CSCI 4140: Natural Language Processing
CSCI/DASC 6040: Computational Analysis of Natural Languages
Spring 2025
Homework 5 – Self-Attention, Transformers, and Pretraining
Due Sunday, April 6, at 11:59 PM

This assignment is an investigation into Transformer self-attention building blocks, and the effects of pretraining. Through this assignment you'll get experience with practical system-building through repurposing an existing codebase.

Extending a research codebase (quick summary): You'll get some experience and intuition for a cutting-edge research topic in NLP – teaching NLP models facts about the world through pretraining, and accessing that knowledge through finetuning. You'll train a Transformer model to attempt to answer simple questions of the form "Where was person [x] born?" – without providing any input text from which to draw the answer. You'll find that models are able to learn some facts about where people were born through pretraining, and access that information during fine-tuning to answer the questions.

Then, you'll take a harder look at the system you built, and reason about the implications and concerns about relying on such implicit pretrained knowledge.

Note. Here is something to keep in mind as you plan your time for this assignment. This assignment involves a pretraining step that takes approximately one hour to perform on a GPU, and you'll have to do it twice. A Jupyter set-up notebook has been provided. The one-hour timeline is an upper bound on the training time assuming older/slower GPU. On faster GPUs, the pretraining can finish in around 30-40 minutes.

1 Pretrained Transformer models and knowledge access (100 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.

You'll need around 3 hours for training, so budget your time accordingly! We have provided a sample notebook with the commands that require GPU training. **Note that dataset multi-processing can fail on local machines without GPU, so to debug locally, you might have to change num workers to 0.**

Your work with this codebase is as follows:

(a) (0 points) **Check out the demo.**
In the `mingpt-demo/` folder is a Jupyter notebook `play_char.ipynb` that trains and samples from a Transformer language model. Take a look at it (locally on your computer) to get somewhat familiar with how it defines and trains models. Some of the code you're writing below will be inspired by what you see in this notebook.
Note that you do not have to write any code, run the notebook or submit written answers for this part.

(b) (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 birthplace 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.

(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.

Hint: Both `run.py` and `play_char.ipynb` use minGPT so the code for this part will be similar to the training code in `play_char.ipynb`.

(d) (15 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
```

Training will take less than 10 minutes (on GPU). Report your model's accuracy on the dev set (as printed by the second command above). We have Tensorboard logging 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 why in Part 4. As a reference point, we want to also calculate the accuracy the model would have achieved if it had just predicted "London" as the birthplace for everyone in the dev set. Fill in `london_baseline.py` to calculate the accuracy of that approach and report your result in the file. You should be able to leverage existing code such that the file is only a few lines long.

(e) (25 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. 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
```

(f) (25 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
```

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

(g) (35 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 this part, you'll implement a different kind of positional embedding, called *RoPE* (Rotary

Positional Embedding).
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., 1,000) 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:

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

where θ is a fixed angle. For two features, the RoPE operation corresponds to a 2D rotation of the features by an angle θ. 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_t defined as $\theta_i = 10000^{-2(i-1)/d}, i \in 1, 2, \ldots, d/2$.

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

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.
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 $e^{it\theta} = \cos t\theta + i \sin \theta$.
For higher dimensional feature vectors, this interpretation allows us to compute Equation 3 more efficiently. Specifically, we can rewrite the RoPE operation as an element-wise multiplication (denoted by $\odot$) of two vectors as follows:

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

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
```

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

2 Considerations in pretrained knowledge (10 points)

Please type the answers to these written questions (to make TA lives easier).

(a) (2 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) (4 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) (4 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 ethical concern this raises for the use of such applications.
(While 2b discussed the problems that could arise from made up predictions, 2c 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.)

3 Submission instructions

Verify that the following files exist at these specified paths within your assignment directory:

- The no-pretraining model and predictions:
  - `vanilla.model.params`,
  - `vanilla.nopretrain.dev.predictions`,
  - `vanilla.nopretrain.test.predictions`
- The London baseline accuracy: `london_baseline_accuracy.txt`
- The pretrain-finetune model and predictions:
  - `vanilla.finetune.params`,

  - `vanilla.pretrain.dev.predictions`,
  - `vanilla.pretrain.test.predictions`

- The RoPE model and predictions:
  - `rope.finetune.params`,
  - `rope.pretrain.dev.predictions`,
  - `rope.pretrain.test.predictions`

Run collect `submission.sh` (on Linux/Mac) or `collect submission.bat` (on Windows) to produce your `homework5.zip` file. Add your written answers to part 2 to the `homework5.zip` file, and then upload it to Canvas.