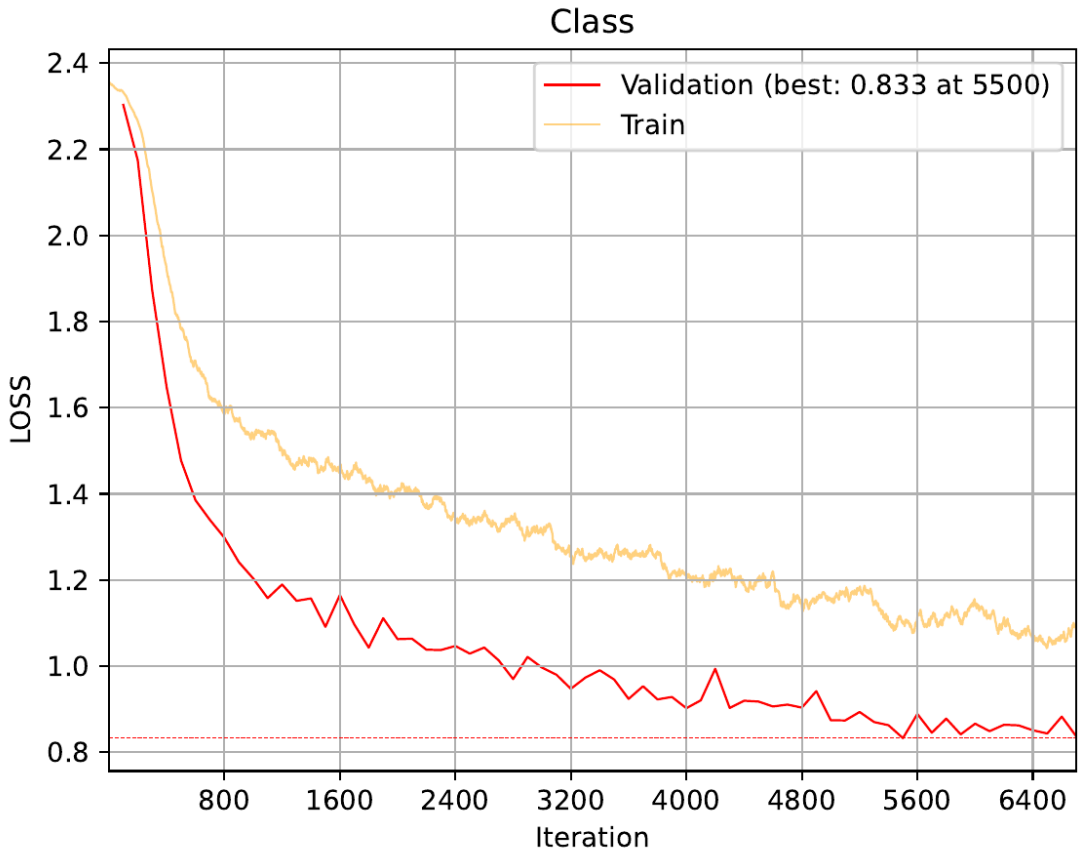
F - Using the Mini Foundation Model for CIFAR10

In this section, we'll use the pre-trained mini foundation model for CIFAR10 image classification. Specifically, we'll load only the image encoder from the mini foundation model, while the rest of the CIFAR10 model will be trained from scratch.

Again, the configuration files for this step are the same as in the baseline, with one change in the training command:

```
eirtrain \
--global_configs eir_tutorials/e_pretraining/02_mini_foundation/conf/cifar/cifar_globals.
↪yaml \
--input_configs eir_tutorials/e_pretraining/02_mini_foundation/conf/cifar/cifar_input.
↪yaml \
--fusion_configs eir_tutorials/e_pretraining/02_mini_foundation/conf/cifar/cifar_fusion.
↪yaml \
--output_configs eir_tutorials/e_pretraining/02_mini_foundation/conf/cifar/cifar_output.
↪yaml \
--cifar_globals.output_folder=eir_tutorials/tutorial_runs/e_pretraining/02_mini_
↪foundation_cifar_from_pretrained \
--cifar_input.pretrained_config.model_path=eir_tutorials/e_pretraining/02_mini_
↪foundation/data/02_mini_foundation/saved_models/02_mini_foundation_model_18000_perf-
↪average=0.0809.pt \
--cifar_input.pretrained_config.load_module_name=image_input
```

Now, let's review the impact on performance:



In contrast to the text-based IMDB model, the CIFAR10 model shows improvements in both the speed of convergence (e.g., the loss at iteration 1500 is lower for the pre-trained model than the model trained from scratch) and the final performance when initialized with the image encoder from the mini foundation model.

These results suggest that the image encoder from the mini foundation model can be transferred to image classification, indicating that one can successfully train and, in a modular fashion, transfer parts of a model to other tasks.

Thank you very much for reading this tutorial!

2.5 Customizing EIR

2.5.1 01 – Customizing EIR: Customized Fusion Tutorial

A - Setup

In this tutorial, we will be looking at how to customize EIR. Specifically, we will be writing our own fusion module through the EIR Python API.

If you want to skip straight to the code, you can find it here: *D - Full Code*.

B - Writing a custom fusion module

Here, we will write a custom fusion module that uses an LSTM to fuse the outputs of the individual feature extractors included in EIR. This is a bit of a contrived example, since, we are only using one input modality, but hopefully it will serve as a good example of how to write a custom fusion module.

First, we define our LSTM fusion module. There are two specific things to note here:

1. We need to define a `num_out_features` attribute / property. This is used to determine the size of the output of the fusion module, which subsequent output modules use.
2. The `forward` method takes a dictionary of inputs, where the keys are the names of the input modalities and the values are the outputs of the corresponding feature extractors. The `forward` method should return a single tensor that is the output of the fusion module.

```
class MyLSTMFusionModule(nn.Module):
    def __init__(self, fusion_in_dim: int, out_dim: int):
        """
        An example of a custom fusion module. Here we use a simple LSTM to
        fuse the inputs, but you could use any PyTorch module here.
        """
        super().__init__()

        self.fusion_in_dim = fusion_in_dim
        self.out_dim = out_dim

        self.fusion = nn.LSTM(
            input_size=fusion_in_dim,
            hidden_size=self.out_dim,
            num_layers=1,
            batch_first=True,
        )

    @property
    def num_out_features(self) -> int:
        return self.out_dim

    def forward(self, inputs: Dict[str, FeatureExtractorProtocol]) -> al_fused_features:
        features = torch.cat(tuple(inputs.values()), dim=1)
        assert features.shape[1] == self.fusion_in_dim

        out, *_ = self.fusion(features)

        return out
```

Having defined our fusion module, we now want to register and run our experiment (which is using our custom fusion module) with EIR. For this demo, we will be use a little function that replaces a couple of attributes in a default experiment, but there are other ways to do this as well. Of note:

1. After defining our fusion module, we also set up the output modules by calling `get_output_modules`. This is necessary because the output modules need to know the size of the output coming from the fusion module.
2. We are using the default `MetaModel` module included in EIR, which is a simple wrapper around the input, fusion and output modules. But you could also use a custom module here.

```
def modify_experiment(experiment: train.Experiment) -> train.Experiment:
    my_experiment_attributes = experiment.__dict__

    input_modules = experiment.model.input_modules
    fusion_in_dim = sum(i.num_out_features for i in input_modules.values())

    my_fusion_module = MyLSTMFusionModule(fusion_in_dim=fusion_in_dim, out_dim=128)
    my_fusion_modules = nn.ModuleDict({"computed": my_fusion_module})

    my_output_modules, _ = get_output_modules(
        outputs_as_dict=experiment.outputs,
        computed_out_dimensions=my_fusion_module.num_out_features,
        device=experiment.configs.global_config.device,
    )

    my_model = MetaModel(
        input_modules=input_modules,
        fusion_modules=my_fusion_modules,
        output_modules=my_output_modules,
        fusion_to_output_mapping={"ancestry_output": "computed"},
    )

    my_optimizer = torch.optim.Adam(
        params=my_model.parameters(),
        lr=1e-4,
    )

    my_experiment_attributes["model"] = my_model
    my_experiment_attributes["optimizer"] = my_optimizer

    my_experiment = train.Experiment(**my_experiment_attributes)

    return my_experiment
```

Finally, we can run our experiment with our custom fusion module. Here we are reusing a couple of functions from `eir.train`.

```
def main():
    configs = get_configs()

    configure_global_eir_logging(output_folder=configs.global_config.output_folder)

    default_hooks = step_logic.get_default_hooks(configs=configs)
    default_experiment = train.get_default_experiment(
        configs=configs,
        hooks=default_hooks,
    )

    my_experiment = modify_experiment(experiment=default_experiment)

    train.run_experiment(experiment=my_experiment)
```

C - Running the custom fusion module

Having defined our custom fusion module and experiment above, we can now run our experiment.

To start, please download processed sample data, The sample data we are using here for predicting ancestry is the public Human Origins dataset, which we have used in previous tutorials (see *01 – Genotype Tutorial: Ancestry Prediction*).

We also have our configuration files:

```
output_folder: eir_tutorials/tutorial_runs/b_customizing_eir/tutorial_01_run
checkpoint_interval: 200
sample_interval: 200
n_epochs: 15
```

```
input_info:
  input_source: eir_tutorials/a_using_eir/01_basic_tutorial/data/processed_sample_data/
↪arrays
  input_name: genotype
  input_type: omics

input_type_info:
  snp_file: eir_tutorials/a_using_eir/01_basic_tutorial/data/processed_sample_data/data_
↪final_gen.bim

model_config:
  model_type: genome-local-net
```

```
output_info:
  output_name: ancestry_output
  output_source: eir_tutorials/a_using_eir/01_basic_tutorial/data/processed_sample_data/
↪human_origins_labels.csv
  output_type: tabular
output_type_info:
  target_cat_columns:
    - Origin
```

Now we can train, using our custom module but taking advantage of the rest of the default EIR functionalities.

```
python \
docs/doc_modules/b_customizing_eir/a_customizing_fusion.py \
--global_configs eir_tutorials/b_customizing_eir/01_customizing_fusion.rst/conf/tutorial_
↪01_globals.yaml \
--input_configs eir_tutorials/b_customizing_eir/01_customizing_fusion.rst/conf/tutorial_
↪01_input.yaml \
--output_configs eir_tutorials/b_customizing_eir/01_customizing_fusion.rst/conf/tutorial_
↪01_outputs.yaml
```

Note: Note that now we are not using the `eirtrain` command, but instead we are using `python` to run our script.

Let's confirm that we used our now model by looking at the `model_info.txt` file:

```
MetaModel(
  (input_modules): ModuleDict(
```

(continues on next page)

(continued from previous page)

```
    (genotype): LCLModel(
      (fc_0): LCL(in_features=4000, num_chunks=500, kernel_size=8, out_feature_sets=4,␣
↪out_features=2000, bias=True)
      (lcl_blocks): Sequential(
        (0): LCLResidualBlock(
          (norm_1): LayerNorm((2000,), eps=1e-05, elementwise_affine=True)
          (fc_1): LCL(in_features=2000, num_chunks=125, kernel_size=16, out_feature_
↪sets=4, out_features=500, bias=True)
          (act_1): Swish(num_parameters=1)
          (do): Dropout(p=0.1, inplace=False)
          (fc_2): LCL(in_features=500, num_chunks=32, kernel_size=16, out_feature_sets=4,
↪ out_features=128, bias=True)
          (downsample_identity): LCL(in_features=2000, num_chunks=128, kernel_size=16,␣
↪out_feature_sets=1, out_features=128, bias=True)
          (stochastic_depth): StochasticDepth(p=0.0, mode=batch)
        )
      )
    )
  )
  (fusion_modules): ModuleDict(
    (computed): MyLSTMFusionModule(
      (fusion): LSTM(128, 128, batch_first=True)
    )
  )
  (output_modules): ModuleDict(
    (ancestry_output): ResidualMLPOutputModule(
      (multi_task_branches): ModuleDict(
        (Origin): Sequential(
          (0): Sequential(
            (0): Sequential(
              (0): MLPResidualBlock(
                (norm_1): LayerNorm((128,), eps=1e-05, elementwise_affine=True)
                (fc_1): Linear(in_features=128, out_features=256, bias=True)
                (act_1): Swish(num_parameters=1)
                (do): Dropout(p=0.1, inplace=False)
                (fc_2): Linear(in_features=256, out_features=256, bias=True)
                (downsample_identity): Linear(in_features=128, out_features=256,␣
↪bias=True)
                (stochastic_depth): StochasticDepth(p=0.1, mode=batch)
              )
            )
            (1): Sequential(
              (0): MLPResidualBlock(
                (norm_1): LayerNorm((256,), eps=1e-05, elementwise_affine=True)
                (fc_1): Linear(in_features=256, out_features=256, bias=True)
                (act_1): Swish(num_parameters=1)
                (do): Dropout(p=0.1, inplace=False)
                (fc_2): Linear(in_features=256, out_features=256, bias=True)
                (stochastic_depth): StochasticDepth(p=0.1, mode=batch)
              )
            )
          )
```

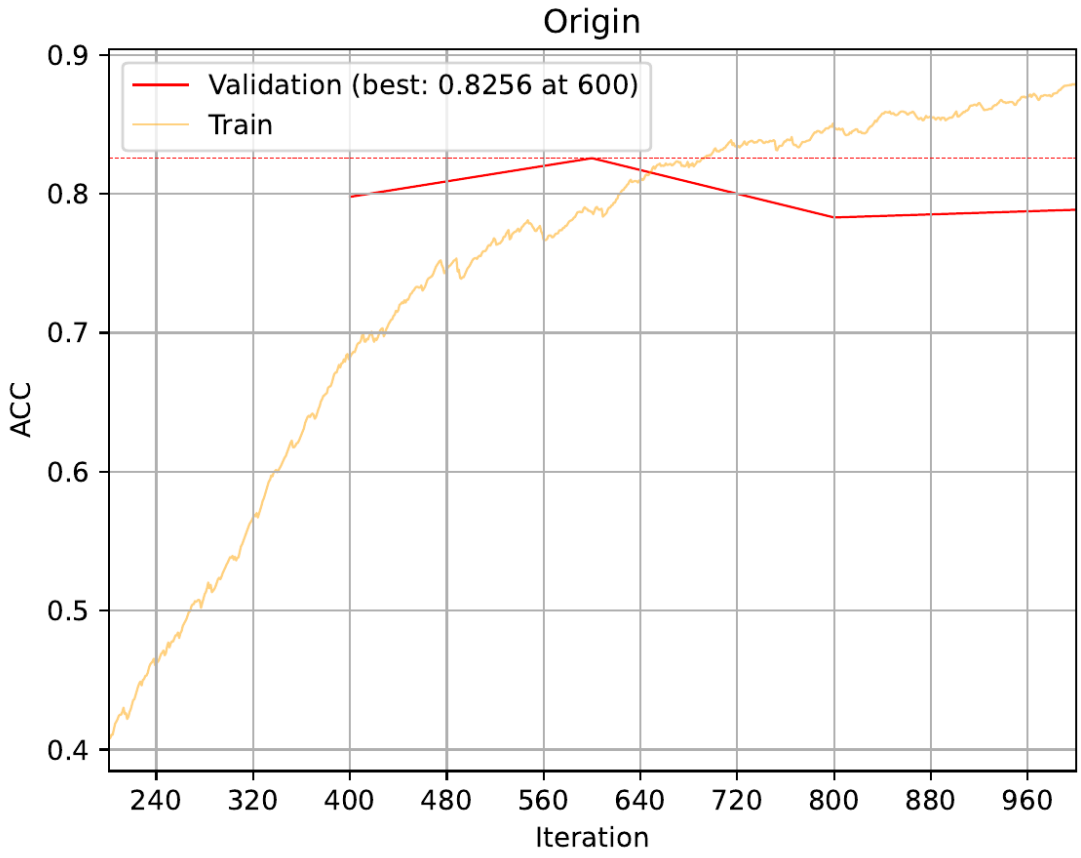
(continues on next page)

(continued from previous page)

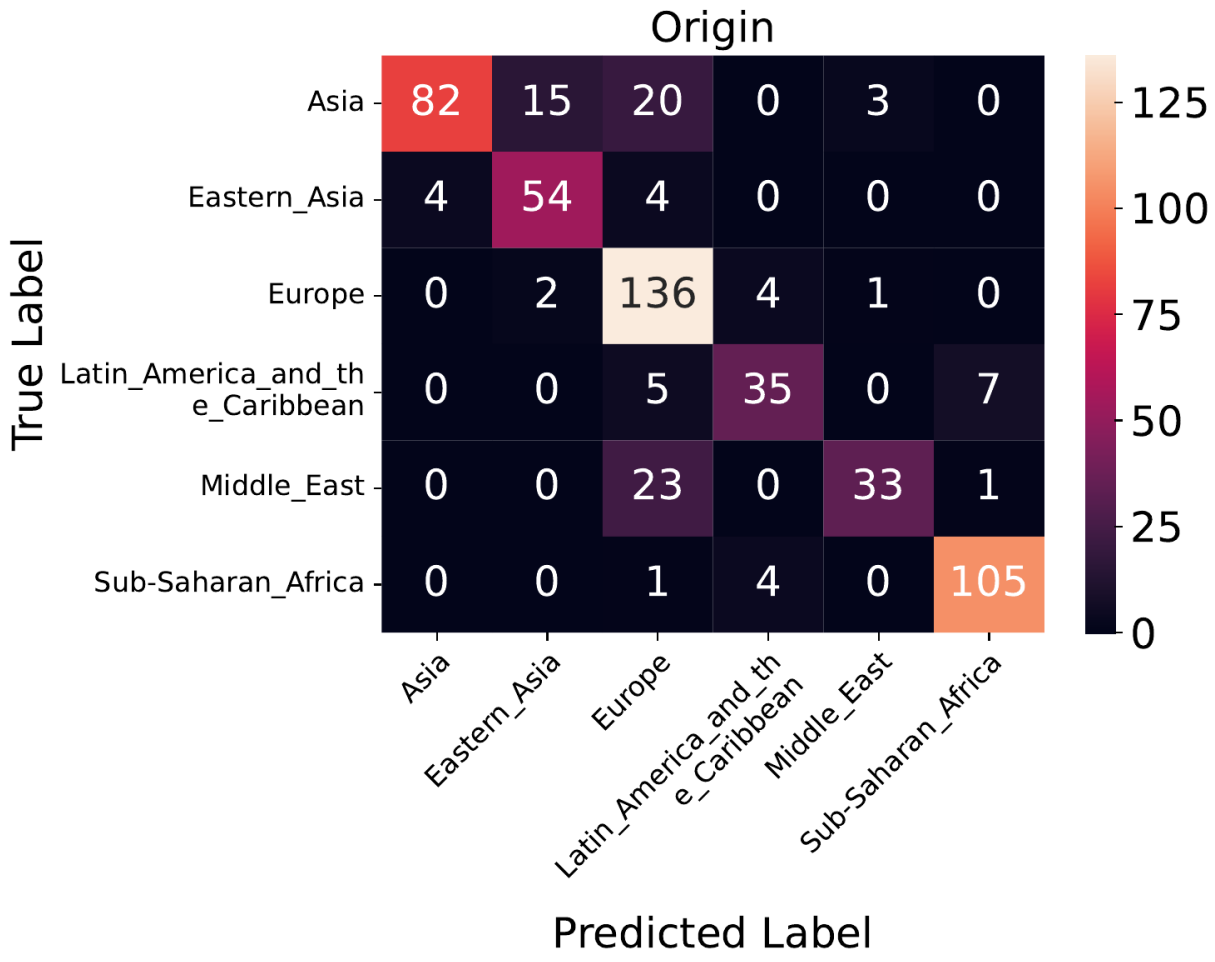
```
          (1): Sequential(
            (norm_final): LayerNorm((256,), eps=1e-05, elementwise_affine=True)
            (act_final): Swish(num_parameters=1)
            (do_final): Dropout(p=0.1, inplace=False)
          )
          (2): Sequential(
            (final): Linear(in_features=256, out_features=6, bias=True)
          )
        )
      )
    )
  )
)
```

So, we can use that our experiment used our custom fusion module, `MyLSTMFusionModule`.

Now let's have a look at how well our model did w.r.t. accuracy:



Not too bad! We can also look at the confusion matrix:



This marks the end of our tutorial on customizing the fusion module in EIR. In the future, there might be more tutorials customizing other aspects of EIR (e.g., the input modules, output modules, etc.), but for now, hopefully this tutorial was helpful.

D - Full Code

```
from typing import Dict

import torch
from torch import nn

from eir import train
from eir.models.meta.meta import FeatureExtractorProtocol, MetaModel, al_fused_features
from eir.models.model_setup_modules.meta_setup import get_output_modules
from eir.setup.config import get_configs
from eir.train_utils import step_logic
from eir.train_utils.utils import configure_global_eir_logging


def main():
```

(continues on next page)

(continued from previous page)

```
    configs = get_configs()

    configure_global_eir_logging(output_folder=configs.global_config.output_folder)

    default_hooks = step_logic.get_default_hooks(configs=configs)
    default_experiment = train.get_default_experiment(
        configs=configs,
        hooks=default_hooks,
    )

    my_experiment = modify_experiment(experiment=default_experiment)

    train.run_experiment(experiment=my_experiment)


class MyLSTMFusionModule(nn.Module):
    def __init__(self, fusion_in_dim: int, out_dim: int):
        """
        An example of a custom fusion module. Here we use a simple LSTM to
        fuse the inputs, but you could use any PyTorch module here.
        """
        super().__init__()

        self.fusion_in_dim = fusion_in_dim
        self.out_dim = out_dim

        self.fusion = nn.LSTM(
            input_size=fusion_in_dim,
            hidden_size=self.out_dim,
            num_layers=1,
            batch_first=True,
        )

    @property
    def num_out_features(self) -> int:
        return self.out_dim

    def forward(self, inputs: Dict[str, FeatureExtractorProtocol]) -> al_fused_features:
        features = torch.cat(tuple(inputs.values()), dim=1)
        assert features.shape[1] == self.fusion_in_dim

        out, *_ = self.fusion(features)

        return out


def modify_experiment(experiment: train.Experiment) -> train.Experiment:
    my_experiment_attributes = experiment.__dict__

    input_modules = experiment.model.input_modules
    fusion_in_dim = sum(i.num_out_features for i in input_modules.values())
```

(continues on next page)

(continued from previous page)

```
    my_fusion_module = MyLSTMFusionModule(fusion_in_dim=fusion_in_dim, out_dim=128)
    my_fusion_modules = nn.ModuleDict({"computed": my_fusion_module})

    my_output_modules, _ = get_output_modules(
        outputs_as_dict=experiment.outputs,
        computed_out_dimensions=my_fusion_module.num_out_features,
        device=experiment.configs.global_config.device,
    )

    my_model = MetaModel(
        input_modules=input_modules,
        fusion_modules=my_fusion_modules,
        output_modules=my_output_modules,
        fusion_to_output_mapping={"ancestry_output": "computed"},
    )

    my_optimizer = torch.optim.Adam(
        params=my_model.parameters(),
        lr=1e-4,
    )

    my_experiment_attributes["model"] = my_model
    my_experiment_attributes["optimizer"] = my_optimizer

    my_experiment = train.Experiment(**my_experiment_attributes)

    return my_experiment


if __name__ == "__main__":
    main()
```

2.6 API

2.6.1 Configuration API

Global Configurations

class eir.setup.schemas.**GlobalConfig**(*output_folder: str*, *n_epochs: int = 10*, *batch_size: int = 64*, *valid_size: float | int = 0.1*, *manual_valid_ids_file: str | None = None*, *dataloader_workers: int = 0*, *device: str = 'cpu'*, *n_iter_before_swa: None | int = None*, *amp: bool = False*, *compile_model: bool = False*, *weighted_sampling_columns: None | Sequence[str] = None*, *lr: float = 0.001*, *lr_lb: float = 0.0*, *find_lr: bool = False*, *lr_schedule: Literal['cycle', 'plateau', 'same', 'cosine'] = 'plateau'*, *lr_plateau_patience: int = 10*, *lr_plateau_factor: float = 0.2*, *gradient_clipping: float = 1.0*, *gradient_accumulation_steps: None | int = None*, *gradient_noise: float = 0.0*, *cat_averaging_metrics: al_cat_averaging_metric_choices | None = None*, *con_averaging_metrics: al_con_averaging_metric_choices | None = None*, *early_stopping_patience: int = 10*, *early_stopping_buffer: None | int = None*, *warmup_steps: Literal['auto'] | int = 'auto'*, *optimizer: Literal['accsgd'], Literal['adabelief'], Literal['adabeliefw'], Literal['adabound'], Literal['adahessian'], Literal['adam'], Literal['adamod'], Literal['adamp'], Literal['adamw'], Literal['aggmo'], Literal['diffgrad'], Literal['lamb'], Literal['lars'], Literal['lookahead'], Literal['madgrad'], Literal['novograd'], Literal['pid'], Literal['qhadam'], Literal['qhm'], Literal['radam'], Literal['ranger'], Literal['rangerqh'], Literal['rangerva'], Literal['sgdm'], Literal['sgdp'], Literal['sgdw'], Literal['shampoo'], Literal['swats'], Literal['yogi'] = 'adam'*, *b1: float = 0.9*, *b2: float = 0.999*, *wd: float = 0.0001*, *memory_dataset: bool = False*, *sample_interval: int = 200*, *save_evaluation_sample_results: bool = True*, *checkpoint_interval: None | int = None*, *n_saved_models: int = 1*, *compute_attributions: bool = False*, *max_attributions_per_class: None | int = None*, *attributions_every_sample_factor: int = 1*, *attribution_background_samples: int = 256*, *plot_lr_schedule: bool = False*, *no_pbar: bool = False*, *log_level: Literal['debug', 'info', 'warning', 'error', 'critical'] = 'info'*, *mixing_alpha: float = 0.0*, *plot_skip_steps: int = 200*, *pretrained_checkpoint: None | str = None*, *strict_pretrained_loading: bool = True*, *latent_sampling: LatentSamplingConfig | None = None*)

Global configurations that are common / relevant for the whole experiment to run.

Parameters

- **output_folder** – What to name the experiment and output folder where results are saved.
- **n_epochs** – Number of epochs for training.
- **batch_size** – Size of batches during training.
- **valid_size** – Size if the validation set, if float then uses a percentage. If int, then raw counts.
- **manual_valid_ids_file** – File with IDs of those samples to manually use as the validation set. Should be one ID per line in the file.
- **dataloader_workers** – Number of workers for multiprocess training and validation data loading.
- **device** – Device to run the training on (e.g. 'cuda:0' / 'cpu' / 'mps'). 'mps' is currently experimental, and might not work for all models.

- **n_iter_before_swa** – Number of iterations to run before activating Stochastic Weight Averaging (SWA).
- **amp** – Whether to use Automatic Mixed Precision. Currently only supported when training on GPUs.
- **compile_model** – Whether to compile the model before training. This can be useful to speed up training, but may not work for all models.
- **weighted_sampling_columns** – Target column to apply weighted sampling on. Only applies to categorical columns. Passing in 'all' here will use an average of all the target columns.
- **lr** – Base learning rate for optimizer.
- **lr_lb** – Lower bound for learning rate when using LR scheduling
- **find_lr** – Whether to perform a range test of different learning rates, with the lower limit being what is passed in for the –lr flag. Produces a plot and exits with status 0 before training if this flag is active.
- **lr_schedule** – Whether to use cyclical, cosine or reduce on plateau learning rate schedule. Otherwise keeps same learning rate
- **lr_plateau_patience** – Number of validation performance steps without improvement over best performance before reducing LR (only relevant when –lr_schedule is 'plateau'.
- **lr_plateau_factor** – Factor to reduce LR when running with plateau schedule.
- **gradient_clipping** – Max norm used for gradient clipping, with p=2.
- **gradient_accumulation_steps** – Number of steps to use for gradient accumulation.
- **gradient_noise** – Gradient noise to inject during training.
- **cat_averaging_metrics** – Which metrics to use for averaging categorical targets. If not set, will use the default metrics for the task type.
- **con_averaging_metrics** – Which metrics to use for averaging continuous targets. If not set, will use the default metrics for the task type.
- **early_stopping_patience** – Number of validation performance steps without improvement over best performance before terminating run.
- **early_stopping_buffer** – Number of iterations to run before activating early stopping checks, useful if networks take a while to 'kick into gear'.
- **warmup_steps** – How many steps to use in warmup. If not set, will automatically compute the number of steps if using an adaptive optimizer, otherwise use 2000.
- **optimizer** – What optimizer to use.
- **b1** – Decay of first order momentum of gradient for relevant optimizers.
- **b2** – Decay of second order momentum of gradient for relevant optimizers.
- **wd** – Weight decay.
- **memory_dataset** – Whether to load all sample into memory during training.
- **sample_interval** – Iteration interval to perform validation and possibly attribution analysis if set.
- **save_evaluation_sample_results** – Whether to save evaluation results (e.g. confusion matrix for classification tasks, regression plot and predictions for regression tasks). Setting to False can be useful to save space during large scale experiments.

- **checkpoint_interval** – Iteration interval to checkpoint (i.e. save) model.
- **n_saved_models** – Number of top N models to saved during training.
- **compute_attributions** – Whether to compute attributions / feature importance scores (using integrated gradients) assigned by the model with respect to the input features.
- **max_attributions_per_class** – Maximum number of samples per class to gather for attribution / feature importance analysis. Good to use when modelling on imbalanced data.
- **attributions_every_sample_factor** – Controls whether the attributions / feature importance values are computed at every sample interval (=1), every other sample interval (=2), etc. Useful when computing the attributions takes a long time and we don't want to do it every time we evaluate.
- **attribution_background_samples** – Number of samples to use for the background in attribution / feature importance computations.
- **plot_lr_schedule** – Whether to run LR search, plot the results and exit with status 0.
- **no_pbar** – Whether to not use progress bars. Useful when stdout/stderr is written to files.
- **log_level** – Logging level to use. Can be one of 'debug', 'info', 'warning', 'error', 'critical'.
- **mixing_alpha** – Alpha parameter used for mixing (higher means more mixing).
- **plot_skip_steps** – How many iterations to skip in plots.
- **pretrained_checkpoint** – Path to a pretrained checkpoint model file (under saved_models/ in the experiment output folder) to load and use as a starting point for training.
- **strict_pretrained_loading** – Whether to enforce that the loaded pretrained model exactly the same architecture as the current model. If False, will only load the layers that match between the two models.
- **latent_sampling** – Configuration to use for latent sampling.

Input Configurations

class eir.setup.schemas.**InputConfig**(*input_info:* InputDataConfig, *input_type_info:* OmicsInputDataConfig | TabularInputDataConfig | SequenceInputDataConfig | ByteInputDataConfig | ImageInputDataConfig | ArrayInputDataConfig, *model_config:* OmicsModelConfig | TabularModelConfig | ImageModelConfig | SequenceModelConfig | ArrayModelConfig, *pretrained_config: None | BasicPretrainedConfig = None*, *interpretation_config: None* | BasicInterpretationConfig = *None*)

Parameters

- **input_info** – Information about the input source, name and type.
- **input_type_info** – Information specific to the input type, e.g. some augmentations are only relevant for omics input. Another example is the type of model to apply to the input.
- **model_config** – Configuration for the chosen model (i.e. feature extractor) for this input.
- **pretrained_config** – Configuration for using leveraging pretraining from a previous experiment.
- **interpretation_config** – Configuration for interpretation analysis when applicable.

Input Data Configuration

class eir.setup.schemas.**InputDataConfig**(*input_source: str*, *input_name: str*, *input_type: Literal['omics', 'tabular', 'sequence', 'image', 'bytes', 'array']*, *input_inner_key: None | str = None*)

Parameters

- **input_source** – Where on the filesystem to locate the input.
- **input_name** – Name to identify the input.
- **input_type** – Type of the input.
- **input_inner_key** – Inner key to use for the input. Only used when input_source is a deeplake dataset.

Input Type Configurations

class eir.setup.schemas.**OmicsInputDataConfig**(*snp_file: str | None = None*, *subset_snps_file: str | None = None*, *na_augment_alpha: float = 1.0*, *na_augment_beta: float = 5.0*, *shuffle_augment_alpha: float = 0.0*, *shuffle_augment_beta: float = 0.0*, *omics_format: Literal['one-hot'] = 'one-hot'*, *mixing_subtype: Literal['mixup', 'cutmix-block', 'cutmix-uniform'] = 'mixup'*, *modality_dropout_rate: float = 0.0*)

Parameters

- **snp_file** – Path to the relevant `.bim` file, used for attribution analysis.
- **subset_snps_file** – Path to a file with corresponding SNP IDs to subset from the main arrays for the modelling. Requires the `snp_file` parameter to be passed in.
- **na_augment_alpha** – Used to control the extent of missing data augmentation in the omics data. A value is sampled from a beta distribution, and the sampled value is used to set a percentage of the SNPs to be 'missing'.

  The alpha () parameter of the beta distribution, influencing the shape of the distribution towards 1. Higher values of alpha (compared to beta) bias the distribution to sample larger percentages of SNPs to be set as 'missing', leading to a higher likelihood of missingness. Conversely, lower values of alpha (compared to beta) result in sampling lower percentages, thus reducing the probability and extent of missingness. For example, setting alpha to 1.0 and beta to 5.0 will skew the distribution towards lower percentages of missingness, since beta is significantly larger. Setting alpha to 5.0 and beta to 1.0 will skew the distribution towards higher percentages of missingness, since alpha is significantly larger. Examples: - alpha = 1.0, beta = 9.0: =E(X)=0.05, =SD(X)=0.0476 (avg 5% missing) - alpha = 1.0, beta = 4.0: =E(X)=0.2, =SD(X)=0.1633 (avg 20% missing)
- **na_augment_beta** – Used to control the extent of missing data augmentation in the omics data. A value is sampled from a beta distribution, and the sampled value is used to set a percentage of the SNPs to be 'missing'.

  Beta () parameter of the beta distribution, influencing the shape of the distribution towards 0. Higher values of beta (compared to alpha) bias the distribution to sample smaller percentages of SNPs to be set as 'missing', leading to a lower likelihood and extent of missingness. Conversely, lower values of beta (compared to alpha) result in sampling larger percentages, thus increasing the probability and extent of missingness.

- **shuffle_augment_alpha** – Used to control the extent of shuffling data augmentation in the omics data. A value is sampled from a beta distribution, and the sampled value is used to determine the percentage of the SNPs to be shuffled.

  The alpha () parameter of the beta distribution, influencing the shape of the distribution towards 1. Higher values of alpha (compared to beta) bias the distribution to sample larger percentages of SNPs to be shuffled, leading to a higher likelihood of extensive shuffling. Conversely, lower values of alpha (compared to beta) result in sampling lower percentages, thus reducing the extent of shuffling. Setting alpha to a significantly larger value than beta will skew the distribution towards higher percentages of shuffling. Examples: - alpha = 1.0, beta = 9.0: =E(X)=0.05, =SD(X)=0.0476 (avg 5% shuffled) - alpha = 1.0, beta = 4.0: =E(X)=0.2, =SD(X)=0.1633 (avg 20% shuffled)

- **shuffle_augment_beta** – Used to control the extent of shuffling data augmentation in the omics data. A value is sampled from a beta distribution, and the sampled value is used to determine the percentage of the SNPs to be shuffled.

  Beta () parameter of the beta distribution, influencing the shape of the distribution towards 0. Higher values of beta (compared to alpha) bias the distribution to sample smaller percentages of SNPs to be shuffled, leading to a lower likelihood and extent of shuffling. Conversely, lower values of beta (compared to alpha) result in sampling larger percentages, thus increasing the likelihood and extent of shuffling.

- **omics_format** – Currently unsupported (i.e. does nothing), which format the omics data is in.

- **mixing_subtype** – Which type of mixing to use on the omics data given that `mixing_alpha` is set >0.0 in the global configuration.

- **modality_dropout_rate** – Dropout rate to apply to the modality, e.g. 0.2 means that 20% of the time, this modality will be dropped out during training.

class eir.setup.schemas.**TabularInputDataConfig**(*input_cat_columns: ~typing.Sequence[str] = <factory>*, *input_con_columns: ~typing.Sequence[str] = <factory>*, *label_parsing_chunk_size: None | int = None*, *mixing_subtype: ~typing.Literal['mixup'] = 'mixup'*, *modality_dropout_rate: float = 0.0*)

Parameters

- **input_cat_columns** – Which columns to use as a categorical inputs from the `input_source` specified in the `input_info` field of the relevant `.yaml`.

- **input_con_columns** – Which columns to use as a continuous inputs from the `input_source` specified in the `input_info` field of the relevant `.yaml`.

- **label_parsing_chunk_size** – Number of rows to process at time when loading in the `input_source`. Useful when RAM is limited.

- **mixing_subtype** – Which type of mixing to use on the tabular data given that `mixing_alpha` is set >0.0 in the global configuration.

- **modality_dropout_rate** – Dropout rate to apply to the modality, e.g. 0.2 means that 20% of the time, this modality will be dropped out during training.

class eir.setup.schemas.**SequenceInputDataConfig**(*vocab_file: None | str = None*, *max_length: int | Literal['max', 'average'] = 'average'*, *sampling_strategy_if_longer: Literal['from_start', 'uniform'] = 'uniform'*, *min_freq: int = 10*, *split_on: str | None = ' '*, *tokenizer: Union[Literal['basic_english'], Literal['spacy'], Literal['moses'], Literal['toktok'], Literal['revtok'], Literal['subword'], Literal['bpe'], NoneType] = None*, *tokenizer_language: str | None = None*, *adaptive_tokenizer_max_vocab_size: int | None = None*, *mixing_subtype: Literal['mixup'] = 'mixup'*, *modality_dropout_rate: float = 0.0*)

Parameters

- **vocab_file** – An optional text file containing pre-defined vocabulary to use for the training. If this is not passed in, the framework will automatically build the vocabulary from the training data. Passing in a vocabulary file is therefore useful if (a) you want to manually specify / limit the vocabulary used and/or (b) you want to save time by pre-computing the vocabulary.
- **max_length** – Maximum length to truncate/pad sequences to. This can be an integer or the values 'max' or 'average'. The 'max' keyword will use the maximum sequence length found in the training data, while the 'average' will use the average length across all training samples.
- **sampling_strategy_if_longer** – Controls how sequences are truncated if they are longer than the specified `max_length` parameter. Using 'from_start' will always truncate from the beginning of the sequence, ensuring the the samples will always be the same during training. Setting this parameter to `uniform` will uniformly sample a slice of a given sample sequence during training. Note that for consistency, the validation/test set samples always use the `from_start` setting when truncating.
- **min_freq** – Minimum number of times a token must appear in the total training data to be included in the vocabulary. Note that this setting will not do anything if passing in `vocab_file`.
- **split_on** – Which token to split the sequence on to generate separate tokens for the vocabulary.
- **tokenizer** – Which tokenizer to use. Relevant if modelling on language, but not as much when doing it on other arbitrary sequences.
- **tokenizer_language** – Which language rules the tokenizer should apply when tokenizing the raw data.
- **adaptive_tokenizer_max_vocab_size** – If using an adaptive tokenizer ("bpe"), this parameter controls the maximum size of the vocabulary.
- **mixing_subtype** – Which type of mixing to use on the sequence data given that `mixing_alpha` is set >0.0 in the global configuration.
- **modality_dropout_rate** – Dropout rate to apply to the modality, e.g. 0.2 means that 20% of the time, this modality will be dropped out during training.

class eir.setup.schemas.**ByteInputDataConfig**(*max_length: int = 256*, *byte_encoding: Literal['uint8'] = 'uint8'*, *sampling_strategy_if_longer: Literal['from_start', 'uniform'] = 'uniform'*, *mixing_subtype: Literal['mixup'] = 'mixup'*, *modality_dropout_rate: float = 0.0*)

Parameters

- **byte_encoding** – Which byte encoding to use when reading the binary data, currently only support uint8.
- **max_length** – Maximum length to truncate/pad sequences to. While in sequence models this generally refers to words, here we are referring to number of bytes.
- **sampling_strategy_if_longer** – Controls how sequences are truncated if they are longer than the specified `max_length` parameter. Using 'from_start' will always truncate from the beginning of the byte sequence, ensuring the the samples will always be the same during training. Setting this parameter to `uniform` will uniformly sample a slice of a given sample sequence during training. Note that for consistency, the validation/test set samples always use the `from_start` setting when truncating.
- **mixing_subtype** – Which type of mixing to use on the bytes data given that `mixing_alpha` is set >0.0 in the global configuration.
- **modality_dropout_rate** – Dropout rate to apply to the modality, e.g. 0.2 means that 20% of the time, this modality will be dropped out during training.

class eir.setup.schemas.**ImageInputDataConfig**(*auto_augment: bool = True*, *size: Sequence[int] = (64,)*, *resize_approach: Literal['resize', 'randomcrop', 'centercrop'] = 'resize'*, *mean_normalization_values: None | Sequence[float] = None*, *stds_normalization_values: None | Sequence[float] = None*, *num_channels: int | None = None*, *mixing_subtype: Literal['mixup'] | Literal['cutmix'] = 'mixup'*, *modality_dropout_rate: float = 0.0*)

Parameters

- **auto_augment** – Setting this to True will use TrivialAugment Wide augmentation.
- **size** – Target size of the images for training. If size is a sequence like (h, w), output size will be matched to this. If size is an int, the image will be resized to (size, size).
- **resize_approach** – The method used for resizing the images. Options are: - "resize": Directly resize the image to the target size. - "randomcrop": Resize the image to a larger size than the target and then apply a random crop to the target size. - "centercrop": Resize the image to a larger size than the target and then apply a center crop to the target size.
- **mean_normalization_values** – Average channel values to normalize images with. This can be a sequence matching the number of channels, or None. If None and using a pretrained model, the values used for the model pretraining will be used. If None and training from scratch, will iterate over training data and compute the running average per channel.
- **stds_normalization_values** – Standard deviation channel values to normalize images with. This can be a sequence mathing the number of channels, or None. If None and using a pretrained model, the values used for the model pretraining will be used. If None and training from scratch, will iterate over training data and compute the running average per channel.
- **num_channels** – Number of channels in the images. If None, will try to infer the number of channels from a random image in the training data.
- **mixing_subtype** – Which type of mixing to use on the image data given that `mixing_alpha` is set >0.0 in the global configuration.
- **modality_dropout_rate** – Dropout rate to apply to the modality, e.g. 0.2 means that 20% of the time, this modality will be dropped out during training.

class eir.setup.schemas.**ArrayInputDataConfig**(*mixing_subtype: Literal['mixup'] = 'mixup'*, *modality_dropout_rate: float = 0.0*, *normalization: Literal['element', 'channel'] | None = 'channel'*, *adaptive_normalization_max_samples: int | None = None*)

Parameters

- **mixing_subtype** – Which type of mixing to use on the image data given that `mixing_alpha` is set >0.0 in the global configuration.
- **modality_dropout_rate** – Dropout rate to apply to the modality, e.g. 0.2 means that 20% of the time, this modality will be dropped out during training.
- **normalization** – Which type of normalization to apply to the array data. If `element`, will normalize each element in the array independently. If `channel`, will normalize each channel in the array independently. For 'channel', assumes PyTorch format where the channel dimension is the first dimension.
- **adaptive_normalization_max_samples** – If using adaptive normalization (channel / element), how many samples to use to compute the normalization parameters. If None, will use all samples.

Input Model Configurations

These configurations are used to specify the input feature extractor architecture, as well as paramters that can be common between different feature extractors. For a given feature extractor (specified with the *model_type* field), there are there are various configurations available through the *model_init_config* field. The documentation below contains more details about the different configurations available for each feature extractor.

class eir.models.input.omics.omics_models.**OmicsModelConfig**(*model_type: Literal['cnn', 'linear', 'lcl-simple', 'genome-local-net']*, *model_init_config:* CNNModelConfig | LinearModelConfig | SimpleLCLModelConfig | LCLModelConfig | IdentityModelConfig)

Parameters

- **model_type** – Which type of image model to use.
- **model_init_config** – Configuration used to initialise model.

class eir.models.input.tabular.tabular.**TabularModelConfig**(*model_init_config:* SimpleTabularModelConfig, *model_type: Literal['tabular'] = 'tabular'*)

Parameters

- **model_type** – Which type of image model to use.
- **model_init_config** – Configuration / arguments used to initialise model.

class eir.models.input.sequence.transformer_models.**SequenceModelConfig**(*model_init_config: BasicTransformerFeatureExtractorModelConfig | Dict*, *model_type: Literal['sequence-default'] | str = 'sequence-default'*, *embedding_dim: int = 64*, *position: Literal['encode', 'embed'] = 'encode'*, *position_dropout: float = 0.1*, *window_size: int = 0*, *pool: Literal['avg'] | Literal['max'] | None = None*, *pretrained_model: bool = False*, *freeze_pretrained_model: bool = False*)

Parameters

- **model_init_config** – Configuration / arguments used to initialise model.
- **model_type** – Which type of image model to use.
- **embedding_dim** – Which dimension to use for the embeddings. If `None`, will automatically set this value based on the number of tokens and attention heads.
- **position** – Whether to encode the token position or use learnable position embeddings.
- **position_dropout** – Dropout for the positional encoding / embedding.
- **window_size** – If set to more than 0, will apply a sliding window of feature extraction over the input, meaning the model (e.g. transformer) will only see a part of the input at a time. Can be Useful to avoid the $O(n^2)$ complexity of transformers, as it becomes O(window_size^2 * n_windows) instead.
- **pool** – Whether and how to pool (max / avg) the final feature maps before being passed to the final fusion module / predictor. Meaning we pool over the sequence (i.e. time) dimension, so the resulting dimensions is embedding_dim instead of sequence_length * embedding_dim. If using windowed / conv transformers, this becomes embedding_dim * number_of_chunks.
- **pretrained_model** – Specify whether the model type is assumed to be pretrained and from the Pytorch Image Models repository.
- **freeze_pretrained_model** – Whether to freeze the pretrained model weights.

See *Sequence Models* for more details about available external sequence models.

class eir.models.input.image.image_models.**ImageModelConfig**(*model_type: Literal['cnn'] | str*, *model_init_config: CNNModelConfig | Dict[str, Any]*, *num_output_features: int = 256*, *pretrained_model: bool = False*, *freeze_pretrained_model: bool = False*)

Parameters

- **model_type** – Which type of image model to use.
- **model_init_config** – Configuration / arguments used to initialise model.
- **num_output_features** – Number of output final output features from image feature extractor, which get passed to fusion module.
- **pretrained_model** – Specify whether the model type is assumed to be pretrained and from the Pytorch Image Models repository.
- **freeze_pretrained_model** – Whether to freeze the pretrained model weights.

See *Image Models* for more details about available external image models.

class eir.models.input.array.array_models.**ArrayModelConfig**(*model_type: Literal['cnn', 'lcl']*, *model_init_config:* CNNModelConfig | LCLModelConfig | *ArrayTransformerConfig*, *pre_normalization: Literal['instancenorm', 'layernorm'] | None = None*)

Parameters

- **model_type** – Which type of image model to use.
- **model_init_config** – Configuration used to initialise model.

Interpretation Configurations

Parameters to have basic control over how interpretation is done. Currently only supported for sequence and image data.

class eir.setup.schemas.**BasicInterpretationConfig**(*interpretation_sampling_strategy: Literal['first_n', 'random_sample'] = 'first_n'*, *num_samples_to_interpret: int = 10*, *manual_samples_to_interpret: Sequence[str] | None = None*)

Parameters

- **interpretation_sampling_strategy** – How to sample sequences for attribution analysis. *first_n* always grabs the same first n values from the beginning of the dataset to interpret, while *random_sample* will sample uniformly from the whole dataset without replacement.
- **num_samples_to_interpret** – How many samples to interpret.
- **manual_samples_to_interpret** – IDs of samples to always interpret, irrespective of *interpretation_sampling_strategy* and *num_samples_to_interpret*. A caveat here is that they must be present in the dataset that is being interpreted (e.g. validation / test dataset), meaning that adding IDs here that happen to be in the training dataset will not work.

Feature Extractor Configurations

The documentation below details what the parameters passed to the respective models (trough the *model_init_config* field in the *--input_configs .yaml* files).

Omics Feature Extractors

class eir.models.input.array.models_cnn.**CNNModelConfig**(*layers: None | List[int] = None*, *num_output_features: int = 256*, *channel_exp_base: int = 2*, *first_channel_expansion: int = 1*, *kernel_width: int = 12*, *first_kernel_expansion_width: int = 1*, *down_stride_width: int = 4*, *first_stride_expansion_width: int = 1*, *dilation_factor_width: int = 1*, *kernel_height: int = 4*, *first_kernel_expansion_height: int = 1*, *down_stride_height: int = 1*, *first_stride_expansion_height: int = 1*, *dilation_factor_height: int = 1*, *cutoff: int = 32*, *rb_do: float = 0.0*, *stochastic_depth_p: float = 0.0*, *attention_inclusion_cutoff: int = 0*, *l1: float = 0.0*)

Parameters

- **layers** – A list that controls the number of layers and channels in the model. Each element in the list represents a layer group with a specified number of layers and channels. Specifically,
  - The first element in the list refers to the number of layers with the number of channels exactly as specified by the `channel_exp_base` parameter.
  - The subsequent elements in the list correspond to an increased number of channels, doubling with each step. For instance, if `channel_exp_base=3` (i.e., `2**3=8` channels), and the `layers` list is `[5, 3, 2]`, the model would be constructed as follows,
    - First case: 5 layers with 8 channels
    - Second case: 3 layers with 16 channels (doubling from the previous case)
    - Third case: 2 layers with 32 channels (doubling from the previous case)
  - The model currently supports a maximum of 4 elements in the list.
  - If set to `None`, the model will automatically set up the number of layer groups until a certain width and height (`stride * 8` for both) are met. In this automatic setup, channels will be increased as the input gets propagated through the network, while the width/height get reduced due to stride.

  Future work includes adding a parameter to control the target width and height.
- **num_output_features** – Output dimension of the last FC layer in the network which accepts the outputs from the convolutional layer.
- **channel_exp_base** – Which power of 2 to use in order to set the number of channels in the network. For example, setting `channel_exp_base=3` means that 2**3=8 channels will be used.

- **first_channel_expansion** – Factor to extend the first layer channels.
- **kernel_width** – Base kernel width of the convolutions.
- **first_kernel_expansion_width** – Factor to extend the first kernel's width.
- **down_stride_width** – Down stride of the convolutional layers along the width.
- **first_stride_expansion_width** – Factor to extend the first layer stride along the width.
- **dilation_factor_width** – Base dilation factor of the convolutions along the width in the network.
- **kernel_height** – Base kernel height of the convolutions.
- **first_kernel_expansion_height** – Factor to extend the first kernel's height.
- **down_stride_height** – Down stride of the convolutional layers along the height.
- **first_stride_expansion_height** – Factor to extend the first layer stride along the height.
- **dilation_factor_height** – Base dilation factor of the convolutions along the height in the network.
- **cutoff** – If the *resulting* dimension of width * height of adding a successive block is less than this value, will stop adding residual blocks to the model in the automated case (i.e., if the layers argument is not specified).
- **rb_do** – Dropout in the convolutional residual blocks.
- **stochastic_depth_p** – Probability of dropping input.
- **attention_inclusion_cutoff** – If the dimension of width * height is less than this value, attention will be included in the model across channels and width * height as embedding dimension after that point (with the channels representing the length of the sequence).
- **l1** – L1 regularization to apply to the first layer.

class eir.models.input.array.models_identity.**IdentityModelConfig**(*flatten: bool = True*, *flatten_shape: Literal['c', 'fortran'] = 'c'*)

Parameters

- **flatten** – Whether to flatten the input.
- **flatten_shape** – What column-row order to flatten the input in.

class eir.models.input.array.models_locally_connected.**SimpleLCLModelConfig**(*fc_repr_dim: int = 12*, *num_lcl_chunks: int = 64*, *l1: float = 0.0*)

Parameters

- **fc_repr_dim** – Controls the number of output sets in the first and only split layer. Analogous to channels in CNNs.
- **num_lcl_chunks** – Controls the number of splits applied to the input. E.g. with a input with of 800, using `num_lcl_chunks=100` will result in a kernel width of 8, meaning 8 elements in the flattened input. If using a SNP inputs with a one-hot encoding of 4 possible values, this will result in 8/2 = 2 SNPs per locally connected area.

- **l1** – L1 regularization applied to the first and only locally connected layer.

class eir.models.input.array.models_locally_connected.**LCLModelConfig**(*patch_size: tuple[int, int, int] | None = None*, *layers: None | List[int] = None*, *kernel_width: int | Literal['patch'] = 16*, *first_kernel_expansion: int = -2*, *channel_exp_base: int = 2*, *first_channel_expansion: int = 1*, *num_lcl_chunks: None | int = None*, *rb_do: float = 0.1*, *stochastic_depth_p: float = 0.0*, *l1: float = 0.0*, *cutoff: int | Literal['auto'] = 1024*, *direction: Literal['down', 'up'] = 'down'*, *attention_inclusion_cutoff: int | None = None*)

Note that when using the automatic network setup, kernel widths will get expanded to ensure that the feature representations become smaller as they are propagated through the network.

Parameters

- **patch_size** – Controls the size of the patches used in the first layer. If set to `None`, the input is flattened according to the torch `flatten` function. Note that when using this parameter, we generally want the kernel width to be set to the multiplication of the patch size. Order follows PyTorch convention, i.e., [channels, height, width].
- **layers** – Controls the number of layers in the model. If set to `None`, the model will automatically set up the number of layers according to the `cutoff` parameter value.
- **kernel_width** – With of the locally connected kernels. Note that in the context of genomic inputs this refers to the flattened input, meaning that if we have a one-hot encoding of 4 values (e.g. SNPs), 12 refers to 12/4 = 3 SNPs per locally connected window. Can be set to `None` if the `num_lcl_chunks` parameter is set, which means that the kernel width will be set automatically according to
- **first_kernel_expansion** – Factor to extend the first kernel. This value can both be positive or negative. For example in the case of `kernel_width=12`, setting `first_kernel_expansion=2` means that the first kernel will have a width of 24, whereas other kernels will have a width of 12. When using a negative value, divides the first kernel by the value instead of multiplying.
- **channel_exp_base** – Which power of 2 to use in order to set the number of channels/weight sets in the network. For example, setting `channel_exp_base=3` means that 2**3=8 weight sets will be used.
- **first_channel_expansion** – Whether to expand / shrink the number of channels in the first layer as compared to other layers in the network. Works analogously to the `first_kernel_expansion` parameter.
- **num_lcl_chunks** – Controls the number of splits applied to the input. E.g. with a input width of 800, using `num_lcl_chunks=100` will result in a kernel width of 8, meaning 8

elements in the flattened input. If using a SNP inputs with a one-hot encoding of 4 possible values, this will result in 8/2 = 2 SNPs per locally connected area.

- **rb_do** – Dropout in the residual blocks.
- **stochastic_depth_p** – Probability of dropping input.
- **l1** – L1 regularization applied to the first layer in the network.
- **cutoff** – Feature dimension cutoff where the automatic network setup stops adding layers. The 'auto' option is only supported when using the model for array *outputs*, and will set the cutoff to roughly the number of output features.
- **direction** – Whether to use a "down" or "up" network. "Down" means that the feature representation will get smaller as it is propagated through the network, whereas "up" means that the feature representation will get larger.
- **attention_inclusion_cutoff** – Cutoff to start including attention blocks in the network. If set to `None`, no attention blocks will be included. The cutoff here refers to the "length" dimension of the input after reshaping according to the output_feature_sets in the preceding layer. For example, if we 1024 output features, and we have 4 output feature sets, the length dimension will be 1024/4 = 256. With an attention cutoff >= 256, the attention block will be included.

class eir.models.input.array.models_linear.**LinearModelConfig**(*fc_repr_dim: int = 32*, *l1: float = 0.0*)

Parameters

- **fc_repr_dim** – Number of output nodes in the first and only hidden layer.
- **l1** – L1 regularisation to apply to the first layer.

Tabular Feature Extractors

class eir.models.input.tabular.tabular.**SimpleTabularModelConfig**(*l1: float = 0.0*, *fc_layer: bool = False*)

Parameters

- **l1** – L1 regularization applied to the embeddings for categorical tabular inputs.
- **fc_layer** – Whether to add a single fully-connected layer to the model, alternative to looking up and passing the inputs through directly.

Sequence and Binary Feature Extractors

Built-in Sequence Feature Extractors

class eir.models.input.sequence.transformer_models.**BasicTransformerFeatureExtractorModelConfig**(*num_heads: int = 8*, *num_layers: int = 2*, *dim_feedforward: int | Literal['auto'] = 'auto'*, *dropout: float = 0.1*)

Parameters

- **num_heads** – The number of heads in the multi-head attention models
- **num_layers** – The number of encoder blocks in the transformer model.
- **dim_feedforward** – The dimension of the feedforward layers in the transformer model.
- **dropout** – Dropout value to use in the encoder layers.

External Sequence Feature Extractors

Please refer to *Sequence Models* for more details about the external image models.

Image Feature Extractors

Built-in Image Feature Extractors

class eir.models.input.array.models_cnn.**CNNModelConfig**(*layers: None | List[int] = None*, *num_output_features: int = 256*, *channel_exp_base: int = 2*, *first_channel_expansion: int = 1*, *kernel_width: int = 12*, *first_kernel_expansion_width: int = 1*, *down_stride_width: int = 4*, *first_stride_expansion_width: int = 1*, *dilation_factor_width: int = 1*, *kernel_height: int = 4*, *first_kernel_expansion_height: int = 1*, *down_stride_height: int = 1*, *first_stride_expansion_height: int = 1*, *dilation_factor_height: int = 1*, *cutoff: int = 32*, *rb_do: float = 0.0*, *stochastic_depth_p: float = 0.0*, *attention_inclusion_cutoff: int = 0*, *l1: float = 0.0*)

Parameters

- **layers** – A list that controls the number of layers and channels in the model. Each element in the list represents a layer group with a specified number of layers and channels. Specifically,
  - The first element in the list refers to the number of layers with the number of channels exactly as specified by the `channel_exp_base` parameter.
  - The subsequent elements in the list correspond to an increased number of channels, doubling with each step. For instance, if `channel_exp_base=3` (i.e., `2**3=8` channels), and the `layers` list is `[5, 3, 2]`, the model would be constructed as follows,
    - First case: 5 layers with 8 channels
    - Second case: 3 layers with 16 channels (doubling from the previous case)
    - Third case: 2 layers with 32 channels (doubling from the previous case)
  - The model currently supports a maximum of 4 elements in the list.
  - If set to `None`, the model will automatically set up the number of layer groups until a certain width and height (`stride * 8` for both) are met. In this automatic setup, channels will be increased as the input gets propagated through the network, while the width/height get reduced due to stride.

  Future work includes adding a parameter to control the target width and height.
- **num_output_features** – Output dimension of the last FC layer in the network which accepts the outputs from the convolutional layer.
- **channel_exp_base** – Which power of 2 to use in order to set the number of channels in the network. For example, setting `channel_exp_base=3` means that 2**3=8 channels will be used.
- **first_channel_expansion** – Factor to extend the first layer channels.
- **kernel_width** – Base kernel width of the convolutions.
- **first_kernel_expansion_width** – Factor to extend the first kernel's width.
- **down_stride_width** – Down stride of the convolutional layers along the width.
- **first_stride_expansion_width** – Factor to extend the first layer stride along the width.
- **dilation_factor_width** – Base dilation factor of the convolutions along the width in the network.
- **kernel_height** – Base kernel height of the convolutions.
- **first_kernel_expansion_height** – Factor to extend the first kernel's height.
- **down_stride_height** – Down stride of the convolutional layers along the height.
- **first_stride_expansion_height** – Factor to extend the first layer stride along the height.
- **dilation_factor_height** – Base dilation factor of the convolutions along the height in the network.
- **cutoff** – If the *resulting* dimension of width * height of adding a successive block is less than this value, will stop adding residual blocks to the model in the automated case (i.e., if the layers argument is not specified).
- **rb_do** – Dropout in the convolutional residual blocks.
- **stochastic_depth_p** – Probability of dropping input.

- **attention_inclusion_cutoff** – If the dimension of width * height is less than this value, attention will be included in the model across channels and width * height as embedding dimension after that point (with the channels representing the length of the sequence).
- **l1** – L1 regularization to apply to the first layer.

External Image Feature Extractors

Please refer to *Image Models* for more details about the external image models.

Array Feature Extractors

class eir.models.input.array.models_cnn.**CNNModelConfig**(*layers: None | List[int] = None*, *num_output_features: int = 256*, *channel_exp_base: int = 2*, *first_channel_expansion: int = 1*, *kernel_width: int = 12*, *first_kernel_expansion_width: int = 1*, *down_stride_width: int = 4*, *first_stride_expansion_width: int = 1*, *dilation_factor_width: int = 1*, *kernel_height: int = 4*, *first_kernel_expansion_height: int = 1*, *down_stride_height: int = 1*, *first_stride_expansion_height: int = 1*, *dilation_factor_height: int = 1*, *cutoff: int = 32*, *rb_do: float = 0.0*, *stochastic_depth_p: float = 0.0*, *attention_inclusion_cutoff: int = 0*, *l1: float = 0.0*)

Parameters

- **layers** – A list that controls the number of layers and channels in the model. Each element in the list represents a layer group with a specified number of layers and channels. Specifically,
  - The first element in the list refers to the number of layers with the number of channels exactly as specified by the `channel_exp_base` parameter.
  - The subsequent elements in the list correspond to an increased number of channels, doubling with each step. For instance, if `channel_exp_base=3` (i.e., `2**3=8` channels), and the `layers` list is `[5, 3, 2]`, the model would be constructed as follows,
    - First case: 5 layers with 8 channels
    - Second case: 3 layers with 16 channels (doubling from the previous case)
    - Third case: 2 layers with 32 channels (doubling from the previous case)
  - The model currently supports a maximum of 4 elements in the list.
  - If set to `None`, the model will automatically set up the number of layer groups until a certain width and height (`stride * 8` for both) are met. In this automatic setup, channels will be increased as the input gets propagated through the network, while the width/height get reduced due to stride.

  Future work includes adding a parameter to control the target width and height.
- **num_output_features** – Output dimension of the last FC layer in the network which accepts the outputs from the convolutional layer.

- **channel_exp_base** – Which power of 2 to use in order to set the number of channels in the network. For example, setting `channel_exp_base=3` means that 2**3=8 channels will be used.
- **first_channel_expansion** – Factor to extend the first layer channels.
- **kernel_width** – Base kernel width of the convolutions.
- **first_kernel_expansion_width** – Factor to extend the first kernel's width.
- **down_stride_width** – Down stride of the convolutional layers along the width.
- **first_stride_expansion_width** – Factor to extend the first layer stride along the width.
- **dilation_factor_width** – Base dilation factor of the convolutions along the width in the network.
- **kernel_height** – Base kernel height of the convolutions.
- **first_kernel_expansion_height** – Factor to extend the first kernel's height.
- **down_stride_height** – Down stride of the convolutional layers along the height.
- **first_stride_expansion_height** – Factor to extend the first layer stride along the height.
- **dilation_factor_height** – Base dilation factor of the convolutions along the height in the network.
- **cutoff** – If the *resulting* dimension of width * height of adding a successive block is less than this value, will stop adding residual blocks to the model in the automated case (i.e., if the layers argument is not specified).
- **rb_do** – Dropout in the convolutional residual blocks.
- **stochastic_depth_p** – Probability of dropping input.
- **attention_inclusion_cutoff** – If the dimension of width * height is less than this value, attention will be included in the model across channels and width * height as embedding dimension after that point (with the channels representing the length of the sequence).
- **l1** – L1 regularization to apply to the first layer.

class eir.models.input.array.models_locally_connected.**LCLModelConfig**(*patch_size: tuple[int, int, int] | None = None*, *layers: None | List[int] = None*, *kernel_width: int | Literal['patch'] = 16*, *first_kernel_expansion: int = -2*, *channel_exp_base: int = 2*, *first_channel_expansion: int = 1*, *num_lcl_chunks: None | int = None*, *rb_do: float = 0.1*, *stochastic_depth_p: float = 0.0*, *l1: float = 0.0*, *cutoff: int | Literal['auto'] = 1024*, *direction: Literal['down', 'up'] = 'down'*, *attention_inclusion_cutoff: int | None = None*)

Note that when using the automatic network setup, kernel widths will get expanded to ensure that the feature representations become smaller as they are propagated through the network.

Parameters

- **patch_size** – Controls the size of the patches used in the first layer. If set to `None`, the input is flattened according to the torch `flatten` function. Note that when using this parameter, we generally want the kernel width to be set to the multiplication of the patch size. Order follows PyTorch convention, i.e., [channels, height, width].
- **layers** – Controls the number of layers in the model. If set to `None`, the model will automatically set up the number of layers according to the `cutoff` parameter value.
- **kernel_width** – With of the locally connected kernels. Note that in the context of genomic inputs this refers to the flattened input, meaning that if we have a one-hot encoding of 4 values (e.g. SNPs), 12 refers to 12/4 = 3 SNPs per locally connected window. Can be set to `None` if the `num_lcl_chunks` parameter is set, which means that the kernel width will be set automatically according to
- **first_kernel_expansion** – Factor to extend the first kernel. This value can both be positive or negative. For example in the case of `kernel_width=12`, setting `first_kernel_expansion=2` means that the first kernel will have a width of 24, whereas other kernels will have a width of 12. When using a negative value, divides the first kernel by the value instead of multiplying.
- **channel_exp_base** – Which power of 2 to use in order to set the number of channels/weight sets in the network. For example, setting `channel_exp_base=3` means that 2**3=8 weight sets will be used.
- **first_channel_expansion** – Whether to expand / shrink the number of channels in the first layer as compared to other layers in the network. Works analogously to the `first_kernel_expansion` parameter.
- **num_lcl_chunks** – Controls the number of splits applied to the input. E.g. with a input width of 800, using `num_lcl_chunks=100` will result in a kernel width of 8, meaning 8 elements in the flattened input. If using a SNP inputs with a one-hot encoding of 4 possible values, this will result in 8/2 = 2 SNPs per locally connected area.

- **rb_do** – Dropout in the residual blocks.
- **stochastic_depth_p** – Probability of dropping input.
- **l1** – L1 regularization applied to the first layer in the network.
- **cutoff** – Feature dimension cutoff where the automatic network setup stops adding layers. The 'auto' option is only supported when using the model for array *outputs*, and will set the cutoff to roughly the number of output features.
- **direction** – Whether to use a "down" or "up" network. "Down" means that the feature representation will get smaller as it is propagated through the network, whereas "up" means that the feature representation will get larger.
- **attention_inclusion_cutoff** – Cutoff to start including attention blocks in the network. If set to `None`, no attention blocks will be included. The cutoff here refers to the "length" dimension of the input after reshaping according to the output_feature_sets in the preceding layer. For example, if we 1024 output features, and we have 4 output feature sets, the length dimension will be 1024/4 = 256. With an attention cutoff >= 256, the attention block will be included.

class eir.models.input.array.models_transformers.**ArrayTransformerConfig**(*patch_size: tuple[int, ...]*, *embedding_dim: int*, *num_heads: int = 8*, *num_layers: int = 2*, *dim_feedforward: int | Literal['auto'] = 'auto'*, *dropout: float = 0.1*, *position: Literal['encode', 'embed'] = 'encode'*, *position_dropout: float = 0.1*)

Parameters

- **patch_size** – Controls the size of the patches used in the first layer. If set to `None`, the input is flattened according to the torch `flatten` function. Note that when using this parameter, we generally want the kernel width to be set to the multiplication of the patch size. Order follows PyTorch convention, i.e., [channels, height, width].
- **embedding_dim** – The embedding dimension each patch is projected to. This is also the dimension of the transformer encoder layers.
- **num_heads** – The number of heads in the multi-head attention layers.
- **num_layers** – The number of transformer encoder layers.
- **dim_feedforward** – The dimension of the feedforward layers in the transformer model.
- **dropout** – The dropout rate to use in the transformer encoder layers.
- **position** – Whether to encode the token position or use learnable position embeddings.
- **position_dropout** – The dropout rate to use in the position encoding/embedding.

Fusion Configurations

class eir.setup.schemas.**FusionConfig**(*model_type: Literal['mlp-residual', 'identity', 'mgmoe', 'pass-through']*, *model_config:* ResidualMLPConfig | IdentityConfig | MGMoEModelConfig)

Parameters

- **model_type** – Which type of fusion model to use.
- **model_config** – Fusion model configuration.

Fusion Module Configuration

class eir.models.fusion.fusion_default.**ResidualMLPConfig**(*layers: ~typing.List[int] = <factory>*, *fc_task_dim: int = 256*, *rb_do: float = 0.1*, *fc_do: float = 0.1*, *stochastic_depth_p: float = 0.1*)

Parameters

- **layers** – Number of residual MLP layers to use in for each output predictor after fusing.
- **fc_task_dim** – Number of hidden nodes in each MLP residual block.
- **rb_do** – Dropout in each MLP residual block.
- **fc_do** – Dropout before final layer.
- **stochastic_depth_p** – Probability of dropping input.

class eir.models.fusion.fusion_mgmoe.**MGMoEModelConfig**(*layers: ~typing.Sequence[int] = <factory>*, *fc_task_dim: int = 64*, *mg_num_experts: int = 8*, *rb_do: float = 0.0*, *fc_do: float = 0.0*, *stochastic_depth_p: float = 0.0*)

Parameters

- **layers** – A sequence of two int values controlling the number of residual MLP blocks in the network. The first item (i.e. `layers[0]`) refers to the number of blocks in the expert branches. The second item (i.e. `layers[1]`) refers to the number of blocks in the predictor branches.
- **fc_task_dim** – Number of hidden nodes in all residual blocks (both expert and predictor) of the network.
- **mg_num_experts** – Number of multi gate experts to use.
- **rb_do** – Dropout in all MLP residual blocks (both expert and predictor).
- **fc_do** – Dropout before the last FC layer.
- **stochastic_depth_p** – Probability of dropping input.

class eir.models.fusion.fusion_identity.**IdentityConfig**

Output Configurations

class eir.setup.schemas.**OutputConfig**(*output_info:* OutputInfoConfig, *output_type_info:* TabularOutputTypeConfig | SequenceOutputTypeConfig | ArrayOutputTypeConfig, *model_config:* TabularOutputModuleConfig | SequenceOutputModuleConfig | ArrayOutputModuleConfig, *sampling_config:* SequenceOutputSamplingConfig | ArrayOutputSamplingConfig | *dict* | *None* = *None*)

Parameters

- **output_info** – Information about the output source, name and type.
- **output_type_info** – Information specific to the output type, e.g. which columns to predict from a tabular file.
- **model_config** – Configuration for the chosen model (i.e. output module after fusion) for this output.
- **sampling_config** – Configuration for how to sample results from the output module.

Output Info Configuration

class eir.setup.schemas.**OutputInfoConfig**(*output_source: str*, *output_name: str*, *output_type: Literal['tabular', 'sequence', 'array']*, *output_inner_key: str* | *None* = *None*)

Parameters

- **output_source** – Where on the filesystem to locate the output (if applicable)
- **output_name** – Name to identify the output.
- **output_type** – Type of the output.

Output Type Configuration

class eir.setup.schemas.**TabularOutputTypeConfig**(*target_cat_columns: ~typing.Sequence[str] = <factory>*, *target_con_columns: ~typing.Sequence[str] = <factory>*, *label_parsing_chunk_size: None | int = None*, *cat_label_smoothing: float = 0.0*, *cat_loss_name: ~typing.Literal['CrossEntropyLoss'] = 'CrossEntropyLoss'*, *con_loss_name: ~typing.Literal['MSELoss', 'L1Loss', 'SmoothL1Loss', 'PoissonNLLLoss', 'HuberLoss'] = 'MSELoss'*, *uncertainty_weighted_mt_loss: bool = True*)

Parameters

- **target_cat_columns** – Which columns from `label_file` to use as categorical targets.
- **target_con_columns** – Which columns from `label_file` to use as continuous targets.
- **label_parsing_chunk_size** – Number of rows to process at time when loading in the `input_source`. Useful when RAM is limited.

- **cat_label_smoothing** – Label smoothing to apply to categorical targets.
- **uncertainty_weighted_mt_loss** – Whether to use uncertainty weighted loss for multi-task / multilabel learning.

class eir.setup.schema_modules.output_schemas_sequence.**SequenceOutputTypeConfig**(*vocab_file: None | str = None, max_length: al_max_sequence_length = 'average', sampling_strategy_if_longer: Literal['from_start', 'uniform'] = 'uniform', min_freq: int = 10, split_on: str | None = ' ', tokenizer: al_tokenizer_choices = None, tokenizer_language: str | None = None, adaptive_tokenizer_max_vocab_siz int | None = None, sequence_operation: Literal['autoregressive', 'mlm'] = 'autoregressive'*)

Parameters

- **vocab_file** – An optional text file containing pre-defined vocabulary to use for the training. If this is not passed in, the framework will automatically build the vocabulary from the training data. Passing in a vocabulary file is therefore useful if (a) you want to manually specify / limit the vocabulary used and/or (b) you want to save time by pre-computing the vocabulary.
- **max_length** – Maximum length to truncate/pad sequences to. This can be an integer or the values 'max' or 'average'. The 'max' keyword will use the maximum sequence length found in the training data, while the 'average' will use the average length across all training samples.
- **sampling_strategy_if_longer** – Controls how sequences are truncated if they are longer than the specified `max_length` parameter. Using 'from_start' will always truncate from the beginning of the sequence, ensuring the the samples will always be the same during training. Setting this parameter to `uniform` will uniformly sample a slice of a given sample sequence during training. Note that for consistency, the validation/test set samples always use the `from_start` setting when truncating.

- **min_freq** – Minimum number of times a token must appear in the total training data to be included in the vocabulary. Note that this setting will not do anything if passing in `vocab_file`.
- **split_on** – Which token to split the sequence on to generate separate tokens for the vocabulary.
- **tokenizer** – Which tokenizer to use. Relevant if modelling on language, but not as much when doing it on other arbitrary sequences.
- **tokenizer_language** – Which language rules the tokenizer should apply when tokenizing the raw data.
- **adaptive_tokenizer_max_vocab_size** – If using an adaptive tokenizer ("bpe"), this parameter controls the maximum size of the vocabulary.
- **sequence_operation** – Which operation to perform on the sequence. Currently only `autoregressive` is supported, which means that the model will be trained to predict the next token in the sequence given the previous tokens.

class eir.setup.schema_modules.output_schemas_array.**ArrayOutputTypeConfig**(*normalization: Literal['element', 'channel'] | None = 'channel'*, *adaptive_normalization_max_samples: int | None = None*)

Parameters

- **normalization** – Which type of normalization to apply to the array data. If `element`, will normalize each element in the array independently. If `channel`, will normalize each channel in the array independently. For 'channel', assumes PyTorch format where the channel dimension is the first dimension.
- **adaptive_normalization_max_samples** – If using adaptive normalization (channel / element), how many samples to use to compute the normalization parameters. If None, will use all samples.

Output Module Configuration

Tabular Output Modules

class eir.models.output.tabular.tabular_output_modules.**TabularOutputModuleConfig**(*model_init_config: ResidualMLPOutputModuleConfig | LinearOutputModuleConfig*, *model_type: Literal['mlp_residual', 'linear'] = 'mlp_residual'*)

Parameters

- **model_init_config** – Configuration / arguments used to initialise model.
- **model_type** – Which type of image model to use.

The documentation below details what the parameters passed to the respective output output heads of the tabular output model. (trough the *model_init_config* field in the *--output_configs .yaml* files).

class eir.models.output.tabular.mlp_residual.**ResidualMLPOutputModuleConfig**(*layers: ~typing.List[int] = <factory>, fc_task_dim: int = 256, rb_do: float = 0.1, fc_do: float = 0.1, stochastic_depth_p: float = 0.1, final_layer_type: ~typing.Literal['linear'] | ~typing.Literal['mlp_residual'] = 'linear'*)

Parameters

- **layers** – Number of residual MLP residual blocks to use in the output module.
- **fc_task_dim** – Number of hidden nodes in each MLP residual block.
- **rb_do** – Dropout in each MLP residual block.
- **fc_do** – Dropout before final layer.
- **stochastic_depth_p** – Stochastic depth probability (probability of dropping input) for each residual block.
- **final_layer_type** – Which type of final layer to use to construct tabular output prediction.

class eir.models.output.tabular.linear.**LinearOutputModuleConfig**

Sequence Output Modules

class eir.models.output.sequence.sequence_output_modules.**SequenceOutputModuleConfig**(*model_init_config: TransformerSequenceOutputModuleConfig*, *model_type: Literal['sequence'] = 'sequence'*, *embedding_dim: int = 64*, *position: Literal['encode', 'embed'] = 'encode'*, *position_dropout: float = 0.1*, *projection_layer_type: Literal['auto', 'lcl', 'lcl_residual', 'linear'] = 'auto'*)

Parameters

- **model_init_config** – Configuration / arguments used to initialise model.
- **model_type** – Which type of image model to use.
- **embedding_dim** – Which dimension to use for the embeddings. If `None`, will automatically set this value based on the number of tokens and attention heads.
- **position** – Whether to encode the token position or use learnable position embeddings.
- **position_dropout** – Dropout for the positional encoding / embedding.

Array Output Modules

class eir.models.output.array.array_output_modules.**ArrayOutputModuleConfig**(*model_type: Literal['lcl', 'cnn']*, *model_init_config: LCLOutputModelConfig*, *pre_normalization: Literal['instancenorm', 'layernorm'] | None = None*)

Parameters

- **model_type** – Which type of image model to use.
- **model_init_config** – Configuration used to initialise model.

Output Sampling Configuration

class eir.setup.schema_modules.output_schemas_sequence.**SequenceOutputSamplingConfig**(*manual_inputs: Sequence[Dict[str, str]] = ()*, *n_eval_inputs: int = 10*, *generated_sequence_length: int = 64*, *top_k: int = 20*, *top_p: float = 0.9*)

Parameters

- **manual_inputs** – Manually specified inputs to use for sequence generation. This is useful if you want to generate sequences based on a specific input. Depending on the input type, different formats are expected:
  - `sequence`: A string written directly in the `.yaml` file.
  - `omics`: A file path to NumPy array of shape `(4, n_SNPs)` on disk.
  - `image`: An image file path on disk.
  - `tabular`: A mapping of `(column key: value)` written directly in the `.yaml` file.
  - `array`: A file path to NumPy array on disk.
  - `bytes`: A file path to a file on disk.
- **n_eval_inputs** – The number of inputs automatically sampled from the validation set for sequence generation.

- **generated_sequence_length** – The length of the output sequences that are generated.
- **top_k** – The number of top candidates to consider when sampling the next token in an output sequence. By default, the model considers the top 20 candidates
- **top_p** – The cumulative probability of the top candidates to consider when sampling the next token in an output sequence. For example, if top_p is 0.9, the model will stop sampling candidates once the cumulative probability of the most likely candidates reaches 0.9.

class eir.setup.schema_modules.output_schemas_array.**ArrayOutputSamplingConfig**(*manual_inputs: Sequence[dict[str, str]] = ()*, *n_eval_inputs: int = 10*)

Parameters

- **manual_inputs** – Manually specified inputs to use for sequence generation. This is useful if you want to generate sequences based on a specific input. Depending on the input type, different formats are expected:
  - `sequence`: A string written directly in the `.yaml` file.
  - `omics`: A file path to NumPy array of shape `(4, n_SNPs)` on disk.
  - `image`: An image file path on disk.
  - `tabular`: A mapping of `(column key: value)` written directly in the `.yaml` file.
  - `array`: A file path to NumPy array on disk.
  - `bytes`: A file path to a file on disk.
- **n_eval_inputs** – The number of inputs automatically sampled from the validation set for sequence generation.

2.6.2 Image Models

This page contains the list of external image models that can be used with EIR, coming from the great timm library.

There are 3 ways to use these models:

- Configure and train specific architectures (e.g. ResNet with chosen number of layers) from scratch.
- Train a specific architecture (e.g. `resnet18`) from scratch.
- Use a pre-trained model (e.g. `resnet18`) and fine-tune it.

Please refer to this page for more detailed information about configurable architectures, and this page for a list of pre-defined architectures, with the option of using pre-trained weights.

Configurable Models

The following models can be configured and trained from scratch.

The model type is specified in the `model_type` field of the configuration, while the model specific configuration is specified in the `model_init_config` field.

For example, the `ResNet` architecture includes the `layers` and `block` parameters, and can be configured as follows:

Listing 121: input_configurable_image_model.yaml

```yaml
input_info:
  input_source: eir_tutorials/a_using_eir/05_image_tutorial/data/hot_dog_not_hot_dog/
↪food_images
  input_name: hot_dog
  input_type: image

input_type_info:
  mixing_subtype: "cutmix"
  size:
    - 64

model_config:
  model_type: "ResNet"
  model_init_config:
    layers: [1, 1, 1, 1]
    block: "BasicBlock"

interpretation_config:
    num_samples_to_interpret: 30
```

class timm.models.beit.**Beit**(*img_size: int | ~typing.Tuple[int, int] = 224*, *patch_size: int | ~typing.Tuple[int, int] = 16*, *in_chans: int = 3*, *num_classes: int = 1000*, *global_pool: str = 'avg'*, *embed_dim: int = 768*, *depth: int = 12*, *num_heads: int = 12*, *qkv_bias: bool = True*, *mlp_ratio: float = 4.0*, *swiglu_mlp: bool = False*, *scale_mlp: bool = False*, *drop_rate: float = 0.0*, *pos_drop_rate: float = 0.0*, *proj_drop_rate: float = 0.0*, *attn_drop_rate: float = 0.0*, *drop_path_rate: float = 0.0*, *norm_layer: ~typing.Callable = <class 'timm.layers.norm.LayerNorm'>*, *init_values: float | None = None*, *use_abs_pos_emb: bool = True*, *use_rel_pos_bias: bool = False*, *use_shared_rel_pos_bias: bool = False*, *head_init_scale: float = 0.001*)

Vision Transformer with support for patch or hybrid CNN input stage

class timm.models.byobnet.**ByobNet**(*cfg: ByoModelCfg*, *num_classes: int = 1000*, *in_chans: int = 3*, *global_pool: str = 'avg'*, *output_stride: int = 32*, *img_size: int | Tuple[int, int] | None = None*, *drop_rate: float = 0.0*, *drop_path_rate: float = 0.0*, *zero_init_last: bool = True*, ***kwargs*)

'Bring-your-own-blocks' Net

A flexible network backbone that allows building model stem + blocks via dataclass cfg definition w/ factory functions for module instantiation.

Current assumption is that both stem and blocks are in conv-bn-act order (w/ block ending in act).

class timm.models.cait.**Cait**(*img_size=224*, *patch_size=16*, *in_chans=3*, *num_classes=1000*, *global_pool='token'*, *embed_dim=768*, *depth=12*, *num_heads=12*, *mlp_ratio=4.0*, *qkv_bias=True*, *drop_rate=0.0*, *pos_drop_rate=0.0*, *proj_drop_rate=0.0*, *attn_drop_rate=0.0*, *drop_path_rate=0.0*, *block_layers=<class 'timm.models.cait.LayerScaleBlock'>*, *block_layers_token=<class 'timm.models.cait.LayerScaleBlockClassAttn'>*, *patch_layer=<class 'timm.layers.patch_embed.PatchEmbed'>*, *norm_layer=functools.partial(<class 'torch.nn.modules.normalization.LayerNorm'>, eps=1e-06)*, *act_layer=<class 'torch.nn.modules.activation.GELU'>*, *attn_block=<class 'timm.models.cait.TalkingHeadAttn'>*, *mlp_block=<class 'timm.layers.mlp.Mlp'>*, *init_values=0.0001*, *attn_block_token_only=<class 'timm.models.cait.ClassAttn'>*, *mlp_block_token_only=<class 'timm.layers.mlp.Mlp'>*, *depth_token_only=2*, *mlp_ratio_token_only=4.0*)

class timm.models.coat.**CoaT**(*img_size=224*, *patch_size=16*, *in_chans=3*, *num_classes=1000*, *embed_dims=(64, 128, 320, 512)*, *serial_depths=(3, 4, 6, 3)*, *parallel_depth=0*, *num_heads=8*, *mlp_ratios=(4, 4, 4, 4)*, *qkv_bias=True*, *drop_rate=0.0*, *proj_drop_rate=0.0*, *attn_drop_rate=0.0*, *drop_path_rate=0.0*, *norm_layer=<class 'timm.layers.norm.LayerNorm'>*, *return_interm_layers=False*, *out_features=None*, *crpe_window=None*, *global_pool='token'*)

CoaT class.

class timm.models.convit.**ConVit**(*img_size=224*, *patch_size=16*, *in_chans=3*, *num_classes=1000*, *global_pool='token'*, *embed_dim=768*, *depth=12*, *num_heads=12*, *mlp_ratio=4.0*, *qkv_bias=False*, *drop_rate=0.0*, *pos_drop_rate=0.0*, *proj_drop_rate=0.0*, *attn_drop_rate=0.0*, *drop_path_rate=0.0*, *hybrid_backbone=None*, *norm_layer=<class 'timm.layers.norm.LayerNorm'>*, *local_up_to_layer=3*, *locality_strength=1.0*, *use_pos_embed=True*)

Vision Transformer with support for patch or hybrid CNN input stage

class timm.models.convmixer.**ConvMixer**(*dim*, *depth*, *kernel_size=9*, *patch_size=7*, *in_chans=3*, *num_classes=1000*, *global_pool='avg'*, *drop_rate=0.0*, *act_layer=<class 'torch.nn.modules.activation.GELU'>*, ***kwargs*)

class timm.models.convnext.**ConvNeXt**(*in_chans: int = 3*, *num_classes: int = 1000*, *global_pool: str = 'avg'*, *output_stride: int = 32*, *depths: Tuple[int, ...] = (3, 3, 9, 3)*, *dims: Tuple[int, ...] = (96, 192, 384, 768)*, *kernel_sizes: int | Tuple[int, ...] = 7*, *ls_init_value: float | None = 1e-06*, *stem_type: str = 'patch'*, *patch_size: int = 4*, *head_init_scale: float = 1.0*, *head_norm_first: bool = False*, *head_hidden_size: int | None = None*, *conv_mlp: bool = False*, *conv_bias: bool = True*, *use_grn: bool = False*, *act_layer: str | Callable = 'gelu'*, *norm_layer: str | Callable | None = None*, *norm_eps: float | None = None*, *drop_rate: float = 0.0*, *drop_path_rate: float = 0.0*)

A PyTorch impl of : *A ConvNet for the 2020s* - https://arxiv.org/pdf/2201.03545.pdf

class timm.models.crossvit.**CrossVit**(*img_size=224*, *img_scale=(1.0, 1.0)*, *patch_size=(8, 16)*, *in_chans=3*, *num_classes=1000*, *embed_dim=(192, 384)*, *depth=((1, 3, 1), (1, 3, 1), (1, 3, 1))*, *num_heads=(6, 12)*, *mlp_ratio=(2.0, 2.0, 4.0)*, *multi_conv=False*, *crop_scale=False*, *qkv_bias=True*, *drop_rate=0.0*, *pos_drop_rate=0.0*, *proj_drop_rate=0.0*, *attn_drop_rate=0.0*, *drop_path_rate=0.0*, *norm_layer=functools.partial(<class 'torch.nn.modules.normalization.LayerNorm'>, eps=1e-06)*, *global_pool='token'*)

Vision Transformer with support for patch or hybrid CNN input stage

class timm.models.cspnet.**CspNet**(*cfg: CspModelCfg*, *in_chans=3*, *num_classes=1000*, *output_stride=32*, *global_pool='avg'*, *drop_rate=0.0*, *drop_path_rate=0.0*, *zero_init_last=True*, ***kwargs*)

Cross Stage Partial base model.

Paper: *CSPNet: A New Backbone that can Enhance Learning Capability of CNN* - https://arxiv.org/abs/1911.11929 Ref Impl: https://github.com/WongKinYiu/CrossStagePartialNetworks

NOTE: There are differences in the way I handle the 1x1 'expansion' conv in this impl vs the darknet impl. I did it this way for simplicity and less special cases.

class timm.models.davit.**DaViT**(*in_chans=3*, *depths=(1, 1, 3, 1)*, *embed_dims=(96, 192, 384, 768)*, *num_heads=(3, 6, 12, 24)*, *window_size=7*, *mlp_ratio=4*, *qkv_bias=True*, *norm_layer='layernorm2d'*, *norm_layer_cl='layernorm'*, *norm_eps=1e-05*, *attn_types=('spatial', 'channel')*, *ffn=True*, *cpe_act=False*, *drop_rate=0.0*, *drop_path_rate=0.0*, *num_classes=1000*, *global_pool='avg'*, *head_norm_first=False*)

DaViT

A PyTorch implementation of *DaViT: Dual Attention Vision Transformers* - https://arxiv.org/abs/2204.03645 Supports arbitrary input sizes and pyramid feature extraction

Parameters

- **in_chans** (*int*) – Number of input image channels. Default: 3
- **num_classes** (*int*) – Number of classes for classification head. Default: 1000
- **depths** (*tuple(int)*) – Number of blocks in each stage. Default: (1, 1, 3, 1)
- **embed_dims** (*tuple(int)*) – Patch embedding dimension. Default: (96, 192, 384, 768)
- **num_heads** (*tuple(int)*) – Number of attention heads in different layers. Default: (3, 6, 12, 24)
- **window_size** (*int*) – Window size. Default: 7
- **mlp_ratio** (*float*) – Ratio of mlp hidden dim to embedding dim. Default: 4
- **qkv_bias** (*bool*) – If True, add a learnable bias to query, key, value. Default: True
- **drop_path_rate** (*float*) – Stochastic depth rate. Default: 0.1
- **norm_layer** (*nn.Module*) – Normalization layer. Default: nn.LayerNorm.

class timm.models.deit.**VisionTransformerDistilled**(**args*, ***kwargs*)

Vision Transformer w/ Distillation Token and Head

Distillation token & head support for *DeiT: Data-efficient Image Transformers*

- https://arxiv.org/abs/2012.12877

class timm.models.densenet.**DenseNet**(*growth_rate=32*, *block_config=(6, 12, 24, 16)*, *num_classes=1000*, *in_chans=3*, *global_pool='avg'*, *bn_size=4*, *stem_type=''*, *act_layer='relu'*, *norm_layer='batchnorm2d'*, *aa_layer=None*, *drop_rate=0.0*, *proj_drop_rate=0.0*, *memory_efficient=False*, *aa_stem_only=True*)

Densenet-BC model class, based on "Densely Connected Convolutional Networks"

Parameters

- **growth_rate** (int) - how many filters to add each layer (*k* in paper)
- **block_config** (*list of 4 ints*)
- **bn_size** (*int*) – (i.e. bn_size * k features in the bottleneck layer)
- **drop_rate** (*float*)
- **proj_drop_rate** (*float*)
- **num_classes** (*int*)
- **memory_efficient** (*bool*) – but slower. Default: *False*. See "paper"

class timm.models.dla.**DLA**(*levels*, *channels*, *output_stride=32*, *num_classes=1000*, *in_chans=3*, *global_pool='avg'*, *cardinality=1*, *base_width=64*, *block=<class 'timm.models.dla.DlaBottle2neck'>*, *shortcut_root=False*, *drop_rate=0.0*)

class timm.models.dpn.**DPN**(*k_sec=(3, 4, 20, 3)*, *inc_sec=(16, 32, 24, 128)*, *k_r=96*, *groups=32*, *num_classes=1000*, *in_chans=3*, *output_stride=32*, *global_pool='avg'*, *small=False*, *num_init_features=64*, *b=False*, *drop_rate=0.0*, *norm_layer='batchnorm2d'*, *act_layer='relu'*, *fc_act_layer='elu'*)

class timm.models.edgenext.**EdgeNeXt**(*in_chans=3*, *num_classes=1000*, *global_pool='avg'*, *dims=(24, 48, 88, 168)*, *depths=(3, 3, 9, 3)*, *global_block_counts=(0, 1, 1, 1)*, *kernel_sizes=(3, 5, 7, 9)*, *heads=(8, 8, 8, 8)*, *d2_scales=(2, 2, 3, 4)*, *use_pos_emb=(False, True, False, False)*, *ls_init_value=1e-06*, *head_init_scale=1.0*, *expand_ratio=4*, *downsample_block=False*, *conv_bias=True*, *stem_type='patch'*, *head_norm_first=False*, *act_layer=<class 'torch.nn.modules.activation.GELU'>*, *drop_path_rate=0.0*, *drop_rate=0.0*)

class timm.models.efficientformer.**EfficientFormer**(*depths*, *embed_dims=None*, *in_chans=3*, *num_classes=1000*, *global_pool='avg'*, *downsamples=None*, *num_vit=0*, *mlp_ratios=4*, *pool_size=3*, *layer_scale_init_value=1e-05*, *act_layer=<class 'torch.nn.modules.activation.GELU'>*, *norm_layer=<class 'torch.nn.modules.batchnorm.BatchNorm2d'>*, *norm_layer_cl=<class 'torch.nn.modules.normalization.LayerNorm'>*, *drop_rate=0.0*, *proj_drop_rate=0.0*, *drop_path_rate=0.0*, ***kwargs*)

class timm.models.efficientnet.**EfficientNet**(*block_args*, *num_classes=1000*, *num_features=1280*, *in_chans=3*, *stem_size=32*, *fix_stem=False*, *output_stride=32*, *pad_type=''*, *round_chs_fn=<function round_channels>*, *act_layer=None*, *norm_layer=None*, *se_layer=None*, *drop_rate=0.0*, *drop_path_rate=0.0*, *global_pool='avg'*)

A flexible and performant PyTorch implementation of efficient network architectures, including:

- EfficientNet-V2 Small, Medium, Large, XL & B0-B3
- EfficientNet B0-B8, L2
- EfficientNet-EdgeTPU
- EfficientNet-CondConv
- MixNet S, M, L, XL
- MnasNet A1, B1, and small
- MobileNet-V2
- FBNet C
- Single-Path NAS Pixel1
- TinyNet

`class timm.models.efficientvit_mit.EfficientVit(`*in_chans=3*, *widths=()*, *depths=()*, *head_dim=32*, *expand_ratio=4*, *norm_layer=<class 'torch.nn.modules.batchnorm.BatchNorm2d'>*, *act_layer=<class 'torch.nn.modules.activation.Hardswish'>*, *global_pool='avg'*, *head_widths=()*, *drop_rate=0.0*, *num_classes=1000*`)`

`class timm.models.efficientvit_msra.EfficientVitMsra(`*img_size=224*, *in_chans=3*, *num_classes=1000*, *embed_dim=(64, 128, 192)*, *key_dim=(16, 16, 16)*, *depth=(1, 2, 3)*, *num_heads=(4, 4, 4)*, *window_size=(7, 7, 7)*, *kernels=(5, 5, 5, 5)*, *down_ops=(('', 1), ('subsample', 2), ('subsample', 2))*, *global_pool='avg'*, *drop_rate=0.0*`)`

`class timm.models.eva.Eva(`*img_size: int | ~typing.Tuple[int, int] = 224*, *patch_size: int | ~typing.Tuple[int, int] = 16*, *in_chans: int = 3*, *num_classes: int = 1000*, *global_pool: str = 'avg'*, *embed_dim: int = 768*, *depth: int = 12*, *num_heads: int = 12*, *qkv_bias: bool = True*, *qkv_fused: bool = True*, *mlp_ratio: float = 4.0*, *swiglu_mlp: bool = False*, *scale_mlp: bool = False*, *scale_attn_inner: bool = False*, *drop_rate: float = 0.0*, *pos_drop_rate: float = 0.0*, *patch_drop_rate: float = 0.0*, *proj_drop_rate: float = 0.0*, *attn_drop_rate: float = 0.0*, *drop_path_rate: float = 0.0*, *norm_layer: ~typing.Callable = <class 'timm.layers.norm.LayerNorm'>*, *init_values: float | None = None*, *class_token: bool = True*, *use_abs_pos_emb: bool = True*, *use_rot_pos_emb: bool = False*, *use_post_norm: bool = False*, *dynamic_img_size: bool = False*, *dynamic_img_pad: bool = False*, *ref_feat_shape: int | ~typing.Tuple[int, int] | None = None*, *head_init_scale: float = 0.001*`)`

Eva Vision Transformer w/ Abs & Rotary Pos Embed

This class implements the EVA and EVA02 models that were based on the BEiT ViT variant

- EVA - abs pos embed, global avg pool
- EVA02 - abs + rope pos embed, global avg pool, SwiGLU, scale Norm in MLP (ala normformer)

class timm.models.focalnet.**FocalNet**(*in_chans: int = 3*, *num_classes: int = 1000*, *global_pool: str = 'avg'*, *embed_dim: int = 96*, *depths: ~typing.Tuple[int, ...] = (2, 2, 6, 2)*, *mlp_ratio: float = 4.0*, *focal_levels: ~typing.Tuple[int, ...] = (2, 2, 2, 2)*, *focal_windows: ~typing.Tuple[int, ...] = (3, 3, 3, 3)*, *use_overlap_down: bool = False*, *use_post_norm: bool = False*, *use_post_norm_in_modulation: bool = False*, *normalize_modulator: bool = False*, *head_hidden_size: int | None = None*, *head_init_scale: float = 1.0*, *layerscale_value: float | None = None*, *drop_rate: bool = 0.0*, *proj_drop_rate: bool = 0.0*, *drop_path_rate: bool = 0.1*, *norm_layer: ~typing.Callable = functools.partial(<class 'timm.layers.norm.LayerNorm2d'>, eps=1e-05)*)

" Focal Modulation Networks (FocalNets)

class timm.models.gcvit.**GlobalContextVit**(*in_chans: int = 3*, *num_classes: int = 1000*, *global_pool: str = 'avg'*, *img_size: Tuple[int, int] = 224*, *window_ratio: Tuple[int, ...] = (32, 32, 16, 32)*, *window_size: Tuple[int, ...] = None*, *embed_dim: int = 64*, *depths: Tuple[int, ...] = (3, 4, 19, 5)*, *num_heads: Tuple[int, ...] = (2, 4, 8, 16)*, *mlp_ratio: float = 3.0*, *qkv_bias: bool = True*, *layer_scale: float | None = None*, *drop_rate: float = 0.0*, *proj_drop_rate: float = 0.0*, *attn_drop_rate: float = 0.0*, *drop_path_rate: float = 0.0*, *weight_init=''*, *act_layer: str = 'gelu'*, *norm_layer: str = 'layernorm2d'*, *norm_layer_cl: str = 'layernorm'*, *norm_eps: float = 1e-05*)

class timm.models.ghostnet.**GhostNet**(*cfgs*, *num_classes=1000*, *width=1.0*, *in_chans=3*, *output_stride=32*, *global_pool='avg'*, *drop_rate=0.2*, *version='v1'*)

class timm.models.hgnet.**HighPerfGpuNet**(*cfg*, *in_chans=3*, *num_classes=1000*, *global_pool='avg'*, *use_last_conv=True*, *class_expand=2048*, *drop_rate=0.0*, *drop_path_rate=0.0*, *use_lab=False*, ***kwargs*)

class timm.models.hrnet.**HighResolutionNet**(*cfg*, *in_chans=3*, *num_classes=1000*, *output_stride=32*, *global_pool='avg'*, *drop_rate=0.0*, *head='classification'*, ***kwargs*)

class timm.models.inception_resnet_v2.**InceptionResnetV2**(*num_classes=1000*, *in_chans=3*, *drop_rate=0.0*, *output_stride=32*, *global_pool='avg'*, *norm_layer='batchnorm2d'*, *norm_eps=0.001*, *act_layer='relu'*)

class timm.models.inception_v3.**InceptionV3**(*num_classes=1000*, *in_chans=3*, *drop_rate=0.0*, *global_pool='avg'*, *aux_logits=False*, *norm_layer='batchnorm2d'*, *norm_eps=0.001*, *act_layer='relu'*)

Inception-V3

class timm.models.inception_v4.**InceptionV4**(*num_classes=1000*, *in_chans=3*, *output_stride=32*, *drop_rate=0.0*, *global_pool='avg'*, *norm_layer='batchnorm2d'*, *norm_eps=0.001*, *act_layer='relu'*)

class timm.models.levit.**Levit**(*img_size=224*, *in_chans=3*, *num_classes=1000*, *embed_dim=(192,)*, *key_dim=64*, *depth=(12,)*, *num_heads=(3,)*, *attn_ratio=2.0*, *mlp_ratio=2.0*, *stem_backbone=None*, *stem_stride=None*, *stem_type='s16'*, *down_op='subsample'*, *act_layer='hard_swish'*, *attn_act_layer=None*, *use_conv=False*, *global_pool='avg'*, *drop_rate=0.0*, *drop_path_rate=0.0*)

Vision Transformer with support for patch or hybrid CNN input stage

NOTE: distillation is defaulted to True since pretrained weights use it, will cause problems w/ train scripts that don't take tuple outputs,

class timm.models.maxxvit.**MaxxVitCfg**(*embed_dim: Tuple[int, ...] = (96, 192, 384, 768), depths: Tuple[int, ...] = (2, 3, 5, 2), block_type: Tuple[Union[str, Tuple[str, ...]], ...] = ('C', 'C', 'T', 'T'), stem_width: Union[int, Tuple[int, int]] = 64, stem_bias: bool = False, conv_cfg: timm.models.maxxvit.MaxxVitConvCfg = <factory>, transformer_cfg: timm.models.maxxvit.MaxxVitTransformerCfg = <factory>, head_hidden_size: int = None, weight_init: str = 'vit_eff'*)

class timm.models.metaformer.**MetaFormer**(*in_chans=3*, *num_classes=1000*, *global_pool='avg'*, *depths=(2, 2, 6, 2)*, *dims=(64, 128, 320, 512)*, *token_mixers=<class 'timm.models.metaformer.Pooling'>*, *mlp_act=<class 'timm.models.metaformer.StarReLU'>*, *mlp_bias=False*, *drop_path_rate=0.0*, *proj_drop_rate=0.0*, *drop_rate=0.0*, *layer_scale_init_values=None*, *res_scale_init_values=(None, None, 1.0, 1.0)*, *downsample_norm=<class 'timm.models.metaformer.LayerNorm2dNoBias'>*, *norm_layers=<class 'timm.models.metaformer.LayerNorm2dNoBias'>*, *output_norm=<class 'timm.layers.norm.LayerNorm2d'>*, *use_mlp_head=True*, ***kwargs*)

A PyTorch impl of
[*MetaFormer Baselines for Vision* -] https://arxiv.org/abs/2210.13452

Parameters

- **in_chans** (*int*) – Number of input image channels.
- **num_classes** (*int*) – Number of classes for classification head.
- **global_pool** – Pooling for classifier head.
- **depths** (*list or tuple*) – Number of blocks at each stage.
- **dims** (*list or tuple*) – Feature dimension at each stage.
- **token_mixers** (*list, tuple or token_fcn*) – Token mixer for each stage.
- **mlp_act** – Activation layer for MLP.
- **mlp_bias** (*boolean*) – Enable or disable mlp bias term.
- **drop_path_rate** (*float*) – Stochastic depth rate.
- **drop_rate** (*float*) – Dropout rate.
- **layer_scale_init_values** (*list, tuple, float or None*) – Init value for Layer Scale. None means not use the layer scale. Form: https://arxiv.org/abs/2103.17239.

- **res_scale_init_values** (*list, tuple, float or None*) – Init value for res Scale on residual connections. None means not use the res scale. From: https://arxiv.org/abs/2110.09456.
- **downsample_norm** (*nn.Module*) – Norm layer used in stem and downsampling layers.
- **norm_layers** (*list, tuple or norm_fcn*) – Norm layers for each stage.
- **output_norm** – Norm layer before classifier head.
- **use_mlp_head** – Use MLP classification head.

class timm.models.mobilenetv3.**MobileNetV3**(*block_args: ~typing.List[~typing.List[~typing.Dict[str, ~typing.Any]]], num_classes: int = 1000, in_chans: int = 3, stem_size: int = 16, fix_stem: bool = False, num_features: int = 1280, head_bias: bool = True, pad_type: str | int | ~typing.Tuple[int, int] = '', act_layer: str | ~typing.Callable | ~typing.Type[~torch.nn.modules.module.Module] | None = None, norm_layer: str | ~typing.Callable | ~typing.Type[~torch.nn.modules.module.Module] | None = None, se_layer: str | ~typing.Callable | ~typing.Type[~torch.nn.modules.module.Module] | None = None, se_from_exp: bool = True, round_chs_fn: ~typing.Callable = <function round_channels>, drop_rate: float = 0.0, drop_path_rate: float = 0.0, global_pool: str = 'avg'*)

MobiletNet-V3

Based on my EfficientNet implementation and building blocks, this model utilizes the MobileNet-v3 specific 'efficient head', where global pooling is done before the head convolution without a final batch-norm layer before the classifier.

Paper: *Searching for MobileNetV3* - https://arxiv.org/abs/1905.02244

Other architectures utilizing MobileNet-V3 efficient head that are supported by this impl include:

- HardCoRe-NAS - https://arxiv.org/abs/2102.11646 (defn in hardcorenas.py uses this class)
- FBNet-V3 - https://arxiv.org/abs/2006.02049
- LCNet - https://arxiv.org/abs/2109.15099

class timm.models.mvitv2.**MultiScaleVit**(*cfg: MultiScaleVitCfg*, *img_size: Tuple[int, int] = (224, 224)*, *in_chans: int = 3*, *global_pool: str | None = None*, *num_classes: int = 1000*, *drop_path_rate: float = 0.0*, *drop_rate: float = 0.0*)

Improved Multiscale Vision Transformers for Classification and Detection Yanghao Li*, Chao-Yuan Wu*, Haoqi Fan, Karttikeya Mangalam, Bo Xiong, Jitendra Malik,

Christoph Feichtenhofer*

https://arxiv.org/abs/2112.01526

Multiscale Vision Transformers Haoqi Fan*, Bo Xiong*, Karttikeya Mangalam*, Yanghao Li*, Zhicheng Yan, Jitendra Malik,

Christoph Feichtenhofer*

https://arxiv.org/abs/2104.11227

class timm.models.nasnet.**NASNetALarge**(*num_classes=1000*, *in_chans=3*, *stem_size=96*, *channel_multiplier=2*, *num_features=4032*, *output_stride=32*, *drop_rate=0.0*, *global_pool='avg'*, *pad_type='same'*)

NASNetALarge (6 @ 4032)

class timm.models.nest.**Nest**(*img_size=224*, *in_chans=3*, *patch_size=4*, *num_levels=3*, *embed_dims=(128, 256, 512)*, *num_heads=(4, 8, 16)*, *depths=(2, 2, 20)*, *num_classes=1000*, *mlp_ratio=4.0*, *qkv_bias=True*, *drop_rate=0.0*, *proj_drop_rate=0.0*, *attn_drop_rate=0.0*, *drop_path_rate=0.5*, *norm_layer=None*, *act_layer=None*, *pad_type=''*, *weight_init=''*, *global_pool='avg'*)

Nested Transformer (NesT)

A PyTorch impl of
[*Aggregating Nested Transformers*]

- https://arxiv.org/abs/2105.12723

class timm.models.nfnet.**NormFreeNet**(*cfg: NfCfg*, *num_classes: int = 1000*, *in_chans: int = 3*, *global_pool: str = 'avg'*, *output_stride: int = 32*, *drop_rate: float = 0.0*, *drop_path_rate: float = 0.0*, ***kwargs*)

Normalization-Free Network

As described in : *Characterizing signal propagation to close the performance gap in unnormalized ResNets*

- https://arxiv.org/abs/2101.08692

and *High-Performance Large-Scale Image Recognition Without Normalization* - https://arxiv.org/abs/2102.06171

This model aims to cover both the NFRegNet-Bx models as detailed in the paper's code snippets and the (preact) ResNet models described earlier in the paper.

There are a few differences:

- **channels are rounded to be divisible by 8 by default (keep tensor core kernels happy),**
  this changes channel dim and param counts slightly from the paper models
- **activation correcting gamma constants are moved into the ScaledStdConv as it has less performance**
  impact in PyTorch when done with the weight scaling there. This likely wasn't a concern in the JAX impl.
- **a config option *gamma_in_act* can be enabled to not apply gamma in StdConv as described above, but**
  apply it in each activation. This is slightly slower, numerically different, but matches official impl.
- **skipinit is disabled by default, it seems to have a rather drastic impact on GPU memory use and throughput**
  for what it is/does. Approx 8-10% throughput loss.

class timm.models.pit.**PoolingVisionTransformer**(*img_size: int = 224*, *patch_size: int = 16*, *stride: int = 8*, *stem_type: str = 'overlap'*, *base_dims: Sequence[int] = (48, 48, 48)*, *depth: Sequence[int] = (2, 6, 4)*, *heads: Sequence[int] = (2, 4, 8)*, *mlp_ratio: float = 4*, *num_classes=1000*, *in_chans=3*, *global_pool='token'*, *distilled=False*, *drop_rate=0.0*, *pos_drop_drate=0.0*, *proj_drop_rate=0.0*, *attn_drop_rate=0.0*, *drop_path_rate=0.0*)

Pooling-based Vision Transformer

A PyTorch implement of 'Rethinking Spatial Dimensions of Vision Transformers'

- https://arxiv.org/abs/2103.16302

class timm.models.pnasnet.**PNASNet5Large**(*num_classes=1000*, *in_chans=3*, *output_stride=32*, *drop_rate=0.0*, *global_pool='avg'*, *pad_type=''*)

class timm.models.pvt_v2.**PyramidVisionTransformerV2**(*in_chans=3*, *num_classes=1000*, *global_pool='avg'*, *depths=(3, 4, 6, 3)*, *embed_dims=(64, 128, 256, 512)*, *num_heads=(1, 2, 4, 8)*, *sr_ratios=(8, 4, 2, 1)*, *mlp_ratios=(8.0, 8.0, 4.0, 4.0)*, *qkv_bias=True*, *linear=False*, *drop_rate=0.0*, *proj_drop_rate=0.0*, *attn_drop_rate=0.0*, *drop_path_rate=0.0*, *norm_layer=<class 'timm.layers.norm.LayerNorm'>*)

class timm.models.regnet.**RegNet**(*cfg: RegNetCfg*, *in_chans=3*, *num_classes=1000*, *output_stride=32*, *global_pool='avg'*, *drop_rate=0.0*, *drop_path_rate=0.0*, *zero_init_last=True*, ***kwargs*)

RegNet-X, Y, and Z Models

Paper: https://arxiv.org/abs/2003.13678 Original Impl: https://github.com/facebookresearch/pycls/blob/master/pycls/models/regnet.py

class timm.models.repghost.**RepGhostNet**(*cfgs*, *num_classes=1000*, *width=1.0*, *in_chans=3*, *output_stride=32*, *global_pool='avg'*, *drop_rate=0.2*, *reparam=True*)

class timm.models.repvit.**RepVit**(*in_chans=3*, *img_size=224*, *embed_dim=(48,)*, *depth=(2,)*, *mlp_ratio=2*, *global_pool='avg'*, *kernel_size=3*, *num_classes=1000*, *act_layer=<class 'torch.nn.modules.activation.GELU'>*, *distillation=True*, *drop_rate=0.0*, *legacy=False*)

class timm.models.resnet.**ResNet**(*block: ~timm.models.resnet.BasicBlock | ~timm.models.resnet.Bottleneck, layers: ~typing.List[int], num_classes: int = 1000, in_chans: int = 3, output_stride: int = 32, global_pool: str = 'avg', cardinality: int = 1, base_width: int = 64, stem_width: int = 64, stem_type: str = '', replace_stem_pool: bool = False, block_reduce_first: int = 1, down_kernel_size: int = 1, avg_down: bool = False, act_layer: str | ~typing.Callable | ~typing.Type[~torch.nn.modules.module.Module] = <class 'torch.nn.modules.activation.ReLU'>, norm_layer: str | ~typing.Callable | ~typing.Type[~torch.nn.modules.module.Module] = <class 'torch.nn.modules.batchnorm.BatchNorm2d'>, aa_layer: ~typing.Type[~torch.nn.modules.module.Module] | None = None, drop_rate: float = 0.0, drop_path_rate: float = 0.0, drop_block_rate: float = 0.0, zero_init_last: bool = True, block_args: ~typing.Dict[str, ~typing.Any] | None = None*)

ResNet / ResNeXt / SE-ResNeXt / SE-Net

This class implements all variants of ResNet, ResNeXt, SE-ResNeXt, and SENet that

- have > 1 stride in the 3x3 conv layer of bottleneck
- have conv-bn-act ordering

This ResNet impl supports a number of stem and downsample options based on the v1c, v1d, v1e, and v1s variants included in the MXNet Gluon ResNetV1b model. The C and D variants are also discussed in the 'Bag of Tricks' paper: https://arxiv.org/pdf/1812.01187. The B variant is equivalent to torchvision default.

ResNet variants (the same modifications can be used in SE/ResNeXt models as well):

- normal, b - 7x7 stem, stem_width = 64, same as torchvision ResNet, NVIDIA ResNet 'v1.5', Gluon v1b
- c - 3 layer deep 3x3 stem, stem_width = 32 (32, 32, 64)
- d - 3 layer deep 3x3 stem, stem_width = 32 (32, 32, 64), average pool in downsample
- e - 3 layer deep 3x3 stem, stem_width = 64 (64, 64, 128), average pool in downsample
- s - 3 layer deep 3x3 stem, stem_width = 64 (64, 64, 128)
- t - 3 layer deep 3x3 stem, stem width = 32 (24, 48, 64), average pool in downsample
- tn - 3 layer deep 3x3 stem, stem width = 32 (24, 32, 64), average pool in downsample

ResNeXt

- normal - 7x7 stem, stem_width = 64, standard cardinality and base widths
- same c,d, e, s variants as ResNet can be enabled

SE-ResNeXt

- normal - 7x7 stem, stem_width = 64
- same c, d, e, s variants as ResNet can be enabled

SENet-154 - 3 layer deep 3x3 stem (same as v1c-v1s), stem_width = 64, cardinality=64,
reduction by 2 on width of first bottleneck convolution, 3x3 downsample convs after first block

class timm.models.resnetv2.**ResNetV2**(*layers*, *channels=(256, 512, 1024, 2048)*, *num_classes=1000*, *in_chans=3*, *global_pool='avg'*, *output_stride=32*, *width_factor=1*, *stem_chs=64*, *stem_type=''*, *avg_down=False*, *preact=True*, *act_layer=<class 'torch.nn.modules.activation.ReLU'>*, *norm_layer=functools.partial(<class 'timm.layers.norm_act.GroupNormAct'>, num_groups=32)*, *conv_layer=<class 'timm.layers.std_conv.StdConv2d'>*, *drop_rate=0.0*, *drop_path_rate=0.0*, *zero_init_last=False*)

Implementation of Pre-activation (v2) ResNet mode.

class timm.models.rexnet.**RexNet**(*in_chans=3*, *num_classes=1000*, *global_pool='avg'*, *output_stride=32*, *initial_chs=16*, *final_chs=180*, *width_mult=1.0*, *depth_mult=1.0*, *se_ratio=0.0833333333333333*, *ch_div=1*, *act_layer='swish'*, *dw_act_layer='relu6'*, *drop_rate=0.2*, *drop_path_rate=0.0*)

class timm.models.selecsls.**SelecSls**(*cfg*, *num_classes=1000*, *in_chans=3*, *drop_rate=0.0*, *global_pool='avg'*)

SelecSls42 / SelecSls60 / SelecSls84

Parameters

- **cfg** (*network config dictionary specifying block type, feature, and head args*)
- **num_classes** (*int, default 1000*) – Number of classification classes.
- **in_chans** (*int, default 3*) – Number of input (color) channels.
- **drop_rate** (*float, default 0.*) – Dropout probability before classifier, for training
- **global_pool** (*str, default 'avg'*) – Global pooling type. One of 'avg', 'max', 'avgmax', 'catavgmax'

class timm.models.senet.**SENet**(*block*, *layers*, *groups*, *reduction*, *drop_rate=0.2*, *in_chans=3*, *inplanes=64*, *input_3x3=False*, *downsample_kernel_size=1*, *downsample_padding=0*, *num_classes=1000*, *global_pool='avg'*)

class timm.models.sequencer.**Sequencer2d**(*num_classes=1000*, *img_size=224*, *in_chans=3*, *global_pool='avg'*, *layers=(4, 3, 8, 3)*, *patch_sizes=(7, 2, 2, 1)*, *embed_dims=(192, 384, 384, 384)*, *hidden_sizes=(48, 96, 96, 96)*, *mlp_ratios=(3.0, 3.0, 3.0, 3.0)*, *block_layer=<class 'timm.models.sequencer.Sequencer2dBlock'>*, *rnn_layer=<class 'timm.models.sequencer.LSTM2d'>*, *mlp_layer=<class 'timm.layers.mlp.Mlp'>*, *norm_layer=functools.partial(<class 'torch.nn.modules.normalization.LayerNorm'>, eps=1e-06)*, *act_layer=<class 'torch.nn.modules.activation.GELU'>*, *num_rnn_layers=1*, *bidirectional=True*, *union='cat'*, *with_fc=True*, *drop_rate=0.0*, *drop_path_rate=0.0*, *nlhb=False*, *stem_norm=False*)

class timm.models.swin_transformer.**SwinTransformer**(*img_size: int | ~typing.Tuple[int, int] = 224*, *patch_size: int = 4*, *in_chans: int = 3*, *num_classes: int = 1000*, *global_pool: str = 'avg'*, *embed_dim: int = 96*, *depths: ~typing.Tuple[int, ...] = (2, 2, 6, 2)*, *num_heads: ~typing.Tuple[int, ...] = (3, 6, 12, 24)*, *head_dim: int | None = None*, *window_size: int | ~typing.Tuple[int, int] = 7*, *mlp_ratio: float = 4.0*, *qkv_bias: bool = True*, *drop_rate: float = 0.0*, *proj_drop_rate: float = 0.0*, *attn_drop_rate: float = 0.0*, *drop_path_rate: float = 0.1*, *embed_layer: ~typing.Callable = <class 'timm.layers.patch_embed.PatchEmbed'>*, *norm_layer: str | ~typing.Callable = <class 'torch.nn.modules.normalization.LayerNorm'>*, *weight_init: str = ''*, ***kwargs*)

Swin Transformer

A PyTorch impl of
[*Swin Transformer: Hierarchical Vision Transformer using Shifted Windows* -] https://arxiv.org/pdf/2103.14030

class timm.models.swin_transformer_v2.**SwinTransformerV2**(*img_size: int | ~typing.Tuple[int, int] = 224*, *patch_size: int = 4*, *in_chans: int = 3*, *num_classes: int = 1000*, *global_pool: str = 'avg'*, *embed_dim: int = 96*, *depths: ~typing.Tuple[int, ...] = (2, 2, 6, 2)*, *num_heads: ~typing.Tuple[int, ...] = (3, 6, 12, 24)*, *window_size: int | ~typing.Tuple[int, int] = 7*, *mlp_ratio: float = 4.0*, *qkv_bias: bool = True*, *drop_rate: float = 0.0*, *proj_drop_rate: float = 0.0*, *attn_drop_rate: float = 0.0*, *drop_path_rate: float = 0.1*, *norm_layer: ~typing.Callable = <class 'torch.nn.modules.normalization.LayerNorm'>*, *pretrained_window_sizes: ~typing.Tuple[int, ...] = (0, 0, 0, 0)*, ***kwargs*)

Swin Transformer V2

A PyTorch impl of
[*Swin Transformer V2: Scaling Up Capacity and Resolution*]

- https://arxiv.org/abs/2111.09883

class timm.models.swin_transformer_v2_cr.**SwinTransformerV2Cr**(*img_size: ~typing.Tuple[int, int] = (224, 224), patch_size: int = 4, window_size: int | None = None, img_window_ratio: int = 32, in_chans: int = 3, num_classes: int = 1000, embed_dim: int = 96, depths: ~typing.Tuple[int, ...] = (2, 2, 6, 2), num_heads: ~typing.Tuple[int, ...] = (3, 6, 12, 24), mlp_ratio: float = 4.0, init_values: float | None = 0.0, drop_rate: float = 0.0, proj_drop_rate: float = 0.0, attn_drop_rate: float = 0.0, drop_path_rate: float = 0.0, norm_layer: ~typing.Type[~torch.nn.modules.module.Module] = <class 'torch.nn.modules.normalization.LayerNorm'>, extra_norm_period: int = 0, extra_norm_stage: bool = False, sequential_attn: bool = False, global_pool: str = 'avg', weight_init='skip', **kwargs: ~typing.Any*)

Swin Transformer V2

A PyTorch impl of
[*Swin Transformer V2: Scaling Up Capacity and Resolution* -] https://arxiv.org/pdf/2111.09883

Parameters

- **img_size** – Input resolution.
- **window_size** – Window size. If None, img_size // window_div
- **img_window_ratio** – Window size to image size ratio.
- **patch_size** – Patch size.
- **in_chans** – Number of input channels.
- **depths** – Depth of the stage (number of layers).
- **num_heads** – Number of attention heads to be utilized.
- **embed_dim** – Patch embedding dimension.
- **num_classes** – Number of output classes.
- **mlp_ratio** – Ratio of the hidden dimension in the FFN to the input channels.
- **drop_rate** – Dropout rate.

- **proj_drop_rate** – Projection dropout rate.
- **attn_drop_rate** – Dropout rate of attention map.
- **drop_path_rate** – Stochastic depth rate.
- **norm_layer** – Type of normalization layer to be utilized.
- **extra_norm_period** – Insert extra norm layer on main branch every N (period) blocks in stage
- **extra_norm_stage** – End each stage with an extra norm layer in main branch
- **sequential_attn** – If true sequential self-attention is performed.

get_classifier() → Module

Method returns the classification head of the model. :returns: Current classification head :rtype: head (nn.Module)

reset_classifier(*num_classes: int*, *global_pool: str | None = None*) → None

Method results the classification head

Parameters

- **num_classes** (*int*) – Number of classes to be predicted
- **global_pool** (*str*) – Unused

update_input_size(*new_img_size: Tuple[int, int] | None = None*, *new_window_size: int | None = None*, *img_window_ratio: int = 32*) → None

Method updates the image resolution to be processed and window size and so the pair-wise relative positions.

Parameters

- **new_window_size** (*Optional[int]*) – New window size, if None based on new_img_size // window_div
- **new_img_size** (*Optional[Tuple[int, int]]*) – New input resolution, if None current resolution is used
- **img_window_ratio** (*int*) – divisor for calculating window size from image size

class timm.models.tiny_vit.**TinyVit**(*in_chans=3*, *num_classes=1000*, *global_pool='avg'*, *embed_dims=(96, 192, 384, 768)*, *depths=(2, 2, 6, 2)*, *num_heads=(3, 6, 12, 24)*, *window_sizes=(7, 7, 14, 7)*, *mlp_ratio=4.0*, *drop_rate=0.0*, *drop_path_rate=0.1*, *use_checkpoint=False*, *mbconv_expand_ratio=4.0*, *local_conv_size=3*, *act_layer=<class 'torch.nn.modules.activation.GELU'>*)

class timm.models.tnt.**TNT**(*img_size=224*, *patch_size=16*, *in_chans=3*, *num_classes=1000*, *global_pool='token'*, *embed_dim=768*, *inner_dim=48*, *depth=12*, *num_heads_inner=4*, *num_heads_outer=12*, *mlp_ratio=4.0*, *qkv_bias=False*, *drop_rate=0.0*, *pos_drop_rate=0.0*, *proj_drop_rate=0.0*, *attn_drop_rate=0.0*, *drop_path_rate=0.0*, *norm_layer=<class 'torch.nn.modules.normalization.LayerNorm'>*, *first_stride=4*)

Transformer in Transformer - https://arxiv.org/abs/2103.00112

class timm.models.tresnet.**TResNet**(*layers*, *in_chans=3*, *num_classes=1000*, *width_factor=1.0*, *v2=False*, *global_pool='fast'*, *drop_rate=0.0*, *drop_path_rate=0.0*)

class timm.models.twins.**Twins**(*img_size=224*, *patch_size=4*, *in_chans=3*, *num_classes=1000*, *global_pool='avg'*, *embed_dims=(64*, *128*, *256*, *512)*, *num_heads=(1*, *2*, *4*, *8)*, *mlp_ratios=(4*, *4*, *4*, *4)*, *depths=(3*, *4*, *6*, *3)*, *sr_ratios=(8*, *4*, *2*, *1)*, *wss=None*, *drop_rate=0.0*, *pos_drop_rate=0.0*, *proj_drop_rate=0.0*, *attn_drop_rate=0.0*, *drop_path_rate=0.0*, *norm_layer=functools.partial(<class 'torch.nn.modules.normalization.LayerNorm'>*, *eps=1e-06)*, *block_cls=<class 'timm.models.twins.Block'>*)

Twins Vision Transfomer (Revisiting Spatial Attention)

Adapted from PVT (PyramidVisionTransformer) class at https://github.com/whai362/PVT.git

class timm.models.vgg.**VGG**(*cfg: ~typing.List[~typing.Any]*, *num_classes: int = 1000*, *in_chans: int = 3*, *output_stride: int = 32*, *mlp_ratio: float = 1.0*, *act_layer: ~torch.nn.modules.module.Module = <class 'torch.nn.modules.activation.ReLU'>*, *conv_layer: ~torch.nn.modules.module.Module = <class 'torch.nn.modules.conv.Conv2d'>*, *norm_layer: ~torch.nn.modules.module.Module = None*, *global_pool: str = 'avg'*, *drop_rate: float = 0.0*)

class timm.models.visformer.**Visformer**(*img_size=224*, *patch_size=16*, *in_chans=3*, *num_classes=1000*, *init_channels=32*, *embed_dim=384*, *depth=12*, *num_heads=6*, *mlp_ratio=4.0*, *drop_rate=0.0*, *pos_drop_rate=0.0*, *proj_drop_rate=0.0*, *attn_drop_rate=0.0*, *drop_path_rate=0.0*, *norm_layer=<class 'timm.layers.norm.LayerNorm2d'>*, *attn_stage='111'*, *use_pos_embed=True*, *spatial_conv='111'*, *vit_stem=False*, *group=8*, *global_pool='avg'*, *conv_init=False*, *embed_norm=None*)

class timm.models.vision_transformer.**VisionTransformer**(*img_size: int | ~typing.Tuple[int, int] = 224*, *patch_size: int | ~typing.Tuple[int, int] = 16*, *in_chans: int = 3*, *num_classes: int = 1000*, *global_pool: ~typing.Literal['', 'avg', 'token', 'map'] = 'token'*, *embed_dim: int = 768*, *depth: int = 12*, *num_heads: int = 12*, *mlp_ratio: float = 4.0*, *qkv_bias: bool = True*, *qk_norm: bool = False*, *init_values: float | None = None*, *class_token: bool = True*, *no_embed_class: bool = False*, *reg_tokens: int = 0*, *pre_norm: bool = False*, *fc_norm: bool | None = None*, *dynamic_img_size: bool = False*, *dynamic_img_pad: bool = False*, *drop_rate: float = 0.0*, *pos_drop_rate: float = 0.0*, *patch_drop_rate: float = 0.0*, *proj_drop_rate: float = 0.0*, *attn_drop_rate: float = 0.0*, *drop_path_rate: float = 0.0*, *weight_init: ~typing.Literal['skip', 'jax', 'jax_nlhb', 'moco', ''] = ''*, *fix_init: bool = False*, *embed_layer: ~typing.Callable = <class 'timm.layers.patch_embed.PatchEmbed'>*, *norm_layer: str | ~typing.Callable | ~typing.Type[~torch.nn.modules.module.Module] | None = None*, *act_layer: str | ~typing.Callable | ~typing.Type[~torch.nn.modules.module.Module] | None = None*, *block_fn: ~typing.Type[~torch.nn.modules.module.Module] = <class 'timm.models.vision_transformer.Block'>*, *mlp_layer: ~typing.Type[~torch.nn.modules.module.Module] = <class 'timm.layers.mlp.Mlp'>*)

Vision Transformer

A PyTorch impl of

[*An Image is Worth 16x16 Words: Transformers for Image Recognition at Scale*]

- https://arxiv.org/abs/2010.11929

get_intermediate_layers(*x: Tensor*, *n: int | Sequence = 1*, *reshape: bool = False*, *return_prefix_tokens: bool = False*, *norm: bool = False*) → Tuple[Tensor | Tuple[Tensor]]

Intermediate layer accessor (NOTE: This is a WIP experiment). Inspired by DINO / DINOv2 interface

class timm.models.vision_transformer_relpos.**VisionTransformerRelPos**(*img_size: int | ~typing.Tuple[int, int] = 224, patch_size: int | ~typing.Tuple[int, int] = 16, in_chans: int = 3, num_classes: int = 1000, global_pool: ~typing.Literal['', 'avg', 'token', 'map'] = 'avg', embed_dim: int = 768, depth: int = 12, num_heads: int = 12, mlp_ratio: float = 4.0, qkv_bias: bool = True, qk_norm: bool = False, init_values: float | None = 1e-06, class_token: bool = False, fc_norm: bool = False, rel_pos_type: str = 'mlp', rel_pos_dim: int | None = None, shared_rel_pos: bool = False, drop_rate: float = 0.0, proj_drop_rate: float = 0.0, attn_drop_rate: float = 0.0, drop_path_rate: float = 0.0, weight_init: ~typing.Literal['skip', 'jax', 'moco', ''] = 'skip', fix_init: bool = False, embed_layer: ~typing.Type[~torch.nn.modules.module.Module] = <class 'timm.layers.patch_embed.PatchEmbed'>, norm_layer: str | ~typing.Callable | ~typing.Type[~torch.nn.modules.module.Module] | None = None, act_layer: str | ~typing.Callable | ~typing.Type[~torch.nn.modules.module.Module] | None = None, block_fn: ~typing.Type[~torch.nn.modules.module.Module] = <class 'timm.models.vision_transformer_relpos.RelPo*

Vision Transformer w/ Relative Position Bias

Differing from classic vit, this impl

- uses relative position index (swin v1 / beit) or relative log coord + mlp (swin v2) pos embed
- defaults to no class token (can be enabled)
- defaults to global avg pool for head (can be changed)

- layer-scale (residual branch gain) enabled

class timm.models.vision_transformer_sam.**VisionTransformerSAM**(*img_size: int = 1024, patch_size: int = 16, in_chans: int = 3, num_classes: int = 768, embed_dim: int = 768, depth: int = 12, num_heads: int = 12, mlp_ratio: float = 4.0, qkv_bias: bool = True, qk_norm: bool = False, init_values: float | None = None, pre_norm: bool = False, drop_rate: float = 0.0, pos_drop_rate: float = 0.0, patch_drop_rate: float = 0.0, proj_drop_rate: float = 0.0, attn_drop_rate: float = 0.0, drop_path_rate: float = 0.0, weight_init: str = '', embed_layer: ~typing.Callable = functools.partial(<class 'timm.layers.patch_embed.PatchEmbed'>, output_fmt=<Format.NHWC: 'NHWC'>, strict_img_size=False), norm_layer: ~typing.Callable | None = <class 'torch.nn.modules.normalization.LayerNorm'>, act_layer: ~typing.Callable | None = <class 'torch.nn.modules.activation.GELU'>, block_fn: ~typing.Callable = <class 'timm.models.vision_transformer_sam.Block'>, mlp_layer: ~typing.Callable = <class 'timm.layers.mlp.Mlp'>, use_abs_pos: bool = True, use_rel_pos: bool = False, use_rope: bool = False, window_size: int = 14, global_attn_indexes: ~typing.Tuple[int, ...] = (), neck_chans: int = 256, global_pool: str = 'avg', head_hidden_size: int | None = None, ref_feat_shape: ~typing.Tuple[~typing.Tuple[int, int], ~typing.Tuple[int, int]] | None = None*)

Vision Transformer for Segment-Anything Model(SAM)

A PyTorch impl of

[*Exploring Plain Vision Transformer Backbones for Object Detection* or *Segment Anything Model (SAM)*]

- https://arxiv.org/abs/2010.11929

class timm.models.volo.**VOLO**(*layers*, *img_size=224*, *in_chans=3*, *num_classes=1000*, *global_pool='token'*, *patch_size=8*, *stem_hidden_dim=64*, *embed_dims=None*, *num_heads=None*, *downsamples=(True*, *False*, *False*, *False)*, *outlook_attention=(True*, *False*, *False*, *False)*, *mlp_ratio=3.0*, *qkv_bias=False*, *drop_rate=0.0*, *pos_drop_rate=0.0*, *attn_drop_rate=0.0*, *drop_path_rate=0.0*, *norm_layer=<class 'torch.nn.modules.normalization.LayerNorm'>*, *post_layers=('ca'*, *'ca')*, *use_aux_head=True*, *use_mix_token=False*, *pooling_scale=2*)

Vision Outlooker, the main class of our model

forward_train(*x*)

A separate forward fn for training with mix_token (if a train script supports). Combining multiple modes in as single forward with different return types is torchscript hell.

class timm.models.vovnet.**VovNet**(*cfg*, *in_chans=3*, *num_classes=1000*, *global_pool='avg'*, *output_stride=32*, *norm_layer=<class 'timm.layers.norm_act.BatchNormAct2d'>*, *act_layer=<class 'torch.nn.modules.activation.ReLU'>*, *drop_rate=0.0*, *drop_path_rate=0.0*, ***kwargs*)

class timm.models.xception.**Xception**(*num_classes=1000*, *in_chans=3*, *drop_rate=0.0*, *global_pool='avg'*)

Xception optimized for the ImageNet dataset, as specified in https://arxiv.org/pdf/1610.02357.pdf

class timm.models.xception_aligned.**XceptionAligned**(*block_cfg: ~typing.List[~typing.Dict]*, *num_classes: int = 1000*, *in_chans: int = 3*, *output_stride: int = 32*, *preact: bool = False*, *act_layer: ~typing.Type[~torch.nn.modules.module.Module] = <class 'torch.nn.modules.activation.ReLU'>*, *norm_layer: ~typing.Type[~torch.nn.modules.module.Module] = <class 'torch.nn.modules.batchnorm.BatchNorm2d'>*, *drop_rate: float = 0.0*, *drop_path_rate: float = 0.0*, *global_pool: str = 'avg'*)

Modified Aligned Xception

class timm.models.xcit.**Xcit**(*img_size=224*, *patch_size=16*, *in_chans=3*, *num_classes=1000*, *global_pool='token'*, *embed_dim=768*, *depth=12*, *num_heads=12*, *mlp_ratio=4.0*, *qkv_bias=True*, *drop_rate=0.0*, *pos_drop_rate=0.0*, *proj_drop_rate=0.0*, *attn_drop_rate=0.0*, *drop_path_rate=0.0*, *act_layer=None*, *norm_layer=None*, *cls_attn_layers=2*, *use_pos_embed=True*, *eta=1.0*, *tokens_norm=False*)

Based on timm and DeiT code bases https://github.com/rwightman/pytorch-image-models/tree/master/timm https://github.com/facebookresearch/deit/

2.6.3 Sequence Models

This page contains the list of external sequence models that can be used with EIR, coming from the excellent Transformers library.

There are 3 ways to use these models:

- Configure and train specific architectures (e.g. BERT with chosen number of layers) from scratch.
- Train a specific architecture (e.g. `bert-base-uncased`) from scratch.
- Use a pre-trained model (e.g. `bert-base-uncased`) and fine-tune it.

Please refer to this page for a complete list of pre-defined architectures, with the option of using pre-trained weights.

Configurable Models

The following models can be configured and trained from scratch.

The model type is specified in the `model_type` field of the configuration, while the model specific configuration is specified in the `model_init_config` field.

For example, the `LongFormer` architecture includes the `num_attention_heads` and `num_hidden_layers` parameters, and can be configured as follows:

Listing 122: input_configurable_sequence_model.yaml

```yaml
input_info:
  input_source: eir_tutorials/a_using_eir/04_pretrained_sequence_tutorial/data/IMDB/IMDB_
↪Reviews
  input_name: imdb_reviews_longformer
  input_type: sequence

input_type_info:
        sampling_strategy_if_longer: "uniform"
        max_length: 512
        split_on: " "
        min_freq: 10
        tokenizer: "basic_english"
        tokenizer_language: "en"

model_config:
      model_type: longformer
      pretrained_model: false
      position: embed
      pool: avg
      model_init_config:
            num_hidden_layers: 2
            hidden_size: 32
            num_attention_heads: 2
            intermediate_size: 32
            attention_window: 64
            max_position_embeddings: 1024
```

Pretrained Models

We can also fine-tune or train a specific architecture from scratch. For example, a `tiny-bert` model like so:

Listing 123: input_pre_trained_sequence_model.yaml

```yaml
input_info:
  input_source: eir_tutorials/a_using_eir/04_pretrained_sequence_tutorial/data/IMDB/IMDB_
↪Reviews
  input_name: imdb_reviews_tiny_bert
  input_type: sequence

input_type_info:
  sampling_strategy_if_longer: "uniform"
  max_length: 512
  split_on: " "
  min_freq: 10

model_config:
    model_type: "prajjwal1/bert-tiny"
    pretrained_model: true
    freeze_pretrained_model: false
    position: embed
    pool: avg
```

Below is a list of the configurable models that can be used with EIR.

class transformers.models.albert.configuration_albert.**AlbertConfig**(*vocab_size=30000*, *embedding_size=128*, *hidden_size=4096*, *num_hidden_layers=12*, *num_hidden_groups=1*, *num_attention_heads=64*, *intermediate_size=16384*, *inner_group_num=1*, *hidden_act='gelu_new'*, *hidden_dropout_prob=0*, *attention_probs_dropout_prob=0*, *max_position_embeddings=512*, *type_vocab_size=2*, *initializer_range=0.02*, *layer_norm_eps=1e-12*, *classifier_dropout_prob=0.1*, *position_embedding_type='absolute'*, *pad_token_id=0*, *bos_token_id=2*, *eos_token_id=3*, ***kwargs*)

The ALBERT model was proposed in ALBERT: A Lite BERT for Self-supervised Learning of Language Representations by Zhenzhong Lan, Mingda Chen, Sebastian Goodman, Kevin Gimpel, Piyush Sharma, Radu Soricut. It presents two parameter-reduction techniques to lower memory consumption and increase the training speed of BERT:

- Splitting the embedding matrix into two smaller matrices.
- Using repeating layers split among groups.

The abstract from the paper is the following:

Increasing model size when pretraining natural language representations often results in improved performance on downstream tasks. However, at some point further model increases become harder due to GPU/TPU memory limitations, longer training times, and unexpected model degradation. To address these problems, we present two parameter-reduction techniques to lower memory consumption and increase the training speed of BERT. Comprehensive empirical evidence shows that our proposed methods lead to models that scale much better compared to the original BERT. We also use a self-supervised loss that focuses on modeling inter-sentence coherence, and show it consistently helps downstream tasks with multi-sentence inputs. As a result, our best model establishes new state-of-the-art results on the GLUE, RACE, and SQuAD benchmarks while having fewer parameters compared to BERT-large.

Tips:

- ALBERT is a model with absolute position embeddings so it's usually advised to pad the inputs on the right rather than the left.
- ALBERT uses repeating layers which results in a small memory footprint, however the computational cost remains similar to a BERT-like architecture with the same number of hidden layers as it has to iterate through the same number of (repeating) layers.
- Embedding size E is different from hidden size H justified because the embeddings are context independent (one embedding vector represents one token), whereas hidden states are context dependent (one hidden state represents a sequence of tokens) so it's more logical to have H >> E. Also, the embedding matrix is large since it's V x E (V being the vocab size). If E < H, it has less parameters.
- Layers are split in groups that share parameters (to save memory).

Next sentence prediction is replaced by a sentence ordering prediction: in the inputs, we have two sentences A and B (that are consecutive) and we either feed A followed by B or B followed by A. The model must predict if they have been swapped or not.

This model was contributed by lysandre. This model jax version was contributed by kamalkraj. The original code can be found here.

Args:

vocab_size (*int*, *optional*, defaults to 30000):
Vocabulary size of the ALBERT model. Defines the number of different tokens that can be represented by the *inputs_ids* passed when calling `AlbertModel` or `TFAlbertModel`.

embedding_size (*int*, *optional*, defaults to 128):
Dimensionality of vocabulary embeddings.

hidden_size (*int*, *optional*, defaults to 4096):
Dimensionality of the encoder layers and the pooler layer.

num_hidden_layers (*int*, *optional*, defaults to 12):
Number of hidden layers in the Transformer encoder.

num_hidden_groups (*int*, *optional*, defaults to 1):
Number of groups for the hidden layers, parameters in the same group are shared.

num_attention_heads (*int*, *optional*, defaults to 64):
Number of attention heads for each attention layer in the Transformer encoder.

intermediate_size (*int*, *optional*, defaults to 16384):
The dimensionality of the "intermediate" (often named feed-forward) layer in the Transformer encoder.

inner_group_num (*int*, *optional*, defaults to 1):
The number of inner repetition of attention and ffn.

hidden_act (*str* or *Callable*, *optional*, defaults to *"gelu_new"*):
The non-linear activation function (function or string) in the encoder and pooler. If string, *"gelu"*, *"relu"*, *"silu"* and *"gelu_new"* are supported.

hidden_dropout_prob (*float*, *optional*, defaults to 0):
The dropout probability for all fully connected layers in the embeddings, encoder, and pooler.

attention_probs_dropout_prob (*float*, *optional*, defaults to 0):
The dropout ratio for the attention probabilities.

max_position_embeddings (*int*, *optional*, defaults to 512):
The maximum sequence length that this model might ever be used with. Typically set this to something large (e.g., 512 or 1024 or 2048).

type_vocab_size (*int*, *optional*, defaults to 2):
The vocabulary size of the *token_type_ids* passed when calling `AlbertModel` or `TFAlbertModel`.

initializer_range (*float*, *optional*, defaults to 0.02):
The standard deviation of the truncated_normal_initializer for initializing all weight matrices.

layer_norm_eps (*float*, *optional*, defaults to 1e-12):
The epsilon used by the layer normalization layers.

classifier_dropout_prob (*float*, *optional*, defaults to 0.1):
The dropout ratio for attached classifiers.

position_embedding_type (*str*, *optional*, defaults to *"absolute"*):
Type of position embedding. Choose one of *"absolute"*, *"relative_key"*, *"relative_key_query"*. For positional embeddings use *"absolute"*. For more information on *"relative_key"*, please refer to Self-Attention with Relative Position Representations (Shaw et al.). For more information on *"relative_key_query"*, please refer to *Method 4* in Improve Transformer Models with Better Relative Position Embeddings (Huang et al.).

pad_token_id (*int*, *optional*, defaults to 0):
Padding token id.

bos_token_id (*int*, *optional*, defaults to 2):
Beginning of stream token id.

eos_token_id (*int*, *optional*, defaults to 3):
End of stream token id.

`class` transformers.models.bart.configuration_bart.**BartConfig**(*vocab_size=50265*, *max_position_embeddings=1024*, *encoder_layers=12*, *encoder_ffn_dim=4096*, *encoder_attention_heads=16*, *decoder_layers=12*, *decoder_ffn_dim=4096*, *decoder_attention_heads=16*, *encoder_layerdrop=0.0*, *decoder_layerdrop=0.0*, *activation_function='gelu'*, *d_model=1024*, *dropout=0.1*, *attention_dropout=0.0*, *activation_dropout=0.0*, *init_std=0.02*, *classifier_dropout=0.0*, *scale_embedding=False*, *use_cache=True*, *num_labels=3*, *pad_token_id=1*, *bos_token_id=0*, *eos_token_id=2*, *is_encoder_decoder=True*, *decoder_start_token_id=2*, *forced_eos_token_id=2*, ***kwargs*)

The Bart model was proposed in BART: Denoising Sequence-to-Sequence Pre-training for Natural Language Generation, Translation, and Comprehension by Mike Lewis, Yinhan Liu, Naman Goyal, Marjan Ghazvininejad, Abdelrahman Mohamed, Omer Levy, Ves Stoyanov and Luke Zettlemoyer on 29 Oct, 2019.

According to the abstract,

- Bart uses a standard seq2seq/machine translation architecture with a bidirectional encoder (like BERT) and a left-to-right decoder (like GPT).
- The pretraining task involves randomly shuffling the order of the original sentences and a novel in-filling scheme, where spans of text are replaced with a single mask token.
- BART is particularly effective when fine tuned for text generation but also works well for comprehension tasks. It matches the performance of RoBERTa with comparable training resources on GLUE and SQuAD, achieves new state-of-the-art results on a range of abstractive dialogue, question answering, and summarization tasks, with gains of up to 6 ROUGE.

Tips:

- BART is a model with absolute position embeddings so it's usually advised to pad the inputs on the right rather than the left.
- Sequence-to-sequence model with an encoder and a decoder. Encoder is fed a corrupted version of the tokens, decoder is fed the original tokens (but has a mask to hide the future words like a regular transformers decoder). A composition of the following transformations are applied on the pretraining tasks for the encoder:
  - mask random tokens (like in BERT)
  - delete random tokens
  - mask a span of k tokens with a single mask token (a span of 0 tokens is an insertion of a mask token)
  - permute sentences
  - rotate the document to make it start at a specific token

This model was contributed by sshleifer. The Authors' code can be found here.

#Args:

vocab_size (*int*, *optional*, defaults to 50265):
Vocabulary size of the BART model. Defines the number of different tokens that can be represented by the *inputs_ids* passed when calling `BartModel` or `TFBartModel`.

d_model (*int*, *optional*, defaults to 1024):
Dimensionality of the layers and the pooler layer.

encoder_layers (*int*, *optional*, defaults to 12):
Number of encoder layers.

decoder_layers (*int*, *optional*, defaults to 12):
Number of decoder layers.

encoder_attention_heads (*int*, *optional*, defaults to 16):
Number of attention heads for each attention layer in the Transformer encoder.

decoder_attention_heads (*int*, *optional*, defaults to 16):
Number of attention heads for each attention layer in the Transformer decoder.

decoder_ffn_dim (*int*, *optional*, defaults to 4096):
Dimensionality of the "intermediate" (often named feed-forward) layer in decoder.

encoder_ffn_dim (*int*, *optional*, defaults to 4096):
Dimensionality of the "intermediate" (often named feed-forward) layer in decoder.

activation_function (*str* or *function*, *optional*, defaults to *"gelu"*):
The non-linear activation function (function or string) in the encoder and pooler. If string, *"gelu"*, *"relu"*, *"silu"* and *"gelu_new"* are supported.

dropout (*float*, *optional*, defaults to 0.1):
The dropout probability for all fully connected layers in the embeddings, encoder, and pooler.

attention_dropout (*float*, *optional*, defaults to 0.0):
The dropout ratio for the attention probabilities.

activation_dropout (*float*, *optional*, defaults to 0.0):
The dropout ratio for activations inside the fully connected layer.

classifier_dropout (*float*, *optional*, defaults to 0.0):
The dropout ratio for classifier.

max_position_embeddings (*int*, *optional*, defaults to 1024):
The maximum sequence length that this model might ever be used with. Typically set this to something large just in case (e.g., 512 or 1024 or 2048).

init_std (*float*, *optional*, defaults to 0.02):
The standard deviation of the truncated_normal_initializer for initializing all weight matrices.

encoder_layerdrop (*float*, *optional*, defaults to 0.0):
The LayerDrop probability for the encoder. See the LayerDrop paper for more details.

decoder_layerdrop (*float*, *optional*, defaults to 0.0):
The LayerDrop probability for the decoder. See the LayerDrop paper for more details.

scale_embedding (*bool*, *optional*, defaults to *False*):
Scale embeddings by diving by sqrt(d_model).

use_cache (*bool*, *optional*, defaults to *True*):
Whether or not the model should return the last key/values attentions (not used by all models).

num_labels (*int*, *optional*, defaults to 3):
The number of labels to use in `BartForSequenceClassification`.

forced_eos_token_id (*int*, *optional*, defaults to 2):
The id of the token to force as the last generated token when *max_length* is reached. Usually set to *eos_token_id*.

`class` transformers.models.bert.configuration_bert.**BertConfig**(*vocab_size=30522*, *hidden_size=768*, *num_hidden_layers=12*, *num_attention_heads=12*, *intermediate_size=3072*, *hidden_act='gelu'*, *hidden_dropout_prob=0.1*, *attention_probs_dropout_prob=0.1*, *max_position_embeddings=512*, *type_vocab_size=2*, *initializer_range=0.02*, *layer_norm_eps=1e-12*, *pad_token_id=0*, *position_embedding_type='absolute'*, *use_cache=True*, *classifier_dropout=None*, ***kwargs*)

The BERT model was proposed in BERT: Pre-training of Deep Bidirectional Transformers for Language Understanding by Jacob Devlin, Ming-Wei Chang, Kenton Lee and Kristina Toutanova. It's a bidirectional transformer pretrained using a combination of masked language modeling objective and next sentence prediction on a large corpus comprising the Toronto Book Corpus and Wikipedia.

The abstract from the paper is the following:

We introduce a new language representation model called BERT, which stands for Bidirectional Encoder Representations from Transformers. Unlike recent language representation models, BERT is designed to pre-train deep bidirectional representations from unlabeled text by jointly conditioning on both left and right context in all layers. As a result, the pre-trained BERT model can be fine-tuned with just one additional output layer to create state-of-the-art models for a wide range of tasks, such as question answering and language inference, without substantial task-specific architecture modifications.

BERT is conceptually simple and empirically powerful. It obtains new state-of-the-art results on eleven natural language processing tasks, including pushing the GLUE score to 80.5% (7.7% point absolute improvement), MultiNLI accuracy to 86.7% (4.6% absolute improvement), SQuAD v1.1 question answering Test F1 to 93.2 (1.5 point absolute improvement) and SQuAD v2.0 Test F1 to 83.1 (5.1 point absolute improvement).

Tips:

- BERT is a model with absolute position embeddings so it's usually advised to pad the inputs on the right rather than the left.
- BERT was trained with the masked language modeling (MLM) and next sentence prediction (NSP) objectives. It is efficient at predicting masked tokens and at NLU in general, but is not optimal for text generation.
- Corrupts the inputs by using random masking, more precisely, during pretraining, a given percentage of tokens (usually 15%) is masked by:
  - a special mask token with probability 0.8
  - a random token different from the one masked with probability 0.1
  - the same token with probability 0.1
- The model must predict the original sentence, but has a second objective: inputs are two sentences A and B (with a separation token in between). With probability 50%, the sentences are consecutive in the corpus, in the remaining 50% they are not related. The model has to predict if the sentences are consecutive or not.

This model was contributed by thomwolf. The original code can be found here.

Args:

vocab_size (*int*, *optional*, defaults to 30522):
Vocabulary size of the BERT model. Defines the number of different tokens that can be represented by the *inputs_ids* passed when calling `BertModel` or `TFBertModel`.

hidden_size (*int*, *optional*, defaults to 768):
Dimensionality of the encoder layers and the pooler layer.

num_hidden_layers (*int*, *optional*, defaults to 12):
Number of hidden layers in the Transformer encoder.

num_attention_heads (*int*, *optional*, defaults to 12):
Number of attention heads for each attention layer in the Transformer encoder.

intermediate_size (*int*, *optional*, defaults to 3072):
Dimensionality of the "intermediate" (often named feed-forward) layer in the Transformer encoder.

hidden_act (*str* or *Callable*, *optional*, defaults to *"gelu"*):
The non-linear activation function (function or string) in the encoder and pooler. If string, *"gelu"*, *"relu"*, *"silu"* and *"gelu_new"* are supported.

hidden_dropout_prob (*float*, *optional*, defaults to 0.1):
The dropout probability for all fully connected layers in the embeddings, encoder, and pooler.

attention_probs_dropout_prob (*float*, *optional*, defaults to 0.1):
The dropout ratio for the attention probabilities.

max_position_embeddings (*int*, *optional*, defaults to 512):
The maximum sequence length that this model might ever be used with. Typically set this to something large just in case (e.g., 512 or 1024 or 2048).

type_vocab_size (*int*, *optional*, defaults to 2):
The vocabulary size of the *token_type_ids* passed when calling `BertModel` or `TFBertModel`.

initializer_range (*float*, *optional*, defaults to 0.02):
The standard deviation of the truncated_normal_initializer for initializing all weight matrices.

layer_norm_eps (*float*, *optional*, defaults to 1e-12):
The epsilon used by the layer normalization layers.

position_embedding_type (*str*, *optional*, defaults to *"absolute"*):
Type of position embedding. Choose one of *"absolute"*, *"relative_key"*, *"relative_key_query"*. For positional embeddings use *"absolute"*. For more information on *"relative_key"*, please refer to Self-Attention with Relative Position Representations (Shaw et al.). For more information on *"relative_key_query"*, please refer to *Method 4* in Improve Transformer Models with Better Relative Position Embeddings (Huang et al.).

is_decoder (*bool*, *optional*, defaults to *False*):
Whether the model is used as a decoder or not. If *False*, the model is used as an encoder.

use_cache (*bool*, *optional*, defaults to *True*):
Whether or not the model should return the last key/values attentions (not used by all models). Only relevant if *config.is_decoder=True*.

classifier_dropout (*float*, *optional*):
The dropout ratio for the classification head.

class transformers.models.bert_generation.configuration_bert_generation.**BertGenerationConfig**(*vocab_size=5 hidden_size=102 num_hidden_ num_attentio intermediate_size=409 hidden_act='gel hidden_dropout_ attention_probs_d max_position initializer_range=0 layer_norm_e 12, pad_token_id bos_token_id eos_token_id= position_embeddi use_cache=T **kwargs*)

The BertGeneration model is a BERT model that can be leveraged for sequence-to-sequence tasks using `EncoderDecoderModel` as proposed in Leveraging Pre-trained Checkpoints for Sequence Generation Tasks by Sascha Rothe, Shashi Narayan, Aliaksei Severyn.

The abstract from the paper is the following:

Unsupervised pretraining of large neural models has recently revolutionized Natural Language Processing. By warm-starting from the publicly released checkpoints, NLP practitioners have pushed the state-of-the-art on multiple benchmarks while saving significant amounts of compute time. So far the focus has been mainly on the Natural Language Understanding tasks. In this paper, we demonstrate the efficacy of pre-trained checkpoints for Sequence Generation. We developed a Transformer-based sequence-to-sequence model that is compatible with publicly available pre-trained BERT, GPT-2 and RoBERTa checkpoints and conducted an extensive empirical study on the utility of initializing our model, both encoder and decoder, with these checkpoints. Our models result in new state-of-the-art results on Machine Translation, Text Summarization, Sentence Splitting, and Sentence Fusion.

class transformers.models.big_bird.configuration_big_bird.**BigBirdConfig**(*vocab_size=50358*, *hidden_size=768*, *num_hidden_layers=12*, *num_attention_heads=12*, *intermediate_size=3072*, *hidden_act='gelu_new'*, *hidden_dropout_prob=0.1*, *attention_probs_dropout_prob=0.1*, *max_position_embeddings=4096*, *type_vocab_size=2*, *initializer_range=0.02*, *layer_norm_eps=1e-12*, *use_cache=True*, *pad_token_id=0*, *bos_token_id=1*, *eos_token_id=2*, *sep_token_id=66*, *attention_type='block_sparse'*, *use_bias=True*, *rescale_embeddings=False*, *block_size=64*, *num_random_blocks=3*, *classifier_dropout=None*, ***kwargs*)

The BigBird model was proposed in Big Bird: Transformers for Longer Sequences by Zaheer, Manzil and Guruganesh, Guru and Dubey, Kumar Avinava and Ainslie, Joshua and Alberti, Chris and Ontanon, Santiago and Pham, Philip and Ravula, Anirudh and Wang, Qifan and Yang, Li and others. BigBird, is a sparse-attention based transformer which extends Transformer based models, such as BERT to much longer sequences. In addition to sparse attention, BigBird also applies global attention as well as random attention to the input sequence. Theoretically, it has been shown that applying sparse, global, and random attention approximates full attention, while being computationally much more efficient for longer sequences. As a consequence of the capability to handle longer context, BigBird has shown improved performance on various long document NLP tasks, such as question answering and summarization, compared to BERT or RoBERTa.

The abstract from the paper is the following:

Transformers-based models, such as BERT, have been one of the most successful deep learning models for NLP. Unfortunately, one of their core limitations is the quadratic dependency (mainly in terms of memory) on the sequence length due to their full attention mechanism. To remedy this, we propose, BigBird, a sparse attention mechanism that reduces this quadratic dependency to linear. We show that BigBird is a universal approximator of sequence functions and is Turing complete, thereby preserving these properties of the quadratic, full attention model. Along the way, our theoretical analysis reveals some of the benefits of having O(1) global tokens (such as CLS), that attend to the entire sequence as part of the sparse attention mechanism. The proposed sparse attention can handle sequences of length up to 8x of what was previously possible using similar hardware. As a consequence of the capability to handle longer context, BigBird drastically improves performance on various NLP tasks such as question answering and summarization. We also propose novel applications to genomics data.

Tips:

- For an in-detail explanation on how BigBird's attention works, see this blog post.
- BigBird comes with 2 implementations: **original_full** & **block_sparse**. For the sequence length < 1024, using **original_full** is advised as there is no benefit in using **block_sparse** attention.
- The code currently uses window size of 3 blocks and 2 global blocks.
- Sequence length must be divisible by block size.
- Current implementation supports only **ITC**.
- Current implementation doesn't support **num_random_blocks = 0**
- BigBird is a model with absolute position embeddings so it's usually advised to pad the inputs on the right rather than the left.

This model was contributed by vasudevgupta. The original code can be found here.

Args:

vocab_size (*int*, *optional*, defaults to 50358):
Vocabulary size of the BigBird model. Defines the number of different tokens that can be represented by the *inputs_ids* passed when calling `BigBirdModel`.

hidden_size (*int*, *optional*, defaults to 768):
Dimension of the encoder layers and the pooler layer.

num_hidden_layers (*int*, *optional*, defaults to 12):
Number of hidden layers in the Transformer encoder.

num_attention_heads (*int*, *optional*, defaults to 12):
Number of attention heads for each attention layer in the Transformer encoder.

intermediate_size (*int*, *optional*, defaults to 3072):
Dimension of the "intermediate" (i.e., feed-forward) layer in the Transformer encoder.

hidden_act (*str* or *function*, *optional*, defaults to *"gelu_new"*):
The non-linear activation function (function or string) in the encoder and pooler. If string, *"gelu"*, *"relu"*, *"selu"* and *"gelu_new"* are supported.

hidden_dropout_prob (*float*, *optional*, defaults to 0.1):
The dropout probability for all fully connected layers in the embeddings, encoder, and pooler.

attention_probs_dropout_prob (*float*, *optional*, defaults to 0.1):
The dropout ratio for the attention probabilities.

max_position_embeddings (*int*, *optional*, defaults to 4096):
The maximum sequence length that this model might ever be used with. Typically set this to something large just in case (e.g., 1024 or 2048 or 4096).

type_vocab_size (*int*, *optional*, defaults to 2):
The vocabulary size of the *token_type_ids* passed when calling `BigBirdModel`.

initializer_range (*float*, *optional*, defaults to 0.02):
The standard deviation of the truncated_normal_initializer for initializing all weight matrices.

layer_norm_eps (*float*, *optional*, defaults to 1e-12):
The epsilon used by the layer normalization layers.

is_decoder (*bool*, *optional*, defaults to *False*):
Whether the model is used as a decoder or not. If *False*, the model is used as an encoder.

use_cache (*bool*, *optional*, defaults to *True*):
Whether or not the model should return the last key/values attentions (not used by all models). Only relevant if *config.is_decoder=True*.

attention_type (*str*, *optional*, defaults to *"block_sparse"*)
Whether to use block sparse attention (with n complexity) as introduced in paper or original attention layer (with n^2 complexity). Possible values are *"original_full"* and *"block_sparse"*.

use_bias (*bool*, *optional*, defaults to *True*)
Whether to use bias in query, key, value.

rescale_embeddings (*bool*, *optional*, defaults to *False*)
Whether to rescale embeddings with (hidden_size ** 0.5).

block_size (*int*, *optional*, defaults to 64)
Size of each block. Useful only when *attention_type* == *"block_sparse"*.

num_random_blocks (*int*, *optional*, defaults to 3)
Each query is going to attend these many number of random blocks. Useful only when *attention_type* == *"block_sparse"*.

classifier_dropout (*float*, *optional*):
The dropout ratio for the classification head.

class transformers.models.bigbird_pegasus.configuration_bigbird_pegasus.**BigBirdPegasusConfig**(*vocab_size=9*
max_position
en-
coder_layers=
en-
coder_ffn_dir
en-
coder_attenti
de-
coder_layers=
de-
coder_ffn_dir
de-
coder_attenti
en-
coder_layerd
de-
coder_layerd
use_cache=T
is_encoder_d
ac-
ti-
va-
tion_function
d_model=102
dropout=0.1,
at-
ten-
tion_dropout=
ac-
ti-
va-
tion_dropout=
init_std=0.02
de-
coder_start_t
clas-
si-
fier_dropout=
scale_embedd
pad_token_id
bos_token_id
eos_token_id=
at-
ten-
tion_type='bl
block_size=6
*num_random_*
use_bias=Fal
***kwargs*)

The BigBird model was proposed in Big Bird: Transformers for Longer Sequences by Zaheer, Manzil and Guruganesh, Guru and Dubey, Kumar Avinava and Ainslie, Joshua and Alberti, Chris and Ontanon, Santiago and Pham, Philip and Ravula, Anirudh and Wang, Qifan and Yang, Li and others. BigBird, is a sparse-attention based transformer which

extends Transformer based models, such as BERT to much longer sequences. In addition to sparse attention, BigBird also applies global attention as well as random attention to the input sequence. Theoretically, it has been shown that applying sparse, global, and random attention approximates full attention, while being computationally much more efficient for longer sequences. As a consequence of the capability to handle longer context, BigBird has shown improved performance on various long document NLP tasks, such as question answering and summarization, compared to BERT or RoBERTa.

The abstract from the paper is the following:

Transformers-based models, such as BERT, have been one of the most successful deep learning models for NLP. Unfortunately, one of their core limitations is the quadratic dependency (mainly in terms of memory) on the sequence length due to their full attention mechanism. To remedy this, we propose, BigBird, a sparse attention mechanism that reduces this quadratic dependency to linear. We show that BigBird is a universal approximator of sequence functions and is Turing complete, thereby preserving these properties of the quadratic, full attention model. Along the way, our theoretical analysis reveals some of the benefits of having O(1) global tokens (such as CLS), that attend to the entire sequence as part of the sparse attention mechanism. The proposed sparse attention can handle sequences of length up to 8x of what was previously possible using similar hardware. As a consequence of the capability to handle longer context, BigBird drastically improves performance on various NLP tasks such as question answering and summarization. We also propose novel applications to genomics data.

Tips:

- For an in-detail explanation on how BigBird's attention works, see this blog post.
- BigBird comes with 2 implementations: **original_full** & **block_sparse**. For the sequence length < 1024, using **original_full** is advised as there is no benefit in using **block_sparse** attention.
- The code currently uses window size of 3 blocks and 2 global blocks.
- Sequence length must be divisible by block size.
- Current implementation supports only **ITC**.
- Current implementation doesn't support **num_random_blocks = 0**.
- BigBirdPegasus uses the PegasusTokenizer.
- BigBird is a model with absolute position embeddings so it's usually advised to pad the inputs on the right rather than the left.

The original code can be found here.

Args:

vocab_size (*int*, *optional*, defaults to 96103):
Vocabulary size of the BigBirdPegasus model. Defines the number of different tokens that can be represented by the *inputs_ids* passed when calling `BigBirdPegasusModel`.

d_model (*int*, *optional*, defaults to 1024):
Dimension of the layers and the pooler layer.

encoder_layers (*int*, *optional*, defaults to 16):
Number of encoder layers.

decoder_layers (*int*, *optional*, defaults to 16):
Number of decoder layers.

encoder_attention_heads (*int*, *optional*, defaults to 16):
Number of attention heads for each attention layer in the Transformer encoder.

decoder_attention_heads (*int*, *optional*, defaults to 16):
Number of attention heads for each attention layer in the Transformer decoder.

decoder_ffn_dim (*int*, *optional*, defaults to 4096):
Dimension of the "intermediate" (often named feed-forward) layer in decoder.

encoder_ffn_dim (*int*, *optional*, defaults to 4096):
Dimension of the "intermediate" (often named feed-forward) layer in decoder.

activation_function (*str* or *function*, *optional*, defaults to *"gelu_new"*):
The non-linear activation function (function or string) in the encoder and pooler. If string, *"gelu"*, *"relu"*, *"silu"* and *"gelu_new"* are supported.

dropout (*float*, *optional*, defaults to 0.1):
The dropout probability for all fully connected layers in the embeddings, encoder, and pooler.

attention_dropout (*float*, *optional*, defaults to 0.0):
The dropout ratio for the attention probabilities.

activation_dropout (*float*, *optional*, defaults to 0.0):
The dropout ratio for activations inside the fully connected layer.

classifier_dropout (*float*, *optional*, defaults to 0.0):
The dropout ratio for classifier.

max_position_embeddings (*int*, *optional*, defaults to 4096):
The maximum sequence length that this model might ever be used with. Typically set this to something large just in case (e.g., 1024 or 2048 or 4096).

init_std (*float*, *optional*, defaults to 0.02):
The standard deviation of the truncated_normal_initializer for initializing all weight matrices.

encoder_layerdrop (*float*, *optional*, defaults to 0.0):
The LayerDrop probability for the encoder. See the LayerDrop paper for more details.

decoder_layerdrop (*float*, *optional*, defaults to 0.0):
The LayerDrop probability for the decoder. See the LayerDrop paper for more details.

use_cache (*bool*, *optional*, defaults to *True*):
Whether or not the model should return the last key/values attentions (not used by all models).

attention_type (*str*, *optional*, defaults to *"block_sparse"*)
Whether to use block sparse attention (with n complexity) as introduced in paper or original attention layer (with n^2 complexity) in encoder. Possible values are *"original_full"* and *"block_sparse"*.

use_bias (*bool*, *optional*, defaults to *False*)
Whether to use bias in query, key, value.

block_size (*int*, *optional*, defaults to 64)
Size of each block. Useful only when *attention_type* == *"block_sparse"*.

num_random_blocks (*int*, *optional*, defaults to 3)
Each query is going to attend these many number of random blocks. Useful only when *attention_type* == *"block_sparse"*.

scale_embeddings (*bool*, *optional*, defaults to *True*)
Whether to rescale embeddings with (hidden_size ** 0.5).

class transformers.models.biogpt.configuration_biogpt.**BioGptConfig**(*vocab_size=42384*, *hidden_size=1024*, *num_hidden_layers=24*, *num_attention_heads=16*, *intermediate_size=4096*, *hidden_act='gelu'*, *hidden_dropout_prob=0.1*, *attention_probs_dropout_prob=0.1*, *max_position_embeddings=1024*, *initializer_range=0.02*, *layer_norm_eps=1e-12*, *scale_embedding=True*, *use_cache=True*, *layerdrop=0.0*, *activation_dropout=0.0*, *pad_token_id=1*, *bos_token_id=0*, *eos_token_id=2*, ***kwargs*)

The BioGPT model was proposed in `BioGPT: generative pre-trained transformer for biomedical text generation and mining
<https://academic.oup.com/bib/advance-article/doi/10.1093/bib/bbac409/6713511?guestAccessKey=a66d9b5d-4f83-4017-bb52-405815c907b9>`__ by Renqian Luo, Liai Sun, Yingce Xia, Tao Qin, Sheng Zhang, Hoifung Poon and Tie-Yan Liu. BioGPT is a domain-specific generative pre-trained Transformer language model for biomedical text generation and mining. BioGPT follows the Transformer language model backbone, and is pre-trained on 15M PubMed abstracts from scratch.

The abstract from the paper is the following:

Pre-trained language models have attracted increasing attention in the biomedical domain, inspired by their great success in the general natural language domain. Among the two main branches of pre-trained language models in the general language domain, i.e. BERT (and its variants) and GPT (and its variants), the first one has been extensively studied in the biomedical domain, such as BioBERT and PubMedBERT. While they have achieved great success on a variety of discriminative downstream biomedical tasks, the lack of generation ability constrains their application scope. In this paper, we propose BioGPT, a domain-specific generative Transformer language model pre-trained on large-scale biomedical literature. We evaluate BioGPT on six biomedical natural language processing tasks and demonstrate that our model outperforms previous models on most tasks. Especially, we get 44.98%, 38.42% and 40.76% F1 score on BC5CDR, KD-DTI and DDI end-to-end relation extraction tasks, respectively, and 78.2% accuracy on PubMedQA, creating a new record. Our case study on text generation further demonstrates the advantage of BioGPT on biomedical literature to generate fluent descriptions for biomedical terms.

Tips:

- BioGPT is a model with absolute position embeddings so it's usually advised to pad the inputs on the right rather than the left.
- BioGPT was trained with a causal language modeling (CLM) objective and is therefore powerful at predicting the next token in a sequence. Leveraging this feature allows BioGPT to generate syntactically coherent text as it can be observed in the run_generation.py example script.
- The model can take the *past_key_values* (for PyTorch) as input, which is the previously computed key/value attention pairs. Using this (past_key_values or past) value prevents the model from re-computing pre-computed values in the context of text generation. For PyTorch, see past_key_values argument of the BioGptForCausalLM.forward() method for more information on its usage.

This model was contributed by kamalkraj. The original code can be found here.

Args:

vocab_size (*int*, *optional*, defaults to 42384):
Vocabulary size of the BioGPT model. Defines the number of different tokens that can be represented by the *inputs_ids* passed when calling `BioGptModel`.

hidden_size (*int*, *optional*, defaults to 1024):
Dimension of the encoder layers and the pooler layer.

num_hidden_layers (*int*, *optional*, defaults to 24):
Number of hidden layers in the Transformer encoder.

num_attention_heads (*int*, *optional*, defaults to 16):
Number of attention heads for each attention layer in the Transformer encoder.

intermediate_size (*int*, *optional*, defaults to 4096):
Dimension of the "intermediate" (i.e., feed-forward) layer in the Transformer encoder.

hidden_act (*str* or *function*, *optional*, defaults to *"gelu"*):
The non-linear activation function (function or string) in the encoder and pooler. If string, *"gelu"*, *"relu"*, *"selu"* and *"gelu_new"* are supported.

hidden_dropout_prob (*float*, *optional*, defaults to 0.1):
The dropout probability for all fully connected layers in the embeddings, encoder, and pooler.

attention_probs_dropout_prob (*float*, *optional*, defaults to 0.1):
The dropout ratio for the attention probabilities.

max_position_embeddings (*int*, *optional*, defaults to 1024):
The maximum sequence length that this model might ever be used with. Typically set this to something large just in case (e.g., 512 or 1024 or 2048).

initializer_range (*float*, *optional*, defaults to 0.02):
The standard deviation of the truncated_normal_initializer for initializing all weight matrices.

layer_norm_eps (*float*, *optional*, defaults to 1e-12):
The epsilon used by the layer normalization layers.

scale_embedding (*bool*, *optional*, defaults to *True*):
Scale embeddings by diving by sqrt(d_model).

use_cache (*bool*, *optional*, defaults to *True*):
Whether or not the model should return the last key/values attentions (not used by all models). Only relevant if *config.is_decoder=True*.

layerdrop (*float*, *optional*, defaults to 0.0):
Please refer to the paper about LayerDrop: https://arxiv.org/abs/1909.11556 for further details

activation_dropout (*float*, *optional*, defaults to 0.0):
The dropout ratio for activations inside the fully connected layer.

pad_token_id (*int*, *optional*, defaults to 1):
Padding token id.

bos_token_id (*int*, *optional*, defaults to 0):
Beginning of stream token id.

eos_token_id (*int*, *optional*, defaults to 2):
End of stream token id.

class transformers.models.blenderbot.configuration_blenderbot.**BlenderbotConfig**(*vocab_size=8008*, *max_position_embeddings=128* *encoder_layers=2*, *encoder_ffn_dim=10240*, *encoder_attention_heads=32*, *decoder_layers=24*, *decoder_ffn_dim=10240*, *decoder_attention_heads=32*, *encoder_layerdrop=0.0*, *decoder_layerdrop=0.0*, *use_cache=True*, *is_encoder_decoder=True*, *activation_function='gelu'*, *d_model=2560*, *dropout=0.1*, *attention_dropout=0.0*, *activation_dropout=0.0*, *init_std=0.02*, *decoder_start_token_id=1*, *scale_embedding=False*, *pad_token_id=0*, *bos_token_id=1*, *eos_token_id=2*, *encoder_no_repeat_ngram_size=* *forced_eos_token_id=2*, ***kwargs*)

The Blender chatbot model was proposed in Recipes for building an open-domain chatbot Stephen Roller, Emily Dinan, Naman Goyal, Da Ju, Mary Williamson, Yinhan Liu, Jing Xu, Myle Ott, Kurt Shuster, Eric M. Smith, Y-Lan Boureau, Jason Weston on 30 Apr 2020.

The abstract of the paper is the following:

Building open-domain chatbots is a challenging area for machine learning research. While prior work has shown that scaling neural models in the number of parameters and the size of the data they are trained on gives improved results, we show that other ingredients are important for a high-performing chatbot. Good conversation requires a number of skills that an expert conversationalist blends in a seamless way: providing engaging talking points and listening to their partners, and displaying knowledge, empathy and personality appropriately, while maintaining a consistent persona. We show that large scale models can learn these skills when given appropriate training data and choice of generation strategy. We build variants of these recipes with 90M, 2.7B and 9.4B parameter models, and make our models and code publicly available. Human evaluations show our best models are superior to existing approaches in multi-turn dialogue in terms of engagingness and humanness measurements. We then discuss the limitations of this

work by analyzing failure cases of our models.

Tips:

- Blenderbot is a model with absolute position embeddings so it's usually advised to pad the inputs on the right rather than the left.

This model was contributed by sshleifer. The authors' code can be found here .

Args:

vocab_size (*int*, *optional*, defaults to 50265):
Vocabulary size of the Blenderbot model. Defines the number of different tokens that can be represented by the *inputs_ids* passed when calling `BlenderbotModel` or `TFBlenderbotModel`.

d_model (*int*, *optional*, defaults to 1024):
Dimensionality of the layers and the pooler layer.

encoder_layers (*int*, *optional*, defaults to 12):
Number of encoder layers.

decoder_layers (*int*, *optional*, defaults to 12):
Number of decoder layers.

encoder_attention_heads (*int*, *optional*, defaults to 16):
Number of attention heads for each attention layer in the Transformer encoder.

decoder_attention_heads (*int*, *optional*, defaults to 16):
Number of attention heads for each attention layer in the Transformer decoder.

decoder_ffn_dim (*int*, *optional*, defaults to 4096):
Dimensionality of the "intermediate" (often named feed-forward) layer in decoder.

encoder_ffn_dim (*int*, *optional*, defaults to 4096):
Dimensionality of the "intermediate" (often named feed-forward) layer in decoder.

activation_function (*str* or *function*, *optional*, defaults to *"gelu"*):
The non-linear activation function (function or string) in the encoder and pooler. If string, *"gelu"*, *"relu"*, *"silu"* and *"gelu_new"* are supported.

dropout (*float*, *optional*, defaults to 0.1):
The dropout probability for all fully connected layers in the embeddings, encoder, and pooler.

attention_dropout (*float*, *optional*, defaults to 0.0):
The dropout ratio for the attention probabilities.

activation_dropout (*float*, *optional*, defaults to 0.0):
The dropout ratio for activations inside the fully connected layer.

max_position_embeddings (*int*, *optional*, defaults to 128):
The maximum sequence length that this model might ever be used with. Typically set this to something large just in case (e.g., 512 or 1024 or 2048).

init_std (*float*, *optional*, defaults to 0.02):
The standard deviation of the truncated_normal_initializer for initializing all weight matrices.

encoder_layerdrop (*float*, *optional*, defaults to 0.0):
The LayerDrop probability for the encoder. See the LayerDrop paper for more details.

decoder_layerdrop (*float*, *optional*, defaults to 0.0):
The LayerDrop probability for the decoder. See the LayerDrop paper for more details.

scale_embedding (*bool*, *optional*, defaults to *False*):
Scale embeddings by diving by sqrt(d_model).

use_cache (*bool*, *optional*, defaults to *True*):
Whether or not the model should return the last key/values attentions (not used by all models)

forced_eos_token_id (*int*, *optional*, defaults to 2):
The id of the token to force as the last generated token when *max_length* is reached. Usually set to *eos_token_id*.

class transformers.models.blenderbot_small.configuration_blenderbot_small.**BlenderbotSmallConfig**(*vocab_siz*
max_posi
en-
coder_lay
en-
coder_ffn
en-
coder_att
de-
coder_lay
de-
coder_ffn
de-
coder_att
en-
coder_lay
de-
coder_lay
use_cach
is_encode
ac-
ti-
va-
tion_func
d_model=
dropout=
at-
ten-
tion_drop
ac-
ti-
va-
tion_drop
init_std=
de-
coder_sta
scale_em
pad_toke
bos_toke
eos_toke
forced_eo
***kwargs*

The Blender chatbot model was proposed in Recipes for building an open-domain chatbot Stephen Roller, Emily Dinan, Naman Goyal, Da Ju, Mary Williamson, Yinhan Liu, Jing Xu, Myle Ott, Kurt Shuster, Eric M. Smith, Y-Lan Boureau, Jason Weston on 30 Apr 2020.

The abstract of the paper is the following:

Building open-domain chatbots is a challenging area for machine learning research. While prior work has shown that scaling neural models in the number of parameters and the size of the data they are trained on gives improved results, we show that other ingredients are important for a high-performing chatbot. Good conversation requires a number of skills that an expert conversationalist blends in a seamless way: providing engaging talking points and listening to their partners, and displaying knowledge, empathy and personality appropriately, while maintaining a consistent persona. We show that large scale models can learn these skills when given appropriate training data and choice of generation strategy. We build variants of these recipes with 90M, 2.7B and 9.4B parameter models, and make our models and code publicly available. Human evaluations show our best models are superior to existing approaches in multi-turn dialogue in terms of engagingness and humanness measurements. We then discuss the limitations of this work by analyzing failure cases of our models.

Tips:

- Blenderbot Small is a model with absolute position embeddings so it's usually advised to pad the inputs on the right rather than the left.

This model was contributed by patrickvonplaten. The authors' code can be found here.

Args:

vocab_size (*int*, *optional*, defaults to 50265):
Vocabulary size of the BlenderbotSmall model. Defines the number of different tokens that can be represented by the *inputs_ids* passed when calling `BlenderbotSmallModel` or `TFBlenderbotSmallModel`.

d_model (*int*, *optional*, defaults to 512):
Dimensionality of the layers and the pooler layer.

encoder_layers (*int*, *optional*, defaults to 8):
Number of encoder layers.

decoder_layers (*int*, *optional*, defaults to 8):
Number of decoder layers.

encoder_attention_heads (*int*, *optional*, defaults to 16):
Number of attention heads for each attention layer in the Transformer encoder.

decoder_attention_heads (*int*, *optional*, defaults to 16):
Number of attention heads for each attention layer in the Transformer decoder.

decoder_ffn_dim (*int*, *optional*, defaults to 2048):
Dimensionality of the "intermediate" (often named feed-forward) layer in decoder.

encoder_ffn_dim (*int*, *optional*, defaults to 2048):
Dimensionality of the "intermediate" (often named feed-forward) layer in decoder.

activation_function (*str* or *function*, *optional*, defaults to *"gelu"*):
The non-linear activation function (function or string) in the encoder and pooler. If string, *"gelu"*, *"relu"*, *"silu"* and *"gelu_new"* are supported.

dropout (*float*, *optional*, defaults to 0.1):
The dropout probability for all fully connected layers in the embeddings, encoder, and pooler.

attention_dropout (*float*, *optional*, defaults to 0.0):
The dropout ratio for the attention probabilities.

activation_dropout (*float*, *optional*, defaults to 0.0):
The dropout ratio for activations inside the fully connected layer.

max_position_embeddings (*int*, *optional*, defaults to 512):
The maximum sequence length that this model might ever be used with. Typically set this to something large just in case (e.g., 512 or 1024 or 2048).

init_std (*float*, *optional*, defaults to 0.02):
The standard deviation of the truncated_normal_initializer for initializing all weight matrices.

encoder_layerdrop (*float*, *optional*, defaults to 0.0):
The LayerDrop probability for the encoder. See the LayerDrop paper for more details.

decoder_layerdrop (*float*, *optional*, defaults to 0.0):
The LayerDrop probability for the decoder. See the LayerDrop paper for more details.

scale_embedding (*bool*, *optional*, defaults to *False*):
Scale embeddings by diving by sqrt(d_model).

use_cache (*bool*, *optional*, defaults to *True*):
Whether or not the model should return the last key/values attentions (not used by all models)

forced_eos_token_id (*int*, *optional*, defaults to 2):
The id of the token to force as the last generated token when *max_length* is reached. Usually set to *eos_token_id*.

`class transformers.models.bloom.configuration_bloom.`**`BloomConfig`**(*vocab_size=250880*, *hidden_size=64*, *n_layer=2*, *n_head=8*, *layer_norm_epsilon=1e-05*, *initializer_range=0.02*, *use_cache=True*, *bos_token_id=1*, *eos_token_id=2*, *apply_residual_connection_post_layernorm=False*, *hidden_dropout=0.0*, *attention_dropout=0.0*, *pretraining_tp=1*, *slow_but_exact=False*, ***kwargs*)

The BLOOM model has been proposed with its various versions through the BigScience Workshop. BigScience is inspired by other open science initiatives where researchers have pooled their time and resources to collectively achieve a higher impact. The architecture of BLOOM is essentially similar to GPT3 (auto-regressive model for next token prediction), but has been trained on 46 different languages and 13 programming languages. Several smaller versions of the models have been trained on the same dataset. BLOOM is available in the following versions:

- bloom-560m
- bloom-1b1
- bloom-1b7
- bloom-3b
- bloom-7b1
- bloom (176B parameters)

Args:

vocab_size (*int*, *optional*, defaults to 250880):
Vocabulary size of the Bloom model. Defines the maximum number of different tokens that can be represented by the *inputs_ids* passed when calling `BloomModel`. Check this discussion on how the *vocab_size* has been defined.

hidden_size (*int*, *optional*, defaults to 64):
Dimensionality of the embeddings and hidden states.

n_layer (*int*, *optional*, defaults to 2):
Number of hidden layers in the Transformer encoder.

n_head (*int*, *optional*, defaults to 8):
Number of attention heads for each attention layer in the Transformer encoder.

layer_norm_epsilon (*float*, *optional*, defaults to 1e-5):
The epsilon to use in the layer normalization layers.

initializer_range (*float*, *optional*, defaults to 0.02):
The standard deviation of the truncated_normal_initializer for initializing all weight matrices.

apply_residual_connection_post_layernorm (*bool*, *optional*, defaults to *False*):
If enabled, use the layer norm of the hidden states as the residual in the transformer blocks

hidden_dropout (*float*, *optional*, defaults to 0.1):
Dropout rate of the dropout function on the bias dropout.

attention_dropout (*float*, *optional*, defaults to 0.1):
Dropout rate applied to the attention probs

use_cache (*bool*, *optional*, defaults to *True*):
Whether or not the model should return the last key/values attentions (not used by all models).

pretraining_tp (*int*, *optional*, defaults to *1*):
Experimental feature. Tensor parallelism rank used during pretraining with Megatron. Please refer to this document to understand more about it. This value is necessary to ensure exact reproducibility of the pretraining results. Please refer to this issue. Note also that this is enabled only when *slow_but_exact=True*.

slow_but_exact (*bool*, *optional*, defaults to *False*):
Experimental feature. Whether to use slow but exact implementation of the attention mechanism. While merging the TP rank tensors, due to slicing operations the results may be slightly different between the model trained on Megatron and our model. Please refer to this issue. A solution to obtain more accurate results is to enable this feature. Enabling this will hurt the computational time of the inference. Will be probably resolved in the future once the main model has been fine-tuned with TP_rank=1.

class transformers.models.camembert.configuration_camembert.**CamembertConfig**(*vocab_size=30522*, *hidden_size=768*, *num_hidden_layers=12*, *num_attention_heads=12*, *intermediate_size=3072*, *hidden_act='gelu'*, *hidden_dropout_prob=0.1*, *attention_probs_dropout_prob=0.1*, *max_position_embeddings=512*, *type_vocab_size=2*, *initializer_range=0.02*, *layer_norm_eps=1e-12*, *pad_token_id=1*, *bos_token_id=0*, *eos_token_id=2*, *position_embedding_type='absolute'*, *use_cache=True*, *classifier_dropout=None*, ***kwargs*)

The CamemBERT model was proposed in CamemBERT: a Tasty French Language Model by Louis Martin, Benjamin Muller, Pedro Javier Ortiz Suárez, Yoann Dupont, Laurent Romary, Éric Villemonte de la Clergerie, Djamé Seddah, and Benoît Sagot. It is based on Facebook's RoBERTa model released in 2019. It is a model trained on 138GB of French text.

The abstract from the paper is the following:

Pretrained language models are now ubiquitous in Natural Language Processing. Despite their success, most available models have either been trained on English data or on the concatenation of data in multiple languages. This makes practical use of such models –in all languages except English– very limited. Aiming to address this issue for French, we release CamemBERT, a French version of the Bi-directional Encoders for Transformers (BERT). We measure the performance of CamemBERT compared to multilingual models in multiple downstream tasks, namely part-of-speech tagging, dependency parsing, named-entity recognition, and natural language inference. CamemBERT improves the state of the art for most of the tasks considered. We release the pretrained model for CamemBERT hoping to foster research and downstream applications for French NLP.

Tips:

- This implementation is the same as RoBERTa. Refer to the documentation of RoBERTa for usage examples as well as the information relative to the inputs and outputs.

This model was contributed by camembert. The original code can be found here.

Args:

vocab_size (*int*, *optional*, defaults to 30522):
Vocabulary size of the BERT model. Defines the number of different tokens that can be represented by the *inputs_ids* passed when calling `CamembertModel` or `TFCamembertModel`.

hidden_size (*int*, *optional*, defaults to 768):
Dimensionality of the encoder layers and the pooler layer.

num_hidden_layers (*int*, *optional*, defaults to 12):
Number of hidden layers in the Transformer encoder.

num_attention_heads (*int*, *optional*, defaults to 12):
Number of attention heads for each attention layer in the Transformer encoder.

intermediate_size (*int*, *optional*, defaults to 3072):
Dimensionality of the "intermediate" (often named feed-forward) layer in the Transformer encoder.

hidden_act (*str* or *Callable*, *optional*, defaults to *"gelu"*):
The non-linear activation function (function or string) in the encoder and pooler. If string, *"gelu"*, *"relu"*, *"silu"* and *"gelu_new"* are supported.

hidden_dropout_prob (*float*, *optional*, defaults to 0.1):
The dropout probability for all fully connected layers in the embeddings, encoder, and pooler.

attention_probs_dropout_prob (*float*, *optional*, defaults to 0.1):
The dropout ratio for the attention probabilities.

max_position_embeddings (*int*, *optional*, defaults to 512):
The maximum sequence length that this model might ever be used with. Typically set this to something large just in case (e.g., 512 or 1024 or 2048).

type_vocab_size (*int*, *optional*, defaults to 2):
The vocabulary size of the *token_type_ids* passed when calling `CamembertModel` or `TFCamembertModel`.

initializer_range (*float*, *optional*, defaults to 0.02):
The standard deviation of the truncated_normal_initializer for initializing all weight matrices.

layer_norm_eps (*float*, *optional*, defaults to 1e-12):
The epsilon used by the layer normalization layers.

position_embedding_type (*str*, *optional*, defaults to *"absolute"*):
Type of position embedding. Choose one of *"absolute"*, *"relative_key"*, *"relative_key_query"*. For positional embeddings use *"absolute"*. For more information on *"relative_key"*, please refer to Self-Attention with Relative Position Representations (Shaw et al.). For more information on *"relative_key_query"*, please refer to *Method 4* in Improve Transformer Models with Better Relative Position Embeddings (Huang et al.).

is_decoder (*bool*, *optional*, defaults to *False*):
Whether the model is used as a decoder or not. If *False*, the model is used as an encoder.

use_cache (*bool*, *optional*, defaults to *True*):
Whether or not the model should return the last key/values attentions (not used by all models). Only relevant if *config.is_decoder=True*.

classifier_dropout (*float*, *optional*):
The dropout ratio for the classification head.

class transformers.models.llama.configuration_llama.**LlamaConfig**(*vocab_size=32000*, *hidden_size=4096*, *intermediate_size=11008*, *num_hidden_layers=32*, *num_attention_heads=32*, *num_key_value_heads=None*, *hidden_act='silu'*, *max_position_embeddings=2048*, *initializer_range=0.02*, *rms_norm_eps=1e-06*, *use_cache=True*, *pad_token_id=None*, *bos_token_id=1*, *eos_token_id=2*, *pretraining_tp=1*, *tie_word_embeddings=False*, *rope_theta=10000.0*, *rope_scaling=None*, *attention_bias=False*, *attention_dropout=0.0*, ***kwargs*)

The Code Llama model was proposed in Code Llama: Open Foundation Models for Code by Baptiste Rozière, Jonas Gehring, Fabian Gloeckle, Sten Sootla, Itai Gat, Xiaoqing Ellen Tan, Yossi Adi, Jingyu Liu, Tal Remez, Jérémy Rapin, Artyom Kozhevnikov, Ivan Evtimov, Joanna Bitton, Manish Bhatt, Cristian Canton Ferrer, Aaron Grattafiori, Wenhan Xiong, Alexandre Défossez, Jade Copet, Faisal Azhar, Hugo Touvron, Louis Martin, Nicolas Usunier, Thomas Scialom, Gabriel Synnaeve.

The abstract from the paper is the following:

We release Code Llama, a family of large language models for code based on Llama 2 providing state-of-the-art performance among open models, infilling capabilities, support for large input contexts, and zero-shot instruction following ability for programming tasks. We provide multiple flavors to cover a wide range of applications: foundation models (Code Llama), Python specializations (Code Llama - Python), and instruction-following models (Code Llama - Instruct) with 7B, 13B and 34B parameters each. All models are trained on sequences of 16k tokens and show improvements on inputs with up to 100k tokens. 7B and 13B Code Llama and Code Llama - Instruct variants support infilling based on surrounding content. Code Llama reaches state-of-the-art performance among open models on several code benchmarks, with scores of up to 53% and 55% on HumanEval and MBPP, respectively. Notably, Code Llama - Python 7B outperforms Llama 2 70B on HumanEval and MBPP, and all our models outperform every other publicly available model on MultiPL-E. We release Code Llama under a permissive license that allows for both research and commercial use.

Check out all Code Llama models here and the officially released ones in the codellama org.

<Tip warning={true}>

The *Llama2* family models, on which Code Llama is based, were trained using *bfloat16*, but the original inference uses *float16*. Let's look at the different precisions:

- *float32*: PyTorch convention on model initialization is to load models in *float32*, no matter with which *dtype* the model weights were stored. *transformers* also follows this convention for consistency with PyTorch. This will be picked by default. If you want the *AutoModel* API to cast the load the checkpoints with the storage weights type, you must specify *torch_dtype="auto"*, e.g. *model = AutoModelForCausalLM.from_pretrained("path", torch_dtype = "auto").*
- *bfloat16*: Code Llama was trained with this precision, so we recommend using it for further training or fine-tuning.

- *float16*: We recommend running inference using this precision, as it's usually faster than *bfloat16*, and evaluation metrics show no discernible degradation with respect to *bfloat16*. You can also run inference using *bfloat16*, and we recommend you check inference results with both *float16* and *bfloat16* after fine-tuning.

As mentioned above, the *dtype* of the storage weights is mostly irrelevant unless you are using *torch_dtype="auto"* when initializing a model using. The reason is that the model will first be downloaded (using the *dtype* of the checkpoints online) and then will be casted to the default *dtype* of *torch* (becomes *torch.float32*). If there is a specified *torch_dtype*, it will be used instead.

</Tip>

Tips:

- These models have the same architecture as the *Llama2* models
- The infilling task is supported out of the box. You should be using the *tokenizer.fill_token* where you want your input to be filled.
- The model conversion script is the same as for the *Llama2* family:

Here is a sample usage ```bash python src/transformers/models/llama/convert_llama_weights_to_hf.py

–input_dir /path/to/downloaded/llama/weights –model_size 7B –output_dir /output/path

``` Note that executing the script requires enough CPU RAM to host the whole model in float16 precision (even if the biggest versions come in several checkpoints they each contain a part of each weight of the model, so we need to load them all in RAM).

- After conversion, the model and tokenizer can be loaded via:

```
>>> from transformers import LlamaForCausalLM, CodeLlamaTokenizer
```

```
>>> tokenizer = CodeLlamaTokenizer.from_pretrained("codellama/CodeLlama-7b-hf")
>>> model = LlamaForCausalLM.from_pretrained("codellama/CodeLlama-7b-hf")
>>> PROMPT = '''def remove_non_ascii(s: str) -> str:
    """ <FILL_ME>
    return result
'''
>>> input_ids = tokenizer(PROMPT, return_tensors="pt")`"input_ids"]
>>> generated_ids = model.generate(input_ids, max_new_tokens=128)
```

```
>>> filling = tokenizer.batch_decode(generated_ids[:, input_ids.shape[1]:], skip_special_
↪tokens = True)[0]
>>> print(PROMPT.replace("<FILL_ME>", filling))
def remove_non_ascii(s: str) -> str:
    """ Remove non-ASCII characters from a string.
```

Args:
s: The string to remove non-ASCII characters from.

Returns:
The string with non-ASCII characters removed.

“”” result = “” for c in s:

if ord(c) < 128:
result += c

return result

If you only want the infilled part:
```

```
>>> from transformers import pipeline
>>> import torch
```

```
>>> generator = pipeline("text-generation",model="codellama/CodeLlama-7b-hf",torch_
↪dtype=torch.float16, device_map="auto")
>>> generator('def remove_non_ascii(s: str) -> str:\n    """ <FILL_ME>\n    return result
↪', max_new_tokens = 128, return_type = 1)
```

Under the hood, the tokenizer [automatically splits by <*FILL_ME*> <https://huggingface.co/docs/transformers/main/model_doc/code_llama#transformers.CodeLlamaTokenizer.fill_token>`__ to create a formatted input string that follows the original training pattern. This is more robust than preparing the pattern yourself: it avoids pitfalls, such as token glueing, that are very hard to debug. To see how much CPU and GPU memory you need for this model or others, try this calculator which can help determine that value.

- The LLaMA tokenizer is a BPE model based on sentencepiece. One quirk of sentencepiece is that when decoding a sequence, if the first token is the start of the word (e.g. "Banana"), the tokenizer does not prepend the prefix space to the string.

This model was contributed by ArthurZucker. The original code of the authors can be found here.

Args:

vocab_size (*int*, *optional*, defaults to 32000):
Vocabulary size of the LLaMA model. Defines the number of different tokens that can be represented by the *inputs_ids* passed when calling `LlamaModel`

hidden_size (*int*, *optional*, defaults to 4096):
Dimension of the hidden representations.

intermediate_size (*int*, *optional*, defaults to 11008):
Dimension of the MLP representations.

num_hidden_layers (*int*, *optional*, defaults to 32):
Number of hidden layers in the Transformer decoder.

num_attention_heads (*int*, *optional*, defaults to 32):
Number of attention heads for each attention layer in the Transformer decoder.

num_key_value_heads (*int*, *optional*):
This is the number of key_value heads that should be used to implement Grouped Query Attention. If *num_key_value_heads=num_attention_heads*, the model will use Multi Head Attention (MHA), if num_key_value_heads=1 the model will use Multi Query Attention (MQA) otherwise GQA is used. When converting a multi-head checkpoint to a GQA checkpoint, each group key and value head should be constructed by meanpooling all the original heads within that group. For more details checkout `this paper. If it is not specified, will default to *num_attention_heads*.

hidden_act (*str* or *function*, *optional*, defaults to *"silu"*):
The non-linear activation function (function or string) in the decoder.

max_position_embeddings (*int*, *optional*, defaults to 2048):
The maximum sequence length that this model might ever be used with. Llama 1 supports up to 2048 tokens, Llama 2 up to 4096, CodeLlama up to 16384.

initializer_range (*float*, *optional*, defaults to 0.02):
The standard deviation of the truncated_normal_initializer for initializing all weight matrices.

rms_norm_eps (*float*, *optional*, defaults to 1e-06):
The epsilon used by the rms normalization layers.

use_cache (*bool*, *optional*, defaults to *True*):
Whether or not the model should return the last key/values attentions (not used by all models). Only relevant if *config.is_decoder=True*.

pad_token_id (*int*, *optional*):
Padding token id.

bos_token_id (*int*, *optional*, defaults to 1):
Beginning of stream token id.

eos_token_id (*int*, *optional*, defaults to 2):
End of stream token id.

pretraining_tp (*int*, *optional*, defaults to 1):
Experimental feature. Tensor parallelism rank used during pretraining. Please refer to this document to understand more about it. This value is necessary to ensure exact reproducibility of the pretraining results. Please refer to this issue.

tie_word_embeddings (*bool*, *optional*, defaults to *False*):
Whether to tie weight embeddings

rope_theta (*float*, *optional*, defaults to 10000.0):
The base period of the RoPE embeddings.

rope_scaling (*Dict*, *optional*):
Dictionary containing the scaling configuration for the RoPE embeddings. Currently supports two scaling strategies: linear and dynamic. Their scaling factor must be a float greater than 1. The expected format is *{"type": strategy name, "factor": scaling factor}*. When using this flag, don't update *max_position_embeddings* to the expected new maximum. See the following thread for more information on how these scaling strategies behave: https://www.reddit.com/r/LocalLLaMA/comments/14mrgpr/dynamically_scaled_rope_further_increases/. This is an experimental feature, subject to breaking API changes in future versions.

attention_bias (*bool*, defaults to *False*, *optional*, defaults to *False*):
Whether to use a bias in the query, key, value and output projection layers during self-attention.

attention_dropout (*float*, *optional*, defaults to 0.0):
The dropout ratio for the attention probabilities.

```
>>> from transformers import LlamaModel, LlamaConfig
```

```
>>> # Initializing a LLaMA llama-7b style configuration
>>> configuration = LlamaConfig()
```

```
>>> # Initializing a model from the llama-7b style configuration
>>> model = LlamaModel(configuration)
```

```
>>> # Accessing the model configuration
>>> configuration = model.config
```

class transformers.models.codegen.configuration_codegen.**CodeGenConfig**(*vocab_size=50400*, *n_positions=2048*, *n_ctx=2048*, *n_embd=4096*, *n_layer=28*, *n_head=16*, *rotary_dim=64*, *n_inner=None*, *activation_function='gelu_new'*, *resid_pdrop=0.0*, *embd_pdrop=0.0*, *attn_pdrop=0.0*, *layer_norm_epsilon=1e-05*, *initializer_range=0.02*, *use_cache=True*, *bos_token_id=50256*, *eos_token_id=50256*, *tie_word_embeddings=False*, ***kwargs*)

The CodeGen model was proposed in A Conversational Paradigm for Program Synthesis by Erik Nijkamp, Bo Pang, Hiroaki Hayashi, Lifu Tu, Huan Wang, Yingbo Zhou, Silvio Savarese, and Caiming Xiong.

CodeGen is an autoregressive language model for program synthesis trained sequentially on The Pile, BigQuery, and BigPython.

The abstract from the paper is the following:

Program synthesis strives to generate a computer program as a solution to a given problem specification. We propose a conversational program synthesis approach via large language models, which addresses the challenges of searching over a vast program space and user intent specification faced in prior approaches. Our new approach casts the process of writing a specification and program as a multi-turn conversation between a user and a system. It treats program synthesis as a sequence prediction problem, in which the specification is expressed in natural language and the desired program is conditionally sampled. We train a family of large language models, called CodeGen, on natural language and programming language data. With weak supervision in the data and the scaling up of data size and model size, conversational capacities emerge from the simple autoregressive language modeling. To study the model behavior on conversational program synthesis, we develop a multi-turn programming benchmark (MTPB), where solving each problem requires multi-step synthesis via multi-turn conversation between the user and the model. Our findings show the emergence of conversational capabilities and the effectiveness of the proposed conversational program synthesis paradigm. In addition, our model CodeGen (with up to 16B parameters trained on TPU-v4) outperforms OpenAI's Codex on the HumanEval benchmark. We make the training library JaxFormer including checkpoints available as open source contribution: `this https URL <https://github.com/salesforce/codegen>`__.

This model was contributed by Hiroaki Hayashi. The original code can be found here.

Args:

vocab_size (*int*, *optional*, defaults to 50400):
Vocabulary size of the CodeGen model. Defines the number of different tokens that can be represented by the *inputs_ids* passed when calling `CodeGenModel`.

n_positions (*int*, *optional*, defaults to 2048):
The maximum sequence length that this model might ever be used with. Typically set this to something large just in case (e.g., 512 or 1024 or 2048).

n_ctx (*int*, *optional*, defaults to 2048):
This attribute is used in *CodeGenModel.__init__* without any real effect.

n_embd (*int*, *optional*, defaults to 4096):
Dimensionality of the embeddings and hidden states.

n_layer (*int*, *optional*, defaults to 28):
Number of hidden layers in the Transformer encoder.

n_head (*int*, *optional*, defaults to 16):
Number of attention heads for each attention layer in the Transformer encoder.

rotary_dim (*int*, *optional*, defaults to 64):
Number of dimensions in the embedding that Rotary Position Embedding is applied to.

n_inner (*int*, *optional*):
Dimensionality of the inner feed-forward layers. *None* will set it to 4 times n_embd

activation_function (*str*, *optional*, defaults to *"gelu_new"*):
Activation function, to be selected in the list *["relu", "silu", "gelu", "tanh", "gelu_new"]*.

resid_pdrop (*float*, *optional*, defaults to 0.0):
The dropout probability for all fully connected layers in the embeddings, encoder, and pooler.

embd_pdrop (*int*, *optional*, defaults to 0.0):
The dropout ratio for the embeddings.

attn_pdrop (*float*, *optional*, defaults to 0.0):
The dropout ratio for the attention.

layer_norm_epsilon (*float*, *optional*, defaults to 1e-05):
The epsilon to use in the layer normalization layers.

initializer_range (*float*, *optional*, defaults to 0.02):
The standard deviation of the truncated_normal_initializer for initializing all weight matrices.

use_cache (*bool*, *optional*, defaults to *True*):
Whether or not the model should return the last key/values attentions (not used by all models).

bos_token_id (*int*, *optional*, defaults to 50256):
Beginning of stream token id.

eos_token_id (*int*, *optional*, defaults to 50256):
End of stream token id.

tie_word_embeddings (*bool*, *optional*, defaults to *False*):
Whether the model's input and output word embeddings should be tied. Note that this is only relevant if the model has a output word embedding layer.

class transformers.models.cohere.configuration_cohere.**CohereConfig**(*vocab_size=256000*, *hidden_size=8192*, *intermediate_size=22528*, *logit_scale=0.0625*, *num_hidden_layers=40*, *num_attention_heads=64*, *num_key_value_heads=None*, *hidden_act='silu'*, *max_position_embeddings=8192*, *initializer_range=0.02*, *layer_norm_eps=1e-05*, *use_cache=True*, *pad_token_id=0*, *bos_token_id=5*, *eos_token_id=255001*, *tie_word_embeddings=True*, *rope_theta=10000.0*, *attention_bias=False*, *attention_dropout=0.0*, ***kwargs*)

The Cohere Command-R model was proposed in the blogpost Command-R: Retrieval Augmented Generation at Production Scale by the Cohere Team.

The abstract from the paper is the following:

Command-R is a scalable generative model targeting RAG and Tool Use to enable production-scale AI for enterprise. Today, we are introducing Command-R, a new LLM aimed at large-scale production workloads. Command-R targets the emerging "scalable" category of models that balance high efficiency with strong accuracy, enabling companies to move beyond proof of concept, and into production.

*Command-R is a generative model optimized for long context tasks such as retrieval augmented generation (RAG) and using external APIs and tools. It is designed to work in concert with our industry-leading Embed and Rerank models to provide best-in-class integration for RAG applications and excel at enterprise use cases. As a model built for companies to implement at scale, Command-R boasts: - Strong accuracy on RAG and Tool Use - Low latency, and high throughput - Longer 128k context and lower pricing - Strong capabilities across 10 key languages - Model weights available on HuggingFace for research and evaluation

Checkout model checkpoints here. This model was contributed by Saurabh Dash and Ahmet Üstün. The code of the implementation in Hugging Face is based on GPT-NeoX here.

Args:

vocab_size (*int*, *optional*, defaults to 256000):
Vocabulary size of the Cohere model. Defines the number of different tokens that can be represented by the *inputs_ids* passed when calling `CohereModel`

hidden_size (*int*, *optional*, defaults to 8192):
Dimension of the hidden representations.

intermediate_size (*int*, *optional*, defaults to 22528):
Dimension of the MLP representations.

logit_scale (*float*, *optional*, defaults to 0.0625):
The scaling factor for the output logits.

num_hidden_layers (*int*, *optional*, defaults to 40):
Number of hidden layers in the Transformer decoder.

num_attention_heads (*int*, *optional*, defaults to 64):
Number of attention heads for each attention layer in the Transformer decoder.

num_key_value_heads (*int*, *optional*):
This is the number of key_value heads that should be used to implement Grouped Query Attention. If *num_key_value_heads=num_attention_heads*, the model will use Multi Head Attention (MHA), if num_key_value_heads=1 the model will use Multi Query Attention (MQA) otherwise GQA is used. When converting a multi-head checkpoint to a GQA checkpoint, each group key and value head should be constructed by meanpooling all the original heads within that group. For more details checkout `this paper. If it is not specified, will default to *num_attention_heads*.

hidden_act (*str* or *function*, *optional*, defaults to *"silu"*):
The non-linear activation function (function or string) in the decoder.

max_position_embeddings (*int*, *optional*, defaults to 8192):
The maximum sequence length that this model might ever be used with.

initializer_range (*float*, *optional*, defaults to 0.02):
The standard deviation of the truncated_normal_initializer for initializing all weight matrices.

layer_norm_eps (*float*, *optional*, defaults to 1e-05):
The epsilon used by the layer normalization.

use_cache (*bool*, *optional*, defaults to *True*):
Whether or not the model should return the last key/values attentions (not used by all models). Only relevant if *config.is_decoder=True*.

pad_token_id (*int*, *optional*, defaults to 0):
Padding token id.

bos_token_id (*int*, *optional*, defaults to 5):
Beginning of stream token id.

eos_token_id (*int*, *optional*, defaults to 255001):
End of stream token id.

tie_word_embeddings (*bool*, *optional*, defaults to *True*):
Whether to tie weight embeddings

rope_theta (*float*, *optional*, defaults to 10000.0):
The base period of the RoPE embeddings.

attention_bias (*bool*, defaults to *False*, *optional*, defaults to *False*):
Whether to use a bias in the query, key, value and output projection layers during self-attention.

attention_dropout (*float*, *optional*, defaults to 0.0):
The dropout ratio for the attention probabilities.

```
>>> from transformers import CohereModel, CohereConfig
```

```
>>> # Initializing a Cohere model configuration
>>> configuration = CohereConfig()
```

```
>>> # Initializing a model from the Cohere configuration
>>> model = CohereModel(configuration)
```

```
>>> # Accessing the model configuration
>>> configuration = model.config
```

class transformers.models.ctrl.configuration_ctrl.**CTRLConfig**(*vocab_size=246534*, *n_positions=256*, *n_embd=1280*, *dff=8192*, *n_layer=48*, *n_head=16*, *resid_pdrop=0.1*, *embd_pdrop=0.1*, *layer_norm_epsilon=1e-06*, *initializer_range=0.02*, *use_cache=True*, ***kwargs*)

CTRL model was proposed in CTRL: A Conditional Transformer Language Model for Controllable Generation by Nitish Shirish Keskar*, Bryan McCann*, Lav R. Varshney, Caiming Xiong and Richard Socher. It's a causal (unidirectional) transformer pre-trained using language modeling on a very large corpus of ~140 GB of text data with the first token reserved as a control code (such as Links, Books, Wikipedia etc.).

The abstract from the paper is the following:

Large-scale language models show promising text generation capabilities, but users cannot easily control particular aspects of the generated text. We release CTRL, a 1.63 billion-parameter conditional transformer language model, trained to condition on control codes that govern style, content, and task-specific behavior. Control codes were derived from structure that naturally co-occurs with raw text, preserving the advantages of unsupervised learning while providing more explicit control over text generation. These codes also allow CTRL to predict which parts of the training data are most likely given a sequence. This provides a potential method for analyzing large amounts of data via model-based source attribution.

Tips:

- CTRL makes use of control codes to generate text: it requires generations to be started by certain words, sentences or links to generate coherent text. Refer to the original implementation for more information.
- CTRL is a model with absolute position embeddings so it's usually advised to pad the inputs on the right rather than the left.
- CTRL was trained with a causal language modeling (CLM) objective and is therefore powerful at predicting the next token in a sequence. Leveraging this feature allows CTRL to generate syntactically coherent text as it can be observed in the *run_generation.py* example script.
- The PyTorch models can take the *past_key_values* as input, which is the previously computed key/value attention pairs. TensorFlow models accepts *past* as input. Using the *past_key_values* value prevents the model from re-computing pre-computed values in the context of text generation. See the ``forward``(model_doc/ctrl#transformers.CTRLModel.forward) method for more information on the usage of this argument.

This model was contributed by keskarnitishr. The original code can be found here.

Args:

vocab_size (*int*, *optional*, defaults to 246534):
Vocabulary size of the CTRL model. Defines the number of different tokens that can be represented by the *inputs_ids* passed when calling `CTRLModel` or `TFCTRLModel`.

n_positions (*int*, *optional*, defaults to 256):
The maximum sequence length that this model might ever be used with. Typically set this to something large just in case (e.g., 512 or 1024 or 2048).

n_embd (*int*, *optional*, defaults to 1280):
Dimensionality of the embeddings and hidden states.

dff (*int*, *optional*, defaults to 8192):
Dimensionality of the inner dimension of the feed forward networks (FFN).

n_layer (*int*, *optional*, defaults to 48):
Number of hidden layers in the Transformer encoder.

n_head (*int*, *optional*, defaults to 16):
Number of attention heads for each attention layer in the Transformer encoder.

resid_pdrop (*float*, *optional*, defaults to 0.1):
The dropout probability for all fully connected layers in the embeddings, encoder, and pooler.

embd_pdrop (*int*, *optional*, defaults to 0.1):
The dropout ratio for the embeddings.

layer_norm_epsilon (*float*, *optional*, defaults to 1e-06):
The epsilon to use in the layer normalization layers

initializer_range (*float*, *optional*, defaults to 0.02):
The standard deviation of the truncated_normal_initializer for initializing all weight matrices.

use_cache (*bool*, *optional*, defaults to *True*):
Whether or not the model should return the last key/values attentions (not used by all models).

`class` transformers.models.data2vec.configuration_data2vec_text.**Data2VecTextConfig**(*vocab_size=30522*, *hidden_size=768*, *num_hidden_layers=12*, *num_attention_heads=12*, *intermediate_size=3072*, *hidden_act='gelu'*, *hidden_dropout_prob=0.1*, *attention_probs_dropout_prob=*(
max_position_embeddings=
type_vocab_size=2, *initializer_range=0.02*, *layer_norm_eps=1e-12*, *pad_token_id=1*, *bos_token_id=0*, *eos_token_id=2*, *position_embedding_type='absc*
use_cache=True, *classifier_dropout=None*, ***kwargs*)

This is the configuration class to store the configuration of a `Data2VecTextModel` and `Data2VecTextModel`. It is used to instantiate a Data2VecText model according to the specified arguments, defining the model architecture. Instantiating a configuration with the defaults will yield a similar configuration to that of the Data2VecText facebook/data2vec-text-base architecture.

Configuration objects inherit from `PretrainedConfig` and can be used to control the model outputs. Read the documentation from `PretrainedConfig` for more information.

Args:

vocab_size (*int*, *optional*, defaults to 30522):
Vocabulary size of the DATA2VEC model. Defines the number of different tokens that can be represented by the *inputs_ids* passed when calling `Data2VecModel`.

hidden_size (*int*, *optional*, defaults to 768):
Dimensionality of the encoder layers and the pooler layer.

num_hidden_layers (*int*, *optional*, defaults to 12):
Number of hidden layers in the Transformer encoder.

num_attention_heads (*int*, *optional*, defaults to 12):
Number of attention heads for each attention layer in the Transformer encoder.

intermediate_size (*int*, *optional*, defaults to 3072):
Dimensionality of the "intermediate" (often named feed-forward) layer in the Transformer encoder.

hidden_act (*str* or *Callable*, *optional*, defaults to *"gelu"*):
The non-linear activation function (function or string) in the encoder and pooler. If string, *"gelu"*, *"relu"*, *"silu"* and *"gelu_new"* are supported.

hidden_dropout_prob (*float*, *optional*, defaults to 0.1):
The dropout probability for all fully connected layers in the embeddings, encoder, and pooler.

attention_probs_dropout_prob (*float*, *optional*, defaults to 0.1):
The dropout ratio for the attention probabilities.

max_position_embeddings (*int*, *optional*, defaults to 512):
The maximum sequence length that this model might ever be used with. Typically set this to something large just in case (e.g., 512 or 1024 or 2048).

type_vocab_size (*int*, *optional*, defaults to 2):
The vocabulary size of the *token_type_ids* passed when calling `Data2VecModel`.

initializer_range (*float*, *optional*, defaults to 0.02):
The standard deviation of the truncated_normal_initializer for initializing all weight matrices.

layer_norm_eps (*float*, *optional*, defaults to 1e-12):
The epsilon used by the layer normalization layers.

position_embedding_type (*str*, *optional*, defaults to *"absolute"*):
Type of position embedding. Choose one of *"absolute"*, *"relative_key"*, *"relative_key_query"*. For positional embeddings use *"absolute"*. For more information on *"relative_key"*, please refer to Self-Attention with Relative Position Representations (Shaw et al.). For more information on *"relative_key_query"*, please refer to *Method 4* in Improve Transformer Models with Better Relative Position Embeddings (Huang et al.).

is_decoder (*bool*, *optional*, defaults to *False*):
Whether the model is used as a decoder or not. If *False*, the model is used as an encoder.

use_cache (*bool*, *optional*, defaults to *True*):
Whether or not the model should return the last key/values attentions (not used by all models). Only relevant if *config.is_decoder=True*.

classifier_dropout (*float*, *optional*):
The dropout ratio for the classification head.

class transformers.models.deberta.configuration_deberta.**DebertaConfig**(*vocab_size=50265*, *hidden_size=768*, *num_hidden_layers=12*, *num_attention_heads=12*, *intermediate_size=3072*, *hidden_act='gelu'*, *hidden_dropout_prob=0.1*, *attention_probs_dropout_prob=0.1*, *max_position_embeddings=512*, *type_vocab_size=0*, *initializer_range=0.02*, *layer_norm_eps=1e-07*, *relative_attention=False*, *max_relative_positions=-1*, *pad_token_id=0*, *position_biased_input=True*, *pos_att_type=None*, *pooler_dropout=0*, *pooler_hidden_act='gelu'*, ***kwargs*)

The DeBERTa model was proposed in DeBERTa: Decoding-enhanced BERT with Disentangled Attention by Pengcheng He, Xiaodong Liu, Jianfeng Gao, Weizhu Chen It is based on Google's BERT model released in 2018 and Facebook's RoBERTa model released in 2019.

It builds on RoBERTa with disentangled attention and enhanced mask decoder training with half of the data used in RoBERTa.

The abstract from the paper is the following:

Recent progress in pre-trained neural language models has significantly improved the performance of many natural language processing (NLP) tasks. In this paper we propose a new model architecture DeBERTa (Decoding-enhanced BERT with disentangled attention) that improves the BERT and RoBERTa models using two novel techniques. The first is the disentangled attention mechanism, where each word is represented using two vectors that encode its content and position, respectively, and the attention weights among words are computed using disentangled matrices on their contents and relative positions. Second, an enhanced mask decoder is used to replace the output softmax layer to predict the masked tokens for model pretraining. We show that these two techniques significantly improve the efficiency of model pretraining and performance of downstream tasks. Compared to RoBERTa-Large, a DeBERTa model trained on half of the training data performs consistently better on a wide range of NLP tasks, achieving improvements on MNLI by +0.9% (90.2% vs. 91.1%), on SQuAD v2.0 by +2.3% (88.4% vs. 90.7%) and RACE by +3.6% (83.2% vs. 86.8%). The DeBERTa code and pre-trained models will be made publicly available at https://github.com/microsoft/DeBERTa.

This model was contributed by DeBERTa. This model TF 2.0 implementation was contributed by kamalkraj . The original code can be found here.

Arguments:

vocab_size (*int*, *optional*, defaults to 30522):
Vocabulary size of the DeBERTa model. Defines the number of different tokens that can be represented by the *inputs_ids* passed when calling `DebertaModel` or `TFDebertaModel`.

hidden_size (*int*, *optional*, defaults to 768):
Dimensionality of the encoder layers and the pooler layer.

num_hidden_layers (*int*, *optional*, defaults to 12):
Number of hidden layers in the Transformer encoder.

num_attention_heads (*int*, *optional*, defaults to 12):
Number of attention heads for each attention layer in the Transformer encoder.

intermediate_size (*int*, *optional*, defaults to 3072):
Dimensionality of the "intermediate" (often named feed-forward) layer in the Transformer encoder.

hidden_act (*str* or *Callable*, *optional*, defaults to *"gelu"*):
The non-linear activation function (function or string) in the encoder and pooler. If string, *"gelu"*, *"relu"*, *"silu"*, *"gelu"*, *"tanh"*, *"gelu_fast"*, *"mish"*, *"linear"*, *"sigmoid"* and *"gelu_new"* are supported.

hidden_dropout_prob (*float*, *optional*, defaults to 0.1):
The dropout probability for all fully connected layers in the embeddings, encoder, and pooler.

attention_probs_dropout_prob (*float*, *optional*, defaults to 0.1):
The dropout ratio for the attention probabilities.

max_position_embeddings (*int*, *optional*, defaults to 512):
The maximum sequence length that this model might ever be used with. Typically set this to something large just in case (e.g., 512 or 1024 or 2048).

type_vocab_size (*int*, *optional*, defaults to 2):
The vocabulary size of the *token_type_ids* passed when calling `DebertaModel` or `TFDebertaModel`.

initializer_range (*float*, *optional*, defaults to 0.02):
The standard deviation of the truncated_normal_initializer for initializing all weight matrices.

layer_norm_eps (*float*, *optional*, defaults to 1e-12):
The epsilon used by the layer normalization layers.

relative_attention (*bool*, *optional*, defaults to *False*):
Whether use relative position encoding.

max_relative_positions (*int*, *optional*, defaults to 1):
The range of relative positions *[-max_position_embeddings, max_position_embeddings]*. Use the same value as *max_position_embeddings*.

pad_token_id (*int*, *optional*, defaults to 0):
The value used to pad input_ids.

position_biased_input (*bool*, *optional*, defaults to *True*):
Whether add absolute position embedding to content embedding.

pos_att_type (*List[str]*, *optional*):
The type of relative position attention, it can be a combination of *["p2c", "c2p"]*, e.g. *["p2c"]*, *["p2c", "c2p"]*.

layer_norm_eps (*float*, optional, defaults to 1e-12):
The epsilon used by the layer normalization layers.

class transformers.models.deberta_v2.configuration_deberta_v2.**DebertaV2Config**(*vocab_size=128100*, *hidden_size=1536*, *num_hidden_layers=24*, *num_attention_heads=24*, *intermediate_size=6144*, *hidden_act='gelu'*, *hidden_dropout_prob=0.1*, *attention_probs_dropout_prob=0.1*, *max_position_embeddings=512*, *type_vocab_size=0*, *initializer_range=0.02*, *layer_norm_eps=1e-07*, *relative_attention=False*, *max_relative_positions=-1*, *pad_token_id=0*, *position_biased_input=True*, *pos_att_type=None*, *pooler_dropout=0*, *pooler_hidden_act='gelu'*, ***kwargs*)

The DeBERTa model was proposed in DeBERTa: Decoding-enhanced BERT with Disentangled Attention by Pengcheng He, Xiaodong Liu, Jianfeng Gao, Weizhu Chen It is based on Google's BERT model released in 2018 and Facebook's RoBERTa model released in 2019.

It builds on RoBERTa with disentangled attention and enhanced mask decoder training with half of the data used in RoBERTa.

The abstract from the paper is the following:

Recent progress in pre-trained neural language models has significantly improved the performance of many natural language processing (NLP) tasks. In this paper we propose a new model architecture DeBERTa (Decoding-enhanced BERT with disentangled attention) that improves the BERT and RoBERTa models using two novel techniques. The first is the disentangled attention mechanism, where each word is represented using two vectors that encode its content and position, respectively, and the attention weights among words are computed using disentangled matrices on their contents and relative positions. Second, an enhanced mask decoder is used to replace the output softmax layer to predict the masked tokens for model pretraining. We show that these two techniques significantly improve the efficiency of model pretraining and performance of downstream tasks. Compared to RoBERTa-Large, a DeBERTa model trained on half of the training data performs consistently better on a wide range of NLP tasks, achieving improvements on MNLI by +0.9% (90.2% vs. 91.1%), on SQuAD v2.0 by +2.3% (88.4% vs. 90.7%) and RACE by +3.6% (83.2% vs. 86.8%). The DeBERTa code and pre-trained models will be made publicly available at https://github.com/microsoft/DeBERTa.

The following information is visible directly on the original implementation repository. DeBERTa v2 is the second version of the DeBERTa model. It includes the 1.5B model used for the SuperGLUE single-model submission and achieving 89.9, versus human baseline 89.8. You can find more details about this submission in the authors' blog

New in v2:

- **Vocabulary** In v2 the tokenizer is changed to use a new vocabulary of size 128K built from the training data. Instead of a GPT2-based tokenizer, the tokenizer is now sentencepiece-based tokenizer.
- **nGiE(nGram Induced Input Encoding)** The DeBERTa-v2 model uses an additional convolution layer aside with the first transformer layer to better learn the local dependency of input tokens.
- **Sharing position projection matrix with content projection matrix in attention layer** Based on previous experiments, this can save parameters without affecting the performance.
- **Apply bucket to encode relative positions** The DeBERTa-v2 model uses log bucket to encode relative positions similar to T5.
- **900M model & 1.5B model** Two additional model sizes are available: 900M and 1.5B, which significantly improves the performance of downstream tasks.

This model was contributed by DeBERTa. This model TF 2.0 implementation was contributed by kamalkraj. The original code can be found here.

Arguments:

vocab_size (*int*, *optional*, defaults to 128100):
Vocabulary size of the DeBERTa-v2 model. Defines the number of different tokens that can be represented by the *inputs_ids* passed when calling `DebertaV2Model`.

hidden_size (*int*, *optional*, defaults to 1536):
Dimensionality of the encoder layers and the pooler layer.

num_hidden_layers (*int*, *optional*, defaults to 24):
Number of hidden layers in the Transformer encoder.

num_attention_heads (*int*, *optional*, defaults to 24):
Number of attention heads for each attention layer in the Transformer encoder.

intermediate_size (*int*, *optional*, defaults to 6144):
Dimensionality of the "intermediate" (often named feed-forward) layer in the Transformer encoder.

hidden_act (*str* or *Callable*, *optional*, defaults to *"gelu"*):
The non-linear activation function (function or string) in the encoder and pooler. If string, *"gelu"*, *"relu"*, *"silu"*, *"gelu"*, *"tanh"*, *"gelu_fast"*, *"mish"*, *"linear"*, *"sigmoid"* and *"gelu_new"* are supported.

hidden_dropout_prob (*float*, *optional*, defaults to 0.1):
The dropout probability for all fully connected layers in the embeddings, encoder, and pooler.

attention_probs_dropout_prob (*float*, *optional*, defaults to 0.1):
The dropout ratio for the attention probabilities.

max_position_embeddings (*int*, *optional*, defaults to 512):
The maximum sequence length that this model might ever be used with. Typically set this to something large just in case (e.g., 512 or 1024 or 2048).

type_vocab_size (*int*, *optional*, defaults to 0):
The vocabulary size of the *token_type_ids* passed when calling `DebertaModel` or `TFDebertaModel`.

initializer_range (*float*, *optional*, defaults to 0.02):
The standard deviation of the truncated_normal_initializer for initializing all weight matrices.

layer_norm_eps (*float*, *optional*, defaults to 1e-7):
The epsilon used by the layer normalization layers.

relative_attention (*bool*, *optional*, defaults to *True*):
Whether use relative position encoding.

max_relative_positions (*int*, *optional*, defaults to -1):
The range of relative positions *[-max_position_embeddings, max_position_embeddings]*. Use the same value as *max_position_embeddings*.

pad_token_id (*int*, *optional*, defaults to 0):
The value used to pad input_ids.

position_biased_input (*bool*, *optional*, defaults to *False*):
Whether add absolute position embedding to content embedding.

pos_att_type (*List[str]*, *optional*):
The type of relative position attention, it can be a combination of *["p2c", "c2p"]*, e.g. *["p2c"]*, *["p2c", "c2p"]*, *["p2c", "c2p"]*.

layer_norm_eps (*float*, optional, defaults to 1e-12):
The epsilon used by the layer normalization layers.

`class transformers.models.distilbert.configuration_distilbert.DistilBertConfig(`*vocab_size=30522*, *max_position_embeddings=512*, *sinusoidal_pos_embds=False*, *n_layers=6*, *n_heads=12*, *dim=768*, *hidden_dim=3072*, *dropout=0.1*, *attention_dropout=0.1*, *activation='gelu'*, *initializer_range=0.02*, *qa_dropout=0.1*, *seq_classif_dropout=0.2*, *pad_token_id=0*, ***kwargs*)

The DistilBERT model was proposed in the blog post Smaller, faster, cheaper, lighter: Introducing DistilBERT, a distilled version of BERT, and the paper DistilBERT, a distilled version of BERT: smaller, faster, cheaper and lighter. DistilBERT is a small, fast, cheap and light Transformer model trained by distilling BERT base. It has 40% less parameters than *bert-base-uncased*, runs 60% faster while preserving over 95% of BERT's performances as measured on the GLUE language understanding benchmark.

The abstract from the paper is the following:

As Transfer Learning from large-scale pre-trained models becomes more prevalent in Natural Language Processing (NLP), operating these large models in on-the-edge and/or under constrained computational training or inference budgets remains challenging. In this work, we propose a method to pre-train a smaller general-purpose language representation model, called DistilBERT, which can then be fine-tuned with good performances on a wide range of tasks like its larger counterparts. While most prior work investigated the use of distillation for building task-specific models, we leverage knowledge distillation during the pretraining phase and show that it is possible to reduce the size of a BERT model by 40%, while retaining 97% of its language understanding capabilities and being 60% faster. To leverage the inductive biases learned by larger models during pretraining, we introduce a triple loss combining language modeling, distillation and cosine-distance losses. Our smaller, faster and lighter model is cheaper to pre-train and we demonstrate its capabilities for on-device computations in a proof-of-concept experiment and a comparative on-device study.

Tips:

- DistilBERT doesn't have *token_type_ids*, you don't need to indicate which token belongs to which segment. Just separate your segments with the separation token *tokenizer.sep_token* (or ``SEP]`).
- DistilBERT doesn't have options to select the input positions (*position_ids* input). This could be added if necessary though, just let us know if you need this option.
- Same as BERT but smaller. Trained by distillation of the pretrained BERT model, meaning it's been trained to predict the same probabilities as the larger model. The actual objective is a combination of:
  - finding the same probabilities as the teacher model
  - predicting the masked tokens correctly (but no next-sentence objective)
  - a cosine similarity between the hidden states of the student and the teacher model

This model was contributed by [victorsanh <https://huggingface.co/victorsanh>`__. This model jax version was contributed by kamalkraj. The original code can be found here.

Args:

vocab_size (*int*, *optional*, defaults to 30522):
Vocabulary size of the DistilBERT model. Defines the number of different tokens that can be represented by the *inputs_ids* passed when calling `DistilBertModel` or `TFDistilBertModel`.

max_position_embeddings (*int*, *optional*, defaults to 512):
The maximum sequence length that this model might ever be used with. Typically set this to something large just in case (e.g., 512 or 1024 or 2048).

sinusoidal_pos_embds (*boolean*, *optional*, defaults to *False*):
Whether to use sinusoidal positional embeddings.

n_layers (*int*, *optional*, defaults to 6):
Number of hidden layers in the Transformer encoder.

n_heads (*int*, *optional*, defaults to 12):
Number of attention heads for each attention layer in the Transformer encoder.

dim (*int*, *optional*, defaults to 768):
Dimensionality of the encoder layers and the pooler layer.

hidden_dim (*int*, *optional*, defaults to 3072):
The size of the "intermediate" (often named feed-forward) layer in the Transformer encoder.

dropout (*float*, *optional*, defaults to 0.1):
The dropout probability for all fully connected layers in the embeddings, encoder, and pooler.

attention_dropout (*float*, *optional*, defaults to 0.1):
The dropout ratio for the attention probabilities.

activation (*str* or *Callable*, *optional*, defaults to *"gelu"*):
The non-linear activation function (function or string) in the encoder and pooler. If string, *"gelu"*, *"relu"*, *"silu"* and *"gelu_new"* are supported.

initializer_range (*float*, *optional*, defaults to 0.02):
The standard deviation of the truncated_normal_initializer for initializing all weight matrices.

qa_dropout (*float*, *optional*, defaults to 0.1):
The dropout probabilities used in the question answering model `DistilBertForQuestionAnswering`.

seq_classif_dropout (*float*, *optional*, defaults to 0.2):
The dropout probabilities used in the sequence classification and the multiple choice model `DistilBertForSequenceClassification`.

class transformers.models.electra.configuration_electra.**ElectraConfig**(*vocab_size=30522*, *embedding_size=128*, *hidden_size=256*, *num_hidden_layers=12*, *num_attention_heads=4*, *intermediate_size=1024*, *hidden_act='gelu'*, *hidden_dropout_prob=0.1*, *attention_probs_dropout_prob=0.1*, *max_position_embeddings=512*, *type_vocab_size=2*, *initializer_range=0.02*, *layer_norm_eps=1e-12*, *summary_type='first'*, *summary_use_proj=True*, *summary_activation='gelu'*, *summary_last_dropout=0.1*, *pad_token_id=0*, *position_embedding_type='absolute'*, *use_cache=True*, *classifier_dropout=None*, ***kwargs*)

The ELECTRA model was proposed in the paper ELECTRA: Pre-training Text Encoders as Discriminators Rather Than Generators. ELECTRA is a new pretraining approach which trains two transformer models: the generator and the discriminator. The generator's role is to replace tokens in a sequence, and is therefore trained as a masked language model. The discriminator, which is the model we're interested in, tries to identify which tokens were replaced by the generator in the sequence.

The abstract from the paper is the following:

Masked language modeling (MLM) pretraining methods such as BERT corrupt the input by replacing some tokens with `MASK] and then train a model to reconstruct the original tokens. While they produce good results when transferred to downstream NLP tasks, they generally require large amounts of compute to be effective. As an alternative, we propose a more sample-efficient pretraining task called replaced token detection. Instead of masking the input, our approach corrupts it by replacing some tokens with plausible alternatives sampled from a small generator network. Then, instead of training a model that predicts the original identities of the corrupted tokens, we train a discriminative model that predicts whether each token in the corrupted input was replaced by a generator sample or not. Thorough experiments demonstrate this new pretraining task is more efficient than MLM because the task is defined over all input tokens rather than just the small subset that was masked out. As a result, the contextual representations learned by our approach substantially outperform the ones learned by BERT given the same model size, data, and compute. The gains are particularly strong for small models; for example, we train a model on one GPU for 4 days that outperforms GPT (trained using 30x more compute) on the GLUE natural language understanding benchmark. Our approach also works well at scale, where it performs comparably to RoBERTa and XLNet while using less than 1/4 of their compute and outperforms them when using the same amount of compute.

Tips:

- ELECTRA is the pretraining approach, therefore there is nearly no changes done to the underlying model: BERT. The only change is the separation of the embedding size and the hidden size: the embedding size is generally smaller, while the hidden size is larger. An additional projection layer (linear) is used to project the embeddings from their embedding size to the hidden size. In the case where the embedding size is the same as the hidden

size, no projection layer is used.

- ELECTRA is a transformer model pretrained with the use of another (small) masked language model. The inputs are corrupted by that language model, which takes an input text that is randomly masked and outputs a text in which ELECTRA has to predict which token is an original and which one has been replaced. Like for GAN training, the small language model is trained for a few steps (but with the original texts as objective, not to fool the ELECTRA model like in a traditional GAN setting) then the ELECTRA model is trained for a few steps.
- The ELECTRA checkpoints saved using [Google Research's implementation <https://github.com/google-research/electra>`__ contain both the generator and discriminator. The conversion script requires the user to name which model to export into the correct architecture. Once converted to the HuggingFace format, these checkpoints may be loaded into all available ELECTRA models, however. This means that the discriminator may be loaded in the `ElectraForMaskedLM` model, and the generator may be loaded in the `ElectraForPreTraining` model (the classification head will be randomly initialized as it doesn't exist in the generator).

This model was contributed by lysandre. The original code can be found here.

Args:

vocab_size (*int*, *optional*, defaults to 30522):
Vocabulary size of the ELECTRA model. Defines the number of different tokens that can be represented by the *inputs_ids* passed when calling `ElectraModel` or `TFElectraModel`.

embedding_size (*int*, *optional*, defaults to 128):
Dimensionality of the encoder layers and the pooler layer.

hidden_size (*int*, *optional*, defaults to 256):
Dimensionality of the encoder layers and the pooler layer.

num_hidden_layers (*int*, *optional*, defaults to 12):
Number of hidden layers in the Transformer encoder.

num_attention_heads (*int*, *optional*, defaults to 4):
Number of attention heads for each attention layer in the Transformer encoder.

intermediate_size (*int*, *optional*, defaults to 1024):
Dimensionality of the "intermediate" (i.e., feed-forward) layer in the Transformer encoder.

hidden_act (*str* or *Callable*, *optional*, defaults to *"gelu"*):
The non-linear activation function (function or string) in the encoder and pooler. If string, *"gelu"*, *"relu"*, *"silu"* and *"gelu_new"* are supported.

hidden_dropout_prob (*float*, *optional*, defaults to 0.1):
The dropout probability for all fully connected layers in the embeddings, encoder, and pooler.

attention_probs_dropout_prob (*float*, *optional*, defaults to 0.1):
The dropout ratio for the attention probabilities.

max_position_embeddings (*int*, *optional*, defaults to 512):
The maximum sequence length that this model might ever be used with. Typically set this to something large just in case (e.g., 512 or 1024 or 2048).

type_vocab_size (*int*, *optional*, defaults to 2):
The vocabulary size of the *token_type_ids* passed when calling `ElectraModel` or `TFElectraModel`.

initializer_range (*float*, *optional*, defaults to 0.02):
The standard deviation of the truncated_normal_initializer for initializing all weight matrices.

layer_norm_eps (*float*, *optional*, defaults to 1e-12):
The epsilon used by the layer normalization layers.

summary_type (*str*, *optional*, defaults to *"first"*):
Argument used when doing sequence summary. Used in the sequence classification and multiple choice models.

Has to be one of the following options:

- *"last"*: Take the last token hidden state (like XLNet).
- *"first"*: Take the first token hidden state (like BERT).
- *"mean"*: Take the mean of all tokens hidden states.
- *"cls_index"*: Supply a Tensor of classification token position (like GPT/GPT-2).
- *"attn"*: Not implemented now, use multi-head attention.

summary_use_proj (*bool*, *optional*, defaults to *True*):
Argument used when doing sequence summary. Used in the sequence classification and multiple choice models.

Whether or not to add a projection after the vector extraction.

summary_activation (*str*, *optional*):
Argument used when doing sequence summary. Used in the sequence classification and multiple choice models.

Pass *"gelu"* for a gelu activation to the output, any other value will result in no activation.

summary_last_dropout (*float*, *optional*, defaults to 0.0):
Argument used when doing sequence summary. Used in the sequence classification and multiple choice models.

The dropout ratio to be used after the projection and activation.

position_embedding_type (*str*, *optional*, defaults to *"absolute"*):
Type of position embedding. Choose one of *"absolute"*, *"relative_key"*, *"relative_key_query"*. For positional embeddings use *"absolute"*. For more information on *"relative_key"*, please refer to Self-Attention with Relative Position Representations (Shaw et al.). For more information on *"relative_key_query"*, please refer to *Method 4* in Improve Transformer Models with Better Relative Position Embeddings (Huang et al.).

use_cache (*bool*, *optional*, defaults to *True*):
Whether or not the model should return the last key/values attentions (not used by all models). Only relevant if *config.is_decoder=True*.

classifier_dropout (*float*, *optional*):
The dropout ratio for the classification head.

class transformers.models.ernie.configuration_ernie.**ErnieConfig**(*vocab_size=30522*, *hidden_size=768*, *num_hidden_layers=12*, *num_attention_heads=12*, *intermediate_size=3072*, *hidden_act='gelu'*, *hidden_dropout_prob=0.1*, *attention_probs_dropout_prob=0.1*, *max_position_embeddings=512*, *type_vocab_size=2*, *task_type_vocab_size=3*, *use_task_id=False*, *initializer_range=0.02*, *layer_norm_eps=1e-12*, *pad_token_id=0*, *position_embedding_type='absolute'*, *use_cache=True*, *classifier_dropout=None*, ***kwargs*)

ERNIE is a series of powerful models proposed by baidu, especially in Chinese tasks, including ERNIE1.0, ERNIE2.0, ERNIE3.0, ERNIE-Gram, ERNIE-health, etc.

These models are contributed by nghuyong and the official code can be found in PaddleNLP (in PaddlePaddle).

#Args:

vocab_size (*int*, *optional*, defaults to 30522):
Vocabulary size of the ERNIE model. Defines the number of different tokens that can be represented by the *inputs_ids* passed when calling `ErnieModel` or `TFErnieModel`.

hidden_size (*int*, *optional*, defaults to 768):
Dimensionality of the encoder layers and the pooler layer.

num_hidden_layers (*int*, *optional*, defaults to 12):
Number of hidden layers in the Transformer encoder.

num_attention_heads (*int*, *optional*, defaults to 12):
Number of attention heads for each attention layer in the Transformer encoder.

intermediate_size (*int*, *optional*, defaults to 3072):
Dimensionality of the "intermediate" (often named feed-forward) layer in the Transformer encoder.

hidden_act (*str* or *Callable*, *optional*, defaults to *"gelu"*):
The non-linear activation function (function or string) in the encoder and pooler. If string, *"gelu"*, *"relu"*, *"silu"* and *"gelu_new"* are supported.

hidden_dropout_prob (*float*, *optional*, defaults to 0.1):
The dropout probability for all fully connected layers in the embeddings, encoder, and pooler.

attention_probs_dropout_prob (*float*, *optional*, defaults to 0.1):
The dropout ratio for the attention probabilities.

max_position_embeddings (*int*, *optional*, defaults to 512):
The maximum sequence length that this model might ever be used with. Typically set this to something large just in case (e.g., 512 or 1024 or 2048).

type_vocab_size (*int*, *optional*, defaults to 2):
The vocabulary size of the *token_type_ids* passed when calling `ErnieModel` or `TFErnieModel`.

task_type_vocab_size (*int*, *optional*, defaults to 3):
The vocabulary size of the *task_type_ids* for ERNIE2.0/ERNIE3.0 model

use_task_id (*bool*, *optional*, defaults to *False*):
Whether or not the model support *task_type_ids*

initializer_range (*float*, *optional*, defaults to 0.02):
The standard deviation of the truncated_normal_initializer for initializing all weight matrices.

layer_norm_eps (*float*, *optional*, defaults to 1e-12):
The epsilon used by the layer normalization layers.

pad_token_id (*int*, *optional*, defaults to 0):
Padding token id.

position_embedding_type (*str*, *optional*, defaults to *"absolute"*):
Type of position embedding. Choose one of *"absolute"*, *"relative_key"*, *"relative_key_query"*. For positional embeddings use *"absolute"*. For more information on *"relative_key"*, please refer to Self-Attention with Relative Position Representations (Shaw et al.). For more information on *"relative_key_query"*, please refer to *Method 4* in Improve Transformer Models with Better Relative Position Embeddings (Huang et al.).

use_cache (*bool*, *optional*, defaults to *True*):
Whether or not the model should return the last key/values attentions (not used by all models). Only relevant if *config.is_decoder=True*.

classifier_dropout (*float*, *optional*):
The dropout ratio for the classification head.

`class transformers.models.falcon.configuration_falcon.`**`FalconConfig`**(*vocab_size=65024*, *hidden_size=4544*, *num_hidden_layers=32*, *num_attention_heads=71*, *layer_norm_epsilon=1e-05*, *initializer_range=0.02*, *use_cache=True*, *hidden_dropout=0.0*, *attention_dropout=0.0*, *num_kv_heads=None*, *alibi=False*, *new_decoder_architecture=False*, *multi_query=True*, *parallel_attn=True*, *bias=False*, *max_position_embeddings=2048*, *rope_theta=10000.0*, *rope_scaling=None*, *bos_token_id=11*, *eos_token_id=11*, ***kwargs*)

Falcon is a class of causal decoder-only models built by TII. The largest Falcon checkpoints have been trained on >=1T tokens of text, with a particular emphasis on the RefinedWeb corpus. They are made available under the Apache 2.0 license.

Falcon's architecture is modern and optimized for inference, with multi-query attention and support for efficient attention variants like *FlashAttention*. Both 'base' models trained only as causal language models as well as 'instruct' models that have received further fine-tuning are available.

Falcon models are (as of 2023) some of the largest and most powerful open-source language models, and consistently rank highly in the OpenLLM leaderboard.

Args:

vocab_size (*int*, *optional*, defaults to 65024):
Vocabulary size of the Falcon model. Defines the number of different tokens that can be represented by the *inputs_ids* passed when calling `FalconModel`

hidden_size (*int*, *optional*, defaults to 4544):
Dimension of the hidden representations.

num_hidden_layers (*int*, *optional*, defaults to 32):
Number of hidden layers in the Transformer decoder.

num_attention_heads (*int*, *optional*, defaults to 71):
Number of attention heads for each attention layer in the Transformer encoder.

layer_norm_epsilon (*float*, *optional*, defaults to 1e-05):
The epsilon used by the layer normalization layers.

initializer_range (*float*, *optional*, defaults to 0.02):
The standard deviation of the truncated_normal_initializer for initializing all weight matrices.

use_cache (*bool*, *optional*, defaults to *True*):
Whether the model should return the last key/values attentions (not used by all models). Only relevant if *config.is_decoder=True*.

hidden_dropout (*float*, *optional*, defaults to 0.0):
The dropout probability for MLP layers.

attention_dropout (*float*, *optional*, defaults to 0.0):
The dropout probability for attention layers.

num_kv_heads (*int*, *optional*):
Number of key-value heads to use per attention layer. If unset, defaults to the same value as *num_attention_heads*.

alibi (*bool*, *optional*, defaults to *False*):
Whether to use ALiBi positional biases during self-attention.

new_decoder_architecture (*bool*, *optional*, defaults to *False*):
Whether to use the new (Falcon-40B) decoder architecture. If *True*, the *multi_query* and *parallel_attn* arguments are ignored, as the new decoder always uses parallel attention.

multi_query (*bool*, *optional*, defaults to *True*):
Whether to use multi-query attention in the decoder. Ignored when *new_decoder_architecture* is *True*.

parallel_attn (*bool*, *optional*, defaults to *True*):
Whether to compute attention in parallel with the feedforward layer. If False, they are consecutive instead, as in the original Transformer architecture. Ignored when *new_decoder_architecture* is *True*.

bias (*bool*, *optional*, defaults to *False*):
Whether to use bias on Linear layers.

max_position_embeddings (*int*, *optional*, defaults to 2048):
The maximum sequence length that this model might ever be used with, when *alibi* is *False*. Pretrained Falcon models with RoPE support up to 2048 tokens.

rope_theta (*float*, *optional*, defaults to 10000.0):
The base period of the RoPE embeddings.

rope_scaling (*Dict*, *optional*):
Dictionary containing the scaling configuration for the RoPE embeddings. Currently supports two scaling strategies: linear and dynamic. Their scaling factor must be a float greater than 1. The expected format is *{"type": strategy name, "factor": scaling factor}*. When using this flag, don't update

max_position_embeddings to the expected new maximum. See the following thread for more information on how these scaling strategies behave: https://www.reddit.com/r/LocalLLaMA/comments/14mrgpr/dynamically_scaled_rope_further_increases/. This is an experimental feature, subject to breaking API changes in future versions.

bos_token_id (*int*, *optional*, defaults to 11):
The id of the "beginning-of-sequence" token.

eos_token_id (*int*, *optional*, defaults to 11):
The id of the "end-of-sequence" token.

class transformers.models.flaubert.configuration_flaubert.**FlaubertConfig**(*pre_norm=False*, *layerdrop=0.0*, *vocab_size=30145*, *emb_dim=2048*, *n_layers=12*, *n_heads=16*, *dropout=0.1*, *attention_dropout=0.1*, *gelu_activation=True*, *sinusoidal_embeddings=False*, *causal=False*, *asm=False*, *n_langs=1*, *use_lang_emb=True*, *max_position_embeddings=512*, *embed_init_std=0.02209708691207961*, *layer_norm_eps=1e-12*, *init_std=0.02*, *bos_index=0*, *eos_index=1*, *pad_index=2*, *unk_index=3*, *mask_index=5*, *is_encoder=True*, *summary_type='first'*, *summary_use_proj=True*, *summary_activation=None*, *summary_proj_to_labels=True*, *summary_first_dropout=0.1*, *start_n_top=5*, *end_n_top=5*, *mask_token_id=0*, *lang_id=0*, *pad_token_id=2*, *bos_token_id=0*, ***kwargs*)

The FlauBERT model was proposed in the paper FlauBERT: Unsupervised Language Model Pre-training for French by

Hang Le et al. It's a transformer model pretrained using a masked language modeling (MLM) objective (like BERT).

The abstract from the paper is the following:

Language models have become a key step to achieve state-of-the art results in many different Natural Language Processing (NLP) tasks. Leveraging the huge amount of unlabeled texts nowadays available, they provide an efficient way to pre-train continuous word representations that can be fine-tuned for a downstream task, along with their contextualization at the sentence level. This has been widely demonstrated for English using contextualized representations (Dai and Le, 2015; Peters et al., 2018; Howard and Ruder, 2018; Radford et al., 2018; Devlin et al., 2019; Yang et al., 2019b). In this paper, we introduce and share FlauBERT, a model learned on a very large and heterogeneous French corpus. Models of different sizes are trained using the new CNRS (French National Centre for Scientific Research) Jean Zay supercomputer. We apply our French language models to diverse NLP tasks (text classification, paraphrasing, natural language inference, parsing, word sense disambiguation) and show that most of the time they outperform other pretraining approaches. Different versions of FlauBERT as well as a unified evaluation protocol for the downstream tasks, called FLUE (French Language Understanding Evaluation), are shared to the research community for further reproducible experiments in French NLP.

This model was contributed by formiel. The original code can be found here.

Tips: - Like RoBERTa, without the sentence ordering prediction (so just trained on the MLM objective).

Args:

pre_norm (*bool*, *optional*, defaults to *False*):
Whether to apply the layer normalization before or after the feed forward layer following the attention in each layer (Vaswani et al., Tensor2Tensor for Neural Machine Translation. 2018)

layerdrop (*float*, *optional*, defaults to 0.0):
Probability to drop layers during training (Fan et al., Reducing Transformer Depth on Demand with Structured Dropout. ICLR 2020)

vocab_size (*int*, *optional*, defaults to 30145):
Vocabulary size of the FlauBERT model. Defines the number of different tokens that can be represented by the *inputs_ids* passed when calling `FlaubertModel` or `TFFlaubertModel`.

emb_dim (*int*, *optional*, defaults to 2048):
Dimensionality of the encoder layers and the pooler layer.

n_layer (*int*, *optional*, defaults to 12):
Number of hidden layers in the Transformer encoder.

n_head (*int*, *optional*, defaults to 16):
Number of attention heads for each attention layer in the Transformer encoder.

dropout (*float*, *optional*, defaults to 0.1):
The dropout probability for all fully connected layers in the embeddings, encoder, and pooler.

attention_dropout (*float*, *optional*, defaults to 0.1):
The dropout probability for the attention mechanism

gelu_activation (*bool*, *optional*, defaults to *True*):
Whether or not to use a *gelu* activation instead of *relu*.

sinusoidal_embeddings (*bool*, *optional*, defaults to *False*):
Whether or not to use sinusoidal positional embeddings instead of absolute positional embeddings.

causal (*bool*, *optional*, defaults to *False*):
Whether or not the model should behave in a causal manner. Causal models use a triangular attention mask in order to only attend to the left-side context instead if a bidirectional context.

asm (*bool*, *optional*, defaults to *False*):
Whether or not to use an adaptive log softmax projection layer instead of a linear layer for the prediction layer.

n_langs (*int*, *optional*, defaults to 1):
The number of languages the model handles. Set to 1 for monolingual models.

use_lang_emb (*bool*, *optional*, defaults to *True*)
Whether to use language embeddings. Some models use additional language embeddings, see the multilingual models page for information on how to use them.

max_position_embeddings (*int*, *optional*, defaults to 512):
The maximum sequence length that this model might ever be used with. Typically set this to something large just in case (e.g., 512 or 1024 or 2048).

embed_init_std (*float*, *optional*, defaults to 2048^-0.5):
The standard deviation of the truncated_normal_initializer for initializing the embedding matrices.

init_std (*int*, *optional*, defaults to 50257):
The standard deviation of the truncated_normal_initializer for initializing all weight matrices except the embedding matrices.

layer_norm_eps (*float*, *optional*, defaults to 1e-12):
The epsilon used by the layer normalization layers.

bos_index (*int*, *optional*, defaults to 0):
The index of the beginning of sentence token in the vocabulary.

eos_index (*int*, *optional*, defaults to 1):
The index of the end of sentence token in the vocabulary.

pad_index (*int*, *optional*, defaults to 2):
The index of the padding token in the vocabulary.

unk_index (*int*, *optional*, defaults to 3):
The index of the unknown token in the vocabulary.

mask_index (*int*, *optional*, defaults to 5):
The index of the masking token in the vocabulary.

is_encoder(*bool*, *optional*, defaults to *True*):
Whether or not the initialized model should be a transformer encoder or decoder as seen in Vaswani et al.

summary_type (*string*, *optional*, defaults to "first"):
Argument used when doing sequence summary. Used in the sequence classification and multiple choice models.

Has to be one of the following options:

- *"last"*: Take the last token hidden state (like XLNet).
- *"first"*: Take the first token hidden state (like BERT).
- *"mean"*: Take the mean of all tokens hidden states.
- *"cls_index"*: Supply a Tensor of classification token position (like GPT/GPT-2).
- *"attn"*: Not implemented now, use multi-head attention.

summary_use_proj (*bool*, *optional*, defaults to *True*):
Argument used when doing sequence summary. Used in the sequence classification and multiple choice models.

Whether or not to add a projection after the vector extraction.

summary_activation (*str*, *optional*):
Argument used when doing sequence summary. Used in the sequence classification and multiple choice models.

Pass *"tanh"* for a tanh activation to the output, any other value will result in no activation.

summary_proj_to_labels (*bool*, *optional*, defaults to *True*):
Used in the sequence classification and multiple choice models.

Whether the projection outputs should have *config.num_labels* or *config.hidden_size* classes.

summary_first_dropout (*float*, *optional*, defaults to 0.1):
Used in the sequence classification and multiple choice models.

The dropout ratio to be used after the projection and activation.

start_n_top (*int*, *optional*, defaults to 5):
Used in the SQuAD evaluation script.

end_n_top (*int*, *optional*, defaults to 5):
Used in the SQuAD evaluation script.

mask_token_id (*int*, *optional*, defaults to 0):
Model agnostic parameter to identify masked tokens when generating text in an MLM context.

lang_id (*int*, *optional*, defaults to 1):
The ID of the language used by the model. This parameter is used when generating text in a given language.

`class transformers.models.fnet.configuration_fnet.FNetConfig`(*vocab_size=32000*, *hidden_size=768*, *num_hidden_layers=12*, *intermediate_size=3072*, *hidden_act='gelu_new'*, *hidden_dropout_prob=0.1*, *max_position_embeddings=512*, *type_vocab_size=4*, *initializer_range=0.02*, *layer_norm_eps=1e-12*, *use_tpu_fourier_optimizations=False*, *tpu_short_seq_length=512*, *pad_token_id=3*, *bos_token_id=1*, *eos_token_id=2*, ***kwargs*)

The FNet model was proposed in FNet: Mixing Tokens with Fourier Transforms by James Lee-Thorp, Joshua Ainslie, Ilya Eckstein, Santiago Ontanon. The model replaces the self-attention layer in a BERT model with a fourier transform which returns only the real parts of the transform. The model is significantly faster than the BERT model because it has fewer parameters and is more memory efficient. The model achieves about 92-97% accuracy of BERT counterparts on GLUE benchmark, and trains much faster than the BERT model. The abstract from the paper is the following:

We show that Transformer encoder architectures can be sped up, with limited accuracy costs, by replacing the self-attention sublayers with simple linear transformations that "mix" input tokens. These linear mixers, along with standard nonlinearities in feed-forward layers, prove competent at modeling semantic relationships in several text classification tasks. Most surprisingly, we find that replacing the self-attention sublayer in a Transformer encoder with a standard, unparameterized Fourier Transform achieves 92-97% of the accuracy of BERT counterparts on the GLUE benchmark, but trains 80% faster on GPUs and 70% faster on TPUs at standard 512 input lengths. At longer input lengths, our FNet model is significantly faster: when compared to the "efficient" Transformers on the Long Range Arena benchmark, FNet matches the accuracy of the most accurate models, while outpacing the fastest models across all sequence lengths on GPUs (and across relatively shorter lengths on TPUs). Finally, FNet has a light memory footprint and is particularly efficient at smaller model sizes; for a fixed speed and accuracy budget, small FNet models outperform Transformer counterparts.

Tips on usage:

- The model was trained without an attention mask as it is based on Fourier Transform. The model was trained with maximum sequence length 512 which includes pad tokens. Hence, it is highly recommended to use the same maximum sequence length for fine-tuning and inference.

This model was contributed by gchhablani. The original code can be found here.

Args:

vocab_size (*int*, *optional*, defaults to 32000):
Vocabulary size of the FNet model. Defines the number of different tokens that can be represented by the *inputs_ids* passed when calling `FNetModel` or `TFFNetModel`.

hidden_size (*int*, *optional*, defaults to 768):
Dimension of the encoder layers and the pooler layer.

num_hidden_layers (*int*, *optional*, defaults to 12):
Number of hidden layers in the Transformer encoder.

intermediate_size (*int*, *optional*, defaults to 3072):
Dimension of the "intermediate" (i.e., feed-forward) layer in the Transformer encoder.

hidden_act (*str* or *function*, *optional*, defaults to *"gelu_new"*):
The non-linear activation function (function or string) in the encoder and pooler. If string, *"gelu"*, *"relu"*, *"selu"* and *"gelu_new"* are supported.

hidden_dropout_prob (*float*, *optional*, defaults to 0.1):
The dropout probability for all fully connected layers in the embeddings, encoder, and pooler.

max_position_embeddings (*int*, *optional*, defaults to 512):
The maximum sequence length that this model might ever be used with. Typically set this to something large just in case (e.g., 512 or 1024 or 2048).

type_vocab_size (*int*, *optional*, defaults to 4):
The vocabulary size of the *token_type_ids* passed when calling `FNetModel` or `TFFNetModel`.

initializer_range (*float*, *optional*, defaults to 0.02):
The standard deviation of the truncated_normal_initializer for initializing all weight matrices.

layer_norm_eps (*float*, *optional*, defaults to 1e-12):
The epsilon used by the layer normalization layers.

use_tpu_fourier_optimizations (*bool*, *optional*, defaults to *False*):
Determines whether to use TPU optimized FFTs. If *True*, the model will favor axis-wise FFTs transforms. Set to *False* for GPU/CPU hardware, in which case n-dimensional FFTs are used.

tpu_short_seq_length (*int*, *optional*, defaults to 512):
The sequence length that is expected by the model when using TPUs. This will be used to initialize the DFT matrix only when *use_tpu_fourier_optimizations* is set to *True* and the input sequence is shorter than or equal to 4096 tokens.

class transformers.models.gemma.configuration_gemma.**GemmaConfig**(*vocab_size=256000*, *hidden_size=3072*, *intermediate_size=24576*, *num_hidden_layers=28*, *num_attention_heads=16*, *num_key_value_heads=16*, *head_dim=256*, *hidden_act='gelu_pytorch_tanh'*, *hidden_activation=None*, *max_position_embeddings=8192*, *initializer_range=0.02*, *rms_norm_eps=1e-06*, *use_cache=True*, *pad_token_id=0*, *eos_token_id=1*, *bos_token_id=2*, *tie_word_embeddings=True*, *rope_theta=10000.0*, *attention_bias=False*, *attention_dropout=0.0*, ***kwargs*)

The Gemma model was proposed in Gemma: Open Models Based on Gemini Technology and Research by Gemma Team, Google. Gemma models are trained on 6T tokens, and released with 2 versions, 2b and 7b.

The abstract from the paper is the following:

This work introduces Gemma, a new family of open language models demonstrating strong performance across academic benchmarks for language understanding, reasoning, and safety. We release two sizes of models (2 billion and 7 billion parameters), and provide both pretrained and fine-tuned checkpoints. Gemma outperforms similarly sized open models on 11 out of 18 text-based tasks, and we present comprehensive evaluations of safety and responsibility aspects of the models, alongside a detailed description of our model development. We believe the responsible release of LLMs is critical for improving the safety of frontier models, and for enabling the next wave of LLM innovations

Tips:

- The original checkpoints can be converted using the conversion script *src/transformers/models/gemma/convert_gemma_weights_to_hf.py*

This model was contributed by Arthur Zucker, Younes Belkada, Sanchit Gandhi, Pedro Cuenca.

Args:

vocab_size (*int*, *optional*, defaults to 256000):
Vocabulary size of the Gemma model. Defines the number of different tokens that can be represented by the *inputs_ids* passed when calling `GemmaModel`

hidden_size (*int*, *optional*, defaults to 3072):
Dimension of the hidden representations.

intermediate_size (*int*, *optional*, defaults to 24576):
Dimension of the MLP representations.

num_hidden_layers (*int*, *optional*, defaults to 28):
Number of hidden layers in the Transformer decoder.

num_attention_heads (*int*, *optional*, defaults to 16):
Number of attention heads for each attention layer in the Transformer decoder.

num_key_value_heads (*int*, *optional*, defaults to 16):
This is the number of key_value heads that should be used to implement Grouped Query Attention. If *num_key_value_heads=num_attention_heads*, the model will use Multi Head Attention (MHA), if num_key_value_heads=1 the model will use Multi Query Attention (MQA) otherwise GQA is used. When converting a multi-head checkpoint to a GQA checkpoint, each group key and value head should be constructed by meanpooling all the original heads within that group. For more details checkout `this paper. If it is not specified, will default to *num_attention_heads*.

head_dim (*int*, *optional*, defaults to 256):
The attention head dimension.

hidden_act (*str* or *function*, *optional*, defaults to *"gelu_pytorch_tanh"*):
The legacy activation function. It is overwritten by the *hidden_activation*.

hidden_activation (*str* or *function*, *optional*):
The non-linear activation function (function or string) in the decoder. Will default to *"gelu_pytorch_tanh"* if not specified. *"gelu_pytorch_tanh"* uses an approximation of the *"gelu"* activation function.

max_position_embeddings (*int*, *optional*, defaults to 8192):
The maximum sequence length that this model might ever be used with.

initializer_range (*float*, *optional*, defaults to 0.02):
The standard deviation of the truncated_normal_initializer for initializing all weight matrices.

rms_norm_eps (*float*, *optional*, defaults to 1e-06):
The epsilon used by the rms normalization layers.

use_cache (*bool*, *optional*, defaults to *True*):
Whether or not the model should return the last key/values attentions (not used by all models). Only relevant if *config.is_decoder=True*.

pad_token_id (*int*, *optional*, defaults to 0):
Padding token id.

eos_token_id (*int*, *optional*, defaults to 1):
End of stream token id.

bos_token_id (*int*, *optional*, defaults to 2):
Beginning of stream token id.

tie_word_embeddings (*bool*, *optional*, defaults to *True*):
Whether to tie weight embeddings

rope_theta (*float*, *optional*, defaults to 10000.0):
The base period of the RoPE embeddings.

attention_bias (*bool*, defaults to *False*, *optional*, defaults to *False*):
Whether to use a bias in the query, key, value and output projection layers during self-attention.

attention_dropout (*float*, *optional*, defaults to 0.0):
The dropout ratio for the attention probabilities.

```
>>> from transformers import GemmaModel, GemmaConfig
```

```
>>> # Initializing a Gemma gemma-7b style configuration
>>> configuration = GemmaConfig()
```

```
>>> # Initializing a model from the gemma-7b style configuration
>>> model = GemmaModel(configuration)
```

```
>>> # Accessing the model configuration
>>> configuration = model.config
```

class transformers.models.git.configuration_git.**GitConfig**(*vision_config=None*, *vocab_size=30522*, *hidden_size=768*, *num_hidden_layers=6*, *num_attention_heads=12*, *intermediate_size=3072*, *hidden_act='gelu'*, *hidden_dropout_prob=0.1*, *attention_probs_dropout_prob=0.1*, *max_position_embeddings=1024*, *initializer_range=0.02*, *layer_norm_eps=1e-12*, *pad_token_id=0*, *position_embedding_type='absolute'*, *use_cache=True*, *tie_word_embeddings=False*, *bos_token_id=101*, *eos_token_id=102*, *num_image_with_embedding=None*, ***kwargs*)

The GIT model was proposed in GIT: A Generative Image-to-text Transformer for Vision and Language by Jianfeng Wang, Zhengyuan Yang, Xiaowei Hu, Linjie Li, Kevin Lin, Zhe Gan, Zicheng Liu, Ce Liu, Lijuan Wang. GIT is a decoder-only Transformer that leverages CLIP's vision encoder to condition the model on vision inputs besides text. The model obtains state-of-the-art results on image captioning and visual question answering benchmarks.

The abstract from the paper is the following:

In this paper, we design and train a Generative Image-to-text Transformer, GIT, to unify vision-language tasks such as image/video captioning and question answering. While generative models provide a consistent network architecture between pre-training and fine-tuning, existing work typically contains complex structures (uni/multi-modal encoder/decoder) and depends on external modules such as object detectors/taggers and optical character recognition (OCR). In GIT, we simplify the architecture as one image encoder and one text decoder under a single language modeling task. We also scale up the pre-training data and the model size to boost the model performance. Without bells and whistles, our GIT establishes new state of the arts on 12 challenging benchmarks with a large margin. For instance, our model surpasses the human performance for the first time on TextCaps (138.2 vs. 125.5 in CIDEr). Furthermore, we present a new scheme of generation-based image classification and scene text recognition, achieving decent performance on standard benchmarks.

Tips:

- GIT is implemented in a very similar way to GPT-2, the only difference being that the model is also conditioned on *pixel_values*.
- One can use `GitProcessor` to prepare images for the model, and the *generate* method for autoregressive generation.

<small> GIT architecture. Taken from the original paper. </small>

This model was contributed by nielsr. The original code can be found here.

Args:

vision_config (*dict*, *optional*):
Dictionary of configuration options used to initialize `GitVisionConfig`.

vocab_size (*int*, *optional*, defaults to 30522):
Vocabulary size of the GIT model. Defines the number of different tokens that can be represented by the *inputs_ids* passed when calling `GitModel`.

hidden_size (*int*, *optional*, defaults to 768):
Dimensionality of the encoder layers and the pooler layer.

num_hidden_layers (*int*, *optional*, defaults to 6):
Number of hidden layers in the Transformer encoder.

num_attention_heads (*int*, *optional*, defaults to 12):
Number of attention heads for each attention layer in the Transformer encoder.

intermediate_size (*int*, *optional*, defaults to 3072):
Dimensionality of the "intermediate" (often named feed-forward) layer in the Transformer encoder.

hidden_act (*str* or *Callable*, *optional*, defaults to *"gelu"*):
The non-linear activation function (function or string) in the encoder and pooler. If string, *"gelu"*, *"relu"*, *"silu"* and *"gelu_new"* are supported.

hidden_dropout_prob (*float*, *optional*, defaults to 0.1):
The dropout probability for all fully connected layers in the embeddings, encoder, and pooler.

attention_probs_dropout_prob (*float*, *optional*, defaults to 0.1):
The dropout ratio for the attention probabilities.

max_position_embeddings (*int*, *optional*, defaults to 1024):
The maximum sequence length that this model might ever be used with. Typically set this to something large just in case (e.g., 512 or 1024 or 2048).

initializer_range (*float*, *optional*, defaults to 0.02):
The standard deviation of the truncated_normal_initializer for initializing all weight matrices.

layer_norm_eps (*float*, *optional*, defaults to 1e-12):
The epsilon used by the layer normalization layers.

position_embedding_type (*str*, *optional*, defaults to *"absolute"*):
Type of position embedding. Choose one of *"absolute"*, *"relative_key"*, *"relative_key_query"*. For positional embeddings use *"absolute"*. For more information on *"relative_key"*, please refer to Self-Attention with Relative Position Representations (Shaw et al.). For more information on *"relative_key_query"*, please refer to *Method 4* in Improve Transformer Models with Better Relative Position Embeddings (Huang et al.).

use_cache (*bool*, *optional*, defaults to *True*):
Whether or not the model should return the last key/values attentions (not used by all models).

num_image_with_embedding (*int*, *optional*):
The number of temporal embeddings to add, in case the model is used for video captioning/VQA.

class transformers.models.gpt2.configuration_gpt2.**GPT2Config**(*vocab_size=50257*, *n_positions=1024*, *n_embd=768*, *n_layer=12*, *n_head=12*, *n_inner=None*, *activation_function='gelu_new'*, *resid_pdrop=0.1*, *embd_pdrop=0.1*, *attn_pdrop=0.1*, *layer_norm_epsilon=1e-05*, *initializer_range=0.02*, *summary_type='cls_index'*, *summary_use_proj=True*, *summary_activation=None*, *summary_proj_to_labels=True*, *summary_first_dropout=0.1*, *scale_attn_weights=True*, *use_cache=True*, *bos_token_id=50256*, *eos_token_id=50256*, *scale_attn_by_inverse_layer_idx=False*, *reorder_and_upcast_attn=False*, ***kwargs*)

The GPT-Sw3 model was first proposed in Lessons Learned from GPT-SW3: Building the First Large-Scale Generative Language Model for Swedish by Ariel Ekgren, Amaru Cuba Gyllensten, Evangelia Gogoulou, Alice Heiman, Severine Verlinden, Joey Öhman, Fredrik Carlsson, Magnus Sahlgren.

Since that first paper the authors have extended their work and trained new models on their new 1.2TB corpora named The Nordic Pile.

GPT-Sw3 is a collection of large decoder-only pretrained transformer language models that were developed by AI Sweden in collaboration with RISE and the WASP WARA for Media and Language. GPT-Sw3 has been trained on a dataset containing 320B tokens in Swedish, Norwegian, Danish, Icelandic, English, and programming code. The model was pretrained using a causal language modeling (CLM) objective utilizing the NeMo Megatron GPT implementation.

This model was contributed by AI Sweden.

The implementation uses the GPT2Model coupled with our *GPTSw3Tokenizer*. This means that *AutoTokenizer* and *AutoModelForCausalLM* map to our tokenizer implementation and the corresponding GPT2 model implementation respectively. *Note that sentencepiece is required to use our tokenizer and can be installed with: pip install transformers[sentencepiece]* or *pip install sentencepiece*

class transformers.models.gpt2.configuration_gpt2.**GPT2Config**(*vocab_size=50257*, *n_positions=1024*, *n_embd=768*, *n_layer=12*, *n_head=12*, *n_inner=None*, *activation_function='gelu_new'*, *resid_pdrop=0.1*, *embd_pdrop=0.1*, *attn_pdrop=0.1*, *layer_norm_epsilon=1e-05*, *initializer_range=0.02*, *summary_type='cls_index'*, *summary_use_proj=True*, *summary_activation=None*, *summary_proj_to_labels=True*, *summary_first_dropout=0.1*, *scale_attn_weights=True*, *use_cache=True*, *bos_token_id=50256*, *eos_token_id=50256*, *scale_attn_by_inverse_layer_idx=False*, *reorder_and_upcast_attn=False*, ***kwargs*)

OpenAI GPT-2 model was proposed in Language Models are Unsupervised Multitask Learners by Alec Radford, Jeffrey Wu, Rewon Child, David Luan, Dario Amodei and Ilya Sutskever from OpenAI. It's a causal (unidirectional) transformer pretrained using language modeling on a very large corpus of ~40 GB of text data.

The abstract from the paper is the following:

GPT-2 is a large transformer-based language model with 1.5 billion parameters, trained on a dataset`1] of 8 million web pages. GPT-2 is trained with a simple objective: predict the next word, given all of the previous words within some text. The diversity of the dataset causes this simple goal to contain naturally occurring demonstrations of many tasks across diverse domains. GPT-2 is a direct scale-up of GPT, with more than 10X the parameters and trained on more than 10X the amount of data.

Tips:

- GPT-2 is a model with absolute position embeddings so it's usually advised to pad the inputs on the right rather than the left.
- GPT-2 was trained with a causal language modeling (CLM) objective and is therefore powerful at predicting the next token in a sequence. Leveraging this feature allows GPT-2 to generate syntactically coherent text as it can be observed in the *run_generation.py* example script.
- The model can take the *past_key_values* (for PyTorch) or *past* (for TF) as input, which is the previously computed key/value attention pairs. Using this (*past_key_values* or *past*) value prevents the model from re-computing pre-computed values in the context of text generation. For PyTorch, see *past_key_values* argument of the `GPT2Model.forward` method, or for TF the *past* argument of the `TFGPT2Model.call` method for more information on its usage.
- Enabling the *scale_attn_by_inverse_layer_idx* and *reorder_and_upcast_attn* flags will apply the training stability improvements from [Mistral <https://github.com/stanford-crfm/mistral/>`__ (for PyTorch only).

Write With Transformer is a webapp created and hosted by Hugging Face showcasing the generative capabilities of several models. GPT-2 is one of them and is available in five different sizes: small, medium, large, xl and a distilled version of the small checkpoint: *distilgpt-2*.

This model was contributed by thomwolf. The original code can be found here.

Args:

vocab_size (*int*, *optional*, defaults to 50257):
Vocabulary size of the GPT-2 model. Defines the number of different tokens that can be represented by the *inputs_ids* passed when calling `GPT2Model` or `TFGPT2Model`.

n_positions (*int*, *optional*, defaults to 1024):
The maximum sequence length that this model might ever be used with. Typically set this to something large just in case (e.g., 512 or 1024 or 2048).

n_embd (*int*, *optional*, defaults to 768):
Dimensionality of the embeddings and hidden states.

n_layer (*int*, *optional*, defaults to 12):
Number of hidden layers in the Transformer encoder.

n_head (*int*, *optional*, defaults to 12):
Number of attention heads for each attention layer in the Transformer encoder.

n_inner (*int*, *optional*):
Dimensionality of the inner feed-forward layers. *None* will set it to 4 times n_embd

activation_function (*str*, *optional*, defaults to *"gelu_new"*):
Activation function, to be selected in the list *["relu", "silu", "gelu", "tanh", "gelu_new"]*.

resid_pdrop (*float*, *optional*, defaults to 0.1):
The dropout probability for all fully connected layers in the embeddings, encoder, and pooler.

embd_pdrop (*float*, *optional*, defaults to 0.1):
The dropout ratio for the embeddings.

attn_pdrop (*float*, *optional*, defaults to 0.1):
The dropout ratio for the attention.

layer_norm_epsilon (*float*, *optional*, defaults to 1e-05):
The epsilon to use in the layer normalization layers.

initializer_range (*float*, *optional*, defaults to 0.02):
The standard deviation of the truncated_normal_initializer for initializing all weight matrices.

summary_type (*string*, *optional*, defaults to *"cls_index"*):
Argument used when doing sequence summary, used in the models `GPT2DoubleHeadsModel` and `TFGPT2DoubleHeadsModel`.

Has to be one of the following options:

- *"last"*: Take the last token hidden state (like XLNet).
- *"first"*: Take the first token hidden state (like BERT).
- *"mean"*: Take the mean of all tokens hidden states.
- *"cls_index"*: Supply a Tensor of classification token position (like GPT/GPT-2).
- *"attn"*: Not implemented now, use multi-head attention.

summary_use_proj (*bool*, *optional*, defaults to *True*):
Argument used when doing sequence summary, used in the models `GPT2DoubleHeadsModel` and `TFGPT2DoubleHeadsModel`.

Whether or not to add a projection after the vector extraction.

summary_activation (*str*, *optional*):
Argument used when doing sequence summary. Used in for the multiple choice head in `GPT2DoubleHeadsModel`.

Pass *"tanh"* for a tanh activation to the output, any other value will result in no activation.

summary_proj_to_labels (*bool*, *optional*, defaults to *True*):
Argument used when doing sequence summary, used in the models `GPT2DoubleHeadsModel` and `TFGPT2DoubleHeadsModel`.

Whether the projection outputs should have *config.num_labels* or *config.hidden_size* classes.

summary_first_dropout (*float*, *optional*, defaults to 0.1):
Argument used when doing sequence summary, used in the models `GPT2DoubleHeadsModel` and `TFGPT2DoubleHeadsModel`.

The dropout ratio to be used after the projection and activation.

scale_attn_weights (*bool*, *optional*, defaults to *True*):
Scale attention weights by dividing by sqrt(hidden_size)..

use_cache (*bool*, *optional*, defaults to *True*):
Whether or not the model should return the last key/values attentions (not used by all models).

bos_token_id (*int*, *optional*, defaults to 50256):
Id of the beginning of sentence token in the vocabulary.

eos_token_id (*int*, *optional*, defaults to 50256):
Id of the end of sentence token in the vocabulary.

scale_attn_by_inverse_layer_idx (*bool*, *optional*, defaults to *False*):
Whether to additionally scale attention weights by *1 / layer_idx + 1*.

reorder_and_upcast_attn (*bool*, *optional*, defaults to *False*):
Whether to scale keys (K) prior to computing attention (dot-product) and upcast attention dot-product/softmax to float() when training with mixed precision.

`class` transformers.models.gpt_bigcode.configuration_gpt_bigcode.**GPTBigCodeConfig**(*vocab_size=50257*, *n_positions=1024*, *n_embd=768*, *n_layer=12*, *n_head=12*, *n_inner=None*, *activation_function='gelu_pytorch_*, *resid_pdrop=0.1*, *embd_pdrop=0.1*, *attn_pdrop=0.1*, *layer_norm_epsilon=1e-05*, *initializer_range=0.02*, *scale_attn_weights=True*, *use_cache=True*, *bos_token_id=50256*, *eos_token_id=50256*, *attention_softmax_in_fp32=True*, *scale_attention_softmax_in_f*, *multi_query=True*, ***kwargs*)

The GPTBigCode model was proposed in SantaCoder: don't reach for the stars! by BigCode. The listed authors are: Loubna Ben Allal, Raymond Li, Denis Kocetkov, Chenghao Mou, Christopher Akiki, Carlos Munoz Ferrandis, Niklas Muennighoff, Mayank Mishra, Alex Gu, Manan Dey, Logesh Kumar Umapathi, Carolyn Jane Anderson, Yangtian

Zi, Joel Lamy Poirier, Hailey Schoelkopf, Sergey Troshin, Dmitry Abulkhanov, Manuel Romero, Michael Lappert, Francesco De Toni, Bernardo García del Río, Qian Liu, Shamik Bose, Urvashi Bhattacharyya, Terry Yue Zhuo, Ian Yu, Paulo Villegas, Marco Zocca, Sourab Mangrulkar, David Lansky, Huu Nguyen, Danish Contractor, Luis Villa, Jia Li, Dzmitry Bahdanau, Yacine Jernite, Sean Hughes, Daniel Fried, Arjun Guha, Harm de Vries, Leandro von Werra.

The abstract from the paper is the following:uery

The BigCode project is an open-scientific collaboration working on the responsible development of large language models for code. This tech report describes the progress of the collaboration until December 2022, outlining the current state of the Personally Identifiable Information (PII) redaction pipeline, the experiments conducted to de-risk the model architecture, and the experiments investigating better preprocessing methods for the training data. We train 1.1B parameter models on the Java, JavaScript, and Python subsets of The Stack and evaluate them on the MultiPL-E text-to-code benchmark. We find that more aggressive filtering of near-duplicates can further boost performance and, surprisingly, that selecting files from repositories with 5+ GitHub stars deteriorates performance significantly. Our best model outperforms previous open-source multilingual code generation models (InCoder-6.7B and CodeGen-Multi-2.7B) in both left-to-right generation and infilling on the Java, JavaScript, and Python portions of MultiPL-E, despite being a substantially smaller model. All models are released under an OpenRAIL license at `this https URL. <https://huggingface.co/bigcode> `__

The model is a an optimized GPT2 model with support for Multi-Query Attention.

Args:

vocab_size (*int*, *optional*, defaults to 50257):
Vocabulary size of the GPT-2 model. Defines the number of different tokens that can be represented by the *inputs_ids* passed when calling `GPTBigCodeModel`.

n_positions (*int*, *optional*, defaults to 1024):
The maximum sequence length that this model might ever be used with. Typically set this to something large just in case (e.g., 512 or 1024 or 2048).

n_embd (*int*, *optional*, defaults to 768):
Dimensionality of the embeddings and hidden states.

n_layer (*int*, *optional*, defaults to 12):
Number of hidden layers in the Transformer encoder.

n_head (*int*, *optional*, defaults to 12):
Number of attention heads for each attention layer in the Transformer encoder.

n_inner (*int*, *optional*, defaults to None):
Dimensionality of the inner feed-forward layers. *None* will set it to 4 times n_embd

activation_function (*str*, *optional*, defaults to *"gelu_pytorch_tanh"*):
Activation function, to be selected in the list *["relu", "silu", "gelu", "tanh", "gelu_new", "gelu_pytorch_tanh"]*.

resid_pdrop (*float*, *optional*, defaults to 0.1):
The dropout probability for all fully connected layers in the embeddings, encoder, and pooler.

embd_pdrop (*float*, *optional*, defaults to 0.1):
The dropout ratio for the embeddings.

attn_pdrop (*float*, *optional*, defaults to 0.1):
The dropout ratio for the attention.

layer_norm_epsilon (*float*, *optional*, defaults to 1e-5):
The epsilon to use in the layer normalization layers.

initializer_range (*float*, *optional*, defaults to 0.02):
The standard deviation of the truncated_normal_initializer for initializing all weight matrices.

scale_attn_weights (*bool*, *optional*, defaults to *True*):
Scale attention weights by dividing by sqrt(hidden_size)..

use_cache (*bool*, *optional*, defaults to *True*):
Whether or not the model should return the last key/values attentions (not used by all models).

attention_softmax_in_fp32 (*bool*, *optional*, defaults to *True*):
Whether to call the fused softmax in float32.

scale_attention_softmax_in_fp32 (*bool*, *optional*, defaults to *True*):
Whether to scale the attention softmax in float32.

attention_type (*bool*, *optional*, defaults to *True*):
Whether to use Multi-Query Attion (*True*) or Multi-Head Attention (*False*).

`class transformers.models.gpt_neox.configuration_gpt_neox.GPTNeoXConfig(`*vocab_size=50432*, *hidden_size=6144*, *num_hidden_layers=44*, *num_attention_heads=64*, *intermediate_size=24576*, *hidden_act='gelu'*, *rotary_pct=0.25*, *rotary_emb_base=10000*, *attention_dropout=0.0*, *hidden_dropout=0.0*, *classifier_dropout=0.1*, *max_position_embeddings=2048*, *initializer_range=0.02*, *layer_norm_eps=1e-05*, *use_cache=True*, *bos_token_id=0*, *eos_token_id=2*, *tie_word_embeddings=False*, *use_parallel_residual=True*, *rope_scaling=None*, *attention_bias=True*, ***kwargs*)

We introduce GPT-NeoX-20B, a 20 billion parameter autoregressive language model trained on the Pile, whose weights will be made freely and openly available to the public through a permissive license. It is, to the best of our knowledge, the largest dense autoregressive model that has publicly available weights at the time of submission. In this work, we describe GPT-NeoX-20B's architecture and training and evaluate its performance on a range of language-understanding, mathematics, and knowledge-based tasks. We find that GPT-NeoX-20B is a particularly powerful few-shot reasoner and gains far more in performance when evaluated five-shot than similarly sized GPT-3 and FairSeq models. We open-source the training and evaluation code, as well as the model weights, at https://github.com/EleutherAI/gpt-neox.

Development of the model was led by Sid Black, Stella Biderman and Eric Hallahan, and the model was trained with generous the support of CoreWeave.

GPT-NeoX-20B was trained with fp16, thus it is recommended to initialize the model as follows:

model = GPTNeoXForCausalLM.from_pretrained("EleutherAI/gpt-neox-20b").half().cuda()

GPT-NeoX-20B also has a different tokenizer from the one used in GPT-J-6B and GPT-Neo. The new tokenizer allocates additional tokens to whitespace characters, making the model more suitable for certain tasks like code generation.

#Args:

vocab_size (*int*, *optional*, defaults to 50432):
Vocabulary size of the GPTNeoX model. Defines the number of different tokens that can be represented by the *inputs_ids* passed when calling `GPTNeoXModel`.

hidden_size (*int*, *optional*, defaults to 6144):
Dimension of the encoder layers and the pooler layer.

num_hidden_layers (*int*, *optional*, defaults to 44):
Number of hidden layers in the Transformer encoder.

num_attention_heads (*int*, *optional*, defaults to 64):
Number of attention heads for each attention layer in the Transformer encoder.

intermediate_size (*int*, *optional*, defaults to 24576):
Dimension of the "intermediate" (i.e., feed-forward) layer in the Transformer encoder.

hidden_act (*str* or *function*, *optional*, defaults to *"gelu"*):
The non-linear activation function (function or string) in the encoder and pooler. If string, *"gelu"*, *"relu"*, *"selu"* and *"gelu_new"* are supported.

rotary_pct (*float*, *optional*, defaults to 0.25):
percentage of hidden dimensions to allocate to rotary embeddings

rotary_emb_base (*int*, *optional*, defaults to 10000)
base for computing rotary embeddings frequency

attention_dropout (*float*, *optional*, defaults to 0.0):
The dropout ratio probability of the attention score.

hidden_dropout (*float*, *optional*, defaults to 0.0):
The dropout ratio of (1) the word embeddings, (2) the post-attention hidden states, and (3) the post-mlp hidden states.

classifier_dropout (*float*, *optional*, defaults to 0.1):
Argument used when doing token classification, used in the model `GPTNeoXForTokenClassification`.

The dropout ratio for the hidden layer.

max_position_embeddings (*int*, *optional*, defaults to 2048):
The maximum sequence length that this model might ever be used with. Typically set this to something large just in case (e.g., 512 or 1024 or 2048).

initializer_range (*float*, *optional*, defaults to 1e-5):
The standard deviation of the truncated_normal_initializer for initializing all weight matrices.

layer_norm_eps (*float*, *optional*, defaults to 1e-12):
The epsilon used by the layer normalization layers.

use_cache (*bool*, *optional*, defaults to *True*):
Whether or not the model should return the last key/values attentions (not used by all models). Only relevant if *config.is_decoder=True*.

use_parallel_residual (*bool*, *optional*, defaults to *True*):
Whether to use a "parallel" formulation in each Transformer layer, which can provide a slight training speedup at large scales (e.g. 20B).

rope_scaling (*Dict*, *optional*):
Dictionary containing the scaling configuration for the RoPE embeddings. Currently supports two scaling strategies: linear and dynamic. Their scaling factor must be a float greater than 1. The expected format is *{"type": strategy name, "factor": scaling factor}*. When using this flag, don't update

max_position_embeddings to the expected new maximum. See the following thread for more information on how these scaling strategies behave: https://www.reddit.com/r/LocalLLaMA/comments/14mrgpr/dynamically_scaled_rope_further_increases/. This is an experimental feature, subject to breaking API changes in future versions.

attention_bias (*bool*, *optional*, defaults to *True*):
Whether to use a bias in the query, key, value and output projection layers during self-attention.

class transformers.models.gpt_neox_japanese.configuration_gpt_neox_japanese.**GPTNeoXJapaneseConfig**(*vocab_ hid- den_si num_h num_a in- ter- me- di- ate_mu hid- den_ac ro- tary_p ro- tary_e max_p ini- tial- izer_ra layer_ 05*, *use_ca bos_to eos_to at- ten- tion_d hid- den_d **kwa*

We introduce GPT-NeoX-Japanese, which is an autoregressive language model for Japanese, trained on top of https://github.com/EleutherAI/gpt-neox. Japanese is a unique language with its large vocabulary and a combination of hiragana, katakana, and kanji writing scripts. To address this distinct structure of the Japanese language, we use a special sub-word tokenizer. We are very grateful to *tanreinama* for open-sourcing this incredibly helpful tokenizer. Following the recommendations from Google's research on PaLM, we have removed bias parameters from transformer blocks, achieving better model performance. Please refer this article in detail.

Development of the model was led by Shinya Otani, Takayoshi Makabe, Anuj Arora, and Kyo Hattori from ABEJA, Inc.. For more information on this model-building activity, please refer here (ja).

#Args:

vocab_size (*int*, *optional*, defaults to 32000):
Vocabulary size of the GPTNeoXJapanese model. Defines the number of different tokens that can be represented by the *inputs_ids* passed when calling `GPTNeoXJapanese`.

hidden_size (*int*, *optional*, defaults to 2560):

Dimension of the encoder layers and the pooler layer.

num_hidden_layers (*int*, *optional*, defaults to 32):
Number of hidden layers in the Transformer encoder.

num_attention_heads (*int*, *optional*, defaults to 32):
Number of attention heads for each attention layer in the Transformer encoder.

intermediate_multiple_size (*int*, *optional*, defaults to 4):
Dimension of the "intermediate" layer in the Transformer encoder is calculated by hidden_size * intermediate_multiple_size.

hidden_act (*str* or *function*, *optional*, defaults to *"gelu"*):
The non-linear activation function (function or string) in the encoder and pooler.

rotary_pct (*float*, *optional*, defaults to 1.00):
percentage of hidden dimensions to allocate to rotary embeddings

rotary_emb_base (*int*, *optional*, defaults to 10000)
base for computing rotary embeddings frequency

max_position_embeddings (*int*, *optional*, defaults to 2048):
The maximum sequence length that this model might ever be used with.

initializer_range (*float*, *optional*, defaults to 0.02):
The standard deviation of the truncated_normal_initializer for initializing all weight matrices.

layer_norm_eps (*float*, *optional*, defaults to 1e-5):
The epsilon used by the layer normalization layers.

use_cache (*bool*, *optional*, defaults to *True*):
Whether or not the model should return the last key/values attentions (not used by all models). Only relevant if *config.is_decoder=True*.

attention_dropout (*float*, *optional*, defaults to 0.1):
The dropout ratio for the attention.

hidden_dropout (*float*, *optional*, defaults to 0.0):
The dropout ratio for the hidden layer.

`class transformers.models.gptj.configuration_gptj.GPTJConfig(`*vocab_size=50400*, *n_positions=2048*, *n_embd=4096*, *n_layer=28*, *n_head=16*, *rotary_dim=64*, *n_inner=None*, *activation_function='gelu_new'*, *resid_pdrop=0.0*, *embd_pdrop=0.0*, *attn_pdrop=0.0*, *layer_norm_epsilon=1e-05*, *initializer_range=0.02*, *use_cache=True*, *bos_token_id=50256*, *eos_token_id=50256*, *tie_word_embeddings=False*, ***kwargs*`)`

The GPT-J model was released in the kingoflolz/mesh-transformer-jax repository by Ben Wang and Aran Komatsuzaki. It is a GPT-2-like causal language model trained on the Pile dataset.

This model was contributed by Stella Biderman.

Tips:

- To load GPT-J in float32 one would need at least 2x model size RAM: 1x for initial weights and another 1x to load the checkpoint. So for GPT-J it would take at least 48GB RAM to just load the model. To reduce the RAM usage there are a few options. The *torch_dtype* argument can be used to initialize the model in half-precision on a CUDA device only. There is also a fp16 branch which stores the fp16 weights, which could be used to further minimize the RAM usage:

```
>>> from transformers import GPTJForCausalLM
>>> import torch
```

```
>>> device = "cuda"
>>> model = GPTJForCausalLM.from_pretrained(
...     "EleutherAI/gpt-j-6B",
...     revision="float16",
...     torch_dtype=torch.float16,
... ).to(device)
```

- The model should fit on 16GB GPU for inference. For training/fine-tuning it would take much more GPU RAM. Adam optimizer for example makes four copies of the model: model, gradients, average and squared average of the gradients. So it would need at least 4x model size GPU memory, even with mixed precision as gradient updates are in fp32. This is not including the activations and data batches, which would again require some more GPU RAM. So one should explore solutions such as DeepSpeed, to train/fine-tune the model. Another option is to use the original codebase to train/fine-tune the model on TPU and then convert the model to Transformers format for inference. Instructions for that could be found here

- Although the embedding matrix has a size of 50400, only 50257 entries are used by the GPT-2 tokenizer. These extra tokens are added for the sake of efficiency on TPUs. To avoid the mismatch between embedding matrix size and vocab size, the tokenizer for GPT-J contains 143 extra tokens <|*extratoken_1*|>... <|*extratoken_143*|>, so the *vocab_size* of tokenizer also becomes 50400.

#Args:

vocab_size (*int*, *optional*, defaults to 50400):
Vocabulary size of the GPT-J model. Defines the number of different tokens that can be represented by the *inputs_ids* passed when calling `GPTJModel`.

n_positions (*int*, *optional*, defaults to 2048):
The maximum sequence length that this model might ever be used with. Typically set this to something large just in case (e.g., 512 or 1024 or 2048).

n_embd (*int*, *optional*, defaults to 4096):
Dimensionality of the embeddings and hidden states.

n_layer (*int*, *optional*, defaults to 28):
Number of hidden layers in the Transformer encoder.

n_head (*int*, *optional*, defaults to 16):
Number of attention heads for each attention layer in the Transformer encoder.

rotary_dim (*int*, *optional*, defaults to 64):
Number of dimensions in the embedding that Rotary Position Embedding is applied to.

n_inner (*int*, *optional*, defaults to None):
Dimensionality of the inner feed-forward layers. *None* will set it to 4 times n_embd

activation_function (*str*, *optional*, defaults to *"gelu_new"*):
Activation function, to be selected in the list *["relu", "silu", "gelu", "tanh", "gelu_new"]*.

resid_pdrop (*float*, *optional*, defaults to 0.1):
The dropout probability for all fully connected layers in the embeddings, encoder, and pooler.

embd_pdrop (*int*, *optional*, defaults to 0.1):
The dropout ratio for the embeddings.

attn_pdrop (*float*, *optional*, defaults to 0.1):
The dropout ratio for the attention.

layer_norm_epsilon (*float*, *optional*, defaults to 1e-5):
The epsilon to use in the layer normalization layers.

initializer_range (*float*, *optional*, defaults to 0.02):
The standard deviation of the truncated_normal_initializer for initializing all weight matrices.

use_cache (*bool*, *optional*, defaults to *True*):
Whether or not the model should return the last key/values attentions (not used by all models).

`class transformers.models.ibert.configuration_ibert.IBertConfig(`*vocab_size=30522*, *hidden_size=768*, *num_hidden_layers=12*, *num_attention_heads=12*, *intermediate_size=3072*, *hidden_act='gelu'*, *hidden_dropout_prob=0.1*, *attention_probs_dropout_prob=0.1*, *max_position_embeddings=512*, *type_vocab_size=2*, *initializer_range=0.02*, *layer_norm_eps=1e-12*, *pad_token_id=1*, *bos_token_id=0*, *eos_token_id=2*, *position_embedding_type='absolute'*, *quant_mode=False*, *force_dequant='none'*, ***kwargs*)

The I-BERT model was proposed in I-BERT: Integer-only BERT Quantization by Sehoon Kim, Amir Gholami, Zhewei Yao, Michael W. Mahoney and Kurt Keutzer. It's a quantized version of RoBERTa running inference up to four times faster.

The abstract from the paper is the following:

Transformer based models, like BERT and RoBERTa, have achieved state-of-the-art results in many Natural Language Processing tasks. However, their memory footprint, inference latency, and power consumption are prohibitive for efficient inference at the edge, and even at the data center. While quantization can be a viable solution for this, previous work on quantizing Transformer based models use floating-point arithmetic during inference, which cannot efficiently utilize integer-only logical units such as the recent Turing Tensor Cores, or traditional integer-only ARM processors. In this work, we propose I-BERT, a novel quantization scheme for Transformer based models that quantizes the entire inference with integer-only arithmetic. Based on lightweight integer-only approximation methods for nonlinear operations, e.g., GELU, Softmax, and Layer Normalization, I-BERT performs an end-to-end integer-only BERT inference without any floating point calculation. We evaluate our approach on GLUE downstream tasks using RoBERTa-Base/Large. We show that for both cases, I-BERT achieves similar (and slightly higher) accuracy as compared to the full-precision baseline. Furthermore, our preliminary implementation of I-BERT shows a speedup of 2.4 - 4.0x for INT8 inference on a T4 GPU system as compared to FP32 inference. The framework has been developed in PyTorch and has been open-sourced.

This model was contributed by kssteven. The original code can be found here.

Args:

vocab_size (*int*, *optional*, defaults to 30522):
Vocabulary size of the I-BERT model. Defines the number of different tokens that can be represented by the *inputs_ids* passed when calling `IBertModel`

hidden_size (*int*, *optional*, defaults to 768):
Dimensionality of the encoder layers and the pooler layer.

num_hidden_layers (*int*, *optional*, defaults to 12):
Number of hidden layers in the Transformer encoder.

num_attention_heads (*int*, *optional*, defaults to 12):
Number of attention heads for each attention layer in the Transformer encoder.

intermediate_size (*int*, *optional*, defaults to 3072):
Dimensionality of the "intermediate" (often named feed-forward) layer in the Transformer encoder.

hidden_act (*str* or *Callable*, *optional*, defaults to *"gelu"*):
The non-linear activation function (function or string) in the encoder and pooler. If string, *"gelu"*, *"relu"*, *"silu"* and *"gelu_new"* are supported.

hidden_dropout_prob (*float*, *optional*, defaults to 0.1):
The dropout probability for all fully connected layers in the embeddings, encoder, and pooler.

attention_probs_dropout_prob (*float*, *optional*, defaults to 0.1):
The dropout ratio for the attention probabilities.

max_position_embeddings (*int*, *optional*, defaults to 512):
The maximum sequence length that this model might ever be used with. Typically set this to something large just in case (e.g., 512 or 1024 or 2048).

type_vocab_size (*int*, *optional*, defaults to 2):
The vocabulary size of the *token_type_ids* passed when calling `IBertModel`

initializer_range (*float*, *optional*, defaults to 0.02):
The standard deviation of the truncated_normal_initializer for initializing all weight matrices.

layer_norm_eps (*float*, *optional*, defaults to 1e-12):
The epsilon used by the layer normalization layers.

position_embedding_type (*str*, *optional*, defaults to *"absolute"*):
Type of position embedding. Choose one of *"absolute"*, *"relative_key"*, *"relative_key_query"*. For positional embeddings use *"absolute"*. For more information on *"relative_key"*, please refer to Self-Attention with Relative Position Representations (Shaw et al.). For more information on *"relative_key_query"*, please refer to *Method 4* in Improve Transformer Models with Better Relative Position Embeddings (Huang et al.).

quant_mode (*bool*, *optional*, defaults to *False*):
Whether to quantize the model or not.

force_dequant (*str*, *optional*, defaults to *"none"*):
Force dequantize specific nonlinear layer. Dequatized layers are then executed with full precision. *"none"*, *"gelu"*, *"softmax"*, *"layernorm"* and *"nonlinear"* are supported. As deafult, it is set as *"none"*, which does not dequantize any layers. Please specify *"gelu"*, *"softmax"*, or *"layernorm"* to dequantize GELU, Softmax, or LayerNorm, respectively. *"nonlinear"* will dequantize all nonlinear layers, i.e., GELU, Softmax, and LayerNorm.

class transformers.models.imagegpt.configuration_imagegpt.**ImageGPTConfig**(*vocab_size=513*, *n_positions=1024*, *n_embd=512*, *n_layer=24*, *n_head=8*, *n_inner=None*, *activation_function='quick_gelu'*, *resid_pdrop=0.1*, *embd_pdrop=0.1*, *attn_pdrop=0.1*, *layer_norm_epsilon=1e-05*, *initializer_range=0.02*, *scale_attn_weights=True*, *use_cache=True*, *tie_word_embeddings=False*, *scale_attn_by_inverse_layer_idx=False*, *reorder_and_upcast_attn=False*, ***kwargs*)

The ImageGPT model was proposed in Generative Pretraining from Pixels by Mark Chen, Alec Radford, Rewon Child, Jeffrey Wu, Heewoo Jun, David Luan, Ilya Sutskever. ImageGPT (iGPT) is a GPT-2-like model trained to predict the next pixel value, allowing for both unconditional and conditional image generation.

The abstract from the paper is the following:

Inspired by progress in unsupervised representation learning for natural language, we examine whether similar models can learn useful representations for images. We train a sequence Transformer to auto-regressively predict pixels, without incorporating knowledge of the 2D input structure. Despite training on low-resolution ImageNet without labels, we find that a GPT-2 scale model learns strong image representations as measured by linear probing, fine-tuning, and low-data classification. On CIFAR-10, we achieve 96.3% accuracy with a linear probe, outperforming a supervised Wide ResNet, and 99.0% accuracy with full fine-tuning, matching the top supervised pre-trained models. We are also competitive with self-supervised benchmarks on ImageNet when substituting pixels for a VQVAE encoding, achieving 69.0% top-1 accuracy on a linear probe of our features.

<small> Summary of the approach. Taken from the original paper. </small>

This model was contributed by nielsr, based on this issue. The original code can be found here.

Tips:

- ImageGPT is almost exactly the same as GPT-2, with the exception that a different activation function is used (namely "quick gelu"), and the layer normalization layers don't mean center the inputs. ImageGPT also doesn't have tied input- and output embeddings.
- As the time- and memory requirements of the attention mechanism of Transformers scales quadratically in the sequence length, the authors pre-trained ImageGPT on smaller input resolutions, such as 32x32 and 64x64. However, feeding a sequence of 32x32x3=3072 tokens from 0..255 into a Transformer is still prohibitively large. Therefore, the authors applied k-means clustering to the (R,G,B) pixel values with k=512. This way, we only have a 32*32 = 1024-long sequence, but now of integers in the range 0..511. So we are shrinking the sequence length at the cost of a bigger embedding matrix. In other words, the vocabulary size of ImageGPT is

512, + 1 for a special "start of sentence" (SOS) token, used at the beginning of every sequence. One can use `ImageGPTImageProcessor` to prepare images for the model.

- Despite being pre-trained entirely unsupervised (i.e. without the use of any labels), ImageGPT produces fairly performant image features useful for downstream tasks, such as image classification. The authors showed that the features in the middle of the network are the most performant, and can be used as-is to train a linear model (such as a sklearn logistic regression model for example). This is also referred to as "linear probing". Features can be easily obtained by first forwarding the image through the model, then specifying *output_hidden_states=True*, and then average-pool the hidden states at whatever layer you like.
- Alternatively, one can further fine-tune the entire model on a downstream dataset, similar to BERT. For this, you can use `ImageGPTForImageClassification`.
- ImageGPT comes in different sizes: there's ImageGPT-small, ImageGPT-medium and ImageGPT-large. The authors did also train an XL variant, which they didn't release. The differences in size are summarized in the following table:

Model variant | Depths | Hidden sizes | Decoder hidden size | Params (M) | ImageNet-1k Top 1 |

|---|---|---|---|---| | MiT-b0 | [2, 2, 2, 2] | [32, 64, 160, 256] | 256 | 3.7 | 70.5 | | MiT-b1 | [2, 2, 2, 2] | [64, 128, 320, 512] | 256 | 14.0 | 78.7 | | MiT-b2 | [3, 4, 6, 3] | [64, 128, 320, 512] | 768 | 25.4 | 81.6 | | MiT-b3 | [3, 4, 18, 3] | [64, 128, 320, 512] | 768 | 45.2 | 83.1 | | MiT-b4 | [3, 8, 27, 3] | [64, 128, 320, 512] | 768 | 62.6 | 83.6 | | MiT-b5 | [3, 6, 40, 3] | [64, 128, 320, 512] | 768 | 82.0 | 83.8 |

Args:

vocab_size (*int*, *optional*, defaults to 512):
Vocabulary size of the GPT-2 model. Defines the number of different tokens that can be represented by the *inputs_ids* passed when calling `ImageGPTModel` or `TFImageGPTModel`.

n_positions (*int*, *optional*, defaults to 32*32):
The maximum sequence length that this model might ever be used with. Typically set this to something large just in case (e.g., 512 or 1024 or 2048).

n_embd (*int*, *optional*, defaults to 512):
Dimensionality of the embeddings and hidden states.

n_layer (*int*, *optional*, defaults to 24):
Number of hidden layers in the Transformer encoder.

n_head (*int*, *optional*, defaults to 8):
Number of attention heads for each attention layer in the Transformer encoder.

n_inner (*int*, *optional*, defaults to None):
Dimensionality of the inner feed-forward layers. *None* will set it to 4 times n_embd

activation_function (*str*, *optional*, defaults to *"quick_gelu"*):
Activation function (can be one of the activation functions defined in src/transformers/activations.py). Defaults to "quick_gelu".

resid_pdrop (*float*, *optional*, defaults to 0.1):
The dropout probability for all fully connected layers in the embeddings, encoder, and pooler.

embd_pdrop (*int*, *optional*, defaults to 0.1):
The dropout ratio for the embeddings.

attn_pdrop (*float*, *optional*, defaults to 0.1):
The dropout ratio for the attention.

layer_norm_epsilon (*float*, *optional*, defaults to 1e-5):
The epsilon to use in the layer normalization layers.

initializer_range (*float*, *optional*, defaults to 0.02):
The standard deviation of the truncated_normal_initializer for initializing all weight matrices.

scale_attn_weights (*bool*, *optional*, defaults to *True*):
Scale attention weights by dividing by sqrt(hidden_size)..

use_cache (*bool*, *optional*, defaults to *True*):
Whether or not the model should return the last key/values attentions (not used by all models).

scale_attn_by_inverse_layer_idx (*bool*, *optional*, defaults to *False*):
Whether to additionally scale attention weights by *1 / layer_idx + 1*.

reorder_and_upcast_attn (*bool*, *optional*, defaults to *False*):
Whether to scale keys (K) prior to computing attention (dot-product) and upcast attention dot-product/softmax to float() when training with mixed precision.

`class transformers.models.layoutlm.configuration_layoutlm.`**`LayoutLMConfig`**(*vocab_size=30522*, *hidden_size=768*, *num_hidden_layers=12*, *num_attention_heads=12*, *intermediate_size=3072*, *hidden_act='gelu'*, *hidden_dropout_prob=0.1*, *attention_probs_dropout_prob=0.1*, *max_position_embeddings=512*, *type_vocab_size=2*, *initializer_range=0.02*, *layer_norm_eps=1e-12*, *pad_token_id=0*, *position_embedding_type='absolute'*, *use_cache=True*, *max_2d_position_embeddings=1024*, ***kwargs*)

The LayoutLM model was proposed in the paper LayoutLM: Pre-training of Text and Layout for Document Image Understanding by Yiheng Xu, Minghao Li, Lei Cui, Shaohan Huang, Furu Wei, and Ming Zhou. It's a simple but effective pretraining method of text and layout for document image understanding and information extraction tasks, such as form understanding and receipt understanding. It obtains state-of-the-art results on several downstream tasks:

- form understanding: the FUNSD dataset (a collection of 199 annotated forms comprising more than 30,000 words).
- receipt understanding: the SROIE dataset (a collection of 626 receipts for training and 347 receipts for testing).
- document image classification: the RVL-CDIP dataset (a collection of 400,000 images belonging to one of 16 classes).

The abstract from the paper is the following:

Pre-training techniques have been verified successfully in a variety of NLP tasks in recent years. Despite the widespread use of pretraining models for NLP applications, they almost exclusively focus on text-level manipulation, while neglecting layout and style information that is vital for document image understanding. In this paper, we propose the LayoutLM

to jointly model interactions between text and layout information across scanned document images, which is beneficial for a great number of real-world document image understanding tasks such as information extraction from scanned documents. Furthermore, we also leverage image features to incorporate words' visual information into LayoutLM. To the best of our knowledge, this is the first time that text and layout are jointly learned in a single framework for document-level pretraining. It achieves new state-of-the-art results in several downstream tasks, including form understanding (from 70.72 to 79.27), receipt understanding (from 94.02 to 95.24) and document image classification (from 93.07 to 94.42).

Tips:

- In addition to *input_ids*, ~`transformers.LayoutLMModel.forward` also expects the input *bbox*, which are the bounding boxes (i.e. 2D-positions) of the input tokens. These can be obtained using an external OCR engine such as Google's Tesseract (there's a Python wrapper available). Each bounding box should be in (x0, y0, x1, y1) format, where (x0, y0) corresponds to the position of the upper left corner in the bounding box, and (x1, y1) represents the position of the lower right corner. Note that one first needs to normalize the bounding boxes to be on a 0-1000 scale. To normalize, you can use the following function:

def normalize_bbox(bbox, width, height):

return `
int(1000 * (bbox[0] / width)), int(1000 * (bbox[1] / height)), int(1000 * (bbox[2] / width)), int(1000 * (bbox[3] / height)),

]

Here, *width* and *height* correspond to the width and height of the original document in which the token occurs. Those can be obtained using the Python Image Library (PIL) library for example, as follows:

from PIL import Image

Document can be a png, jpg, etc. PDFs must be converted to images. image = Image.open(name_of_your_document).convert("RGB")

width, height = image.size

Args:

vocab_size (*int*, *optional*, defaults to 30522):
Vocabulary size of the LayoutLM model. Defines the different tokens that can be represented by the *inputs_ids* passed to the forward method of `LayoutLMModel`.

hidden_size (*int*, *optional*, defaults to 768):
Dimensionality of the encoder layers and the pooler layer.

num_hidden_layers (*int*, *optional*, defaults to 12):
Number of hidden layers in the Transformer encoder.

num_attention_heads (*int*, *optional*, defaults to 12):
Number of attention heads for each attention layer in the Transformer encoder.

intermediate_size (*int*, *optional*, defaults to 3072):
Dimensionality of the "intermediate" (i.e., feed-forward) layer in the Transformer encoder.

hidden_act (*str* or *function*, *optional*, defaults to *"gelu"*):
The non-linear activation function (function or string) in the encoder and pooler. If string, *"gelu"*, *"relu"*, *"silu"* and *"gelu_new"* are supported.

hidden_dropout_prob (*float*, *optional*, defaults to 0.1):
The dropout probability for all fully connected layers in the embeddings, encoder, and pooler.

attention_probs_dropout_prob (*float*, *optional*, defaults to 0.1):
The dropout ratio for the attention probabilities.

max_position_embeddings (*int*, *optional*, defaults to 512):
The maximum sequence length that this model might ever be used with. Typically set this to something large just in case (e.g., 512 or 1024 or 2048).

type_vocab_size (*int*, *optional*, defaults to 2):
The vocabulary size of the *token_type_ids* passed into `LayoutLMModel`.

initializer_range (*float*, *optional*, defaults to 0.02):
The standard deviation of the truncated_normal_initializer for initializing all weight matrices.

layer_norm_eps (*float*, *optional*, defaults to 1e-12):
The epsilon used by the layer normalization layers.

pad_token_id (*int*, *optional*, defaults to 0):
The value used to pad input_ids.

position_embedding_type (*str*, *optional*, defaults to *"absolute"*):
Type of position embedding. Choose one of *"absolute"*, *"relative_key"*, *"relative_key_query"*. For positional embeddings use *"absolute"*. For more information on *"relative_key"*, please refer to [Self-Attention with Relative Position Representations (Shaw et al.) <https://arxiv.org/abs/1803.02155>`__. For more information on *"relative_key_query"*, please refer to *Method 4* in Improve Transformer Models with Better Relative Position Embeddings (Huang et al.).

use_cache (*bool*, *optional*, defaults to *True*):
Whether or not the model should return the last key/values attentions (not used by all models). Only relevant if *config.is_decoder=True*.

max_2d_position_embeddings (*int*, *optional*, defaults to 1024):
The maximum value that the 2D position embedding might ever used. Typically set this to something large just in case (e.g., 1024).

`class transformers.models.led.configuration_led.`**`LEDConfig`**(*vocab_size=50265*, *max_encoder_position_embeddings=16384*, *max_decoder_position_embeddings=1024*, *encoder_layers=12*, *encoder_ffn_dim=4096*, *encoder_attention_heads=16*, *decoder_layers=12*, *decoder_ffn_dim=4096*, *decoder_attention_heads=16*, *encoder_layerdrop=0.0*, *decoder_layerdrop=0.0*, *use_cache=True*, *is_encoder_decoder=True*, *activation_function='gelu'*, *d_model=1024*, *dropout=0.1*, *attention_dropout=0.0*, *activation_dropout=0.0*, *init_std=0.02*, *decoder_start_token_id=2*, *classifier_dropout=0.0*, *pad_token_id=1*, *bos_token_id=0*, *eos_token_id=2*, *attention_window: List[int] | int = 512*, ***kwargs*)

The LED model was proposed in Longformer: The Long-Document Transformer by Iz Beltagy, Matthew E. Peters, Arman Cohan.

The abstract from the paper is the following:

Transformer-based models are unable to process long sequences due to their self-attention operation, which scales quadratically with the sequence length. To address this limitation, we introduce the Longformer with an attention mechanism that scales linearly with sequence length, making it easy to process documents of thousands of tokens or longer. Longformer's attention mechanism is a drop-in replacement for the standard self-attention and combines a local windowed attention with a task motivated global attention. Following prior work on long-sequence transformers, we evaluate Longformer on character-level language modeling and achieve state-of-the-art results on text8 and enwik8. In contrast to most prior work, we also pretrain Longformer and finetune it on a variety of downstream tasks. Our pretrained Longformer consistently outperforms RoBERTa on long document tasks and sets new state-of-the-art results on WikiHop and TriviaQA. We finally introduce the Longformer-Encoder-Decoder (LED), a Longformer variant for supporting long document generative sequence-to-sequence tasks, and demonstrate its effectiveness on the arXiv summarization dataset.

Tips:

- `LEDForConditionalGeneration` is an extension of `BartForConditionalGeneration` exchanging the traditional *self-attention* layer with *Longformer*'s *chunked self-attention* layer. `LEDTokenizer` is an alias of `BartTokenizer`.
- LED works very well on long-range *sequence-to-sequence* tasks where the *input_ids* largely exceed a length of 1024 tokens.
- LED pads the *input_ids* to be a multiple of *config.attention_window* if required. Therefore a small speed-up is gained, when `LEDTokenizer` is used with the *pad_to_multiple_of* argument.
- LED makes use of *global attention* by means of the *global_attention_mask* (see `LongformerModel`). For summarization, it is advised to put *global attention* only on the first <*s*> token. For question answering, it is advised to put *global attention* on all tokens of the question.
- To fine-tune LED on all 16384, *gradient checkpointing* can be enabled in case training leads to out-of-memory (OOM) errors. This can be done by executing *model.gradient_checkpointing_enable()*.

  Moreover, the *use_cache=False*
  flag can be used to disable the caching mechanism to save memory.
- A notebook showing how to evaluate LED, can be accessed here.
- A notebook showing how to fine-tune LED, can be accessed here.
- LED is a model with absolute position embeddings so it's usually advised to pad the inputs on the right rather than the left.

This model was contributed by patrickvonplaten.

Args:

vocab_size (*int*, *optional*, defaults to 50265):
Vocabulary size of the LED model. Defines the number of different tokens that can be represented by the *inputs_ids* passed when calling `LEDModel` or `TFLEDModel`.

d_model (*int*, *optional*, defaults to 1024):
Dimensionality of the layers and the pooler layer.

encoder_layers (*int*, *optional*, defaults to 12):
Number of encoder layers.

decoder_layers (*int*, *optional*, defaults to 12):
Number of decoder layers.

encoder_attention_heads (*int*, *optional*, defaults to 16):
Number of attention heads for each attention layer in the Transformer encoder.

decoder_attention_heads (*int*, *optional*, defaults to 16):
Number of attention heads for each attention layer in the Transformer decoder.

decoder_ffn_dim (*int*, *optional*, defaults to 4096):
Dimensionality of the "intermediate" (often named feed-forward) layer in decoder.

encoder_ffn_dim (*int*, *optional*, defaults to 4096):
Dimensionality of the "intermediate" (often named feed-forward) layer in decoder.

activation_function (*str* or *function*, *optional*, defaults to *"gelu"*):
The non-linear activation function (function or string) in the encoder and pooler. If string, *"gelu"*, *"relu"*, *"silu"* and *"gelu_new"* are supported.

dropout (*float*, *optional*, defaults to 0.1):
The dropout probability for all fully connected layers in the embeddings, encoder, and pooler.

attention_dropout (*float*, *optional*, defaults to 0.0):
The dropout ratio for the attention probabilities.

activation_dropout (*float*, *optional*, defaults to 0.0):
The dropout ratio for activations inside the fully connected layer.

classifier_dropout (*float*, *optional*, defaults to 0.0):
The dropout ratio for classifier.

max_encoder_position_embeddings (*int*, *optional*, defaults to 16384):
The maximum sequence length that the encoder might ever be used with.

max_decoder_position_embeddings (*int*, *optional*, defaults to 16384):
The maximum sequence length that the decoder might ever be used with.

init_std (*float*, *optional*, defaults to 0.02):
The standard deviation of the truncated_normal_initializer for initializing all weight matrices.

encoder_layerdrop (*float*, *optional*, defaults to 0.0):
The LayerDrop probability for the encoder. See the LayerDrop paper for more details.

decoder_layerdrop (*float*, *optional*, defaults to 0.0):
The LayerDrop probability for the decoder. See the LayerDrop paper for more details.

use_cache (*bool*, *optional*, defaults to *True*):
Whether or not the model should return the last key/values attentions (not used by all models)

`class transformers.models.llama.configuration_llama.LlamaConfig`(*vocab_size=32000*, *hidden_size=4096*, *intermediate_size=11008*, *num_hidden_layers=32*, *num_attention_heads=32*, *num_key_value_heads=None*, *hidden_act='silu'*, *max_position_embeddings=2048*, *initializer_range=0.02*, *rms_norm_eps=1e-06*, *use_cache=True*, *pad_token_id=None*, *bos_token_id=1*, *eos_token_id=2*, *pretraining_tp=1*, *tie_word_embeddings=False*, *rope_theta=10000.0*, *rope_scaling=None*, *attention_bias=False*, *attention_dropout=0.0*, ***kwargs*)

The LLaMA model was proposed in LLaMA: Open and Efficient Foundation Language Models by Hugo Touvron, Thibaut Lavril, Gautier Izacard, Xavier Martinet, Marie-Anne Lachaux, Timothée Lacroix, Baptiste Rozière, Naman Goyal, Eric Hambro, Faisal Azhar, Aurelien Rodriguez, Armand Joulin, Edouard Grave, Guillaume Lample. It is a collection of foundation language models ranging from 7B to 65B parameters.

The abstract from the paper is the following:

*We introduce LLaMA, a collection of foundation language models ranging from 7B to 65B parameters. We train our models on trillions of tokens, and show that it is possible to train state-of-the-art models using publicly available datasets exclusively, without resorting to proprietary and inaccessible datasets. In particular, LLaMA-13B outperforms GPT-3 (175B) on most benchmarks, and LLaMA-65B is competitive with the best models, Chinchilla-70B and PaLM-540B. We release all our models to the research community. *

Tips:

- Weights for the LLaMA models can be obtained from by filling out this form
- After downloading the weights, they will need to be converted to the Hugging Face Transformers format using the conversion script. The script can be called with the following (example) command:

```bash python src/transformers/models/llama/convert_llama_weights_to_hf.py

–input_dir /path/to/downloaded/llama/weights –model_size 7B –output_dir /output/path

```

- After conversion, the model and tokenizer can be loaded via:

from transformers import LlamaForCausalLM, LlamaTokenizer

tokenizer = LlamaTokenizer.from_pretrained("/output/path") model = LlamaForCausalLM.from_pretrained("/output/path")

Note that executing the script requires enough CPU RAM to host the whole model in float16 precision (even if the biggest versions come in several checkpoints they each contain a part of each weight of the model, so we need to load them all in RAM). For the 65B model, it's thus 130GB of RAM needed.

- The LLaMA tokenizer is a BPE model based on sentencepiece. One quirk of sentencepiece is that when decoding a sequence, if the first token is the start of the word (e.g. "Banana"), the tokenizer does not prepend the prefix space to the string.

This model was contributed by zphang with contributions from BlackSamorez. The code of the implementation in Hugging Face is based on GPT-NeoX here. The original code of the authors can be found here.

Args:

vocab_size (*int*, *optional*, defaults to 32000):
Vocabulary size of the LLaMA model. Defines the number of different tokens that can be represented by the *inputs_ids* passed when calling `LlamaModel`

hidden_size (*int*, *optional*, defaults to 4096):
Dimension of the hidden representations.

intermediate_size (*int*, *optional*, defaults to 11008):
Dimension of the MLP representations.

num_hidden_layers (*int*, *optional*, defaults to 32):
Number of hidden layers in the Transformer decoder.

num_attention_heads (*int*, *optional*, defaults to 32):
Number of attention heads for each attention layer in the Transformer decoder.

num_key_value_heads (*int*, *optional*):
This is the number of key_value heads that should be used to implement Grouped Query Attention. If *num_key_value_heads=num_attention_heads*, the model will use Multi Head Attention (MHA), if num_key_value_heads=1 the model will use Multi Query Attention (MQA) otherwise GQA is used. When converting a multi-head checkpoint to a GQA checkpoint, each group key and value head should be constructed by meanpooling all the original heads within that group. For more details checkout `this paper. If it is not specified, will default to *num_attention_heads*.

hidden_act (*str* or *function*, *optional*, defaults to *"silu"*):
The non-linear activation function (function or string) in the decoder.

max_position_embeddings (*int*, *optional*, defaults to 2048):
The maximum sequence length that this model might ever be used with. Llama 1 supports up to 2048 tokens, Llama 2 up to 4096, CodeLlama up to 16384.

initializer_range (*float*, *optional*, defaults to 0.02):
The standard deviation of the truncated_normal_initializer for initializing all weight matrices.

rms_norm_eps (*float*, *optional*, defaults to 1e-06):
The epsilon used by the rms normalization layers.

use_cache (*bool*, *optional*, defaults to *True*):
Whether or not the model should return the last key/values attentions (not used by all models). Only relevant if *config.is_decoder=True*.

pad_token_id (*int*, *optional*):
Padding token id.

bos_token_id (*int*, *optional*, defaults to 1):
Beginning of stream token id.

eos_token_id (*int*, *optional*, defaults to 2):
End of stream token id.

pretraining_tp (*int*, *optional*, defaults to 1):
Experimental feature. Tensor parallelism rank used during pretraining. Please refer to this document to understand more about it. This value is necessary to ensure exact reproducibility of the pretraining results. Please refer to this issue.

tie_word_embeddings (*bool*, *optional*, defaults to *False*):
Whether to tie weight embeddings

rope_theta (*float*, *optional*, defaults to 10000.0):
The base period of the RoPE embeddings.

rope_scaling (*Dict*, *optional*):
Dictionary containing the scaling configuration for the RoPE embeddings. Currently supports two scaling strategies: linear and dynamic. Their scaling factor must be a float greater than 1. The expected format is *{"type": strategy name, "factor": scaling factor}*. When using this flag, don't update *max_position_embeddings* to the expected new maximum. See the following thread for more information on how these scaling strategies behave: https://www.reddit.com/r/LocalLLaMA/comments/14mrgpr/dynamically_scaled_rope_further_increases/. This is an experimental feature, subject to breaking API changes in future versions.

attention_bias (*bool*, defaults to *False*, *optional*, defaults to *False*):
Whether to use a bias in the query, key, value and output projection layers during self-attention.

attention_dropout (*float*, *optional*, defaults to 0.0):
The dropout ratio for the attention probabilities.

```
>>> from transformers import LlamaModel, LlamaConfig
```

```
>>> # Initializing a LLaMA llama-7b style configuration
>>> configuration = LlamaConfig()
```

```
>>> # Initializing a model from the llama-7b style configuration
>>> model = LlamaModel(configuration)
```

```
>>> # Accessing the model configuration
>>> configuration = model.config
```

class transformers.models.longformer.configuration_longformer.**LongformerConfig**(*attention_window: List[int] | int = 512*, *sep_token_id: int = 2*, *pad_token_id: int = 1*, *bos_token_id: int = 0*, *eos_token_id: int = 2*, *vocab_size: int = 30522*, *hidden_size: int = 768*, *num_hidden_layers: int = 12*, *num_attention_heads: int = 12*, *intermediate_size: int = 3072*, *hidden_act: str = 'gelu'*, *hidden_dropout_prob: float = 0.1*, *attention_probs_dropout_prob: float = 0.1*, *max_position_embeddings: int = 512*, *type_vocab_size: int = 2*, *initializer_range: float = 0.02*, *layer_norm_eps: float = 1e-12*, *onnx_export: bool = False*, ***kwargs*)

The Longformer model was presented in Longformer: The Long-Document Transformer by Iz Beltagy, Matthew E. Peters, Arman Cohan.

The abstract from the paper is the following:

Transformer-based models are unable to process long sequences due to their self-attention operation, which scales quadratically with the sequence length. To address this limitation, we introduce the Longformer with an attention mechanism that scales linearly with sequence length, making it easy to process documents of thousands of tokens or longer. Longformer's attention mechanism is a drop-in replacement for the standard self-attention and combines a local windowed attention with a task motivated global attention. Following prior work on long-sequence transformers, we evaluate Longformer on character-level language modeling and achieve state-of-the-art results on text8 and enwik8. In contrast to most prior work, we also pretrain Longformer and finetune it on a variety of downstream tasks. Our pretrained Longformer consistently outperforms RoBERTa on long document tasks and sets new state-of-the-art results on WikiHop and TriviaQA.

Tips:

- Since the Longformer is based on RoBERTa, it doesn't have *token_type_ids*. You don't need to indicate which token belongs to which segment. Just separate your segments with the separation token *tokenizer.sep_token* (or *</s>*).
- A transformer model replacing the attention matrices by sparse matrices to go faster. Often, the local context (e.g., what are the two tokens left and right?) is enough to take action for a given token. Some preselected input tokens are still given global attention, but the attention matrix has way less parameters, resulting in a speed-up. See the local attention section for more information.

This model was contributed by beltagy. The Authors' code can be found here.

Args:

vocab_size (*int*, *optional*, defaults to 30522):
Vocabulary size of the Longformer model. Defines the number of different tokens that can be represented by the *inputs_ids* passed when calling `LongformerModel` or `TFLongformerModel`.

hidden_size (*int*, *optional*, defaults to 768):
Dimensionality of the encoder layers and the pooler layer.

num_hidden_layers (*int*, *optional*, defaults to 12):
Number of hidden layers in the Transformer encoder.

num_attention_heads (*int*, *optional*, defaults to 12):
Number of attention heads for each attention layer in the Transformer encoder.

intermediate_size (*int*, *optional*, defaults to 3072):
Dimensionality of the "intermediate" (often named feed-forward) layer in the Transformer encoder.

hidden_act (*str* or *Callable*, *optional*, defaults to *"gelu"*):
The non-linear activation function (function or string) in the encoder and pooler. If string, *"gelu"*, *"relu"*, *"silu"* and *"gelu_new"* are supported.

hidden_dropout_prob (*float*, *optional*, defaults to 0.1):
The dropout probability for all fully connected layers in the embeddings, encoder, and pooler.

attention_probs_dropout_prob (*float*, *optional*, defaults to 0.1):
The dropout ratio for the attention probabilities.

max_position_embeddings (*int*, *optional*, defaults to 512):
The maximum sequence length that this model might ever be used with. Typically set this to something large just in case (e.g., 512 or 1024 or 2048).

type_vocab_size (*int*, *optional*, defaults to 2):
The vocabulary size of the *token_type_ids* passed when calling `LongformerModel` or `TFLongformerModel`.

initializer_range (*float*, *optional*, defaults to 0.02):
The standard deviation of the truncated_normal_initializer for initializing all weight matrices.

layer_norm_eps (*float*, *optional*, defaults to 1e-12):
The epsilon used by the layer normalization layers.

attention_window (*int* or *List[int]*, *optional*, defaults to 512):
Size of an attention window around each token. If an *int*, use the same size for all layers. To specify a different window size for each layer, use a *List[int]* where *len(attention_window) == num_hidden_layers*.

`class` transformers.models.longt5.configuration_longt5.**LongT5Config**(*vocab_size=32128*, *d_model=512*, *d_kv=64*, *d_ff=2048*, *num_layers=6*, *num_decoder_layers=None*, *num_heads=8*, *local_radius=127*, *global_block_size=16*, *relative_attention_num_buckets=32*, *relative_attention_max_distance=128*, *dropout_rate=0.1*, *layer_norm_epsilon=1e-06*, *initializer_factor=1.0*, *feed_forward_proj='relu'*, *is_encoder_decoder=True*, *encoder_attention_type='local'*, *use_cache=True*, *pad_token_id=0*, *eos_token_id=1*, ***kwargs*)

The LongT5 model was proposed in LongT5: Efficient Text-To-Text Transformer for Long Sequences by Mandy Guo, Joshua Ainslie, David Uthus, Santiago Ontanon, Jianmo Ni, Yun-Hsuan Sung and Yinfei Yang. It's an encoder-decoder transformer pre-trained in a text-to-text denoising generative setting. LongT5 model is an extension of T5 model, and it enables using one of the two different efficient attention mechanisms - (1) Local attention, or (2) Transient-Global attention.

The abstract from the paper is the following:

Recent work has shown that either (1) increasing the input length or (2) increasing model size can improve the performance of Transformer-based neural models. In this paper, we present a new model, called LongT5, with which we explore the effects of scaling both the input length and model size at the same time. Specifically, we integrated attention ideas from long-input transformers (ETC), and adopted pre-training strategies from summarization pre-training (PEGASUS) into the scalable T5 architecture. The result is a new attention mechanism we call {em Transient Global} (TGlobal), which mimics ETC's local/global attention mechanism, but without requiring additional side-inputs. We are able to achieve state-of-the-art results on several summarization tasks and outperform the original T5 models on question answering tasks.

Tips:

- `LongT5ForConditionalGeneration` is an extension of `T5ForConditionalGeneration` exchanging the traditional

encoder *self-attention* layer with efficient either *local* attention or *transient-global* (*tglobal*) attention. - Unlike the T5 model, LongT5 does not use a task prefix. Furthermore, it uses a different pre-training objective inspired by the pre-

training of `PegasusForConditionalGeneration`. - LongT5 model is designed to work efficiently and very well on long-range *sequence-to-sequence* tasks where the input sequence exceeds commonly used 512 tokens. It is capable of handling input sequences of a length up to 16,384 tokens. - For *Local Attention*, the sparse sliding-window local attention operation allows a given token to attend only *r* tokens to the left and right of it (with *r=127* by default). *Local Attention* does not introduce any new parameters to the model. The complexity of the mechanism is linear in input sequence length *l*: *O(l*r)*. - *Transient Global Attention* is an extension of the *Local Attention*. It, furthermore, allows each input token to interact with all other tokens in the layer. This is achieved via splitting an input sequence into blocks of a fixed length *k* (with a default *k=16*). Then, a global token for such a block is obtained via summing and normalizing the embeddings of every token in the block. Thanks to this, the attention allows each token to attend to both nearby tokens like in Local attention, and also every global token like in the case of standard global attention (*transient* represents the fact the global tokens are constructed dynamically within each attention operation). As a consequence, *TGlobal* attention introduces a few new parameters – global relative position biases and a layer normalization for global token's embedding. The complexity of this mechanism is *O(l(r + l/k))*. - An example showing how to evaluate a fine-tuned LongT5 model on the pubmed dataset is below.

```
>>> import evaluate
>>> from datasets import load_dataset
>>> from transformers import AutoTokenizer, LongT5ForConditionalGeneration
```

```
>>> dataset = load_dataset("scientific_papers", "pubmed", split="validation")
>>> model = (
...     LongT5ForConditionalGeneration.from_pretrained("Stancld/longt5-tglobal-large-
↪16384-pubmed-3k_steps")
...     .to("cuda")
...     .half()
... )
>>> tokenizer = AutoTokenizer.from_pretrained("Stancld/longt5-tglobal-large-16384-pubmed-
↪3k_steps")
```

```
>>> def generate_answers(batch):
...     inputs_dict = tokenizer(
...         batch`"article"], max_length=16384, padding="max_length", truncation=True,␣
↪return_tensors="pt"
...     )
...     input_ids = inputs_dict.input_ids.to("cuda")
...     attention_mask = inputs_dict.attention_mask.to("cuda")
...     output_ids = model.generate(input_ids, attention_mask=attention_mask, max_
↪length=512, num_beams=2)
...     batch["predicted_abstract"] = tokenizer.batch_decode(output_ids, skip_special_
↪tokens=True)
...     return batch
```

```
>>> result = dataset.map(generate_answer, batched=True, batch_size=2)
>>> rouge = evaluate.load("rouge")
>>> rouge.compute(predictions=result["predicted_abstract"], references=result["abstract
↪"])
```

This model was contributed by [stancld <https://huggingface.co/stancld>`__. The original code can be found here.

Arguments:

vocab_size (*int*, *optional*, defaults to 32128):
Vocabulary size of the LongT5 model. Defines the number of different tokens that can be represented by the *inputs_ids* passed when calling `LongT5Model`.

d_model (*int*, *optional*, defaults to 512):
Size of the encoder layers and the pooler layer.

d_kv (*int*, *optional*, defaults to 64):
Size of the key, query, value projections per attention head. *d_kv* has to be equal to *d_model // num_heads*.

d_ff (*int*, *optional*, defaults to 2048):
Size of the intermediate feed forward layer in each *LongT5Block*.

num_layers (*int*, *optional*, defaults to 6):
Number of hidden layers in the Transformer encoder.

num_decoder_layers (*int*, *optional*):
Number of hidden layers in the Transformer decoder. Will use the same value as *num_layers* if not set.

num_heads (*int*, *optional*, defaults to 8):
Number of attention heads for each attention layer in the Transformer encoder.

local_radius (*int*, *optional*, defaults to 127)
Number of tokens to the left/right for each token to locally self-attend in a local attention mechanism.

global_block_size (*int*, *optional*, defaults to 16)
Lenght of blocks an input sequence is divided into for a global token representation. Used only for *encoder_attention_type = "transient-global"*.

relative_attention_num_buckets (*int*, *optional*, defaults to 32):
The number of buckets to use for each attention layer.

relative_attention_max_distance (*int*, *optional*, defaults to 128):
The maximum distance of the longer sequences for the bucket separation.

dropout_rate (*float*, *optional*, defaults to 0.1):
The ratio for all dropout layers.

layer_norm_eps (*float*, *optional*, defaults to 1e-6):
The epsilon used by the layer normalization layers.

initializer_factor (*float*, *optional*, defaults to 1):
A factor for initializing all weight matrices (should be kept to 1, used internally for initialization testing).

feed_forward_proj (*string*, *optional*, defaults to *"relu"*):
Type of feed forward layer to be used. Should be one of *"relu"* or *"gated-gelu"*. LongT5v1.1 uses the *"gated-gelu"* feed forward projection. Original LongT5 implementation uses *"gated-gelu"*.

encoder_attention_type (*string*, *optional*, defaults to *"local"*):
Type of encoder attention to be used. Should be one of *"local"* or *"transient-global"*, which are supported by LongT5 implementation.

use_cache (*bool*, *optional*, defaults to *True*):
Whether or not the model should return the last key/values attentions (not used by all models).

class transformers.models.luke.configuration_luke.**LukeConfig**(*vocab_size=50267*, *entity_vocab_size=500000*, *hidden_size=768*, *entity_emb_size=256*, *num_hidden_layers=12*, *num_attention_heads=12*, *intermediate_size=3072*, *hidden_act='gelu'*, *hidden_dropout_prob=0.1*, *attention_probs_dropout_prob=0.1*, *max_position_embeddings=512*, *type_vocab_size=2*, *initializer_range=0.02*, *layer_norm_eps=1e-12*, *use_entity_aware_attention=True*, *classifier_dropout=None*, *pad_token_id=1*, *bos_token_id=0*, *eos_token_id=2*, ***kwargs*)

The LUKE model was proposed in LUKE: Deep Contextualized Entity Representations with Entity-aware Self-attention by Ikuya Yamada, Akari Asai, Hiroyuki Shindo, Hideaki Takeda and Yuji Matsumoto. It is based on RoBERTa and adds entity embeddings as well as an entity-aware self-attention mechanism, which helps improve performance on various downstream tasks involving reasoning about entities such as named entity recognition, extractive and cloze-style question answering, entity typing, and relation classification.

The abstract from the paper is the following:

Entity representations are useful in natural language tasks involving entities. In this paper, we propose new pretrained contextualized representations of words and entities based on the bidirectional transformer. The proposed model treats words and entities in a given text as independent tokens, and outputs contextualized representations of them. Our model is trained using a new pretraining task based on the masked language model of BERT. The task involves predicting randomly masked words and entities in a large entity-annotated corpus retrieved from Wikipedia. We also propose an entity-aware self-attention mechanism that is an extension of the self-attention mechanism of the transformer, and considers the types of tokens (words or entities) when computing attention scores. The proposed model achieves impressive empirical performance on a wide range of entity-related tasks. In particular, it obtains state-of-the-art results on five well-known datasets: Open Entity (entity typing), TACRED (relation classification), CoNLL-2003 (named entity recognition), ReCoRD (cloze-style question answering), and SQuAD 1.1 (extractive question answering).

Tips:

- This implementation is the same as `RobertaModel` with the addition of entity embeddings as well as an entity-aware self-attention mechanism, which improves performance on tasks involving reasoning about entities.
- LUKE treats entities as input tokens; therefore, it takes *entity_ids*, *entity_attention_mask*, *entity_token_type_ids* and *entity_position_ids* as extra input. You can obtain those using `LukeTokenizer`.
- `LukeTokenizer` takes *entities* and *entity_spans* (character-based start and end positions of the entities in the input text) as extra input. *entities* typically consist of `MASK] entities or Wikipedia entities. The brief description when inputting these entities are as follows:
  - *Inputting [MASK] entities to compute entity representations*: The [MASK] entity is used to mask entities to be predicted during pretraining. When LUKE receives the [MASK] entity, it tries to predict the original entity by gathering the information about the entity from the input text. Therefore, the [MASK] entity can be used to address downstream tasks requiring the information of entities in text such as entity typing, relation classification, and named entity recognition.
  - *Inputting Wikipedia entities to compute knowledge-enhanced token representations*: LUKE learns rich information (or knowledge) about Wikipedia entities during pretraining and stores the information in its entity

embedding. By using Wikipedia entities as input tokens, LUKE outputs token representations enriched by the information stored in the embeddings of these entities. This is particularly effective for tasks requiring real-world knowledge, such as question answering.

- There are three head models for the former use case:
  - `LukeForEntityClassification`, for tasks to classify a single entity in an input text such as entity typing, e.g. the [Open Entity dataset <https://www.cs.utexas.edu/~eunsol/html_pages/open_entity.html>`__. This model places a linear head on top of the output entity representation.
  - `LukeForEntityPairClassification`, for tasks to classify the relationship between two entities such as relation classification, e.g. the TACRED dataset. This model places a linear head on top of the concatenated output representation of the pair of given entities.
  - `LukeForEntitySpanClassification`, for tasks to classify the sequence of entity spans, such as named entity recognition (NER). This model places a linear head on top of the output entity representations. You can address NER using this model by inputting all possible entity spans in the text to the model.

  `LukeTokenizer` has a *task* argument, which enables you to easily create an input to these head models by specifying *task="entity_classification"*, *task="entity_pair_classification"*, or *task="entity_span_classification"*. Please refer to the example code of each head models.

  A demo notebook on how to fine-tune `LukeForEntityPairClassification` for relation classification can be found here.

  There are also 3 notebooks available, which showcase how you can reproduce the results as reported in the paper with the HuggingFace implementation of LUKE. They can be found here.

class transformers.models.m2m_100.configuration_m2m_100.**M2M100Config**(*vocab_size=128112*, *max_position_embeddings=1024*, *encoder_layers=12*, *encoder_ffn_dim=4096*, *encoder_attention_heads=16*, *decoder_layers=12*, *decoder_ffn_dim=4096*, *decoder_attention_heads=16*, *encoder_layerdrop=0.05*, *decoder_layerdrop=0.05*, *use_cache=True*, *is_encoder_decoder=True*, *activation_function='relu'*, *d_model=1024*, *dropout=0.1*, *attention_dropout=0.1*, *activation_dropout=0.0*, *init_std=0.02*, *decoder_start_token_id=2*, *scale_embedding=True*, *pad_token_id=1*, *bos_token_id=0*, *eos_token_id=2*, ***kwargs*)

The M2M100 model was proposed in Beyond English-Centric Multilingual Machine Translation by Angela Fan, Shruti Bhosale, Holger Schwenk, Zhiyi Ma, Ahmed El-Kishky, Siddharth Goyal, Mandeep Baines, Onur Celebi, Guillaume

Wenzek, Vishrav Chaudhary, Naman Goyal, Tom Birch, Vitaliy Liptchinsky, Sergey Edunov, Edouard Grave, Michael Auli, Armand Joulin.

The abstract from the paper is the following:

Existing work in translation demonstrated the potential of massively multilingual machine translation by training a single model able to translate between any pair of languages. However, much of this work is English-Centric by training only on data which was translated from or to English. While this is supported by large sources of training data, it does not reflect translation needs worldwide. In this work, we create a true Many-to-Many multilingual translation model that can translate directly between any pair of 100 languages. We build and open source a training dataset that covers thousands of language directions with supervised data, created through large-scale mining. Then, we explore how to effectively increase model capacity through a combination of dense scaling and language-specific sparse parameters to create high quality models. Our focus on non-English-Centric models brings gains of more than 10 BLEU when directly translating between non-English directions while performing competitively to the best single systems of WMT. We open-source our scripts so that others may reproduce the data, evaluation, and final M2M-100 model.

This model was contributed by valhalla.

#Args:

vocab_size (*int*, *optional*, defaults to 50265):
Vocabulary size of the M2M100 model. Defines the number of different tokens that can be represented by the *inputs_ids* passed when calling `M2M100Model` or

d_model (*int*, *optional*, defaults to 1024):
Dimensionality of the layers and the pooler layer.

encoder_layers (*int*, *optional*, defaults to 12):
Number of encoder layers.

decoder_layers (*int*, *optional*, defaults to 12):
Number of decoder layers.

encoder_attention_heads (*int*, *optional*, defaults to 16):
Number of attention heads for each attention layer in the Transformer encoder.

decoder_attention_heads (*int*, *optional*, defaults to 16):
Number of attention heads for each attention layer in the Transformer decoder.

decoder_ffn_dim (*int*, *optional*, defaults to 4096):
Dimensionality of the "intermediate" (often named feed-forward) layer in decoder.

encoder_ffn_dim (*int*, *optional*, defaults to 4096):
Dimensionality of the "intermediate" (often named feed-forward) layer in decoder.

activation_function (*str* or *function*, *optional*, defaults to *"gelu"*):
The non-linear activation function (function or string) in the encoder and pooler. If string, *"gelu"*, *"relu"*, *"silu"* and *"gelu_new"* are supported.

dropout (*float*, *optional*, defaults to 0.1):
The dropout probability for all fully connected layers in the embeddings, encoder, and pooler.

attention_dropout (*float*, *optional*, defaults to 0.0):
The dropout ratio for the attention probabilities.

activation_dropout (*float*, *optional*, defaults to 0.0):
The dropout ratio for activations inside the fully connected layer.

classifier_dropout (*float*, *optional*, defaults to 0.0):
The dropout ratio for classifier.

max_position_embeddings (*int*, *optional*, defaults to 1024):
The maximum sequence length that this model might ever be used with. Typically set this to something large just in case (e.g., 512 or 1024 or 2048).

init_std (*float*, *optional*, defaults to 0.02):
The standard deviation of the truncated_normal_initializer for initializing all weight matrices.

encoder_layerdrop (*float*, *optional*, defaults to 0.0):
The LayerDrop probability for the encoder. See the LayerDrop paper for more details.

decoder_layerdrop (*float*, *optional*, defaults to 0.0):
The LayerDrop probability for the decoder. See the LayerDrop paper for more details.

use_cache (*bool*, *optional*, defaults to *True*):
Whether or not the model should return the last key/values attentions (not used by all models).

`class transformers.models.mamba.configuration_mamba.`**`MambaConfig`**(*vocab_size=50280*, *hidden_size=768*, *state_size=16*, *num_hidden_layers=32*, *layer_norm_epsilon=1e-05*, *pad_token_id=0*, *bos_token_id=0*, *eos_token_id=0*, *expand=2*, *conv_kernel=4*, *use_bias=False*, *use_conv_bias=True*, *hidden_act='silu'*, *initializer_range=0.1*, *residual_in_fp32=True*, *time_step_rank='auto'*, *time_step_scale=1.0*, *time_step_min=0.001*, *time_step_max=0.1*, *time_step_init_scheme='random'*, *time_step_floor=0.0001*, *rescale_prenorm_residual=False*, *use_cache=True*, ***kwargs*)

The Mamba model was proposed in Mamba: Linear-Time Sequence Modeling with Selective State Spaces by Albert Gu and Tri Dao.

This model is a new paradigm architecture based on *state-space-models*. You can read more about the intuition behind these here.

The abstract from the paper is the following:

Foundation models, now powering most of the exciting applications in deep learning, are almost universally based on the Transformer architecture and its core attention module. Many subquadratic-time architectures such as linear attention, gated convolution and recurrent models, and structured state space models (SSMs) have been developed to address Transformers' computational inefficiency on long sequences, but they have not performed as well as attention on important modalities such as language. We identify that a key weakness of such models is their inability to perform content-based reasoning, and make several improvements. First, simply letting the SSM parameters be functions of the input addresses their weakness with discrete modalities, allowing the model to selectively propagate or forget information along the sequence length dimension depending on the current token. Second, even though this change prevents the use of efficient convolutions, we design a hardware-aware parallel algorithm in recurrent mode. We integrate these selective SSMs into a simplified end-to-end neural network architecture without attention or even MLP blocks (Mamba). Mamba enjoys fast inference (5× higher throughput than Transformers) and linear scaling in sequence length, and its performance improves on real data up to million-length sequences. As a general sequence model backbone, Mamba achieves state-of-the-art performance across several modalities such as language, audio, and genomics. On language

modeling, our Mamba-3B model outperforms Transformers of the same size and matches Transformers twice its size, both in pretraining and downstream evaluation.

Tips:

- Mamba is a new *state space model* architecture that rivals the classic Transformers. It is based on the line of progress on structured state space models, with an efficient hardware-aware design and implementation in the spirit of FlashAttention.
- Mamba stacks *mixer* layers, which are the equivalent of *Attention* layers. The core logic of *mamba* is held in the *MambaMixer* class.
- Two implementations cohabit: one is optimized and uses fast cuda kernels, while the other one is naive but can run on any device!
- The current implementation leverages the original cuda kernels: the equivalent of flash attention for Mamba are hosted in the ``mamba-ssm``(https://github.com/state-spaces/mamba) and the ``causal_conv1d``(https://github.com/Dao-AILab/causal-conv1d) repositories. Make sure to install them if your hardware supports them!
- Contributions to make the naive path faster are welcome

This model was contributed by ArthurZ. The original code can be found here.

Usage

#Args:

vocab_size (*int*, *optional*, defaults to 50280):
Vocabulary size of the MAMBA model. Defines the number of different tokens that can be represented by the *inputs_ids* passed when calling `MambaModel`.

hidden_size (*int*, *optional*, defaults to 768):
Dimensionality of the embeddings and hidden states.

state_size (*int*, *optional*, defaults to 16): shape of the state space latents. num_hidden_layers (*int*, *optional*, defaults to 32):

Number of hidden layers in the model.

layer_norm_epsilon (*float*, *optional*, defaults to 1e-05):
The epsilon to use in the layer normalization layers.

pad_token_id (*int*, *optional*, defaults to 0):
Padding token id.

bos_token_id (*int*, *optional*, defaults to 0):
The id of the beginning of sentence token in the vocabulary.

eos_token_id (*int*, *optional*, defaults to 0):
The id of the end of sentence token in the vocabulary.

expand (*int*, *optional*, defaults to 2): Expanding factor used to determine the intermediate size. conv_kernel (*int*, *optional*, defaults to 4): Size of the convolution kernel. use_bias (*bool*, *optional*, defaults to *False*):

Whether or not to use bias in [“in_proj”, “out_proj”] of the mixer block

use_conv_bias (*bool*, *optional*, defaults to *True*):
Whether or not to use bias in the convolution layer of the mixer block.

hidden_act (*str*, *optional*, defaults to *“silu”*):
The non-linear activation function (function or string) in the decoder.

initializer_range (*float*, *optional*, defaults to 0.1):
The standard deviation of the truncated_normal_initializer for initializing all weight matrices.

residual_in_fp32 (*bool*, *optional*, defaults to *True*):
Whether or not residuals should be in *float32*. If set to *False* residuals will keep the same *dtype* as the rest of the model

time_step_rank (*Union[int,str]*, *optional*, defaults to *"auto"*):
Rank of the the discretization projection matrix. *"auto"* means that it will default to *math.ceil(self.hidden_size / 16)*

time_step_scale (*float*, *optional*, defaults to 1.0):
Scale used used to scale *dt_proj.bias*.

time_step_min (*float*, *optional*, defaults to 0.001):
Minimum *time_step* used to bound *dt_proj.bias*.

time_step_max (*float*, *optional*, defaults to 0.1):
Maximum *time_step* used to bound *dt_proj.bias*.

time_step_init_scheme (*float*, *optional*, defaults to *"random"*):
Init scheme used for *dt_proj.weight*. Should be one of *["random","uniform"]*

time_step_floor (*float*, *optional*, defaults to 0.0001):
Minimum clamping value of the *dt_proj.bias* layer initialization.

rescale_prenorm_residual (*bool*, *optional*, defaults to *False*):
Whether or not to rescale *out_proj* weights when initializing.

use_cache (*bool*, *optional*, defaults to *True*):
Whether or not the cache should be used.

`class` transformers.models.marian.configuration_marian.**MarianConfig**(*vocab_size=58101*, *decoder_vocab_size=None*, *max_position_embeddings=1024*, *encoder_layers=12*, *encoder_ffn_dim=4096*, *encoder_attention_heads=16*, *decoder_layers=12*, *decoder_ffn_dim=4096*, *decoder_attention_heads=16*, *encoder_layerdrop=0.0*, *decoder_layerdrop=0.0*, *use_cache=True*, *is_encoder_decoder=True*, *activation_function='gelu'*, *d_model=1024*, *dropout=0.1*, *attention_dropout=0.0*, *activation_dropout=0.0*, *init_std=0.02*, *decoder_start_token_id=58100*, *scale_embedding=False*, *pad_token_id=58100*, *eos_token_id=0*, *forced_eos_token_id=0*, *share_encoder_decoder_embeddings=True*, ***kwargs*)

A framework for translation models, using the same models as BART. Translations should be similar, but not identical to output in the test set linked to in each model card. This model was contributed by sshleifer.

Args:

vocab_size (*int*, *optional*, defaults to 58101):
Vocabulary size of the Marian model. Defines the number of different tokens that can be represented by the *inputs_ids* passed when calling `MarianModel` or `TFMarianModel`.

d_model (*int*, *optional*, defaults to 1024):
Dimensionality of the layers and the pooler layer.

encoder_layers (*int*, *optional*, defaults to 12):
Number of encoder layers.

decoder_layers (*int*, *optional*, defaults to 12):
Number of decoder layers.

encoder_attention_heads (*int*, *optional*, defaults to 16):
Number of attention heads for each attention layer in the Transformer encoder.

decoder_attention_heads (*int*, *optional*, defaults to 16):
Number of attention heads for each attention layer in the Transformer decoder.

decoder_ffn_dim (*int*, *optional*, defaults to 4096):
Dimensionality of the "intermediate" (often named feed-forward) layer in decoder.

encoder_ffn_dim (*int*, *optional*, defaults to 4096):
Dimensionality of the "intermediate" (often named feed-forward) layer in decoder.

activation_function (*str* or *function*, *optional*, defaults to *"gelu"*):
The non-linear activation function (function or string) in the encoder and pooler. If string, *"gelu"*, *"relu"*, *"silu"* and *"gelu_new"* are supported.

dropout (*float*, *optional*, defaults to 0.1):
The dropout probability for all fully connected layers in the embeddings, encoder, and pooler.

attention_dropout (*float*, *optional*, defaults to 0.0):
The dropout ratio for the attention probabilities.

activation_dropout (*float*, *optional*, defaults to 0.0):
The dropout ratio for activations inside the fully connected layer.

max_position_embeddings (*int*, *optional*, defaults to 1024):
The maximum sequence length that this model might ever be used with. Typically set this to something large just in case (e.g., 512 or 1024 or 2048).

init_std (*float*, *optional*, defaults to 0.02):
The standard deviation of the truncated_normal_initializer for initializing all weight matrices.

encoder_layerdrop (*float*, *optional*, defaults to 0.0):
The LayerDrop probability for the encoder. See the LayerDrop paper for more details.

decoder_layerdrop (*float*, *optional*, defaults to 0.0):
The LayerDrop probability for the decoder. See the LayerDrop paper for more details.

scale_embedding (*bool*, *optional*, defaults to *False*):
Scale embeddings by diving by sqrt(d_model).

use_cache (*bool*, *optional*, defaults to *True*):
Whether or not the model should return the last key/values attentions (not used by all models)

forced_eos_token_id (*int*, *optional*, defaults to 0):
The id of the token to force as the last generated token when *max_length* is reached. Usually set to *eos_token_id*.

class transformers.models.markuplm.configuration_markuplm.**MarkupLMConfig**(*vocab_size=30522*, *hidden_size=768*, *num_hidden_layers=12*, *num_attention_heads=12*, *intermediate_size=3072*, *hidden_act='gelu'*, *hidden_dropout_prob=0.1*, *attention_probs_dropout_prob=0.1*, *max_position_embeddings=512*, *type_vocab_size=2*, *initializer_range=0.02*, *layer_norm_eps=1e-12*, *pad_token_id=0*, *bos_token_id=0*, *eos_token_id=2*, *max_xpath_tag_unit_embeddings=256*, *max_xpath_subs_unit_embeddings=102*, *tag_pad_id=216*, *subs_pad_id=1001*, *xpath_unit_hidden_size=32*, *max_depth=50*, *position_embedding_type='absolute'*, *use_cache=True*, *classifier_dropout=None*, ***kwargs*)

The MarkupLM model was proposed in MarkupLM: Pre-training of Text and Markup Language for Visually-rich Document Understanding by Junlong Li, Yiheng Xu, Lei Cui, Furu Wei. MarkupLM is BERT, but applied to HTML pages instead of raw text documents. The model incorporates additional embedding layers to improve performance, similar to LayoutLM.

The model can be used for tasks like question answering on web pages or information extraction from web pages. It obtains state-of-the-art results on 2 important benchmarks: - WebSRC, a dataset for Web-Based Structural Reading Comprehension (a bit like SQuAD but for web pages) - SWDE, a dataset for information extraction from web pages (basically named-entity recogntion on web pages)

The abstract from the paper is the following:

Multimodal pre-training with text, layout, and image has made significant progress for Visually-rich Document Understanding (VrDU), especially the fixed-layout documents such as scanned document images. While, there are still a large number of digital documents where the layout information is not fixed and needs to be interactively and dynamically rendered for visualization, making existing layout-based pre-training approaches not easy to apply. In this paper, we propose MarkupLM for document understanding tasks with markup languages as the backbone such as HTML/XML-based documents, where text and markup information is jointly pre-trained. Experiment results show that the pre-trained MarkupLM significantly outperforms the existing strong baseline models on several document understanding tasks. The pre-trained model and code will be publicly available.

Tips: - In addition to *input_ids*, ~`MarkupLMModel.forward` expects 2 additional inputs, namely *xpath_tags_seq* and *xpath_subs_seq*. These are the XPATH tags and subscripts respectively for each token in the input sequence. - One can use `MarkupLMProcessor` to prepare all data for the model. Refer to the *usage guide* for more info. - Demo notebooks can be found here.

<small> MarkupLM architecture. Taken from the original paper. </small>

This model was contributed by nielsr. The original code can be found here.

Args:

vocab_size (*int*, *optional*, defaults to 30522):
Vocabulary size of the MarkupLM model. Defines the different tokens that can be represented by the *inputs_ids* passed to the forward method of `MarkupLMModel`.

hidden_size (*int*, *optional*, defaults to 768):
Dimensionality of the encoder layers and the pooler layer.

num_hidden_layers (*int*, *optional*, defaults to 12):
Number of hidden layers in the Transformer encoder.

num_attention_heads (*int*, *optional*, defaults to 12):
Number of attention heads for each attention layer in the Transformer encoder.

intermediate_size (*int*, *optional*, defaults to 3072):
Dimensionality of the "intermediate" (i.e., feed-forward) layer in the Transformer encoder.

hidden_act (*str* or *function*, *optional*, defaults to *"gelu"*):
The non-linear activation function (function or string) in the encoder and pooler. If string, *"gelu"*, *"relu"*, *"silu"* and *"gelu_new"* are supported.

hidden_dropout_prob (*float*, *optional*, defaults to 0.1):
The dropout probability for all fully connected layers in the embeddings, encoder, and pooler.

attention_probs_dropout_prob (*float*, *optional*, defaults to 0.1):
The dropout ratio for the attention probabilities.

max_position_embeddings (*int*, *optional*, defaults to 512):
The maximum sequence length that this model might ever be used with. Typically set this to something large just in case (e.g., 512 or 1024 or 2048).

type_vocab_size (*int*, *optional*, defaults to 2):
The vocabulary size of the *token_type_ids* passed into `MarkupLMModel`.

initializer_range (*float*, *optional*, defaults to 0.02):
The standard deviation of the truncated_normal_initializer for initializing all weight matrices.

layer_norm_eps (*float*, *optional*, defaults to 1e-12):
The epsilon used by the layer normalization layers.

max_tree_id_unit_embeddings (*int*, *optional*, defaults to 1024):
The maximum value that the tree id unit embedding might ever use. Typically set this to something large just in case (e.g., 1024).

max_xpath_tag_unit_embeddings (*int*, *optional*, defaults to 256):
The maximum value that the xpath tag unit embedding might ever use. Typically set this to something large just in case (e.g., 256).

max_xpath_subs_unit_embeddings (*int*, *optional*, defaults to 1024):
The maximum value that the xpath subscript unit embedding might ever use. Typically set this to something large just in case (e.g., 1024).

tag_pad_id (*int*, *optional*, defaults to 216):
The id of the padding token in the xpath tags.

subs_pad_id (*int*, *optional*, defaults to 1001):
The id of the padding token in the xpath subscripts.

xpath_tag_unit_hidden_size (*int*, *optional*, defaults to 32):
The hidden size of each tree id unit. One complete tree index will have (50*xpath_tag_unit_hidden_size)-dim.

max_depth (*int*, *optional*, defaults to 50):
The maximum depth in xpath.

`class transformers.models.mbart.configuration_mbart.MBartConfig(`*vocab_size=50265*, *max_position_embeddings=1024*, *encoder_layers=12*, *encoder_ffn_dim=4096*, *encoder_attention_heads=16*, *decoder_layers=12*, *decoder_ffn_dim=4096*, *decoder_attention_heads=16*, *encoder_layerdrop=0.0*, *decoder_layerdrop=0.0*, *use_cache=True*, *is_encoder_decoder=True*, *activation_function='gelu'*, *d_model=1024*, *dropout=0.1*, *attention_dropout=0.0*, *activation_dropout=0.0*, *init_std=0.02*, *classifier_dropout=0.0*, *scale_embedding=False*, *pad_token_id=1*, *bos_token_id=0*, *eos_token_id=2*, *forced_eos_token_id=2*, ***kwargs*)

of MBart

The MBart model was presented in Multilingual Denoising Pre-training for Neural Machine Translation by Yinhan Liu, Jiatao Gu, Naman Goyal, Xian Li, Sergey Edunov Marjan Ghazvininejad, Mike Lewis, Luke Zettlemoyer.

According to the abstract, MBART is a sequence-to-sequence denoising auto-encoder pretrained on large-scale monolingual corpora in many languages using the BART objective. mBART is one of the first methods for pretraining a complete sequence-to-sequence model by denoising full texts in multiple languages, while previous approaches have focused only on the encoder, decoder, or reconstructing parts of the text.

This model was contributed by valhalla. The Authors' code can be found here

#Args:

vocab_size (*int*, *optional*, defaults to 50265):
Vocabulary size of the MBART model. Defines the number of different tokens that can be represented by the *inputs_ids* passed when calling `MBartModel` or `TFMBartModel`.

d_model (*int*, *optional*, defaults to 1024):
Dimensionality of the layers and the pooler layer.

encoder_layers (*int*, *optional*, defaults to 12):
Number of encoder layers.

decoder_layers (*int*, *optional*, defaults to 12):
Number of decoder layers.

encoder_attention_heads (*int*, *optional*, defaults to 16):
Number of attention heads for each attention layer in the Transformer encoder.

decoder_attention_heads (*int*, *optional*, defaults to 16):
Number of attention heads for each attention layer in the Transformer decoder.

decoder_ffn_dim (*int*, *optional*, defaults to 4096):
Dimensionality of the "intermediate" (often named feed-forward) layer in decoder.

encoder_ffn_dim (*int*, *optional*, defaults to 4096):
Dimensionality of the "intermediate" (often named feed-forward) layer in decoder.

activation_function (*str* or *function*, *optional*, defaults to *"gelu"*):
The non-linear activation function (function or string) in the encoder and pooler. If string, *"gelu"*, *"relu"*, *"silu"* and *"gelu_new"* are supported.

dropout (*float*, *optional*, defaults to 0.1):
The dropout probability for all fully connected layers in the embeddings, encoder, and pooler.

attention_dropout (*float*, *optional*, defaults to 0.0):
The dropout ratio for the attention probabilities.

activation_dropout (*float*, *optional*, defaults to 0.0):
The dropout ratio for activations inside the fully connected layer.

classifier_dropout (*float*, *optional*, defaults to 0.0):
The dropout ratio for classifier.

max_position_embeddings (*int*, *optional*, defaults to 1024):
The maximum sequence length that this model might ever be used with. Typically set this to something large just in case (e.g., 512 or 1024 or 2048).

init_std (*float*, *optional*, defaults to 0.02):
The standard deviation of the truncated_normal_initializer for initializing all weight matrices.

encoder_layerdrop (*float*, *optional*, defaults to 0.0):
The LayerDrop probability for the encoder. See the LayerDrop paper for more details.

decoder_layerdrop (*float*, *optional*, defaults to 0.0):
The LayerDrop probability for the decoder. See the LayerDrop paper for more details.

scale_embedding (*bool*, *optional*, defaults to *False*):
Scale embeddings by diving by sqrt(d_model).

use_cache (*bool*, *optional*, defaults to *True*):
Whether or not the model should return the last key/values attentions (not used by all models)

forced_eos_token_id (*int*, *optional*, defaults to 2):
The id of the token to force as the last generated token when *max_length* is reached. Usually set to *eos_token_id*.

class transformers.models.mega.configuration_mega.**MegaConfig**(*vocab_size=30522*, *hidden_size=128*, *num_hidden_layers=4*, *intermediate_size=256*, *ema_projection_size=16*, *bidirectional=True*, *shared_representation_size=64*, *use_chunking=False*, *chunk_size=-1*, *truncation=None*, *normalize_before_mega=True*, *normalization_type='scalenorm'*, *norm_affine=True*, *activation='silu'*, *attention_activation='softmax'*, *dropout_prob=0.1*, *hidden_dropout_prob=0.1*, *attention_probs_dropout_prob=0.1*, *use_feature_dropout=False*, *use_normalized_ffn=True*, *nffn_hidden_size=256*, *normalize_before_ffn=True*, *nffn_activation_dropout_prob=0.1*, *max_positions=2048*, *add_token_type_embeddings=False*, *type_vocab_size=2*, *initializer_range=0.02*, *ema_delta_alpha_range=0.2*, *ema_beta_range=0.02*, *ema_gamma_omega_range=1.0*, *pad_token_id=1*, *bos_token_id=0*, *eos_token_id=2*, *relative_positional_bias='rotary'*, *classifier_dropout=None*, *use_cache=True*, *add_lm_hidden_dense_layer=True*, ***kwargs*)

The MEGA model was proposed in Mega: Moving Average Equipped Gated Attention by Xuezhe Ma, Chunting Zhou, Xiang Kong, Junxian He, Liangke Gui, Graham Neubig, Jonathan May, and Luke Zettlemoyer. MEGA proposes a new approach to self-attention with each encoder layer having a multi-headed exponential moving average in addition to a single head of standard dot-product attention, giving the attention mechanism stronger positional biases. This allows MEGA to perform competitively to Transformers on standard benchmarks including LRA while also having significantly fewer parameters. MEGA's compute efficiency allows it to scale to very long sequences, making it an attractive option for long-document NLP tasks.

The abstract from the paper is the following:

> *The design choices in the Transformer attention mechanism, including weak inductive bias and quadratic computational complexity, have limited its application for modeling long sequences. In this paper, we introduce Mega, a simple, theoretically grounded, single-head gated attention mechanism equipped with (exponential) moving average to incorporate inductive bias of position-aware local dependencies into the position-agnostic attention mechanism. We further propose a variant of Mega that offers linear time and space complexity yet yields only minimal quality loss, by efficiently splitting the whole sequence into multiple chunks with fixed length. Extensive experiments on a wide range of sequence modeling benchmarks, including the Long Range Arena, neural machine translation, auto-regressive language modeling, and image and speech classification, show that Mega achieves significant improvements over other sequence

models, including variants of Transformers and recent state space models. *

Tips:

- MEGA can perform quite well with relatively few parameters. See Appendix D in the MEGA paper for examples of architectural specs which perform well in various settings. If using MEGA as a decoder, be sure to set *bidirectional=False* to avoid errors with default bidirectional.
- Mega-chunk is a variant of mega that reduces time and spaces complexity from quadratic to linear. Utilize chunking with MegaConfig.use_chunking and control chunk size with MegaConfig.chunk_size

This model was contributed by mnaylor. The original code can be found here.

Implementation Notes:

- The original implementation of MEGA had an inconsistent expectation of attention masks for padding and causal self-attention between the softmax attention and Laplace/squared ReLU method. This implementation addresses that inconsistency.
- The original implementation did not include token type embeddings; this implementation adds support for these, with the option controlled by MegaConfig.add_token_type_embeddings

Args:

vocab_size (*int*, *optional*, defaults to 30522):
Vocabulary size of the Mega model. Defines the number of different tokens that can be represented by the *inputs_ids* passed when calling `MegaModel`.

hidden_size (*int*, *optional*, defaults to 128):
Dimensionality of the encoder layers and the pooler layer.

num_hidden_layers (*int*, *optional*, defaults to 4):
Number of hidden layers in the Mega encoder.

intermediate_size (*int*, *optional*, defaults to 256):
Dimensionality of the hidden size (self-attention value projection) within the Mega encoder

ema_projection_size (*int*, *optional*, defaults to 16):
Dimensionality of the MegaMultiDimensionDampedEma

bidirectional (*bool*, *optional*, defaults to *True*):
Whether the MegaMultiDimensionDampedEma used in Mega's self-attention should work bidirectionally (*True*) or unidirectionally (*False*). Bidirectional EMA is incompatible with causal decoding, so this should be False if you intend to use the model as a decoder.

shared_representation_size (*int*, *optional*, defaults to 64):
Dimensionality of the linear projection for shared representation of self-attention queries and keys

use_chunking (*bool*, *optional*, defaults to *False*):
Whether to chunk inputs for linear self-attention complexity (described as Mega-chunk in the paper)

chunk_size (*int*, *optional*, defaults to -1):
If *use_chunking* is set to *True*, determines the size of the chunks to apply to the input sequence. If chunking is used, input sequences must be padded to a multiple of *chunk_size*

truncation (*int*, *optional*):
If specified, the sequence length for which to truncate MegaMultiDimensionDampedEma

normalize_before_mega (*bool*, *optional*, defaults to *True*):
Whether to normalize before (*True*) or after (*False*) passing through Mega encoder blocks

normalization_type (*str*, *optional*, defaults to *"scalenorm"*):
Type of normalization to use in Mega encoder blocks. Choose one of *"scalenorm"*, *"layernorm"*, *"rmsnorm"*, *"batchnorm"*, or *"syncbatchnorm"* (GPU required for syncbatchnorm)

norm_affine (*bool*, *optional*, defaults to *True*):
If *True*, applies a parameterized affine transformation to inputs during normalization

activation (*str*, *optional*, defaults to *"silu"*):
Activation function to apply within Mega encoder blocks. Choose one of *"silu"*, *"relu"*, *"linear"*, *"gelu"*, or *"gelu_accurate"*

attention_activation (*str*, *optional*, defaults to *"softmax"*):
Activation function to apply for single-headed self-attention (a la Transformer). Choose one of *"softmax"*, *"laplace"*, or *"relu2"*

dropout_prob (*float*, *optional*, defaults to 0.1):
The dropout probability for EMA self-attention

hidden_dropout_prob (*float*, *optional*, defaults to 0.1):
The dropout probability for all fully connected layers in the embeddings, encoder, and pooler.

attention_probs_dropout_prob (*float*, *optional*, defaults to 0.1):
The dropout ratio for the attention probabilities.

use_feature_dropout (*bool*, *optional*, defaults to *False*):
Whether to use feature-based (*True*) or standard dropout (*False*)

use_normalized_ffn (*bool*, *optional*, defaults to *True*):
Whether to use the normalized feed-forward sub-layer in Mega blocks (*True*) or pass Mega encoder output as-is (*False*)

nffn_hidden_size (*int*, *optional*, defaults to 256):
If using the normalized feed-forward network (NFFN) layer within Mega (*use_normalized_ffn* = *True*), this is the hidden size of the NFFN

normalize_before_ffn (*bool*, *optional*, defaults to *True*):
Whether to normalize before (*True*) or after (*False*) the feed-forward portion of NFFN

nffn_activation_dropout_prob (*float*, *optional*, defaults to 0.1):
The dropout ratio for the NFFN component.

max_positions (*int*, *optional*, defaults to 2048):
The maximum sequence length to use for positional representations. For *"simple"* relative positional bias, this is a hard limit on input length; *"rotary"* relative positional bias will extrapolate to longer sequences

add_token_type_embeddings (*bool*, *optional*, defaults to *True*):
Whether to account for token types in embeddings. Left as optional to maintain compatibility with original implementation while adding support for token types.

type_vocab_size (*int*, *optional*, defaults to 2):
The vocabulary size of the *token_type_ids* passed when calling `MegaModel`. Only used if *add_token_type_embeddings* = *True*

initializer_range (*float*, *optional*, defaults to 0.02):
The standard deviation of the truncated_normal_initializer for initializing all weight matrices.

ema_delta_alpha_range (*float*, *optional*, defaults to 0.2):
The standard deviation for initializing the delta (damping factor) and alpha (decay factor) parameters in MegaMultiDimensionDampedEma.

ema_beta_range (*float*, *optional*, defaults to 0.02):
The standard deviation for initializing the beta parameter (expansion matrix) in MegaMultiDimensionDampedEma.

ema_gamma_omega_range (*float*, *optional*, defaults to 1.0):
The standard deviation for initializing the gamma (projection matrix) and omega (residual weight) parameters in MultiDimensionEMA.

relative_positional_bias (*str*, *optional*, defaults to *"rotary"*):
Type of relative positional encoding. Choose one of *"rotary"* or *"simple"*. If *"simple"* is selected, *max_positions* is used as a limit on input size, while *"rotary"* extrapolates beyond *max_positions*.

is_decoder (*bool*, *optional*, defaults to *False*):
Whether the model is used as a decoder or not. If *False*, the model is used as an encoder.

use_cache (*bool*, *optional*, defaults to *True*):
Whether or not the model should return the last key/values attentions (not used by all models). Only relevant if *config.is_decoder=True*.

classifier_dropout (*float*, *optional*):
The dropout ratio for the classification head.

add_lm_hidden_dense_layer (*bool*, *optional*, defaults to *True*):
Whether to include a hidden layer for projection between encoder outputs and LM heads (*True*) or pass hidden states directly to LM head (*False*). Remains optional for compatibility with original implementation

`class transformers.models.megatron_bert.configuration_megatron_bert.`**`MegatronBertConfig`**(*vocab_size=29056*, *hidden_size=1024*, *num_hidden_layers=* *num_attention_heads* *intermediate_size=4096*, *hidden_act='gelu'*, *hidden_dropout_prob=0* *attention_probs_dropout_* *max_position_embed* *type_vocab_size=2*, *initializer_range=0.02*, *layer_norm_eps=1e-12*, *pad_token_id=0*, *position_embedding_type* *use_cache=True*, ***kwargs*)

The MegatronBERT model was proposed in Megatron-LM: Training Multi-Billion Parameter Language Models Using Model Parallelism by Mohammad Shoeybi, Mostofa Patwary, Raul Puri, Patrick LeGresley, Jared Casper and Bryan Catanzaro.

The abstract from the paper is the following:

Recent work in language modeling demonstrates that training large transformer models advances the state of the art in Natural Language Processing applications. However, very large models can be quite difficult to train due to memory constraints. In this work, we present our techniques for training very large transformer models and implement a simple, efficient intra-layer model parallel approach that enables training transformer models with billions of parameters. Our approach does not require a new compiler or library changes, is orthogonal and complimentary to pipeline model parallelism, and can be fully implemented with the insertion of a few communication operations in native PyTorch. We illustrate this approach by converging transformer based models up to 8.3 billion parameters using 512 GPUs. We sustain 15.1 PetaFLOPs across the entire application with 76% scaling efficiency when compared to a strong single GPU baseline that sustains 39 TeraFLOPs, which is 30% of peak FLOPs. To demonstrate that large language models can further advance the state of the art (SOTA), we train an 8.3 billion parameter transformer language model similar to GPT-2 and a 3.9 billion parameter model similar to BERT. We show that careful attention to the placement of layer normalization in BERT-like models is critical to achieving increased performance as the model size grows. Using the GPT-2 model we achieve SOTA results on the WikiText103 (10.8 compared to SOTA perplexity of 15.8) and LAMBADA (66.5% compared to SOTA accuracy of 63.2%) datasets. Our BERT model achieves SOTA results on the RACE dataset (90.9% compared to SOTA accuracy of 89.4%).

Tips:

We have provided pretrained BERT-345M checkpoints for use to evaluate or finetuning downstream tasks.

To access these checkpoints, first sign up for and setup the NVIDIA GPU Cloud (NGC) Registry CLI. Further documentation for downloading models can be found in the NGC documentation.

Alternatively, you can directly download the checkpoints using:

BERT-345M-uncased:

`bash wget --content-disposition https://api.ngc.nvidia.com/v2/models/nvidia/megatron_bert_345m/versions/v0.1_uncased/zip -O megatron_bert_345m_v0_1_uncased.zip `

BERT-345M-cased:

`bash wget --content-disposition https://api.ngc.nvidia.com/v2/models/nvidia/megatron_bert_345m/versions/v0.1_cased/zip -O megatron_bert_345m_v0_1_cased.zip `

Once you have obtained the checkpoints from NVIDIA GPU Cloud (NGC), you have to convert them to a format that will easily be loaded by Hugging Face Transformers and our port of the BERT code.

The following commands allow you to do the conversion. We assume that the folder *models/megatron_bert* contains *megatron_bert_345m_v0_1_{cased, uncased}.zip* and that the commands are run from inside that folder:

`bash python3 $PATH_TO_TRANSFORMERS/models/megatron_bert/convert_megatron_bert_checkpoint.py megatron_bert_345m_v0_1_uncased.zip `

`bash python3 $PATH_TO_TRANSFORMERS/models/megatron_bert/convert_megatron_bert_checkpoint.py megatron_bert_345m_v0_1_cased.zip `

This model was contributed by jdemouth. The original code can be found here. That repository contains a multi-GPU and multi-node implementation of the Megatron Language models. In particular, it contains a hybrid model parallel approach using "tensor parallel" and "pipeline parallel" techniques.

Args:

vocab_size (*int*, *optional*, defaults to 29056):
Vocabulary size of the MEGATRON_BERT model. Defines the number of different tokens that can be represented by the *inputs_ids* passed when calling `MegatronBertModel`.

hidden_size (*int*, *optional*, defaults to 1024):
Dimensionality of the encoder layers and the pooler layer.

num_hidden_layers (*int*, *optional*, defaults to 24):
Number of hidden layers in the Transformer encoder.

num_attention_heads (*int*, *optional*, defaults to 16):
Number of attention heads for each attention layer in the Transformer encoder.

intermediate_size (*int*, *optional*, defaults to 4096):
Dimensionality of the "intermediate" (often named feed-forward) layer in the Transformer encoder.

hidden_act (*str* or *Callable*, *optional*, defaults to *"gelu"*):
The non-linear activation function (function or string) in the encoder and pooler. If string, *"gelu"*, *"relu"*, *"silu"* and *"gelu_new"* are supported.

hidden_dropout_prob (*float*, *optional*, defaults to 0.1):
The dropout probability for all fully connected layers in the embeddings, encoder, and pooler.

attention_probs_dropout_prob (*float*, *optional*, defaults to 0.1):
The dropout ratio for the attention probabilities.

max_position_embeddings (*int*, *optional*, defaults to 512):
The maximum sequence length that this model might ever be used with. Typically set this to something large just in case (e.g., 512 or 1024 or 2048).

type_vocab_size (*int*, *optional*, defaults to 2):
The vocabulary size of the *token_type_ids* passed when calling `MegatronBertModel`.

initializer_range (*float*, *optional*, defaults to 0.02):
The standard deviation of the truncated_normal_initializer for initializing all weight matrices.

layer_norm_eps (*float*, *optional*, defaults to 1e-12):
The epsilon used by the layer normalization layers.

position_embedding_type (*str*, *optional*, defaults to *"absolute"*):
Type of position embedding. Choose one of *"absolute"*, *"relative_key"*, *"relative_key_query"*. For positional embeddings use *"absolute"*. For more information on *"relative_key"*, please refer to Self-Attention with Relative Position Representations (Shaw et al.). For more information on *"relative_key_query"*, please refer to *Method 4* in Improve Transformer Models with Better Relative Position Embeddings (Huang et al.).

is_decoder (*bool*, *optional*, defaults to *False*):
Whether the model is used as a decoder or not. If *False*, the model is used as an encoder.

use_cache (*bool*, *optional*, defaults to *True*):
Whether or not the model should return the last key/values attentions (not used by all models). Only relevant if *config.is_decoder=True*.

class transformers.models.mixtral.configuration_mixtral.**MixtralConfig**(*vocab_size=32000*, *hidden_size=4096*, *intermediate_size=14336*, *num_hidden_layers=32*, *num_attention_heads=32*, *num_key_value_heads=8*, *hidden_act='silu'*, *max_position_embeddings=131072*, *initializer_range=0.02*, *rms_norm_eps=1e-05*, *use_cache=True*, *pad_token_id=None*, *bos_token_id=1*, *eos_token_id=2*, *tie_word_embeddings=False*, *rope_theta=1000000.0*, *sliding_window=None*, *attention_dropout=0.0*, *num_experts_per_tok=2*, *num_local_experts=8*, *output_router_logits=False*, *router_aux_loss_coef=0.001*, ***kwargs*)

Mixtral-8x7B is Mistral AI's second Large Language Model (LLM).

The Mixtral model was proposed by the Mistral AI team.

It was introduced in the Mixtral of Experts blogpost with the following introduction:

Today, the team is proud to release Mixtral 8x7B, a high-quality sparse mixture of experts models (SMoE) with open weights. Licensed under Apache 2.0. Mixtral outperforms Llama 2 70B on most benchmarks with 6x faster inference. It is the strongest open-weight model with a permissive license and the best model overall regarding cost/performance trade-offs. In particular, it matches or outperforms GPT3.5 on most standard benchmarks.

Tips:

- The model needs to be converted using the conversion script.
- If the model is quantized to 4bits, a single A100 is enough to fit the entire 45B model.

This model was contributed by Younes Belkada and Arthur Zucker . The original code can be found here.

#Args:

vocab_size (*int*, *optional*, defaults to 32000):
Vocabulary size of the Mixtral model. Defines the number of different tokens that can be represented by the *inputs_ids* passed when calling `MixtralModel`

hidden_size (*int*, *optional*, defaults to 4096):
Dimension of the hidden representations.

intermediate_size (*int*, *optional*, defaults to 14336):
Dimension of the MLP representations.

num_hidden_layers (*int*, *optional*, defaults to 32):
Number of hidden layers in the Transformer encoder.

num_attention_heads (*int*, *optional*, defaults to 32):
Number of attention heads for each attention layer in the Transformer encoder.

num_key_value_heads (*int*, *optional*, defaults to 8):
This is the number of key_value heads that should be used to implement Grouped Query Attention. If *num_key_value_heads=num_attention_heads*, the model will use Multi Head Attention (MHA), if num_key_value_heads=1 the model will use Multi Query Attention (MQA) otherwise GQA is used. When converting a multi-head checkpoint to a GQA checkpoint, each group key and value head should be constructed by meanpooling all the original heads within that group. For more details checkout `this paper. If it is not specified, will default to *8*.

hidden_act (*str* or *function*, *optional*, defaults to *"silu"*):
The non-linear activation function (function or string) in the decoder.

max_position_embeddings (*int*, *optional*, defaults to *4096*32*):
The maximum sequence length that this model might ever be used with. Mixtral's sliding window attention allows sequence of up to 4096*32 tokens.

initializer_range (*float*, *optional*, defaults to 0.02):
The standard deviation of the truncated_normal_initializer for initializing all weight matrices.

rms_norm_eps (*float*, *optional*, defaults to 1e-05):
The epsilon used by the rms normalization layers.

use_cache (*bool*, *optional*, defaults to *True*):
Whether or not the model should return the last key/values attentions (not used by all models). Only relevant if *config.is_decoder=True*.

pad_token_id (*int*, *optional*):
The id of the padding token.

bos_token_id (*int*, *optional*, defaults to 1):
The id of the "beginning-of-sequence" token.

eos_token_id (*int*, *optional*, defaults to 2):
The id of the "end-of-sequence" token.

tie_word_embeddings (*bool*, *optional*, defaults to *False*):
Whether the model's input and output word embeddings should be tied.

rope_theta (*float*, *optional*, defaults to 1000000.0):
The base period of the RoPE embeddings.

sliding_window (*int*, *optional*):
Sliding window attention window size. If not specified, will default to *4096*.

attention_dropout (*float*, *optional*, defaults to 0.0):
The dropout ratio for the attention probabilities.

num_experts_per_tok (*int*, *optional*, defaults to 2):
The number of experts to root per-token, can be also interpreted as the *top-p* routing parameter

num_local_experts (*int*, *optional*, defaults to 8):
Number of experts per Sparse MLP layer.

output_router_logits (*bool*, *optional*, defaults to *False*):
Whether or not the router logits should be returned by the model. Enabeling this will also allow the model to output the auxiliary loss. See **`here <>`__** for more details

router_aux_loss_coef (*float*, *optional*, defaults to 0.001):
The aux loss factor for the total loss.

```
>>> from transformers import MixtralModel, MixtralConfig
```

```
>>> # Initializing a Mixtral 7B style configuration
>>> configuration = MixtralConfig()
```

```
>>> # Initializing a model from the Mixtral 7B style configuration
>>> model = MixtralModel(configuration)
```

```
>>> # Accessing the model configuration
>>> configuration = model.config
```

class transformers.models.mobilebert.configuration_mobilebert.**MobileBertConfig**(*vocab_size=30522*, *hidden_size=512*, *num_hidden_layers=24*, *num_attention_heads=4*, *intermediate_size=512*, *hidden_act='relu'*, *hidden_dropout_prob=0.0*, *attention_probs_dropout_prob=0.1*, *max_position_embeddings=512* *type_vocab_size=2*, *initializer_range=0.02*, *layer_norm_eps=1e-12*, *pad_token_id=0*, *embedding_size=128*, *trigram_input=True*, *use_bottleneck=True*, *intra_bottleneck_size=128*, *use_bottleneck_attention=False* *key_query_shared_bottleneck=* *num_feedforward_networks=4*, *normalization_type='no_norm'*, *classifier_activation=True*, *classifier_dropout=None*, ***kwargs*)

The MobileBERT model was proposed in MobileBERT: a Compact Task-Agnostic BERT for Resource-Limited Devices by Zhiqing Sun, Hongkun Yu, Xiaodan Song, Renjie Liu, Yiming Yang, and Denny Zhou. It's a bidirectional transformer based on the BERT model, which is compressed and accelerated using several approaches.

The abstract from the paper is the following:

Natural Language Processing (NLP) has recently achieved great success by using huge pre-trained models with hun-

dreds of millions of parameters. However, these models suffer from heavy model sizes and high latency such that they cannot be deployed to resource-limited mobile devices. In this paper, we propose MobileBERT for compressing and accelerating the popular BERT model. Like the original BERT, MobileBERT is task-agnostic, that is, it can be generically applied to various downstream NLP tasks via simple fine-tuning. Basically, MobileBERT is a thin version of BERT_LARGE, while equipped with bottleneck structures and a carefully designed balance between self-attentions and feed-forward networks. To train MobileBERT, we first train a specially designed teacher model, an inverted-bottleneck incorporated BERT_LARGE model. Then, we conduct knowledge transfer from this teacher to MobileBERT. Empirical studies show that MobileBERT is 4.3x smaller and 5.5x faster than BERT_BASE while achieving competitive results on well-known benchmarks. On the natural language inference tasks of GLUE, MobileBERT achieves a GLUEscore o 77.7 (0.6 lower than BERT_BASE), and 62 ms latency on a Pixel 4 phone. On the SQuAD v1.1/v2.0 question answering task, MobileBERT achieves a dev F1 score of 90.0/79.2 (1.5/2.1 higher than BERT_BASE).

Tips:

- MobileBERT is a model with absolute position embeddings so it's usually advised to pad the inputs on the right rather than the left.
- MobileBERT is similar to BERT and therefore relies on the masked language modeling (MLM) objective. It is therefore efficient at predicting masked tokens and at NLU in general, but is not optimal for text generation. Models trained with a causal language modeling (CLM) objective are better in that regard.

This model was contributed by vshampor. The original code can be found here.

Args:

vocab_size (*int*, *optional*, defaults to 30522):
Vocabulary size of the MobileBERT model. Defines the number of different tokens that can be represented by the *inputs_ids* passed when calling `MobileBertModel` or `TFMobileBertModel`.

hidden_size (*int*, *optional*, defaults to 512):
Dimensionality of the encoder layers and the pooler layer.

num_hidden_layers (*int*, *optional*, defaults to 24):
Number of hidden layers in the Transformer encoder.

num_attention_heads (*int*, *optional*, defaults to 4):
Number of attention heads for each attention layer in the Transformer encoder.

intermediate_size (*int*, *optional*, defaults to 512):
Dimensionality of the "intermediate" (often named feed-forward) layer in the Transformer encoder.

hidden_act (*str* or *function*, *optional*, defaults to *"relu"*):
The non-linear activation function (function or string) in the encoder and pooler. If string, *"gelu"*, *"relu"*, *"silu"* and *"gelu_new"* are supported.

hidden_dropout_prob (*float*, *optional*, defaults to 0.0):
The dropout probability for all fully connected layers in the embeddings, encoder, and pooler.

attention_probs_dropout_prob (*float*, *optional*, defaults to 0.1):
The dropout ratio for the attention probabilities.

max_position_embeddings (*int*, *optional*, defaults to 512):
The maximum sequence length that this model might ever be used with. Typically set this to something large just in case (e.g., 512 or 1024 or 2048).

type_vocab_size (*int*, *optional*, defaults to 2):
The vocabulary size of the *token_type_ids* passed when calling `MobileBertModel` or `TFMobileBertModel`.

initializer_range (*float*, *optional*, defaults to 0.02):
The standard deviation of the truncated_normal_initializer for initializing all weight matrices.

layer_norm_eps (*float*, *optional*, defaults to 1e-12):
The epsilon used by the layer normalization layers.

pad_token_id (*int*, *optional*, defaults to 0):
The ID of the token in the word embedding to use as padding.

embedding_size (*int*, *optional*, defaults to 128):
The dimension of the word embedding vectors.

trigram_input (*bool*, *optional*, defaults to *True*):
Use a convolution of trigram as input.

use_bottleneck (*bool*, *optional*, defaults to *True*):
Whether to use bottleneck in BERT.

intra_bottleneck_size (*int*, *optional*, defaults to 128):
Size of bottleneck layer output.

use_bottleneck_attention (*bool*, *optional*, defaults to *False*):
Whether to use attention inputs from the bottleneck transformation.

key_query_shared_bottleneck (*bool*, *optional*, defaults to *True*):
Whether to use the same linear transformation for query&key in the bottleneck.

num_feedforward_networks (*int*, *optional*, defaults to 4):
Number of FFNs in a block.

normalization_type (*str*, *optional*, defaults to *"no_norm"*):
The normalization type in MobileBERT.

classifier_dropout (*float*, *optional*):
The dropout ratio for the classification head.

`class transformers.models.mpnet.configuration_mpnet.MPNetConfig(`*vocab_size=30527*, *hidden_size=768*, *num_hidden_layers=12*, *num_attention_heads=12*, *intermediate_size=3072*, *hidden_act='gelu'*, *hidden_dropout_prob=0.1*, *attention_probs_dropout_prob=0.1*, *max_position_embeddings=512*, *initializer_range=0.02*, *layer_norm_eps=1e-12*, *relative_attention_num_buckets=32*, *pad_token_id=1*, *bos_token_id=0*, *eos_token_id=2*, ***kwargs*`)`

The MPNet model was proposed in MPNet: Masked and Permuted Pre-training for Language Understanding by Kaitao Song, Xu Tan, Tao Qin, Jianfeng Lu, Tie-Yan Liu.

MPNet adopts a novel pre-training method, named masked and permuted language modeling, to inherit the advantages of masked language modeling and permuted language modeling for natural language understanding.

The abstract from the paper is the following:

BERT adopts masked language modeling (MLM) for pre-training and is one of the most successful pre-training models. Since BERT neglects dependency among predicted tokens, XLNet introduces permuted language modeling (PLM) for pre-training to address this problem. However, XLNet does not leverage the full position information of a sentence and

thus suffers from position discrepancy between pre-training and fine-tuning. In this paper, we propose MPNet, a novel pre-training method that inherits the advantages of BERT and XLNet and avoids their limitations. MPNet leverages the dependency among predicted tokens through permuted language modeling (vs. MLM in BERT), and takes auxiliary position information as input to make the model see a full sentence and thus reducing the position discrepancy (vs. PLM in XLNet). We pre-train MPNet on a large-scale dataset (over 160GB text corpora) and fine-tune on a variety of down-streaming tasks (GLUE, SQuAD, etc). Experimental results show that MPNet outperforms MLM and PLM by a large margin, and achieves better results on these tasks compared with previous state-of-the-art pre-trained methods (e.g., BERT, XLNet, RoBERTa) under the same model setting.

Tips:

- MPNet doesn't have *token_type_ids*, you don't need to indicate which token belongs to which segment. just separate your segments with the separation token *tokenizer.sep_token* (or ``sep]`).

The original code can be found [here <https://github.com/microsoft/MPNet>`__.

Args:

vocab_size (*int*, *optional*, defaults to 30527):
Vocabulary size of the MPNet model. Defines the number of different tokens that can be represented by the *inputs_ids* passed when calling `MPNetModel` or `TFMPNetModel`.

hidden_size (*int*, *optional*, defaults to 768):
Dimensionality of the encoder layers and the pooler layer.

num_hidden_layers (*int*, *optional*, defaults to 12):
Number of hidden layers in the Transformer encoder.

num_attention_heads (*int*, *optional*, defaults to 12):
Number of attention heads for each attention layer in the Transformer encoder.

intermediate_size (*int*, *optional*, defaults to 3072):
Dimensionality of the "intermediate" (often named feed-forward) layer in the Transformer encoder.

hidden_act (*str* or *Callable*, *optional*, defaults to *"gelu"*):
The non-linear activation function (function or string) in the encoder and pooler. If string, *"gelu"*, *"relu"*, *"silu"* and *"gelu_new"* are supported.

hidden_dropout_prob (*float*, *optional*, defaults to 0.1):
The dropout probability for all fully connected layers in the embeddings, encoder, and pooler.

attention_probs_dropout_prob (*float*, *optional*, defaults to 0.1):
The dropout ratio for the attention probabilities.

max_position_embeddings (*int*, *optional*, defaults to 512):
The maximum sequence length that this model might ever be used with. Typically set this to something large just in case (e.g., 512 or 1024 or 2048).

initializer_range (*float*, *optional*, defaults to 0.02):
The standard deviation of the truncated_normal_initializer for initializing all weight matrices.

layer_norm_eps (*float*, *optional*, defaults to 1e-12):
The epsilon used by the layer normalization layers.

relative_attention_num_buckets (*int*, *optional*, defaults to 32):
The number of buckets to use for each attention layer.

class transformers.models.mpt.configuration_mpt.**MptConfig**(*d_model: int = 2048*, *n_heads: int = 16*, *n_layers: int = 24*, *expansion_ratio: int = 4*, *max_seq_len: int = 2048*, *vocab_size: int = 50368*, *resid_pdrop: float = 0.0*, *layer_norm_epsilon: float = 1e-05*, *emb_pdrop: float = 0.0*, *learned_pos_emb: bool = True*, *attn_config: transformers.models.mpt.configuration_mpt.MptAttentionConfig = None*, *init_device: str = 'cpu'*, *logit_scale: float | str | NoneType = None*, *no_bias: bool = True*, *verbose: int = 0*, *embedding_fraction: float = 1.0*, *norm_type: str = 'low_precision_layernorm'*, *use_cache: bool = False*, *initializer_range=0.02*, ***kwargs*)

The MPT model was proposed by the MosaicML team and released with multiple sizes and finetuned variants. The MPT models is a series of open source and commercially usable LLMs pre-trained on 1T tokens.

MPT models are GPT-style decoder-only transformers with several improvements: performance-optimized layer implementations, architecture changes that provide greater training stability, and the elimination of context length limits by replacing positional embeddings with ALiBi.

- MPT base: MPT base pre-trained models on next token prediction
- MPT instruct: MPT base models fine-tuned on instruction based tasks
- MPT storywriter: MPT base models fine-tuned for 2500 steps on 65k-token excerpts of fiction books contained in the books3 corpus, this enables the model to handle very long sequences

The original code is available at the ``llm-foundry``(https://github.com/mosaicml/llm-foundry/tree/main) repository.

Read more about it in the release blogpost

Tips:

- Learn more about some techniques behind training of the model in this section of llm-foundry repository
- If you want to use the advanced version of the model (triton kernels, direct flash attention integration), you can still use the original model implementation by adding *trust_remote_code=True* when calling *from_pretrained*.
- Fine-tuning Notebook on how to fine-tune MPT-7B on a free Google Colab instance to turn the model into a Chatbot.

Args:

d_model (*int*, *optional*, defaults to 2048):
Dimensionality of the embeddings and hidden states.

n_heads (*int*, *optional*, defaults to 16):
Number of attention heads for each attention layer in the Transformer encoder.

n_layers (*int*, *optional*, defaults to 24):
Number of hidden layers in the Transformer encoder.

expansion_ratio (*int*, *optional*, defaults to 4):
The ratio of the up/down scale in the MLP.

max_seq_len (*int*, *optional*, defaults to 2048):
The maximum sequence length of the model.

vocab_size (*int*, *optional*, defaults to 50368):
Vocabulary size of the Mpt model. Defines the maximum number of different tokens that can be represented by the *inputs_ids* passed when calling `MptModel`. Check this discussion on how the *vocab_size* has been defined.

resid_pdrop (*float*, *optional*, defaults to 0.0):
The dropout probability applied to the attention output before combining with residual.

layer_norm_epsilon (*float*, *optional*, defaults to 1e-05):
The epsilon to use in the layer normalization layers.

emb_pdrop (*float*, *optional*, defaults to 0.0):
The dropout probability for the embedding layer.

learned_pos_emb (*bool*, *optional*, defaults to *True*):
Whether to use learned positional embeddings.

attn_config (*dict*, *optional*):
A dictionary used to configure the model's attention module.

init_device (*str*, *optional*, defaults to *"cpu"*):
The device to use for parameter initialization. Defined for backward compatibility

logit_scale (*float*, *optional*):
If not None, scale the logits by this value.

no_bias (*bool*, *optional*, defaults to *True*):
Whether to use bias in all linear layers.

verbose (*int*, *optional*, defaults to 0):
The verbosity level to use for logging. Used in the previous versions of MPT models for logging. This argument is deprecated.

embedding_fraction (*float*, *optional*, defaults to 1.0):
The fraction to scale the gradients of the embedding layer by.

norm_type (*str*, *optional*, defaults to *"low_precision_layernorm"*):
Type of layer norm to use. All MPT models uses the same layer norm implementation. Defined for backward compatibility.

use_cache (*bool*, *optional*, defaults to *False*):
Whether or not the model should return the last key/values attentions (not used by all models).

initializer_range (*float*, *optional*, defaults to 0.02):
The standard deviation of the truncated_normal_initializer for initializing all weight matrices.

class transformers.models.mra.configuration_mra.**MraConfig**(*vocab_size=50265*, *hidden_size=768*, *num_hidden_layers=12*, *num_attention_heads=12*, *intermediate_size=3072*, *hidden_act='gelu'*, *hidden_dropout_prob=0.1*, *attention_probs_dropout_prob=0.1*, *max_position_embeddings=512*, *type_vocab_size=1*, *initializer_range=0.02*, *layer_norm_eps=1e-05*, *position_embedding_type='absolute'*, *block_per_row=4*, *approx_mode='full'*, *initial_prior_first_n_blocks=0*, *initial_prior_diagonal_n_blocks=0*, *pad_token_id=1*, *bos_token_id=0*, *eos_token_id=2*, ***kwargs*)

The MRA model was proposed in Multi Resolution Analysis (MRA) for Approximate Self-Attention by Zhanpeng Zeng, Sourav Pal, Jeffery Kline, Glenn M Fung, and Vikas Singh.

The abstract from the paper is the following:

Transformers have emerged as a preferred model for many tasks in natural langugage processing and vision. Recent efforts on training and deploying Transformers more efficiently have identified many strategies to approximate the self-attention matrix, a key module in a Transformer architecture. Effective ideas include various prespecified sparsity patterns, low-rank basis expansions and combinations thereof. In this paper, we revisit classical Multiresolution Analysis (MRA) concepts such as Wavelets, whose potential value in this setting remains underexplored thus far. We show that simple approximations based on empirical feedback and design choices informed by modern hardware and implementation challenges, eventually yield a MRA-based approach for self-attention with an excellent performance profile across most criteria of interest. We undertake an extensive set of experiments and demonstrate that this multi-resolution scheme outperforms most efficient self-attention proposals and is favorable for both short and long sequences. Code is available at https://github.com/mlpen/mra-attention.

This model was contributed by novice03. The original code can be found here.

Args:

vocab_size (*int*, *optional*, defaults to 50265):
Vocabulary size of the Mra model. Defines the number of different tokens that can be represented by the *inputs_ids* passed when calling `MraModel`.

hidden_size (*int*, *optional*, defaults to 768):
Dimension of the encoder layers and the pooler layer.

num_hidden_layers (*int*, *optional*, defaults to 12):
Number of hidden layers in the Transformer encoder.

num_attention_heads (*int*, *optional*, defaults to 12):
Number of attention heads for each attention layer in the Transformer encoder.

intermediate_size (*int*, *optional*, defaults to 3072):
Dimension of the "intermediate" (i.e., feed-forward) layer in the Transformer encoder.

hidden_act (*str* or *function*, *optional*, defaults to *"gelu"*):
The non-linear activation function (function or string) in the encoder and pooler. If string, *"gelu"*, *"relu"*, *"selu"* and *"gelu_new"* are supported.

hidden_dropout_prob (*float*, *optional*, defaults to 0.1):
The dropout probability for all fully connected layers in the embeddings, encoder, and pooler.

attention_probs_dropout_prob (*float*, *optional*, defaults to 0.1):
The dropout ratio for the attention probabilities.

max_position_embeddings (*int*, *optional*, defaults to 512):
The maximum sequence length that this model might ever be used with. Typically set this to something large just in case (e.g., 512 or 1024 or 2048).

type_vocab_size (*int*, *optional*, defaults to 1):
The vocabulary size of the *token_type_ids* passed when calling `MraModel`.

initializer_range (*float*, *optional*, defaults to 0.02):
The standard deviation of the truncated_normal_initializer for initializing all weight matrices.

layer_norm_eps (*float*, *optional*, defaults to 1e-5):
The epsilon used by the layer normalization layers.

position_embedding_type (*str*, *optional*, defaults to *"absolute"*):
Type of position embedding. Choose one of *"absolute"*, *"relative_key"*, *"relative_key_query"*.

block_per_row (*int*, *optional*, defaults to 4):
Used to set the budget for the high resolution scale.

approx_mode (*str*, *optional*, defaults to *"full"*):
Controls whether both low and high resolution approximations are used. Set to *"full"* for both low and high resolution and *"sparse"* for only low resolution.

initial_prior_first_n_blocks (*int*, *optional*, defaults to 0):
The initial number of blocks for which high resolution is used.

initial_prior_diagonal_n_blocks (*int*, *optional*, defaults to 0):
The number of diagonal blocks for which high resolution is used.

`class transformers.models.mvp.configuration_mvp.`**`MvpConfig`**(*vocab_size=50267*, *max_position_embeddings=1024*, *encoder_layers=12*, *encoder_ffn_dim=4096*, *encoder_attention_heads=16*, *decoder_layers=12*, *decoder_ffn_dim=4096*, *decoder_attention_heads=16*, *encoder_layerdrop=0.0*, *decoder_layerdrop=0.0*, *activation_function='gelu'*, *d_model=1024*, *dropout=0.1*, *attention_dropout=0.0*, *activation_dropout=0.0*, *init_std=0.02*, *classifier_dropout=0.0*, *scale_embedding=False*, *use_cache=True*, *pad_token_id=1*, *bos_token_id=0*, *eos_token_id=2*, *is_encoder_decoder=True*, *decoder_start_token_id=2*, *forced_eos_token_id=2*, *use_prompt=False*, *prompt_length=100*, *prompt_mid_dim=800*, ***kwargs*)

The MVP model was proposed in MVP: Multi-task Supervised Pre-training for Natural Language Generation by Tianyi Tang, Junyi Li, Wayne Xin Zhao and Ji-Rong Wen.

According to the abstract,

- MVP follows a standard Transformer encoder-decoder architecture.
- MVP is supervised pre-trained using labeled datasets.
- MVP also has task-specific soft prompts to stimulate the model's capacity in performing a certain task.
- MVP is specially designed for natural language generation and can be adapted to a wide range of generation tasks, including but not limited to summarization, data-to-text generation, open-ended dialogue system, story generation, question answering, question generation, task-oriented dialogue system, commonsense generation, paraphrase generation, text style transfer, and text simplification. Our model can also be adapted to natural language understanding tasks such as sequence classification and (extractive) question answering.

Tips: - We have released a series of models here, including MVP, MVP with task-specific prompts, and multi-task pre-trained variants. - If you want to use a model without prompts (standard Transformer), you can load it through *MvpForConditionalGeneration.from_pretrained('RUCAIBox/mvp')*. - If you want to use a model with task-specific prompts, such as summarization, you can load it through *MvpForConditionalGeneration.from_pretrained('RUCAIBox/mvp-summarization')*. - Our model supports lightweight prompt tuning following Prefix-tuning with method *set_lightweight_tuning()*.

This model was contributed by Tianyi Tang. The detailed information and instructions can be found here.

Args:

vocab_size (*int*, *optional*, defaults to 50267):
Vocabulary size of the MVP model. Defines the number of different tokens that can be represented by the *inputs_ids* passed when calling `MvpModel`.

d_model (*int*, *optional*, defaults to 1024):
Dimensionality of the layers and the pooler layer.

encoder_layers (*int*, *optional*, defaults to 12):
Number of encoder layers.

decoder_layers (*int*, *optional*, defaults to 12):
Number of decoder layers.

encoder_attention_heads (*int*, *optional*, defaults to 16):
Number of attention heads for each attention layer in the Transformer encoder.

decoder_attention_heads (*int*, *optional*, defaults to 16):
Number of attention heads for each attention layer in the Transformer decoder.

decoder_ffn_dim (*int*, *optional*, defaults to 4096):
Dimensionality of the "intermediate" (often named feed-forward) layer in decoder.

encoder_ffn_dim (*int*, *optional*, defaults to 4096):
Dimensionality of the "intermediate" (often named feed-forward) layer in decoder.

activation_function (*str* or *function*, *optional*, defaults to *"gelu"*):
The non-linear activation function (function or string) in the encoder and pooler. If string, *"gelu"*, *"relu"*, *"silu"* and *"gelu_new"* are supported.

dropout (*float*, *optional*, defaults to 0.1):
The dropout probability for all fully connected layers in the embeddings, encoder, and pooler.

attention_dropout (*float*, *optional*, defaults to 0.0):
The dropout ratio for the attention probabilities.

activation_dropout (*float*, *optional*, defaults to 0.0):
The dropout ratio for activations inside the fully connected layer.

classifier_dropout (*float*, *optional*, defaults to 0.0):
The dropout ratio for classifier.

max_position_embeddings (*int*, *optional*, defaults to 1024):
The maximum sequence length that this model might ever be used with. Typically set this to something large just in case (e.g., 512 or 1024 or 2048).

init_std (*float*, *optional*, defaults to 0.02):
The standard deviation of the truncated_normal_initializer for initializing all weight matrices.

encoder_layerdrop (*float*, *optional*, defaults to 0.0):
The LayerDrop probability for the encoder. See the LayerDrop paper for more details.

decoder_layerdrop (*float*, *optional*, defaults to 0.0):
The LayerDrop probability for the decoder. See the LayerDrop paper for more details.

scale_embedding (*bool*, *optional*, defaults to *False*):
Scale embeddings by diving by sqrt(d_model).

use_cache (*bool*, *optional*, defaults to *True*):
Whether or not the model should return the last key/values attentions (not used by all models).

forced_eos_token_id (*int*, *optional*, defaults to 2):
The id of the token to force as the last generated token when *max_length* is reached. Usually set to *eos_token_id*.

use_prompt (*bool*, *optional*, defaults to *False*):
Whether or not to use prompt.

prompt_length (*int*, *optional*, defaults to 100):
The length of prompt.

prompt_mid_dim (*int*, *optional*, defaults to 800):
Dimensionality of the "intermediate" layer in prompt.

class transformers.models.nezha.configuration_nezha.**NezhaConfig**(*vocab_size=21128*, *hidden_size=768*, *num_hidden_layers=12*, *num_attention_heads=12*, *intermediate_size=3072*, *hidden_act='gelu'*, *hidden_dropout_prob=0.1*, *attention_probs_dropout_prob=0.1*, *max_position_embeddings=512*, *max_relative_position=64*, *type_vocab_size=2*, *initializer_range=0.02*, *layer_norm_eps=1e-12*, *classifier_dropout=0.1*, *pad_token_id=0*, *bos_token_id=2*, *eos_token_id=3*, *use_cache=True*, ***kwargs*)

The Nezha model was proposed in NEZHA: Neural Contextualized Representation for Chinese Language Understanding by Junqiu Wei et al.

The abstract from the paper is the following:

The pre-trained language models have achieved great successes in various natural language understanding (NLU) tasks due to its capacity to capture the deep contextualized information in text by pre-training on large-scale corpora. In this technical report, we present our practice of pre-training language models named NEZHA (NEural contextualiZed

representation for CHinese lAnguage understanding) on Chinese corpora and finetuning for the Chinese NLU tasks. The current version of NEZHA is based on BERT with a collection of proven improvements, which include Functional Relative Positional Encoding as an effective positional encoding scheme, Whole Word Masking strategy, Mixed Precision Training and the LAMB Optimizer in training the models. The experimental results show that NEZHA achieves the state-of-the-art performances when finetuned on several representative Chinese tasks, including named entity recognition (People's Daily NER), sentence matching (LCQMC), Chinese sentiment classification (ChnSenti) and natural language inference (XNLI).

This model was contributed by sijunhe. The original code can be found here.

Args:

vocab_size (*int*, optional, defaults to 21128):
Vocabulary size of the NEZHA model. Defines the different tokens that can be represented by the *inputs_ids* passed to the forward method of `NezhaModel`.

hidden_size (*int*, optional, defaults to 768):
Dimensionality of the encoder layers and the pooler layer.

num_hidden_layers (*int*, optional, defaults to 12):
Number of hidden layers in the Transformer encoder.

num_attention_heads (*int*, optional, defaults to 12):
Number of attention heads for each attention layer in the Transformer encoder.

intermediate_size (*int*, optional, defaults to 3072):
The dimensionality of the "intermediate" (i.e., feed-forward) layer in the Transformer encoder.

hidden_act (*str* or *function*, optional, defaults to "gelu"):
The non-linear activation function (function or string) in the encoder and pooler.

hidden_dropout_prob (*float*, optional, defaults to 0.1):
The dropout probability for all fully connected layers in the embeddings, encoder, and pooler.

attention_probs_dropout_prob (*float*, optional, defaults to 0.1):
The dropout ratio for the attention probabilities.

max_position_embeddings (*int*, optional, defaults to 512):
The maximum sequence length that this model might ever be used with. Typically set this to something large (e.g., 512 or 1024 or 2048).

type_vocab_size (*int*, optional, defaults to 2):
The vocabulary size of the *token_type_ids* passed into `NezhaModel`.

initializer_range (*float*, optional, defaults to 0.02):
The standard deviation of the truncated_normal_initializer for initializing all weight matrices.

layer_norm_eps (*float*, optional, defaults to 1e-12):
The epsilon used by the layer normalization layers.

classifier_dropout (*float*, optional, defaults to 0.1):
The dropout ratio for attached classifiers.

is_decoder (*bool*, *optional*, defaults to *False*):
Whether the model is used as a decoder or not. If *False*, the model is used as an encoder.

class transformers.models.nllb_moe.configuration_nllb_moe.**NllbMoeConfig**(*vocab_size=128112*, *max_position_embeddings=1024*, *encoder_layers=12*, *encoder_ffn_dim=4096*, *encoder_attention_heads=16*, *decoder_layers=12*, *decoder_ffn_dim=4096*, *decoder_attention_heads=16*, *encoder_layerdrop=0.05*, *decoder_layerdrop=0.05*, *use_cache=True*, *is_encoder_decoder=True*, *activation_function='relu'*, *d_model=1024*, *dropout=0.1*, *attention_dropout=0.1*, *activation_dropout=0.0*, *init_std=0.02*, *decoder_start_token_id=2*, *scale_embedding=True*, *router_bias=False*, *router_dtype='float32'*, *router_ignore_padding_tokens=False*, *num_experts=128*, *expert_capacity=64*, *encoder_sparse_step=4*, *decoder_sparse_step=4*, *router_z_loss_coef=0.001*, *router_aux_loss_coef=0.001*, *second_expert_policy='all'*, *normalize_router_prob_before_dropping=False* *batch_prioritized_routing=False*, *moe_eval_capacity_token_fraction=1.0*, *moe_token_dropout=0.2*, *pad_token_id=1*, *bos_token_id=0*, *eos_token_id=2*, *output_router_logits=False*, ***kwargs*)

The NLLB model was presented in No Language Left Behind: Scaling Human-Centered Machine Translation by Marta R. Costa-jussà, James Cross, Onur Çelebi, Maha Elbayad, Kenneth Heafield, Kevin Heffernan, Elahe Kalbassi, Janice

Lam, Daniel Licht, Jean Maillard, Anna Sun, Skyler Wang, Guillaume Wenzek, Al Youngblood, Bapi Akula, Loic Barrault, Gabriel Mejia Gonzalez, Prangthip Hansanti, John Hoffman, Semarley Jarrett, Kaushik Ram Sadagopan, Dirk Rowe, Shannon Spruit, Chau Tran, Pierre Andrews, Necip Fazil Ayan, Shruti Bhosale, Sergey Edunov, Angela Fan, Cynthia Gao, Vedanuj Goswami, Francisco Guzmán, Philipp Koehn, Alexandre Mourachko, Christophe Ropers, Safiyyah Saleem, Holger Schwenk, and Jeff Wang.

The abstract of the paper is the following:

Driven by the goal of eradicating language barriers on a global scale, machine translation has solidified itself as a key focus of artificial intelligence research today. However, such efforts have coalesced around a small subset of languages, leaving behind the vast majority of mostly low-resource languages. What does it take to break the 200 language barrier while ensuring safe, high quality results, all while keeping ethical considerations in mind? In No Language Left Behind, we took on this challenge by first contextualizing the need for low-resource language translation support through exploratory interviews with native speakers. Then, we created datasets and models aimed at narrowing the performance gap between low and high-resource languages. More specifically, we developed a conditional compute model based on Sparsely Gated Mixture of Experts that is trained on data obtained with novel and effective data mining techniques tailored for low-resource languages. We propose multiple architectural and training improvements to counteract overfitting while training on thousands of tasks. Critically, we evaluated the performance of over 40,000 different translation directions using a human-translated benchmark, Flores-200, and combined human evaluation with a novel toxicity benchmark covering all languages in Flores-200 to assess translation safety. Our model achieves an improvement of 44% BLEU relative to the previous state-of-the-art, laying important groundwork towards realizing a universal translation system.

Tips:

- M2M100ForConditionalGeneration is the base model for both NLLB and NLLB MoE
- The NLLB-MoE is very similar to the NLLB model, but it's feed forward layer is based on the implementation of SwitchTransformers.
- The tokenizer is the same as the NLLB models.

This model was contributed by Arthur Zucker. The original code can be found here.

Args:

vocab_size (*int*, *optional*, defaults to 50265):
Vocabulary size of the NllbMoe model. Defines the number of different tokens that can be represented by the *inputs_ids* passed when calling `NllbMoeModel` or

d_model (*int*, *optional*, defaults to 1024):
Dimensionality of the layers and the pooler layer.

encoder_layers (*int*, *optional*, defaults to 12):
Number of encoder layers.

decoder_layers (*int*, *optional*, defaults to 12):
Number of decoder layers.

encoder_attention_heads (*int*, *optional*, defaults to 16):
Number of attention heads for each attention layer in the Transformer encoder.

decoder_attention_heads (*int*, *optional*, defaults to 16):
Number of attention heads for each attention layer in the Transformer decoder.

decoder_ffn_dim (*int*, *optional*, defaults to 4096):
Dimensionality of the "intermediate" (often named feed-forward) layer in decoder.

encoder_ffn_dim (*int*, *optional*, defaults to 4096):
Dimensionality of the "intermediate" (often named feed-forward) layer in encoder.

activation_function (*str* or *function*, *optional*, defaults to *"gelu"*):
The non-linear activation function (function or string) in the encoder and pooler. If string, *"gelu"*, *"relu"*, *"silu"* and *"gelu_new"* are supported.

dropout (*float*, *optional*, defaults to 0.1):
The dropout probability for all fully connected layers in the embeddings, encoder, and pooler.

attention_dropout (*float*, *optional*, defaults to 0.0):
The dropout ratio for the attention probabilities.

activation_dropout (*float*, *optional*, defaults to 0.0):
The dropout ratio for activations inside the fully connected layer.

classifier_dropout (*float*, *optional*, defaults to 0.0):
The dropout ratio for classifier.

max_position_embeddings (*int*, *optional*, defaults to 1024):
The maximum sequence length that this model might ever be used with. Typically set this to something large just in case (e.g., 512 or 1024 or 2048).

init_std (*float*, *optional*, defaults to 0.02):
The standard deviation of the truncated_normal_initializer for initializing all weight matrices.

encoder_layerdrop (*float*, *optional*, defaults to 0.0):
The LayerDrop probability for the encoder. See the LayerDrop paper for more details.

decoder_layerdrop (*float*, *optional*, defaults to 0.0):
The LayerDrop probability for the decoder. See the LayerDrop paper for more details.

second_expert_policy (*str*, *optional*, default to *"all"*):
The policy used for the sampling the probability of being sampled to a second expert for each token.

normalize_router_prob_before_dropping (*bool*, *optional*, defaults to *True*):
Whether or not to normalize the router probabilities before applying a mask based on the experts capacity (capacity dropping).

batch_prioritized_routing (*bool*, *optional*, defaults to *True*):
Whether or not to orders the tokens by their router probabilities before capacity dropping. This means that the tokens that have the highest probabilities will be routed before other tokens that might be further in the sequence.

moe_eval_capacity_token_fraction (*float*, *optional*, defaults to 1.0):
Fraction of tokens as capacity during validation, if set to negative, uses the same as training. Should be in range: (0.0, 1.0].

num_experts (*int*, *optional*, defaults to 128):
Number of experts for each NllbMoeSparseMlp layer.

expert_capacity (*int*, *optional*, defaults to 64):
Number of tokens that can be stored in each expert.

encoder_sparse_step (*int*, *optional*, defaults to 4):
Frequency of the sparse layers in the encoder. 4 means that one out of 4 layers will be sparse.

decoder_sparse_step (*int*, *optional*, defaults to 4):
Frequency of the sparse layers in the decoder. 4 means that one out of 4 layers will be sparse.

router_dtype (*str*, *optional*, default to *"float32"*):
The *dtype* used for the routers. It is preferable to keep the *dtype* to *"float32"* as specified in the *selective precision* discussion in the paper.

router_ignore_padding_tokens (*bool*, *optional*, defaults to *False*):
Whether to ignore padding tokens when routing. if *False*, the padding tokens are not routed to any experts.

router_bias (*bool*, *optional*, defaults to *False*):
Whether or not the classifier of the router should have a bias.

moe_token_dropout (*float*, *optional*, defualt ot 0.2):
Masking rate for MoE expert output masking (EOM), which is implemented via a Dropout2d on the expert outputs.

output_router_logits (*bool*, *optional*, defaults to *False*):
Whether or not to return the router logits. Only set to *True* to get the auxiliary loss when training.

use_cache (*bool*, *optional*, defaults to *True*):
Whether or not the model should return the last key/values attentions (not used by all models).

`class transformers.models.nystromformer.configuration_nystromformer.`**`NystromformerConfig`**(*vocab_size=30000*, *hidden_size=768*, *num_hidden_layers=* *num_attention_head* *intermediate_size=3072*, *hidden_act='gelu_new'* *hidden_dropout_prob=* *attention_probs_dropout* *max_position_embe* *type_vocab_size=2*, *segment_means_seq_le* *num_landmarks=64* *conv_kernel_size=6* *inv_coeff_init_optio* *initializer_range=0.02*, *layer_norm_eps=1e-05*, *pad_token_id=1*, *bos_token_id=0*, *eos_token_id=2*, ***kwargs*)

The Nyströmformer model was proposed in *Nyströmformer: A Nyström-Based Algorithm for Approximating Self-Attention* by Yunyang Xiong, Zhanpeng Zeng, Rudrasis Chakraborty, Mingxing Tan, Glenn Fung, Yin Li, and Vikas Singh.

The abstract from the paper is the following:

Transformers have emerged as a powerful tool for a broad range of natural language processing tasks. A key component that drives the impressive performance of Transformers is the self-attention mechanism that encodes the influence or dependence of other tokens on each specific token. While beneficial, the quadratic complexity of self-attention on the input sequence length has limited its application to longer sequences – a topic being actively studied in the community.

To address this limitation, we propose Nyströmformer – a model that exhibits favorable scalability as a function of sequence length. Our idea is based on adapting the Nyström method to approximate standard self-attention with O(n) complexity. The scalability of Nyströmformer enables application to longer sequences with thousands of tokens. We perform evaluations on multiple downstream tasks on the GLUE benchmark and IMDB reviews with standard sequence length, and find that our Nyströmformer performs comparably, or in a few cases, even slightly better, than standard self-attention. On longer sequence tasks in the Long Range Arena (LRA) benchmark, Nyströmformer performs favorably relative to other efficient self-attention methods. Our code is available at this https URL.

This model was contributed by novice03. The original code can be found here.

Args:

vocab_size (*int*, *optional*, defaults to 30000):
Vocabulary size of the Nystromformer model. Defines the number of different tokens that can be represented by the *inputs_ids* passed when calling `NystromformerModel`.

hidden_size (*int*, *optional*, defaults to 768):
Dimension of the encoder layers and the pooler layer.

num_hidden_layers (*int*, *optional*, defaults to 12):
Number of hidden layers in the Transformer encoder.

num_attention_heads (*int*, *optional*, defaults to 12):
Number of attention heads for each attention layer in the Transformer encoder.

intermediate_size (*int*, *optional*, defaults to 3072):
Dimension of the "intermediate" (i.e., feed-forward) layer in the Transformer encoder.

hidden_act (*str* or *function*, *optional*, defaults to *"gelu"*):
The non-linear activation function (function or string) in the encoder and pooler. If string, *"gelu"*, *"relu"*, *"selu"* and *"gelu_new"* are supported.

hidden_dropout_prob (*float*, *optional*, defaults to 0.1):
The dropout probability for all fully connected layers in the embeddings, encoder, and pooler.

attention_probs_dropout_prob (*float*, *optional*, defaults to 0.1):
The dropout ratio for the attention probabilities.

max_position_embeddings (*int*, *optional*, defaults to 512):
The maximum sequence length that this model might ever be used with. Typically set this to something large just in case (e.g., 512 or 1024 or 2048).

type_vocab_size (*int*, *optional*, defaults to 2):
The vocabulary size of the *token_type_ids* passed when calling `NystromformerModel`.

segment_means_seq_len (*int*, *optional*, defaults to 64):
Sequence length used in segment-means.

num_landmarks (*int*, *optional*, defaults to 64):
The number of landmark (or Nystrom) points to use in Nystrom approximation of the softmax self-attention matrix.

conv_kernel_size (*int*, *optional*, defaults to 65):
The kernel size of depthwise convolution used in Nystrom approximation.

inv_coeff_init_option (*bool*, *optional*, defaults to *False*):
Whether or not to use exact coefficient computation for the initial values for the iterative method of calculating the Moore-Penrose inverse of a matrix.

initializer_range (*float*, *optional*, defaults to 0.02):
The standard deviation of the truncated_normal_initializer for initializing all weight matrices.

layer_norm_eps (*float*, *optional*, defaults to 1e-12):
The epsilon used by the layer normalization layers.

class transformers.models.openai.configuration_openai.**OpenAIGPTConfig**(*vocab_size=40478*, *n_positions=512*, *n_embd=768*, *n_layer=12*, *n_head=12*, *afn='gelu'*, *resid_pdrop=0.1*, *embd_pdrop=0.1*, *attn_pdrop=0.1*, *layer_norm_epsilon=1e-05*, *initializer_range=0.02*, *summary_type='cls_index'*, *summary_use_proj=True*, *summary_activation=None*, *summary_proj_to_labels=True*, *summary_first_dropout=0.1*, ***kwargs*)

OpenAI GPT model was proposed in Improving Language Understanding by Generative Pre-Training by Alec Radford, Karthik Narasimhan, Tim Salimans and Ilya Sutskever. It's a causal (unidirectional) transformer pre-trained using language modeling on a large corpus will long range dependencies, the Toronto Book Corpus.

The abstract from the paper is the following:

Natural language understanding comprises a wide range of diverse tasks such as textual entailment, question answering, semantic similarity assessment, and document classification. Although large unlabeled text corpora are abundant, labeled data for learning these specific tasks is scarce, making it challenging for discriminatively trained models to perform adequately. We demonstrate that large gains on these tasks can be realized by generative pretraining of a language model on a diverse corpus of unlabeled text, followed by discriminative fine-tuning on each specific task. In contrast to previous approaches, we make use of task-aware input transformations during fine-tuning to achieve effective transfer while requiring minimal changes to the model architecture. We demonstrate the effectiveness of our approach on a wide range of benchmarks for natural language understanding. Our general task-agnostic model outperforms discriminatively trained models that use architectures specifically crafted for each task, significantly improving upon the state of the art in 9 out of the 12 tasks studied.

Tips:

- GPT is a model with absolute position embeddings so it's usually advised to pad the inputs on the right rather than the left.
- GPT was trained with a causal language modeling (CLM) objective and is therefore powerful at predicting the next token in a sequence. Leveraging this feature allows GPT-2 to generate syntactically coherent text as it can be observed in the *run_generation.py* example script.

Write With Transformer is a webapp created and hosted by Hugging Face showcasing the generative capabilities of several models. GPT is one of them.

This model was contributed by thomwolf. The original code can be found here.

Note:

If you want to reproduce the original tokenization process of the *OpenAI GPT* paper, you will need to install *ftfy* and *SpaCy*:

```bash pip install spacy ftfy==4.4.3 python -m spacy download en ```

If you don't install *ftfy* and *SpaCy*, the `OpenAIGPTTokenizer` will default to tokenize using BERT's *BasicTokenizer* followed by Byte-Pair Encoding (which should be fine for most usage, don't worry).

Args:

vocab_size (*int*, *optional*, defaults to 40478):
Vocabulary size of the GPT-2 model. Defines the number of different tokens that can be represented by the *inputs_ids* passed when calling `OpenAIGPTModel` or `TFOpenAIGPTModel`.

n_positions (*int*, *optional*, defaults to 512):
The maximum sequence length that this model might ever be used with. Typically set this to something large just in case (e.g., 512 or 1024 or 2048).

n_embd (*int*, *optional*, defaults to 768):
Dimensionality of the embeddings and hidden states.

n_layer (*int*, *optional*, defaults to 12):
Number of hidden layers in the Transformer encoder.

n_head (*int*, *optional*, defaults to 12):
Number of attention heads for each attention layer in the Transformer encoder.

afn (*str* or *Callable*, *optional*, defaults to *"gelu"*):
The non-linear activation function (function or string) in the encoder and pooler. If string, *"gelu"*, *"relu"*, *"silu"* and *"gelu_new"* are supported.

resid_pdrop (*float*, *optional*, defaults to 0.1):
The dropout probability for all fully connected layers in the embeddings, encoder, and pooler.

embd_pdrop (*int*, *optional*, defaults to 0.1):
The dropout ratio for the embeddings.

attn_pdrop (*float*, *optional*, defaults to 0.1):
The dropout ratio for the attention.

layer_norm_epsilon (*float*, *optional*, defaults to 1e-05):
The epsilon to use in the layer normalization layers

initializer_range (*float*, *optional*, defaults to 0.02):
The standard deviation of the truncated_normal_initializer for initializing all weight matrices.

summary_type (*str*, *optional*, defaults to *"cls_index"*):
Argument used when doing sequence summary, used in the models `OpenAIGPTDoubleHeadsModel` and `OpenAIGPTDoubleHeadsModel`.

Has to be one of the following options:

- *"last"*: Take the last token hidden state (like XLNet).
- *"first"*: Take the first token hidden state (like BERT).
- *"mean"*: Take the mean of all tokens hidden states.
- *"cls_index"*: Supply a Tensor of classification token position (like GPT/GPT-2).
- *"attn"*: Not implemented now, use multi-head attention.

summary_use_proj (*bool*, *optional*, defaults to *True*):
Argument used when doing sequence summary, used in the models `OpenAIGPTDoubleHeadsModel` and `OpenAIGPTDoubleHeadsModel`.
```

Whether or not to add a projection after the vector extraction.

summary_activation (*str*, *optional*):
Argument used when doing sequence summary, used in the models `OpenAIGPTDoubleHeadsModel` and `OpenAIGPTDoubleHeadsModel`.

Pass *"tanh"* for a tanh activation to the output, any other value will result in no activation.

summary_proj_to_labels (*bool*, *optional*, defaults to *True*):
Argument used when doing sequence summary, used in the models `OpenAIGPTDoubleHeadsModel` and `OpenAIGPTDoubleHeadsModel`.

Whether the projection outputs should have *config.num_labels* or *config.hidden_size* classes.

summary_first_dropout (*float*, *optional*, defaults to 0.1):
Argument used when doing sequence summary, used in the models `OpenAIGPTDoubleHeadsModel` and `OpenAIGPTDoubleHeadsModel`.

The dropout ratio to be used after the projection and activation.

class transformers.models.opt.configuration_opt.**OPTConfig**(*vocab_size=50272*, *hidden_size=768*, *num_hidden_layers=12*, *ffn_dim=3072*, *max_position_embeddings=2048*, *do_layer_norm_before=True*, *_remove_final_layer_norm=False*, *word_embed_proj_dim=None*, *dropout=0.1*, *attention_dropout=0.0*, *num_attention_heads=12*, *activation_function='relu'*, *layerdrop=0.0*, *init_std=0.02*, *use_cache=True*, *pad_token_id=1*, *bos_token_id=2*, *eos_token_id=2*, *enable_bias=True*, *layer_norm_elementwise_affine=True*, ***kwargs*)

The OPT model was proposed in Open Pre-trained Transformer Language Models by Meta AI. OPT is a series of open-sourced large causal language models which perform similar in performance to GPT3.

The abstract from the paper is the following:

Large language models, which are often trained for hundreds of thousands of compute days, have shown remarkable capabilities for zero- and few-shot learning. Given their computational cost, these models are difficult to replicate without significant capital. For the few that are available through APIs, no access is granted to the full model weights, making them difficult to study. We present Open Pre-trained Transformers (OPT), a suite of decoder-only pre-trained transformers ranging from 125M to 175B parameters, which we aim to fully and responsibly share with interested researchers. We show that OPT-175B is comparable to GPT-3, while requiring only 1/7th the carbon footprint to develop. We are also releasing our logbook detailing the infrastructure challenges we faced, along with code for experimenting with all of the released models.

Tips: - OPT has the same architecture as `BartDecoder`. - Contrary to GPT2, OPT adds the EOS token </s> to the beginning of every prompt.

This model was contributed by Arthur Zucker, Younes Belkada, and Patrick Von Platen. The original code can be found here.

Args:

vocab_size (*int*, *optional*, defaults to 50272):
Vocabulary size of the OPT model. Defines the number of different tokens that can be represented by the *inputs_ids* passed when calling `OPTModel`

hidden_size (*int*, *optional*, defaults to 768):
Dimensionality of the layers and the pooler layer.

num_hidden_layers (*int*, *optional*, defaults to 12):
Number of decoder layers.

ffn_dim (*int*, *optional*, defaults to 3072):
Dimensionality of the "intermediate" (often named feed-forward) layer in decoder.

num_attention_heads (*int*, *optional*, defaults to 12):
Number of attention heads for each attention layer in the Transformer decoder.

activation_function (*str* or *function*, *optional*, defaults to *"relu"*):
The non-linear activation function (function or string) in the encoder and pooler. If string, *"gelu"*, *"relu"*, *"silu"* and *"gelu_new"* are supported.

max_position_embeddings (*int*, *optional*, defaults to 2048):
The maximum sequence length that this model might ever be used with. Typically set this to something large just in case (e.g., 512 or 1024 or 2048).

do_layer_norm_before (*bool*, *optional*, defaults to *True*):
Whether to perform layer normalization before the attention block.

word_embed_proj_dim (*int*, *optional*):
word_embed_proj_dim can be set to down-project word embeddings, *e.g. opt-350m*. Defaults to *hidden_size*.

dropout (*float*, *optional*, defaults to 0.1):
The dropout probability for all fully connected layers in the embeddings, encoder, and pooler.

attention_dropout (*float*, *optional*, defaults to 0.0):
The dropout ratio for the attention probabilities.

layerdrop (*float*, *optional*, defaults to 0.0):
The LayerDrop probability. See the LayerDrop paper for more details.

init_std (*float*, *optional*, defaults to 0.02):
The standard deviation of the truncated_normal_initializer for initializing all weight matrices.

use_cache (*bool*, *optional*, defaults to *True*):
Whether or not the model should return the last key/values attentions (not used by all models).

enable_bias (*bool*, *optional*, defaults to *True*):
Whether or not if the linear layers in the attention blocks should use the bias term.

layer_norm_elementwise_affine (*bool*, *optional*, defaults to *True*):
Whether or not if the layer norms should have learnable parameters.

class transformers.models.pegasus.configuration_pegasus.**PegasusConfig**(*vocab_size=50265*, *max_position_embeddings=1024*, *encoder_layers=12*, *encoder_ffn_dim=4096*, *encoder_attention_heads=16*, *decoder_layers=12*, *decoder_ffn_dim=4096*, *decoder_attention_heads=16*, *encoder_layerdrop=0.0*, *decoder_layerdrop=0.0*, *use_cache=True*, *is_encoder_decoder=True*, *activation_function='gelu'*, *d_model=1024*, *dropout=0.1*, *attention_dropout=0.0*, *activation_dropout=0.0*, *init_std=0.02*, *decoder_start_token_id=0*, *scale_embedding=False*, *pad_token_id=0*, *eos_token_id=1*, *forced_eos_token_id=1*, ***kwargs*)

The Pegasus model was proposed in PEGASUS: Pre-training with Extracted Gap-sentences for Abstractive Summarization by Jingqing Zhang, Yao Zhao, Mohammad Saleh and Peter J. Liu on Dec 18, 2019.

According to the abstract,

- Pegasus' pretraining task is intentionally similar to summarization: important sentences are removed/masked from an input document and are generated together as one output sequence from the remaining sentences, similar to an extractive summary.
- Pegasus achieves SOTA summarization performance on all 12 downstream tasks, as measured by ROUGE and human eval.

This model was contributed by sshleifer. The Authors' code can be found here.

Tips:

- Sequence-to-sequence model with the same encoder-decoder model architecture as BART. Pegasus is pre-trained jointly on two self-supervised objective functions: Masked Language Modeling (MLM) and a novel summarization specific pretraining objective, called Gap Sentence Generation (GSG).
  - MLM: encoder input tokens are randomly replaced by a mask tokens and have to be predicted by the encoder (like in BERT)
  - GSG: whole encoder input sentences are replaced by a second mask token and fed to the decoder, but which has a causal mask to hide the future words like a regular auto-regressive transformer decoder.

Args:

vocab_size (*int*, *optional*, defaults to 50265):
Vocabulary size of the PEGASUS model. Defines the number of different tokens that can be represented by the *inputs_ids* passed when calling `PegasusModel` or `TFPegasusModel`.

d_model (*int*, *optional*, defaults to 1024):
Dimensionality of the layers and the pooler layer.

encoder_layers (*int*, *optional*, defaults to 12):
Number of encoder layers.

decoder_layers (*int*, *optional*, defaults to 12):
Number of decoder layers.

encoder_attention_heads (*int*, *optional*, defaults to 16):
Number of attention heads for each attention layer in the Transformer encoder.

decoder_attention_heads (*int*, *optional*, defaults to 16):
Number of attention heads for each attention layer in the Transformer decoder.

decoder_ffn_dim (*int*, *optional*, defaults to 4096):
Dimensionality of the "intermediate" (often named feed-forward) layer in decoder.

encoder_ffn_dim (*int*, *optional*, defaults to 4096):
Dimensionality of the "intermediate" (often named feed-forward) layer in decoder.

activation_function (*str* or *function*, *optional*, defaults to *"gelu"*):
The non-linear activation function (function or string) in the encoder and pooler. If string, *"gelu"*, *"relu"*, *"silu"* and *"gelu_new"* are supported.

dropout (*float*, *optional*, defaults to 0.1):
The dropout probability for all fully connected layers in the embeddings, encoder, and pooler.

attention_dropout (*float*, *optional*, defaults to 0.0):
The dropout ratio for the attention probabilities.

activation_dropout (*float*, *optional*, defaults to 0.0):
The dropout ratio for activations inside the fully connected layer.

max_position_embeddings (*int*, *optional*, defaults to 1024):
The maximum sequence length that this model might ever be used with. Typically set this to something large just in case (e.g., 512 or 1024 or 2048).

init_std (*float*, *optional*, defaults to 0.02):
The standard deviation of the truncated_normal_initializer for initializing all weight matrices.

encoder_layerdrop (*float*, *optional*, defaults to 0.0):
The LayerDrop probability for the encoder. See the LayerDrop paper for more details.

decoder_layerdrop (*float*, *optional*, defaults to 0.0):
The LayerDrop probability for the decoder. See the LayerDrop paper for more details.

scale_embedding (*bool*, *optional*, defaults to *False*):
Scale embeddings by diving by sqrt(d_model).

use_cache (*bool*, *optional*, defaults to *True*):
Whether or not the model should return the last key/values attentions (not used by all models)

forced_eos_token_id (*int*, *optional*, defaults to 1):
The id of the token to force as the last generated token when *max_length* is reached. Usually set to *eos_token_id*.

class transformers.models.pegasus_x.configuration_pegasus_x.**PegasusXConfig**(*vocab_size=96103*, *max_position_embeddings=16384*, *encoder_layers=16*, *encoder_ffn_dim=4096*, *encoder_attention_heads=16*, *decoder_layers=16*, *decoder_ffn_dim=4096*, *decoder_attention_heads=16*, *encoder_layerdrop=0.0*, *decoder_layerdrop=0.0*, *use_cache=True*, *is_encoder_decoder=True*, *activation_function='gelu'*, *d_model=1024*, *dropout=0.1*, *attention_dropout=0.0*, *activation_dropout=0.0*, *init_std=0.02*, *decoder_start_token_id=0*, *scale_embedding=True*, *pad_token_id=0*, *eos_token_id=1*, *forced_eos_token_id=1*, *num_global_tokens=32*, *block_size=512*, *stagger_local_blocks=True*, ***kwargs*)

The PEGASUS-X model was proposed in Investigating Efficiently Extending Transformers for Long Input Summarization by Jason Phang, Yao Zhao and Peter J. Liu.

PEGASUS-X (PEGASUS eXtended) extends the PEGASUS models for long input summarization through additional long input pretraining and using staggered block-local attention with global tokens in the encoder.

The abstract from the paper is the following:

While large pretrained Transformer models have proven highly capable at tackling natural language tasks, handling long sequence inputs continues to be a significant challenge. One such task is long input summarization, where inputs are longer than the maximum input context of most pretrained models. Through an extensive set of experiments, we investigate what model architectural changes and pretraining paradigms can most efficiently adapt a pretrained Transformer for long input summarization. We find that a staggered, block-local Transformer with global encoder tokens strikes a good balance of performance and efficiency, and that an additional pretraining phase on long sequences meaningfully improves downstream summarization performance. Based on our findings, we introduce PEGASUS-X, an extension of the PEGASUS model with additional long input pretraining to handle inputs of up to 16K tokens.

PEGASUS-X achieves strong performance on long input summarization tasks comparable with much larger models while adding few additional parameters and not requiring model parallelism to train.

Tips:

- PEGASUS-X uses the same tokenizer as PEGASUS.

This model was contributed by **`zphang <<https://huggingface.co/zphang>`__**. The original code can be found here.

Args:

vocab_size (*int*, *optional*, defaults to 96103):
Vocabulary size of the PEGASUS-X model. Defines the number of different tokens that can be represented by the *inputs_ids* passed when calling `PegasusXModel`.

d_model (*int*, *optional*, defaults to 1024):
Dimension of the layers and the pooler layer.

encoder_layers (*int*, *optional*, defaults to 16):
Number of encoder layers.

decoder_layers (*int*, *optional*, defaults to 16):
Number of decoder layers.

encoder_attention_heads (*int*, *optional*, defaults to 16):
Number of attention heads for each attention layer in the Transformer encoder.

decoder_attention_heads (*int*, *optional*, defaults to 16):
Number of attention heads for each attention layer in the Transformer decoder.

decoder_ffn_dim (*int*, *optional*, defaults to 4096):
Dimension of the "intermediate" (often named feed-forward) layer in decoder.

encoder_ffn_dim (*int*, *optional*, defaults to 4096):
Dimension of the "intermediate" (often named feed-forward) layer in decoder.

activation_function (*str* or *function*, *optional*, defaults to *"gelu"*):
The non-linear activation function (function or string) in the encoder and pooler. If string, *"gelu"*, *"relu"*, *"silu"* and *"gelu_new"* are supported.

dropout (*float*, *optional*, defaults to 0.1):
The dropout probability for all fully connected layers in the embeddings, encoder, and pooler.

attention_dropout (*float*, *optional*, defaults to 0.0):
The dropout ratio for the attention probabilities.

activation_dropout (*float*, *optional*, defaults to 0.0):
The dropout ratio for activations inside the fully connected layer.

max_position_embeddings (*int*, *optional*, defaults to 16384):
The maximum sequence length that this model might ever be used with. Typically set this to something large just in case (e.g., 512 or 1024 or 2048).

init_std (*float*, *optional*, defaults to 0.02):
The standard deviation of the truncated_normal_initializer for initializing all weight matrices.

encoder_layerdrop (*float*, *optional*, defaults to 0.0):
The LayerDrop probability for the encoder. See the LayerDrop paper for more details.

decoder_layerdrop (*float*, *optional*, defaults to 0.0):
The LayerDrop probability for the decoder. See the LayerDrop paper for more details.

use_cache (*bool*, *optional*, defaults to *True*):
Whether or not the model should return the last key/values attentions (not used by all models)

forced_eos_token_id (*int*, *optional*, defaults to 1):
The id of the token to force as the last generated token when *max_length* is reached. Usually set to *eos_token_id*.

num_global_tokens (*int*, *optional*, defaults to 128):
Number of global tokens to use for the encoder

block_size (*int*, *optional*, defaults to 512):
Block size for encoder local attention. Sequence length should be an exact multiple of block size. block_size must be a multiple of 2 if stagger_local_block is True

stagger_local_block (*bool*, *optional*, defaults to *True*):
Whether to stagger every other local attention by half a block

`class` transformers.models.persimmon.configuration_persimmon.**PersimmonConfig**(*vocab_size=262144*, *hidden_size=4096*, *intermediate_size=16384*, *num_hidden_layers=36*, *num_attention_heads=64*, *hidden_act='relu2'*, *max_position_embeddings=16384*, *initializer_range=0.02*, *layer_norm_eps=1e-05*, *use_cache=True*, *tie_word_embeddings=False*, *rope_theta=25000.0*, *rope_scaling=None*, *qk_layernorm=True*, *hidden_dropout=0.0*, *attention_dropout=0.0*, *partial_rotary_factor=0.5*, *pad_token_id=None*, *bos_token_id=1*, *eos_token_id=2*, ***kwargs*)

The Persimmon model was created by ADEPT, and authored by Erich Elsen, Augustus Odena, Maxwell Nye, Sağnak Taşırlar, Tri Dao, Curtis Hawthorne, Deepak Moparthi, Arushi Somani.

The authors introduced Persimmon-8B, a decoder model based on the classic transformers architecture, with query and key normalization. Persimmon-8B is a fully permissively-licensed model with approximately 8 billion parameters, released under the Apache license. Some of the key attributes of Persimmon-8B are long context size (16K), performance, and capabilities for multimodal extensions.

The authors showcase their approach to model evaluation, focusing on practical text generation, mirroring how users interact with language models. The work also includes a comparative analysis, pitting Persimmon-8B against other prominent models (MPT 7B Instruct and Llama 2 Base 7B 1-Shot), across various evaluation tasks. The results demonstrate Persimmon-8B's competitive performance, even with limited training data.

In terms of model details, the work outlines the architecture and training methodology of Persimmon-8B, providing

insights into its design choices, sequence length, and dataset composition. The authors present a fast inference code that outperforms traditional implementations through operator fusion and CUDA graph utilization while maintaining code coherence. They express their anticipation of how the community will leverage this contribution to drive innovation, hinting at further upcoming releases as part of an ongoing series of developments.

<Tip warning={true}>

The *Persimmon* models were trained using *bfloat16*, but the original inference uses *float16* The checkpoints uploaded on the hub use *torch_dtype = 'float16'* which will be used by the *AutoModel* API to cast the checkpoints from *torch.float32* to *torch.float16*.

The *dtype* of the online weights is mostly irrelevant, unless you are using *torch_dtype="auto"* when initializing a model using *model = AutoModelForCausalLM.from_pretrained("path", torch_dtype = "auto").* The reason is that the model will first be downloaded (using the *dtype* of the checkpoints online) then it will be cast to the default *dtype* of *torch* (becomes *torch.float32*). Users should specify the *torch_dtype* they want, and if they don't it will be *torch.float32*.

Finetuning the model in *float16* is not recommended and known to produce *nan*, as such the model should be fine-tuned in *bfloat16*.

</Tip>

Tips:

- To convert the model, you need to clone the original repository using *git clone https://github.com/persimmon-ai-labs/adept-inference*, then get the checkpoints:

```bash git clone https://github.com/persimmon-ai-labs/adept-inference wget https://axtkn4xl5cip.objectstorage.us-phoenix-1.oci.customer-oci.com/n/axtkn4xl5cip/b/adept-public-data/o/8b_base_model_release.tar tar -xvf 8b_base_model_release.tar python src/transformers/models/persimmon/convert_persimmon_weights_to_hf.py –input_dir /path/to/downloaded/persimmon/weights/ –output_dir /output/path

    –pt_model_path /path/to/8b_chat_model_release/iter_0001251/mp_rank_00/model_optim_rng.pt –ada_lib_path /path/to/adept-inference

```

For the chat model: `bash wget https://axtkn4xl5cip.objectstorage.us-phoenix-1.oci.customer-oci.com/n/axtkn4xl5cip/b/adept-public-data/o/8b_chat_model_release.tar tar -xvf 8b_base_model_release.tar `

Thereafter, models can be loaded via:

```py from transformers import PersimmonForCausalLM, PersimmonTokenizer

model = PersimmonForCausalLM.from_pretrained("/output/path") tokenizer = PersimmonTokenizer.from_pretrained("/output/path") ```

This model was contributed by ArthurZ. The original code can be found here.

- Perismmon uses a *sentencepiece* based tokenizer, with a *Unigram* model. It supports bytefallback, which is only available in *tokenizers==0.14.0* for the fast tokenizer.

The *LlamaTokenizer* is used as it is a standard wrapper around sentencepiece. The *chat* template will be updated with the templating functions in a follow up PR!

- The authors suggest to use the following prompt format for the chat mode: *f"human: {prompt}nnadept:"*

**Args:**

**vocab_size (*int*, *optional*, defaults to 262144):**
Vocabulary size of the Persimmon model. Defines the number of different tokens that can be represented by the *inputs_ids* passed when calling `PersimmonModel`
```

hidden_size (*int*, *optional*, defaults to 4096):
Dimension of the hidden representations.

intermediate_size (*int*, *optional*, defaults to 16384):
Dimension of the MLP representations.

num_hidden_layers (*int*, *optional*, defaults to 36):
Number of hidden layers in the Transformer encoder.

num_attention_heads (*int*, *optional*, defaults to 64):
Number of attention heads for each attention layer in the Transformer encoder.

hidden_act (*str* or *function*, *optional*, defaults to *"relu2"*):
The non-linear activation function (function or string) in the decoder.

max_position_embeddings (*int*, *optional*, defaults to 16384):
The maximum sequence length that this model might ever be used with.

initializer_range (*float*, *optional*, defaults to 0.02):
The standard deviation of the truncated_normal_initializer for initializing all weight matrices.

layer_norm_eps (*float*, *optional*, defaults to 1e-5):
The epsilon used by the rms normalization layers.

use_cache (*bool*, *optional*, defaults to *True*):
Whether or not the model should return the last key/values attentions (not used by all models). Only relevant if *config.is_decoder=True*.

tie_word_embeddings(*bool*, *optional*, defaults to *False*):
Whether to tie weight embeddings

rope_theta (*float*, *optional*, defaults to 25000.0):
The base period of the RoPE embeddings.

rope_scaling (*Dict*, *optional*):
Dictionary containing the scaling configuration for the RoPE embeddings. Currently supports two scaling strategies: linear and dynamic. Their scaling factor must be a float greater than 1. The expected format is *{"type": strategy name, "factor": scaling factor}*. When using this flag, don't update *max_position_embeddings* to the expected new maximum. See the following thread for more information on how these scaling strategies behave: https://www.reddit.com/r/LocalPersimmon/comments/14mrgpr/dynamically_scaled_rope_further_increases/. This is an experimental feature, subject to breaking API changes in future versions.

qk_layernorm (*bool*, *optional*, default to *True*):
Whether or not to normalize the Queries and Keys after projecting the hidden states

hidden_dropout (*float*, *optional*, default to 0.0):
The dropout ratio after applying the MLP to the hidden states.

attention_dropout (*float*, *optional*, default to 0.0):
The dropout ratio after computing the attention scores.

partial_rotary_factor (*float*, *optional*, default to 0.5):
Percentage of the query and keys which will have rotary embedding.

class transformers.models.phi.configuration_phi.**PhiConfig**(*vocab_size=51200*, *hidden_size=2048*, *intermediate_size=8192*, *num_hidden_layers=24*, *num_attention_heads=32*, *num_key_value_heads=None*, *resid_pdrop=0.0*, *embd_pdrop=0.0*, *attention_dropout=0.0*, *hidden_act='gelu_new'*, *max_position_embeddings=2048*, *initializer_range=0.02*, *layer_norm_eps=1e-05*, *use_cache=True*, *tie_word_embeddings=False*, *rope_theta=10000.0*, *rope_scaling=None*, *partial_rotary_factor=0.5*, *qk_layernorm=False*, *bos_token_id=1*, *eos_token_id=2*, ***kwargs*)

The Phi-1 model was proposed in Textbooks Are All You Need by Suriya Gunasekar, Yi Zhang, Jyoti Aneja, Caio César Teodoro Mendes, Allie Del Giorno, Sivakanth Gopi, Mojan Javaheripi, Piero Kauffmann, Gustavo de Rosa, Olli Saarikivi, Adil Salim, Shital Shah, Harkirat Singh Behl, Xin Wang, Sébastien Bubeck, Ronen Eldan, Adam Tauman Kalai, Yin Tat Lee and Yuanzhi Li.

The Phi-1.5 model was proposed in Textbooks Are All You Need II: phi-1.5 technical report by Yuanzhi Li, Sébastien Bubeck, Ronen Eldan, Allie Del Giorno, Suriya Gunasekar and Yin Tat Lee.

#Args:

vocab_size (*int*, *optional*, defaults to 51200):
Vocabulary size of the Phi model. Defines the number of different tokens that can be represented by the *inputs_ids* passed when calling `PhiModel`.

hidden_size (*int*, *optional*, defaults to 2048):
Dimension of the hidden representations.

intermediate_size (*int*, *optional*, defaults to 8192):
Dimension of the MLP representations.

num_hidden_layers (*int*, *optional*, defaults to 24):
Number of hidden layers in the Transformer decoder.

num_attention_heads (*int*, *optional*, defaults to 32):
Number of attention heads for each attention layer in the Transformer decoder.

num_key_value_heads (*int*, *optional*):
This is the number of key_value heads that should be used to implement Grouped Query Attention. If *num_key_value_heads=num_attention_heads*, the model will use Multi Head Attention (MHA), if num_key_value_heads=1 the model will use Multi Query Attention (MQA) otherwise GQA is used. When converting a multi-head checkpoint to a GQA checkpoint, each group key and value head should be constructed by meanpooling all the original heads within that group. For more details checkout `this paper. If it is not specified, will default to *num_attention_heads*.

resid_pdrop (*float*, *optional*, defaults to 0.0):
Dropout probability for mlp outputs.

embd_pdrop (*int*, *optional*, defaults to 0.0):
The dropout ratio for the embeddings.

attention_dropout (*float*, *optional*, defaults to 0.0):
The dropout ratio after computing the attention scores.

hidden_act (*str* or *function*, *optional*, defaults to *"gelu_new"*):
The non-linear activation function (function or string) in the decoder.

max_position_embeddings (*int*, *optional*, defaults to 2048):
The maximum sequence length that this model might ever be used with. Phi-1 and Phi-1.5 supports up to 2048 tokens.

initializer_range (*float*, *optional*, defaults to 0.02):
The standard deviation of the truncated_normal_initializer for initializing all weight matrices.

layer_norm_eps (*float*, *optional*, defaults to 1e-05):
The epsilon used by the rms normalization layers.

use_cache (*bool*, *optional*, defaults to *True*):
Whether or not the model should return the last key/values attentions (not used by all models). Only relevant if *config.is_decoder=True*. Whether to tie weight embeddings or not.

tie_word_embeddings (*bool*, *optional*, defaults to *False*):
Whether to tie weight embeddings

rope_theta (*float*, *optional*, defaults to 10000.0):
The base period of the RoPE embeddings.

rope_scaling (*Dict*, *optional*):
Dictionary containing the scaling configuration for the RoPE embeddings. Currently supports two scaling strategies: linear and dynamic. Their scaling factor must be an float greater than 1. The expected format is *{"type": strategy name, "factor": scaling factor}*. When using this flag, don't update *max_position_embeddings* to the expected new maximum. See the following thread for more information on how these scaling strategies behave: https://www.reddit.com/r/LocalPersimmon/comments/14mrgpr/dynamically_scaled_rope_further_increases/. This is an experimental feature, subject to breaking API changes in future versions.

partial_rotary_factor (*float*, *optional*, defaults to 0.5):
Percentage of the query and keys which will have rotary embedding.

qk_layernorm (*bool*, *optional*, defaults to *False*):
Whether or not to normalize the Queries and Keys after projecting the hidden states.

bos_token_id (*int*, *optional*, defaults to 1):
Denotes beginning of sequences token id.

eos_token_id (*int*, *optional*, defaults to 2):
Denotes end of sequences token id.

class transformers.models.plbart.configuration_plbart.**PLBartConfig**(*vocab_size=50005*, *max_position_embeddings=1024*, *encoder_layers=6*, *encoder_ffn_dim=3072*, *encoder_attention_heads=12*, *decoder_layers=6*, *decoder_ffn_dim=3072*, *decoder_attention_heads=12*, *encoder_layerdrop=0.0*, *decoder_layerdrop=0.0*, *use_cache=True*, *is_encoder_decoder=True*, *activation_function='gelu'*, *d_model=768*, *dropout=0.1*, *attention_dropout=0.1*, *activation_dropout=0.0*, *init_std=0.02*, *classifier_dropout=0.0*, *scale_embedding=True*, *pad_token_id=1*, *bos_token_id=0*, *eos_token_id=2*, *forced_eos_token_id=2*, ***kwargs*)

of PLBart

The PLBART model was proposed in Unified Pre-training for Program Understanding and Generation by Wasi Uddin Ahmad, Saikat Chakraborty, Baishakhi Ray, Kai-Wei Chang. This is a BART-like model which can be used to perform code-summarization, code-generation, and code-translation tasks. The pre-trained model *plbart-base* has been trained using multilingual denoising task on Java, Python and English.

According to the abstract

Code summarization and generation empower conversion between programming language (PL) and natural language (NL), while code translation avails the migration of legacy code from one PL to another. This paper introduces PLBART, a sequence-to-sequence model capable of performing a broad spectrum of program and language understanding and generation tasks. PLBART is pre-trained on an extensive collection of Java and Python functions and associated NL text via denoising autoencoding. Experiments on code summarization in the English language, code generation, and code translation in seven programming languages show that PLBART outperforms or rivals state-of-the-art models. Moreover, experiments on discriminative tasks, e.g., program repair, clone detection, and vulnerable code detection, demonstrate PLBART's effectiveness in program understanding. Furthermore, analysis reveals that PLBART learns program syntax, style (e.g., identifier naming convention), logical flow (e.g., if block inside an else block is equivalent to else if block) that are crucial to program semantics and thus excels even with limited annotations.

This model was contributed by gchhablani. The Authors' code can be found here.

#Args:

vocab_size (*int*, *optional*, defaults to 50005):
Vocabulary size of the PLBART model. Defines the number of different tokens that can be represented by the *inputs_ids* passed when calling `PLBartModel`.

d_model (*int*, *optional*, defaults to 768):
Dimensionality of the layers and the pooler layer.

encoder_layers (*int*, *optional*, defaults to 6):
Number of encoder layers.

decoder_layers (*int*, *optional*, defaults to 6):
Number of decoder layers.

encoder_attention_heads (*int*, *optional*, defaults to 12):
Number of attention heads for each attention layer in the Transformer encoder.

decoder_attention_heads (*int*, *optional*, defaults to 12):
Number of attention heads for each attention layer in the Transformer decoder.

decoder_ffn_dim (*int*, *optional*, defaults to 3072):
Dimensionality of the "intermediate" (often named feed-forward) layer in decoder.

encoder_ffn_dim (*int*, *optional*, defaults to 3072):
Dimensionality of the "intermediate" (often named feed-forward) layer in decoder.

activation_function (*str* or *function*, *optional*, defaults to *"gelu"*):
The non-linear activation function (function or string) in the encoder and pooler. If string, *"gelu"*, *"relu"*, *"silu"* and *"gelu_new"* are supported.

dropout (*float*, *optional*, defaults to 0.1):
The dropout probability for all fully connected layers in the embeddings, encoder, and pooler.

attention_dropout (*float*, *optional*, defaults to 0.1):
The dropout ratio for the attention probabilities.

activation_dropout (*float*, *optional*, defaults to 0.0):
The dropout ratio for activations inside the fully connected layer.

classifier_dropout (*float*, *optional*, defaults to 0.0):
The dropout ratio for classifier.

max_position_embeddings (*int*, *optional*, defaults to 1024):
The maximum sequence length that this model might ever be used with. Typically set this to something large just in case (e.g., 512 or 1024 or 2048).

init_std (*float*, *optional*, defaults to 0.02):
The standard deviation of the truncated_normal_initializer for initializing all weight matrices.

encoder_layerdrop (*float*, *optional*, defaults to 0.0):
The LayerDrop probability for the encoder. See the LayerDrop paper for more details.

decoder_layerdrop (*float*, *optional*, defaults to 0.0):
The LayerDrop probability for the decoder. See the LayerDrop paper for more details.

scale_embedding (*bool*, *optional*, defaults to *True*):
Scale embeddings by diving by sqrt(d_model).

use_cache (*bool*, *optional*, defaults to *True*):
Whether or not the model should return the last key/values attentions (not used by all models)

forced_eos_token_id (*int*, *optional*, defaults to 2):
The id of the token to force as the last generated token when *max_length* is reached. Usually set to *eos_token_id*.

class transformers.models.prophetnet.configuration_prophetnet.**ProphetNetConfig**(*activation_dropout: float | None = 0.1*, *activation_function: str | Callable | NoneType = 'gelu'*, *vocab_size: int | None = 30522*, *hidden_size: int | None = 1024*, *encoder_ffn_dim: int | None = 4096*, *num_encoder_layers: int | None = 12*, *num_encoder_attention_heads: int | None = 16*, *decoder_ffn_dim: int | None = 4096*, *num_decoder_layers: int | None = 12*, *num_decoder_attention_heads: int | None = 16*, *attention_dropout: float | None = 0.1*, *dropout: float | None = 0.1*, *max_position_embeddings: int | None = 512*, *init_std: float | None = 0.02*, *is_encoder_decoder: bool | None = True*, *add_cross_attention: bool | None = True*, *decoder_start_token_id: int | None = 0*, *ngram: int | None = 2*, *num_buckets: int | None = 32*, *relative_max_distance: int | None = 128*, *disable_ngram_loss: bool | None = False*, *eps:*

The ProphetNet model was proposed in ProphetNet: Predicting Future N-gram for Sequence-to-Sequence Pre-training, by Yu Yan, Weizhen Qi, Yeyun Gong, Dayiheng Liu, Nan Duan, Jiusheng Chen, Ruofei Zhang, Ming Zhou on 13 Jan, 2020.

ProphetNet is an encoder-decoder model and can predict n-future tokens for "ngram" language modeling instead of just the next token.

The abstract from the paper is the following:

In this paper, we present a new sequence-to-sequence pretraining model called ProphetNet, which introduces a novel self-supervised objective named future n-gram prediction and the proposed n-stream self-attention mechanism. Instead of the optimization of one-step ahead prediction in traditional sequence-to-sequence model, the ProphetNet is optimized by n-step ahead prediction which predicts the next n tokens simultaneously based on previous context tokens at each time step. The future n-gram prediction explicitly encourages the model to plan for the future tokens and prevent overfitting on strong local correlations. We pre-train ProphetNet using a base scale dataset (16GB) and a large scale dataset (160GB) respectively. Then we conduct experiments on CNN/DailyMail, Gigaword, and SQuAD 1.1 benchmarks for abstractive summarization and question generation tasks. Experimental results show that ProphetNet achieves new state-of-the-art results on all these datasets compared to the models using the same scale pretraining corpus.

Tips:

- ProphetNet is a model with absolute position embeddings so it's usually advised to pad the inputs on the right rather than the left.
- The model architecture is based on the original Transformer, but replaces the "standard" self-attention mechanism in the decoder by a a main self-attention mechanism and a self and n-stream (predict) self-attention mechanism.

The Authors' code can be found here.

Args:

activation_dropout (*float*, *optional*, defaults to 0.1):
The dropout ratio for activations inside the fully connected layer.

activation_function (*str* or *function*, *optional*, defaults to *"gelu"*):
The non-linear activation function (function or string) in the encoder and pooler. If string, *"gelu"*, *"relu"*, *"silu"* and *"gelu_new"* are supported.

vocab_size (*int*, *optional*, defaults to 30522):
Vocabulary size of the ProphetNET model. Defines the number of different tokens that can be represented by the *inputs_ids* passed when calling `ProphetNetModel`.

hidden_size (*int*, *optional*, defaults to 1024):
Dimensionality of the layers and the pooler layer.

encoder_ffn_dim (*int*, *optional*, defaults to 4096):
Dimensionality of the "intermediate" (often named feed-forward) layer in decoder.

num_encoder_layers (*int*, *optional*, defaults to 12):
Number of encoder layers.

num_encoder_attention_heads (*int*, *optional*, defaults to 16):
Number of attention heads for each attention layer in the Transformer encoder.

decoder_ffn_dim (*int*, *optional*, defaults to 4096):
Dimensionality of the *intermediate* (often named feed-forward) layer in decoder.

num_decoder_layers (*int*, *optional*, defaults to 12):
Number of decoder layers.

num_decoder_attention_heads (*int*, *optional*, defaults to 16):
Number of attention heads for each attention layer in the Transformer decoder.

attention_dropout (*float*, *optional*, defaults to 0.1):
The dropout ratio for the attention probabilities.

dropout (*float*, *optional*, defaults to 0.1):
The dropout probability for all fully connected layers in the embeddings, encoder, and pooler.

max_position_embeddings (*int*, *optional*, defaults to 512):
The maximum sequence length that this model might ever be used with. Typically set this to something large just in case (e.g., 512 or 1024 or 2048).

init_std (*float*, *optional*, defaults to 0.02):
The standard deviation of the truncated_normal_initializer for initializing all weight matrices.

add_cross_attention (*bool*, *optional*, defaults to *True*):
Whether cross-attention layers should be added to the model.

is_encoder_decoder (*bool*, *optional*, defaults to *True*):
Whether this is an encoder/decoder model.

pad_token_id (*int*, *optional*, defaults to 1)
Padding token id.

bos_token_id (*int*, *optional*, defaults to 0)
Beginning of stream token id.

eos_token_id (*int*, *optional*, defaults to 2)
End of stream token id.

ngram (*int*, *optional*, defaults to 2)
Number of future tokens to predict. Set to 1 to be same as traditional Language model to predict next first token.

num_buckets (*int*, *optional*, defaults to 32)
The number of buckets to use for each attention layer. This is for relative position calculation. See the T5 paper for more details.

relative_max_distance (*int*, *optional*, defaults to 128)
Relative distances greater than this number will be put into the last same bucket. This is for relative position calculation. See the T5 paper for more details.

disable_ngram_loss (*bool*, *optional*, defaults to *False*):
Whether be trained predicting only the next first token.

eps (*float*, *optional*, defaults to 0.0):
Controls the *epsilon* parameter value for label smoothing in the loss calculation. If set to 0, no label smoothing is performed.

use_cache (*bool*, *optional*, defaults to *True*):
Whether or not the model should return the last key/values attentions (not used by all models).

class transformers.models.qwen2.configuration_qwen2.**Qwen2Config**(*vocab_size=151936*, *hidden_size=4096*, *intermediate_size=22016*, *num_hidden_layers=32*, *num_attention_heads=32*, *num_key_value_heads=32*, *hidden_act='silu'*, *max_position_embeddings=32768*, *initializer_range=0.02*, *rms_norm_eps=1e-06*, *use_cache=True*, *tie_word_embeddings=False*, *rope_theta=10000.0*, *use_sliding_window=False*, *sliding_window=4096*, *max_window_layers=28*, *attention_dropout=0.0*, ***kwargs*)

Qwen2 is the new model series of large language models from the Qwen team. Previously, we released the Qwen series, including Qwen-72B, Qwen-1.8B, Qwen-VL, Qwen-Audio, etc.

#Args:

vocab_size (*int*, *optional*, defaults to 151936):
Vocabulary size of the Qwen2 model. Defines the number of different tokens that can be represented by the *inputs_ids* passed when calling `Qwen2Model`

hidden_size (*int*, *optional*, defaults to 4096):
Dimension of the hidden representations.

intermediate_size (*int*, *optional*, defaults to 22016):
Dimension of the MLP representations.

num_hidden_layers (*int*, *optional*, defaults to 32):
Number of hidden layers in the Transformer encoder.

num_attention_heads (*int*, *optional*, defaults to 32):
Number of attention heads for each attention layer in the Transformer encoder.

num_key_value_heads (*int*, *optional*, defaults to 32):
This is the number of key_value heads that should be used to implement Grouped Query Attention. If *num_key_value_heads=num_attention_heads*, the model will use Multi Head Attention (MHA), if num_key_value_heads=1 the model will use Multi Query Attention (MQA) otherwise GQA is used. When converting a multi-head checkpoint to a GQA checkpoint, each group key and value head should be constructed by meanpooling all the original heads within that group. For more details checkout `this paper. If it is not specified, will default to *32*.

hidden_act (*str* or *function*, *optional*, defaults to *"silu"*):
The non-linear activation function (function or string) in the decoder.

max_position_embeddings (*int*, *optional*, defaults to 32768):
The maximum sequence length that this model might ever be used with.

initializer_range (*float*, *optional*, defaults to 0.02):
The standard deviation of the truncated_normal_initializer for initializing all weight matrices.

rms_norm_eps (*float*, *optional*, defaults to 1e-06):
The epsilon used by the rms normalization layers.

use_cache (*bool*, *optional*, defaults to *True*):
Whether or not the model should return the last key/values attentions (not used by all models). Only relevant if *config.is_decoder=True*.

tie_word_embeddings (*bool*, *optional*, defaults to *False*):
Whether the model's input and output word embeddings should be tied.

rope_theta (*float*, *optional*, defaults to 10000.0):
The base period of the RoPE embeddings.

use_sliding_window (*bool*, *optional*, defaults to *False*):
Whether to use sliding window attention.

sliding_window (*int*, *optional*, defaults to 4096):
Sliding window attention (SWA) window size. If not specified, will default to *4096*.

max_window_layers (*int*, *optional*, defaults to 28):
The number of layers that use SWA (Sliding Window Attention). The bottom layers use SWA while the top use full attention.

attention_dropout (*float*, *optional*, defaults to 0.0):
The dropout ratio for the attention probabilities.

```
>>> from transformers import Qwen2Model, Qwen2Config
```

```
>>> # Initializing a Qwen2 style configuration
>>> configuration = Qwen2Config()
```

```
>>> # Initializing a model from the Qwen2-7B style configuration
>>> model = Qwen2Model(configuration)
```

```
>>> # Accessing the model configuration
>>> configuration = model.config
```

class transformers.models.reformer.configuration_reformer.**ReformerConfig**(*attention_head_size=64*, *attn_layers=['local', 'lsh', 'local', 'lsh', 'local', 'lsh']*, *axial_norm_std=1.0*, *axial_pos_embds=True*, *axial_pos_shape=[64, 64]*, *axial_pos_embds_dim=[64, 192]*, *chunk_size_lm_head=0*, *eos_token_id=2*, *feed_forward_size=512*, *hash_seed=None*, *hidden_act='relu'*, *hidden_dropout_prob=0.05*, *hidden_size=256*, *initializer_range=0.02*, *is_decoder=False*, *layer_norm_eps=1e-12*, *local_num_chunks_before=1*, *local_num_chunks_after=0*, *local_attention_probs_dropout_prob=0.0* *local_attn_chunk_length=64*, *lsh_attn_chunk_length=64*, *lsh_attention_probs_dropout_prob=0.0* *lsh_num_chunks_before=1*, *lsh_num_chunks_after=0*, *max_position_embeddings=4096*, *num_attention_heads=12*, *num_buckets=None*, *num_hashes=1*, *pad_token_id=0*, *vocab_size=320*, *tie_word_embeddings=False*, *use_cache=True*, *classifier_dropout=None*, ***kwargs*)

The Reformer model was proposed in the paper Reformer: The Efficient Transformer by Nikita Kitaev, Łukasz Kaiser, Anselm Levskaya.

The abstract from the paper is the following:

Large Transformer models routinely achieve state-of-the-art results on a number of tasks but training these models

can be prohibitively costly, especially on long sequences. We introduce two techniques to improve the efficiency of Transformers. For one, we replace dot-product attention by one that uses locality-sensitive hashing, changing its complexity from O(L^2) to O(Llog(L)), where L is the length of the sequence. Furthermore, we use reversible residual layers instead of the standard residuals, which allows storing activations only once in the training process instead of N times, where N is the number of layers. The resulting model, the Reformer, performs on par with Transformer models while being much more memory-efficient and much faster on long sequences.

This model was contributed by patrickvonplaten. The Authors' code can be found here.

Tips:

- Reformer does **not** work with *torch.nn.DataParallel* due to a bug in PyTorch, see issue #36035.
- Use Axial position encoding (see below for more details). It's a mechanism to avoid having a huge positional encoding matrix (when the sequence length is very big) by factorizing it into smaller matrices.
- Replace traditional attention by LSH (local-sensitive hashing) attention (see below for more details). It's a technique to avoid computing the full product query-key in the attention layers.
- Avoid storing the intermediate results of each layer by using reversible transformer layers to obtain them during the backward pass (subtracting the residuals from the input of the next layer gives them back) or recomputing them for results inside a given layer (less efficient than storing them but saves memory).
- Compute the feedforward operations by chunks and not on the whole batch.

Args:

attention_head_size (*int*, *optional*, defaults to 64):
Dimensionality of the projected key, query and value vectors

attn_layers (*List`str]*, *optional*, defaults to *["local", "lsh", "local", "lsh", "local", "lsh"]*):
List of attention layer types in ascending order. It can be chosen between a LSHSelfAttention layer (*"lsh"*) and a LocalSelfAttention layer (*"local"*).

For more information on LSHSelfAttention layer, see [LSH Self Attention <reformer#lsh-self-attention>`__. For more information on LocalSelfAttention layer, see Local Self Attention.

axial_pos_embds (*bool*, *optional*, defaults to *True*):
Whether or not to use axial position embeddings. For more information on how axial position embeddings work, see Axial Position Encodings.

axial_norm_std (*float*, *optional*, defaults to 1.0):
The standard deviation of the normal_initializer for initializing the weight matrices of the axial positional encodings.

axial_pos_shape (*List`int]*, *optional*, defaults to *[64, 64]*):
The position dims of the axial position encodings. During training, the product of the position dims has to be equal to the sequence length.

For more information on how axial position embeddings work, see [Axial Position Encodings <reformer#axial-positional-encodings>`__.

axial_pos_embds_dim (*List`int]*, *optional*, defaults to *[64, 192]*):
The embedding dims of the axial position encodings. The sum of the embedding dims has to be equal to the hidden size.

For more information on how axial position embeddings work, see [Axial Position Encodings <reformer#axial-positional-encodings>`__.

chunk_size_lm_head (*int*, *optional*, defaults to 0):
The chunk size of the final language model feed forward head layer. A chunk size of 0 means that the feed forward layer is not chunked. A chunk size of n means that the feed forward layer processes n < sequence_length embeddings at a time.

For more information on feed forward chunking, see How does Feed Forward Chunking work?.

eos_token_id (*int*, *optional*, defaults to 2):
The token id for the end-of-sentence token.

feed_forward_size (*int*, *optional*, defaults to 512):
Dimensionality of the feed_forward layer in the residual attention block.

hash_seed (*int*, *optional*):
Seed that can be used to make local sensitive hashing in *LSHSelfAttention* deterministic. This should only be set for testing purposed. For evaluation and training purposes *hash_seed* should be left as *None* to ensure fully random rotations in local sensitive hashing scheme.

hidden_act (*str* or *Callable*, *optional*, defaults to *"relu"*):
The non-linear activation function (function or string) in the feed forward layer in the residual attention block. If string, *"gelu"*, *"relu"*, *"silu"* and *"gelu_new"* are supported.

hidden_dropout_prob (*float*, *optional*, defaults to 0.05):
The dropout probability for all fully connected layers in the embeddings, encoder, and pooler.

hidden_size (*int*, *optional*, defaults to 256):
Dimensionality of the output hidden states of the residual attention blocks.

initializer_range (*float*, *optional*, defaults to 0.02):
The standard deviation of the truncated_normal_initializer for initializing all weight matrices.

is_decoder (*bool*, *optional*, defaults to *False*):
Whether or not to use a causal mask in addition to the *attention_mask* passed to `ReformerModel`. When using the Reformer for causal language modeling, this argument should be set to *True*.

layer_norm_eps (*float*, *optional*, defaults to 1e-12):
The epsilon used by the layer normalization layers.

local_chunk_length (*int*, *optional*, defaults to 64):
Length of chunk which attends to itself in *LocalSelfAttention*. Chunking reduces memory complexity from sequence length x sequence length (self attention) to chunk length x chunk length x sequence length / chunk length (chunked self attention).

local_num_chunks_before (*int*, *optional*, defaults to 1):
Number of previous neighbouring chunks to attend to in *LocalSelfAttention* layer to itself.

local_num_chunks_after (*int*, *optional*, defaults to 0):
Number of following neighbouring chunks to attend to in *LocalSelfAttention* layer in addition to itself.

local_attention_probs_dropout_prob (*float*, *optional*, defaults to 0.1):
The dropout ratio for the attention probabilities in *LocalSelfAttention*.

lsh_attn_chunk_length (*int*, *optional*, defaults to 64):
Length of chunk which attends to itself in *LSHSelfAttention*. Chunking reduces memory complexity from sequence length x sequence length (self attention) to chunk length x chunk length x sequence length / chunk length (chunked self attention).

lsh_num_chunks_before (*int*, *optional*, defaults to 1):
Number of previous neighbouring chunks to attend to in *LSHSelfAttention* layer to itself.

lsh_num_chunks_after (*int*, *optional*, defaults to 0):
Number of following neighbouring chunks to attend to in *LSHSelfAttention* layer to itself.

lsh_attention_probs_dropout_prob (*float*, *optional*, defaults to 0.1):
The dropout ratio for the attention probabilities in *LSHSelfAttention*.

max_position_embeddings (*int*, *optional*, defaults to 4096):
The maximum sequence length that this model might ever be used with. Typically set this to something large just in case (e.g., 512 or 1024 or 2048).

num_attention_heads (*int*, *optional*, defaults to 12):
Number of attention heads for each attention layer in the Transformer encoder.

num_buckets (*int* or *List[int]*, *optional*):
Number of buckets, the key query vectors can be "hashed into" using the locality sensitive hashing scheme. Each query key vector is hashed into a hash in *1, …, num_buckets*. The number of buckets can also be factorized into a list for improved memory complexity. In this case, each query key vector is hashed into a hash in *1-1, 1-2, …, num_buckets[0]-1, …, num_buckets[0]-num_buckets[1]* if *num_buckets* is factorized into two factors. The number of buckets (or the product the factors) should approximately equal sequence length / lsh_chunk_length. If *num_buckets* not set, a good value is calculated on the fly.

num_hashes (*int*, *optional*, defaults to 1):
Number of hashing rounds (e.g., number of random rotations) in Local Sensitive Hashing scheme. The higher *num_hashes*, the more accurate the *LSHSelfAttention* becomes, but also the more memory and time intensive the hashing becomes.

pad_token_id (*int*, *optional*, defaults to 0):
The token id for the padding token.

vocab_size (*int*, *optional*, defaults to 320):
Vocabulary size of the Reformer model. Defines the number of different tokens that can be represented by the *inputs_ids* passed when calling `ReformerModel`.

tie_word_embeddings (*bool*, *optional*, defaults to *False*):
Whether to tie input and output embeddings.

use_cache (*bool*, *optional*, defaults to *True*):
Whether or not the model should return the last key/values attentions (not used by all models).

classifier_dropout (*float*, *optional*):
The dropout ratio for the classification head.

class transformers.models.rembert.configuration_rembert.**RemBertConfig**(*vocab_size=250300*, *hidden_size=1152*, *num_hidden_layers=32*, *num_attention_heads=18*, *input_embedding_size=256*, *output_embedding_size=1664*, *intermediate_size=4608*, *hidden_act='gelu'*, *hidden_dropout_prob=0.0*, *attention_probs_dropout_prob=0.0*, *classifier_dropout_prob=0.1*, *max_position_embeddings=512*, *type_vocab_size=2*, *initializer_range=0.02*, *layer_norm_eps=1e-12*, *use_cache=True*, *pad_token_id=0*, *bos_token_id=312*, *eos_token_id=313*, ***kwargs*)

The RemBERT model was proposed in Rethinking Embedding Coupling in Pre-trained Language Models by Hyung Won Chung, Thibault Févry, Henry Tsai, Melvin Johnson, Sebastian Ruder.

The abstract from the paper is the following:

We re-evaluate the standard practice of sharing weights between input and output embeddings in state-of-the-art pre-trained language models. We show that decoupled embeddings provide increased modeling flexibility, allowing us to significantly improve the efficiency of parameter allocation in the input embedding of multilingual models. By reallocating the input embedding parameters in the Transformer layers, we achieve dramatically better performance on standard natural language understanding tasks with the same number of parameters during fine-tuning. We also show that allocating additional capacity to the output embedding provides benefits to the model that persist through the fine-tuning stage even though the output embedding is discarded after pre-training. Our analysis shows that larger output embeddings prevent the model's last layers from overspecializing to the pre-training task and encourage Transformer representations to be more general and more transferable to other tasks and languages. Harnessing these findings, we are able to train models that achieve strong performance on the XTREME benchmark without increasing the number of parameters at the fine-tuning stage.

Tips:

For fine-tuning, RemBERT can be thought of as a bigger version of mBERT with an ALBERT-like factorization of the embedding layer. The embeddings are not tied in pre-training, in contrast with BERT, which enables smaller input embeddings (preserved during fine-tuning) and bigger output embeddings (discarded at fine-tuning). The tokenizer is also similar to the Albert one rather than the BERT one.

Args:

vocab_size (*int*, *optional*, defaults to 250300):
Vocabulary size of the RemBERT model. Defines the number of different tokens that can be represented by the *inputs_ids* passed when calling `RemBertModel` or `TFRemBertModel`. Vocabulary size of the model. Defines the different tokens that can be represented by the *inputs_ids* passed to the forward method of `RemBertModel`.

hidden_size (*int*, *optional*, defaults to 1152):
Dimensionality of the encoder layers and the pooler layer.

num_hidden_layers (*int*, *optional*, defaults to 32):
Number of hidden layers in the Transformer encoder.

num_attention_heads (*int*, *optional*, defaults to 18):
Number of attention heads for each attention layer in the Transformer encoder.

input_embedding_size (*int*, *optional*, defaults to 256):
Dimensionality of the input embeddings.

output_embedding_size (*int*, *optional*, defaults to 1664):
Dimensionality of the output embeddings.

intermediate_size (*int*, *optional*, defaults to 4608):
Dimensionality of the "intermediate" (i.e., feed-forward) layer in the Transformer encoder.

hidden_act (*str* or *function*, *optional*, defaults to *"gelu"*):
The non-linear activation function (function or string) in the encoder and pooler. If string, *"gelu"*, *"relu"*, *"selu"* and *"gelu_new"* are supported.

hidden_dropout_prob (*float*, *optional*, defaults to 0):
The dropout probability for all fully connected layers in the embeddings, encoder, and pooler.

attention_probs_dropout_prob (*float*, *optional*, defaults to 0):
The dropout ratio for the attention probabilities.

classifier_dropout_prob (*float*, *optional*, defaults to 0.1):
The dropout ratio for the classifier layer when fine-tuning.

max_position_embeddings (*int*, *optional*, defaults to 512):
The maximum sequence length that this model might ever be used with. Typically set this to something large just in case (e.g., 512 or 1024 or 2048).

type_vocab_size (*int*, *optional*, defaults to 2):
The vocabulary size of the *token_type_ids* passed when calling `RemBertModel` or `TFRemBertModel`.

initializer_range (*float*, *optional*, defaults to 0.02):
The standard deviation of the truncated_normal_initializer for initializing all weight matrices.

layer_norm_eps (*float*, *optional*, defaults to 1e-12):
The epsilon used by the layer normalization layers.

is_decoder (*bool*, *optional*, defaults to *False*):
Whether the model is used as a decoder or not. If *False*, the model is used as an encoder.

use_cache (*bool*, *optional*, defaults to *True*):
Whether or not the model should return the last key/values attentions (not used by all models). Only relevant if *config.is_decoder=True*.

class transformers.models.roberta.configuration_roberta.**RobertaConfig**(*vocab_size=50265*, *hidden_size=768*, *num_hidden_layers=12*, *num_attention_heads=12*, *intermediate_size=3072*, *hidden_act='gelu'*, *hidden_dropout_prob=0.1*, *attention_probs_dropout_prob=0.1*, *max_position_embeddings=512*, *type_vocab_size=2*, *initializer_range=0.02*, *layer_norm_eps=1e-12*, *pad_token_id=1*, *bos_token_id=0*, *eos_token_id=2*, *position_embedding_type='absolute'*, *use_cache=True*, *classifier_dropout=None*, ***kwargs*)

The RoBERTa model was proposed in RoBERTa: A Robustly Optimized BERT Pretraining Approach by Yinhan Liu, Myle Ott, Naman Goyal, Jingfei Du, Mandar Joshi, Danqi Chen, Omer Levy, Mike Lewis, Luke Zettlemoyer, Veselin Stoyanov. It is based on Google's BERT model released in 2018.

It builds on BERT and modifies key hyperparameters, removing the next-sentence pretraining objective and training with much larger mini-batches and learning rates.

The abstract from the paper is the following:

Language model pretraining has led to significant performance gains but careful comparison between different approaches is challenging. Training is computationally expensive, often done on private datasets of different sizes, and, as we will show, hyperparameter choices have significant impact on the final results. We present a replication study of BERT pretraining (Devlin et al., 2019) that carefully measures the impact of many key hyperparameters and training data size. We find that BERT was significantly undertrained, and can match or exceed the performance of every model published after it. Our best model achieves state-of-the-art results on GLUE, RACE and SQuAD. These results highlight the importance of previously overlooked design choices, and raise questions about the source of recently reported improvements. We release our models and code.

Tips:

- This implementation is the same as `BertModel` with a tiny embeddings tweak as well as a setup for Roberta pretrained models.
- RoBERTa has the same architecture as BERT, but uses a byte-level BPE as a tokenizer (same as GPT-2) and uses a different pretraining scheme.
- RoBERTa doesn't have *token_type_ids*, you don't need to indicate which token belongs to which segment. Just separate your segments with the separation token *tokenizer.sep_token* (or *</s>*)
- Same as BERT with better pretraining tricks:
  - dynamic masking: tokens are masked differently at each epoch, whereas BERT does it once and for all
  - together to reach 512 tokens (so the sentences are in an order than may span several documents)
  - train with larger batches
  - use BPE with bytes as a subunit and not characters (because of unicode characters)
- CamemBERT is a wrapper around RoBERTa. Refer to this page for usage examples.

This model was contributed by julien-c. The original code can be found here.

Args:

vocab_size (*int*, *optional*, defaults to 50265):
Vocabulary size of the RoBERTa model. Defines the number of different tokens that can be represented by the *inputs_ids* passed when calling `RobertaModel` or `TFRobertaModel`.

hidden_size (*int*, *optional*, defaults to 768):
Dimensionality of the encoder layers and the pooler layer.

num_hidden_layers (*int*, *optional*, defaults to 12):
Number of hidden layers in the Transformer encoder.

num_attention_heads (*int*, *optional*, defaults to 12):
Number of attention heads for each attention layer in the Transformer encoder.

intermediate_size (*int*, *optional*, defaults to 3072):
Dimensionality of the "intermediate" (often named feed-forward) layer in the Transformer encoder.

hidden_act (*str* or *Callable*, *optional*, defaults to *"gelu"*):
The non-linear activation function (function or string) in the encoder and pooler. If string, *"gelu"*, *"relu"*, *"silu"* and *"gelu_new"* are supported.

hidden_dropout_prob (*float*, *optional*, defaults to 0.1):
The dropout probability for all fully connected layers in the embeddings, encoder, and pooler.

attention_probs_dropout_prob (*float*, *optional*, defaults to 0.1):
The dropout ratio for the attention probabilities.

max_position_embeddings (*int*, *optional*, defaults to 512):
The maximum sequence length that this model might ever be used with. Typically set this to something large just in case (e.g., 512 or 1024 or 2048).

type_vocab_size (*int*, *optional*, defaults to 2):
The vocabulary size of the *token_type_ids* passed when calling `RobertaModel` or `TFRobertaModel`.

initializer_range (*float*, *optional*, defaults to 0.02):
The standard deviation of the truncated_normal_initializer for initializing all weight matrices.

layer_norm_eps (*float*, *optional*, defaults to 1e-12):
The epsilon used by the layer normalization layers.

position_embedding_type (*str*, *optional*, defaults to *"absolute"*):
Type of position embedding. Choose one of *"absolute"*, *"relative_key"*, *"relative_key_query"*. For positional embeddings use *"absolute"*. For more information on *"relative_key"*, please refer to Self-Attention with Relative Position Representations (Shaw et al.). For more information on *"relative_key_query"*, please refer to *Method 4* in Improve Transformer Models with Better Relative Position Embeddings (Huang et al.).

is_decoder (*bool*, *optional*, defaults to *False*):
Whether the model is used as a decoder or not. If *False*, the model is used as an encoder.

use_cache (*bool*, *optional*, defaults to *True*):
Whether or not the model should return the last key/values attentions (not used by all models). Only relevant if *config.is_decoder=True*.

classifier_dropout (*float*, *optional*):
The dropout ratio for the classification head.

class transformers.models.roberta_prelayernorm.configuration_roberta_prelayernorm.**RobertaPreLayerNormCo**

The RoBERTa-PreLayerNorm model was proposed in fairseq: A Fast, Extensible Toolkit for Sequence Modeling by Myle Ott, Sergey Edunov, Alexei Baevski, Angela Fan, Sam Gross, Nathan Ng, David Grangier, Michael Auli. It is identical to using the *–encoder-normalize-before* flag in fairseq.

The abstract from the paper is the following:

fairseq is an open-source sequence modeling toolkit that allows researchers and developers to train custom models for translation, summarization, language modeling, and other text generation tasks. The toolkit is based on PyTorch and supports distributed training across multiple GPUs and machines. We also support fast mixed-precision training and inference on modern GPUs.

Tips:

- The implementation is the same as Roberta except instead of using _Add and **Norm_** it does _Norm and **Add_**. _Add_ and _Norm_ refers to the Addition and LayerNormalization as described in Attention Is All You Need.
- This is identical to using the *–encoder-normalize-before* flag in fairseq.

This model was contributed by andreasmaden. The original code can be found here.

Args:

vocab_size (*int*, *optional*, defaults to 50265):
Vocabulary size of the RoBERTa-PreLayerNorm model. Defines the number of different tokens that can be represented by the *inputs_ids* passed when calling `RobertaPreLayerNormModel` or `TFRobertaPreLayerNormModel`.

hidden_size (*int*, *optional*, defaults to 768):
Dimensionality of the encoder layers and the pooler layer.

num_hidden_layers (*int*, *optional*, defaults to 12):
Number of hidden layers in the Transformer encoder.

num_attention_heads (*int*, *optional*, defaults to 12):
Number of attention heads for each attention layer in the Transformer encoder.

intermediate_size (*int*, *optional*, defaults to 3072):
Dimensionality of the "intermediate" (often named feed-forward) layer in the Transformer encoder.

hidden_act (*str* or *Callable*, *optional*, defaults to *"gelu"*):
The non-linear activation function (function or string) in the encoder and pooler. If string, *"gelu"*, *"relu"*, *"silu"* and *"gelu_new"* are supported.

hidden_dropout_prob (*float*, *optional*, defaults to 0.1):
The dropout probability for all fully connected layers in the embeddings, encoder, and pooler.

attention_probs_dropout_prob (*float*, *optional*, defaults to 0.1):
The dropout ratio for the attention probabilities.

max_position_embeddings (*int*, *optional*, defaults to 512):
The maximum sequence length that this model might ever be used with. Typically set this to something large just in case (e.g., 512 or 1024 or 2048).

type_vocab_size (*int*, *optional*, defaults to 2):
The vocabulary size of the *token_type_ids* passed when calling `RobertaPreLayerNormModel` or `TFRobertaPreLayerNormModel`.

initializer_range (*float*, *optional*, defaults to 0.02):
The standard deviation of the truncated_normal_initializer for initializing all weight matrices.

layer_norm_eps (*float*, *optional*, defaults to 1e-12):
The epsilon used by the layer normalization layers.

position_embedding_type (*str*, *optional*, defaults to *"absolute"*):
Type of position embedding. Choose one of *"absolute"*, *"relative_key"*, *"relative_key_query"*. For positional embeddings use *"absolute"*. For more information on *"relative_key"*, please refer to Self-Attention with Relative Position Representations (Shaw et al.). For more information on *"relative_key_query"*, please refer to *Method 4* in Improve Transformer Models with Better Relative Position Embeddings (Huang et al.).

is_decoder (*bool*, *optional*, defaults to *False*):
Whether the model is used as a decoder or not. If *False*, the model is used as an encoder.

use_cache (*bool*, *optional*, defaults to *True*):
Whether or not the model should return the last key/values attentions (not used by all models). Only relevant if *config.is_decoder=True*.

classifier_dropout (*float*, *optional*):
The dropout ratio for the classification head.

class transformers.models.roc_bert.configuration_roc_bert.**RoCBertConfig**(*vocab_size=30522*, *hidden_size=768*, *num_hidden_layers=12*, *num_attention_heads=12*, *intermediate_size=3072*, *hidden_act='gelu'*, *hidden_dropout_prob=0.1*, *attention_probs_dropout_prob=0.1*, *max_position_embeddings=512*, *type_vocab_size=2*, *initializer_range=0.02*, *layer_norm_eps=1e-12*, *use_cache=True*, *pad_token_id=0*, *position_embedding_type='absolute'*, *classifier_dropout=None*, *enable_pronunciation=True*, *enable_shape=True*, *pronunciation_embed_dim=768*, *pronunciation_vocab_size=910*, *shape_embed_dim=512*, *shape_vocab_size=24858*, *concat_input=True*, ***kwargs*)

The RoCBert model was proposed in RoCBert: Robust Chinese Bert with Multimodal Contrastive Pretraining by HuiSu, WeiweiShi, XiaoyuShen, XiaoZhou, TuoJi, JiaruiFang, JieZhou. It's a pretrained Chinese language model that is robust under various forms of adversarial attacks.

The abstract from the paper is the following:

Large-scale pretrained language models have achieved SOTA results on NLP tasks. However, they have been shown vulnerable to adversarial attacks especially for logographic languages like Chinese. In this work, we propose ROCBERT: a pretrained Chinese Bert that is robust to various forms of adversarial attacks like word perturbation, synonyms, typos, etc. It is pretrained with the contrastive learning objective which maximizes the label consistency under different synthesized adversarial examples. The model takes as input multimodal information including the semantic, phonetic and visual features. We show all these features are important to the model robustness since the attack can be performed in all the three forms. Across 5 Chinese NLU tasks, ROCBERT outperforms strong baselines under three blackbox adversarial algorithms without sacrificing the performance on clean testset. It also performs the best in the toxic content detection task under human-made attacks.

This model was contributed by weiweishi.

Args:

vocab_size (*int*, *optional*, defaults to 30522):
Vocabulary size of the RoCBert model. Defines the number of different tokens that can be represented by the *inputs_ids* passed when calling `RoCBertModel`.

hidden_size (*int*, *optional*, defaults to 768):
Dimension of the encoder layers and the pooler layer.

num_hidden_layers (*int*, *optional*, defaults to 12):
Number of hidden layers in the Transformer encoder.

num_attention_heads (*int*, *optional*, defaults to 12):
Number of attention heads for each attention layer in the Transformer encoder.

intermediate_size (*int*, *optional*, defaults to 3072):
Dimension of the "intermediate" (i.e., feed-forward) layer in the Transformer encoder.

hidden_act (*str* or *function*, *optional*, defaults to *"gelu"*):
The non-linear activation function (function or string) in the encoder and pooler. If string, *"gelu"*, *"relu"*, *"selu"* and *"gelu_new"* are supported.

hidden_dropout_prob (*float*, *optional*, defaults to 0.1):
The dropout probability for all fully connected layers in the embeddings, encoder, and pooler.

attention_probs_dropout_prob (*float*, *optional*, defaults to 0.1):
The dropout ratio for the attention probabilities.

max_position_embeddings (*int*, *optional*, defaults to 512):
The maximum sequence length that this model might ever be used with. Typically set this to something large just in case (e.g., 512 or 1024 or 2048).

type_vocab_size (*int*, *optional*, defaults to 2):
The vocabulary size of the *token_type_ids* passed when calling `RoCBertModel`.

initializer_range (*float*, *optional*, defaults to 0.02):
The standard deviation of the truncated_normal_initializer for initializing all weight matrices.

layer_norm_eps (*float*, *optional*, defaults to 1e-12):
The epsilon used by the layer normalization layers.

is_decoder (*bool*, *optional*, defaults to *False*):
Whether the model is used as a decoder or not. If *False*, the model is used as an encoder.

use_cache (*bool*, *optional*, defaults to *True*):
Whether or not the model should return the last key/values attentions (not used by all models). Only relevant if *config.is_decoder=True*.

position_embedding_type (*str*, *optional*, defaults to *"absolute"*):
Type of position embedding. Choose one of *"absolute"*, *"relative_key"*, *"relative_key_query"*. For positional embeddings use *"absolute"*. For more information on *"relative_key"*, please refer to Self-Attention with Relative Position Representations (Shaw et al.). For more information on *"relative_key_query"*, please refer to *Method 4* in Improve Transformer Models with Better Relative Position Embeddings (Huang et al.).

classifier_dropout (*float*, *optional*):
The dropout ratio for the classification head.

enable_pronunciation (*bool*, *optional*, defaults to *True*):
Whether or not the model use pronunciation embed when training.

enable_shape (*bool*, *optional*, defaults to *True*):
Whether or not the model use shape embed when training.

pronunciation_embed_dim (*int*, *optional*, defaults to 768):
Dimension of the pronunciation_embed.

pronunciation_vocab_size (*int*, *optional*, defaults to 910):
Pronunciation Vocabulary size of the RoCBert model. Defines the number of different tokens that can be represented by the *input_pronunciation_ids* passed when calling `RoCBertModel`.

shape_embed_dim (*int*, *optional*, defaults to 512):
Dimension of the shape_embed.

shape_vocab_size (*int*, *optional*, defaults to 24858):
Shape Vocabulary size of the RoCBert model. Defines the number of different tokens that can be represented by the *input_shape_ids* passed when calling `RoCBertModel`.

concat_input (*bool*, *optional*, defaults to *True*):
Defines the way of merging the shape_embed, pronunciation_embed and word_embed, if the value is true, output_embed = torch.cat((word_embed, shape_embed, pronunciation_embed), -1), else output_embed = (word_embed + shape_embed + pronunciation_embed) / 3

`class transformers.models.roformer.configuration_roformer.`**`RoFormerConfig`**(*vocab_size=50000*, *embedding_size=None*, *hidden_size=768*, *num_hidden_layers=12*, *num_attention_heads=12*, *intermediate_size=3072*, *hidden_act='gelu'*, *hidden_dropout_prob=0.1*, *attention_probs_dropout_prob=0.1*, *max_position_embeddings=1536*, *type_vocab_size=2*, *initializer_range=0.02*, *layer_norm_eps=1e-12*, *pad_token_id=0*, *rotary_value=False*, *use_cache=True*, ***kwargs*)

The RoFormer model was proposed in RoFormer: Enhanced Transformer with Rotary Position Embedding by Jianlin Su and Yu Lu and Shengfeng Pan and Bo Wen and Yunfeng Liu.

The abstract from the paper is the following:

Position encoding in transformer architecture provides supervision for dependency modeling between elements at different positions in the sequence. We investigate various methods to encode positional information in transformer-based language models and propose a novel implementation named Rotary Position Embedding(RoPE). The proposed RoPE encodes absolute positional information with rotation matrix and naturally incorporates explicit relative position dependency in self-attention formulation. Notably, RoPE comes with valuable properties such as flexibility of being expand to any sequence lengths, decaying inter-token dependency with increasing relative distances, and capability of equipping the linear self-attention with relative position encoding. As a result, the enhanced transformer with rotary position embedding, or RoFormer, achieves superior performance in tasks with long texts. We release the theoretical analysis along with some preliminary experiment results on Chinese data. The undergoing experiment for English benchmark will soon be updated.

Tips:

- RoFormer is a BERT-like autoencoding model with rotary position embeddings. Rotary position embeddings have shown improved performance on classification tasks with long texts.

This model was contributed by junnyu. The original code can be found here.

Args:

vocab_size (*int*, *optional*, defaults to 50000):
Vocabulary size of the RoFormer model. Defines the number of different tokens that can be represented by the *inputs_ids* passed when calling `RoFormerModel` or `TFRoFormerModel`.

embedding_size (*int*, *optional*, defaults to None):
Dimensionality of the encoder layers and the pooler layer. Defaults to the *hidden_size* if not provided.

hidden_size (*int*, *optional*, defaults to 768):
Dimension of the encoder layers and the pooler layer.

num_hidden_layers (*int*, *optional*, defaults to 12):
Number of hidden layers in the Transformer encoder.

num_attention_heads (*int*, *optional*, defaults to 12):
Number of attention heads for each attention layer in the Transformer encoder.

intermediate_size (*int*, *optional*, defaults to 3072):
Dimension of the "intermediate" (i.e., feed-forward) layer in the Transformer encoder.

hidden_act (*str* or *function*, *optional*, defaults to *"gelu"*):
The non-linear activation function (function or string) in the encoder and pooler. If string, *"gelu"*, *"relu"*, *"selu"* and *"gelu_new"* are supported.

hidden_dropout_prob (*float*, *optional*, defaults to 0.1):
The dropout probability for all fully connected layers in the embeddings, encoder, and pooler.

attention_probs_dropout_prob (*float*, *optional*, defaults to 0.1):
The dropout ratio for the attention probabilities.

max_position_embeddings (*int*, *optional*, defaults to 1536):
The maximum sequence length that this model might ever be used with. Typically set this to something large just in case (e.g., 512 or 1024 or 1536).

type_vocab_size (*int*, *optional*, defaults to 2):
The vocabulary size of the *token_type_ids* passed when calling `RoFormerModel` or `TFRoFormerModel`.

initializer_range (*float*, *optional*, defaults to 0.02):
The standard deviation of the truncated_normal_initializer for initializing all weight matrices.

layer_norm_eps (*float*, *optional*, defaults to 1e-12):
The epsilon used by the layer normalization layers.

is_decoder (*bool*, *optional*, defaults to *False*):
Whether the model is used as a decoder or not. If *False*, the model is used as an encoder.

use_cache (*bool*, *optional*, defaults to *True*):
Whether or not the model should return the last key/values attentions (not used by all models). Only relevant if *config.is_decoder=True*.

rotary_value (*bool*, *optional*, defaults to *False*):
Whether or not apply rotary position embeddings on value layer.

`class transformers.models.rwkv.configuration_rwkv.`**`RwkvConfig`**(*vocab_size=50277*, *context_length=1024*, *hidden_size=4096*, *num_hidden_layers=32*, *attention_hidden_size=None*, *intermediate_size=None*, *layer_norm_epsilon=1e-05*, *bos_token_id=0*, *eos_token_id=0*, *rescale_every=6*, *tie_word_embeddings=False*, *use_cache=True*, ***kwargs*)

The RWKV model was proposed in this repo

It suggests a tweak in the traditional Transformer attention to make it linear. This way, the model can be used as recurrent network: passing inputs for timestamp 0 and timestamp 1 together is the same as passing inputs at timestamp 0, then inputs at timestamp 1 along with the state of timestamp 0 (see example below).

This can be more efficient than a regular Transformer and can deal with sentence of any length (even if the model uses a fixed context length for training).

This model was contributed by sgugger. The original code can be found here.

Example of use as an RNN:

```py import torch from transformers import AutoTokenizer, RwkvConfig, RwkvModel

model = RwkvModel.from_pretrained("sgugger/rwkv-430M-pile") tokenizer = AutoTokenizer.from_pretrained("sgugger/rwkv-430M-pile")

inputs = tokenizer("This is an example.", return_tensors="pt") # Feed everything to the model outputs = model(inputs["input_ids"]) output_whole = outputs.last_hidden_state

outputs = model(inputs["input_ids"][:, :2]) output_one = outputs.last_hidden_state

# Using the state computed on the first inputs, we will get the same output outputs = model(inputs["input_ids"][:, 2:], state=outputs.state) output_two = outputs.last_hidden_state

torch.allclose(torch.cat([output_one, output_two], dim=1), output_whole, atol=1e-5) ```

**Args:**

**vocab_size (*int*, *optional*, defaults to 50277):**
Vocabulary size of the RWKV model. Defines the number of different tokens that can be represented by the *inputs_ids* passed when calling `RwkvModel`.

**context_length (*int*, *optional*, defaults to 1024):**
The maximum sequence length that this model can be be used with in a single forward (using it in RNN mode lets use any sequence length).

**hidden_size (*int*, *optional*, defaults to 4096):**
Dimensionality of the embeddings and hidden states.

**num_hidden_layers (*int*, *optional*, defaults to 32):**
Number of hidden layers in the model.

**attention_hidden_size (*int*, *optional*):**
Dimensionality of the attention hidden states. Will default to *hidden_size* if unset.

**intermediate_size (*int*, *optional*):**
Dimensionality of the inner feed-forward layers. Will default to 4 times *hidden_size* if unset.

**layer_norm_epsilon (*float*, *optional*, defaults to 1e-05):**
The epsilon to use in the layer normalization layers.

**bos_token_id (*int*, *optional*, defaults to 0):**
The id of the beginning of sentence token in the vocabulary. Defaults to 0 as RWKV uses the same tokenizer as GPTNeoX.

**eos_token_id (*int*, *optional*, defaults to 0):**
The id of the end of sentence token in the vocabulary. Defaults to 0 as RWKV uses the same tokenizer as GPTNeoX.
```

rescale_every (*int*, *optional*, defaults to 6):
At inference, the hidden states (and weights of the correponding output layers) are divided by 2 every *rescale_every* layer. If set to 0 or a negative number, no rescale is done.

tie_word_embeddings (*bool*, *optional*, defaults to *False*):
Whether or not to tie the word embeddings with the input token embeddings.

use_cache (*bool*, *optional*, defaults to *True*):
Whether or not the model should return the last state.

class transformers.models.splinter.configuration_splinter.**SplinterConfig**(*vocab_size=30522*, *hidden_size=768*, *num_hidden_layers=12*, *num_attention_heads=12*, *intermediate_size=3072*, *hidden_act='gelu'*, *hidden_dropout_prob=0.1*, *attention_probs_dropout_prob=0.1*, *max_position_embeddings=512*, *type_vocab_size=2*, *initializer_range=0.02*, *layer_norm_eps=1e-12*, *use_cache=True*, *pad_token_id=0*, *question_token_id=104*, ***kwargs*)

The Splinter model was proposed in Few-Shot Question Answering by Pretraining Span Selection by Ori Ram, Yuval Kirstain, Jonathan Berant, Amir Globerson, Omer Levy. Splinter is an encoder-only transformer (similar to BERT) pretrained using the recurring span selection task on a large corpus comprising Wikipedia and the Toronto Book Corpus.

The abstract from the paper is the following:

In several question answering benchmarks, pretrained models have reached human parity through fine-tuning on an order of 100,000 annotated questions and answers. We explore the more realistic few-shot setting, where only a few hundred training examples are available, and observe that standard models perform poorly, highlighting the discrepancy between current pretraining objectives and question answering. We propose a new pretraining scheme tailored for question answering: recurring span selection. Given a passage with multiple sets of recurring spans, we mask in each set all recurring spans but one, and ask the model to select the correct span in the passage for each masked span. Masked spans are replaced with a special token, viewed as a question representation, that is later used during fine-tuning to select the answer span. The resulting model obtains surprisingly good results on multiple benchmarks (e.g., 72.7 F1 on SQuAD with only 128 training examples), while maintaining competitive performance in the high-resource setting.

Tips:

- Splinter was trained to predict answers spans conditioned on a special *QUESTION] token. These tokens contextualize to question representations which are used to predict the answers. This layer is called QASS, and is the default behaviour in the ``SplinterForQuestionAnswering`* class. Therefore:
- Use `SplinterTokenizer` (rather than `BertTokenizer`), as it already contains this special token. Also, its default behavior is to use this token when two sequences are given (for example, in the *run_qa.py* script).

- If you plan on using Splinter outside *run_qa.py*, please keep in mind the question token - it might be important for the success of your model, especially in a few-shot setting.
- Please note there are two different checkpoints for each size of Splinter. Both are basically the same, except that one also has the pretrained weights of the QASS layer (*tau/splinter-base-qass* and *tau/splinter-large-qass*) and one doesn't (*tau/splinter-base* and *tau/splinter-large*). This is done to support randomly initializing this layer at fine-tuning, as it is shown to yield better results for some cases in the paper.

This model was contributed by [yuvalkirstain <https://huggingface.co/yuvalkirstain>`__ and oriram. The original code can be found here.

Args:

vocab_size (*int*, *optional*, defaults to 30522):
Vocabulary size of the Splinter model. Defines the number of different tokens that can be represented by the *inputs_ids* passed when calling `SplinterModel`.

hidden_size (*int*, *optional*, defaults to 768):
Dimension of the encoder layers and the pooler layer.

num_hidden_layers (*int*, *optional*, defaults to 12):
Number of hidden layers in the Transformer encoder.

num_attention_heads (*int*, *optional*, defaults to 12):
Number of attention heads for each attention layer in the Transformer encoder.

intermediate_size (*int*, *optional*, defaults to 3072):
Dimension of the "intermediate" (i.e., feed-forward) layer in the Transformer encoder.

hidden_act (*str* or *function*, *optional*, defaults to *"gelu"*):
The non-linear activation function (function or string) in the encoder and pooler. If string, *"gelu"*, *"relu"*, *"selu"* and *"gelu_new"* are supported.

hidden_dropout_prob (*float*, *optional*, defaults to 0.1):
The dropout probability for all fully connected layers in the embeddings, encoder, and pooler.

attention_probs_dropout_prob (*float*, *optional*, defaults to 0.1):
The dropout ratio for the attention probabilities.

max_position_embeddings (*int*, *optional*, defaults to 512):
The maximum sequence length that this model might ever be used with. Typically set this to something large just in case (e.g., 512 or 1024 or 2048).

type_vocab_size (*int*, *optional*, defaults to 2):
The vocabulary size of the *token_type_ids* passed when calling `SplinterModel`.

initializer_range (*float*, *optional*, defaults to 0.02):
The standard deviation of the truncated_normal_initializer for initializing all weight matrices.

layer_norm_eps (*float*, *optional*, defaults to 1e-12):
The epsilon used by the layer normalization layers.

use_cache (*bool*, *optional*, defaults to *True*):
Whether or not the model should return the last key/values attentions (not used by all models). Only relevant if *config.is_decoder=True*.

question_token_id (*int*, *optional*, defaults to 104):
The id of the *[QUESTION]* token.

class transformers.models.squeezebert.configuration_squeezebert.**SqueezeBertConfig**(*vocab_size=30522*, *hidden_size=768*, *num_hidden_layers=12*, *num_attention_heads=12*, *intermediate_size=3072*, *hidden_act='gelu'*, *hidden_dropout_prob=0.1*, *attention_probs_dropout_prob=*(
max_position_embeddings=
type_vocab_size=2, *initializer_range=0.02*, *layer_norm_eps=1e-12*, *pad_token_id=0*, *embedding_size=768*, *q_groups=4*, *k_groups=4*, *v_groups=4*, *post_attention_groups=1*, *intermediate_groups=4*, *output_groups=4*, ***kwargs*)

The SqueezeBERT model was proposed in SqueezeBERT: What can computer vision teach NLP about efficient neural networks? by Forrest N. Iandola, Albert E. Shaw, Ravi Krishna, Kurt W. Keutzer. It's a bidirectional transformer similar to the BERT model. The key difference between the BERT architecture and the SqueezeBERT architecture is that SqueezeBERT uses grouped convolutions instead of fully-connected layers for the Q, K, V and FFN layers.

The abstract from the paper is the following:

Humans read and write hundreds of billions of messages every day. Further, due to the availability of large datasets, large computing systems, and better neural network models, natural language processing (NLP) technology has made significant strides in understanding, proofreading, and organizing these messages. Thus, there is a significant opportunity to deploy NLP in myriad applications to help web users, social networks, and businesses. In particular, we consider smartphones and other mobile devices as crucial platforms for deploying NLP models at scale. However, today's highly-accurate NLP neural network models such as BERT and RoBERTa are extremely computationally expensive, with BERT-base taking 1.7 seconds to classify a text snippet on a Pixel 3 smartphone. In this work, we observe that methods such as grouped convolutions have yielded significant speedups for computer vision networks, but many of these techniques have not been adopted by NLP neural network designers. We demonstrate how to replace several operations in self-attention layers with grouped convolutions, and we use this technique in a novel network architecture called SqueezeBERT, which runs 4.3x faster than BERT-base on the Pixel 3 while achieving competitive accuracy on the GLUE test set. The SqueezeBERT code will be released.

Tips:

- SqueezeBERT is a model with absolute position embeddings so it's usually advised to pad the inputs on the right rather than the left.
- SqueezeBERT is similar to BERT and therefore relies on the masked language modeling (MLM) objective. It is therefore efficient at predicting masked tokens and at NLU in general, but is not optimal for text generation. Models trained with a causal language modeling (CLM) objective are better in that regard.
- For best results when finetuning on sequence classification tasks, it is recommended to start with the *squeezebert/squeezebert-mnli-headless* checkpoint.

This model was contributed by forresti.

Args:

vocab_size (*int*, *optional*, defaults to 30522):
Vocabulary size of the SqueezeBERT model. Defines the number of different tokens that can be represented by the *inputs_ids* passed when calling `SqueezeBertModel`.

hidden_size (*int*, *optional*, defaults to 768):
Dimensionality of the encoder layers and the pooler layer.

num_hidden_layers (*int*, *optional*, defaults to 12):
Number of hidden layers in the Transformer encoder.

num_attention_heads (*int*, *optional*, defaults to 12):
Number of attention heads for each attention layer in the Transformer encoder.

intermediate_size (*int*, *optional*, defaults to 3072):
Dimensionality of the "intermediate" (often named feed-forward) layer in the Transformer encoder.

hidden_act (*str* or *Callable*, *optional*, defaults to *"gelu"*):
The non-linear activation function (function or string) in the encoder and pooler. If string, *"gelu"*, *"relu"*, *"silu"* and *"gelu_new"* are supported.

hidden_dropout_prob (*float*, *optional*, defaults to 0.1):
The dropout probability for all fully connected layers in the embeddings, encoder, and pooler.

attention_probs_dropout_prob (*float*, *optional*, defaults to 0.1):
The dropout ratio for the attention probabilities.

max_position_embeddings (*int*, *optional*, defaults to 512):
The maximum sequence length that this model might ever be used with. Typically set this to something large just in case (e.g., 512 or 1024 or 2048).

type_vocab_size (*int*, *optional*, defaults to 2):
The vocabulary size of the *token_type_ids* passed when calling `BertModel` or `TFBertModel`.

initializer_range (*float*, *optional*, defaults to 0.02):
The standard deviation of the truncated_normal_initializer for initializing all weight matrices.

layer_norm_eps (*float*, *optional*, defaults to 1e-12):

pad_token_id (*int*, *optional*, defaults to 0):
The ID of the token in the word embedding to use as padding.

embedding_size (*int*, *optional*, defaults to 768):
The dimension of the word embedding vectors.

q_groups (*int*, *optional*, defaults to 4):
The number of groups in Q layer.

k_groups (*int*, *optional*, defaults to 4):
The number of groups in K layer.

v_groups (*int*, *optional*, defaults to 4):
The number of groups in V layer.

post_attention_groups (*int*, *optional*, defaults to 1):
The number of groups in the first feed forward network layer.

intermediate_groups (*int*, *optional*, defaults to 4):
The number of groups in the second feed forward network layer.

output_groups (*int*, *optional*, defaults to 4):
The number of groups in the third feed forward network layer.

`class transformers.models.stablelm.configuration_stablelm.StableLmConfig(`*vocab_size=50304*, *intermediate_size=6912*, *hidden_size=2560*, *num_hidden_layers=32*, *num_attention_heads=32*, *num_key_value_heads=32*, *hidden_act='silu'*, *max_position_embeddings=4096*, *initializer_range=0.02*, *layer_norm_eps=1e-05*, *use_cache=True*, *tie_word_embeddings=False*, *rope_theta=10000*, *rope_scaling=None*, *use_qkv_bias=False*, *hidden_dropout=0.0*, *attention_dropout=0.0*, *partial_rotary_factor=0.25*, *bos_token_id=0*, *eos_token_id=0*, ***kwargs*)

StableLM 3B 4E1T was proposed in ``StableLM 3B 4E1T`: Technical Report <https://stability.wandb.io/stability-llm/stable-lm/reports/StableLM-3B-4E1T–VmlldzoyMjU4?accessToken=u3zujipenkx5g7rtcj9qojjgxpconyjktjkli2po09nffrffdhhchq045
by Stability AI and is the first model in a series of multi-epoch pre-trained language models.

#Args:

vocab_size (*int*, *optional*, defaults to 50304):
Vocabulary size of the StableLM model. Defines the number of different tokens that can be represented by the *inputs_ids* passed when calling `StableLmModel`.

intermediate_size (*int*, *optional*, defaults to 6912):
Dimension of the MLP representations.

hidden_size (*int*, *optional*, defaults to 2560):
Number of hidden layers in the Transformer decoder.

num_hidden_layers (*int*, *optional*, defaults to 32):
Number of hidden layers in the Transformer decoder.

num_attention_heads (*int*, *optional*, defaults to 32):
Number of attention heads for each attention layer in the Transformer encoder.

num_key_value_heads (*int*, *optional*, defaults to 32):
This is the number of key_value heads that should be used to implement Grouped Query Attention. If *num_key_value_heads=num_attention_heads*, the model will use Multi Head Attention (MHA), if num_key_value_heads=1 the model will use Multi Query Attention (MQA) otherwise GQA is used. When converting a multi-head checkpoint to a GQA checkpoint, each group key and value head should be constructed by meanpooling all the original heads within that group. For more details checkout `this paper. If it is not specified, will default to *num_attention_heads*.

hidden_act (*str* or *function*, *optional*, defaults to *"silu"*):
The non-linear activation function (function or string).

max_position_embeddings (*int*, *optional*, defaults to 4096):
The maximum sequence length that this model might ever be used with. Typically set this to something large just in case (e.g., 512 or 1024 or 2048).

initializer_range (*float*, *optional*, defaults to 0.02):

The standard deviation of the truncated_normal_initializer for initializing
all weight matrices.

layer_norm_eps (*float*, *optional*, defaults to 1e-05):
The epsilon used by the normalization layers.

use_cache (*bool*, *optional*, defaults to *True*):
Whether or not the model should return the last key/values attentions (not used by all models). Only relevant if *config.is_decoder=True*.

tie_word_embeddings (*bool*, *optional*, defaults to *False*):
Whether the model's input and output word embeddings should be tied.

rope_theta (*float*, *optional*, defaults to *10000.0*):
The base period of the RoPE embeddings.

rope_scaling (*Dict*, *optional*):
Dictionary containing the scaling configuration for the RoPE embeddings. Currently supports two scaling strategies: linear and dynamic. Their scaling factor must be a float greater than 1. The expected format is *{"type": strategy name, "factor": scaling factor}*. When using this flag, don't update *max_position_embeddings* to the expected new maximum. See the following thread for more information on how these scaling strategies behave: https://www.reddit.com/r/LocalLLaMA/comments/14mrgpr/dynamically_scaled_rope_further_increases/. This is an experimental feature, subject to breaking API changes in future versions.

use_qkv_bias (*bool*, *optional*, defaults to *False*):
Whether or not the model should use bias for qkv layers.

hidden_dropout (*float*, *optional*, defaults to 0.0):
The dropout ratio after applying the MLP to the hidden states.

attention_dropout (*float*, *optional*, defaults to 0.0):
The dropout ratio for the attention probabilities.

partial_rotary_factor (*float*, *optional*, defaults to 0.25):
Percentage of the query and keys which will have rotary embedding.

bos_token_id (int, *optional*, defaults to 0):
The id of the *BOS* token in the vocabulary.

eos_token_id (int, *optional*, defaults to 0):
The id of the *EOS* token in the vocabulary.

class transformers.models.starcoder2.configuration_starcoder2.**Starcoder2Config**(*vocab_size=49152*, *hidden_size=3072*, *intermediate_size=12288*, *num_hidden_layers=30*, *num_attention_heads=24*, *num_key_value_heads=2*, *hidden_act='gelu_pytorch_tanh'*, *max_position_embeddings=409* *initializer_range=0.018042*, *norm_epsilon=1e-05*, *use_cache=True*, *bos_token_id=50256*, *eos_token_id=50256*, *rope_theta=10000.0*, *sliding_window=None*, *attention_dropout=0.0*, *residual_dropout=0.0*, *embedding_dropout=0.0*, *use_bias=True*, ***kwargs*)

StarCoder2 is a family of open LLMs for code and comes in 3 different sizes with 3B, 7B and 15B parameters. The flagship StarCoder2-15B model is trained on over 4 trillion tokens and 600+ programming languages from The Stack v2. All models use Grouped Query Attention, a context window of 16,384 tokens with a sliding window attention of 4,096 tokens, and were trained using the Fill-in-the-Middle objective. The models have been released with the paper StarCoder 2 and The Stack v2: The Next Generation by Anton Lozhkov, Raymond Li, Loubna Ben Allal, Federico Cassano, Joel Lamy-Poirier, Nouamane Tazi, Ao Tang, Dmytro Pykhtar, Jiawei Liu, Yuxiang Wei, Tianyang Liu, Max Tian, Denis Kocetkov, Arthur Zucker, Younes Belkada, Zijian Wang, Qian Liu, Dmitry Abulkhanov, Indraneil Paul, Zhuang Li, Wen-Ding Li, Megan Risdal, Jia Li, Jian Zhu, Terry Yue Zhuo, Evgenii Zheltonozhskii, Nii Osae Osae Dade, Wenhao Yu, Lucas Krauß, Naman Jain, Yixuan Su, Xuanli He, Manan Dey, Edoardo Abati, Yekun Chai, Niklas Muennighoff, Xiangru Tang, Muhtasham Oblokulov, Christopher Akiki, Marc Marone, Chenghao Mou, Mayank Mishra, Alex Gu, Binyuan Hui, Tri Dao, Armel Zebaze, Olivier Dehaene, Nicolas Patry, Canwen Xu, Julian McAuley, Han Hu, Torsten Scholak, Sebastien Paquet, Jennifer Robinson, Carolyn Jane Anderson, Nicolas Chapados, Mostofa Patwary, Nima Tajbakhsh, Yacine Jernite, Carlos Muñoz Ferrandis, Lingming Zhang, Sean Hughes, Thomas Wolf, Arjun Guha, Leandro von Werra, and Harm de Vries.

The abstract of the paper is the following:

> The BigCode project, an open-scientific collaboration focused on the responsible development of Large Language Models for Code (Code LLMs), introduces StarCoder2. In partnership with Software Heritage (SWH), we build The Stack v2 on top of the digital commons of their source code archive. Alongside the SWH repositories spanning 619 programming languages, we carefully select other high-quality data sources, such as GitHub pull requests, Kaggle notebooks, and code documentation. This results in a training set that is 4x larger than the first StarCoder dataset. We train StarCoder2 models with 3B, 7B, and 15B parameters on 3.3 to 4.3 trillion tokens and thoroughly evaluate them on a comprehensive set of Code LLM benchmarks. We find that our small model, StarCoder2-3B, outperforms other Code LLMs of similar size on most benchmarks, and also outperforms StarCoderBase-15B. Our large model,

StarCoder2- 15B, significantly outperforms other models of comparable size. In addition, it matches or outperforms CodeLlama-34B, a model more than twice its size. Although DeepSeekCoder- 33B is the best-performing model at code completion for high-resource languages, we find that StarCoder2-15B outperforms it on math and code reasoning benchmarks, as well as several low-resource languages. We make the model weights available under an OpenRAIL license and ensure full transparency regarding the training data by releasing the SoftWare Heritage persistent IDentifiers (SWHIDs) of the source code data. Args:

vocab_size (*int*, *optional*, defaults to 49152):
Vocabulary size of the Starcoder2 model. Defines the number of different tokens that can be represented by the *inputs_ids* passed when calling `Starcoder2Model`

hidden_size (*int*, *optional*, defaults to 3072):
Dimension of the hidden representations.

intermediate_size (*int*, *optional*, defaults to 12288):
Dimension of the MLP representations.

num_hidden_layers (*int*, *optional*, defaults to 30):
Number of hidden layers in the Transformer encoder.

num_attention_heads (*int*, *optional*, defaults to 24):
Number of attention heads for each attention layer in the Transformer encoder.

num_key_value_heads (*int*, *optional*, defaults to 2):
This is the number of key_value heads that should be used to implement Grouped Query Attention. If *num_key_value_heads=num_attention_heads*, the model will use Multi Head Attention (MHA), if num_key_value_heads=1 the model will use Multi Query Attention (MQA) otherwise GQA is used. When converting a multi-head checkpoint to a GQA checkpoint, each group key and value head should be constructed by meanpooling all the original heads within that group. For more details checkout `this paper. If it is not specified, will default to *8*.

hidden_act (*str* or *function*, *optional*, defaults to *"gelu_pytorch_tanh"*):
The non-linear activation function (function or string) in the decoder.

max_position_embeddings (*int*, *optional*, defaults to 4096):
The maximum sequence length that this model might ever be used with. Starcoder2's sliding window attention allows sequence of up to 4096*32 tokens.

initializer_range (*float*, *optional*, defaults to 0.02):
The standard deviation of the truncated_normal_initializer for initializing all weight matrices.

norm_epsilon (*float*, *optional*, defaults to 1e-05):
Epsilon value for the layer norm

use_cache (*bool*, *optional*, defaults to *True*):
Whether or not the model should return the last key/values attentions (not used by all models). Only relevant if *config.is_decoder=True*.

bos_token_id (*int*, *optional*, defaults to 50256):
The id of the "beginning-of-sequence" token.

eos_token_id (*int*, *optional*, defaults to 50256):
The id of the "end-of-sequence" token.

rope_theta (*float*, *optional*, defaults to 10000.0):
The base period of the RoPE embeddings.

sliding_window (*int*, *optional*):
Sliding window attention window size. If not specified, will default to *None* (no sliding window).

attention_dropout (*float*, *optional*, defaults to 0.0):
The dropout ratio for the attention probabilities.

residual_dropout (*float*, *optional*, defaults to 0.0):
Residual connection dropout value.

embedding_dropout (*float*, *optional*, defaults to 0.0):
Embedding dropout.

use_bias (*bool*, *optional*, defaults to *True*):
Whether to use bias term on linear layers of the model.

```
>>> from transformers import Starcoder2Model, Starcoder2Config
```

```
>>> # Initializing a Starcoder2 7B style configuration
>>> configuration = Starcoder2Config()
```

```
>>> # Initializing a model from the Starcoder2 7B style configuration
>>> model = Starcoder2Model(configuration)
```

```
>>> # Accessing the model configuration
>>> configuration = model.config
```

class transformers.models.switch_transformers.configuration_switch_transformers.**SwitchTransformersConfi**

The SwitchTransformers model was proposed in Switch Transformers: Scaling to Trillion Parameter Models with Simple and Efficient Sparsity by William Fedus, Barret Zoph, Noam Shazeer.

The Switch Transformer model uses a sparse T5 encoder-decoder architecture, where the MLP are replaced by a Mixture of Experts (MoE). A routing mechanism (top 1 in this case) associates each token to one of the expert, where each expert is a dense MLP. While switch transformers have a lot more weights than their equivalent dense models, the sparsity allows better scaling and better finetuning performance at scale. During a forward pass, only a fraction of the weights are used. The routing mechanism allows the model to select relevant weights on the fly which increases the model capacity without increasing the number of operations.

The abstract from the paper is the following:

In deep learning, models typically reuse the same parameters for all inputs. Mixture of Experts (MoE) defies this and instead selects different parameters for each incoming example. The result is a sparsely-activated model – with outrageous numbers of parameters – but a constant computational cost. However, despite several notable successes of MoE, widespread adoption has been hindered by complexity, communication costs and training instability – we address these with the Switch Transformer. We simplify the MoE routing algorithm and design intuitive improved models with reduced communication and computational costs. Our proposed training techniques help wrangle the instabilities and

we show large sparse models may be trained, for the first time, with lower precision (bfloat16) formats. We design models based off T5-Base and T5-Large to obtain up to 7x increases in pre-training speed with the same computational resources. These improvements extend into multilingual settings where we measure gains over the mT5-Base version across all 101 languages. Finally, we advance the current scale of language models by pre-training up to trillion parameter models on the "Colossal Clean Crawled Corpus" and achieve a 4x speedup over the T5-XXL model.

Tips:

- SwitchTransformers uses the `T5Tokenizer`, which can be loaded directly from each model's repository.
- The released weights are pretrained on English Masked Language Modeling task, and should be finetuned.

This model was contributed by Younes Belkada and Arthur Zucker . The original code can be found here.

Arguments:

vocab_size (*int*, *optional*, defaults to 32128):
Vocabulary size of the SwitchTransformers model. Defines the number of different tokens that can be represented by the *inputs_ids* passed when calling `SwitchTransformersModel`.

d_model (*int*, *optional*, defaults to 768):
Size of the encoder layers and the pooler layer.

d_kv (*int*, *optional*, defaults to 64):
Size of the key, query, value projections per attention head. *d_kv* has to be equal to *d_model // num_heads*.

d_ff (*int*, *optional*, defaults to 2048):
Size of the intermediate feed forward layer in each *SwitchTransformersBlock*.

expert_capacity (*int*, *optional*, defaults to 64):
Number of tokens that can be stored in each expert. If set to 1, the model will behave like a regular Transformer.

num_layers (*int*, *optional*, defaults to 12):
Number of dense hidden layers in the Transformer encoder layer.

num_sparse_encoder_layers (*int*, *optional*, defaults to 3):
Number of sparse (MoE) dense hidden layers in the Transformer encoder layer.

num_decoder_layers (*int*, *optional*, defaults to 12):
Number of hidden layers in the Transformer decoder. Will use the same value as *num_layers* if not set.

num_sparse_decoder_layers (*int*, *optional*, defaults to 3):
Number of sparse (MoE) dense hidden layers in the Transformer decoder layer.

num_heads (*int*, *optional*, defaults to 12):
Number of attention heads for each attention layer in the Transformer encoder.

num_experts (*int*, *optional*, defaults to 8):
Number of experts for each SwitchTransformer layer.

router_bias (*bool*, *optional*, defaults to *False*):
Whether to add a bias to the router.

router_jitter_noise (*float*, *optional*, defaults to 0.01):
Amount of noise to add to the router.

router_dtype (*str*, *optional*, default to *"float32"*):
The *dtype* used for the routers. It is preferable to keep the *dtype* to *"float32"* as specified in the *selective precision* discussion in the paper.

router_ignore_padding_tokens (*bool*, *optional*, defaults to *False*):
Whether to ignore padding tokens when routing.

relative_attention_num_buckets (*int*, *optional*, defaults to 32):
The number of buckets to use for each attention layer.

relative_attention_max_distance (*int*, *optional*, defaults to 128):
The maximum distance of the longer sequences for the bucket separation.

dropout_rate (*float*, *optional*, defaults to 0.1):
The ratio for all dropout layers.

layer_norm_eps (*float*, *optional*, defaults to 1e-6):
The epsilon used by the layer normalization layers.

router_z_loss_coef (*float*, *optional*, defaults to 0.001):
The z loss factor for the total loss.

router_aux_loss_coef (*float*, *optional*, defaults to 0.001):
The aux loss factor for the total loss.

initializer_factor (*float*, *optional*, defaults to 1.0):
A factor for initializing all weight matrices (should be kept to 1, used internally for initialization testing).

dense_act_fn (*string*, *optional*, defaults to *"relu"*):
Type of feed forward layer to be used. Should be one of *"relu"* or *"gated-gelu"*. SwitchTransformersv1.1 uses the *"gated-gelu"* feed forward projection. Original SwitchTransformers uses *"relu"*.

add_router_probs (*bool*, *optional*, defaults to *False*):
Whether to output router probabilities to compute router auxiliary loss.

use_cache (*bool*, *optional*, defaults to *True*):
Whether or not the model should return the last key/values attentions (not used by all models).

`class transformers.models.t5.configuration_t5.T5Config(`*vocab_size=32128*, *d_model=512*, *d_kv=64*, *d_ff=2048*, *num_layers=6*, *num_decoder_layers=None*, *num_heads=8*, *relative_attention_num_buckets=32*, *relative_attention_max_distance=128*, *dropout_rate=0.1*, *layer_norm_epsilon=1e-06*, *initializer_factor=1.0*, *feed_forward_proj='relu'*, *is_encoder_decoder=True*, *use_cache=True*, *pad_token_id=0*, *eos_token_id=1*, *classifier_dropout=0.0*, ***kwargs*`)`

The T5 model was presented in Exploring the Limits of Transfer Learning with a Unified Text-to-Text Transformer by Colin Raffel, Noam Shazeer, Adam Roberts, Katherine Lee, Sharan Narang, Michael Matena, Yanqi Zhou, Wei Li, Peter J. Liu.

The abstract from the paper is the following:

Transfer learning, where a model is first pre-trained on a data-rich task before being fine-tuned on a downstream task, has emerged as a powerful technique in natural language processing (NLP). The effectiveness of transfer learning has given rise to a diversity of approaches, methodology, and practice. In this paper, we explore the landscape of transfer learning techniques for NLP by introducing a unified framework that converts every language problem into a text-to-text format. Our systematic study compares pretraining objectives, architectures, unlabeled datasets, transfer approaches, and other factors on dozens of language understanding tasks. By combining the insights from our exploration with scale and our new "Colossal Clean Crawled Corpus", we achieve state-of-the-art results on many benchmarks covering summarization, question answering, text classification, and more. To facilitate future work on transfer learning for NLP, we release our dataset, pre-trained models, and code.

Tips:

- T5 is an encoder-decoder model pre-trained on a multi-task mixture of unsupervised and supervised tasks and for which

each task is converted into a text-to-text format. T5 works well on a variety of tasks out-of-the-box by prepending a different prefix to the input corresponding to each task, e.g., for translation: *translate English to German: …*, for summarization: *summarize: ….* - The pretraining includes both supervised and self-supervised training. Supervised training is conducted on downstream tasks provided by the GLUE and SuperGLUE benchmarks (converting them into text-to-text tasks as explained above). - Self-supervised training uses corrupted tokens, by randomly removing 15% of the tokens and replacing them with individual sentinel tokens (if several consecutive tokens are marked for removal, the whole group is replaced with a single sentinel token). The input of the encoder is the corrupted sentence, the input of the decoder is the original sentence and the target is then the dropped out tokens delimited by their sentinel tokens.

- T5 uses relative scalar embeddings. Encoder input padding can be done on the left and on the right.
- See the *training*, *inference* and *scripts* sections below for all details regarding usage.

T5 comes in different sizes:

- t5-small
- t5-base
- t5-large
- t5-3b
- t5-11b.

Based on the original T5 model, Google has released some follow-up works:

- **T5v1.1**: T5v1.1 is an improved version of T5 with some architectural tweaks, and is pre-trained on C4 only without mixing in the supervised tasks. Refer to the documentation of T5v1.1 which can be found here.
- **mT5**: mT5 is a multilingual T5 model. It is pre-trained on the mC4 corpus, which includes 101 languages. Refer to the documentation of mT5 which can be found here.
- **byT5**: byT5 is a T5 model pre-trained on byte sequences rather than SentencePiece subword token sequences. Refer to the documentation of byT5 which can be found here.
- **UL2**: UL2 is a T5 like model pretrained on various denoising objectives
- **Flan-T5: Flan is a pretraining methods that is based on prompting. The Flan-T5 are T5 models trained on the Flan collection of**
  datasets which include: *taskmaster2*, *djaym7/wiki_dialog*, *deepmind/code_contests*, *lambada*, *gsm8k*, *aqua_rat*, *esnli*, *quasc* and *qed*.
- **FLan-UL2** : the UL2 model finetuned using the "Flan" prompt tuning and dataset collection.
- **UMT5**: UmT5 is a multilingual T5 model trained on an improved and refreshed mC4 multilingual corpus, 29 trillion characters across 107 language, using a new sampling method, UniMax. Refer to

  the documentation of mT5 which can be found here.

All checkpoints can be found on the hub.

This model was contributed by thomwolf. The original code can be found here.

Arguments:

vocab_size (*int*, *optional*, defaults to 32128):
Vocabulary size of the T5 model. Defines the number of different tokens that can be represented by the *inputs_ids* passed when calling `T5Model` or `TFT5Model`.

d_model (*int*, *optional*, defaults to 512):
Size of the encoder layers and the pooler layer.

d_kv (*int*, *optional*, defaults to 64):
Size of the key, query, value projections per attention head. The *inner_dim* of the projection layer will be defined as *num_heads * d_kv*.

d_ff (*int*, *optional*, defaults to 2048):
Size of the intermediate feed forward layer in each *T5Block*.

num_layers (*int*, *optional*, defaults to 6):
Number of hidden layers in the Transformer encoder.

num_decoder_layers (*int*, *optional*):
Number of hidden layers in the Transformer decoder. Will use the same value as *num_layers* if not set.

num_heads (*int*, *optional*, defaults to 8):
Number of attention heads for each attention layer in the Transformer encoder.

relative_attention_num_buckets (*int*, *optional*, defaults to 32):
The number of buckets to use for each attention layer.

relative_attention_max_distance (*int*, *optional*, defaults to 128):
The maximum distance of the longer sequences for the bucket separation.

dropout_rate (*float*, *optional*, defaults to 0.1):
The ratio for all dropout layers.

classifier_dropout (*float*, *optional*, defaults to 0.0):
The dropout ratio for classifier.

layer_norm_eps (*float*, *optional*, defaults to 1e-6):
The epsilon used by the layer normalization layers.

initializer_factor (*float*, *optional*, defaults to 1):
A factor for initializing all weight matrices (should be kept to 1, used internally for initialization testing).

feed_forward_proj (*string*, *optional*, defaults to *"relu"*):
Type of feed forward layer to be used. Should be one of *"relu"* or *"gated-gelu"*. T5v1.1 uses the *"gated-gelu"* feed forward projection. Original T5 uses *"relu"*.

use_cache (*bool*, *optional*, defaults to *True*):
Whether or not the model should return the last key/values attentions (not used by all models).

class transformers.models.visual_bert.configuration_visual_bert.**VisualBertConfig**(*vocab_size=30522*, *hidden_size=768*, *visual_embedding_dim=512*, *num_hidden_layers=12*, *num_attention_heads=12*, *intermediate_size=3072*, *hidden_act='gelu'*, *hidden_dropout_prob=0.1*, *attention_probs_dropout_prob=0.*, *max_position_embeddings=5*, *type_vocab_size=2*, *initializer_range=0.02*, *layer_norm_eps=1e-12*, *bypass_transformer=False*, *special_visual_initialize=True*, *pad_token_id=1*, *bos_token_id=0*, *eos_token_id=2*, ***kwargs*)

The VisualBERT model was proposed in VisualBERT: A Simple and Performant Baseline for Vision and Language by Liunian Harold Li, Mark Yatskar, Da Yin, Cho-Jui Hsieh, Kai-Wei Chang. VisualBERT is a neural network trained on a variety of (image, text) pairs.

The abstract from the paper is the following:

We propose VisualBERT, a simple and flexible framework for modeling a broad range of vision-and-language tasks. VisualBERT consists of a stack of Transformer layers that implicitly align elements of an input text and regions in an associated input image with self-attention. We further propose two visually-grounded language model objectives for pre-training VisualBERT on image caption data. Experiments on four vision-and-language tasks including VQA, VCR, NLVR2, and Flickr30K show that VisualBERT outperforms or rivals with state-of-the-art models while being significantly simpler. Further analysis demonstrates that VisualBERT can ground elements of language to image regions without any explicit supervision and is even sensitive to syntactic relationships, tracking, for example, associations between verbs and image regions corresponding to their arguments.

Tips:

1. Most of the checkpoints provided work with the `VisualBertForPreTraining` configuration. Other checkpoints provided are the fine-tuned checkpoints for down-stream tasks - VQA ('visualbert-vqa'), VCR ('visualbert-vcr'), NLVR2 ('visualbert-nlvr2'). Hence, if you are not working on these downstream tasks, it is recommended that you use the pretrained checkpoints.
2. For the VCR task, the authors use a fine-tuned detector for generating visual embeddings, for all the checkpoints. We do not provide the detector and its weights as a part of the package, but it will be available in the research projects, and the states can be loaded directly into the detector provided.

Args:

vocab_size (*int*, *optional*, defaults to 30522):
Vocabulary size of the VisualBERT model. Defines the number of different tokens that can be represented by the *inputs_ids* passed when calling `VisualBertModel`. Vocabulary size of the model. Defines the different tokens that can be represented by the *inputs_ids* passed to the forward method of `VisualBertModel`.

hidden_size (*int*, *optional*, defaults to 768):
Dimensionality of the encoder layers and the pooler layer.

visual_embedding_dim (*int*, *optional*, defaults to 512):
Dimensionality of the visual embeddings to be passed to the model.

num_hidden_layers (*int*, *optional*, defaults to 12):
Number of hidden layers in the Transformer encoder.

num_attention_heads (*int*, *optional*, defaults to 12):
Number of attention heads for each attention layer in the Transformer encoder.

intermediate_size (*int*, *optional*, defaults to 3072):
Dimensionality of the "intermediate" (i.e., feed-forward) layer in the Transformer encoder.

hidden_act (*str* or *function*, *optional*, defaults to *"gelu"*):
The non-linear activation function (function or string) in the encoder and pooler. If string, *"gelu"*, *"relu"*, *"selu"* and *"gelu_new"* are supported.

hidden_dropout_prob (*float*, *optional*, defaults to 0.1):
The dropout probability for all fully connected layers in the embeddings, encoder, and pooler.

attention_probs_dropout_prob (*float*, *optional*, defaults to 0.1):
The dropout ratio for the attention probabilities.

max_position_embeddings (*int*, *optional*, defaults to 512):
The maximum sequence length that this model might ever be used with. Typically set this to something large just in case (e.g., 512 or 1024 or 2048).

type_vocab_size (*int*, *optional*, defaults to 2):
The vocabulary size of the *token_type_ids* passed when calling `VisualBertModel`.

initializer_range (*float*, *optional*, defaults to 0.02):
The standard deviation of the truncated_normal_initializer for initializing all weight matrices.

layer_norm_eps (*float*, *optional*, defaults to 1e-12):
The epsilon used by the layer normalization layers.

bypass_transformer (*bool*, *optional*, defaults to *False*):
Whether or not the model should bypass the transformer for the visual embeddings. If set to *True*, the model directly concatenates the visual embeddings from `VisualBertEmbeddings` with text output from transformers, and then pass it to a self-attention layer.

special_visual_initialize (*bool*, *optional*, defaults to *True*):
Whether or not the visual token type and position type embedding weights should be initialized the same as the textual token type and positive type embeddings. When set to *True*, the weights of the textual token type and position type embeddings are copied to the respective visual embedding layers.

class transformers.models.xglm.configuration_xglm.**XGLMConfig**(*vocab_size=256008*, *max_position_embeddings=2048*, *d_model=1024*, *ffn_dim=4096*, *num_layers=24*, *attention_heads=16*, *activation_function='gelu'*, *dropout=0.1*, *attention_dropout=0.1*, *activation_dropout=0.0*, *layerdrop=0.0*, *init_std=0.02*, *scale_embedding=True*, *use_cache=True*, *decoder_start_token_id=2*, *pad_token_id=1*, *bos_token_id=0*, *eos_token_id=2*, ***kwargs*)

The XGLM model was proposed in Few-shot Learning with Multilingual Language Models by Xi Victoria Lin, Todor Mihaylov, Mikel Artetxe, Tianlu Wang, Shuohui Chen, Daniel Simig, Myle Ott, Naman Goyal, Shruti Bhosale, Jingfei Du, Ramakanth Pasunuru, Sam Shleifer, Punit Singh Koura, Vishrav Chaudhary, Brian O'Horo, Jeff Wang, Luke Zettlemoyer, Zornitsa Kozareva, Mona Diab, Veselin Stoyanov, Xian Li.

The abstract from the paper is the following:

Large-scale autoregressive language models such as GPT-3 are few-shot learners that can perform a wide range of language tasks without fine-tuning. While these models are known to be able to jointly represent many different languages, their training data is dominated by English, potentially limiting their cross-lingual generalization. In this work, we train multilingual autoregressive language models on a balanced corpus covering a diverse set of languages, and study their few- and zero-shot learning capabilities in a wide range of tasks. Our largest model with 7.5 billion parameters sets new state of the art in few-shot learning in more than 20 representative languages, outperforming GPT-3 of comparable size in multilingual commonsense reasoning (with +7.4% absolute accuracy improvement in 0-shot settings and +9.4% in 4-shot settings) and natural language inference (+5.4% in each of 0-shot and 4-shot settings). On the FLORES-101 machine translation benchmark, our model outperforms GPT-3 on 171 out of 182 translation directions with 32 training examples, while surpassing the official supervised baseline in 45 directions. We present a detailed analysis of where the model succeeds and fails, showing in particular that it enables cross-lingual in-context learning on some tasks, while there is still room for improvement on surface form robustness and adaptation to tasks that do not have a natural cloze form. Finally, we evaluate our models in social value tasks such as hate speech detection in five languages and find it has limitations similar to comparable sized GPT-3 models.

This model was contributed by Suraj. The original code can be found here.

Args:

vocab_size (*int*, *optional*, defaults to 256008):
Vocabulary size of the XGLM model. Defines the number of different tokens that can be represented by the *inputs_ids* passed when calling `XGLMModel` or `FlaxXGLMModel`.

max_position_embeddings (*int*, *optional*, defaults to 2048):
The maximum sequence length that this model might ever be used with. Typically set this to something large just in case (e.g., 512 or 1024 or 2048).

d_model (*int*, *optional*, defaults to 1024):
Dimension of the layers and the pooler layer.

ffn_dim (*int*, *optional*, defaults to 4096):
Dimension of the "intermediate" (often named feed-forward) layer in decoder.

num_layers (*int*, *optional*, defaults to 24):
Number of hidden layers Transformer decoder.

attention_heads (*int*, *optional*, defaults to 16):
Number of attention heads for each attention layer in the Transformer decoder.

activation_function (*str* or *function*, *optional*, defaults to *"gelu"*):
The non-linear activation function (function or string) in the encoder and pooler. If string, *"gelu"*, *"relu"*, *"silu"* and *"gelu_new"* are supported.

dropout (*float*, *optional*, defaults to 0.1):
The dropout probability for all fully connected layers in the embeddings, dencoder, and pooler.

attention_dropout (*float*, *optional*, defaults to 0.1):
The dropout ratio for the attention probabilities.

activation_dropout (*float*, *optional*, defaults to 0.0):
The dropout ratio for activations inside the fully connected layer.

layerdrop (*float*, *optional*, defaults to 0.0):
The LayerDrop probability for the encoder. See the LayerDrop paper for more details.

init_std (*float*, *optional*, defaults to 0.02):
The standard deviation of the truncated_normal_initializer for initializing all weight matrices.

scale_embedding (*bool*, *optional*, defaults to *True*):
Scale embeddings by diving by sqrt(d_model).

use_cache (*bool*, *optional*, defaults to *True*):
Whether or not the model should return the last key/values attentions (not used by all models).

`class transformers.models.xlm.configuration_xlm.XLMConfig(`*vocab_size=30145*, *emb_dim=2048*, *n_layers=12*, *n_heads=16*, *dropout=0.1*, *attention_dropout=0.1*, *gelu_activation=True*, *sinusoidal_embeddings=False*, *causal=False*, *asm=False*, *n_langs=1*, *use_lang_emb=True*, *max_position_embeddings=512*, *embed_init_std=0.02209708691207961*, *layer_norm_eps=1e-12*, *init_std=0.02*, *bos_index=0*, *eos_index=1*, *pad_index=2*, *unk_index=3*, *mask_index=5*, *is_encoder=True*, *summary_type='first'*, *summary_use_proj=True*, *summary_activation=None*, *summary_proj_to_labels=True*, *summary_first_dropout=0.1*, *start_n_top=5*, *end_n_top=5*, *mask_token_id=0*, *lang_id=0*, *pad_token_id=2*, *bos_token_id=0*, ***kwargs*)

The XLM model was proposed in Cross-lingual Language Model Pretraining by Guillaume Lample, Alexis Conneau. It's a transformer pretrained using one of the following objectives:

- a causal language modeling (CLM) objective (next token prediction),
- a masked language modeling (MLM) objective (BERT-like), or
- a Translation Language Modeling (TLM) object (extension of BERT's MLM to multiple language inputs)

The abstract from the paper is the following:

Recent studies have demonstrated the efficiency of generative pretraining for English natural language understanding. In this work, we extend this approach to multiple languages and show the effectiveness of cross-lingual pretraining. We propose two methods to learn cross-lingual language models (XLMs): one unsupervised that only relies on monolingual data, and one supervised that leverages parallel data with a new cross-lingual language model objective. We obtain state-of-the-art results on cross-lingual classification, unsupervised and supervised machine translation. On XNLI, our approach pushes the state of the art by an absolute gain of 4.9% accuracy. On unsupervised machine translation, we obtain 34.3 BLEU on WMT'16 German-English, improving the previous state of the art by more than 9 BLEU. On supervised machine translation, we obtain a new state of the art of 38.5 BLEU on WMT'16 Romanian-English, outperforming the previous best approach by more than 4 BLEU. Our code and pretrained models will be made publicly available.

Tips:

- XLM has many different checkpoints, which were trained using different objectives: CLM, MLM or TLM. Make sure to select the correct objective for your task (e.g. MLM checkpoints are not suitable for generation).
- XLM has multilingual checkpoints which leverage a specific *lang* parameter. Check out the multi-lingual page for more information.
- A transformer model trained on several languages. There are three different type of training for this model and the library provides checkpoints for all of them:
  - Causal language modeling (CLM) which is the traditional autoregressive training (so this model could be in the previous section as well). One of the languages is selected for each training sample, and the model input is a sentence of 256 tokens, that may span over several documents in one of those languages.
  - Masked language modeling (MLM) which is like RoBERTa. One of the languages is selected for each training sample, and the model input is a sentence of 256 tokens, that may span over several documents in one of those languages, with dynamic masking of the tokens.
  - A combination of MLM and translation language modeling (TLM). This consists of concatenating a sentence in two different languages, with random masking. To predict one of the masked tokens, the model can use both, the surrounding context in language 1 and the context given by language 2.

This model was contributed by thomwolf. The original code can be found here.

Args:

vocab_size (*int*, *optional*, defaults to 30145):
Vocabulary size of the BERT model. Defines the number of different tokens that can be represented by the *inputs_ids* passed when calling `XLMModel` or `TFXLMModel`.

emb_dim (*int*, *optional*, defaults to 2048):
Dimensionality of the encoder layers and the pooler layer.

n_layer (*int*, *optional*, defaults to 12):
Number of hidden layers in the Transformer encoder.

n_head (*int*, *optional*, defaults to 16):
Number of attention heads for each attention layer in the Transformer encoder.

dropout (*float*, *optional*, defaults to 0.1):
The dropout probability for all fully connected layers in the embeddings, encoder, and pooler.

attention_dropout (*float*, *optional*, defaults to 0.1):
The dropout probability for the attention mechanism

gelu_activation (*bool*, *optional*, defaults to *True*):
Whether or not to use *gelu* for the activations instead of *relu*.

sinusoidal_embeddings (*bool*, *optional*, defaults to *False*):
Whether or not to use sinusoidal positional embeddings instead of absolute positional embeddings.

causal (*bool*, *optional*, defaults to *False*):
Whether or not the model should behave in a causal manner. Causal models use a triangular attention mask in order to only attend to the left-side context instead if a bidirectional context.

asm (*bool*, *optional*, defaults to *False*):
Whether or not to use an adaptive log softmax projection layer instead of a linear layer for the prediction layer.

n_langs (*int*, *optional*, defaults to 1):
The number of languages the model handles. Set to 1 for monolingual models.

use_lang_emb (*bool*, *optional*, defaults to *True*)
Whether to use language embeddings. Some models use additional language embeddings, see the multilingual models page for information on how to use them.

max_position_embeddings (*int*, *optional*, defaults to 512):
The maximum sequence length that this model might ever be used with. Typically set this to something large just in case (e.g., 512 or 1024 or 2048).

embed_init_std (*float*, *optional*, defaults to 2048^-0.5):
The standard deviation of the truncated_normal_initializer for initializing the embedding matrices.

init_std (*int*, *optional*, defaults to 50257):
The standard deviation of the truncated_normal_initializer for initializing all weight matrices except the embedding matrices.

layer_norm_eps (*float*, *optional*, defaults to 1e-12):
The epsilon used by the layer normalization layers.

bos_index (*int*, *optional*, defaults to 0):
The index of the beginning of sentence token in the vocabulary.

eos_index (*int*, *optional*, defaults to 1):
The index of the end of sentence token in the vocabulary.

pad_index (*int*, *optional*, defaults to 2):
The index of the padding token in the vocabulary.

unk_index (*int*, *optional*, defaults to 3):
The index of the unknown token in the vocabulary.

mask_index (*int*, *optional*, defaults to 5):
The index of the masking token in the vocabulary.

is_encoder(*bool*, *optional*, defaults to *True*):
Whether or not the initialized model should be a transformer encoder or decoder as seen in Vaswani et al.

summary_type (*string*, *optional*, defaults to "first"):
Argument used when doing sequence summary. Used in the sequence classification and multiple choice models.

Has to be one of the following options:

- *"last"*: Take the last token hidden state (like XLNet).
- *"first"*: Take the first token hidden state (like BERT).
- *"mean"*: Take the mean of all tokens hidden states.
- *"cls_index"*: Supply a Tensor of classification token position (like GPT/GPT-2).

- *"attn"*: Not implemented now, use multi-head attention.

summary_use_proj (*bool*, *optional*, defaults to *True*):
Argument used when doing sequence summary. Used in the sequence classification and multiple choice models.

Whether or not to add a projection after the vector extraction.

summary_activation (*str*, *optional*):
Argument used when doing sequence summary. Used in the sequence classification and multiple choice models.

Pass *"tanh"* for a tanh activation to the output, any other value will result in no activation.

summary_proj_to_labels (*bool*, *optional*, defaults to *True*):
Used in the sequence classification and multiple choice models.

Whether the projection outputs should have *config.num_labels* or *config.hidden_size* classes.

summary_first_dropout (*float*, *optional*, defaults to 0.1):
Used in the sequence classification and multiple choice models.

The dropout ratio to be used after the projection and activation.

start_n_top (*int*, *optional*, defaults to 5):
Used in the SQuAD evaluation script.

end_n_top (*int*, *optional*, defaults to 5):
Used in the SQuAD evaluation script.

mask_token_id (*int*, *optional*, defaults to 0):
Model agnostic parameter to identify masked tokens when generating text in an MLM context.

lang_id (*int*, *optional*, defaults to 1):
The ID of the language used by the model. This parameter is used when generating text in a given language.

class transformers.models.xlm_prophetnet.configuration_xlm_prophetnet.**XLMProphetNetConfig**(*activation_dropout*: *float* | *None* = *0.1*, *activation_function*: *str* | *Callable* | *NoneType* = *'gelu'*, *vocab_size*: *int* | *None* = *30522*, *hidden_size*: *int* | *None* = *1024*, *encoder_ffn_dim*: *int* | *None* = *4096*, *num_encoder_layers*: *int* | *None* = *12*, *num_encoder_attention_heads*: *int* | *None* = *16*, *decoder_ffn_dim*: *int* | *None* = *4096*, *num_decoder_layers*: *int*

The XLM-ProphetNet model was proposed in ProphetNet: Predicting Future N-gram for Sequence-to-Sequence Pre-training, by Yu Yan, Weizhen Qi, Yeyun Gong, Dayiheng Liu, Nan Duan, Jiusheng Chen, Ruofei Zhang, Ming Zhou on 13 Jan, 2020.

XLM-ProphetNet is an encoder-decoder model and can predict n-future tokens for "ngram" language modeling instead of just the next token. Its architecture is identical to ProhpetNet, but the model was trained on the multi-lingual "wiki100" Wikipedia dump.

The abstract from the paper is the following:

In this paper, we present a new sequence-to-sequence pretraining model called ProphetNet, which introduces a novel self-supervised objective named future n-gram prediction and the proposed n-stream self-attention mechanism. Instead of the optimization of one-step ahead prediction in traditional sequence-to-sequence model, the ProphetNet is optimized by n-step ahead prediction which predicts the next n tokens simultaneously based on previous context tokens at each time step. The future n-gram prediction explicitly encourages the model to plan for the future tokens and prevent overfitting on strong local correlations. We pre-train ProphetNet using a base scale dataset (16GB) and a large scale dataset (160GB) respectively. Then we conduct experiments on CNN/DailyMail, Gigaword, and SQuAD 1.1 benchmarks for abstractive summarization and question generation tasks. Experimental results show that ProphetNet achieves new state-of-the-art results on all these datasets compared to the models using the same scale pretraining corpus.

The Authors' code can be found here.

Tips:

- XLM-ProphetNet's model architecture and pretraining objective is same as ProphetNet, but XLM-ProphetNet was pre-trained on the cross-lingual dataset XGLUE.

Args:

activation_dropout (*float*, *optional*, defaults to 0.1):
The dropout ratio for activations inside the fully connected layer.

activation_function (*str* or *function*, *optional*, defaults to *"gelu"*):
The non-linear activation function (function or string) in the encoder and pooler. If string, *"gelu"*, *"relu"*, *"silu"* and *"gelu_new"* are supported.

vocab_size (*int*, *optional*, defaults to 30522):
Vocabulary size of the ProphetNET model. Defines the number of different tokens that can be represented by the *inputs_ids* passed when calling `XLMProphetNetModel`.

hidden_size (*int*, *optional*, defaults to 1024):
Dimensionality of the layers and the pooler layer.

encoder_ffn_dim (*int*, *optional*, defaults to 4096):
Dimensionality of the "intermediate" (often named feed-forward) layer in decoder.

num_encoder_layers (*int*, *optional*, defaults to 12):
Number of encoder layers.

num_encoder_attention_heads (*int*, *optional*, defaults to 16):
Number of attention heads for each attention layer in the Transformer encoder.

decoder_ffn_dim (*int*, *optional*, defaults to 4096):
Dimensionality of the *intermediate* (often named feed-forward) layer in decoder.

num_decoder_layers (*int*, *optional*, defaults to 12):
Number of decoder layers.

num_decoder_attention_heads (*int*, *optional*, defaults to 16):
Number of attention heads for each attention layer in the Transformer decoder.

attention_dropout (*float*, *optional*, defaults to 0.1):
The dropout ratio for the attention probabilities.

dropout (*float*, *optional*, defaults to 0.1):
The dropout probability for all fully connected layers in the embeddings, encoder, and pooler.

max_position_embeddings (*int*, *optional*, defaults to 512):
The maximum sequence length that this model might ever be used with. Typically set this to something large just in case (e.g., 512 or 1024 or 2048).

init_std (*float*, *optional*, defaults to 0.02):
The standard deviation of the truncated_normal_initializer for initializing all weight matrices.

add_cross_attention (*bool*, *optional*, defaults to *True*):
Whether cross-attention layers should be added to the model.

is_encoder_decoder (*bool*, *optional*, defaults to *True*):
Whether this is an encoder/decoder model.

pad_token_id (*int*, *optional*, defaults to 1)
Padding token id.

bos_token_id (*int*, *optional*, defaults to 0)
Beginning of stream token id.

eos_token_id (*int*, *optional*, defaults to 2)
End of stream token id.

ngram (*int*, *optional*, defaults to 2)
Number of future tokens to predict. Set to 1 to be same as traditional Language model to predict next first token.

num_buckets (*int*, *optional*, defaults to 32)
The number of buckets to use for each attention layer. This is for relative position calculation. See the T5 paper for more details.

relative_max_distance (*int*, *optional*, defaults to 128)
Relative distances greater than this number will be put into the last same bucket. This is for relative position calculation. See the T5 paper for more details.

disable_ngram_loss (*bool*, *optional*, defaults to *False*):
Whether be trained predicting only the next first token.

eps (*float*, *optional*, defaults to 0.0):
Controls the *epsilon* parameter value for label smoothing in the loss calculation. If set to 0, no label smoothing is performed.

use_cache (*bool*, *optional*, defaults to *True*):
Whether or not the model should return the last key/values attentions (not used by all models).

class transformers.models.xlm_roberta.configuration_xlm_roberta.**XLMRobertaConfig**(*vocab_size=30522*, *hidden_size=768*, *num_hidden_layers=12*, *num_attention_heads=12*, *intermediate_size=3072*, *hidden_act='gelu'*, *hidden_dropout_prob=0.1*, *attention_probs_dropout_prob=0.*, *max_position_embeddings=5*, *type_vocab_size=2*, *initializer_range=0.02*, *layer_norm_eps=1e-12*, *pad_token_id=1*, *bos_token_id=0*, *eos_token_id=2*, *position_embedding_type='absol*, *use_cache=True*, *classifier_dropout=None*, ***kwargs*)

The XLM-RoBERTa model was proposed in Unsupervised Cross-lingual Representation Learning at Scale by Alexis Conneau, Kartikay Khandelwal, Naman Goyal, Vishrav Chaudhary, Guillaume Wenzek, Francisco Guzmán, Edouard Grave, Myle Ott, Luke Zettlemoyer and Veselin Stoyanov. It is based on Facebook's RoBERTa model released in 2019. It is a large multi-lingual language model, trained on 2.5TB of filtered CommonCrawl data.

The abstract from the paper is the following:

This paper shows that pretraining multilingual language models at scale leads to significant performance gains for a wide range of cross-lingual transfer tasks. We train a Transformer-based masked language model on one hundred languages, using more than two terabytes of filtered CommonCrawl data. Our model, dubbed XLM-R, significantly outperforms multilingual BERT (mBERT) on a variety of cross-lingual benchmarks, including +13.8% average accuracy on XNLI, +12.3% average F1 score on MLQA, and +2.1% average F1 score on NER. XLM-R performs particularly well on low-resource languages, improving 11.8% in XNLI accuracy for Swahili and 9.2% for Urdu over the previous XLM model. We also present a detailed empirical evaluation of the key factors that are required to achieve these gains, including the trade-offs between (1) positive transfer and capacity dilution and (2) the performance of high and low resource languages at scale. Finally, we show, for the first time, the possibility of multilingual modeling without sacrificing per-language performance; XLM-Ris very competitive with strong monolingual models on the GLUE and XNLI benchmarks. We will make XLM-R code, data, and models publicly available.

Tips:

- XLM-RoBERTa is a multilingual model trained on 100 different languages. Unlike some XLM multilingual models, it does not require *lang* tensors to understand which language is used, and should be able to determine the correct language from the input ids.
- Uses RoBERTa tricks on the XLM approach, but does not use the translation language modeling objective. It only uses masked language modeling on sentences coming from one language.

- This implementation is the same as RoBERTa. Refer to the documentation of RoBERTa for usage examples as well as the information relative to the inputs and outputs.

This model was contributed by stefan-it. The original code can be found here.

Args:

vocab_size (*int*, *optional*, defaults to 30522):
Vocabulary size of the XLM-RoBERTa model. Defines the number of different tokens that can be represented by the *inputs_ids* passed when calling `XLMRobertaModel` or `TFXLMRobertaModel`.

hidden_size (*int*, *optional*, defaults to 768):
Dimensionality of the encoder layers and the pooler layer.

num_hidden_layers (*int*, *optional*, defaults to 12):
Number of hidden layers in the Transformer encoder.

num_attention_heads (*int*, *optional*, defaults to 12):
Number of attention heads for each attention layer in the Transformer encoder.

intermediate_size (*int*, *optional*, defaults to 3072):
Dimensionality of the "intermediate" (often named feed-forward) layer in the Transformer encoder.

hidden_act (*str* or *Callable*, *optional*, defaults to *"gelu"*):
The non-linear activation function (function or string) in the encoder and pooler. If string, *"gelu"*, *"relu"*, *"silu"* and *"gelu_new"* are supported.

hidden_dropout_prob (*float*, *optional*, defaults to 0.1):
The dropout probability for all fully connected layers in the embeddings, encoder, and pooler.

attention_probs_dropout_prob (*float*, *optional*, defaults to 0.1):
The dropout ratio for the attention probabilities.

max_position_embeddings (*int*, *optional*, defaults to 512):
The maximum sequence length that this model might ever be used with. Typically set this to something large just in case (e.g., 512 or 1024 or 2048).

type_vocab_size (*int*, *optional*, defaults to 2):
The vocabulary size of the *token_type_ids* passed when calling `XLMRobertaModel` or `TFXLMRobertaModel`.

initializer_range (*float*, *optional*, defaults to 0.02):
The standard deviation of the truncated_normal_initializer for initializing all weight matrices.

layer_norm_eps (*float*, *optional*, defaults to 1e-12):
The epsilon used by the layer normalization layers.

position_embedding_type (*str*, *optional*, defaults to *"absolute"*):
Type of position embedding. Choose one of *"absolute"*, *"relative_key"*, *"relative_key_query"*. For positional embeddings use *"absolute"*. For more information on *"relative_key"*, please refer to Self-Attention with Relative Position Representations (Shaw et al.). For more information on *"relative_key_query"*, please refer to *Method 4* in Improve Transformer Models with Better Relative Position Embeddings (Huang et al.).

is_decoder (*bool*, *optional*, defaults to *False*):
Whether the model is used as a decoder or not. If *False*, the model is used as an encoder.

use_cache (*bool*, *optional*, defaults to *True*):
Whether or not the model should return the last key/values attentions (not used by all models). Only relevant if *config.is_decoder=True*.

classifier_dropout (*float*, *optional*):
The dropout ratio for the classification head.

class transformers.models.xlm_roberta_xl.configuration_xlm_roberta_xl.**XLMRobertaXLConfig**(*vocab_size=25088 hidden_size=2560, num_hidden_layer num_attention_hea intermediate_size=10240, hidden_act='gelu', hidden_dropout_prob attention_probs_dropou max_position_emb type_vocab_size=1 initializer_range=0.02, layer_norm_eps=1 05, pad_token_id=1, bos_token_id=0, eos_token_id=2, position_embedding_ty use_cache=True, classifier_dropout=None **kwargs*)

The XLM-RoBERTa-XL model was proposed in Larger-Scale Transformers for Multilingual Masked Language Modeling by Naman Goyal, Jingfei Du, Myle Ott, Giri Anantharaman, Alexis Conneau.

The abstract from the paper is the following:

Recent work has demonstrated the effectiveness of cross-lingual language model pretraining for cross-lingual understanding. In this study, we present the results of two larger multilingual masked language models, with 3.5B and 10.7B parameters. Our two new models dubbed XLM-R XL and XLM-R XXL outperform XLM-R by 1.8% and 2.4% average accuracy on XNLI. Our model also outperforms the RoBERTa-Large model on several English tasks of the GLUE benchmark by 0.3% on average while handling 99 more languages. This suggests pretrained models with larger capacity may obtain both strong performance on high-resource languages while greatly improving low-resource languages. We make our code and models publicly available.

Tips:

- XLM-RoBERTa-XL is a multilingual model trained on 100 different languages. Unlike some XLM multilingual models, it does not require *lang* tensors to understand which language is used, and should be able to determine the correct language from the input ids.

This model was contributed by Soonhwan-Kwon and stefan-it. The original code can be found here.

Args:

vocab_size (*int*, *optional*, defaults to 250880):
Vocabulary size of the XLM_ROBERTA_XL model. Defines the number of different tokens that can be represented by the *inputs_ids* passed when calling `XLMRobertaXLModel`.

hidden_size (*int*, *optional*, defaults to 2560):
Dimensionality of the encoder layers and the pooler layer.

num_hidden_layers (*int*, *optional*, defaults to 36):
Number of hidden layers in the Transformer encoder.

num_attention_heads (*int*, *optional*, defaults to 32):
Number of attention heads for each attention layer in the Transformer encoder.

intermediate_size (*int*, *optional*, defaults to 10240):
Dimensionality of the "intermediate" (often named feed-forward) layer in the Transformer encoder.

hidden_act (*str* or *Callable*, *optional*, defaults to *"gelu"*):
The non-linear activation function (function or string) in the encoder and pooler. If string, *"gelu"*, *"relu"*, *"silu"* and *"gelu_new"* are supported.

hidden_dropout_prob (*float*, *optional*, defaults to 0.1):
The dropout probability for all fully connected layers in the embeddings, encoder, and pooler.

attention_probs_dropout_prob (*float*, *optional*, defaults to 0.1):
The dropout ratio for the attention probabilities.

max_position_embeddings (*int*, *optional*, defaults to 514):
The maximum sequence length that this model might ever be used with. Typically set this to something large just in case (e.g., 512 or 1024 or 2048).

type_vocab_size (*int*, *optional*, defaults to 1):
The vocabulary size of the *token_type_ids* passed when calling `XLMRobertaXLModel` or `TFXLMRobertaXLModel`.

initializer_range (*float*, *optional*, defaults to 0.02):
The standard deviation of the truncated_normal_initializer for initializing all weight matrices.

layer_norm_eps (*float*, *optional*, defaults to 1e-5):
The epsilon used by the layer normalization layers.

position_embedding_type (*str*, *optional*, defaults to *"absolute"*):
Type of position embedding. Choose one of *"absolute"*, *"relative_key"*, *"relative_key_query"*. For positional embeddings use *"absolute"*. For more information on *"relative_key"*, please refer to Self-Attention with Relative Position Representations (Shaw et al.). For more information on *"relative_key_query"*, please refer to *Method 4* in Improve Transformer Models with Better Relative Position Embeddings (Huang et al.).

use_cache (*bool*, *optional*, defaults to *True*):
Whether or not the model should return the last key/values attentions (not used by all models). Only relevant if *config.is_decoder=True*.

classifier_dropout (*float*, *optional*):
The dropout ratio for the classification head.

class transformers.models.xlnet.configuration_xlnet.**XLNetConfig**(*vocab_size=32000*, *d_model=1024*, *n_layer=24*, *n_head=16*, *d_inner=4096*, *ff_activation='gelu'*, *untie_r=True*, *attn_type='bi'*, *initializer_range=0.02*, *layer_norm_eps=1e-12*, *dropout=0.1*, *mem_len=512*, *reuse_len=None*, *use_mems_eval=True*, *use_mems_train=False*, *bi_data=False*, *clamp_len=-1*, *same_length=False*, *summary_type='last'*, *summary_use_proj=True*, *summary_activation='tanh'*, *summary_last_dropout=0.1*, *start_n_top=5*, *end_n_top=5*, *pad_token_id=5*, *bos_token_id=1*, *eos_token_id=2*, ***kwargs*)

The XLNet model was proposed in XLNet: Generalized Autoregressive Pretraining for Language Understanding by Zhilin Yang, Zihang Dai, Yiming Yang, Jaime Carbonell, Ruslan Salakhutdinov, Quoc V. Le. XLnet is an extension of the Transformer-XL model pre-trained using an autoregressive method to learn bidirectional contexts by maximizing the expected likelihood over all permutations of the input sequence factorization order.

The abstract from the paper is the following:

With the capability of modeling bidirectional contexts, denoising autoencoding based pretraining like BERT achieves better performance than pretraining approaches based on autoregressive language modeling. However, relying on corrupting the input with masks, BERT neglects dependency between the masked positions and suffers from a pretrain-finetune discrepancy. In light of these pros and cons, we propose XLNet, a generalized autoregressive pretraining method that (1) enables learning bidirectional contexts by maximizing the expected likelihood over all permutations of the factorization order and (2) overcomes the limitations of BERT thanks to its autoregressive formulation. Furthermore, XLNet integrates ideas from Transformer-XL, the state-of-the-art autoregressive model, into pretraining. Empirically, under comparable experiment settings, XLNet outperforms BERT on 20 tasks, often by a large margin, including question answering, natural language inference, sentiment analysis, and document ranking.

Tips:

- The specific attention pattern can be controlled at training and test time using the *perm_mask* input.
- Due to the difficulty of training a fully auto-regressive model over various factorization order, XLNet is pretrained using only a sub-set of the output tokens as target which are selected with the *target_mapping* input.
- To use XLNet for sequential decoding (i.e. not in fully bi-directional setting), use the *perm_mask* and *target_mapping* inputs to control the attention span and outputs (see examples in *examples/pytorch/text-generation/run_generation.py*)
- XLNet is one of the few models that has no sequence length limit.
- XLNet is not a traditional autoregressive model but uses a training strategy that builds on that. It permutes the tokens in the sentence, then allows the model to use the last n tokens to predict the token n+1. Since this is all done with a mask, the sentence is actually fed in the model in the right order, but instead of masking the first n tokens for n+1, XLNet uses a mask that hides the previous tokens in some given permutation of 1,…,sequence length.
- XLNet also uses the same recurrence mechanism as Transformer-XL to build long-term dependencies.

This model was contributed by thomwolf. The original code can be found here.

Args:

vocab_size (*int*, *optional*, defaults to 32000):
Vocabulary size of the XLNet model. Defines the number of different tokens that can be represented by the *inputs_ids* passed when calling `XLNetModel` or `TFXLNetModel`.

d_model (*int*, *optional*, defaults to 1024):
Dimensionality of the encoder layers and the pooler layer.

n_layer (*int*, *optional*, defaults to 24):
Number of hidden layers in the Transformer encoder.

n_head (*int*, *optional*, defaults to 16):
Number of attention heads for each attention layer in the Transformer encoder.

d_inner (*int*, *optional*, defaults to 4096):
Dimensionality of the "intermediate" (often named feed-forward) layer in the Transformer encoder.

ff_activation (*str* or *Callable*, *optional*, defaults to *"gelu"*):
The non-linear activation function (function or string) in the If string, *"gelu"*, *"relu"*, *"silu"* and *"gelu_new"* are supported.

untie_r (*bool*, *optional*, defaults to *True*):
Whether or not to untie relative position biases

attn_type (*str*, *optional*, defaults to *"bi"*):
The attention type used by the model. Set *"bi"* for XLNet, *"uni"* for Transformer-XL.

initializer_range (*float*, *optional*, defaults to 0.02):
The standard deviation of the truncated_normal_initializer for initializing all weight matrices.

layer_norm_eps (*float*, *optional*, defaults to 1e-12):
The epsilon used by the layer normalization layers.

dropout (*float*, *optional*, defaults to 0.1):
The dropout probability for all fully connected layers in the embeddings, encoder, and pooler.

mem_len (*int* or *None*, *optional*):
The number of tokens to cache. The key/value pairs that have already been pre-computed in a previous forward pass won't be re-computed. See the quickstart for more information.

reuse_len (*int*, *optional*):
The number of tokens in the current batch to be cached and reused in the future.

bi_data (*bool*, *optional*, defaults to *False*):
Whether or not to use bidirectional input pipeline. Usually set to *True* during pretraining and *False* during finetuning.

clamp_len (*int*, *optional*, defaults to -1):
Clamp all relative distances larger than clamp_len. Setting this attribute to -1 means no clamping.

same_length (*bool*, *optional*, defaults to *False*):
Whether or not to use the same attention length for each token.

summary_type (*str*, *optional*, defaults to "last"):
Argument used when doing sequence summary. Used in the sequence classification and multiple choice models.

Has to be one of the following options:

- *"last"*: Take the last token hidden state (like XLNet).

- *"first"*: Take the first token hidden state (like BERT).
- *"mean"*: Take the mean of all tokens hidden states.
- *"cls_index"*: Supply a Tensor of classification token position (like GPT/GPT-2).
- *"attn"*: Not implemented now, use multi-head attention.

summary_use_proj (*bool*, *optional*, defaults to *True*):
Argument used when doing sequence summary. Used in the sequence classification and multiple choice models.

Whether or not to add a projection after the vector extraction.

summary_activation (*str*, *optional*):
Argument used when doing sequence summary. Used in the sequence classification and multiple choice models.

Pass *"tanh"* for a tanh activation to the output, any other value will result in no activation.

summary_proj_to_labels (*boo*, *optional*, defaults to *True*):
Used in the sequence classification and multiple choice models.

Whether the projection outputs should have *config.num_labels* or *config.hidden_size* classes.

summary_last_dropout (*float*, *optional*, defaults to 0.1):
Used in the sequence classification and multiple choice models.

The dropout ratio to be used after the projection and activation.

start_n_top (*int*, *optional*, defaults to 5):
Used in the SQuAD evaluation script.

end_n_top (*int*, *optional*, defaults to 5):
Used in the SQuAD evaluation script.

use_mems_eval (*bool*, *optional*, defaults to *True*):
Whether or not the model should make use of the recurrent memory mechanism in evaluation mode.

use_mems_train (*bool*, *optional*, defaults to *False*):
Whether or not the model should make use of the recurrent memory mechanism in train mode.

<Tip>

For pretraining, it is recommended to set *use_mems_train* to *True*. For fine-tuning, it is recommended to set *use_mems_train* to *False* as discussed here. If *use_mems_train* is set to *True*, one has to make sure that the train batches are correctly pre-processed, *e.g. batch_1 = [[This line is], [This is the]]* and *batch_2 = [[the first line], [second line]]* and that all batches are of equal size.

</Tip>

class transformers.models.yoso.configuration_yoso.**YosoConfig**(*vocab_size=50265*, *hidden_size=768*, *num_hidden_layers=12*, *num_attention_heads=12*, *intermediate_size=3072*, *hidden_act='gelu'*, *hidden_dropout_prob=0.1*, *attention_probs_dropout_prob=0.1*, *max_position_embeddings=4096*, *type_vocab_size=1*, *initializer_range=0.02*, *layer_norm_eps=1e-12*, *position_embedding_type='absolute'*, *use_expectation=True*, *hash_code_len=9*, *num_hash=64*, *conv_window=None*, *use_fast_hash=True*, *lsh_backward=True*, *pad_token_id=1*, *bos_token_id=0*, *eos_token_id=2*, ***kwargs*)

The YOSO model was proposed in You Only Sample (Almost) Once: Linear Cost Self-Attention Via Bernoulli Sampling by Zhanpeng Zeng, Yunyang Xiong, Sathya N. Ravi, Shailesh Acharya, Glenn Fung, Vikas Singh. YOSO approximates standard softmax self-attention via a Bernoulli sampling scheme based on Locality Sensitive Hashing (LSH). In principle, all the Bernoulli random variables can be sampled with a single hash.

The abstract from the paper is the following:

Transformer-based models are widely used in natural language processing (NLP). Central to the transformer model is the self-attention mechanism, which captures the interactions of token pairs in the input sequences and depends quadratically on the sequence length. Training such models on longer sequences is expensive. In this paper, we show that a Bernoulli sampling attention mechanism based on Locality Sensitive Hashing (LSH), decreases the quadratic complexity of such models to linear. We bypass the quadratic cost by considering self-attention as a sum of individual tokens associated with Bernoulli random variables that can, in principle, be sampled at once by a single hash (although in practice, this number may be a small constant). This leads to an efficient sampling scheme to estimate self-attention which relies on specific modifications of LSH (to enable deployment on GPU architectures). We evaluate our algorithm on the GLUE benchmark with standard 512 sequence length where we see favorable performance relative to a standard pretrained Transformer. On the Long Range Arena (LRA) benchmark, for evaluating performance on long sequences, our method achieves results consistent with softmax self-attention but with sizable speed-ups and memory savings and often outperforms other efficient self-attention methods. Our code is available at this https URL

Tips:

- The YOSO attention algorithm is implemented through custom CUDA kernels, functions written in CUDA C++ that can be executed multiple times

in parallel on a GPU. - The kernels provide a *fast_hash* function, which approximates the random projections of the queries and keys using the Fast Hadamard Transform. Using these hash codes, the *lsh_cumulation* function approximates self-attention via LSH-based Bernoulli sampling. - To use the custom kernels, the user should set *config.use_expectation = False*. To ensure that the kernels are compiled successfully, the user must install the correct version of PyTorch and cudatoolkit. By default, *config.use_expectation = True*, which uses YOSO-E and does not require compiling CUDA kernels.

<small> YOSO Attention Algorithm. Taken from the original paper.</small>

This model was contributed by novice03. The original code can be found here.

Args:

vocab_size (*int*, *optional*, defaults to 50265):
Vocabulary size of the YOSO model. Defines the number of different tokens that can be represented by the *inputs_ids* passed when calling `YosoModel`.

hidden_size (*int*, *optional*, defaults to 768):
Dimension of the encoder layers and the pooler layer.

num_hidden_layers (*int*, *optional*, defaults to 12):
Number of hidden layers in the Transformer encoder.

num_attention_heads (*int*, *optional*, defaults to 12):
Number of attention heads for each attention layer in the Transformer encoder.

intermediate_size (*int*, *optional*, defaults to 3072):
Dimension of the "intermediate" (i.e., feed-forward) layer in the Transformer encoder.

hidden_act (*str* or *function*, *optional*, defaults to *"gelu"*):
The non-linear activation function (function or string) in the encoder and pooler. If string, *"gelu"*, *"relu"*, *"selu"* and *"gelu_new"* are supported.

hidden_dropout_prob (*float*, *optional*, defaults to 0.1):
The dropout probability for all fully connected layers in the embeddings, encoder, and pooler.

attention_probs_dropout_prob (*float*, *optional*, defaults to 0.1):
The dropout ratio for the attention probabilities.

max_position_embeddings (*int*, *optional*, defaults to 512):
The maximum sequence length that this model might ever be used with. Typically set this to something large just in case (e.g., 512 or 1024 or 2048).

type_vocab_size (*int*, *optional*, defaults to 2):
The vocabulary size of the *token_type_ids* passed when calling `YosoModel`.

initializer_range (*float*, *optional*, defaults to 0.02):
The standard deviation of the truncated_normal_initializer for initializing all weight matrices.

layer_norm_eps (*float*, *optional*, defaults to 1e-12):
The epsilon used by the layer normalization layers.

position_embedding_type (*str*, *optional*, defaults to *"absolute"*):
Type of position embedding. Choose one of *"absolute"*, *"relative_key"*, *"relative_key_query"*.

use_expectation (*bool*, *optional*, defaults to *True*):
Whether or not to use YOSO Expectation. Overrides any effect of num_hash.

hash_code_len (*int*, *optional*, defaults to 9):
The length of hashes generated by the hash functions.

num_hash (*int*, *optional*, defaults to 64):
Number of hash functions used in `YosoSelfAttention`.

conv_window (*int*, *optional*):
Kernel size of depth-wise convolution.

use_fast_hash (*bool*, *optional*, defaults to *False*):
Whether or not to use custom cuda kernels which perform fast random projection via hadamard transform.

lsh_backward (*bool*, *optional*, defaults to *True*):
Whether or not to perform backpropagation using Locality Sensitive Hashing.

2.7 License

GNU AFFERO GENERAL PUBLIC LICENSE
Version 3, 19 November 2007

Preamble

The GNU Affero General Public License is a free, copyleft license for

software and other kinds of works, specifically designed to ensure cooperation with the community in the case of network server software.

The licenses for most software and other practical works are designed

to take away your freedom to share and change the works. By contrast, our General Public Licenses are intended to guarantee your freedom to share and change all versions of a program–to make sure it remains free software for all its users.

When we speak of free software, we are referring to freedom, not

price. Our General Public Licenses are designed to make sure that you have the freedom to distribute copies of free software (and charge for them if you wish), that you receive source code or can get it if you want it, that you can change the software or use pieces of it in new free programs, and that you know you can do these things.

Developers that use our General Public Licenses protect your rights

with two steps: (1) assert copyright on the software, and (2) offer you this License which gives you legal permission to copy, distribute and/or modify the software.

A secondary benefit of defending all users' freedom is that

improvements made in alternate versions of the program, if they receive widespread use, become available for other developers to incorporate. Many developers of free software are heartened and encouraged by the resulting cooperation. However, in the case of software used on network servers, this result may fail to come about. The GNU General Public License permits making a modified version and letting the public access it on a server without ever releasing its source code to the public.

The GNU Affero General Public License is designed specifically to

ensure that, in such cases, the modified source code becomes available to the community. It requires the operator of a network server to provide the source code of the modified version running there to the users of that server. Therefore, public use of a modified version, on a publicly accessible server, gives the public access to the source code of the modified version.

An older license, called the Affero General Public License and

published by Affero, was designed to accomplish similar goals. This is a different license, not a version of the Affero GPL, but Affero has released a new version of the Affero GPL which permits relicensing under this license.

The precise terms and conditions for copying, distribution and

modification follow.

TERMS AND CONDITIONS

0. Definitions.

"This License" refers to version 3 of the GNU Affero General Public License.

"Copyright" also means copyright-like laws that apply to other kinds of

works, such as semiconductor masks.

"The Program" refers to any copyrightable work licensed under this

License. Each licensee is addressed as "you". "Licensees" and "recipients" may be individuals or organizations.

To "modify" a work means to copy from or adapt all or part of the work

in a fashion requiring copyright permission, other than the making of an exact copy. The resulting work is called a "modified version" of the earlier work or a work "based on" the earlier work.

A "covered work" means either the unmodified Program or a work based

on the Program.

To "propagate" a work means to do anything with it that, without

permission, would make you directly or secondarily liable for infringement under applicable copyright law, except executing it on a computer or modifying a private copy. Propagation includes copying, distribution (with or without modification), making available to the public, and in some countries other activities as well.

To "convey" a work means any kind of propagation that enables other

parties to make or receive copies. Mere interaction with a user through a computer network, with no transfer of a copy, is not conveying.

An interactive user interface displays "Appropriate Legal Notices"

to the extent that it includes a convenient and prominently visible feature that (1) displays an appropriate copyright notice, and (2) tells the user that there is no warranty for the work (except to the extent that warranties are provided), that licensees may convey the work under this License, and how to view a copy of this License. If the interface presents a list of user commands or options, such as a menu, a prominent item in the list meets this criterion.

1. Source Code.

The "source code" for a work means the preferred form of the work

for making modifications to it. "Object code" means any non-source form of a work.

A "Standard Interface" means an interface that either is an official

standard defined by a recognized standards body, or, in the case of interfaces specified for a particular programming language, one that is widely used among developers working in that language.

The "System Libraries" of an executable work include anything, other

than the work as a whole, that (a) is included in the normal form of packaging a Major Component, but which is not part of that Major Component, and (b) serves only to enable use of the work with that Major Component, or to implement a Standard Interface for which an implementation is available to the public in source code form. A "Major Component", in this context, means a major essential component (kernel, window system, and so on) of the specific operating system (if any) on which the executable work runs, or a compiler used to produce the work, or an object code interpreter used to run it.

The "Corresponding Source" for a work in object code form means all

the source code needed to generate, install, and (for an executable work) run the object code and to modify the work, including scripts to control those activities. However, it does not include the work's System Libraries, or general-purpose tools or generally available free programs which are used unmodified in performing those activities but which are not part of the work. For example, Corresponding Source includes interface definition files associated with source files for the work, and the source code for shared libraries and dynamically linked subprograms that the work is specifically designed to require, such as by intimate data communication or control flow between those subprograms and other parts of the work.

The Corresponding Source need not include anything that users

can regenerate automatically from other parts of the Corresponding Source.

The Corresponding Source for a work in source code form is that

same work.

2. Basic Permissions.

All rights granted under this License are granted for the term of

copyright on the Program, and are irrevocable provided the stated conditions are met. This License explicitly affirms your unlimited permission to run the unmodified Program. The output from running a covered work is covered by this License only if the output, given its content, constitutes a covered work. This License acknowledges your rights of fair use or other equivalent, as provided by copyright law.

You may make, run and propagate covered works that you do not

convey, without conditions so long as your license otherwise remains in force. You may convey covered works to others for the sole purpose of having them make modifications exclusively for you, or provide you with facilities for running those works, provided that you comply with the terms of this License in conveying all material for which you do not control copyright. Those thus making or running the covered works for you must do so exclusively on your behalf, under your direction and control, on terms that prohibit them from making any copies of your copyrighted material outside their relationship with you.

Conveying under any other circumstances is permitted solely under

the conditions stated below. Sublicensing is not allowed; section 10 makes it unnecessary.

3. Protecting Users' Legal Rights From Anti-Circumvention Law.

No covered work shall be deemed part of an effective technological

measure under any applicable law fulfilling obligations under article 11 of the WIPO copyright treaty adopted on 20 December 1996, or similar laws prohibiting or restricting circumvention of such measures.

When you convey a covered work, you waive any legal power to forbid

circumvention of technological measures to the extent such circumvention is effected by exercising rights under this License with respect to the covered work, and you disclaim any intention to limit operation or modification of the work as a means of enforcing, against the work's users, your or third parties' legal rights to forbid circumvention of technological measures.

4. Conveying Verbatim Copies.

You may convey verbatim copies of the Program's source code as you

receive it, in any medium, provided that you conspicuously and appropriately publish on each copy an appropriate copyright notice; keep intact all notices stating that this License and any non-permissive terms added in accord with section 7 apply to the code; keep intact all notices of the absence of any warranty; and give all recipients a copy of this License along with the Program.

You may charge any price or no price for each copy that you convey,

and you may offer support or warranty protection for a fee.

5. Conveying Modified Source Versions.

You may convey a work based on the Program, or the modifications to

produce it from the Program, in the form of source code under the terms of section 4, provided that you also meet all of these conditions:

> a) The work must carry prominent notices stating that you modified it, and giving a relevant date.

b) The work must carry prominent notices stating that it is released under this License and any conditions added under section 7. This requirement modifies the requirement in section 4 to "keep intact all notices".

c) You must license the entire work, as a whole, under this License to anyone who comes into possession of a copy. This License will therefore apply, along with any applicable section 7 additional terms, to the whole of the work, and all its parts, regardless of how they are packaged. This License gives no permission to license the work in any other way, but it does not invalidate such permission if you have separately received it.

d) If the work has interactive user interfaces, each must display Appropriate Legal Notices; however, if the Program has interactive interfaces that do not display Appropriate Legal Notices, your work need not make them do so.

A compilation of a covered work with other separate and independent

works, which are not by their nature extensions of the covered work, and which are not combined with it such as to form a larger program, in or on a volume of a storage or distribution medium, is called an "aggregate" if the compilation and its resulting copyright are not used to limit the access or legal rights of the compilation's users beyond what the individual works permit. Inclusion of a covered work in an aggregate does not cause this License to apply to the other parts of the aggregate.

6. Conveying Non-Source Forms.

You may convey a covered work in object code form under the terms

of sections 4 and 5, provided that you also convey the machine-readable Corresponding Source under the terms of this License, in one of these ways:

a) Convey the object code in, or embodied in, a physical product (including a physical distribution medium), accompanied by the Corresponding Source fixed on a durable physical medium customarily used for software interchange.

b) Convey the object code in, or embodied in, a physical product (including a physical distribution medium), accompanied by a written offer, valid for at least three years and valid for as long as you offer spare parts or customer support for that product model, to give anyone who possesses the object code either (1) a copy of the Corresponding Source for all the software in the product that is covered by this License, on a durable physical medium customarily used for software interchange, for a price no more than your reasonable cost of physically performing this conveying of source, or (2) access to copy the Corresponding Source from a network server at no charge.

c) Convey individual copies of the object code with a copy of the written offer to provide the Corresponding Source. This alternative is allowed only occasionally and noncommercially, and only if you received the object code with such an offer, in accord with subsection 6b.

d) Convey the object code by offering access from a designated place (gratis or for a charge), and offer equivalent access to the Corresponding Source in the same way through the same place at no further charge. You need not require recipients to copy the Corresponding Source along with the object code. If the place to copy the object code is a network server, the Corresponding Source may be on a different server (operated by you or a third party) that supports equivalent copying facilities, provided you maintain clear directions next to the object code saying where to find the Corresponding Source. Regardless of what server hosts the Corresponding Source, you remain obligated to ensure that it is available for as long as needed to satisfy these requirements.

e) Convey the object code using peer-to-peer transmission, provided you inform other peers where the object code and Corresponding Source of the work are being offered to the general public at no charge under subsection 6d.

A separable portion of the object code, whose source code is excluded

from the Corresponding Source as a System Library, need not be included in conveying the object code work.

A "User Product" is either (1) a "consumer product", which means any

tangible personal property which is normally used for personal, family, or household purposes, or (2) anything designed or sold for incorporation into a dwelling. In determining whether a product is a consumer product, doubtful cases shall be resolved in favor of coverage. For a particular product received by a particular user, "normally used" refers to a typical or common use of that class of product, regardless of the status of the particular user or of the way in which the particular user actually uses, or expects or is expected to use, the product. A product is a consumer product regardless of whether the product has substantial commercial, industrial or non-consumer uses, unless such uses represent the only significant mode of use of the product.

"Installation Information" for a User Product means any methods,

procedures, authorization keys, or other information required to install and execute modified versions of a covered work in that User Product from a modified version of its Corresponding Source. The information must suffice to ensure that the continued functioning of the modified object code is in no case prevented or interfered with solely because modification has been made.

If you convey an object code work under this section in, or with, or

specifically for use in, a User Product, and the conveying occurs as part of a transaction in which the right of possession and use of the User Product is transferred to the recipient in perpetuity or for a fixed term (regardless of how the transaction is characterized), the Corresponding Source conveyed under this section must be accompanied by the Installation Information. But this requirement does not apply if neither you nor any third party retains the ability to install modified object code on the User Product (for example, the work has been installed in ROM).

The requirement to provide Installation Information does not include a

requirement to continue to provide support service, warranty, or updates for a work that has been modified or installed by the recipient, or for the User Product in which it has been modified or installed. Access to a network may be denied when the modification itself materially and adversely affects the operation of the network or violates the rules and protocols for communication across the network.

Corresponding Source conveyed, and Installation Information provided,

in accord with this section must be in a format that is publicly documented (and with an implementation available to the public in source code form), and must require no special password or key for unpacking, reading or copying.

7. Additional Terms.

"Additional permissions" are terms that supplement the terms of this

License by making exceptions from one or more of its conditions. Additional permissions that are applicable to the entire Program shall be treated as though they were included in this License, to the extent that they are valid under applicable law. If additional permissions apply only to part of the Program, that part may be used separately under those permissions, but the entire Program remains governed by this License without regard to the additional permissions.

When you convey a copy of a covered work, you may at your option

remove any additional permissions from that copy, or from any part of it. (Additional permissions may be written to require their own removal in certain cases when you modify the work.) You may place additional permissions on material, added by you to a covered work, for which you have or can give appropriate copyright permission.

Notwithstanding any other provision of this License, for material you

add to a covered work, you may (if authorized by the copyright holders of that material) supplement the terms of this License with terms:

> a) Disclaiming warranty or limiting liability differently from the terms of sections 15 and 16 of this License; or

b) Requiring preservation of specified reasonable legal notices or author attributions in that material or in the Appropriate Legal Notices displayed by works containing it; or

c) Prohibiting misrepresentation of the origin of that material, or requiring that modified versions of such material be marked in reasonable ways as different from the original version; or

d) Limiting the use for publicity purposes of names of licensors or authors of the material; or

e) Declining to grant rights under trademark law for use of some trade names, trademarks, or service marks; or

f) Requiring indemnification of licensors and authors of that material by anyone who conveys the material (or modified versions of it) with contractual assumptions of liability to the recipient, for any liability that these contractual assumptions directly impose on those licensors and authors.

All other non-permissive additional terms are considered "further

restrictions" within the meaning of section 10. If the Program as you received it, or any part of it, contains a notice stating that it is governed by this License along with a term that is a further restriction, you may remove that term. If a license document contains a further restriction but permits relicensing or conveying under this License, you may add to a covered work material governed by the terms of that license document, provided that the further restriction does not survive such relicensing or conveying.

If you add terms to a covered work in accord with this section, you

must place, in the relevant source files, a statement of the additional terms that apply to those files, or a notice indicating where to find the applicable terms.

Additional terms, permissive or non-permissive, may be stated in the

form of a separately written license, or stated as exceptions; the above requirements apply either way.

8. Termination.

You may not propagate or modify a covered work except as expressly

provided under this License. Any attempt otherwise to propagate or modify it is void, and will automatically terminate your rights under this License (including any patent licenses granted under the third paragraph of section 11).

However, if you cease all violation of this License, then your

license from a particular copyright holder is reinstated (a) provisionally, unless and until the copyright holder explicitly and finally terminates your license, and (b) permanently, if the copyright holder fails to notify you of the violation by some reasonable means prior to 60 days after the cessation.

Moreover, your license from a particular copyright holder is

reinstated permanently if the copyright holder notifies you of the violation by some reasonable means, this is the first time you have received notice of violation of this License (for any work) from that copyright holder, and you cure the violation prior to 30 days after your receipt of the notice.

Termination of your rights under this section does not terminate the

licenses of parties who have received copies or rights from you under this License. If your rights have been terminated and not permanently reinstated, you do not qualify to receive new licenses for the same material under section 10.

9. Acceptance Not Required for Having Copies.

You are not required to accept this License in order to receive or

run a copy of the Program. Ancillary propagation of a covered work occurring solely as a consequence of using peer-to-peer transmission to receive a copy likewise does not require acceptance. However, nothing other than this License grants you permission to propagate or modify any covered work. These actions infringe copyright if you do not accept

this License. Therefore, by modifying or propagating a covered work, you indicate your acceptance of this License to do so.

10. Automatic Licensing of Downstream Recipients.

Each time you convey a covered work, the recipient automatically

receives a license from the original licensors, to run, modify and propagate that work, subject to this License. You are not responsible for enforcing compliance by third parties with this License.

An "entity transaction" is a transaction transferring control of an

organization, or substantially all assets of one, or subdividing an organization, or merging organizations. If propagation of a covered work results from an entity transaction, each party to that transaction who receives a copy of the work also receives whatever licenses to the work the party's predecessor in interest had or could give under the previous paragraph, plus a right to possession of the Corresponding Source of the work from the predecessor in interest, if the predecessor has it or can get it with reasonable efforts.

You may not impose any further restrictions on the exercise of the

rights granted or affirmed under this License. For example, you may not impose a license fee, royalty, or other charge for exercise of rights granted under this License, and you may not initiate litigation (including a cross-claim or counterclaim in a lawsuit) alleging that any patent claim is infringed by making, using, selling, offering for sale, or importing the Program or any portion of it.

11. Patents.

A "contributor" is a copyright holder who authorizes use under this

License of the Program or a work on which the Program is based. The work thus licensed is called the contributor's "contributor version".

A contributor's "essential patent claims" are all patent claims

owned or controlled by the contributor, whether already acquired or hereafter acquired, that would be infringed by some manner, permitted by this License, of making, using, or selling its contributor version, but do not include claims that would be infringed only as a consequence of further modification of the contributor version. For purposes of this definition, "control" includes the right to grant patent sublicenses in a manner consistent with the requirements of this License.

Each contributor grants you a non-exclusive, worldwide, royalty-free

patent license under the contributor's essential patent claims, to make, use, sell, offer for sale, import and otherwise run, modify and propagate the contents of its contributor version.

In the following three paragraphs, a "patent license" is any express

agreement or commitment, however denominated, not to enforce a patent (such as an express permission to practice a patent or covenant not to sue for patent infringement). To "grant" such a patent license to a party means to make such an agreement or commitment not to enforce a patent against the party.

If you convey a covered work, knowingly relying on a patent license,

and the Corresponding Source of the work is not available for anyone to copy, free of charge and under the terms of this License, through a publicly available network server or other readily accessible means, then you must either (1) cause the Corresponding Source to be so available, or (2) arrange to deprive yourself of the benefit of the patent license for this particular work, or (3) arrange, in a manner consistent with the requirements of this License, to extend the patent license to downstream recipients. "Knowingly relying" means you have actual knowledge that, but for the patent license, your conveying the covered work in a country, or your recipient's use of the covered work in a country, would infringe one or more identifiable patents in that country that you have reason to believe are valid.

If, pursuant to or in connection with a single transaction or

arrangement, you convey, or propagate by procuring conveyance of, a covered work, and grant a patent license to some of the parties receiving the covered work authorizing them to use, propagate, modify or convey a specific copy of the covered work, then the patent license you grant is automatically extended to all recipients of the covered work and works based on it.

A patent license is "discriminatory" if it does not include within

the scope of its coverage, prohibits the exercise of, or is conditioned on the non-exercise of one or more of the rights that are specifically granted under this License. You may not convey a covered work if you are a party to an arrangement with a third party that is in the business of distributing software, under which you make payment to the third party based on the extent of your activity of conveying the work, and under which the third party grants, to any of the parties who would receive the covered work from you, a discriminatory patent license (a) in connection with copies of the covered work conveyed by you (or copies made from those copies), or (b) primarily for and in connection with specific products or compilations that contain the covered work, unless you entered into that arrangement, or that patent license was granted, prior to 28 March 2007.

Nothing in this License shall be construed as excluding or limiting

any implied license or other defenses to infringement that may otherwise be available to you under applicable patent law.

12. No Surrender of Others' Freedom.

If conditions are imposed on you (whether by court order, agreement or

otherwise) that contradict the conditions of this License, they do not excuse you from the conditions of this License. If you cannot convey a covered work so as to satisfy simultaneously your obligations under this License and any other pertinent obligations, then as a consequence you may not convey it at all. For example, if you agree to terms that obligate you to collect a royalty for further conveying from those to whom you convey the Program, the only way you could satisfy both those terms and this License would be to refrain entirely from conveying the Program.

13. Remote Network Interaction; Use with the GNU General Public License.

Notwithstanding any other provision of this License, if you modify the

Program, your modified version must prominently offer all users interacting with it remotely through a computer network (if your version supports such interaction) an opportunity to receive the Corresponding Source of your version by providing access to the Corresponding Source from a network server at no charge, through some standard or customary means of facilitating copying of software. This Corresponding Source shall include the Corresponding Source for any work covered by version 3 of the GNU General Public License that is incorporated pursuant to the following paragraph.

Notwithstanding any other provision of this License, you have

permission to link or combine any covered work with a work licensed under version 3 of the GNU General Public License into a single combined work, and to convey the resulting work. The terms of this License will continue to apply to the part which is the covered work, but the work with which it is combined will remain governed by version 3 of the GNU General Public License.

14. Revised Versions of this License.

The Free Software Foundation may publish revised and/or new versions of

the GNU Affero General Public License from time to time. Such new versions will be similar in spirit to the present version, but may differ in detail to address new problems or concerns.

Each version is given a distinguishing version number. If the

Program specifies that a certain numbered version of the GNU Affero General Public License "or any later version" applies to it, you have the option of following the terms and conditions either of that numbered version or of any later version published by the Free Software Foundation. If the Program does not specify a version number of the GNU Affero General Public License, you may choose any version ever published by the Free Software Foundation.

If the Program specifies that a proxy can decide which future

versions of the GNU Affero General Public License can be used, that proxy's public statement of acceptance of a version permanently authorizes you to choose that version for the Program.

Later license versions may give you additional or different

permissions. However, no additional obligations are imposed on any author or copyright holder as a result of your choosing to follow a later version.

15. Disclaimer of Warranty.

THERE IS NO WARRANTY FOR THE PROGRAM, TO THE EXTENT PERMITTED BY

APPLICABLE LAW. EXCEPT WHEN OTHERWISE STATED IN WRITING THE COPYRIGHT HOLDERS AND/OR OTHER PARTIES PROVIDE THE PROGRAM "AS IS" WITHOUT WARRANTY OF ANY KIND, EITHER EXPRESSED OR IMPLIED, INCLUDING, BUT NOT LIMITED TO, THE IMPLIED WARRANTIES OF MERCHANTABILITY AND FITNESS FOR A PARTICULAR PURPOSE. THE ENTIRE RISK AS TO THE QUALITY AND PERFORMANCE OF THE PROGRAM IS WITH YOU. SHOULD THE PROGRAM PROVE DEFECTIVE, YOU ASSUME THE COST OF ALL NECESSARY SERVICING, REPAIR OR CORRECTION.

16. Limitation of Liability.

IN NO EVENT UNLESS REQUIRED BY APPLICABLE LAW OR AGREED TO IN WRITING

WILL ANY COPYRIGHT HOLDER, OR ANY OTHER PARTY WHO MODIFIES AND/OR CONVEYS THE PROGRAM AS PERMITTED ABOVE, BE LIABLE TO YOU FOR DAMAGES, INCLUDING ANY GENERAL, SPECIAL, INCIDENTAL OR CONSEQUENTIAL DAMAGES ARISING OUT OF THE USE OR INABILITY TO USE THE PROGRAM (INCLUDING BUT NOT LIMITED TO LOSS OF DATA OR DATA BEING RENDERED INACCURATE OR LOSSES SUSTAINED BY YOU OR THIRD PARTIES OR A FAILURE OF THE PROGRAM TO OPERATE WITH ANY OTHER PROGRAMS), EVEN IF SUCH HOLDER OR OTHER PARTY HAS BEEN ADVISED OF THE POSSIBILITY OF SUCH DAMAGES.

17. Interpretation of Sections 15 and 16.

If the disclaimer of warranty and limitation of liability provided

above cannot be given local legal effect according to their terms, reviewing courts shall apply local law that most closely approximates an absolute waiver of all civil liability in connection with the Program, unless a warranty or assumption of liability accompanies a copy of the Program in return for a fee.

END OF TERMS AND CONDITIONS

How to Apply These Terms to Your New Programs

If you develop a new program, and you want it to be of the greatest

possible use to the public, the best way to achieve this is to make it free software which everyone can redistribute and change under these terms.

To do so, attach the following notices to the program. It is safest

to attach them to the start of each source file to most effectively state the exclusion of warranty; and each file should have at least the "copyright" line and a pointer to where the full notice is found.

<one line to give the program's name and a brief idea of what it does.> Copyright (C) <year> <name of author>

This program is free software: you can redistribute it and/or modify it under the terms of the GNU Affero General Public License as published by the Free Software Foundation, either version 3 of the License, or (at your option) any later version.

This program is distributed in the hope that it will be useful, but WITHOUT ANY WARRANTY; without even the implied warranty of MERCHANTABILITY or FITNESS FOR A PARTICULAR PURPOSE. See the GNU Affero General Public License for more details.

You should have received a copy of the GNU Affero General Public License along with this program. If not, see <https://www.gnu.org/licenses/>.

Also add information on how to contact you by electronic and paper mail.

If your software can interact with users remotely through a computer

network, you should also make sure that it provides a way for users to get its source. For example, if your program is a web application, its interface could display a "Source" link that leads users to an archive of the code. There are many ways you could offer source, and different solutions will be better for different programs; see section 13 for the specific requirements.

You should also get your employer (if you work as a programmer) or school,

if any, to sign a "copyright disclaimer" for the program, if necessary. For more information on this, and how to apply and follow the GNU AGPL, see <https://www.gnu.org/licenses/>.

2.8 Acknowledgements

This project would not have been possible without the people developing, maintaining and releasing open source projects online. Therefore, I would like to thank the people that developed the packages this project directly depends on, those that developed the packages those projects depend on, and so on:

- `pandas` : Powerful data structures for data analysis, time series, and statistics
- `numpy` – Travis E. Oliphant et al.: Fundamental package for array computing in Python
- `torch` – PyTorch Team: Tensors and Dynamic neural networks in Python with strong GPU acceleration
- `torchvision` – PyTorch Core Team: image and video datasets and models for torch deep learning
- `py` – holger krekel, Ronny Pfannschmidt, Benjamin Peterson and others: library with cross-python path, ini-parsing, io, code, log facilities
- `matplotlib` – John D. Hunter, Michael Droettboom: Python plotting package
- `pytorch-ignite` – PyTorch-Ignite Team: A lightweight library to help with training neural networks in PyTorch.
- `tqdm` : Fast, Extensible Progress Meter
- `sympy` – SymPy development team: Computer algebra system (CAS) in Python
- `scikit-learn` : A set of python modules for machine learning and data mining
- `seaborn` : Statistical data visualization
- `joblib` : Lightweight pipelining with Python functions
- `tensorboard` – Google Inc.: TensorBoard lets you watch Tensors Flow
- `ConfigArgParse` : A drop-in replacement for argparse that allows options to also be set via config files and/or environment variables.
- `torch-optimizer` – Nikolay Novik: pytorch-optimizer
- `adabelief-pytorch` – Juntang Zhuang: PyTorch implementation of AdaBelief Optimizer
- `dill` – Mike McKerns: serialize all of Python
- `aislib` – Arnor Sigurdsson:
- `colorama` : Cross-platform colored terminal text.
- `torchtext` – PyTorch Text Team: Text utilities, models, transforms, and datasets for PyTorch.

- `transformers` – The Hugging Face team (past and future) with the help of all our contributors (https://github.com/huggingface/transformers/graphs/contributors): State-of-the-art Machine Learning for JAX, PyTorch and TensorFlow
- `sentencepiece` – Taku Kudo: SentencePiece python wrapper
- `ipython` – The IPython Development Team: IPython: Productive Interactive Computing
- `timm` : PyTorch Image Models
- `captum` – PyTorch Team: Model interpretability for PyTorch
- `deeplake` – activeloop.ai: Activeloop Deep Lake
- `aioboto3` – Terry Cain: Async boto3 wrapper
- `termcolor` : ANSI color formatting for output in terminal
- `tokenizers` – Anthony MOI <m.anthony.moi@gmail.com>:
- `pyarrow` : Python library for Apache Arrow
- `einops` – Alex Rogozhnikov: A new flavour of deep learning operations
- `umap-learn` : Uniform Manifold Approximation and Projection
- `fastapi` : FastAPI framework, high performance, easy to learn, fast to code, ready for production
- `uvicorn` : The lightning-fast ASGI server.
- `pydantic` : Data validation using Python type hints
- `tiktoken` – Shantanu Jain: tiktoken is a fast BPE tokeniser for use with OpenAI's models
- `memory-profiler` – Fabian Pedregosa: A module for monitoring memory usage of a python program
- `pytest` – Holger Krekel, Bruno Oliveira, Ronny Pfannschmidt, Floris Bruynooghe, Brianna Laugher, Florian Bruhin, Others (See AUTHORS): pytest: simple powerful testing with Python
- `tox` : tox is a generic virtualenv management and test command line tool
- `flake8` – Tarek Ziade: the modular source code checker: pep8 pyflakes and co
- `jupyter` – Jupyter Development Team: Jupyter metapackage. Install all the Jupyter components in one go.
- `ipykernel` : IPython Kernel for Jupyter
- `coverage` – Ned Batchelder and 225 others: Code coverage measurement for Python
- `snakeviz` : A web-based viewer for Python profiler output
- `pytest-cov` – Marc Schlaich: Pytest plugin for measuring coverage.
- `pynvim` – Neovim Authors: Python client for Neovim
- `pre-commit` – Anthony Sottile: A framework for managing and maintaining multi-language pre-commit hooks.
- `gpustat` – Jongwook Choi: An utility to monitor NVIDIA GPU status and usage
- `black` : The uncompromising code formatter.
- `Sphinx` : Python documentation generator
- `sphinx-rtd-theme` – Dave Snider, Read the Docs, Inc. & contributors: Read the Docs theme for Sphinx
- `sphinx-copybutton` – Executable Book Project: Add a copy button to each of your code cells.
- `tomlkit` – Sébastien Eustace: Style preserving TOML library
- `gdown` : Google Drive Public File/Folder Downloader

- `hypothesis` – David R. MacIver and Zac Hatfield-Dodds: A library for property-based testing
- `pdf2image` – Edouard Belval: A wrapper around the pdftoppm and pdftocairo command line tools to convert PDF to a PIL Image list.
- `vulture` – Jendrik Seipp: Find dead code
- `mypy` – Jukka Lehtosalo: Optional static typing for Python
- `types-pyyaml` : Typing stubs for PyYAML
- `isort` – Timothy Crosley: A Python utility / library to sort Python imports.
- `pytest-split` – Jerry Pussinen: Pytest plugin which splits the test suite to equally sized sub suites based on test execution time.

INDEX

L

M

N

O

P

R

S

T

U

V

X