CS 224N Spring 2024 Assignment 4
Self-Attention, Transformers, and Pretraining

Note. Here are some things to keep in mind as you plan your time for this assignment.

- There are math questions again!
- The total amount of PyTorch code to write, and code complexity, of this assignment is lower than Assignment 3. However, you're also given less guidance or scaffolding in how to write the code.
- This assignment involves a pretraining step that takes approximately 1 hour to perform on GCP, and you'll have to do it twice. Colab set-up notebook has been provided similar to Assignment 4. The 1 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.

This assignment is an investigation into Transformer self-attention building blocks, and the effects of pretraining. It covers mathematical properties of Transformers and self-attention through written questions. Further, you'll get experience with practical system-building through repurposing an existing codebase. The assignment is split into a written (mathematical) part and a coding part, with its own written questions. Here's a quick summary:

1. **Mathematical exploration:** What kinds of operations can self-attention easily implement? Why should we use fancier things like multi-headed self-attention? This section will use some mathematical investigations to illuminate a few of the motivations of self-attention and Transformer networks. **Note:** for all questions, you should justify your answer with mathematical reasoning when required.

2. **Extending a research codebase:** In this portion of the assignment, 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.

1. Attention Exploration (14 points)

Multi-head self-attention is the core modeling component of Transformers. In this question, we'll get some practice working with the self-attention equations, and motivate why multi-headed self-attention can be preferable to single-headed self-attention.

Recall that attention can be viewed as an operation on a *query* vector $q \in \mathbb{R}^d$, a set of *value* vectors $\{v_1, \ldots, v_n\}, v_i \in \mathbb{R}^d$, and a set of *key* vectors $\{k_1, \ldots, k_n\}, k_i \in \mathbb{R}^d$, specified as follows:

$$c = \sum_{i=1}^{n} v_i \alpha_i \tag{1}$$

$$\alpha_i = \frac{\exp(k_i^\top q)}{\sum_{j=1}^{n} \exp(k_j^\top q)} \tag{2}$$

with $alpha = \{\alpha_1, \ldots, \alpha_n\}$ termed the "attention weights". Observe that the output $c \in \mathbb{R}^d$ is an average over the value vectors weighted with respect to α.

(a) (3 points) **Copying in attention.** One advantage of attention is that it's particularly easy to "copy" a value vector to the output c. In this problem, we'll motivate why this is the case.

i. (2 points) The distribution α is typically relatively "diffuse"; the probability mass is spread out between many different α_i. However, this is not always the case. **Describe** (in one sentence) under what conditions the categorical distribution α puts almost all of its weight on some α_j, where $j \in \{1, \ldots, n\}$ (i.e. $\alpha_j \gg \sum_{i \neq j} \alpha_i$). What must be true about the query q and/or the keys $\{k_1, \ldots, k_n\}$?

Answer: Query $q \approx k_j$, and the keys $\{k_1, \ldots, k_n\}$ are nearly orthogonal.

ii. (1 point) Under the conditions you gave in (i), **describe** the output c.

Answer: Given the conditions, the output $c \approx v_j$.

(b) (2 points) **An average of two.** Instead of focusing on just one vector v_j, a Transformer model might want to incorporate information from *multiple* source vectors.

Consider the case where we instead want to incorporate information from **two** vectors v_a and v_b, with corresponding key vectors k_a and k_b. Assume that (1) all key vectors are orthogonal, so $k_i^\top k_j = 0$ for all $i \neq j$; and (2) all key vectors have norm 1. **Find an expression** for a query vector q such that $c \approx \frac{1}{2}(v_a + v_b)$, and **justify your answer**.* (Recall what you learned in part (a).)

Answer: Let $q = \lambda(k_a + k_b)$, where λ is a large positive scalar. Then $k_j^\top q = 0$ for $k_j \in \{k_1, \ldots, k_n\} \setminus \{k_a, k_b\}$.

$$\begin{aligned}
\alpha_a &= \frac{\exp(k_a^\top \lambda(k_a + k_b))}{\sum_{j=1}^{n} \exp(k_j^\top q)} \\
&= \frac{\exp(\lambda)}{\sum_{j=1}^{n} \exp(k_j^\top q)} \\
\alpha_b &= \frac{\exp(k_b^\top \lambda(k_a + k_b))}{\sum_{j=1}^{n} \exp(k_j^\top q)} \\
&= \frac{\exp(\lambda)}{\sum_{j=1}^{n} \exp(k_j^\top q)} \\
\Rightarrow \alpha_a = \alpha_b &= \frac{\exp(\lambda)}{2 \cdot \exp(\lambda) + n - 2} \approx \frac{1}{2} \\
\alpha_i &\approx 0 \text{ for } i \neq a, b \\
c &\approx \frac{1}{2}(v_a + v_b)
\end{aligned}$$

*Hint: while the softmax function will never *exactly* average the two vectors, you can get close by using a large scalar multiple in the expression.

(c) (5 points) **Drawbacks of single-headed attention:** In the previous part, we saw how it was *possible* for a single-headed attention to focus equally on two values. The same concept could easily be extended to any subset of values. In this question we'll see why it's not a *practical* solution.

Consider a set of key vectors $\{k_1, \ldots, k_n\}$ that are now randomly sampled, $k_i \sim \mathcal{N}(\mu_i, \Sigma_i)$, where the means $\mu_i \in \mathbb{R}^d$ are known to you, but the covariances Σ_i are unknown (unless specified otherwise in the question). Further, assume that the means μ_i are all perpendicular; $\mu_i^\top \mu_j = 0$ if $i \neq j$, and unit norm, $\|\mu_i\| = 1$.

i. (2 points) Assume that the covariance matrices are $\Sigma_i = \alpha I, \forall i \in \{1, 2, \ldots, n\}$, for vanishingly small α. Design a query q in terms of the μ_i such that as before, $c \approx \frac{1}{2}(v_a + v_b)$, and provide a brief argument as to why it works.

Answer: Let $q = \lambda(\mu_a + \mu_b)$, where λ is a large positive scalar. Then $\mu_j^\top q = 0$ for $j \in \{\mu_1, \ldots, \mu_n\} \setminus \{\mu_a, \mu_b\}$. Since α is vanishingly small, we can approximate $k_i \approx \mu_i$.

$$
\begin{aligned}
\alpha_a &= \frac{\exp(\mu_a^\top \lambda(\mu_a + \mu_b))}{\sum_{j=1}^n \exp(\mu_j^\top q)} \\
&= \frac{\exp(\lambda)}{\sum_{j=1}^n \exp(\mu_j^\top q)} \\
\alpha_b &= \frac{\exp(\mu_b^\top \lambda(\mu_a + \mu_b))}{\sum_{j=1}^n \exp(\mu_j^\top q)} \\
&= \frac{\exp(\lambda)}{\sum_{j=1}^n \exp(\mu_j^\top q)} \\
\Rightarrow \alpha_a = \alpha_b &= \frac{\exp(\lambda)}{2 \cdot \exp(\lambda) + n - 2} \approx \frac{1}{2} \\
\alpha_i &\approx 0 \text{ for } i \neq a, b \\
c &\approx \frac{1}{2}(v_a + v_b)
\end{aligned}
$$

ii. (3 points) Though single-headed attention is resistant to small perturbations in the keys, some types of larger perturbations may pose a bigger issue. In some cases, one key vector k_a may be larger or smaller in norm than the others, while still pointing in the same direction as μ_a.[†]

As an example, let us consider a covariance for item a as $\Sigma_a = \alpha I + \frac{1}{2}(\mu_a \mu_a^\top)$ for vanishingly small α (as shown in figure 1). This causes k_a to point in roughly the same direction as μ_a, but with large variances in magnitude. Further, let $\Sigma_i = \alpha I$ for all $i \neq a$.

When you sample $\{k_1, \ldots, k_n\}$ multiple times, and use the q vector that you defined in part i., what do you expect the vector c will look like qualitatively for different samples? Think about how it differs from part (i) and how c's variance would be affected.

[†]Unlike the original Transformer, newer Transformer models apply layer normalization before attention. In these pre-layernorm models, norms of keys cannot be too different which makes the situation in this question less likely to occur.

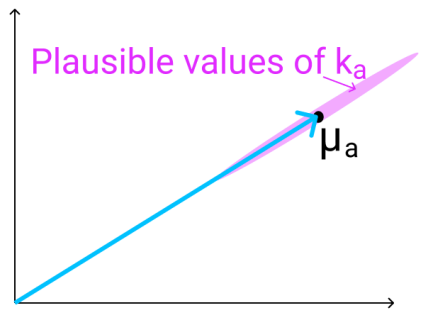

Figure 1: The vector μ_a (shown here in 2D as an example), with the range of possible values of k_a shown in red. As mentioned previously, k_a points in roughly the same direction as μ_a, but may have larger or smaller magnitude.

Answer: When sampling k_i from $\{k_1, \ldots, k_n\}$, similar to part (i), if $i \neq a, b$, we have $k_i^\top q \approx 0$; if $i = b$, we have $k_i^\top q \approx \lambda$.
If $i = a$, let $k_a = \mu_a + \epsilon_a$, where $\epsilon_a \sim \mathcal{N}(0, \alpha I + \frac{1}{2}(\mu_a \mu_a^\top))$.

$$\begin{aligned}
k_a^\top q &= (\mu_a + \epsilon_a)^\top \lambda(\mu_a + \mu_b) \\
&= \lambda + \lambda(\mu_a + \mu_b)^\top \epsilon_a \\
\mathbb{E}[k_a^\top q] &= \lambda + \lambda(\mu_a + \mu_b)^\top \mathbb{E}[\epsilon_a] \\
&= \lambda \\
\mathrm{Var}[k_a^\top q] &= \lambda^2 \mathrm{Var}[(\mu_a + \mu_b)^\top \epsilon_a] \\
&= \lambda^2 (\mu_a + \mu_b)(\alpha I + \frac{1}{2}\mu_a \mu_a^T)(\mu_a + \mu_b) \\
&= \lambda^2 (\alpha \|\mu_a + \mu_b\|^2 + \frac{1}{2}((\mu_a + \mu_b)^\top \mu_a)^2) \\
&= \lambda^2 (2\alpha + \frac{1}{2}) \\
&\approx \frac{\lambda^2}{2}
\end{aligned}$$

While the mean of $k_a^\top q$ is the same, its variance is considerable as λ is a large scalar. This results in an oscillating α_a. If k_a is large, the vector $c \approx v_a$, and vice versa. This prevents proper averaging of the two value vectors.

(d) (3 points) **Benefits of multi-headed attention:** Now we'll see some of the power of multi-headed attention. We'll consider a simple version of multi-headed attention which is identical to single-headed self-attention as we've presented it, except two query vectors (q_1 and q_2) are defined, which leads to a pair of vectors (c_1 and c_2), each the output of single-headed attention given its respective query vector. The final output of the multi-headed attention is their average, $\frac{1}{2}(c_1 + c_2)$.

As in question 1(c), consider a set of key vectors $\{k_1, \ldots, k_n\}$ that are randomly sampled, $k_i \sim \mathcal{N}(\mu_i, \Sigma_i)$, where the means μ_i are known to you, but the covariances Σ_i are unknown. Also as before, assume that the means μ_i are mutually orthogonal; $\mu_i^\top \mu_j = 0$ if $i \neq j$, and unit norm, $\|\mu_i\| = 1$.

i. (1 point) Assume that the covariance matrices are $\Sigma_i = \alpha I$, for vanishingly small α. Design q_1 and q_2 in terms of μ_i such that c is approximately equal to $\frac{1}{2}(v_a + v_b)$. Note that q_1 and q_2 should have different expressions.

Answer: Let $q_1 = \lambda \mu_a$ and $q_2 = \lambda \mu_b$, where λ is a large positive scalar.

ii. (2 points) Assume that the covariance matrices are $\Sigma_a = \alpha I + \frac{1}{2}(\mu_a \mu_a^\top)$ for vanishingly small α, and $\Sigma_i = \alpha I$ for all $i \neq a$. Take the query vectors q_1 and q_2 that you designed in part i. What, qualitatively, do you expect the output c to look like across different samples of the key vectors? Explain briefly in terms of variance in c_1 and c_2. You can ignore cases in which $k_a^\top q_i < 0$.

Answer: By a similar process to that in question 1(c), we have $\mathrm{Var}[k_a^T q_1] \approx \frac{\lambda^2}{2}$, and $\mathrm{Var}[k_a^T q_2] \approx 0$. While variance of c_1 is high, the vector c_2 has consistent low variance. If k_a is large, the output $c \approx \frac{1}{2}(v_a + v_b)$. If k_a is small, the output $c \approx \frac{1}{2} v_b$.

(e) (1 point) Based on part (d), briefly summarize how multi-headed attention overcomes the drawbacks of single-headed attention that you identified in part (c).

Answer: Due to the stablizing effect of c_2, the output never completely ignore v_b. Averaging c_1 with the stable c_2 also reduces the variance of c.

2. 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:

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

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:

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

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: $\mathrm{softmax}(\mathbf{P}\mathbf{A}\mathbf{P}^\top) = \mathbf{P}\ \mathrm{softmax}(\mathbf{A})\ \mathbf{P}^\top$ and $\mathrm{ReLU}(\mathbf{P}\mathbf{A}) = \mathbf{P}\ \mathrm{ReLU}(\mathbf{A})$.

[‡]Outer product with $\mathbf{1}$ represents broadcasting operation and makes feed forward network notations mathematically sound.

Answer:

$$\begin{aligned}
\mathbf{H}_{\text{perm}} &= \text{softmax}\left(\frac{\mathbf{PXW}_Q(\mathbf{PXW}_K)^\top}{\sqrt{d}}\right)\mathbf{PXW}_V \\
&= \mathbf{P}\cdot\text{softmax}\left(\frac{\mathbf{QK}}{\sqrt{d}}\right)\mathbf{P}^\top\mathbf{PV} \\
&= \mathbf{PH} \\
\mathbf{Z}_{\text{perm}} &= \text{ReLU}(\mathbf{PHW}_1 + 1\cdot\mathbf{b}_1)\mathbf{W}_2 + 1\cdot\mathbf{b}_2 \\
&= \mathbf{P}\cdot\text{ReLU}(\mathbf{HW}_1 + 1\cdot\mathbf{b}_1)\mathbf{W}_2 + \mathbf{P}\cdot 1\cdot\mathbf{b}_2 \\
&= \mathbf{PZ}
\end{aligned}$$

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.

Answer: There is no inherent notion of order. The model cannot capture the sequential information and positional dependence in text, which makes it insufficient for language processing because language heavily relies on word order to convey meaning.

(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:

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

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.

Answer: Yes. Since the positional information are now encoded in input word embeddings, the model can distinguish tokens by position and learn to process them based on their order.

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.

Answer: Yes, but only for impractically large sequence lengths. For instance,

$$\Phi_{(t_1,2i)} = \sin\left(\frac{t_1}{10000^{2i/d}}\right) = \sin\left(\frac{t_2}{10000^{2i/d}}\right) = \Phi_{(t_2,2i)}$$

where $t_2 - t_1$ is the least common multiple of $\{2\pi\cdot 10000^{0/d}, \ldots, 2\pi\cdot 10000^{2(d/2-1)/d}\}$. This far exceeds any reasonable sequence length.

3. 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

[§]Here d is assumed even which is typically the case for most models.

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].

As in previous assignments, you will want to develop on your machine locally, then run training on GCP/Colab. You can use the same conda environment from previous assignments for local development, and the same process for training on a GPU.[¶]

You'll need around 3 hours for training, so budget your time accordingly! We have provided a sample Colab with the 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 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.

(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

[¶]See CS224n GCP Guide for a refresher on GCP.

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) (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
```

Training will take less than 10 minutes (on GCP). Report your model's accuracy on the dev set (as printed by the second command above). Similar to assignment 3, we also have Tensorboard logging in assignment 4 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 birth place 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.

Answer:

```
# Accuracy on the dev set
Correct: 12.0 out of 500.0: 2.4%
```

The model would have achieved 5% accuracy if it had just always predicted "London" as the birth place.

(e) (10 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.

This question will be graded via autograder based on whether your span corruption function implements some basic properties of our spec. We'll instantiate the CharCorruptionDataset with our own data, and draw examples from it.

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) (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
```

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

Answer:

```
# Accuracy on the dev set
Correct: 88.0 out of 500.0: 17.599999999999998%
```

(g) (11 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:

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

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

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

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.

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 3 more efficiently. Specifically, we can rewrite the RoPE operation as an element-wise multiplication (denoted by $\odot$) of two vectors as follows:

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

Show that the elements of the vector in Equation 3 can be obtained from Equation 4. 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.

Answer: For Equation 3, the first two elements of the vector are

$$\cos t\theta_1 x_t^{(1)} - \sin t\theta_1 x_t^{(2)}, \sin t\theta_1 x_t^{(1)} + \cos t\theta_1 x_t^{(2)}$$

For Equation 4, the first element of the vector is

$$(\cos t\theta_1 x_t^{(1)} - \sin t\theta_1 x_t^{(2)}) + i(\sin t\theta_1 x_t^{(1)} + \cos t\theta_1 x_t^{(2)})$$

where the elements of the vector in Equation 3 are the real and the imaginary parts. The rest of the elements can be obtained following a similar process

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 $\langle \text{RoPE}(z_1, t_1), \text{RoPE}(z_2, t_2)\rangle = \langle \text{RoPE}(z_1, t_1 - t_2), \text{RoPE}(z_2, 0)\rangle$ where $\langle \cdot, \cdot \rangle$ denotes the dot product and $\text{RoPE}(z, t)$ is the RoPE embedding of vector z at position t.
(Hint: Dot product of vectors represented as complex numbers is given by $\langle z_1, z_2\rangle = \text{Re}(\overline{z_1} z_2)$. For a complex number $z = a + ib$ $(a, b \in \mathbb{R})$, $\text{Re}(z) = a$ indicates the real component of z and $\bar{z} = a - ib$ is the complex conjugate of z.)
Answer:

$$\begin{aligned}
&\langle \text{RoPE}(z_1, t_1), \text{RoPE}(z_2, t_2)\rangle \\
=&\text{Re}(((\cos t_1\theta z_1^{(1)} - \sin t_1\theta z_1^{(2)}) - i(\sin t_1\theta z_1^{(1)} + \cos t_1\theta z_1^{(2)})) \\
&((\cos t_2\theta z_2^{(1)} - \sin t_2\theta z_1^{(2)}) + i(\sin t_2\theta z_2^{(1)} + \cos t_2\theta z_2^{(2)}))) \\
=&\cos(t_1 - t_2)\theta z_1^{(1)} z_2^{(1)} - \sin(t_1 - t_2)\theta z_1^{(2)} z_2^{(1)} \\
&+ \sin(t_1 - t_2)\theta z_1^{(1)} z_2^{(1)} + \cos(t_1 - t_2)\theta z_1^{(2)} z_2^{(2)} \\
=&\text{Re}(((\cos(t_1 - t_2)\theta z_1^{(1)} - \sin(t_1 - t_2)\theta z_1^{(2)}) \\
&- i(\sin(t_1 - t_2)\theta z_1^{(1)} + \cos(t_1 - t_2)\theta z_1^{(2)}))(z_2^{(1)} + i z_2^{(2)})) \\
=&\langle \text{RoPE}(z_1, t_1 - t_2), \text{RoPE}(z_2, 0)\rangle
\end{aligned}$$

iii. (8 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
```

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

Answer:

```
# Accuracy on dev set
Correct: 171.0 out of 500.0: 34.2%
```

4. Considerations in pretrained knowledge (5 points)

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

(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.

Answer: While the pretrained model has learnt some general language patterns and possibly some facts during pretraining to make educated guesses, the non-pretrained model lacked any prior knowledge and is much less likely to guess correctly by chance.

(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.

Answer: 1. Lack of accountability. For example, it is impossible for a user to verify sources or hold the system accountable for errors. 2. Generating misinformation. For example, if a user is not aware of the unreliableness of the model, incorrect information could be propagated.

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

Answer: The model might predict a birth place with superficial patterns of names, e.g. associating names with frequently co-occured birthplaces. This could reinforce the stereotype of assuming nationality by names or overlook minority groups by predicting common locations.

Submission Instructions

You will submit this assignment on GradeScope as two submissions – one for **Assignment 4 [coding]** and another for **Assignment 4 [written]**:

1. 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

2. Run collect_submission.sh (on Linux/Mac) or collect_submission.bat (on Windows) to produce your assignment4.zip file.
3. Upload your assignment4.zip file to GradeScope to **Assignment 4 [coding]**.
4. Check that the public autograder tests passed correctly.
5. Upload your written solutions, for questions 1, parts of 2, and 3, to GradeScope to **Assignment 4 [written]**. Tag it properly!

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.