Spring 2025
Introduction to Deep Learning 18786

Homework 5: Transformers

Due: Tuesday, April 8, 2025 11:59 pm

Submitting your work: For all problems on this homework we will be using **Gradescope.** To receive full credit, you need to submit both the PDF solution and a zip file containing your code solution. Please be super clear, neat, and complete when you write your solutions. **If the TAs cannot follow your work, you will not receive full credit for that problem.**

Collaboration Policy: Homework will be done individually: each student must hand in their own answers. It is acceptable for students to collaborate in figuring out answers and helping each other solve the problems. We will be assuming that, as participants in a graduate course, you will be taking the responsibility to make sure you personally understand the solution to any work arising from such collaboration. You must also indicate on each homework with whom you collaborated.

1. Position Embeddings Exploration (6 points)

Position embeddings are an important component of the Transformer architecture, allowing the model to differentiate between tokens based on their position in the sequence. In this question, we'll explore the need for positional embeddings in Transformers and how they can be designed.

Recall that the crucial components of the Transformer architecture are the self-attention layer and the feed-forward neural network layer. Given an input tensor $\mathbf{X} \in \mathbb{R}^{T \times d}$, where T is the sequence length and d is the hidden dimension, the self-attention layer computes the following:

$$\mathbf{Q} = \mathbf{X}\mathbf{W}_Q, \quad \mathbf{K} = \mathbf{X}\mathbf{W}_K, \quad \mathbf{V} = \mathbf{X}\mathbf{W}_V$$
$$\mathbf{H} = \text{softmax}\left(\frac{\mathbf{Q}\mathbf{K}^\top}{\sqrt{d}}\right)\mathbf{V}$$

where $\mathbf{W}_Q, \mathbf{W}_K, \mathbf{W}_V \in \mathbb{R}^{d \times d}$ are weight matrices, and $\mathbf{H} \in \mathbb{R}^{T \times d}$ is the output.

Next, the feed-forward layer applies the following transformation:

$$\mathbf{Z} = \text{ReLU}(\mathbf{H}\mathbf{W}_1 + \mathbf{1} \cdot \mathbf{b}_1)\mathbf{W}_2 + \mathbf{1} \cdot \mathbf{b}_2$$

where $\mathbf{W}_1, \mathbf{W}_2 \in \mathbb{R}^{d \times d}$ and $\mathbf{b}_1, \mathbf{b}_2 \in \mathbb{R}^{1 \times d}$ are weights and biases; $\mathbf{1} \in \mathbb{R}^{T \times 1}$ is a vector of ones[*]; and $\mathbf{Z} \in \mathbb{R}^{T \times d}$ is the final output.

(Note that we have omitted some details of the Transformer architecture for simplicity.)

(a) (4 points) **Permuting the input.**

i. (3 points) Suppose we permute the input sequence $\mathbf{X}$ such that the tokens are shuffled randomly. This can be represented as multiplication by a permutation matrix $\mathbf{P} \in \mathbb{R}^{T \times T}$, i.e. $\mathbf{X}_{\text{perm}} = \mathbf{P}\mathbf{X}$. (See Wikipedia for a recap on permutation matrices.)

Show that the output $\mathbf{Z}_{\text{perm}}$ for the permuted input $\mathbf{X}_{\text{perm}}$ will be $\mathbf{Z}_{\text{perm}} = \mathbf{P}\mathbf{Z}$.

You are given that for any permutation matrix $\mathbf{P}$ and any matrix $\mathbf{A}$, the following hold: $\text{softmax}(\mathbf{P}\mathbf{A}\mathbf{P}^\top) = \mathbf{P}\,\text{softmax}(\mathbf{A})\,\mathbf{P}^\top$ and $\text{ReLU}(\mathbf{P}\mathbf{A}) = \mathbf{P}\,\text{ReLU}(\mathbf{A})$.

ii. (1 point) Think about the implications of the result you derived in part i. **Explain** why this property of the Transformer model could be problematic when processing text.

(b) (2 points) **Position embeddings** are vectors that encode the position of each token in the sequence. They are added to the input word embeddings before feeding them into the Transformer.

One approach is to generate position embedding using a fixed function of the position and the dimension of the embedding. If the input word embeddings are $\mathbf{X} \in \mathbb{R}^{T \times d}$, the position embeddings $\Phi \in \mathbb{R}^{T \times d}$ are generated as follows:

$$\Phi_{(t,2i)} = \sin\left(t/10000^{2i/d}\right)$$
$$\Phi_{(t,2i+1)} = \cos\left(t/10000^{2i/d}\right)$$

where $t \in \{0, 1, \ldots T-1\}$ and $i \in \{0, 1, \ldots d/2 - 1\}$[†].

Specifically, the position embeddings are added to the input word embeddings:

$$\mathbf{X}_{\text{pos}} = \mathbf{X} + \Phi$$

i. (1 point) Do you think the position embeddings will help the issue you identified in part (a)? If yes, explain how and if not, explain why not.

[*]Outer product with $\mathbf{1}$ represents broadcasting operation and makes feed forward network notations mathematically sound.
[†]Here d is assumed even which is typically the case for most models.

ii. (1 point) Can the position embeddings for two different tokens in the input sequence be the same? If yes, provide an example. If not, explain why not.

2. Pretrained Transformer models and knowledge access (35 points)

You'll train a Transformer to perform a task that involves accessing knowledge about the world — knowledge which isn't provided via the task's training data (at least if you want to generalize outside the training set). You'll find that it more or less fails entirely at the task. You'll then learn how to pretrain that Transformer on Wikipedia text that contains world knowledge, and find that finetuning that Transformer on the same knowledge-intensive task enables the model to access some of the knowledge learned at pretraining time. You'll find that this enables models to perform considerably above chance on a held out development set.

The code you're provided with is a fork of Andrej Karpathy's `minGPT`. It's nicer than most research code in that it's relatively simple and transparent. The "GPT" in `minGPT` refers to the Transformer language model of OpenAI, originally described in this paper [1].

You will want to develop on your machine locally, then optionally run training on Colab with GPU. You may also choose to upload your homework folder to google drive and modify the files in google colab by mounting your drive.

You can use the same conda environment from previous assignments for local development, and the same process for training on a GPU (You may also run with CPU, and it should be able to finish in a reasonable amount of time).

You may need around 3 hours for training, so budget your time accordingly!

Your work with this codebase is as follows:

(a) (0 points) **Read through `NameDataset` in `src/dataset.py`, our dataset for reading name-birthplace pairs.**
The task we'll be working on with our pretrained models is attempting to access the birth place of a notable person, as written in their Wikipedia page. We'll think of this as a particularly simple form of question answering:

Q: Where was [person] born?
A: [place]

From now on, you'll be working with the `src/` folder. The code in `mingpt-demo/` won't be changed or evaluated for this assignment. In `dataset.py`, you'll find the the class `NameDataset`, which reads a TSV (tab-separated values) file of name/place pairs and produces examples of the above form that we can feed to our Transformer model.

To get a sense of the examples we'll be working with, if you run the following code, it'll load your `NameDataset` on the training set `birth_places_train.tsv` and print out a few examples.

```
python src/dataset.py namedata
```

Note that you do not have to write any code or submit written answers for this part.

(b) (4 points) **Complete Self-Attention Implementation.**
Self-attention is the primary module used in the transformer models. Causal self attention, also known as autoregressive self-attention, masks out part of the attention mask that corresponds to the future of the current token in this sequence, so that the current token can only attend to its past and itself. The provided self-attention forward code and mask initialization in `src/attention.py` is incomplete, please finish the implementation based on the provided skeleton and comments.

(c) (0 points) **Implement finetuning (without pretraining).**
Take a look at `run.py`. It has some skeleton code specifying flags you'll eventually need to handle as command line arguments. In particular, you might want to *pretrain*, *finetune*, or *evaluate* a model with this code. For now, we'll focus on the finetuning function, in the case without pretraining.

Taking inspiration from the training code in the `play_char.ipynb` file, write code to finetune a Transformer model on the name/birthplace dataset, via examples from the `NameDataset` class. For now, implement the case without pretraining (i.e. create a model from scratch and train it on the birthplace prediction task from part (b)). You'll have to modify two sections, marked `[part c]` in the code: one to initialize the model, and one to finetune it. Note that you only need to initialize the model in the case labeled "vanilla" for now (later in section (g), we will explore a model variant). Use the hyperparameters for the `Trainer` specified in the `run.py` code.

Also take a look at the *evaluation* code which has been implemented for you. It samples predictions from the trained model and calls `evaluate_places()` to get the total percentage of correct place predictions. You will run this code in part (d) to evaluate your trained models.

This is an intermediate step for later portions, including Part d, which contains commands you can run to check your implementation. No written answer is required for this part.

(d) (4 points) **Make predictions (without pretraining).**
Train your model on `birth_places_train.tsv`, and evaluate on `birth_dev.tsv`. Specifically, you should now be able to run the following three commands:

```
# Train on the names dataset
python src/run.py finetune vanilla wiki.txt \
       --writing_params_path vanilla.model.params \
       --finetune_corpus_path birth_places_train.tsv

# Evaluate on the dev set, writing out predictions
python src/run.py evaluate vanilla wiki.txt \
       --reading_params_path vanilla.model.params \
       --eval_corpus_path birth_dev.tsv \
       --outputs_path vanilla.nopretrain.dev.predictions

# Evaluate on the test set, writing out predictions
python src/run.py evaluate vanilla wiki.txt \
       --reading_params_path vanilla.model.params \
       --eval_corpus_path birth_test_inputs.tsv \
       --outputs_path vanilla.nopretrain.test.predictions
```

Report your model's accuracy on the dev set (as printed by the second command above). We also have Tensorboard logging in assignment for debugging. It can be launched using `tensorboard --logdir expt/`. Don't be surprised if it is well below 10%; we will be digging into it. As a reference point, we want to also calculate the accuracy the model would have achieved if it had just predicted "London" as the birth place for everyone in the dev set. Fill in `london_baseline.py` to calculate the accuracy of that approach and **report your result**. You should be able to leverage existing code such that the file is only a few lines long.

(e) (8 points) **Define a *span corruption* function for pretraining.**
In the file `src/dataset.py`, implement the `__getitem__()` function for the dataset class `CharCorruptionDataset`. Follow the instructions provided in the comments in `dataset.py`. Span corruption is explored in the T5 paper [2]. It randomly selects spans of text in a document and replaces them with unique tokens (noising). Models take this noised text, and are required to output

a pattern of each unique sentinel followed by the tokens that were replaced by that sentinel in the input. In this question, you'll implement a simplification that only masks out a single sequence of characters.

To help you debug, if you run the following code, it'll sample a few examples from your `CharCorruptionDataset` on the pretraining dataset `wiki.txt` and print them out for you.

```
python src/dataset.py charcorruption
```

Include a screenshot of the output in your report

(f) (10 points) **Pretrain, finetune, and make predictions. Budget about 1 hour for training.** Now fill in the *pretrain* portion of `run.py`, which will pretrain a model on the span corruption task. Additionally, modify your *finetune* portion to handle finetuning in the case *with* pretraining. In particular, if a path to a pretrained model is provided in the bash command, load this model before finetuning it on the birthplace prediction task. Pretrain your model on `wiki.txt` (which should take approximately 40-60 minutes), finetune it on `NameDataset` and evaluate it. Specifically, you should be able to run the following four commands:

```
# Pretrain the model
python src/run.py pretrain vanilla wiki.txt \
       --writing_params_path vanilla.pretrain.params

# Finetune the model
python src/run.py finetune vanilla wiki.txt \
       --reading_params_path vanilla.pretrain.params \
       --writing_params_path vanilla.finetune.params \
       --finetune_corpus_path birth_places_train.tsv

# Evaluate on the dev set; write to disk
python src/run.py evaluate vanilla wiki.txt \
       --reading_params_path vanilla.finetune.params \
       --eval_corpus_path birth_dev.tsv \
       --outputs_path vanilla.pretrain.dev.predictions

# Evaluate on the test set; write to disk
python src/run.py evaluate vanilla wiki.txt \
       --reading_params_path vanilla.finetune.params \
       --eval_corpus_path birth_test_inputs.tsv \
       --outputs_path vanilla.pretrain.test.predictions
```

Please report your accuracy. We expect the dev accuracy will be at least 15%, and will expect a similar accuracy on the held out test set.

(g) (9 points) **Write and try out a different kind of position embeddings (Budget about 1 hour for training)**

In the previous part, you used the vanilla Transformer model, which used learned positional embeddings. In the written part, you also learned about the sinusoidal positional embeddings used in the original Transformer paper. In this part, you'll implement a different kind of positional embedding, called *RoPE* (Rotary Positional Embedding) [3].

RoPE is a fixed positional embedding that is designed to encode relative position rather than absolute position. The issue with absolute positions is that if the transformer won't perform well on context lengths (e.g. 1000) much larger than it was trained on (e.g. 128), because the distribution of the position embeddings will be very different from the ones it was trained on. Relative position embeddings like RoPE alleviate this issue.

Given a feature vector with two features $x_t^{(1)}$ and $x_t^{(2)}$ at position t in the sequence, the RoPE positional embedding is defined as:

$$\mathrm{RoPE}(x_t^{(1)}, x_t^{(2)}, t) = \begin{pmatrix} \cos t\theta & -\sin t\theta \\ \sin t\theta & \cos t\theta \end{pmatrix} \begin{pmatrix} x_t^{(1)} \\ x_t^{(2)} \end{pmatrix}$$

where θ is a fixed angle. For two features, the RoPE operation corresponds to a 2D rotation of the features by an angle $t\theta$. Note that the angle is a function of the position t.

For a d dimensional feature, RoPE is applied to each pair of features with an angle θ_i defined as $\theta_i = 10000^{-2(i-1)/d}$, $i \in \{1, 2, \dots, d/2\}$.

$$\begin{pmatrix} \cos t\theta_1 & -\sin t\theta_1 & 0 & 0 & \cdots & 0 & 0 \\ \sin t\theta_1 & \cos t\theta_1 & 0 & 0 & \cdots & 0 & 0 \\ 0 & 0 & \cos t\theta_2 & -\sin t\theta_2 & \cdots & 0 & 0 \\ 0 & 0 & \sin t\theta_2 & \cos t\theta_2 & \cdots & 0 & 0 \\ \vdots & \vdots & \vdots & \vdots & \ddots & \vdots & \vdots \\ 0 & 0 & 0 & 0 & \cdots & \cos t\theta_{d/2} & -\sin t\theta_{d/2} \\ 0 & 0 & 0 & 0 & \cdots & \sin t\theta_{d/2} & \cos t\theta_{d/2} \end{pmatrix} \begin{pmatrix} x_t^{(1)} \\ x_t^{(2)} \\ x_t^{(3)} \\ x_t^{(4)} \\ \vdots \\ x_t^{(d-1)} \\ x_t^{(d)} \end{pmatrix} \tag{1}$$

Finally, instead of adding the positional embeddings to the input embeddings, RoPE is applied to the key and query vectors for each head in the attention block for all the Transformer layers.

i. (2 points) Using the rotation interpretation, RoPE operation can be viewed as rotation of the complex number $x_t^{(1)} + ix_t^{(2)}$ by an angle $t\theta$. Recall that this corresponds to multiplication by $e^{it\theta} = \cos t\theta + i \sin t\theta$.

For higher dimensional feature vectors, this interpretation allows us to compute Equation 1 more efficiently. Specifically, we can rewrite the RoPE operation as an element-wise multiplication (denoted by $\odot$) of two vectors as follows:

$$\begin{pmatrix} \cos t\theta_1 + i \sin t\theta_1 \\ \cos t\theta_2 + i \sin t\theta_2 \\ \vdots \\ \cos t\theta_{d/2} + i \sin t\theta_{d/2} \end{pmatrix} \odot \begin{pmatrix} x_t^{(1)} + ix_t^{(2)} \\ x_t^{(3)} + ix_t^{(4)} \\ \vdots \\ x_t^{(d-1)} + ix_t^{(d)} \end{pmatrix} \tag{2}$$

Show that the elements of the vector in Equation 1 can be obtained from Equation 2. Note that some additional operations like reshaping are necessary to make the two expressions equal but you do not need to provide a detailed derivation for full points.

ii. (1 point) **Relative Embeddings.** Now we will show that the dot product of the RoPE embeddings of two vectors at positions t_1 and t_2 depends on the relative position $t_1 - t_2$ only.

For simiplicity, we will assume two dimensional feature vectors (eg. $[a, b]$) and work with their complex number representations (eg. $a + ib$).

Show that $\langle \mathrm{RoPE}(z_1, t_1), \mathrm{RoPE}(z_2, t_2) \rangle = \langle \mathrm{RoPE}(z_1, t_1 - t_2), \mathrm{RoPE}(z_2, 0) \rangle$ where $\langle \cdot, \cdot \rangle$ denotes the dot product and $\mathrm{RoPE}(z, t)$ is the RoPE embedding of vector z at position t.

(Hint: Dot product of vectors represented as complex numbers is given by $\langle z_1, z_2 \rangle = \mathrm{Re}(\overline{z_1} z_2)$. For a complex number $z = a + ib$ $(a, b \in \mathbb{R})$, $\mathrm{Re}(z) = a$ indicates the real component of z and $\bar{z} = a - ib$ is the complex conjugate of z.)

iii. (7 points) In the provided code, RoPE is implemented using the functions `precompute_rotary_emb` and `apply_rotary_emb` in `src/attention.py`. You need to implement these functions and the parts of code marked `[part g]` in `src/attention.py` and `src/run.py` to use RoPE in the model.

Train a model with RoPE on the span corruption task and finetune it on the birthplace prediction task. Specifically, you should be able to run the following four commands:

```
# Pretrain the model
python src/run.py pretrain rope wiki.txt \
       --writing_params_path rope.pretrain.params

# Finetune the model
python src/run.py finetune rope wiki.txt \
       --reading_params_path rope.pretrain.params \
       --writing_params_path rope.finetune.params \
       --finetune_corpus_path birth_places_train.tsv

# Evaluate on the dev set; write to disk
python src/run.py evaluate rope wiki.txt \
       --reading_params_path rope.finetune.params \
       --eval_corpus_path birth_dev.tsv \
       --outputs_path rope.pretrain.dev.predictions

# Evaluate on the test set; write to disk
python src/run.py evaluate rope wiki.txt \
       --reading_params_path rope.finetune.params \
       --eval_corpus_path birth_test_inputs.tsv \
       --outputs_path rope.pretrain.test.predictions
```

Please report your obtained accuracy. We'll score your model as to whether it gets at least 30% accuracy on the test set, which has answers held out.

3. Considerations in pretrained knowledge (5 points)

(a) (1 point) Succinctly explain why the pretrained (vanilla) model was able to achieve an accuracy of above 10%, whereas the non-pretrained model was not.

(b) (2 points) Take a look at some of the correct predictions of the pretrain+finetuned vanilla model, as well as some of the errors. We think you'll find that it's impossible to tell, just looking at the output, whether the model *retrieved* the correct birth place, or *made up* an incorrect birth place. Consider the implications of this for user-facing systems that involve pretrained NLP components. Come up with two **distinct** reasons why this model behavior (i.e. unable to tell whether it's retrieved or made up) may cause concern for such applications, and an example for each reason.

(c) (2 points) If your model didn't see a person's name at pretraining time, and that person was not seen at fine-tuning time either, it is not possible for it to have "learned" where they lived. Yet, your model will produce *something* as a predicted birth place for that person's name if asked. Concisely describe a strategy your model might take for predicting a birth place for that person's name, and one reason why this should cause concern for the use of such applications.

(While 3b discussed the problems that could arise from made up predictions, 3c asks for a mechanism the model could be using for generating birth places of people not seen at fine-tuning time and why such a mechanism could be problematic.)

Submission Instructions

You will submit this assignment on GradeScope as two submissions – one for **Assignment 5 [coding]** and another for **Assignment 5 [written]**:

1. Upload your `assignment5.zip` file to GradeScope to **Assignment 5 [coding]**. Ensure all .py files are included!

2. Upload your written report to GradeScope to **Assignment 5 [written]**. **Please type the answers to these written questions (to make TA lives easier).**

References

[1] Radford, A., Narasimhan, K., Salimans, T., and Sutskever, I. Improving language understanding with unsupervised learning. *Technical report, OpenAI* (2018).

[2] Raffel, C., Shazeer, N., Roberts, A., Lee, K., Narang, S., Matena, M., Zhou, Y., Li, W., and Liu, P. J. Exploring the limits of transfer learning with a unified text-to-text transformer. *Journal of Machine Learning Research 21*, 140 (2020), 1–67.

[3] Su, J., Ahmed, M., Lu, Y., Pan, S., Bo, W., and Liu, Y. Roformer: Enhanced transformer with rotary position embedding. *Neurocomputing 568* (2024), 127063.