CS 224N: Assignment 5: Self-Attention, Transformers, and Pretraining

mantasu

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 4. However, you're also given less guidance or scaffolding in how to write the code.
- This assignment involves a pretraining step that takes approximately 2 hours to perform on Azure, and you'll have to do it twice.

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.

This assignment was originally created by John Hewitt, CS 224N Head TA in Winter 2021.

1. Attention exploration (22 points)

Multi-headed 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* $q \in \mathbb{R}^d$, 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 α_i 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_i.

(a) (4 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. (1 point) **Explain** why α can be interpreted as a categorical probability distribution.

Answer. There are n α scores - one for each value in a sequence. Each score is between 0 and 1 and can be interpreted as a probability. It is a distribution because all scores are normalized, i.e., they sum up to 1.

ii. (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. If the key values k_j compared to other key values $k_{i \neq j}$ are large (i.e., $k_j \gg k_i$, for $i \in \{1, ..., n\}$ and $i \neq j$), then the dot product between the key and the query will be large. This will cause softmax to put most of its probability mass onto this large value.

iii. (1 point) Under the conditions you gave in (ii), **describe** what properties the output c might have.

Answer. j^{th} value will have the most weight thus c will be similar to v_j, i.e., $c \approx v_j$.

iv. (1 point) **Explain** (in two sentences or fewer) what your answer to (ii) and (iii) means intuitively.

Answer. If the dot product (similarity) between some j^{th} word's *key* and a *query* is very large compared to other words' *keys* and the same *query*, then the *attention output* for that j^{th} word will approach its *value*. It's as if the *value* is "coppied" to the output.

(b) (7 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.

i. (3 points) How should we combine two d-dimensional vectors v_a, v_b into one output vector c in a way that preserves information from both vectors? In machine learning, one common way to do so is to take the average: $c = \frac{1}{2}(v_a + v_b)$. It might seem hard to extract information about the original vectors v_a and v_b from the resulting c, but under certain conditions one can do so. In this problem, we'll see why this is the case.

Suppose that although we don't know v_a or v_b, we do know that v_a lies in a subspace A formed by the m basis vectors $\{a_1, a_2, \ldots, a_m\}$, while v_b lies in a subspace B formed by the p basis vectors $\{b_1, b_2, \ldots, b_p\}$. (This means that any v_a can be expressed as a linear combination of its basis vectors, as can v_b. All basis vectors have norm 1 and orthogonal to each other.) Additionally, suppose that the two subspaces are orthogonal; i.e. $a_j^\top b_k = 0$ for all j, k.
Using the basis vectors $\{a_1, a_2, \ldots, a_m\}$, construct a matrix M such that for arbitrary vectors $v_a \in A$ and $v_b \in B$, we can use M to extract v_a from the sum vector $s = v_a + v_b$. In other words, we want to construct M such that for any v_a, v_b, $Ms = v_a$).
Note: both M and v_a, v_b should be expressed as a vector in $\mathbb{R}^d$, not in terms of vectors from A and B.

Hint: Given that the vectors $\{a_1, a_2, \ldots, a_m\}$ are both *orthogonal* and *form a basis* for v_a, we know that there exist some $c_1, c_2, \ldots, c_m$ such that $v_a = c_1 a_1 + c_2 a_2 + \cdots + c_m a_m$. Can you create a vector of these weights c?

Answer. Assume that $A \in \mathbb{R}^{d \times m}$ is a matrix of concatenated basis vectors $\{\mathbf{a}_1, \mathbf{a}_2, \ldots, \mathbf{a}_m\}$ and $B \in \mathbb{R}^{d \times p}$ is a matrix of concatenated basis vectors $\{\mathbf{b}_1, \mathbf{b}_2, \ldots, \mathbf{b}_p\}$. Linear combinations of vectors $\mathbf{v}_a$ and $\mathbf{v}_b$ can then be expressed as:

$$\mathbf{v}_a = c_1\mathbf{a}_1 + c_2\mathbf{a}_2 + \cdots + c_m\mathbf{a}_m = A\mathbf{c}$$
$$\mathbf{v}_b = c'_1\mathbf{b}_1 + c'_2\mathbf{b}_2 + \cdots + c'_p\mathbf{b}_p = B\mathbf{c}'$$

We need to construct such M which, when multiplied with $\mathbf{v}_b$, produces $\mathbf{0}$ and, when multiplied with $\mathbf{v}_a$, produces the same vector (in terms of its own space):

$$M\mathbf{s} = \mathbf{v}_a$$
$$M\mathbf{v}_a + M\mathbf{v}_b = \mathbf{v}_a$$

It is easy to see that, since $\mathbf{a}_j^\top \mathbf{b}_k = 0$ for all j, k, $A^\top B = \mathbf{0}$. And, since $\mathbf{a}_i^\top \mathbf{a}_j = 0$ whenever $j \neq i$ (orhtogonal) and since $\mathbf{a}_i^\top \mathbf{a}_j = 1$ whenever $j = i$ (norm 1), $A^\top A = I$. If we substitute M with $AA^\top$, $\mathbf{v}_a$ with $A\mathbf{c}$, and $\mathbf{v}_b$ with $B\mathbf{c}'$:

$$AA^\top A\mathbf{c} + AA^\top B\mathbf{c}' = AI\mathbf{c} + A\mathbf{0}\mathbf{c}' = A\mathbf{c} = \mathbf{v}_a$$

Thus, in terms of $\mathbb{R}^d$, $M = AA^\top$ because $M \in \mathbb{R}^{d \times d}$ (however, in terms of A, $\mathbf{v}_a$ would just be $\mathbf{c}$, in which case $M = A^\top$).

ii. (4 points) As before, let v_a and v_b be two value vectors corresponding to key vectors k_a and k_b, respectively. 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.[1] **Find an expression** for a query vector q such that $c \approx \frac{1}{2}(v_a + v_b)$. [2]

[1] Recall that a vector x has norm 1 iff $x^\top x = 1$.
[2] 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.

Answer. Assume that $\mathbf{c}$ is approximated as follows:

$$\mathbf{c} \approx 0.5\mathbf{v}_a + 0.5\mathbf{v}_b$$

This means we want $\alpha_a \approx 0.5$ and $\alpha_b \approx 0.5$, which can be achieved when (whenever $i \neq a$ and $i \neq b$):

$$\mathbf{k}_a^\top \mathbf{q} \approx \mathbf{k}_b^\top \mathbf{q} \gg \mathbf{k}_i^\top \mathbf{q}$$

Like explained in the previous question, if the dot product is big, the probability mass will also be big and we want a balanced mass between α_a and α_b. $\mathbf{q}$ will be largest for $\mathbf{k}_a$ and $\mathbf{k}_b$ when it is a large multiplicative of a vector that contains a component in $\mathbf{k}_a$ direction and in $\mathbf{k}_b$ direction:

$$\mathbf{q} = \beta(\mathbf{k}_a + \mathbf{k}_b), \quad \text{where } \beta \gg 0$$

Now, since the keys are orthogonal to each other, it is easy to see that:

$$\mathbf{k}_a^\top \mathbf{q} = \beta; \quad \mathbf{k}_b^\top \mathbf{q} = \beta; \quad \mathbf{k}_i^\top \mathbf{q} = 0, \text{ whever } i \neq a \text{ and } i \neq b$$

Thus when we exponentiate, only $\exp(\beta)$ will matter, because $\exp(0)$ will be insignificant to the probability mass. We get that:

$$\alpha_a = \alpha_b = \frac{\exp(\beta)}{n - 2 + 2\exp(\beta)} \approx \frac{\exp(\beta)}{2\exp(\beta)} \approx \frac{1}{2}, \text{ for } \beta \gg 0$$

(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. 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. Since the variances (diagonal covariance values) for $i \in \{1, 2, ..., n\}$ are vanishingly small, we can assume each key vector is close to its mean vector:

$$\mathbf{k}_i \approx \mu_i$$

Because all the mean vectors are perpendicular, the problem reduces to the previous case when all keys were perpendicular to each other. $\mathbf{q}$ can now be expressed as:

$$\mathbf{q} = \beta(\mu_a + \mu_b), \quad \text{where } \beta \gg 0$$

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. Specifically, 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 $i \neq a$.

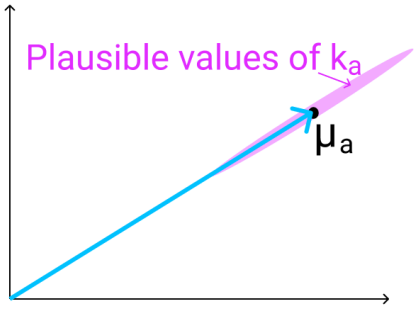

Figure 1: The vector μ_a (shown here in 2D as an example), with the range of possible values of k_a shown in red. As mentioned previously, k_a points in roughly the same direction as μ_a, but may have larger or smaller magnitude.

When you sample $\{k_1, \ldots, k_n\}$ multiple times, and use the q vector that you defined in part i., what qualitatively do you expect the vector c will look like for different samples?

Answer. Since $\mu_i^\top \mu_i = 1$, $\mathbf{k}_a$ varies between $(\alpha + 0.5)\mu_a$ and $(\alpha + 1.5)\mu_a$. All other $\mathbf{k}_i$, whenever $i \neq a$, almost don't vary at all. Noting that α is vanishingly small:

$$\mathbf{k}_a \approx \gamma \mu_a, \quad \text{where } \gamma \sim \mathcal{N}(1, 0.5)$$
$$\mathbf{k}_i \approx \mu_i, \quad \text{whenever } i \neq a$$

Since $\mathbf{q}$ is most similar in directions $\mathbf{k}_a$ and $\mathbf{k}_b$, we can assume that the dot product between $\mathbf{q}$ and any other key vector is 0 (since all key vectors are orthogonal). Thus there are 2 cases to consider (note that means are normalized and orthogonal to each other):

$$\mathbf{k}_a^\top \mathbf{q} \approx \gamma \mu_a^\top \beta(\mu_a + \mu_b) \approx \gamma\beta, \quad \text{where } \beta \gg 0$$
$$\mathbf{k}_b^\top \mathbf{q} \approx \mu_b^\top \beta(\mu_a + \mu_b) \approx \beta, \quad \text{where } \beta \gg 0$$

We can now directly solve for coefficients α_a and α_b, remembering that for large β values $\exp(0)$ are insignificant (note how $\frac{\exp(a)}{\exp(a)+\exp(b)} = \frac{\exp(a)}{\exp(a)+\exp(b)}\frac{\exp(-a)}{\exp(-a)} = \frac{1}{1+\exp(b-a)}$):

$$\alpha_a \approx \frac{\exp(\gamma\beta)}{\exp(\gamma\beta) + \exp(\beta)} \approx \frac{1}{1 + \exp(\beta(1-\gamma))}$$
$$\alpha_b \approx \frac{\exp(\beta)}{\exp(\beta) + \exp(\gamma\beta)} \approx \frac{1}{1 + \exp(\beta(\gamma-1))}$$

Since γ varies between 0.5 and 1.5, and since $\beta \gg 0$, we have that:

$$\alpha_a \approx \frac{1}{1+\infty} \approx 0; \quad \alpha_b \approx \frac{1}{1+0} \approx 1; \quad \text{when } \gamma = 0.5$$
$$\alpha_a \approx \frac{1}{1+0} \approx 1; \quad \alpha_b \approx \frac{1}{1+\infty} \approx 0; \quad \text{when } \gamma = 1.5$$

Since $\mathbf{c} \approx \alpha_a \mathbf{v}_a + \alpha_b \mathbf{v}_b$ because other terms are insignificant when β is large, we can see that $\mathbf{c}$ oscillates between $\mathbf{v}_a$ and $\mathbf{v}_b$:

$$\mathbf{c} \approx \mathbf{v}_b, \text{ when } \gamma \to 0.5; \qquad \mathbf{c} \approx \mathbf{v}_a, \text{ when } \gamma \to 1.5$$

(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 in this homework, 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 such that c is approximately equal to $\frac{1}{2}(v_a + v_b)$.

Answer. With the same assumptions as before, we can design $\mathbf{q}_1$ and $\mathbf{q}_2$ such that one of them copies $\mathbf{v}_a$ and another copies $\mathbf{v}_b$. Since all keys are similar to their means and following the explanation in question **(a) iv.**, we express the queries as:

$$\mathbf{q}_1 = \beta\mu_a, \quad \mathbf{q}_2 = \beta\mu_b, \quad \text{for } \beta \gg 0$$

This gives us (since means are orthogonal):

$$\mathbf{c}_1 \approx \mathbf{v}_a; \quad \mathbf{c}_2 \approx \mathbf{v}_b$$

And since multiheaded attention is just an average of the 2 values, we can see that:

$$\mathbf{c} \approx \frac{1}{2}(\mathbf{v}_a + \mathbf{v}_b)$$

Note extra answers:

1. It is also possible to set $\mathbf{q}_1$ to $\beta\mu_b$ and $\mathbf{q}_2$ to $\beta\mu_a$ which would yield the same answer, just that $\mathbf{v}_a$ and $\mathbf{v}_b$ would be swapped, i.e., $\mathbf{c}_1 = \mathbf{v}_b$ and $\mathbf{c}_2 = \mathbf{v}_a$.
2. It is even possible to use the same query designed in the previous question which would be the same for both queries in this question, i.e., $\mathbf{q}_1 = \mathbf{q}_2 = \beta(\mathbf{v}_a + \mathbf{v}_b)$. Then $\mathbf{c}_1 = \mathbf{c}_2 = \mathbf{c}$, i.e., an average of equal averages is the same average.

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? Please briefly explain why. You can ignore cases in which $k_a^\top q_i < 0$.

Answer. With regards to question **(c) ii.**, if we choose $\mathbf{q}_1 = \beta\mu_a$ and $\mathbf{q}_2 = \beta\mu_b$, we get that (note that all other key-query dot products will be insignificant):

$$\mathbf{k}_a^\top \mathbf{q}_1 = \gamma\mu_a^\top \beta\mu_a = \gamma\beta, \text{where } \beta \gg 0$$

$$\mathbf{k}_b^\top \mathbf{q}_2 = \mu_b^\top \beta\mu_b = \beta, \text{where } \beta \gg 0$$

We can solve for α values (again, note that all other key-query dot products will be insignificant when β is large):

$$\alpha_{a1} \approx \frac{\exp(\gamma\beta)}{\exp(\gamma\beta)} \approx 1; \quad \alpha_{b2} \approx \frac{\exp(\beta)}{\exp(\beta)} \approx 1$$

Since we can say that $\alpha_{i1} \approx 0$ for any $i \neq a$ and $\alpha_{i2} \approx 0$ for any $i \neq b$ is is easy to see that:

$$\mathbf{c}_1 \approx \mathbf{v}_a, \quad \mathbf{c}_2 \approx \mathbf{v}_b$$

Which means that the final output will always approximately be an average of the values:

$$\mathbf{c} \approx \frac{1}{2}(\mathbf{v}_a + \mathbf{v}_b)$$

Extra answers:

1. Now if we choose $\mathbf{q}_1 = \beta\mu_b$ and $\mathbf{q}_2 = \beta\mu_b$, a similar conclusion could be shown, just that the outputs would swap places, i.e., $\mathbf{c}_1 \approx \mathbf{v}_b$ and $\mathbf{c}_2 \approx \mathbf{v}_a$.
2. If we choose $\mathbf{q}_1 = \mathbf{q}_2 = \beta(\mathbf{v}_a + \mathbf{v}_b)$ then the problem would be similar to question **(c) ii.** as it is easy to show that, when $\gamma \to 0.5$, then $\alpha_{a1} = \alpha_{a2} \approx 0$ and $\alpha_{b1} = \alpha_{b2} \approx 1$, and, when $\gamma \to 1.5$, then $\alpha_{a1} = \alpha_{a2} \approx 1$ and $\alpha_{b1} = \alpha_{b2} \approx 0$. Then it is clear that $\mathbf{c}$ will approach $\frac{1}{2}(\mathbf{v}_a + \mathbf{v}_b)$ when $\gamma \to 1$ (i.e., when $\mathbf{k}_a$ is close to its mean μ_a).

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].

As in previous assignments, you will want to develop on your machine locally, then run training on Azure. You can use the same conda environment from previous assignments for local development, and the same process for training on Azure (see the CS224n Azure Guide for a refresher). Specifically, you'll still be running "`conda activate py37_pytorch`" on the Azure machine. You'll need around 5 hours for training, so budget your time accordingly!

Your work with this codebase is as follows:

(a) (0 points) **Check out the demo.**
In the `mingpt-demo/` folder is a Jupyter notebook 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 or submit written answers for this part.

(b) (0 points) **Read through `NameDataset`, 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 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) (5 points) **Make predictions (without pretraining).**
Train your model on `wiki.txt`, 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 Azure). Report your model's accuracy on the dev set (as printed by the second command above). Don't be surprised if it is well below 10%; we will be digging into why in Part 3. 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 your write-up. You should be able to leverage existing code such that the file is only a few lines long.

Answer.

- **Model's accuracy**: Correct: 8.0 out of 500.0: 1.6%
- **If only "London"**: Correct: 25.0 out of 500.0: 5.0%

(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
```

No written answer is required for this part.

(f) (10 points) **Pretrain, finetune, and make predictions. Budget 2 hours 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 two hours), finetune it on `NameDataset` and evaluate it. Specifically, you should be able to run the following four commands: (Don't be concerned if the loss appears to plateau in the middle of pretraining; it will eventually go back down.)

```
# 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
```

Report the accuracy on the dev set (printed by the third command above). We expect the dev accuracy will be at least 10%, and will expect a similar accuracy on the held out test set.

Answer. dev accuracy: Correct: 77.0 out of 500.0: 15.4%

(g) (10 points) **Research! Write and try out the *synthesizer* variant (Budget 2 hours for pretraining!)**

We'll now go to changing the Transformer architecture itself – specifically, the self-attention module. While we've been using a self-attention scoring function based on dot products, this involves a rather intensive computation that's quadratic in the sequence length. This is because the dot product between ℓ^2 pairs of word vectors is computed in each computation. *Synthesized attention* [3] is a very recent alternative that has potential benefits by removing this dot product (and quadratic computation) entirely. It's a promising idea, and one way for us to ask, "What's important/right about the Transformer architecture, and where can we improve/prune aspects of it?" In `attention.py`, implement the `forward()` method of `SynthesizerAttention`, which implements a variant of the Synthesizer proposed in the cited paper.

The provided `CausalSelfAttention` implements the following attention for each head of the multi-headed attention: Let $X \in \mathbb{R}^{\ell \times d}$ (where ℓ is the block size and d is the total dimensionality, d/h is the dimensionality per head.).[3] Let $Q, K, V \in \mathbb{R}^{d \times d/h}$. Then the output of the self-attention head is

$$Y_i = \text{softmax}\left(\frac{(XQ_i)(XK_i)^\top}{\sqrt{d/h}}\right)(XV_i) \tag{3}$$

where $Y_i \in \mathbb{R}^{\ell \times d/h}$. Then the output of the self-attention is a linear transformation of the concatenation of the heads:

$$Y = [Y_1; \ldots; Y_h]A \tag{4}$$

where $A \in \mathbb{R}^{d \times d}$ and $[Y_1; \ldots; Y_h] \in \mathbb{R}^{\ell \times d}$. The code also includes dropout layers which we haven't written here. We suggest looking at the provided code and noting how this equation is implemented in PyTorch.

Your job is to implement the following variant of attention. Instead of Equation 3, implement the following in `SynthesizerAttention`:

$$Y_i = \text{softmax}\big(\text{ReLU}(XA_i + b_1)B_i + b_2\big)(XV_i), \tag{5}$$

where $A_i \in \mathbb{R}^{d \times d/h}$, $B_i \in \mathbb{R}^{d/h \times \ell}$, and $V_i \in \mathbb{R}^{d \times d/h}$.[4] One way to interpret this is as follows: The term $(XQ_i)(XK_i)^\top$ is an $\ell \times \ell$ matrix of attention scores, computed as all pairs of dot products between word embeddings. The synthesizer variant eschews the all-pairs dot product and directly computes the $\ell \times \ell$ matrix of attention scores by mapping each d-dimensional vector of each head for X to an ℓ-dimesional vector of unnormalized attention weights.

In the rest of the code in the `src/` folder, modify your model to support using either `CausalSelfAttention` or `SynthesizerAttention`. Add the ability to switch between these attention variants depending on whether "vanilla" (for causal self-attention) or "synthesizer" (for the synthesizer variant) is selected in the command line arguments (see the section marked `[part g]` in `src/run.py`). You are free to implement this functionality in any way you choose, so long as it supports these command line arguments.

[3] Note that these dimensionalities do not include the minibatch dimension.

[4] Hint: copy over the `CausalSelfAttention` class, and modify it minimally for this.

Below are bash commands that your code should support in order to pretrain the model, finetune it, and make predictions on the dev and test sets. Note that the pretraining process will take approximately 2 hours.

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

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

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

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

Report the accuracy of your synthesizer attention model on birthplace prediction on `birth_dev.tsv` after pretraining and fine-tuning.

i. (8 points) We'll score your model as to whether it gets at least 5% accuracy on the test set, which has answers held out.

ii. (2 points) Why might the *synthesizer* self-attention not be able to do, in a single layer, what the key-query-value self-attention can do?

Answer.

1. **dev accuracy**: Correct: 46.0 out of 500.0: 9.2%
2. Synthesizer attention is unable to do what the causal attention can do because it does not extract keys and queries from the input - it estimates that by remapping input to a lower dimensional space for each head. In other words, it is more difficult to represent context because for every word in sequence, that word is unable to choose which parts to pay attention to in the rest of the sequence (because it does not have access to their keys). Thus the synthesizer may not be able to capture the relevance between word pairs.

3. 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. A pretrained model was able to learn the relationships between words. It learned the how parts of a sentence depend on one another whereas only the finetuned model was trained to extract certain parts of the sentence without really "knowing" how they relate to given input. It is also worth to note that the dataset for the pretraining was a lot larger thus the model could have "memorised" the relevant parts of the questions.

(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. Such behaviour may cause users to refer to false information in their work unintentionally. For instance, if the user wants to mention a birthplace of a famous person, they might provide a wrong place if the model does not provide true information. This could affect user's work quality.
2. Such behaviour may cause users to spread false information around the subject. For example, if the user learns something about a famous person, such as their birthplace, from a model that retrieves information, but that information is false, the person might unintentionally spread false facts about famous people and others may believe it. This could cause confusion and arguments in society.

(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. (You do not need to submit the same answer for 3c as for 3b.)

Answer. The model, given a name, will try to maximize its relevance with its parameters (it will look for information that belongs to people with similar names to the provided name). If there is a typo in the name, then such functionality is desirable, however, if a new name is only similar to one of the names the model has learnt about, then retrieved information will be false and might cause quality and social concerns as mentioned in *3b*.

Submission Instructions

You will submit this assignment on GradeScope as two submissions – one for **Assignment 5 [coding]** and another for **Assignment 5 [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 pretrain-finetune model and predictions: `vanilla.finetune.params`, `vanilla.pretrain.dev.predictions`, `vanilla.pretrain.test.predictions`

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

2. Run the `collect_submission.sh` script to produce your `assignment5.zip` file.
3. Upload your `assignment5.zip` file to GradeScope to **Assignment 5 [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 5 [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] Tay, Y., Bahri, D., Metzler, D., Juan, D.-C., Zhao, Z., and Zheng, C. Synthesizer: Rethinking self-attention in transformer models. *arXiv preprint arXiv:2005.00743* (2020).