CS 224N (Spring 2024) Default Final Project: minBERT and Downstream Tasks

Contents

Note: This project was adapted from the "minbert" assignment developed for Carnegie Mellon University's CS11-711 Advanced NLP class by Shuyan Zhou, Zhengbao Jiang, Ritam Dutt, Brendon Boldt, Aditya Veerubhotla, and Graham Neubig.

1 Overview

The default final project has two parts. In the first part, you will fill in missing code blocks to complete an implementation of the BERT model. Using pre-trained weights loaded into your BERT model, you will perform sentiment analysis on the SST dataset and the CFIMDB dataset.

In the second part, you will explore extensions of your BERT model to achieve the highest performance that you can on multiple sentence-level tasks: **sentiment analysis**, **paraphrase detection**, and **semantic textual similarity**. The goal of this section is for you to engineer and experiment with improvements to your BERT model to obtain robust and generalizable sentence embeddings that perform well in more than one setting.

Note on default project vs. custom project: It is not intended for the default final project to require less effort or work than the custom final project. The default project simply excludes the difficulty of devising your own project idea and evaluation methods, allowing students to commit an equivalent amount of effort to the provided problem.

1.1 Bidirectional Encoder Representations from Transformers: BERT

Bidirectional Encoder Representations from Transformers, or "BERT," is a transformer-based model that generates contextual word representations [1]. With its backbone being the transformer and by making use of bidirectional word representations, BERT took a large leap forward when it was released in 2018 for contextual word embeddings/large language models/foundational models.

1.2 Sentiment Analysis

A basic task in understanding a given text is classifying its polarity, i.e., whether the expressed opinion in a text is positive, negative, or neutral. Sentiment analysis can be utilized to determine individual feelings towards particular products, politicians, or within news reports.

As a concrete example of a sentiment analysis dataset, the Stanford Sentiment Treebank[1] [2] consists of 11,855 single sentences extracted from movie reviews. The dataset was parsed with the Stanford parser[2] and includes a total of 215,154 unique phrases, annotated by 3 human judges. Each phrase has a label of negative, somewhat negative, neutral, somewhat positive, or positive. Below are three examples extracted from the SST dataset:

Movie Review: Light, silly, photographed with colour and depth, and rather a good time.
Sentiment: 4 (Positive)

Movie Review: Opening with some contrived banter, cliches and some loose ends, the screenplay only comes into its own in the second half.
Sentiment: 2 (Neutral)

Movie Review: ... a sour little movie at its core; an exploration of the emptiness that underlay the relentless gaiety of the 1920's ... The film's ending has a "What was it all for?"
Sentiment: 0 (Negative)

[1] https://nlp.stanford.edu/sentiment/treebank.html
[2] https://nlp.stanford.edu/software/lex-parser.shtml

1.3 Paraphrase Detection

Paraphrase detection is the task of finding paraphrases of texts in a large corpus of passages. A paraphrase is a "restatement (or reuse) of text giving the meaning in another form" [3]. Therefore, paraphrase detection seeks to determine whether particular words or phrases convey the same semantic meaning. From a research perspective, paraphrase detection is an interesting task because it provides a measure of how well systems can 'understand' fine-grained notions of semantic meaning.

The website Quora[3] often receives questions that are duplicates of other questions. To better redirect users and prevent unnecessary work, Quora released a dataset that labels whether different questions are paraphrases of each other. Below are two examples extracted from the Quora dataset:

Question Pair: (1) "What is the step by step guide to invest in share market in india?", (2) "What is the step by step guide to invest in share market?"
Is Paraphrase: No

Question Pair: (1) "I am a Capricorn Sun Cap moon and cap rising...what does that say about me?", (2) "I'm a triple Capricorn (Sun, Moon and ascendant in Capricorn) What does this say about me?
Is Paraphrase: Yes

1.4 Semantic Textual Similarity (STS)

The semantic textual similarity (STS) task seeks to capture the notion that some texts are more similar than others; STS seeks to measure the degree of semantic equivalence [4]. STS differs from paraphrasing in that it is not a yes or no decision. Rather, STS allows for degrees of similarity. For example, on a scale from 0 (not at all related) to 5 (same meaning), the following sentences are rated according to similarity:[4]

(5) The sentences are completely equivalent, as they mean the same thing:
The bird is bathing in the sink.
Birdie is washing itself in the water basin

(4) The two sentences are mostly equivalent but some unimportant details differ:
In May 2010, the troops attempted to invade Kabul.
The US army invaded Kabul on May 7th last year, 2010.

(3) The two sentences are roughly equivalent, but some important information differs:
John said he is considered a witness but not a suspect
"He is not a suspect anymore."

(2) The two sentences are not equivalent, but do share some details:
They flew out of the nest in groups.
They flew into the nest together.

(1) The two sentences are not equivalent, but are on the same topic:
The woman is playing the violin.
The young lady enjoys listening to the guitar.

[3] https://www.quora.com/q/quoradata/First-Quora-Dataset-Release-Question-Pairs
[4] These sentences and labels come from https://aclanthology.org/S13-1004.pdf [4]

(0) The two sentences are on different topics:
John went horseback riding at dawn with a whole group of friends.
Sunrise at dawn is a magnificent view to take in if you wake up early enough for it.

1.5 This Project

The goal of this project is for you to implement some of the key aspects of the original BERT model and explore extensions to improve upon your baseline. The first part focuses on implementing the multi-head self-attention and transformer layers of the original BERT model. You will then utilize your completed BERT model to perform sentiment analysis on the Stanford Sentiment Treebank dataset as well as another dataset of movie reviews. In the second half of this project, you will extend your BERT model to improve its performance across a wide range of downstream tasks and enter your model's predictions in a class-wide competition. In Section 5, we describe several techniques that are commonly used to create more robust and semantically-rich sentence embeddings in BERT models—most come from recent research papers. We provide these suggestions to help you get started.

Though you're not required to implement something original, the best projects will pursue some form of originality and, in fact, may become research papers in the future. Originality does not necessarily mean a completely new approach. Small but well-motivated changes to existing models are valuable, especially if followed by thorough and logical analysis. Showing quantitatively and qualitatively that your small but original changes improve a state-of-the-art model (and even better, explain what particular problem it solves and how) will mean that you have done extremely well.

Like the custom final project, the second part of the default final project is open-ended. It will be up to you to explore the possibilities. We are expecting you to exercise the judgment and intuition that you've gained from the class so far to build your highest performing model. For more information on grading criteria, see Section 7.

Note that this document only describes the coding portion of the Default Final Project. For more details on the write-up, see the course website and the handout *CS224N: Project Report Instructions*.

2 Getting Started

For this project, you will need a machine with GPUs to train your models efficiently. For this, you have access to Google Cloud, similar to Assignments 4 and 5.

We advise that you **develop your code on your local machine** (or one of the Stanford machines, like `rice`), using PyTorch without GPUs, and move to your Google Cloud VM only after you've debugged your code and are ready to train. We advise that you use a private GitHub repository to manage your codebase and sync files between the two machines and between team members. When you work through this *Getting Started* section for the first time, do so on your local machine. You will then repeat the process on your Google Cloud VM.

Once you are on an appropriate machine, clone the project GitHub repository with the following command:

```
https://github.com/amahankali10/CS224N-Spring2024-DFP-Student-Handout
```

This repository contains the starter code and a minimalist implementation of the BERT model (minBERT) that we will be using. We encourage you to `git clone` our repository, rather than simply downloading it, so you can easily integrate any bug fixes we make into the code. In fact, you should periodically check whether there are any new fixes that you need to download. To do so, run the `git pull` command while inside the repository.

If you use GitHub to manage your code, you must keep your repository private.

2.1 Code overview

The repository `minbert-default-final-project/` contains the following files:

- Files that should remain unedited:
  - `base_bert.py`: A base BERT implementation that can load pre-trained weights.
  - `config.py`: Classes for runtime configurations.
  - `optimizer_test.npy` A `NumPy` file containing weights for `optimizer_test.py`.
  - `optimizer_test.py` A test for your completed Adam Optimizer.
  - `sanity_check.data` A `data` file for `sanity_check.py`.
  - `sanity_check.py` A test for your completed `bert.py` file.
  - `tokenizer.py`: Implements `BertTokenizer` for text preprocessing.
  - `utils.py`: Utility functions and classes.
- Files with missing code blocks to be completed for Part 1:
  - `bert.py`: Your implementation of the BERT Model.
  - `classifier.py`: A classifier pipeline for running sentiment analysis.
  - `optimizer.py`: Your implementation of the Adam Optimizer.
- Files central to Part 2:
  - `multitask_classifier.py`: A classifier pipeline where you will train your minBERT implementation to simultaneously perform sentiment analysis, paraphrase detection, and semantic textual similarity tasks. There are missing code blocks in this file.

  - `datasets.py`: Functions and classes that load data for the three downstream tasks. You may edit this if you wish.
  - `evaluation.py`: Functions that evaluate an input model on the three downstream tasks. You will likely not have to edit this file.

In addition, there are two directories:

- `data/`. This directory contains the `train`, `dev`, and `test` splits of `sst` and `cfimdb` datasets as `.csv` files that you will be using in the first half of this projects. This directory also contains the `train`, `dev`, and `test` splits for later datasets that you will be using in the second half of this project. Note that we do not provide labels for the test sets.
- `predictions/` This directory will become populated with your model's output predictions on each of the provided datasets.

2.2 Setup

Once you are on an appropriate machine and have cloned the project repository, it's time to run the setup commands.

- Make sure you have Anaconda or Miniconda installed.
- `cd` into `minbert-default-final-project` and run `source setup.sh`
  - This creates a conda environment called `cs224n_dfp`.
  - In addition to the defaults installed, you may also have to install the following packages: `zipp-3.11.0`, `idna-3.4`, and `chardet-4.0.0`.
  - For the first part of this project, you are only allowed to use libraries that are installed by `setup.sh`. No other external libraries are allowed (e.g., `transformers`).
- Run `conda activate cs224n_dfp`
  - This activates the `cs224n_dfp` environment.
  - Remember to do this each time you work on your code.
- *(Optional)* If you would like to use PyCharm, select the `cs224n_dfp` environment. Example instructions for Mac OS X:
  - Open the `minbert-default-final-project` directory in PyCharm.
  - Go to `PyCharm > Preferences > Project > Project interpreter`.
  - Click the gear in the top-right corner, then `Add`.
  - Select `Conda environment > Existing environment > Click '...' on the right`.
  - Select `/Users/YOUR_USERNAME/miniconda3/envs/cs224n_dfp/bin/python`.
  - Select `OK` then `Apply`.

3 Implementing minBERT

We have provided you with several of the building blocks for implementing minBERT. In this section, we will describe the baseline BERT model as well as the sections of it that you must implement.

3.1 Details of BERT

Here, we walk through the BERT architecture and briefly mention details about BERT's original training procedure.

Tokenization (`tokenizer.py`)

The BERT model converts sentence input into tokens before performing any additional processing. Specifically, the BERT model uses a `WordPiece` tokenizer that splits sentences into word pieces. BERT has a predefined set of 30K different word pieces. These word pieces are then converted into ids for use in the rest of the BERT model. As an example, the following words are converted into the following word pieces:

Word	Word Pieces
snow	[snow]
snowing	[snow, ##ing]
fight	[fight]
fighting	[fight,##ing]
snowboard	[snow,##board]

In addition to separating each sentence into its constituent word pieces tokens, word pieces that have previously not been seen (i.e., that are not part of the original 30K word pieces) will be set as the [UNK] token.

To ensure that all input sentences have the same length, each input sentence is padded to a given `max_length` (512) with the [PAD] token. Finally, for many downstream tasks, BERT's hidden state of the first token is commonly used as the embedding for the entire sentence. To accommodate this, the [CLS] token is prepended to the token representation of each input sentence. In this first part of this assignment, you will be working with the hidden state of this token.

Lastly, BERT uses the [SEP] token to introduce an artificial separation between two input sentences. This separation token was essential for BERT's pretraining task of next-sentence prediction, but it will be largely irrelevant for our task of sentiment analysis.

Embedding Layer (`bert.BertModel.embed`)

After tokenizing and converting each token to ids, the BERT model utilizes a trainable embedding layer across each token. The input embeddings that are used in later portions of BERT are the sum of the token embeddings, the segmentation embeddings, and the position embeddings (Figure 1). Each embedding layer in the base version of BERT has a dimensionality of 768.

The learnable token embeddings map the individual input ids into vector representation for later use. More concretely, given some input word piece indices[5] $\boldsymbol{w}_1, \ldots, \boldsymbol{w}_k \in \mathbb{N}$, the embedding layer performs an embedding lookup to convert the indices into token embeddings $\boldsymbol{v}_1, \ldots, \boldsymbol{v}_k \in \mathbb{R}^D$.

[5] A *token index* is an integer that tells you which row (or column) of the embedding matrix contains the word's embedding.

The learnable segmentation embeddings are used to differentiate between sentence types. For this project, we only consider individual sentences and not next-sentence prediction tasks, so the segmentation embeddings are implemented with placeholders within our provided code base.

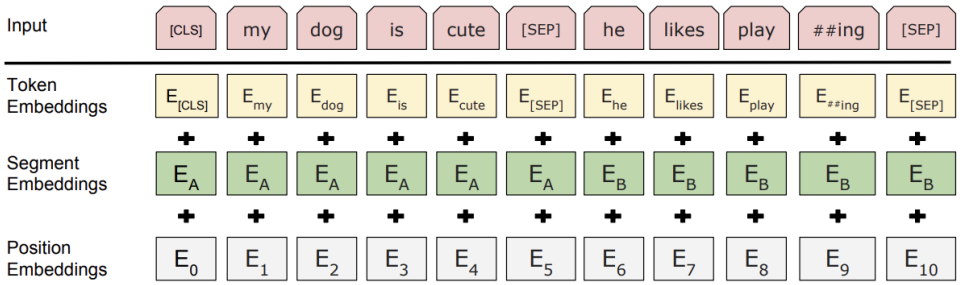

Figure 1: BERT embedding layer. The input embeddings that are used later in the model are the sum of the token embeddings, the segmentation embeddings, and the position embeddings. Figure 2 from [1].

Finally, the positional embeddings are used to encode the position of different words within the input. Like the token embeddings, position embeddings are parameterized embeddings that are learned for each of the 512 positions in a given BERT input.

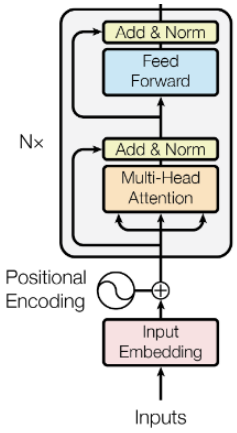

Figure 2: Encoder Layer of Transformer used in BERT. Left of Figure 1 from [5].

BERT Transformer Layer (`bert.BertLayer`)

As described in the original BERT paper [1], the base BERT makes use of 12 Encoder Transformer layers. These layers were defined initially in the work Attention is All You Need [5]. The Transformer layer of the BERT transformer, as seen in Figure 2, consists of multi-head attention, followed by an additive and normalization layer with a residual connection, a feed-forward layer, and a final additive and normalization layer with a residual connection. We briefly cover each of these layers here. We recommend that you read Section 3 of both cited papers for additional details.

Multi-head attention (`bert.BertSelfAttention.attention`) The multi-head attention layer [5] is the core of the transformer architecture that transforms hidden states for each element of a sequence based on the other elements (the fully-connected layers act on each element separately). The multi-head layer, which we write as $\text{MH}(\cdot)$, consists of n different dot-product attention mechanisms. At a high level, attention

represents a sequence element with a weighted sum of the hidden states of all the sequence elements. In multi-head attention the weights in the sum use dot product similarity between transformed hidden states. Concretely, the ith attention mechanism 'head' is:

$$\text{Attention}_i(\mathbf{h}_j) = \sum_t \text{softmax}\left(\frac{W_i^q \mathbf{h}_j \cdot W_i^k \mathbf{h}_t}{\sqrt{d/n}}\right) W_i^v \mathbf{h}_t \tag{1}$$

where $\mathbf{h}_j$ (we will drop the index "j" in the following equations to simplify) is a d dimensional hidden vector for a particular sequence element, and t runs over every sequence element. In BERT the W_i^q, W_i^k and W_i^v are matrices of size $d/n \times d$, and so each 'head' projects down to a different subspace of size d/n, attending to different information.
Finally the outputs of the n attention heads (each of size d/n) are concatenated together (which we show as $[\cdot, ..., \cdot]$) and linearly transformed with W^o (a $d \times d$ matrix)[6]:

$$\text{MH}(\mathbf{h}) = W^o \left[\text{Attention}_1(\mathbf{h}), ..., \text{Attention}_n(\mathbf{h})\right] \tag{2}$$

We further define another component of a BERT layer, the self-attention layer, which we write as $\text{SA}(\cdot)$:

$$\mathbf{h}' = \text{LN}(\mathbf{h} + \text{MH}(\mathbf{h})) \tag{3}$$

$$\text{SA}(\mathbf{h}) = \text{FFN}(\mathbf{h}'), \tag{4}$$

$\text{LN}(\cdot)$ is *layer normalisation* [6]. FFN is a standard *feed-forward network*,

$$\text{FFN}(\mathbf{h}) = W_2 f(W_1 \mathbf{h} + b_1) + b_2, \tag{5}$$

with $f(\cdot)$ a non-linearity, GeLU [7] in BERT. Matrix W_1 has size $d_{ff} \times d$ and W_2 has size $d \times d_{ff}$.
Putting this together, a BERT layer, which we write $\text{BL}(\cdot)$, is layer-norm applied to the output of a self-attention layer, with a residual connection.

$$\text{BL}(\mathbf{h}) = \text{LN}(\mathbf{h}' + \text{SA}(\mathbf{h})) \tag{6}$$

Dropout We lastly note that BERT applies dropout to the output of each sub-layer, **before it is added to the sub-layer input and normalized**. BERT also applies dropout to the sums of the embeddings and the positional encodings. BERT uses a setting of $p_{\text{drop}} = 0.1$.

BERT output (`bert.BertModel.forward`)

As specified throughout this section, BERT consists of

1. An embedding layer that consists of a word embedder and a position embedder.
2. BERT encoder layers, which are a stack of `config.num_hidden_layers` (in our case, 12) `BertLayers`.

The model outputs consist of:

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

Training BERT

The original version of BERT was trained using two unsupervised tasks on Wikipedia articles.

[6] [5] provide a more detailed motivation and discussion.

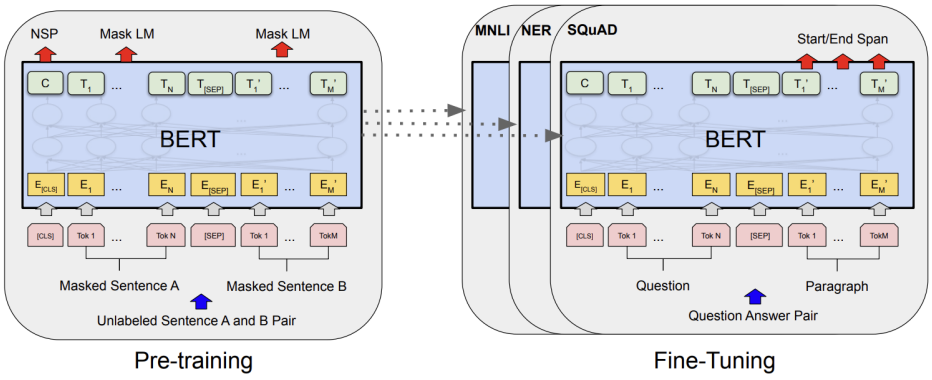

Figure 3: The original BERT model was trained on two unsupervised tasks, masked token prediction and next sentence prediction. Figure 1 from [1].

Masked Language Modeling In order to train BERT to extract deep bidirectional representations, the training procedure masks some percentage (15% in the original paper) of the word piece tokens and attempts to predict them. Specifically, the final hidden vectors that correspond to the masked tokens are fed into an output softmax layer over the vocabulary and are subsequently predicted.

To prevent a mismatch between initial pre-training and later fine-tuning, the "masked" tokens are not always replaced by the [MASK] token during training. Rather, the training data generator chooses 15% of the token positions at random for prediction; then, 80% of the chosen tokens are replaced with [MASK], 10% of the tokens are replaced with a random token, and another 10% of the tokens remain unchanged.

Next Sentence Prediction In order for BERT to understand the relationships between two sentences, BERT is further fine-tuned on the Next Sentence Prediction task. Specifically, the BERT model is shown a sentence and its next sentence 50% of the time; for the other 50% of the time, it is shown a random second sentence. The BERT model predicts whether the second input sentence is actually the next sentence.

3.2 Code To Be Implemented: Multi-head Self-attention and the Transformer Layer

We have provided you with much of the code for a BERT baseline model. Having gone over the basic structure of the BERT Transformer model, we will now describe the sections that need to be implemented:

BERT Multi-head Self-attention: `bert.BertSelfAttention.attention`

You must implement 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 as input the query and key.

BERT Transformer Layer: `bert.BertLayer` and `bert.BertModel`

After implementing the BERT multi-head self-attention layer, you must implement several more functions to realize the full BERT transformer layer. These code blocks can be found at `bert.BertLayer.add_norm`, `bert.BertLayer.forward`, and `bert.BertModel.embed`.

After finishing these steps, you can run a sanity check to verify the correctness of your implementation:

```
python sanity_check.py
```

The sanity check will load two embeddings we computed with our reference implementation and check whether your implementation outputs match ours.

4 Sentiment Analysis with the BERT

Having implemented a working minBERT model, you will now utilize pre-trained model weights and the output embeddings from your implemented BERT model to perform sentiment analysis on two datasets. In addition to loading these pre-trained model weights, you will (as done in Assignment 5), fine-tune these embeddings on each respective dataset to achieve better results. You will find both datasets in the `data` subfolder.

4.1 Datasets

Stanford Sentiment Treebank (SST) dataset

The Stanford Sentiment Treebank[7] [2] consists of 11,855 single sentences from movie reviews extracted from movie reviews. The dataset was parsed with the Stanford parser[8] and includes a total of 215,154 unique phrases from those parse trees, annotated by 3 human judges. Each phrase has a label of **negative**, **somewhat negative**, **neutral**, **somewhat positive**, or **positive**. You will utilize BERT embeddings to predict these sentiment classification labels.

For the SST dataset we have the following splits:

- `train` (8,544 examples)
- `dev` (1,101 examples)
- `test` (2,210 examples)

CFIMDB dataset

The CFIMDB dataset consists of 2,434 highly polar movie reviews. Each movie review has a binary label of **negative** or **positive**. You will utilize BERT embeddings to predict these sentiment classifications.

For the CFIMDB dataset we have the following splits:

- `train` (1,701 examples)
- `dev` (245 examples)
- `test` (488 examples)

4.2 Code To Be Implemented: Sentiment Classification with BERT embeddings

In the `classifer.py` file, you will find a pipeline that

1. Calls the BERT model to encode the sentences for their contextualized representations
2. Trains and evaluates your BERT model on the sentence classification examples
3. Saves your predictions

for both datasets.

In this file, you are to implement `BertSentimentClassifer`. This class encodes sentences using BERT and obtains the pooled representation of each sentence.[9] It then classifies the sentence by applying `dropout` on the pooled output and then projecting it using a `linear` layer. Already implemented is the model's capability to freeze or train parameters depending on whether we are using pre-trained weights or fine-tuning.

[7] https://nlp.stanford.edu/sentiment/treebank.html
[8] https://nlp.stanford.edu/software/lex-parser.shtml
[9] See the `forward` function in `bert.py` for how to access this representation.

4.3 Adam Optimizer

In addition to implementing `BertSentimentClassifer`, you will also implement the `step()` function of the Adam Optimizer based on Decoupled Weight Decay Regularization [8] and Adam: A Method for Stochastic Optimization [9].

Overview of the Adam Optimizer

The Adam optimizer is a method for efficient stochastic optimization that only requires first-order gradients. The method computes adaptive learning rates for different parameters by estimating the first and second moments of the gradients. Specifically, at each time step, the algorithm updates exponential moving averages of the gradient m_t and the squared gradient v_t where the hyperparameters $\beta_1, \beta_2 \in [0, 1)$ control the rate of exponential decay of these averages. Given that these moving averages are initialized at 0 at the initial time step, these averages are biased towards zero. As a result, a key aspect of this algorithm is performing bias correction to obtain $\hat{m}_t$ and $\hat{v}_t$ at each time step. We present the full algorithm below:

Algorithm 1 Adam algorithm. g_t^2 indicates the element-wise square $g_t \odot g_t$. All operations on vectors are element-wise. With B_1^t and B_2^t, we denote B_1 and B_2 to the power t.

Require: α : Stepsize
Require: $\beta_1, \beta_2 \in [0, 1)$: Exponential decay rates for the moment estimates
Require: $f(\theta)$: Stochastic objective function with parameters θ
Require: θ_0: Initial parameter vector
  $m_0 \leftarrow 0$ (Initialize 1st moment vector)
  $v_0 \leftarrow 0$ (Initialize 2nd moment vector)
  $t \leftarrow 0$ (Initialize time step)
  while θ_t not converged **do**
    $t \leftarrow t + 1$
    $g_t \leftarrow \nabla f_t(\theta_{t-1})$ (Get gradients w.r.t. stochastic objective function at timestep t)
    $m_t \leftarrow \beta_1 \cdot m_{t-1} + (1 - \beta_1) \cdot g_t$ (Update biased first moment estimate)
    $v_t \leftarrow \beta_2 \cdot v_{t-1} + (1 - \beta_2) \cdot g_t^2$ (Update biased second raw moment estimate)
    $\hat{m}_t \leftarrow m_t / (1 - \beta_1^t)$ (Compute bias-corrected first moment estimate)
    $\hat{v}_t \leftarrow v_t / (1 - \beta_2^t)$ (Compute bias-corrected second raw moment estimate)
    $\theta_t \leftarrow \theta_{t-1} - \alpha \cdot \hat{m}_t / (\sqrt{\hat{v}_t} + \epsilon)$
  return θ_t (Resulting parameters)

Note that, at the expense of clarity, there is a more *efficient version* of the above algorithm where the last three lines in the loop are replaced with the following two lines: $\alpha_t \leftarrow \alpha \cdot \sqrt{1 - \beta_2^t}/(1 - \beta_1^t)$ and $\theta_t \leftarrow \theta_{t-1} - \alpha_t \cdot m_t / (\sqrt{v_t} + \epsilon)$.

Code To Be Implemented: `optimizer.step`

You should implement the `step()` function of the Adam Optimizer. Our reference uses the "efficient" method of computing the bias correction presented in Section 2 "Algorithm" of [9], also mentioned in the above pseudo-code. Lastly, you should incorporate weight decay (by adding $\frac{\lambda}{2}\theta_t^2$ to your loss function, which is equivalent to subtracting $\alpha\lambda\theta_t$ from your parameters at the end of each gradient descent step, where λ is the weight decay regularization parameter) as your final update to the parameters. You can test your implementation by running:

```
python3 optimizer_test.py
```

4.4 Training minBERT for Sentiment Classification

After completing Part 1's implementations, you will experiment with your completed model using both pre-trained and fine-tuned embeddings on the SST and the CFIMDB datasets. You can run training by using the following command:

```
python3 classifier.py --fine-tune-mode {full-model,last-linear-layer} --lr {1e-3,1e-5}
```

Training for each dataset should take no more than 5 and 15 minutes (depending on your GPU). Use a learning rate of 1E-03 when running `last-linear-layer` and 1E-05 when running `full-model`. Following, check that your predictions directory is populated with the following 8 files:

```
{full-model,last-linear-layer}-sst-dev-out.csv
{full-model,last-linear-layer}-sst-test-out.csv
{full-model,last-linear-layer}-cfimdb-dev-out.csv
{full-model,last-linear-layer}-cfimdb-test-out.csv
```

As a baseline, your implementation should have results similar to the following (mean reference accuracies over 10 random seeds with their standard deviation shown in brackets):

Fine-tuning the last linear layer for SST: Dev Accuracy: **0.390** (0.007)
Fine-tuning the last linear layer for CFIMDB: Dev Accuracy: **0.780** (0.002)
Fine-tuning the full model for SST: Dev Accuracy: **0.515** (0.004)
Fine-tuning the full model for CFIMDB: Dev Accuracy: **0.966** (0.007)

You may *only* use the provided training set and dev set to train, tune, and evaluate your models. **If you use the official test data of these datasets to train, to tune, or to evaluate your models, or if you manually modify your CSV solutions in any way, you are committing an honor code violation.** For this section, for grading, we will largely be looking at your code/implementation (as well as your accuracies on the test set).

Submission Instructions for minBERT

You will submit the minBERT part of this project on Gradescope:

1. Verify that the following files exist at these specified paths within your assignment directory:
   - `bert.py`
   - `classifier.py`
   - `optimizer.py`
   - `predictions/{full-model,last-linear-layer}-sst-dev-out.csv`
   - `predictions/{full-model,last-linear-layer}-sst-test-out.csv`
   - `predictions/{full-model,last-linear-layer}-cfimdb-dev-out.csv`
   - `predictions/{full-model,last-linear-layer}-cfimdb-test-out.csv`
2. Run `prepare_submit.py` to produce your `cs224n_default_final_project_submission.zip` file.
3. Upload your `cs224n_default_final_project_submission.zip` file to the appropriate assignment on GradeScope.

At a high level, the submission file for the SST and CFIMDB `dev/test` datasets should look like the following:

```
id, Predicted_Sentiment
001fefa37a13cdd53fd82f617, 4
00415cf9abb539fbb7989beba, 2
00a4cc38bd041e9a4c4e545ff, 1
...
fffcaebf1e674a54ecb3c39df, 3
```

5 Extensions and Improvements for Additional Downstream Tasks

While we have focused on implementing key aspects of BERT in the first half of this project, for the rest of this project (and the part that will make up the bulk of your grade on the final project), you will have free rein to explore other datasets to better fine-tune and otherwise adjust your BERT embeddings so that they can simultaneously perform the following three tasks: **sentiment analysis**, **paraphrase detection**, and **semantic textual similarity**. The goal of this latter part of the project is to explore how to build robust embeddings that can perform well across a large range of different tasks!

In this section, we will be using the SST dataset for sentiment analysis, the Quora dataset for paraphrase detection, and the SemEval dataset for semantic textual analysis. You will find the `train`, `dev`, and `test` sets for each of these datasets in the `data` folder. You may *only* use our training set and our dev set to train, tune and evaluate your models. **If you use the official test data of these datasets to train, to tune, or to evaluate your models, or if you manually modify your CSV solutions in any way, you are committing an honor code violation.**

In addition to embeddings extracted from pre-trained BERT weights provided to you in the first part of this assignment, you are allowed to use other pre-existing NLP tools such as a POS tagger, dependency parser, Wordnet, coreference module, etc. that are not built on top of your trained embeddings. Your usage of external resources is limited to a reasonable degree. For example, you may **not** utilize pre-trained embeddings from the `transformers` library.

5.1 Dataset Overview

Quora Dataset

The Quora dataset, as previously described in Section 1, consists of 404,298 question pairs with labels indicating whether particular instances are paraphrases of one another. We have provided you with the following splits:

- `train` (283,010 examples)
- `dev` (40,429 examples)
- `test` (80,859 examples)

Given the binary labels of this dataset, we use accuracy to evaluate your performance.

SemEval STS Benchmark Dataset

The SemEval STS Benchmark dataset as described in Section 1 consists of 8,628 different sentence pairs of varying similarity on a scale from 0 (unrelated) to 5 (equivalent meaning).

For the STS dataset, we have the following splits:

- `train` (6,040 examples)
- `dev` (863 examples)
- `test` (1,725 examples)

When testing this dataset, we will use, as in the original SemEval [4] paper, Pearson correlation of the true similarity values against the predicted similarity values.

5.2 Code Overview

We have provided you a set of classes and functions, some containing missing code blocks, that will help you train and evaluate for Part 2 of this project. We are not grading your code for this part, so you are free to re-organize, create new classes and functions, or otherwise retrofit your old code. Here, we give a brief overview of the provided code:

- `multitask_classifier.MultitaskBERT`: A class that imports the weights of a pre-trained BERT model and predicts sentiment, paraphrases, and semantic textual similarity.
- `multitask_classifier.MultitaskBERT.forward`: Invokes your previously implemented BERT model to output sentence embeddings. You can choose to experiment with the contextual word embeddings of particular word pieces or extract just the `pooler_output` as in `classifier.py`.
- `multitask_classifier.MultitaskBERT.predict_sentiment`: Predicts the sentiment of a sentence based on BERT embeddings. As a baseline, you should call the new `forward()` method above followed by a dropout and linear layer as in `classifier.py`.
- `multitask_classifier.MultitaskBERT.predict_paraphrase`: Predicts whether two sentences are paraphrases of each other based on their embeddings.
- `multitask_classifier.MultitaskBERT.predict_similarity`: Predicts the similarity of two sentences based on their embeddings.
- `multitask_classifier.train_multitask()`: A function for training your model. It is largely your choice how to train your model. As a placeholder, you will find a copy of the original code from `classifier.py`, which trains your model on only the SST sentiment dataset.
- `multitask_classifier.test_multitask()`: A function for testing your model. This function also saves predictions for both dev and test sets of all three tasks. It's unlikely that you will edit this function.
- `evaluation.model_eval_multitask()`: A function that evaluates an input Multitask BERT model on the dev sets of all three tasks. You may find it useful to use this function in your implementation of `multitask_classifier.train_multitask()`.
- `datasets.SentenceClassificationDataset`: A class for handling the SST sentiment dataset. Feel free to edit this class if you wish to modify the way in which the data examples are preprocessed.
- `datasets.SentencePairDataset`: A class for handling the SemEval and Quora datasets. Feel free to edit this class if you wish to modify the way in which the data examples are preprocessed.

The following command is a convenient way to train, evaluate, and save the predictions of your multitask BERT model for the three datasets all at once:

```
python3 multitask_classifier.py [OPTIONS]
```

5.3 Possible Extensions

There are many possible extensions that can improve your model's performance on the SST, Quora, and SemEval STS datasets simultaneously. We recommend that you find a relevant research paper for each improvement that you wish to attempt. Here, we provide some suggestions, but you might look elsewhere for interesting ways of improving sentence embeddings for the three selected tasks.

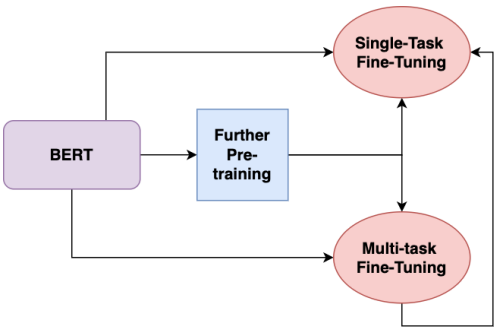

Figure 4: To improve your BERT model, there are several different paths that you can take including (1) additional pre-training on BERT's original objectives, (2) fine-tuning your model directly on a single task, (3) multi-task training your BERT model.

Additional Pretraining

Original Paper: How to Fine-Tune BERT for Text Classification? [10]
As outlined in Section 3, BERT was trained in a general domain, which has a different data distribution than the datasets that we will use to grade your project. A natural way to improve your model would be to further pre-train your BERT model with target-domain data. This would involve implementing and training on the masked LM objective or predicting tokens as outlined in Section 3, using the training datasets that we provided. For more details on BERT's pre-training, see [1].

Multiple Negatives Ranking Loss Learning

Original Paper: Efficient Natural Language Response Suggestion for Smart Reply [11]

Another effective way of improving your embeddings would be to fine-tune your model with Multiple Negative Ranking Loss[10]. With this loss function, training data consists of sets of K sentence pairs $[(a_1, b_1), \ldots, (a_n, b_n)]$ where a_i, b_i are labeled as similar sentences and all (a_i, b_j) where $i \neq j$ are not similar sentences. The loss function then minimizes the distance between a_i, b_i while it simultaneously maximizing the distance (a_i, b_j) where $i \neq j$. Specifically, training is to minimize the approximated mean negative log probability of the data. For a single batch, this is calculated as

$$\begin{aligned}\mathcal{J}(x, y, \theta) &= -\frac{1}{K}\sum_{i=1}^{K} \log P_{\text{approx}}(y_i|x_i) \\ &= -\frac{1}{K}\sum_{i=1}^{K} \left[S(x_i, y_i) - \log \sum_{j=1}^{K} e^{S(x_i, y_j)}\right],\end{aligned}$$

where θ represents the word embeddings and neural network parameters used to calculate S, a scoring function. See sbert.net[11] and Henderson et al. [11] for additional details.

Cosine-Similarity Fine-Tuning

Sentence-BERT: Sentence Embeddings using Siamese BERT-Networks [12]

Additional fine-tuning can further improve your BERT model on one of the pre-selected tasks. The similarity between two embeddings is often computed using their cosine similarity. A way of potentially improving

[10] https://www.sbert.net/docs/package_reference/losses.html
[11] https://www.sbert.net/examples/training/nli/README.html#multiplenegativesrankingloss

your embeddings would thus be to utilize CosineEmbeddingLoss[12] while fine-tuning on the SemEval dataset. In this setup, sentences that are the equivalent have a cosine similarity of 1 and those that are unrelated have a cosine similarity score of 0.

Fine-Tuning with Regularized Optimization

Original paper: SMART: Robust and Efficient Fine-Tuning for Pre-trained Natural Language Models through Principled Regularized Optimization [13]

Aggressive fine-tuning can often cause over-fitting. This can cause the model to fail to generalize to unseen data. To combat this in a principled manner, Jiang et al. propose (1) Smoothness-inducing regularization, which effectively manages the complexity of the model and (2) Bregman proximal point optimization, which is an instance of a trust-region method and can prevent aggressive updating.

Smoothness-Inducing Adversarial Regularization Specifically, given the model $f(\cdot;\theta)$ and n data points of the target task denoted by $\{(x_i, y_i)\}_{i=1}^n$ where x_i's denote the embedding of the input sentences obtained from the first embedding layer of the language model and y_i's are the associated labels, Jiang et al.'s method essentially solves the following optimization for fine-tuning:

$$\min_{\theta} = \mathcal{L}(\theta) + \lambda_s \mathcal{R}_s(\theta) \tag{7}$$

where $\mathcal{L}(\theta)$ is the loss function defined as:

$$\mathcal{L}(\theta) = \frac{1}{n}\sum_{i=1}^{n} l(f(x_i;\theta), y_i), \tag{8}$$

and $l(\cdot,\cdot)$ is the loss function depending on the target task, $\lambda_s > 0$ is a tuning parameters and $\mathcal{R}_s(\theta)$ is the smoothness-inducing regularizer defined as

$$\mathcal{R}_s(\theta) = \frac{1}{n}\sum_{i}^{n} \max_{||\tilde{x_i} - x_i||_p \leq \epsilon} l(f(\tilde{x_i};\theta), f(x_i;\theta)), \tag{9}$$

where $\epsilon > 0$ is a tuning parameter. Note that for classification tasks, $f(\cdot;\theta)$ outputs a probability simplex and l_s is chosen as the symmetrized KL-divergence, *i.e.*,

$$l_s(P, Q) = \mathcal{D}_{KL}(P||Q) + \mathcal{D}_{KL}(Q||P) \tag{10}$$

Bergman Proximal Point Optimziation Jiang et al. also propose a class of Bregman proximal point optimization[13] methods to solve Equation 7. Such optimization methods impose a strong penalty at each iteration to prevent the model from aggressive updating. Specifically, they use a pre-trained model as the initialization denoted by $f(\cdot;\theta_0)$. At the $(t+1)$-th iteration, the vanilla Bregman proximal point (VBPP) method takes:

$$\theta_{t+1} = \arg\min_{\theta} \mathcal{F}(\theta) + \mu \mathcal{D}_{\text{Breg}}(\theta, \theta_t) \tag{11}$$

where $\mu > 0$ is a tuning parameter and $\mathcal{D}_{\text{Breg}}(\cdot,\cdot)$ is the Bregman divergence defined as:

$$\mathcal{D}_{\text{Breg}}(\theta, \theta_t) = l_s(f(\tilde{x_i};\theta), f(x_i;\theta_t)) \tag{12}$$

See https://github.com/namisan/mt-dnn and [13] for additional details.

[12] https://pytorch.org/docs/stable/generated/torch.nn.CosineEmbeddingLoss.html
[13] https://www.stat.cmu.edu/~ryantibs/convexopt/lectures/bregman.pdf

Multitask Fine-Tuning

Original paper: BERT and PALs: Projected Attention Layers for Efficient Adaptation in Multi-Task Learning [14]
Original paper: MTRec: Multi-Task Learning over BERT for News Recommendation [15]
Original paper: Gradient surgery for multi-task learning. [16]

Rather than fine-tuning BERT on individual tasks, you can alternatively make use of multi-task learning to update BERT. For example, Bi et al. [15], use multi-task learning adding together the losses on the tasks of category classification and named entity recognition.

$$\mathcal{L}_{total} = \mathcal{L}_{task1} + \mathcal{L}_{task2} \tag{13}$$

Using multi-task learning, however, depending on how the model is fine-tuned, is not always beneficial. Gradient directions of different tasks may conflict with one another. Yu et al. [16] recommend a technique called Gradient Surgery that projects the gradient of the i-th task $\mathbf{g_i}$ onto the normal plane of another conflicting task's gradient $\mathbf{g_j}$:

$$\mathbf{g_i} = \mathbf{g_i} - \frac{\mathbf{g_i} \cdot \mathbf{g_j}}{||\mathbf{g_j}||^2} \cdot \mathbf{g_j} \tag{14}$$

Contrastive Learning

Original paper: Simple Contrastive Learning of Sentence Embeddings [17]

Gao et al [17] proposed a simple contrastive learning framework called SimCSE that works with both unlabeled and labeled data. Unsupervised SimCSE simply takes an input sentence and predicts itself in a contrastive learning framework, with only standard dropout used as noise. In contrast, supervised SimCSE incorporates annotated pairs from NLI datasets into contrastive learning by using entailment pairs as positives and contradiction pairs as hard negatives. You can utilize a similar approach to better your sentence embeddings across your different models.

5.3.1 Parameter Efficient Finetuning Methods

Original paper: LoRA: Low-Rank Adaptation of Large Language Models [18]
Original paper: DoRA: Weight-Decomposed Low-Rank Adaptation [19]

Several techniques exist to reduce the memory usage/running time required to finetune language models, while preserving performance. One such technique is LoRA [18], which is motivated by the hypothesis that even during full fine-tuning, the updates to the weight matrices of the language model are essentially low rank. Thus, given a pre-trained model, Hu, et al. [18] propose re-parameterizing the weight matrices W of the model as $W_0 + BA$, where W_0 denotes the weight matrix obtained from pre-training, and A, B are parameters newly added during fine-tuning. Here, if W_0 is of shape $d \times k$, then B and A will be of shapes $d \times r$ and $r \times k$ respectively, for some $r \ll d, k$. A significant advantage of this technique is that less memory is required for the optimizer (which needs to store some state for each parameter being optimized).

It can be the case that LoRA performs worse than standard fine-tuning. Liu, et al. [19] propose DoRA, which aims to address the cause of this. In DoRa, the pre-trained $d \times k$ weight matrix W_0 is now re-parameterized as $m \cdot \frac{V}{\|V\|_c}$, where V is a $d \times k$ matrix, $\|V\|_c$ is a $1 \times k$ vector where the i^{th} entry is the magnitude of the i^{th} column of V, and m is a $1 \times k$ vector which contains a learnable magnitude for each column. In other words, DoRA attempts to decouple updates to the magnitude of W_0 and updates to the direction of W_0.

Afterwards, the direction component V is updated using LoRA. In other words, V is itself re-parameterized as $W_0 + BA$, where B and A are of shape $d \times r$ and $r \times k$ respectively, for some $r \ll d, k$. Afterwards, only

B and A receive updates. To significantly reduce memory required for backpropagation, Liu, et al. propose treating the factor $\frac{1}{\|V\|_c}$ as a constant, and detaching it from the backpropagation graph (while otherwise computing it as usual).

Additional Datasets

Original paper: SMART: Robust and Efficient Fine-Tuning for Pre-trained Natural Language Models through Principled Regularized Optimization [13]

Original paper: Sentence-BERT: Sentence Embeddings using Siamese BERT-Networks [12]

Fine-tuning the BERT model on different datasets is an additional approach that you could apply. There are a host of different datasets and tasks that you can potentially apply to your model to get richer and more robust embeddings. See the following for some example datasets:

- https://arxiv.org/abs/1508.05326
- http://sbert.net/datasets/paraphrases
- https://arxiv.org/abs/1704.05426

Additional input features

Although deep learning is able to learn end-to-end without the need for feature engineering, it turns out that using the right input features (e.g., part-of-speech tag, named entity type, etc...) can still boost performance significantly. If you implement a model like this, comment on the tradeoff between feature engineering and end-to-end learning in your report.

Other improvements

There are many other subtle ways to improve your performance. The suggestions in this section are just some examples; it will take time to run the necessary experiments and draw the necessary comparisons.

- **Regularization.** The baseline code uses dropout. You could further experiment with different values of dropout and different types of regularization.
- **Sharing weights.** The baseline code outlines a way to use distinctive heads for predicting whether sentences are paraphrases, their semantic similarity, and each sentence's sentiment. You could potentially share some layers between task heads to improve performance.
- **Model size and the number of layers.** With any model, you can try increasing layer size or the number of layers.
- **Optimization algorithms.** The baseline uses the Adam optimizer. PyTorch supports many other optimization algorithms. You can also try varying the learning rate.
- **Ensembling.** Ensembling almost always boosts performance, so try combining several of your models together for your final submission. However, ensembles are more computationally expensive to run.
- **Hyperparameter optimization.** While we provide some defaults for various hyperparameters, these do not necessarily lead to the best results. Another approach would be to perform a hyperparameter search to find the best hyperparameters for your model.

Other Approaches

The models and techniques we have presented here are far from exhaustive. There are many published papers on both related and unrelated tasks.[14] These papers may contain interesting ideas that you can apply to build more robust and semantically rich embeddings.

[14] http://nlpprogress.com/

Submission Instructions

You will submit the Extensions part of this project on Gradescope.

1. Run `prepare_submit.py`. This command should capture all your `*.py` files and prediction `*.csv` files in a single zip file.
2. Verify that the generated `cs224n_default_final_project_submission.zip` file includes your model predictions on the three downstream tasks in a `predictions/*` directory as well as all necessary code for replicating your results.
3. Upload your `cs224n_default_final_project_submission.zip` file to the appropriate assignment on Gradescope.

6 Submitting to the Leaderboard

6.1 Overview

In addition to submitting your `.zip` file, you should submit your predictions on the three datsets to two Gradescope leaderboards.[15]

You are allowed to submit to the `dev` leaderboard as many times as you like, but **you will only be allowed 3 successful submissions to the `test` leaderboard**. For your final report, we will ask you to choose a single test leaderboard submission to consider for your final performance. Therefore, you must make at least one submission to the test leaderboard, but be careful not to use up your test submissions before you have finished developing your best model.

Submitting to the leaderboard is similar to submitting any other assignment on Gradescope, except that your submission is a CSV file of predictions on the `dev/test` set. At a high level, the submission file for the SST dataset should look like the following:

```
id, Predicted_Sentiment
 001fefa37a13cdd53fd82f617, 4
 00415cf9abb539fbb7989beba, 2
 00a4cc38bd041e9a4c4e545ff, 1
 ...
 fffcaebf1e674a54ecb3c39df, 3
```

The submission file for the STS dataset should look like the following:

```
 id, Predicted_Similarity
 8f4d49b9f4558f9e45423e84c, 1.000
 1c5cd37407630a3ba19a0f2ad, 0.4051
 318c885e36cc9e6f6bb7de7dd, 0.2138
 ...
 4e1ef3b635d01039a8a8f059b, 0.7462
```

The submission file for the Paraphrase dataset should look like the following:

```
 id,  Predicted_Is_Paraphrase
 872887985e1e0f2dd5b690ffd, 1
 472398907a6adb9ed2f660550, 0
 c3ceaaed421cc008282efdf8a, 0
 ...
 5e10dfc4ac8ae205f3e114445, 1
```

The header is required as well as the first column being a 25-digit hexadecimal ID for each example (IDs defined in each of the respective test/dev files), and the last column is your predicted answer (or the empty string for no answer). The rows can be in any order. You must submit a prediction for every example.

6.2 Submission Steps

Here are the concrete steps for submitting to the leaderboard:

1. Generate prediction .csv files using your multitask BERT model.

[15] We will display the accuracy of your final model on the SST `test/dev` dataset, the accuracy of your model on the Quora `test/dev` dataset, the Pearson score for your model on the STS SemEval `test/dev` dataset, as well as an aggregate score across all three tasks.

2. **Choose the correct leaderboard for your split (DEV vs. TEST).**

3. Upload your submission and wait for your scores. The submission output will tell you the submission's accuracies/Pearson correlation on the `dev/test` datasets. It will also display your current performance's delta w.r.t. your best submission. Your placement on the leaderboard is according to your best submission, not necessarily your most recent submission.

There should be useful error messages if anything goes wrong. If you get an error that you do not understand, please make a post on Ed.

7 Grading Criteria

The final project will be graded holistically. This means we will look at many factors when determining your grade: the creativity, complexity, and technical correctness of your approach, your thoroughness in exploring and comparing various approaches, the strength of your results, the effort you applied, and the quality of your write-up, evaluation, and error analysis. Generally, implementing more complicated models represents more effort, and implementing more unusual models (e.g., ones that we have not mentioned in this handout) represents more creativity. You are not required to pursue original ideas, but the best projects in this class will go beyond the ideas described in this handout and may in fact become published work themselves!

As in previous years, for Part 2 of this project, an aspect of your grade will be your performance relative to the leaderboard as a whole across all tasks. Note that the strength of your results on the leaderboard is only one of the many factors we consider in grading. Our focus is on evaluating your well-reasoned research questions, explanations, and experiments that clearly evaluate those questions.

There is no pre-defined accuracy (SST, Quora) or Pearson score (SemEval) to ensure a good grade. Similarly, there is no pre-defined rule for which of the extensions in Section 5 (or elsewhere) will ensure a good grade. Implementing a small number of things with good results and thorough experimentation/analysis is better than implementing a large number of things that don't work or barely work. In addition, the quality of your write-up and experimentation is important: we expect you to convincingly show that your techniques are effective and describe why they work (or clearly elucidate reasoning for the cases that don't work).

As with all final projects, larger teams are expected to do correspondingly larger projects. We will expect more complex things implemented, more thorough experimentation, and better results from teams with more people.

8 Honor Code

Any honor code guidelines that apply to the custom final project in general also apply to the default final project. Here are some guidelines that are specifically relevant to the default final project:

1. You **may not** use a pre-existing implementation of minBERT.
2. You are **not** allowed to use pre-trained contextual embeddings (such as ELMO, GPT, etc) for your system. You are allowed to use other pre-existing NLP tools such as a POS tagger, dependency parser, and coreference module that are not built on top of pre-trained contextual embeddings.
3. You are free to discuss ideas and implementation details with other teams (in fact, we encourage it!). However, under no circumstances may you look at another CS 224N team's code or incorporate their code into your project.
4. As described elsewhere, it is an honor code violation to use the official SST, Quora, or SemEval datasets in any way.
5. Do not share your code publicly (e.g., in a public GitHub repo) until after the class has finished.

References

[1] Jacob Devlin, Ming-Wei Chang, Kenton Lee, and Kristina Toutanova. Bert: Pre-training of deep bidirectional transformers for language understanding. *arXiv preprint arXiv:1810.04805*, 2018.

[2] Richard Socher, Alex Perelygin, Jean Wu, Jason Chuang, Christopher D Manning, Andrew Y Ng, and Christopher Potts. Recursive deep models for semantic compositionality over a sentiment treebank. In *Proceedings of the 2013 conference on empirical methods in natural language processing*, pages 1631–1642, 2013.

[3] Samuel Fernando and Mark Stevenson. A semantic similarity approach to paraphrase detection. In *Proceedings of the 11th annual research colloquium of the UK special interest group for computational linguistics*, pages 45–52, 2008.

[4] Eneko Agirre, Daniel Cer, Mona Diab, Aitor Gonzalez-Agirre, and Weiwei Guo. * sem 2013 shared task: Semantic textual similarity. In *Second joint conference on lexical and computational semantics (* SEM), volume 1: proceedings of the Main conference and the shared task: semantic textual similarity*, pages 32–43, 2013.

[5] Ashish Vaswani, Noam Shazeer, Niki Parmar, Jakob Uszkoreit, Llion Jones, Aidan N Gomez, Łukasz Kaiser, and Illia Polosukhin. Attention is all you need. In *Advances in Neural Information Processing Systems*, pages 5998–6008, 2017.

[6] Jimmy Lei Ba, Jamie Ryan Kiros, and Geoffrey E Hinton. Layer normalization. *arXiv preprint arXiv:1607.06450*, 2016.

[7] Dan Hendrycks and Kevin Gimpel. Gaussian error linear units (gelus). *arXiv preprint arXiv:1606.08415*, 2016.

[8] Ilya Loshchilov and Frank Hutter. Decoupled weight decay regularization. *arXiv preprint arXiv:1711.05101*, 2017.

[9] Diederik P Kingma and Jimmy Ba. Adam: A method for stochastic optimization. *arXiv preprint arXiv:1412.6980*, 2014.

[10] Chi Sun, Xipeng Qiu, Yige Xu, and Xuanjing Huang. How to fine-tune bert for text classification? In *Chinese Computational Linguistics: 18th China National Conference, CCL 2019, Kunming, China, October 18–20, 2019, Proceedings 18*, pages 194–206. Springer, 2019.

[11] Matthew Henderson, Rami Al-Rfou, Brian Strope, Yun-Hsuan Sung, László Lukács, Ruiqi Guo, Sanjiv Kumar, Balint Miklos, and Ray Kurzweil. Efficient natural language response suggestion for smart reply. *arXiv preprint arXiv:1705.00652*, 2017.

[12] Nils Reimers and Iryna Gurevych. Sentence-bert: Sentence embeddings using siamese bert-networks. In *Proceedings of the 2019 Conference on Empirical Methods in Natural Language Processing and the 9th International Joint Conference on Natural Language Processing (EMNLP-IJCNLP)*, pages 3982–3992, 2019.

[13] Haoming Jiang, Pengcheng He, Weizhu Chen, Xiaodong Liu, Jianfeng Gao, and Tuo Zhao. Smart: Robust and efficient fine-tuning for pre-trained natural language models through principled regularized optimization. *arXiv preprint arXiv:1911.03437*, 2019.

[14] Asa Cooper Stickland and Iain Murray. Bert and pals: Projected attention layers for efficient adaptation in multi-task learning. In *International Conference on Machine Learning*, pages 5986–5995. PMLR, 2019.

[15] Qiwei Bi, Jian Li, Lifeng Shang, Xin Jiang, Qun Liu, and Hanfang Yang. Mtrec: Multi-task learning over bert for news recommendation. In *Findings of the Association for Computational Linguistics: ACL 2022*, pages 2663–2669, 2022.

[16] Tianhe Yu, Saurabh Kumar, Abhishek Gupta, Sergey Levine, Karol Hausman, and Chelsea Finn. Gradient surgery for multi-task learning. *Advances in Neural Information Processing Systems*, 33:5824–5836, 2020.

[17] Tianyu Gao, Xingcheng Yao, and Danqi Chen. SimCSE: Simple contrastive learning of sentence embeddings. In *Empirical Methods in Natural Language Processing (EMNLP)*, 2021.

[18] Edward J. Hu, Yelong Shen, Phillip Wallis, Zeyuan Allen-Zhu, Yuanzhi Li, Shean Wang, Lu Wang, and Weizhu Chen. Lora: Low-rank adaptation of large language models. In *The Tenth International Conference on Learning Representations, ICLR 2022, Virtual Event, April 25-29, 2022*. OpenReview.net, 2022.

[19] Shih-Yang Liu, Chien-Yi Wang, Hongxu Yin, Pavlo Molchanov, Yu-Chiang Frank Wang, Kwang-Ting Cheng, and Min-Hung Chen. Dora: Weight-decomposed low-rank adaptation. *CoRR*, abs/2402.09353, 2024.