CS 224N: Default Final Project: minBERT and Downstream Tasks

Contents

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

1 Overview

In this assignment, you will implement some of the most important components of the BERT model so that you can better understand its architecture. Using pre-trained weights loaded into your BERT model, you will then perform sentence classification on `sst` dataset and `cfimdb` dataset with the BERT model.

After going through this initial exercise, you will examine how to fine-tune BERT's contextualized embeddings to simultaneously perform well on multiple sentence-level tasks (**sentiment analysis**, **paraphrase detection**, and **semantic textual similarity**). The goal of this section is to allow you to experiment with different options to obtain robust and generalizable sentence embeddings that perform well in different settings.

Note on default project vs custom project: The effort/work/difficulty that goes into the default final project is not intended to be less compared to the custom project. It is just that the specific kind of difficulty around coming up with your own problem and evaluation methods was intended to be excluded, allowing students to focus an equivalent amount of effort on this 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 the deeply bidirectional word representations, released in 2018, BERT took a large leap forward 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 dataset example, 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 from those parse trees, each annotated by 3 human judges. Each phrase has a label of negative, somewhat negative, neutral, somewhat positive, or positive.

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. Paraphrases are "rewordings of something written or spoken by someone else"; paraphrase detection thus essentially seeks to determine whether particular words or phrases convey the same semantic meaning [3]. 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.

As a concrete dataset example, 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 labeled whether different questions were paraphrases of each other.

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 it is not a yes or no decision; rather STS allows for degrees of similarity. For example, on a scale from 5 (same meaning) to 0 (not at all related), the following sentences have the following relationships to each other:[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/S131004.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.

The goal models trained on the STS task (often using a cosine similarity metric based on their word embedding) would be to appropriately train a model to predict the similarity of each of the above sentences in terms of their semantic content.

1.5 This Project

The goal of this project is for you to first implement some of the key aspects of the original BERT model including multi-head self-attention as well as a Transformer layer. Subsequently, you will utilize your completed BERT model to perform sentiment analysis on the Stanford Sentiment Treebank dataset as well as another dataset of movie reviews. Finally, in the latter half of this project, you will fine-tune and otherwise extend the BERT model to create sentence embeddings that can perform well across a wide range of downstream tasks. In Section 5, we describe several techniques that are commonly used to create more robust and semantically-rich sentence embeddings from BERT models – most come from recent research papers. We provide these suggestions to help you get started. They should all improve over the baseline if implemented correctly (and note that there is usually more than one way to implement something correctly).

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 doesn't necessarily have to be a completely new approach – small but well-motivated changes to existing models are very valuable, especially if followed by good analysis. If you can show quantitatively and qualitatively that your small but original change improves a state-of-the-art model (and even better, explain what particular problem it solves and how), then you will have done extremely well.

Like the custom final project, the default final project is open-ended – it will be up to you to figure out what to do. In many cases there won't be one correct answer for how to do something – it will take experimentation to determine which way is best. We are expecting you to exercise the judgment and intuition that you've gained from the class so far to build your models. For more information on grading criteria, see Section 7.

Note that this document only describes the code portion of the Default Final Project. For more details on the written portion see the course website and the handout *CS224n: Project Proposal 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 Azure, similarly to Assignments 4 and 5 – remember you can refer to the *Azure Guide* and *Practical Guide to VMs* linked on the class webpage. As before, remember that Azure credit is charged for every minute that your VM is on, so it's important that your VM is only turned on when you are actually training your models.

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 Azure VM only once you've debugged your code and you're ready to train. We advise that you use GitHub to manage your codebase and sync it between the two machines (and between team members) – the *Practical Guide to VMs* has more information on this.

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

```
git clone https://github.com/gpoesia/minbert-default-final-project
```

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, navigate to the `minbert-default-final-project` directory and run the `git pull` command.

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:

- `base_bert.py`: A base BERT implementation that can load pre-trained weights against which you can test your own implementation.
- `bert.py`: This file contains the BERT Model. There are several sections of this implementation that need to be completed.
- `config.py`: This is where the configuration class is defined. You won't need to modify this file in this assignment.
- `classifier.py`: A classifier pipeline for running sentiment analysis. There are several sections of this implementation that need to be completed.
- `multitask_classifier.py`: A classifier pipeline for the second half of the project where you will train your minBERT implementation to simultaneously perform sentiment analysis, paraphrase detection, and semantic textual similarity tasks.
- `datasets.py`: A dataset handling script for the second half of this project.
- `evaluation.py`: A evaluations handling script for the second half of this project.
- `optimizer.py`: An implementation of the Adam Optimizer. The `step()` function of the Adam optimizer needs to be completed.

- `optimizer_test.py` A test for your completed Adam Optimizer.
- `optimizer_test.npy` A `numPy` file containing weights for use in the `optimizer_test.py`
- `sanity_check.py` A test for your completed `bert.py` file.
- `sanity_check.data` A `data` file for use in the `sanity_check.py`.
- `tokenizer.py`: This is where `BertTokenizer` is implemented. You won't need to modify this file in this assignment.
- `utils.py`: Utility functions and classes.

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.
- `predictions/` This directory will contain the outputted predictions of your models 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 assignment, you are only allowed to use libraries that are installed by `setup.sh`, no other external libraries are allowed (e.g., `transformers`).
  - Do not change any of the existing command options (including defaults) or add any new required parameters
- 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

Bidirectional Encoder Representations from Transformers or BERT is a transformer-based model that generates contextual word representations. Here we will, walk through the BERT model as well as give an overview of how the original BERT model was trained.

Tokenization (`tokenizer.py`)

The BERT model converts sentence input into tokens before performing any additional processing. Specifically, the BERT model utilizes a `WordPiece` tokenizer that splits sentences into individual words into word pieces. BET 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 further padded to a given `max_length` (512) with the [PAD] token. Finally, for many downstream tasks, BERT represents sentence embeddings with the hidden state of the first token. As a result, in BERT's implementation [CLS] 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.

Note, a part of vanilla BERT's training was a next-sentence prediction task. To help differentiate the first sentence from the second sentence input (given that BERT does not take in two distinct inputs as other models), the [SEP] token was further added to introduce an artificial separation between inputted sentences.

Embedding Layer (`bert.BertModel.embed`)

After tokenizing and converting each token to ids, the BERT model subsequently 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. 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 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 utilized to differentiate amongst different sentences input into the model. We note that for this project, we do not consider the segmentation embeddings (we only consider individual sentences and not next-sentence prediction tasks) and they are implemented as only a placeholder within our provided code base.

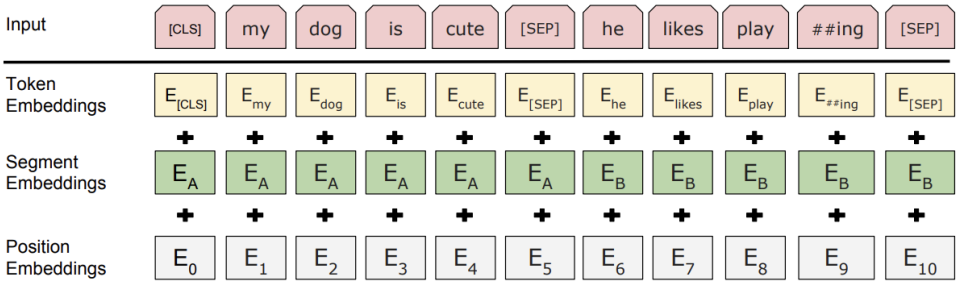

Figure 1: BERT embedding layer. The input embeddings that are utilized later in the model are the sum of the token embeddings, the segmentation embeddings, and the position embeddings. Figure from [1]

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

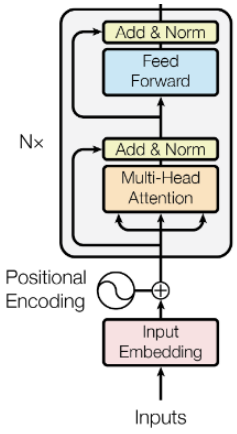

Figure 2: Encoder Layer of Transformer used in BERT. Figure from [5]

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 3 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.

Multiheaded Self-Attention (`bert.BertSelfAttention.attention`) Multi-head Self-Attention consists of a scaled-dot product applied across multiple different heads. Specifically, the input to each head is to a

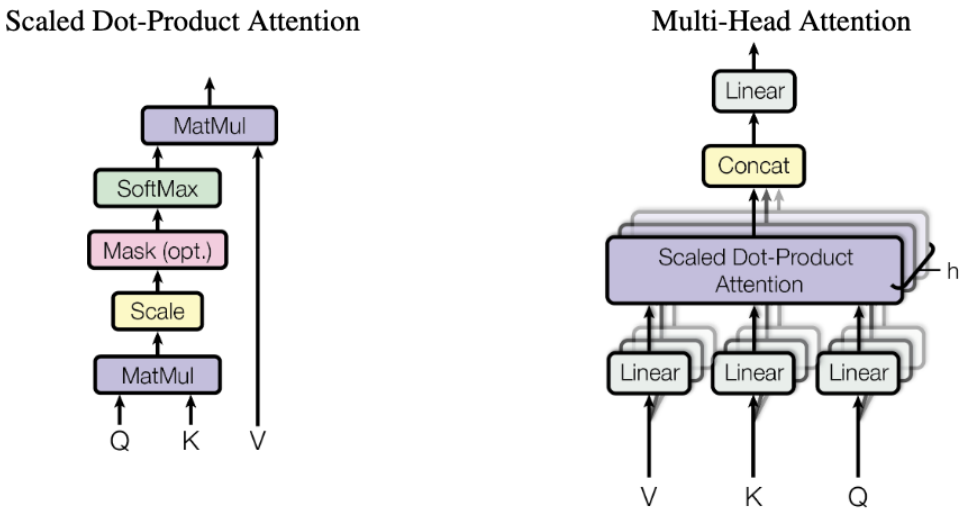

Figure 3: Scaled Dot-Product and Multi-Head Self-Attention. Figure from [5]

scaled-dot product that consists of queries and keys of dimension d_k, and values of dimension d_v. BERT computes the dot products of the query with all keys, divides each by $\sqrt{d_k}$, and applies a softmax function to obtain the weights on the values. In practice, BERT computes the attention function on a set of queries simultaneously, packed together into a matrix Q. The keys and values are also packed together into matrices K and V. Scaled dot-product attention is thus computed as:

$$Attention(Q, K, V) = Softmax(\frac{QK^T}{\sqrt{d_k}})V \tag{1}$$

Multi-head attention allows the model to jointly attend to information from different representation subspaces at different positions. With a single attention head, averaging inhibits this. Multi-head attention is computed as:

$$MultiHead(Q, K, V) = Concat(head_1, ..., head_h)W^O$$
$$\text{where}\, head_i = Attention(QW_i^Q, KW_i^K, VW_i^V)$$

Where the projections are parameter matrices $W_i^Q \in \mathbb{R}^{d_{model} \times d_k}$, $W_i^K \in \mathbb{R}^{d_{model} \times d_k}$, $W_i^V \in \mathbb{R}^{d_{model} \times d_v}$ and $W^O \in \mathbb{R}^{hd_v \times d_{model}}$.

Position-wise Feed-Forward Networks In addition to the attention sublayer, each transformer layer includes two linear transformations with a ReLU activation function [6].

$$FFN(x) = max(0, xW_1 + b_1)W_2 + b_2$$

The feed-forward linear layers are followed by normalization layers. Thus as specified in this section, multi-head attention consists of:

1. Linearly projecting the queries, keys, and values with their corresponding linear layers. Namely, for each word piece embedding, BERT creates a query vector of dimension d_k, a key vector also of dimension d_k, and a value vector d_v.

2. Splitting the vectors for multi-head attention
3. Following the Attention equation to compute the attended output of each head
4. Concatenating multi-head attention outputs to recover the original shape

Dropout We lastly note that BERT applies [33] 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 $p_{drop} = 0.1$.

BERT output (`bert.BertModel.forward`)

As specified throughout this section, BERT consists of

1. An embedding layer that consists of token embedding `token_embedding` and positional embedding `pos_embedding`.
2. BERT encoder layers which are a stack of 12 `config.num_hidden_layers` `BertLayer`

After going through the respective layers the 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.

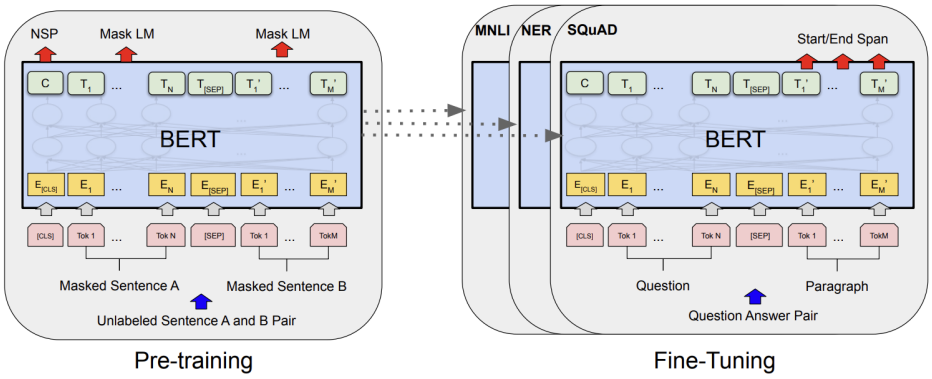

Figure 4: The original BERT model was trained on two unsupervised tasks, masked token prediction and next sentence prediction. Figure from [1].

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 corresponding 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, in the training procedure the "masked" tokens are not always replaced by the [MASK] token. Rather the training data generator chooses

15% of the token positions at random for prediction, then in 80% of these cases the token is replaced [MASK], in 10% of cases the token is replaced with a random token, and in another 10% of cases, the token will remain unchanged.

Next Sentence Prediction In order to allow BERT to understand the relationships between two sentences, BERT is further fine-tuned on the Next Sentence Prediction task. Specifically across training with these sentence pairs, the BERT model, 50% of the time is shown the actual next sentence, and 50% of the time it is shown a random sentence. The BERT model then predicts across these pairs, whether the second inputted sentence was 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.SelfAttention.attention`

The first function that you should implement is 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.

You can implement this attention function within `bert.SelfAttention.attention`.

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

After implementing the BERT multi-head self-attention layer, you can next implement the sections to realize the full BERT transformer layer. These functions can be filled in at `bert.BertLayer.add_norm`, `bert.BertLayer.forward`, and `bert.BertModel.embed`.

After finishing these steps, note that 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.

```
python3 sanity_check.py
```

4 Sentiment Analysis with the BERT

Having implemented a working minBERT model, you will now utilize pre-trained model weights and the outputted embeddings from your implemented BERT model to perform sentiment analysis on two datasets. In addition to running these pre-trained model weights to classify different sentences, 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[6] [2] consists of 11,855 single sentences from movie reviews extracted from movie reviews. The dataset was parsed with the Stanford parser[7] and includes a total of 215,154 unique phrases from those parse trees, each annotated by 3 human judges. Each phrase has a label of **negative**, **somewhat negative**, **neutral**, **somewhat positive**, or **positive**. Within this project, you will utilize BERT embeddings to predict these sentiment classification labels.

To summarize, 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**. We note that many of these reviews are longer than one sentence. Within this project, you will utilize BERT embeddings to predict these sentiment classifications.

To summarize, 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

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

1. Calls the BERT model to encode the sentences for their contextualized representations
2. Feeds in the encoded representations for the sentence classification task
3. Fine-tunes the Bert model on the downstream tasks (e.g. sentence classification)

Within this file, you are to implement the `BertSentimentClassifer`. You will implement this class to encode sentences using BERT and obtain the pooled representation of each sentence.[8] The class will then classify the sentence by applying on `dropout` the pooled output and then projecting it using a `linear` layer. Finally (already implemented), the model must be able to adjust its parameters depending on whether we are using pre-trained weights or are fine-tuning.

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

4.3 Adam Optimizer

In addition to implementing `BertSentimentClassifer`, you will further implement the `step()` function of the Adam Optimizer based on Decoupled Weight Decay Regularization [7] and Adam: A Method for Stochastic Optimization [8] in order to train a sentiment classifier.

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, 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: Implementing the `step()` function of the Adam Optimizer: `optimizer.step`

You should implement the `step()` function of the Adam Optimizer. Our reference uses the "efficient" method of computing the bias correction mentioned at the end of section 2 "Algorithm" of in Kigma and Ba [8] (and at the end of the algorithm above) in place of the intermediate $\hat{m}$ and $\hat{v}$ method. Similarly, the learning rate should be incorporated into the weight decay update. You can test your implementation by running:

```
python3 optimizer_test.py
```

4.4 Training minBERT for Sentiment Classification

For both the SST and the CFIMDB datasets, you should test 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 –option [pretrain/finetune] –epochs NUM_EPOCHS –lr LR –batch_size=BATCH_SIZE hidden_dropout_prob=RATE
```

You should utilize the `dev_out` and `test_out` flags to output your results to the following files for each dataset respectively (by running `classifier.py` these files should be automatically output).

```
pretrain/finetune-sst-dev-out.csv
pretrain/finetune-sst-test-out.csv
pretrain/finetune-cfimdb-dev-out.csv
pretrain/finetune-cfimdb-test-out.csv
```

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

Pretraining for SST: Dev Accuracy: **0.390** (0.007)
Pretraining for CFIMDB: Dev Accuracy: **0.780** (0.002)
Finetuning for SST: Dev Accuracy: **0.515** (0.004)
Finetuning for CFIMDB: Dev Accuracy: **0.966** (0.007)

You may *only* use the training set and our dev set to train, tune and evaluate your models. For this section, for grading, we will largely be looking at your code/implementation (as well as your accuracies on the test set).
Training for each dataset should take no more than 5 and 15 minutes (depending on your GPU).

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`
   - `pretrain/finetune-sst-dev-out.csv`
   - `pretrain/finetune-sst-test-out.csv`
   - `pretrain/finetune-cfimdb-dev-out.csv`
   - `pretrain/finetune-cfimdb-test-out.csv`
   - `setup.py`
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 GradeScope to **Default Final Project**.

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 assignment), 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, not just one!

Note that for this section, we will be testing you 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 the `test` dataset for each of these datasets within 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 prior part of this assignment, you are allowed to utilize other pre-existing NLP tools such as a POS tagger, dependency parser, Wordnet, coreference module, *etc...* that are not built on top of pre-trained contextual embeddings. You may further utilize other embeddings that you train yourself. However, for this assignment as an example, you may **not** for instance 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 400,000 question pairs with labels indicating whether particular instances are paraphrases of one another. We have provided you with a subset of this dataset with the following splits. For the Quora dataset, we have the following splits:

- `train` (141,506 examples)
- `dev` (20,215 examples)
- `test` (40,431 examples)

Given the binary labels of this dataset, as with the SST dataset, the metric that we utilize to test this dataset is accuracy.

SemEval STS Benchmark Dataset

The SemEval STS Benchmark dataset as briefly 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 some examples of these pairs see Section 1.

For the STS dataset, we have the following splits:

- `train` (6,041 examples)
- `dev` (864 examples)
- `test` (1,726 examples)

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

5.2 Code Overview

For this next part of the project, you are free to re-organize the functions inside each class, create new classes, and otherwise retrofit your code. However, we have provided you with function definitions that predict the sentiment scores of sentences, predict whether a sentence pair are paraphrases of each other, and finally predict the similarity of two input texts. We similarly provide you with ready-made code that loads in the training data of the STS, SemEval, and Quora datasets and evaluates your model on the provided `dev` sets provided. We note that it is up to you whether you wish to keep this formulation for training. Here we give a brief overview of this code.

- `multitask_classifier.MultitaskBERT`: A class that imports the weights of a pre-trained BERT model and can predict sentiment, paraphrases, and semantic textual similarity.
- `multitask_classifier.MultitaskBERT.forward`: The output of your BERT model. 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_paraphrase`: Predicts whether two sentences are paraphrases of each other based on BERT embeddings.
- `multitask_classifier.MultitaskBERT.predict_similarity`: Predicts the similarity of two sentences based on BERT embeddings.
- `multitask_classifier.MultitaskBERT.predict_sentiment`: Predicts the sentiment of a sentence based on BERT embeddings. As a baseline, you should call the new `forward()` function followed by a dropout and linear layer as in `classifier.py`.
- `multitask_classifier.train_multitask()`: A function for training your model. It is largely your choice how to train your model. As a baseline, you will find the original code from `classifier.py` to train your model on the SST sentiment dataset.
- `datasets.SentenceClassificationDataset`: A class for handling the SST sentiment dataset.
- `datasets.SentencePairDataset`: A class for handling the SemEval and Quora datasets.
- `evaluations.test_multitask()`: A function for testing your model. You should call this function after loading in an appropriate checkpoint of your model.

5.3 Possible Extensions

There are many possible extensions that can simultaneously improve your model's performance on the SST, Quora, and SemEval STS datasets. 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.

Additional Pretraining

Original Paper: How to Fine-Tune BERT for Text Classification? [9]
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 for the training datasets that we provided. For more details on BERT's pre-training, see [1].

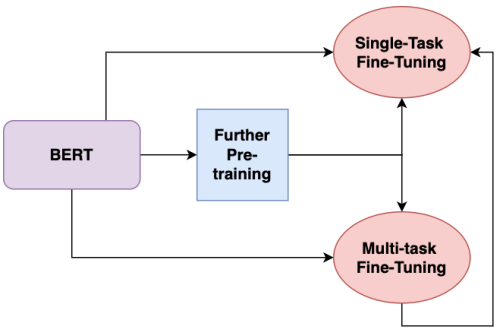

Figure 5: To improve your BERT model, there are several different paths that you can take including (1) additional pre-training on BERT's original objectives, (2) fine-tuning your model directly on a single task, (3) multi-task training your BERT model.

Multiple Negatives Ranking Loss Learning

Original Paper: Efficient Natural Language Response Suggestion for Smart Reply [10]

Another effective way of improving your embeddings would be to fine-tune your model with Multiple Negative Ranking Loss[9]. 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

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

where θ represents the word embeddings and neural network parameters used to calculate S, a scoring function. See sbert[10] and Henderson et al [10] for additional details.

Cosine-Similarity Fine-Tuning

Sentence-BERT: Sentence Embeddings using Siamese BERT-Networks [11]

Additional fine-tuning can further improve your BERT model on one of the pre-selected tasks. SemEval dataset the similarity between two embeddings is often computed using their cosine similarity. A way of potentially improving your embeddings would thus be to utilize CosineEmbeddingLoss[11] while fine-tuning on this 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.

[9] https://www.sbert.net/docs/package_reference/losses.html
[10] https://www.sbert.net/examples/training/nli/README.html#multiplenegativesrankingloss
[11] https://pytorch.org/docs/stable/generated/torch.nn.CosineEmbeddingLoss.html

Fine-Tuning with Regularized Optimization

Original paper: SMART: Robust and Efficient Fine-Tuning for Pre-trained Natural Language Models through Principled Regularized Optimization [12]

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 trust-region methods 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:

$$min_\theta = \mathcal{L}(\theta) + \lambda_s \mathcal{R}_s(\theta) \tag{2}$$

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

$$\mathcal{L}(\theta) = \frac{1}{n}\sum_{i=1}^{n} l(f(x_i;\theta), y_i), \tag{3}$$

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

$$\mathcal{R}_s(\theta) = \frac{1}{n}\sum_{i}^{n} \max_{||\tilde{x_i} - x_i||_p \leq \epsilon} l(f(\tilde{x_i};\theta), f(x_i;\theta)), \tag{4}$$

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.*,

$$l_s(P, Q) = \mathcal{D}_{KL}(P||Q) + \mathcal{D}_{KL}(Q||P) \tag{5}$$

Bergman Proximal Point Optimziation Jiang et al. also propose a class of Bregman point proximal point optimization[12] methods to solve Equation 2. 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:

$$\theta_{t+1} = argmin_\theta \mathcal{F}(\theta) + \mu \mathcal{D}_{Breg}(\theta, \theta_t) \tag{6}$$

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

$$\mathcal{D}_{Breg}(\theta, \theta_t) = l_s(f(\tilde{x_i};\theta), f(x_i;\theta_t)) \tag{7}$$

See https://github.com/namisan/mt-dnn and [12] for additional details.

[12] 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 [13]
Original paper: MTRec: Multi-Task Learning over BERT for News Recommendation [14]
Original paper: Gradient surgery for multi-task learning. [15]

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. [14], use multi-task learning adding each together each loss on the tasks of category classification and named entity recognition.

$$\mathcal{L}_{total} = \mathcal{L}_{task1} + \mathcal{L}_{task2} \tag{8}$$

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. [15] 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 $\mathbf{g_j}$:

$$\mathbf{g_i} = \mathbf{g_i} - \frac{\mathbf{g_i} \cdot \mathbf{g_j}}{||\mathbf{g_j}||^2} \cdot \mathbf{g_j} \tag{9}$$

Contrastive Learning

Original paper: Simple Contrastive Learning of Sentence Embeddings [16]

Gao et al [16] proposed a simple contrastive learning framework that works with both unlabeled and labeled data called SimCSE. 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. The following figure is an illustration of their model. You can utilize a similar approach to better your sentence embeddings across your different models.

Additional Datasets

Original paper: SMART: Robust and Efficient Fine-Tuning for Pre-trained Natural Language Models through Principled Regularized Optimization [12]

Original paper: Sentence-BERT: Sentence Embeddings using Siamese BERT-Networks [11]

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 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, reflect on the tradeoff between feature engineering and end-to-end learning, and comment on it in your report.

Other improvements

There are many other things besides training changes that you can do 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. Remember that we will be grading your experimental thoroughness, so do not neglect the hyperparameter search!

- **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 different distinctive layers for predicting whether sentences are paraphrases, their semantic similarity, and each sentence's sentiment. You could potentially share some layers amongst these different tasks to improve performance.
- **Model size and the number of layers.** With any model, you can try increasing the number of layers utilized to predict each of tasks
- **Optimization algorithms.** The baseline uses the Adam optimizer. PyTorch supports many other optimization algorithms. You should 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 the tasks that we are testing — there may be new ones that we haven't seen yet! In addition, there is lots of deep learning research on a large of amount of different tasks that may help improve your model.[13] These papers may contain interesting ideas that you can apply to build more robust and semantically rich embeddings.

[13] http://nlpprogress.com/

Submission Instructions

You will submit the Extensions part of this project on Gradescope. No particular files **BESIDES** the following are required:

```
predictions/sst-dev-out.csv
predictions/sst-test-out.csv
predictions/sts-dev-out.csv
predictions/sts-test-out.csv
predictions/para-dev-out.csv
predictions/para-test-out.csv
```

1. Verify that all files exist at these specified paths within your assignment directory that you developed.
2. Run `prepare_submit.py`
3. Upload your `cs224n_default_final_project_submission.zip` file to Gradescope to **Default Final Project Extension**.
4. Upload your project report to Gradescope to **Default Final Project [written]**.

6 Submitting to the Leaderboard

6.1 Overview

We are hosting two leaderboards on Gradescope, where you can compare your performance against that of your classmates, both for the `test` and `dev` datasets across each dataset provided.[14] The leaderboards can be found at the following links:

1. **Dev:** TBA
2. **Test:** TBA

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. You may use the `evaluations.test_multitask()` function in the `evaluations.py` script to generate a submission file of the correct format. 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. For the `test` and `dev` leaderboard, you must submit a prediction for every example.

[14] 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.

6.2 Submission Steps

Here are the concrete steps for submitting to the leaderboard:

1. Generate a submission file (*e.g.,* by running the `evaluations.test_mulitask(args)` function. This function by default should output files for both the `test` and `dev` for each of the datasets.
2. Use the URLs above to navigate to the leaderboard. **Make sure to choose the correct leaderboard for your split (DEV *vs.* TEST).**
3. Find the submit button in Gradescope, and choose the appropriate CSV file to upload.
4. Click upload and wait for your scores. The submission output will tell you the submission's accuracy/Pearson correlation on the `dev/test` datasets.

There should be useful error messages if anything goes wrong. If you get an error that you cannot 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 peoples' 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. Though we have run some preliminary tests to get some ballpark scores, it is impossible to say in advance what distribution of scores will be reasonably achievable for students in the provided timeframe. For similar reasons, there is no pre-defined rule for which of the extension proposed in Section 5 (or elsewhere) would 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 writeup and experimentation is important: we expect you to convincingly show that your techniques are effective and describe why they work (or the cases when they 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 final project in general also apply to the default final project. Here are some guidelines that are specifically relevant to the IID SQuAD track of the default final project:

1. You **may not** use a pre-existing implementation for the minBERT challenge as your starting point unless you wrote that implementation yourself.
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 CS224n team's code, or incorporate their code into your project.
4. As described in Section 5, it is an honor code violation to use the official SST, Quora and SemEval training and test data, and their `test` sets 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] Abien Fred Agarap. Deep learning using rectified linear units (relu). *arXiv preprint arXiv:1803.08375*, 2018.

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

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

[9] 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.

[10] 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.

[11] 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.

[12] 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.

[13] 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.

[14] 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.

[15] 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.

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