Homework 1
Implement your own BERT[1]

CS 678 Advanced Natural Language Processing (Spring 2023)
https://nlp.cs.gmu.edu/course/cs678-spring23/
Note: this assignment was created by Graham Neubig

OUT: January 23, 2023
DUE: February 16, 2023

Your name: Abhishek Jallawaram

Your GID: G01373042

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 (beyond the ones linked here) you must **properly include references to these resources** in your assignment report and collaboration questions.

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).

Graded Questions: 100 points
Bonus Points: 20 points
Total Points Available: 120/100

Additional Notes:

- For this assignment, you don't need to upload anything to Gradescope – just upload your submission to Blackboard.
- Your Blackboard submission should also **include the required files**, as described below.
- For text-based answers, you should replace the text that says "Write your answer here..." with your actual answer.

[1]Compiled on Friday 17th February, 2023 at 23:44

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=Sklgs0NFvr). 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 based on the hidden test set 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`).
- 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):

```
mkdir -p GMUID

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`):

```
GMUID/
    base_bert.py
    bert.py
    classifier.py
    config.py
    optimizer.py
    sanity_check.py
    tokenizer.py
    utils.py
    README.md
    structure.md
    sanity_check.data
    setup.py
    sst-dev-output.pretrain.txt
    sst-test-output.pretrain.txt
    cfimdb-dev-output.pretrain.txt
    cfimdb-test-output.pretrain.txt
    sst-dev-output.finetune.txt
    sst-test-output.finetune.txt
    cfimdb-dev-output.finetune.txt
    cfimdb-test-output.finetune.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`.

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.[2] We recommend walking through Section 3 of the paper to understand each component of the transformer. The actual BERT paper is different[3], since BERT is just the encoder component of the full transformer model. There exist a lot of blogposts and other online resources. We recommend "the Annotated Transformer"[4] which includes code along with the original transformer paper, as well as "the Illustrated BERT"[5]. We also recommend the paper "Transformers without Tears" for a deep dive into the the mechanics of training these models.[6] Last, this blogpost using Named Tensor Notation[7] should also help you if you're confused about implementation details.[8]

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

[2] https://arxiv.org/pdf/1706.03762.pdf
[3] https://arxiv.org/pdf/1810.04805.pdf
[4] https://nlp.seas.harvard.edu/2018/04/03/attention.html
[5] https://jalammar.github.io/illustrated-bert/
[6] https://arxiv.org/pdf/1910.05895.pdf
[7] https://arxiv.org/pdf/2102.13196.pdf
[8] https://hackmd.io/@mlelarge/HkVlvrc8j

3. follow the equation to compute the attended output of each head
4. concatenate multi-head attention outputs to recover the original shape

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

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.

2.3 Grading

We provide details on how the above section will be graded:

- We will run `sanity_check.py` on your code. If (the unmodified) sanity check passes, then very likely you have a correct implementation and will receive full marks.
- If your code does not pass the sanity check, we will look carefully into your implementation.
- Notes on the point distribution:
  - BERTSelftAttention: 1 point for correct implementation of the attention score, 1 point for correct projection to keys, values, queries, and for recovering the original shape after the attention
  - BERTLayer: 1 point for calling the self-attention layer correctly, 1 for calling the feed-forward layer correctly, 1 for correctly calling the two layer-norms correctly (.5 each)
  - BertModel: 1 for properly putting together the input$\rightarrow$model$\rightarrow$output, 1 for returning the [CLS] token and the rest of the representations

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`.

2.4 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

2.5 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.

First, use the pretrained model (without finetuning). Run your code with the following command (where you replace GMUID with your ID)[9]:

```
mkdir -p GMUID
python3 classifier.py --option pretrain --epochs NUM_EPOCHS --lr LR --train data/sst-
    train.txt --dev data/sst-dev.txt --test data/sst-test.txt
```

Report below the accuracies that you obtain:

SST Dev Accuracy	SST Test Accuracy	CFIMDB Dev Accuracy
38.5 %	39.9 %	70.2 %

CFIMDB Test Accuracy
45.3 %

Don't forget to include the outputs for the CFIMDB test set output too!

Then, use finetune the model and try to optimize the hyperparameters. Run your code with the following command[10]:

```
python3 classifier.py --option finetune --epochs NUM_EPOCHS --lr LR --train data/sst-
    train.txt --dev data/sst-dev.txt --test data/sst-test.txt
```

Report below the accuracies that you obtain:

SST Dev FT Accuracy	SST Test FT Accuracy	CFIMDB Dev FT Accuracy
51.4 %	52.4 %	96.7 %

[9] and similarly for the other dataset
[10] and similarly for the other dataset

CFIMDB Test FT Accuracy

51.0 %

Don't forget to include the outputs for the CFIMDB test set output too!

Observation

Adding additional Linear layers for classification improved the pre-train phase of the model and provide a marginal increase in accuracy for both datasets.