

HOMWORK 2

IMPLEMENT YOUR OWN BERT¹

CS 678 ADVANCED NATURAL LANGUAGE PROCESSING (FALL 2022)

<https://nlp.cs.gmu.edu/course/cs678-fall22/>

OUT: September 16, 2022

DUE: September 30, 2022

Academic Honesty: Please note that students should complete the assignment independently. While you may discuss the assignment with other students, all work you submit must be your own! In addition, if you use any external resources for the assignment, you must include references to these resources in your assignment report.

Goal: In this assignment, you will implement some important components of the BERT model to gain a better understanding its architecture. The implementation will be based on PyTorch (<https://pytorch.org/>). Other generic Python tools such as numpy are also allowed (if you are not sure, please consult the instructor).

1 Assignment Material Overview

Datasets All data are available under the `data` directory of the assignment materials. We will use two datasets for sentence classification:

- **sst**: the Stanford Sentiment Treebank is a corpus with fully labeled parse trees that allows for a complete analysis of the compositional effects of sentiment in language. The corpus is based on the dataset introduced by Pang and Lee (2005) and consists of 11,855 single sentences extracted from movie reviews. There are 8544 train examples, 1101 dev examples, and 2210 test sentences. There are five possible output labels (`[0--4]`): highly negative, negative, neutral, positive, and highly positive.. All data are already tokenized.
- **cfimdb**: This is a dataset based on IMDB reviews (see details here: <https://openreview.net/pdf?id=Sk1gs0NFvr>). These are longer documents than the **sst** dataset, but the output labels are binary (`[0,1]`). There are 1707 training examples, 245 examples for dev, and 488 instances for test. All data are already tokenized. **The `cfimdb-test.txt` file does not provide the correct test set labels. The correct labels are *hidden*, and we will reveal them after the homework deadline. The top-3 submissions for each section, based on the hidden test et performance, will receive up to 2 extra credits.**

Code You are given skeleton code to build upon.

- Follow `setup.sh` to properly setup the environment and install dependencies. Make sure to do the rest of your work on the appropriate environment.
- There is a detailed description of the code structure in the `structure.md` as well as below, including a description of which parts you will need to implement.
- You are only allowed to use `torch`, no other external libraries are allowed (e.g., `huggingface transformers`).

¹Compiled on Friday 16th September, 2022 at 22:25

- We will run your code with the following commands, so make sure that whatever your best results are reproducible using these commands (where you replace GMUID with your ID):

```
1 mkdir -p GMUID
2
3 python3 classifier.py --option [pretrain/finetune] --epochs NUM_EPOCHS --lr LR --
  train data/sst-train.txt --dev data/sst-dev.txt --test data/sst-test.txt
```

Reference accuracies: Mean reference accuracies over 10 random seeds with their standard deviation shown in brackets.

- Pretraining for SST:
Dev Accuracy: 0.391 (0.007)
Test Accuracy: 0.403 (0.008)
- Finetuning for SST:
Dev Accuracy: 0.515 (0.004)
Test Accuracy: 0.526 (0.008)
- Finetuning for CFIMDB:
Dev Accuracy: 0.966 (0.007)
Test Accuracy: *hidden*

Submission: You will upload your code and your report to Blackboard.

In summary, your report should include (1) the final dev and test set performance of your finetuned classifier on the `sst` dataset; (2) the dev set performance on the `cfimdb` dataset; (3) a minimal analysis of the performance on the dev sets, and a description of anything additional you did (if you try anything on top of the requirements of this homework). The grading will also be based on whether the grader could execute your code and successfully obtain expected model outputs within a reasonable time limit. Accuracy of your model on dev set and the blind test set for `cfimdb` will also be checked.

We ask that you include the predictions of your classifier on the dev and test sets of both datasets, so that we can compare accuracies.

Note that partial credit is awarded even for non-functional solutions. If you have even partially implemented functions, please submit them and we will score your submission appropriately.

The submission file should be a zip file with the following structure (assuming your GMU id is GMUID):

```
1 GMUID/
2   base_bert.py
3   bert.py
4   classifier.py
5   config.py
6   optimizer.py
7   sanity_check.py
8   tokenizer.py
9   utils.py
10  README.md
11  structure.md
12  sanity_check.data
13  setup.py
14  sst-dev-output.txt
```

```

15 sst-test-output.txt
16 cfimdb-dev-output.txt
17 cfimdb-test-output.txt

```

`prepare_submit.py` can help to create(1) or check(2) the to-be-submitted zip file. It will throw assertion errors if the format is not expected, and we will **not accept submissions that fail this check**. Usage: (1) To create and check a zip file with your outputs, run `python3 prepare_submit.py path/to/your/output/dir GMUID`, (2) To check your zip file, run `python3 prepare_submit.py path/to/your/submit/zip/file.zip GMUID`.

Grading

- A+ (10+): You additionally implement something else on top of the requirements for A, and achieve significant accuracy improvements. Please write down the things you implemented and experiments you performed in the report. You are also welcome to provide additional materials such as commands to run your code in a script and training logs.
 - perform continued pre-training (<https://arxiv.org/abs/2004.10964>) using the MLM objective to do domain adaptation
 - try alternative fine-tuning algorithms (<https://www.aclweb.org/anthology/2020.acl-main.197>)
 - add other model components on top of the model.
- A (10): You implement all the missing pieces and the original `classifier.py` with `--option pretrain` and `--option finetune` code that achieves comparable accuracy to our reference implementation
- A (8-9): You implement all the missing pieces and the original `classifier.py` with `--option pretrain` and `--option finetune` code but accuracy is not comparable to the reference.
- B+ (6-7): All missing pieces are implemented and pass tests in `sanity_check.py`
- B (0-5): Some parts of the missing pieces are not implemented.

Additionally, there will be up to 2 extra credits awarded to the top-3 submissions from each section based on their blind `cfimdb` test set performance.

Notes on the Code

base_bert.py This is the base class for the `BertModel`. It contains functions to (1.) initialize the weights `init_weights`, `_init_weights` and (2.) restore pre-trained weights `from_pretrained`. Since we are using the weights from HuggingFace, we are doing a few mappings to match the parameter names. You won't need to modify this file in this assignment.

tokenizer.py This is where `BertTokenizer` is defined. You won't need to modify this file in this assignment.

config.py This is where the configuration class is defined. You won't need to modify this file in this assignment.

utils.py This file contains utility functions for various purpose. You won't need to modify this file in this assignment.

2 Implement your Own BERT [10 points]

Part 1: bert.py [7 points total]

This file contains the BERT Model whose backbone is the transformer (<https://arxiv.org/pdf/1706.03762.pdf>). We recommend walking through Section 3 of the paper to understand each component of the transformer.

2.1 To be implemented

Components that require your implementations are commented with `#todo`. The detailed instructions can be found in their corresponding code blocks

- `bert.BertSelfAttention.attention` [2 points]
- `bert.BertLayer.add_norm` [1 points]
- `bert.BertLayer.forward` [2 points]
- `bert.BertModel.embed` [2 points]

IMPORTANT NOTE: you are free to re-organize the functions inside each class, but please do not change the variable names that correspond to BERT parameters. Any change to these variable names will fail to load the pre-trained weights.

BertSelfAttention [2 points]

The multi-head attention layer of the transformer. This layer maps a query and a set of key-value pairs to an output. The output is calculated as the weighted sum of the values, where the weight of each value is computed by a function that takes the query and the corresponding key. To implement this layer, you have to:

1. linearly project the queries, keys, and values with their corresponding linear layers
2. split the vectors for multi-head attention
3. follow the equation to compute the attended output of each head
4. concatenate multi-head attention outputs to recover the original shape

$$\text{Attention}(Q, K, V) = \text{softmax}\left(\frac{QK^T}{\sqrt{d_k}}\right)V$$

BertLayer [3 points]

This corresponds to one transformer layer which has

1. a multi-head attention layer
2. add-norm layer
3. a feed-forward layer
4. another add-norm layer

BertModel [2 points]

This is the BertModel that takes the input ids and returns the contextualized representation for each word. The structure of the BertModel is:

1. an embedding layer that consists of word embeddings `word_embedding` and positional embeddings `pos_embedding`.
2. bert encoder which is a stack of `config.num_hidden_layers BertLayer`
3. a projection layer for the [CLS] token which is often used for classification tasks

The desired outputs are:

1. `last_hidden_state`: the contextualized embedding for each word of the sentence, taken from the last BertLayer (i.e. the output of the bert encoder)
2. `pooler_output`: the [CLS] token embedding

2.2 Sanity check

We provide a sanity check function at `sanity_check.py` to test your implementation. It will reload two embeddings we computed with our reference implementation and check whether your implementation outputs match ours.

Part 2: classifier.py [3 points total]

This file contains the pipeline to:

- call the BERT model to encode the sentences for their contextualized representations
- feed in the encoded representations for the sentence classification task
- fine-tune the Bert model on the downstream tasks (e.g. sentence classification).

As before, you will need to fill in the missing parts of the code marked with `#todo`.

2.3 BertSentClassifier (to be implemented) [2 points]

This class is used to:

- encode the sentences using BERT to obtain the pooled output representation of the sentence.
- classify the sentence by applying dropout to the pooled-output and project it using a linear layer.
- adjust the model paramters depending on whether we are pre-training or fine-tuning BERT

2.4 Output Predictions [1 points]

With everything implemented, you should be able to run your models over both datasets and obtain outputs both for the dev and the test sets. Include these outputs with your submission.