Carnegie Mellon University, Fall 2023
11-667: Large Language Models
Assignment 2, Due Tuesday, October 24th at 2 PM
Creators: Yiming Zhang and Clement Fung

In this homework, you will implement and train a decoder-only transformer model from scratch. You will use your trained language model for text generation and sentiment analysis. You will gain an understanding of implementation details and training methods for transformers.

Instructions

This homework will be graded in two parts. You will be asked qualitative questions about your implementation and are expected to share insights after exploring different implementation trade-offs; these questions do not require code submissions and are marked as (*Written*). You must fill out the answers in this Latex template, and include it in your `.zip` submission.

Your code implementations will also be graded with unit tests (all the tests are provided to you); these questions are marked as (*Coding*). You will be expected to submit your code, which will be checked for plagiarism.

Prepare a submission `[andrew-id].zip` file with the following files:

1. Code files: `model.py`, `train.py`, `generate.py`, `classify.py`
2. Final model checkpoint `model.pt` (See question 2.6)
3. PDF of your written answers

0 Setting up the Environment [0 points]

To start, follow the instructions in starter code to setup the AWS instance and the development environment. The test case in `test_env.py` should be passing.

1 Implementing a Decoder-only Transformer Model [35 points]

You will first implement a decoder-only transformer model. An outline of the code is provided for you in `model.py`. This outline contains all the class and function declarations that are expected for the submission. **Do not modify the classes, functions, or their arguments. Do not import new Python dependencies. This may break the automatic code test pipeline and result in failed unit tests.** You should aim to have an efficient implementation for the model to train in a reasonable amount of time. This means calling PyTorch functions whenever possible, and also means that you should not write matrix operations using a for loop. That said, you may not use layers or functions (e.g., `torch.nn.TransformerDecoder`) that make implementation trivial. If you are unsure whether using something is acceptable, ask course staff.

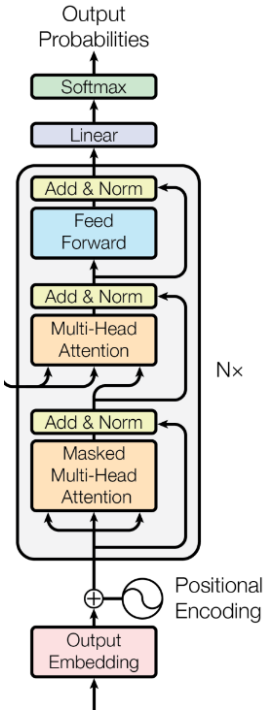

Figure 1: Transformer decoder

Recall the Transformer Decoder from Lecture 2, shown in Figure 1. There are four classes within the transformer that you are expected to implement:

1. `MultiHeadAttention` - the "Masked Multi-head Attention" module.
2. `FeedForward` - the "Feed Forward" module.
3. `DecoderBlock` - a single decoder block, as described in "The Decoder Step-by-Step" in Lecture 2. Note that since we are implementing the decoder only, you do not need to implement the Encoder-Decoder Multi-Head Attention or the second "Add & Norm" operation.
4. `DecoderLM` - the full decoder model: the embedding step, multiple decoder blocks, and the final output logits.

[**Question 1.1**] (*Written, 5 points*): Press and Wolf (2017) propose a weight tying technique for projecting hidden states of a language model to token logits. Read this paper, and in a few sentences, explain what weight tying does.

Write your answer here:

[**Question 1.2**] (*Written, 5 points*): Let d be the hidden size of the model, v be the vocab size, b be the batch size, and s be the sequence length. Suppose you have hidden states $h \in \mathbb{R}^{b \times s \times d}$ and token embeddings $E \in \mathbb{R}^{v \times d}$ stored in PyTorch tensors. Write one line of Python code (potentially calling functions in PyTorch) that computes the token logits using weight tying.

Write your answer here:

[**Question 1.3**] (*Coding, 25 points*): Complete `model.py`, implementing all of the classes above. All the unit tests are provided in the test script `test_model.py`. Your implementation will be awarded points for each of the five unit tests that pass.

2 Training the Transformer [85 points]

Now that you have implemented the transformer, it is time to train the model! For ease of implementation and testing, we will provide the tokenized input for you. We will train on a subset of the C4 corpus[1], which is itself a subset of the Common Crawl web corpus[2]. This dataset is downloaded automatically as a part of the training script, and there is no need for you to access it manually.

An outline of the code is provided for you in `train.py`. Keep in mind the various hyperparameters that are relevant for training: batch size, learning rate (and its scheduler), gradient accumulation, etc. These hyperparameters are read from a configuration file. We have provided sample configuration files for you for adjusting the hyperparameters. There are five functions that you are expected to implement:

1. `train` - the main training loop.

[1] `https://huggingface.co/datasets/allenai/c4`

[2] `https://commoncrawl.org/`

2. `random_batch_sampler` - a data sampling function used in training that yields randomly shuffled batches of the data.

3. `sequential_batch_sampler` - a data sampling function used in validation that yields a sequential pass through the data.

4. `cosine_lr_schedule` - learning rate scheduler with Cosine annealing (see question 2.2).

5. `compute_language_modeling_loss` - the loss function used for training and evaluating the model.

[**Question 2.1**] (*Coding, 20 points*): Complete `train.py`, implementing the above functions. All the unit tests are provided in the test script `test_train.py`. Your implementation will be awarded points for each of the five unit tests that pass.

[**Question 2.2**] (*Written, 10 points*): Cosine annealing with warmup is a commonly used dynamic learning rate strategy in training neural nets. It has two phases, in the warmup phase where $t \in [0, a)$, the learning rate increases linearly from 0 to lr_{max}. In the annealing phase where $t \in [a, b)$, the learning rate decays from lr_{max} to lr_{min} following a half-cosine curve. When $t \geq b$, the learning rate stays at lr_{min}. Figure 2 shows an example of this schedule.

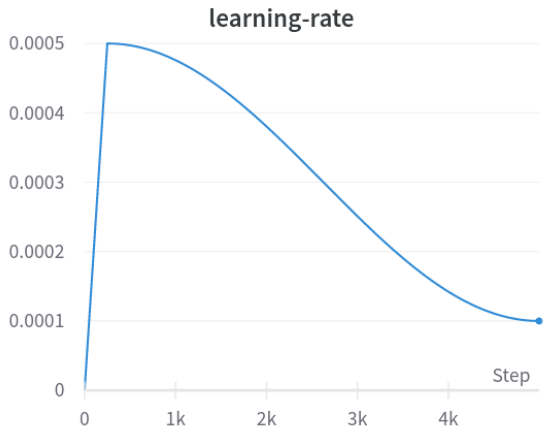

Figure 2: Example Cosine Schedule

Using the symbols provided, write down two expressions, one for the learning rate during warmup and one for the learning rate during annealing.

Write your expression for cosine annealing here:

Why would one want to use cosine annealing? What are some advantages of cosine annealing over a constant learning rate? Write your answer here:

[**Question 2.3**] (*Written, 10 points*): What is the validation loss after training with the configuration provided in `GPT-tiny.yaml`? For approximately how many training steps should the model be trained to achieve optimal performance? Report the training loss curve (use a screenshot from weights & biases: `https://wandb.ai/site`).

Write the validation loss for `GPT-tiny` here:

7.66062

Write the number of training steps needed for `GPT-tiny` here:

2000

Include your training curve for `GPT-tiny` below:

Figure 3: Broken training cases: case-1 and case-2

[**Question 2.4**]: (*Written, 5 points*) Figure 3 shows two mis-configured training runs, compared against their expected case with the proper configuration. For case-1 and case-2, in only a couple sentences each, can you explain what is wrong with the training setup? The loss function and transformer implementations are correct.

Write your answer here for case-1:

Write your answer here for case-2:

[**Question 2.5**] (*Written, 20 points*): Next, you will perform hyperparameter tuning. When optimizing neural networks, a common way to measure the total computation needed is with floating-point operations (FLOPs). E.g., a single multiply-add of floats is counted as a FLOP. Assume that you have a compute budget of 1e+15 FLOPs for training. Experiment with your model and training hyperparameters to find the best configuration when training with at most 1e+15 FLOPs (the code for computing the number of FLOPs used to train a model is provided in `train.py`). Note that the FLOP limit is intentionally low: it should not take more than 10 minutes per hyperparameter test.

As long as you stay within the FLOP budget, you are free to modify any of the following hyperparameters, all of which affect the number of FLOPs used: model hyperparameters such as `n_embd`, `n_head`, `n_positions` and `n_layer`; and training hyperparameters such as `batch_size`, `seq_len`, `grad_accumulation_steps`, and `num_training_steps`. Describe your experiment: report results for **at most five** hyperparameter results: describe what hyperparameter values are modified, and their resulting validation perplexity (PPL). For your best performing setting, report the final configuration (YAML file) for this setting.

Report your experiment procedure and results here:

Paste your final YAML configuration here:

[**Question 2.6**] (*Coding, 20 points*): Using the model configuration reported above, train your final model. For training your final model, you may train 100x more FLOPs (i.e. up to 1e+17 FLOPS), but do not change your model hyperparameters. Your final model should be able to achieve a PPL under 50 on the validation set. Please include your final model checkpoint `model.pt` in your submission. We will verify your model with an offline correctness test, which will measure the perplexity on a test dataset (not provided). Submissions with PPL under 50 are guaranteed to get full score, and we will relax this requirement if needed.
Report your final validation loss and PPL:

3 Using the Language Model for Downstream Tasks [40 points]

Text Generation

Next, you will use your trained LLM to generate text. An outline of the code for text generation is provided for you in `generate.py`. To generate text, prompts must be provided to the LLM; we have provided an input file `prefixs.json` with sample inputs.

There are two functions that you are expected to implement:

1. `generate` - given a `DecoderLM` and a list of prompts, generate tokens.
2. `softmax_with_temperature` - convert a given a set of logits into probabilities with the softmax function, using temperature.

[**Question 3.1**] (*Coding, 10 points*): Complete `generate.py`, implementing the above functions. One unit test are provided in the test script `test_generate.py`. Your implementation will be awarded points for passing the unit test and correctly implementing `generate`.

[**Question 3.2**] (*Written, 5 points*): Using the three prefixes provided in `prefixs.json`, report generations from the prompts. Repeat this process, experimenting with different temperature values and prompts. What happens to the generations if the temperature is near zero, or near one? Do you notice anything interesting with the generated text?

Report your generations here:

Write your answer here:

[**Question 3.3**] (*Written, 5 points*): Beyond adjusting hyperparameters for sampling, how can the quality of generation be improved? Provide at least two suggestions for how generation quality can be improved and reasons why you expected an improvement.

Write your answer here:

Sentiment Analysis

Finally, your trained LLM can be used to perform sentiment analysis. An outline of the code for sentiment analysis is provided for in `classify.py`. To evaluate sentiment analysis, use the Yelp polarity dataset: a labelled dataset of positive/negative Yelp reviews. During inference, you will place the text into a Yelp text template and determine if the token "good" or "bad" is more likely.
There are two functions that you are expected to implement:

1. `score` - score the next token given a list of prefix strings.
2. `classify_binary_sentiment` - given a `DecoderLM` and a set of input texts, predict if the input texts are "positive" reviews.

[**Question 3.4**] (*Coding, 10 points*): Complete `classify.py`, implementing the above functions. One unit test are provided in the test script `test_classify.py`. Your implementation will be awarded points for passing the unit test and correctly implementing `score`.

[**Question 3.5**] (*Written, 5 points*): Report the classification accuracy on the Yelp polarity dataset *without* calibration. Is the performance higher than random (50%) or majority class (53.1%)? Explain why this is the case.
Classification accuracy without calibration:

Write your answer here:

[**Question 3.6**] (*Written, 5 points*): When applying language models directly to classification, the model can often be miscalibrated. An example of a model biased to the positive label is shown in Figure 4. Assuming there are roughly the same number of positive and negative examples in the dataset, how can you find a better decision boundary for a miscalibrated model than the default value (0.5) without access to labels or a separate development set? In other words, your calibration strategy should depend exclusively on the predicted **p**(Positive) for all test instances.

Under this new decision boundary you came up with, report the classification accuracy on the Yelp polarity dataset. Other than training a bigger model or training for longer, what are some other potential ways to improve the classification performance?

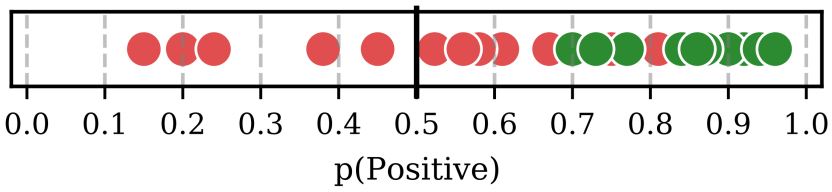

Figure 4: A miscalibrated classifier that is biased to the positive class, taken from Zhao et al. (2021). Negative groundtruth examples are marked with ●, and positive groundtruth examples are marked with ●. Note: This figure is for illustration, and the plotted distribution will likely be different from the actual outputs of your model.

Your proposed method of computing the calibrated decision boundary:

Classification accuracy with calibration:

Other ways to improve classification performance:

4 Optional: Give us Feedback

Was this homework enjoyable? Was it too easy or too hard? Do you have any suggestions for making the homework run more smoothly? Giving us feedback is completely optional and will not factor into your grade.