CS 224N: Default Final Project: Build GPT-2

Contents

1 Overview

In this assignment, you will build GPT-2, the precursor of OpenAI's ChatGPT language model. Specifically, you will implement some of the most important components of the architecture, load the official model weights from HuggingFace into your implementation, and explore its capabilities on a variety of downstream applications. This default final project has two parts.

In the first part, you will fill in the missing blocks of code to complete GPT-2. Similarly, you will implement part of the Adam optimizer (the algorithm that is used to train language models) by completing its step function. Finally, you will explore fine-tuning your implementation on two Sentiment Analysis datasets—i.e., predicting the sentiment (positive, negative, neutral) of various sentences—effectively turning your generative language model into a classification model.

In the second part, you will finetune your model on paraphrase detection, i.e. predicting if one sentence is a paraphrase of another. However, rather than finetuning your model for binary classification, you will instead formulate this as a cloze-style task, generating a word "yes" or "no" when asking if one sentence is a paraphrase of another. Next, you will finetune your model on the task of poem generation using a dataset of sonnets. This will introduce you to multi-token generation. You will evaluate your model's generated sonnets on a broader set of metrics and analyze some of the outputs qualitatively.

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 Project Philosophy

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 9.

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 Description of Tasks

In this section, we describe the tasks you will be asked to explore.

2.1 Sentiment Analysis

A basic task in language understandin is classifying the *polarity* of a text (*i.e.,* whether the expressed opinion in a text is positive, negative, or neutral). For example, 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] [1] 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: 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: 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: Negative

2.2 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 [2]. 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

2.3 Sonnet Generation

Sonnet Generation is a more open-ended task which we provide as an example of a realistic text generation task that you may want to use a language model for. Sonnets are a type of poem that fully a relatively strict structure, so they are amenable for small language models to learn. We hope it will be interesting to see what interesting new sonnets your model generates after you finetune it!

The example below is William Shakespeare's *Sonnet 5*. Note the alternating rhyme scheme and 14-line length.

[1] https://nlp.stanford.edu/sentiment/treebank.html
[2] https://nlp.stanford.edu/software/lex-parser.shtml
[3] https://www.quora.com/q/quoradata/First-Quora-Dataset-Release-Question-Pairs

Those hours that with gentle work did frame
The lovely gaze where every eye doth dwell,
Will play the tyrants to the very same
And that unfair which fairly doth excel:

For never-resting time leads summer on
To hideous winter and confounds him there;
Sap check'd with frost and lusty leaves quite gone,
Beauty o'ersnow'd and bareness every where:

Then, were not summer's distillation left,
A liquid prisoner pent in walls of glass,
Beauty's effect with beauty were bereft,
Nor it, nor no remembrance what it was:

But flowers distill'd, though they with winter meet,
Leese but their show; their substance still lives sweet.

3 GPT: Generative Pretrained Transformer

GPT (Generative Pretrained Transformer) models are built on the idea of using massive amounts of unlabeled text to pre-train a *decoder-only* Transformer architecture in an unsupervised manner, then fine-tuning on downstream tasks.

GPT-1, introduced by OpenAI in 2018 in [3], is a *decoder-only* Transformer that models text in a unidirectional manner, i.e., it predicts the next word by attending only to the leftwards context in each layer. It was among the first large-scale language models to show that purely unsupervised pre-training on vast amounts of text—followed by task-specific fine-tuning—can significantly improve performance on a variety of NLP tasks (like question answering or sentiment analysis). The key insight was that by first learning general linguistic and contextual patterns from raw text, the model gains "language understanding" that is broadly useful for different applications like sentiment classification or paraphrase detection.

GPT-2 [4] continued this trend in 2019, greatly scaling the size of GPT-1 and training it on an order of magnitude more data. Concretely, the differences are:

- **Scale:** GPT-2 has more parameters (up to 1.5 billion in its largest publicly released version) compared to GPT-1, which had around 117 million. This increase in parameter count and model depth/width allows GPT-2 to capture more complex language patterns.
- **Training Data:** GPT-2 was trained on a larger and more diverse dataset (on the order of billions of tokens) compared to GPT-1. This broader coverage further improves the model's ability to generalize to new topics and tasks.
- **Performance:** GPT-2 exhibits stronger zero-shot performance on tasks such as translation, question-answering, and summarization, meaning it can tackle tasks without explicit task-specific fine-tuning. GPT-1 mostly demonstrated improvements in a fine-tuning context.

GPT-1 and GPT-2 had a transformative effect on NLP research and practice. By showing that large-scale unsupervised pre-training can yield strong improvements on many downstream tasks, they helped popularize the "pre-train, then fine-tune" paradigm. GPT-2, in particular, showcased how increasing scale—both in

terms of data and model size—can lead to surprising leaps in model capability and language generation quality. This sparked further exploration into even larger models (e.g., GPT-3) and laid the groundwork for the rise of powerful language models used in numerous applications, ranging from dialogue agents to creative text generation. Case in point, this entire section was actually written by ChatGPT, with some minor edits by the authors.

4 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 at the following location:

```
https://github.com/stanfordnlp/cs224n_gpt
```

This repository contains the starter code and a minimalist implementation of the GPT models 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 `cs224n_gpt` directory and run the `git pull` command.

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

4.1 Code overview

The repository `cs224n_gpt/` contains the following files:

- `models/base_gpt.py`: A base GPT-2 implementation your implementation will inherit from.
- `models/gpt2.py`: This file contains your GPT-2 scaffolding. **There are several sections of this implementation that need to be completed.**
- `modules/attention.py`: This file contains the self-attention layer implementation. **There are several sections of this implementation that need to be completed.**
- `modules/gpt2_layer.py`: This file contains the basic GPT2 building block layer. **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.
- `sanity_check.py` A test for your completed GPT-2 model.
- `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`.
- `classifier.py`: A classifier pipeline for running sentiment analysis. There are several sections of this implementation that need to be completed.
- `evaluation.py`: A evaluations handling script for the second half of this project.

- `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.

4.2 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 `cs224n_gpt` and run `source setup.sh`
  - This creates a conda environment called `cs224n_dfp`.
  - For the first part of this assignment, you are only allowed to use libraries that are installed by `setup.sh`.
  - Do not change any of the existing command options (including defaults) or add any new required parameters
  - Don't forget to reactivate this environment 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 `cs224n_gpt` 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`.

5 Implementing GPT-2

We have provided you with several of the building blocks for implementing GPT-2. In this section, we will describe the GPT-2 model as well as the sections of it that you will implement.

5.1 Details of the GPT-2 Model

GPT-2 is a decoder-only transformer model that use a sequence of previous tokens to predict the next token. Here, we will walk you through the GPT-2 model as well as provide an overview of how it is trained.

Tokenization

The GPT-2 model uses byte pair encoding (BPE) tokenization. While its functionality is hidden by the Transformer library's Tokenizer class, interested students should view the huggingface tutorial to understand how words are decomposed into tokens. They can also play with this visualization by OpenAI, the BPE to tokenize sentences used in OpenAI's latest models.

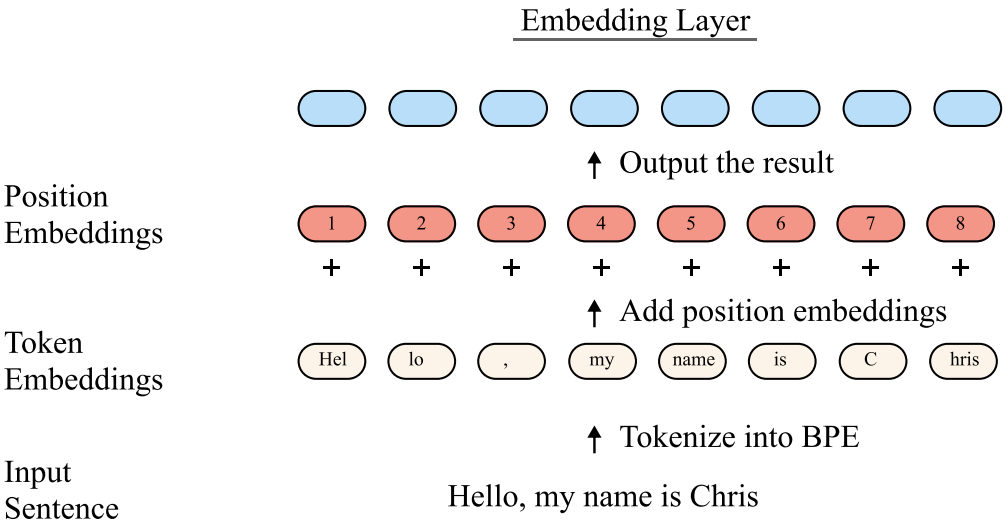

Figure 1: The embedding layer used in GPT-2. An input sentence (or sentences) is tokenized with byte pair encodings into tokens. Learnable positional embeddings are added to the tokens, and the result is output from the embedding layer.

Embedding Layer

After tokenizing and converting each token to ids, GPT-2 subsequently utilizes a trainable embedding layer across each token. The input embeddings that are used in later portions are the sum of the token embeddings and the position embeddings. Each embedding layer has a dimensionality of 768, and GPT-2 can process as many as 1024 tokens in a single example.

The learnable token embeddings map the individual input ids into vector representation for later use. More concretely, given some input token indices[4] $\boldsymbol{w}_1, \ldots, \boldsymbol{w}_k \in \mathbb{N}$, the embedding layer performs an embedding lookup to convert the indices into token embeddings $\boldsymbol{v}_1, \ldots, \boldsymbol{v}_k \in \mathbb{R}^D$.

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 1024

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

positions in GPT-2 for a given input. We call this last number the "context length"; it represents the maximum number of tokens the model can process at once for a single example.

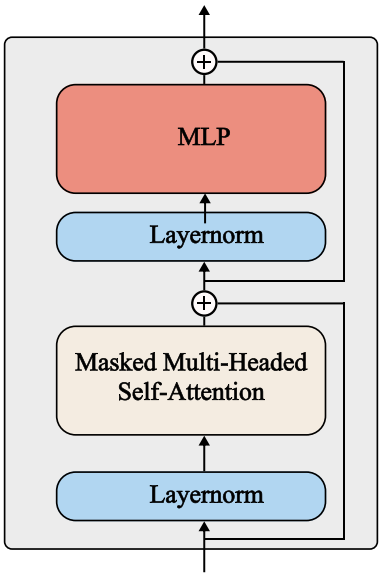

Figure 2: GPT-2 transformer layer. 12 of these layers are stacked, one after the other, to create the (small) version of GPT-2 that you will implement.

GPT-2 Transformer Layer (`gpt2_layer.GPT2Layer`)

The GPT-2 (small version) makes use of 12 Decoder Transformer layers. These layers were defined initially in the work Attention is All You Need [5]. The Transformer layers of the GPT-2 model is visualized in 2. For both models, it is composed of masked multi-head attention, skip connections, a multi-layer perceptron (MLP), and layernorm layers.

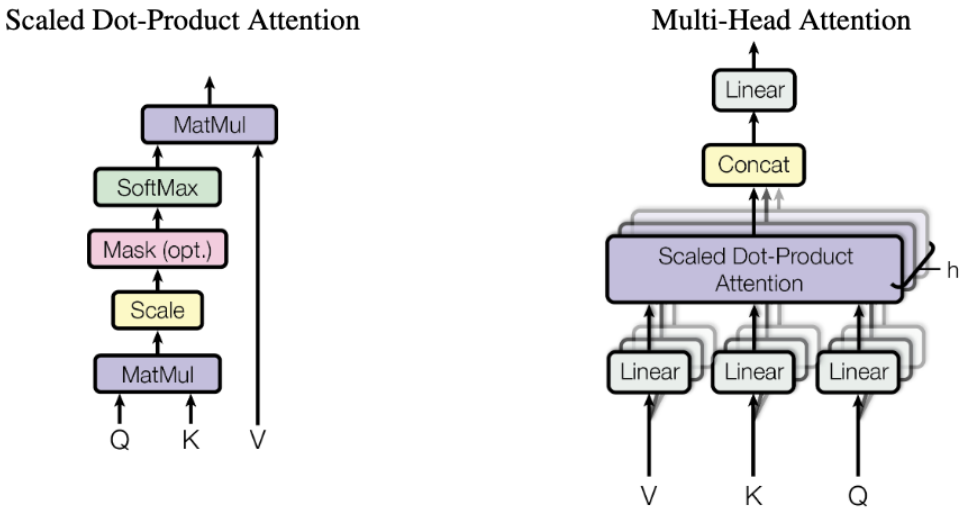

Figure 3: Scaled Dot-Product and Multi-Head Self-Attention. Figure from [5]

Multiheaded Self-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 scaled-dot product that consists of queries and keys of dimension d_k, and values of dimension d_v. GPT 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, GPT 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 $W^O \in \mathbb{R}^{hd_v \times d_{model}}$.

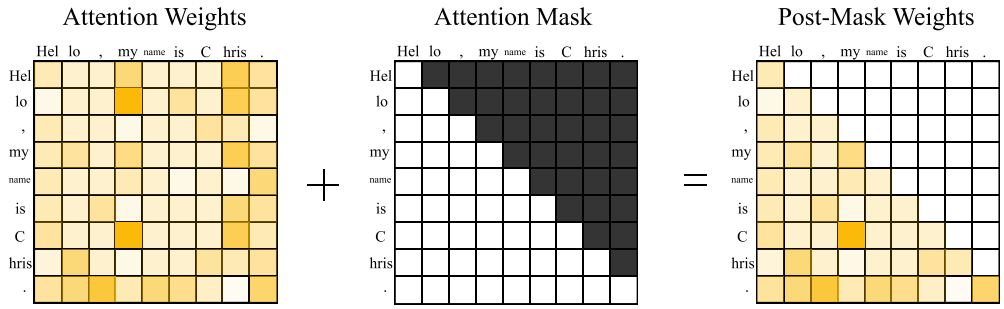

Figure 4: Example of causal masking in multi-headed self-attention **before** softmax normalization.

Masked Multiheaded Self-Attention (`attention.CausalSelfAttention.attention`) GPT-2 does not use multi-headed self-attention, but rather a masked (equivalently, casual) variant to prevent tokens from attending to future positions. This is to prevent tokens from learning to look up their label, i.e. the immediate next token, during training. Masked multi-head self-attention applies an upper-triangular mask (torch.triu) to the attention weights before the softmax to reduce the weighing on future positions to approximately zero.

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

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

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, GPT-2 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 GPT-2 applies dropout after each attention layer as well as after each MLP before the residual connection. GPT-2 also applies dropout to the sums of the embeddings and the positional encodings. GPT-2 uses a setting of $p_{drop} = 0.1$.

GPT-2 output

As specified throughout this section, GPT-2 consists of

1. An embedding layer that consists of token embedding `token_embedding` and positional embedding `pos_embedding`.
2. GPT-2 layers which are a stack of 12 `config.num_hidden_layers` `GPTLayer`

After going through the respective layers the outputs consist of:

1. `last_hidden_state`: the contextualized embedding for each token of the sentence from the last `GPTLayer` (i.e. the output of the GPT-2 transformer layers)
2. `last_token`: the last token embedding

Training GPT-2

GPT-2 [4] was trained on next-token prediction, predicting the next word given the previous context. For a sequence of tokens $x_1, x_2, ..., x_n$, the model was trained to maximize the log-likelihood:

$$\log P(x_1, x_2, ..., x_n) = \log \prod_{i=1}^{n} P(x_i | x_1, x_2, ..., x_{i-1}) = \sum_{i=1}^{n} \log P(x_i | x_1, x_2, ..., x_{i-1})$$

The goal was to use massive next-token prediction pre-training to teach the model general language understanding and give it the ability to generate coherent text. Later, the model could be fine-tuned to perform various tasks like sentiment classification, paraphrase detection, etc., taking advantage of its pre-trained language understanding to solve these tasks.

Unlike other language models of the 2019s, GPT-2 greatly increased the model scale (from the 117M parameters in GPT-1 to 1.5B parameters in GPT-2) and number of training examples. GPT-2 showed that autoregressive transformer models exhibited powerful scaling laws, with the performance scaling linear-log with respect to model size, dataset size, and training compute budget.

5.2 Code To Be Implemented: Masked Multi-head Self-Attention and the GPT-2 Layers

We have provided you with much of the code for a GPT-2 baseline model. Having gone over the basic structure of the GPT-2 Transformer model, we will now describe the sections that need to be implemented:

Masked Multi-head Self-Attention `attention.CausalSelfAttention.attention`

The first function that you should implement is the masked multi-head attention layer. 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 `attention.CausalSelfAttention.attention`.

GPT-2 Transformer Layers `modules.gpt2_layer.py`

After implementing the masked multi-head self-attention layer, you can next implement the sections to realize the full GPT transformer layers. For GPT-2, these functions can be filled in at `modules.gpt2_layer.add` and `modules.gpt2_layer.forward`.

GPT-2 Model `modules.gpt2.py`

Finally, you can implement the token and positional embedding function within the model class. These functions can be filled in at `models.gpt2.embed`.

After finishing these steps, note that we provide a sanity check function at sanity_check.py to test your implementation. It will compare your implementation to the official GPT-2 implementation on Huggingface. python3 sanity_check.py

5.3 Adam Optimizer

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

6 Sentiment Analysis with GPT-2

Having implemented a working GPT-2 model, you will now utilize pre-trained model weights 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.

6.1 Datasets

Stanford Sentiment Treebank (SST) dataset

The Stanford Sentiment Treebank[5] [1] consists of 11,855 single sentences from movie reviews extracted from movie reviews. The dataset was parsed with the Stanford parser[6] 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 GPT-2's last token embedding 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 GPT-2's last token 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)

6.2 Code To Be Implemented: Sentiment Classification with GPT-2

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

1. Calls the GPT model to encode the sentences and output the last token's representation
2. Feeds in the encoded representations for the sentence classification task
3. Fine-tunes the GPT-2 model on the downstream tasks (e.g. sentence classification)

Within this file, you are to implement the `GPT2SentimentClassifer`. You will implement this class to encode sentences using GPT 2 and obtain the last token's representation for each sentence.[7] The class will then classify the sentence by applying on `dropout` on this representation 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.

[5] https://nlp.stanford.edu/sentiment/treebank.html
[6] https://nlp.stanford.edu/software/lex-parser.shtml
[7] See the `forward` function in `models/gpt2.py` for how to access this representation

6.3 Training GPT-2 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 –fine-tune-mode [last-linear-layer/full-model] –batch_size BATCH_SIZE –lr LR hidden_dropout_prob=RATE –epochs=NUM_EPOCHS
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).

```
predictions/last-linear-layer-sst-dev-out.csv
predictions/last-linear-layer-sst-test-out.csv
predictions/full-model-sst-dev-out.csv
predictions/full-model-sst-test-out.csv
predictions/last-linear-layer-cfimdb-dev-out.csv
predictions/last-linear-layer-cfimdb-test-out.csv
predictions/full-model-cfimdb-dev-out.csv
predictions/full-model-cfimdb-test-out.csv
```

As a baseline, your implementation should have results similar to the following on the `dev` datasets:

Last Linear Layer for SST: Dev Accuracy: **0.462**
Full Model for SST: Dev Accuracy: **0.513**
Last Linear Layer for CFIMDB: Dev Accuracy: **0.861**
Full Model for CFIMDB: Dev Accuracy: **0.976**

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 GPT-2

You will submit the GPT-2 part of this project on Gradescope:

1. Verify that the following files exist at these specified paths within your assignment directory:
   - `modules/attention.py`
   - `modules/gpt2_layer.py`
   - `models/base_gpt.py`
   - `models/gpt2.py`
   - `classifier.py`
   - `optimizer.py`
   - `predictions/last-linear-layer-sst-dev-out.csv`
   - `predictions/last-linear-layer-sst-test-out.csv`
   - `predictions/full-model-sst-dev-out.csv`
   - `predictions/full-model-sst-test-out.csv`
   - `predictions/last-linear-layer-cfimdb-dev-out.csv`
   - `predictions/last-linear-layer-cfimdb-test-out.csv`
   - `predictions/full-model-cfimdb-dev-out.csv`
   - `predictions/full-model-cfimdb-test-out.csv`
2. Run `prepare_submit.py` to produce your `cs224n_default_final_project_submission.zip` file.
3. Upload this '.zip' file to GradeScope to **Default Final Project [Base]**.

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
```

7 Extensions and Improvements for Additional Downstream Tasks

While we have focused on implementing key aspects of GPT-2 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 model to improve its performance on paraphrase detection as well as sonnet generation. The goal of this latter part of the project is to explore how to improve the performance of your model from the mindset of a machine learning researcher.

For paraphrase detection, we will be testing you using the Quora dataset. You will find the `train`, `dev`, and the `test` dataset 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.**

For sonnet generation, we provide a small dataset of Shakespearian sonnets, with each document being a single sonnet. You will find the `train`, `dev`, and the `test` dataset 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.**

7.1 Cloze-Style Paraphrase Detection

Quora is a binary classification task, but GPT-2 outputs a distribution over next tokens. You can make GPT-2 perform classification by reformulating binary classification as a "close-style" task. Specifically, rather than fine-tuning GPT-2 to perform binary classification, you can instead formulate this task as generating "yes" or "no" to asking the model if one sentence is a paraphrase of the other. For example, the input:

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?"

can be reformulated as

Is "What is the step by step guide to invest in share market?" a paraphrase of "What is the step by step guide to invest in share market in india?"?

and then generating either "yes" or "no" as a response.

7.2 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, the metric that we utilize to test this dataset is accuracy.

Sonnet Dataset

The Sonnet Dataset consists of 154 sonnets written by Shakespeare. Each sonnet consists of 14 lines of text following the rhyme scheme ABAB CDCD EFEF GG, where lines that are assigned the same letter ID rhyme with each other. We have the following splits for this dataset:

- `train` (143 sonnets)
- `test` (12 sonnets)

During training, you will use cross-entropy loss on the training sonnets to fine-tune your GPT2 to generate Shakespeare. Then during evaluation, you will be given the first 3 lines for each of the 12 held-out (test) sonnets and asked to generate the rest. Your test-conditioned generated sonnets will be evaluated by how close the generated language is to Shakespeare's original sonnets using the chrF metric, which is a character-level n-gram comparison metric similar to BLEU [9].

7.3 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. We provide a brief overview of the starter code we provide for both paraphrase detection and sonnet generation tasks.

7.3.1 Paraphrase Detection

We have provided you with function definitions that predict whether a sentence pair are paraphrases of each other. We similarly provide you with ready-made code that loads in the training data of the Quora dataset and evaluates your model on the provided `dev` and `test` dataset splits. We recommend you modify—or entirely delete or replace—this starter code with your extensions. We provide a brief overview of the starter code below:

- `paraphrase_detection.ParaphraseGPT`: A class that imports the weights of a pre-trained GPT-2 model and can predict if one sentence is a paraphrase of another in a cloze-style task.
- `paraphrase_detection.ParaphraseGPT.forward`: The output of your paraphrase GPT-2 model. Note this is a cloze-style task, so you will be outputting a "yes" or a "no" to answer if one input is a paraphrase of another.
- `models/gpt2.GPT2Model.hidden_state_to_token`: Maps an output token to a probability distribution over all tokens in your vocabulary. Useful to transforming the output hidden state into a predicted next token.
- `paraphrase_detection.train`: A function for training your model. It is largely your choice how to train your model. A baseline implementation is provided, but you may want to improve upon this.
- `paraphrase_detection.test`: A function for evaluating your model on the validation (dev) and test dataset splits. It will save your predictions to disk, and you can upload these predictions to the public leaderboard on Gradescope. For the purposes of the autograder, we map a "yes" token to "8505" (i.e. the bpe token id for "yes") and a "no" token to "3919" (i.e. the bpe token id for "no").
- `datasets`: This file contains dataset functions and utilities for loading the quora dataset and creating a torch Dataset class.
- `evaluation`: This file contains functions and utilities for evaluating your model.

7.3.2 Sonnet Generation

We provide some starter code for generating your own sonnets. Critically, you will condition on the first 3 lines of a sonnet to generate the rest. Your model will be evaluated with a CHRF score: how closely your generated sonnets follow William Shakespeare's language distribution.

- `sonnet_generation.SonnetGPT`: A class that imports the weights of a pre-trained GPT-2 model and can generate new sonnets via autoregressive language modeling. You should implement the forward() method and consider improving the generate() method.
- `sonnet_generation.train`: A function for training your model. It is largely your choice how to train your model. A baseline implementation is provided, but you may want to improve upon this.
- `sonnet_generation.generate_submission_sonnets`: A function for generating the remaining lines of a sonnet conditioned on the first three lines. It will then save your generated sonnets to disk, and this is the file that you should submit to the public leaderboard on Gradescope.
- `datasets`: This file contains dataset functions and utilities for loading the sonnets dataset and creating a torch Dataset class.

7.4 Possible Extensions

There are many possible extensions that can improve your model's performance on the Quora and Sonnet 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 the performance of your model.

Perfecting Performance with PEFTs

Original paper: LoRA: Low-Rank Adaptation of Large Language Models [10]
Original paper: ReFT: Representation Finetuning for Language Models [11]

Parameter-efficient finetuning (PEFT) is a family of techniques for finetuning models that reduce the trainable parameter count relative to full finetuning. This is achieved by freezing most or all of the model (so gradients are not stored for most parameters), which reduces training time and the memory needed for storing the finetuned model (since only a weight update on the base model is needed). PEFT thus enables finetuning of larger models given a particular budget of GPU memory and disk capacity.

Several approaches to PEFT have been presented in the literature. An extension to your project might involve (1) loading in a larger GPT-2 model, and (2) implement a PEFT (or multiple PEFTs) for GPT-2, and (3) finetuning the larger model on any of the tasks we covered in this handout, or a novel task.
Some useful resources are the HuggingFace PEFT library (`https://github.com/huggingface/peft`) and the StanfordNLP `pyreft` library (`https://github.com/stanfordnlp/pyreft`).

Preference Optimization with Pair-wise Data

Original paper: Direct Preference Optimization: Your Language Model is Secretly a Reward Model [12]

Another direction to explore is Direct Preference Optimization (DPO), an approach to aligning language models with human preferences through pairwise comparison data. Unlike other RL-based alternative methods such as Reinforcement Learning from Human Feedback (RLHF), DPO provides a more stable training paradigm that operates in a supervised manner.

At its heart, DPO trains models using paired examples: for each input prompt, we have both a preferred output and a less preferred alternative. The model learns to assign higher probability to preferred outputs by optimizing the following objective:

$$\mathcal{L}_{\text{DPO}} = -\mathbb{E}_{(x,y_w,y_l)\sim D}\Big[\log \sigma\Big(\beta \log \frac{\pi_\theta(y_w|x)}{\pi_{\text{ref}}(y_w|x)} - \beta \log \frac{\pi_\theta(y_l|x)}{\pi_{\text{ref}}(y_l|x)}\Big)\Big]$$

Where:

- σ is the sigmoid function
- π_θ represents the model being optimized
- π_{ref} is a fixed reference model (typically the pretrained or fine-tuned base model)
- β is a temperature parameter controlling the strength of the preference signal
- (x, y_w, y_l) represents a prompt and its winning/losing output pair

Please refer to [12] for more details.

Constructing Pairwise Data. Unlike standard supervised fine-tuning datasets, the provided sonnet and paraphrase datasets contain only "positive" or desired samples (e.g., correct paraphrases, canonical sonnets). There are no explicitly labeled "negative" or less preferred responses. To use DPO, you would need to *construct* such losing samples:

- **Automatic Generation:** Use your current model or a simpler baseline model to produce alternate (and potentially incorrect) outputs, then label them as lower quality. These can be classified as lower quality either through human annotation or automated heuristics (e.g., analyzing structural elements like line count, rhyme scheme adherence, etc).
- **Heuristic Modifications:** Randomly shuffle, truncate, or corrupt the positive samples to produce obviously flawed outputs.

Once you have these pairs (winning vs. losing), you can train your GPT-2 model to increase the likelihood of the winning samples while decreasing that of the losing samples.
If you choose to pursue DPO, be prepared to carefully design your pairwise data collection/creation process. The negative examples should be realistic enough to provide meaningful training signal while being clearly distinguishable from high-quality outputs.

Accelerating Attention

Original paper: FlashAttention: Fast and Memory-Efficient Exact Attention with IO-Awareness [13]
Original paper: Longformer: The Long-Document Transformer [14]

Although the Transformers paper popularized scaled dot product attention, there exist dozens of alternate mechanisms [5] . For example, FlashAttention uses tiling and CUDA optimizations to compute attention in chunks, making it faster for large sequence lengths [13]. Similarly, Sliding Window Attention reduces time complexity from quadratic to linear (proposed in Longformer, transformers for long documents) [14].

One extension could involve experimenting with different types of attention mechanisms and analyzing the tradeoff between time and model utility. Some starting resources include: https://paperswithcode.com/methods/category/attention-mechanisms and https://huggingface.co/blog/tomaarsen/attention-sinks. Triton may also be your friend.

Questing for Quantization

GPT-specific paper: Quadapter: Adapter for GPT-2 Quantization [15]
GPT-specific paper: GPTQ: Accurate Post-Training Quantization for GPTs [16]

Quantization enables us to reduce memory usage and compute costs by representing model weights in lower precision (e.g., int8, bfloat16) instead of the high precision with which it was trained. For this extension, we can fine-tune and run models at different quantization levels and analyze how utility (paraphrase detection/sonnet quality) changes. Huggingface provides a guide (https://huggingface.co/docs/optimum/en/concept_guides/quantization), but you can also experiment with implementing your own methods.

One extension specific to GPT-2 could be quantization-aware fine-tuning, ensuring that fine-tuning with quantization does not lead to overfitting. To start, check out Quantization Adapters for GPT-2 [15].

Preconditioning for Proficiency (ft. optimizers)

Reference #1: Optimizing Neural Networks with Kronecker-factored Approximate Curvature [17]
Reference #2: Shampoo: Preconditioned Stochastic Tensor Optimization [18]
Reference #3: Scalable Second Order Optimization for Deep Learning [19]
Reference #4: SOAP: Improving and Stabilizing SHAMPOO USING ADAM [20]

Recently, second-order optimizers (i.e. optimizers that precondition the gradient with higher-order terms) have been gaining traction in train (or finetuning) LLMs. Specifically, these optimizers are being used at Google to train some of their largest models.

This extension focuses on using a pre-conditioning method to replace Adam to fine-tune your model. It should train **substantively** faster, using second-order information to speed up convergence. These methods operate by "preconditioning" the gradient; this is a fancy way to say you apply a transformation to the gradient vector to make it sensitive to curvature. If the normal update rule resembles:

$$\theta_{t+1} = \theta_t - \eta \nabla f(\theta_t)$$

then we the preconditioned gradient with preconditioner G would be:

$$\theta_{t+1} = \theta_t - \eta G \nabla f(\theta_t)$$

You may use K-FAC [17], Shampoo [18] or another preconditioning method entirely [20], there are many to choose among.

Fine-Tuning with Regularized Optimization

Original paper: SMART: Robust and Efficient Fine-Tuning for Pre-trained Natural Language Models through Principled Regularized Optimization [21]

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[8] 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 [21] for additional details.

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.
- **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.

[8] https://www.stat.cmu.edu/~ryantibs/convexopt/lectures/bregman.pdf

- **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.[9] These papers may contain interesting ideas that you can apply to build more robust and semantically rich embeddings.

[9] http://nlpprogress.com/

7.5 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 `*.cvs` files in a single zip file.
2. Verify that the generated `cs224n_default_final_project_submission.zip` includes your model predictions in a `predictions/*` directory as well as all necessary code in replicating your results.
3. Upload your `cs224n_default_final_project_submission.zip` file to the appropriate assignment on Gradescope.
4. Upload your project report to Gradescope to **Default Final Project [written]**.

8 Submitting to the Leaderboard

8.1 Overview

We are hosting four leaderboards on Gradescope, where you can compare your performance against that of your classmates. The first and second leaderboards are for the `test` and `dev` datasets for the Quora paraphrase detection dataset.[10] The third and fourth leaderboards are for the `test` and `dev` dataset for the Sonnet generation task.[11] The leaderboards can be found at the following links[12]:

1. **Paraphrase Dev:** Default Final Project [Paraphrase Dev]
2. **Paraphrase Test:** Default Final Project [Paraphrase Test]
3. **Sonnet Dev:** Default Final Project [Sonnet Dev]
4. **Sonnet Test:** Default Final Project [Sonnet Test]

For both paraphrase and sonnet tasks: 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 the exact same as submitting to the base autograder (implementation/sentiment predictions) on Gradescope. You will be using the same steps as 7.5. Although your `.zip` should be structured the same, the only file the leaderboard autograder will be looking at is the CSV file of predictions on the `dev/test` set (i.e `predictions/para-dev-output.csv` or `predictions/para-test-output.csv`).
At a high level, 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. The rows can be in any order. For the `test` and `dev` leaderboard, you must submit a prediction for every example.

For Sonnet Generation, the submission file should look like the following:

```
--Generated Sonnets--



0
Those lips that Love's own hand did make
Breathed forth the sound that said "I hate"
Some other generated text
...
```

You should have completions for all 12 sonnets in the dev/test tests (each).

[10] We will display the accuracy of your model on the Quora `test/dev` dataset.
[11] We will display the CHRF score for your generated Sonnets.
[12] An Ed post will be made soon containing the links to these leaderboards. This document will be updated accordingly.

8.2 Submission Steps

Here are the concrete steps for submitting to the leaderboard for the Paraphrase Detection task (**Sonnet Generation is identical**):

1. Generate the prediction `.csv` files using your ParaphraseGPT model. Use the steps in 7.5 (i.e running `prepare_submit.py`) to generate your `.zip` and ensure that the relevant `.csv` is found in `predictions/`.
2. Use the URLs above to navigate to the leaderboard. **Make sure to choose the correct leaderboard for your split (DEV *vs.* TEST)**, and remember that you only have **three submissions** for **TEST**.
3. Find the submit button in Gradescope, and choose your submission `.zip` to upload.
4. Click upload and wait for your scores. The submission output will tell you the submission's accuracy correlation on the `dev/test` datasets. Your placement on the leaderboard is according to your best submission, not necessarily your most recent one. *Please choose an appropriate Leaderboard Name.*

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.

9 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 on paraphrase detection or perplexity score on sonnet generation 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 7 (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.

10 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 default final project:

1. You **may not** use a pre-existing GPT-2 implementation for the sentiment analysis task as your starting point unless you wrote that implementation yourself.
2. 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.
3. As described in Section 7, it is an honor code violation to use the official Quora and Sonnets training and test data, and their `test` sets in any way.
4. Do not share your code publicly (e.g., in a public GitHub repo) until after the class has finished.

References

[1] 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.

[2] 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.

[3] Alec Radford. Improving language understanding by generative pre-training. 2018.

[4] Alec Radford, Jeffrey Wu, Rewon Child, David Luan, Dario Amodei, Ilya Sutskever, et al. Language models are unsupervised multitask learners. *OpenAI blog*, 1(8):9, 2019.

[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] Maja Popović. chrF: character n-gram F-score for automatic MT evaluation. In Ondřej Bojar, Rajan Chatterjee, Christian Federmann, Barry Haddow, Chris Hokamp, Matthias Huck, Varvara Logacheva, and Pavel Pecina, editors, *Proceedings of the Tenth Workshop on Statistical Machine Translation*, pages 392–395, Lisbon, Portugal, September 2015. Association for Computational Linguistics.

[10] 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, 2021.

[11] Zhengxuan Wu, Aryaman Arora, Zheng Wang, Atticus Geiger, Dan Jurafsky, Christopher D. Manning, and Christopher Potts. Reft: Representation finetuning for language models, 2024.

[12] Rafael Rafailov, Archit Sharma, Eric Mitchell, Stefano Ermon, Christopher D. Manning, and Chelsea Finn. Direct preference optimization: Your language model is secretly a reward model, 2024.

[13] Tri Dao, Daniel Y. Fu, Stefano Ermon, Atri Rudra, and Christopher Ré. FlashAttention: Fast and memory-efficient exact attention with IO-awareness. In *Advances in Neural Information Processing Systems (NeurIPS)*, 2022.

[14] Iz Beltagy, Matthew E. Peters, and Arman Cohan. Longformer: The long-document transformer. *arXiv:2004.05150*, 2020.

[15] Minseop Park, Jaeseong You, Markus Nagel, and Simyung Chang. Quadapter: Adapter for GPT-2 quantization. In Yoav Goldberg, Zornitsa Kozareva, and Yue Zhang, editors, *Findings of the Association for Computational Linguistics: EMNLP 2022*, pages 2510–2517, Abu Dhabi, United Arab Emirates, December 2022. Association for Computational Linguistics.

[16] Elias Frantar, Saleh Ashkboos, Torsten Hoefler, and Dan Alistarh. OPTQ: Accurate quantization for generative pre-trained transformers. In *The Eleventh International Conference on Learning Representations*, 2023.

[17] James Martens and Roger Grosse. Optimizing neural networks with kronecker-factored approximate curvature. In *International conference on machine learning*, pages 2408–2417. PMLR, 2015.

[18] Vineet Gupta, Tomer Koren, and Yoram Singer. Shampoo: Preconditioned stochastic tensor optimization. In *International Conference on Machine Learning*, pages 1842–1850. PMLR, 2018.

[19] Rohan Anil, Vineet Gupta, Tomer Koren, Kevin Regan, and Yoram Singer. Scalable second order optimization for deep learning. *arXiv preprint arXiv:2002.09018*, 2020.

[20] Nikhil Vyas, Depen Morwani, Rosie Zhao, Itai Shapira, David Brandfonbrener, Lucas Janson, and Sham Kakade. Soap: Improving and stabilizing shampoo using adam. *arXiv preprint arXiv:2409.11321*, 2024.

[21] 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.