Stanford CS224n Assignment 5

Aman Chadha / amanc@stanford.edu

February 20, 2021

1. Attention exploration (21 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.

(a) (2 points) **Copying in 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:

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

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

where α_i are frequently called the "attention weights", and the output $c \in \mathbb{R}^d$ is a correspondingly weighted average over the value vectors.

We'll first show that it's particularly simple for attention to "copy" a value vector to the output c. Describe (in one sentence) what properties of the inputs to the attention operation would result in the output c being approximately equal to v_j for some $j \in \{1, \ldots, n\}$. Specifically, what must be true about the query q, the values $\{v_1, \ldots, v_n\}$ and/or the keys $\{k_1, \ldots, k_n\}$?

Answer:

For $c \approx v_j$, α_j should be ≈ 1, while $\alpha_{i \neq j}$ should be ≈ 0. In other words, the dot product of q with k_j must be much larger than $\sum_{i \neq j} k_i^\top q$, so that the probability mass is concentrated on j, and thus,

$$c = \sum_{i=1}^{n} \frac{\exp\left(k_i^\top q\right)}{\sum_{i=1}^{n} \exp\left(k_j^\top q\right)} v_i \approx v_j$$

Now,

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

Since $\alpha_j \approx 1$,

$$\sum_{j=1}^{n} \exp(k_i^\top q) \approx \exp(k_j^\top q)$$

$$\implies \exp(k_i^\top q) \approx 0 \text{ for i} \neq \text{j}$$
$$\implies k_i^\top q \approx -\infty$$

(b) (4 points) **An average of two:** Consider a set of key vectors $\{k_1, \ldots, k_n\}$ where all key vectors are perpendicular, that is $k_i \perp k_j$ for all $i \neq j$. Let $\|k_i\| = 1$ for all i. Let $\{v_1, \ldots, v_n\}$ be a set of arbitrary value vectors. Let $v_a, v_b \in \{v_1, \ldots, v_n\}$ be two of the value vectors. Give an expression for a query vector q such that the output c is approximately equal to the average of v_a and v_b, that is, $\frac{1}{2}(v_a + v_b)$.[1] Note that you can reference the corresponding key vector of v_a and v_b as k_a and k_b.

Answer:

$c = \frac{v_a + v_b}{2} = \frac{1}{2}v_a + \frac{1}{2}v_b$. Comparing this with $c = \sum_{i=1}^{n} v_i \alpha_i$, we get,

$$\alpha_a = \frac{1}{2};\ \alpha_b = \frac{1}{2}$$

$$\begin{aligned} \implies \exp(k_a^\top q) &= \exp(k_b^\top q) \\ \implies k_a^\top q &= k_b^\top q \end{aligned}$$

Let q be $\beta\,(k_a + k_b)$, where β is a large number acting as a scalar multiple.

Now, $\alpha_i = \frac{\exp(k_i^\top q)}{\sum_{j=1}^{n} \exp(k_j^\top q)}$. Consider the numerator of this expression, $\exp(k_a^\top q)$. This can be simplified as,

$$\begin{aligned} \exp(k_a^\top q) &= \exp(k_a^\top\,(\beta\,(k_a + k_b))) \\ &= \exp(k_a^\top k_a \beta + k_a^\top k_b \beta) \\ &= \exp(\beta + k_a^\top k_b \beta)\ (\text{since } k_a^\top k_a = 1) \\ &= \exp(\beta + k_a^\top k_b \beta) \\ &= \exp(\beta)\ (\text{since } \beta \text{ is any large number}) \end{aligned}$$

Similarly, $\exp(k_b^\top q) = \exp(\beta)$.

Thus for all $j \in \{a, b\}$,

$$\exp(k_a^\top q) = \exp(k_b^\top q) = \exp(\beta)$$

and for all $j \notin \{a, b\}$, since the dot product similarity between the key and query vectors (since the output is simply average of v_a and v_b). As such, we have $\exp(k_j^\top q) = \exp(0) = 1$. Thus,

$$\alpha_a = \alpha_b = \frac{\exp(k_i^\top q)}{\sum_{j=1}^{n} \exp(k_j^\top q)} = \frac{\exp(\beta)}{(n-2) + 2\exp(\beta)}$$

which approaches $\frac{1}{2}$ as β grows and n is fixed.

This gives,

$c = \alpha_a v_a + \alpha_b v_b = \frac{1}{2}v_a + \frac{1}{2}v_b = \frac{v_a + v_b}{2}$.

Note that you may also instead let q have negative dot product with the $k_i, i \notin \{a, b\}$, that is, $q = \beta\left(k_a + k_b - \sum_{i \notin \{a,b\}} k_i\right)$ such that,

$$\alpha_a = \alpha_b = \frac{\mathrm{coxp}(\beta)}{(n \in 2)\exp(-\beta) + 2\exp(\beta)}$$

which converges faster to $\frac{1}{2}$ as a function of β.

[1] 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 $\{k_1, \dots, k_n\}$ that are now 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. 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 $\Sigma_i = \alpha I$, 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: The setting is effectively identical to Question 1 (b). Let $q = \beta\,(\mu_a + \mu_b)$ for large β. The variance being approximately 0 means $k_a \approx \mu_a$ and $k_b \approx \mu_b$, and the argument from Question 1 (b) holds.

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. Further, let $\Sigma_i = \alpha I$ for all $i \neq a$. When you sample $\{k_1, \dots, k_n\}$ 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: Upon sampling $\{k_1, \dots, k_n\}$ multiple times, the key vectors would show large variance since one key vector may be larger or smaller in norm than the others. This would imply that some samples c would be weighted much more towards v_a and some towards v_b. This is because each sample k_a would be approximately some constant multiple of μ_a; some will be more like $\frac{1}{2}\mu_a$, some more like $\frac{3}{2}\mu_a$. This implies that the values of $\exp(k_a^\top q)$ and $\exp(k_b^\top q)$ corresponding to two keys k_a and k_b would not be equal; one or the other will be greater for each sample, and so c will be weighted more towards one or the other.

(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 $\{k_1, \dots, k_n\}$ 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: Let $q_1 = \beta\mu_a$ and let $q_2 = \beta\mu_b$, and the argument from single-headed attention applies to each of the two cases.

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 $q_i^\top k_a < 0$.

Answer: For most samples, c_1 and c_2 will look like v_a and v_b, respectively, and so c will look like $\frac{1}{2}\,(v_a + v_b)$, without much variation. This is because even though there's variation in the value of k_a, as long as $k_a^\top q_1$ is positive, as β grows, the attention weight will be concentrated on a. Independently, there's no variation in k_b, and so c_2 will look like k_b without much variation.

(e) (7 points) **Key-Query-Value self-attention in neural networks:** So far, we've discussed attention as a function on a set of key vectors, a set of value vectors, and a query vector. In Transformers, we perform *self-attention*, which roughly means that we draw the keys, values, and queries from the same data. More precisely, let $\{x_1, \dots, x_n\}$ be a sequence of vectors in $\mathbb{R}^d$. Think of each x_i as representing word i in a sentence. One form of self-attention defines keys, queries, and values as follows. Let $V, K, Q \in \mathbb{R}^{d\times d}$ be parameter matrices. Then

$$v_i = Vx_i \quad i \in \{1, \dots, n\} \tag{3}$$

$$k_i = Kx_i \quad i \in \{1, \dots, n\} \tag{4}$$

$$q_i = Qx_i \quad i \in \{1, \dots, n\} \tag{5}$$

Then we get a context vector for each input i; we have $c_i = \sum_{j=1}^n \alpha_{ij} v_j$, where α_{ij} is defined as $\alpha_{ij} = \frac{\exp(k_j^\top q_i)}{\sum_{\ell=1}^n \exp(k_\ell^\top q_i)}$. Note that this is single-headed self-attention.

In this question, we'll show how key-value-query attention like this allows the network to use different aspects of the input vectors x_i in how it defines keys, queries, and values. Intuitively, this allows networks to choose different aspects of x_i to be the "content" (value vector) versus what it uses to determine "where to look" for content (keys and queries.)

i. (3 points) First, consider if we didn't have key-query-value attention. For keys, queries, and values we'll just use x_i; that is, $v_i = q_i = k_i = x_i$. We'll consider a specific set of x_i. In particular, let u_a, u_b, u_c, u_d be mutually orthogonal vectors in $\mathbb{R}^d$, each with equal norm $\|u_a\| = \|u_b\| = \|u_c\| = \|u_d\| = \beta$, where β is very large. Now, let our x_i be:

$$x_1 = u_d + u_b \tag{6}$$

$$x_2 = u_a \tag{7}$$

$$x_3 = u_c + u_b \tag{8}$$

If we perform self-attention with these vectors, what vector does c_2 approximate? Would it be possible for c_2 to approximate u_b by adding either u_d or u_c to x_2? Explain why or why not (either math or English is fine).

Answer: c_2 approximates u_a. It is not possible for c_2 to approximate u_b just by adding u_d or u_c to x_2. Adding u_d or u_c to x_2 would cause c_2 to incorporate either $u_d + u_b$ or $u_c + u_b$, but it's impossible to isolate u_b itself.

ii. (4 points) Now consider using key-query-value attention as we've defined it originally. Using the same definitions of x_1, x_2 and x_3 as in part i, specify matrices K, Q, V such that $c_2 \approx u_b$, and $c_1 \approx u_b - u_c$. There are many solutions to this problem, so it will be easier for you (and the graders), if you first find V such that $v_1 = u_b$ and $v_3 = u_b - u_c$, then work on Q and K. Some outer product properties may be helpful (as summarized in this footnote)[2].

Answer: Following the hint to start with $v_1 = u_b$ and $v_3 = u_b - u_c$, we know that by the definition of key-query-value self-attention, $Vx_1 = v_1 = u_b$ and $Vx_3 = v_3 = u_b - u_c$. Using the properties of outer products (as given in the footnote), we can set $V = \beta^{-2} u_b u_b^T$ to make it true that $Vx_1 = u_b$. On an intuitive level, we must choose an outer product such that the first term is the vector we want (in v_1), while the second term is the vector we have (in x_1). Here is a more detailed explanation of why this works:

From eq. 3,

$$Vx_1 = v_1$$

[2] For orthogonal vectors $u, v, w \in \mathbb{R}^d$, the outer product $uv^\top$ is a matrix in $\mathbb{R}^{d\times d}$, and $(uv^\top)v = u(v^\top v) = u\|v\|_2^2$, and $(uv^\top)w = u(v^\top w) = u * 0$. (The last equality is because v and w are orthogonal.)

Substituting,

$$\beta^{-2}\left(u_b u_b^\top\right)\left(u_d + u_b\right) = u_b$$

Distributing terms,

$$\beta^{-2}\left[u_b\left(u_b^\top u_d\right) + u_b\left(u_b^\top u_b\right)\right] = u_b$$

Using the property of the dot product,

$$\beta^{-2}\left[0 + u_b\left\|u_b\right\|_2^2\right] = u_b$$

Using the definition of β,

$$\beta^{-2}\left[0 + u_b\beta^2\right] = u_b$$

Canceling β,

$$u_b = u_b$$

We can then add a $-u_c u_c^\top$ term to V to cover the v_3 case as well. This yields $V = \beta^{-2}\left(u_b u_b^\top - u_c u_c^\top\right)$. This can be verified to work for v_3 as shown above.

Now that we have V, we can set about finding Q and K such that $c_1 \approx u_b - u_c$ and $c_2 \approx u_b$. To get $c_1 \approx v_3$, we must have $\alpha_{13} \approx 1$ and $\alpha_{1j\neq 3} \approx 0$. To accomplish this, $k_3^\top q_1$ should be much larger than $k_{j\neq 3}^\top q_1$. By similar reasoning, to get $c_2 \approx v_1$, $k_1^\top q_2$ should be much larger than $k_{j\neq 1}^\top q_2$. To make things simple, let us accomplish that by setting $k_1 = q_2, k_2 = 0$ and $k_3 = q_1$. We also know that $k_1 \neq k_3$. To satisfy these constraints, we'll need Q and K to pick out two unique terms from our x_1, x_2 and x_3. Looking at the structure of x_1, x_2 and x_3, we must use u_d and u_c. Let us pick $K = u_d u_d^\top + u_c u_c^\top$ and $Q = u_d u_a^\top + u_c u_d^\top$. It can be shown that these satisfy our constraints (though note that K and Q are interchangeable).

Thus we have $Q = u_d u_a^\top + u_c u_d^\top$, $V = \beta^{-2}\left(u_b u_b^\top - u_c u_c^\top\right)$, $K = u_d u_d^\top + u_c u_c^\top$.

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-birth place 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/birth place 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 birth-place 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).**

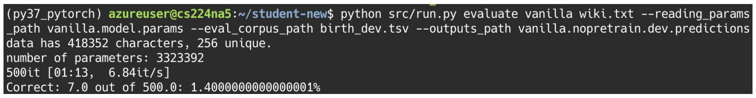
Train your model on `wiki_places_train.tsv`, and evaluate on `birth_dev.tsv`. Specifically, you should now be able to run the following three commands:

```
# 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
```

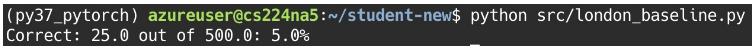
Training will take less than 10 minutes (on Azure). Report your model's accuracy on the dev set (as printed by the second command above). Don't be surprised if it is well below 10%; we will be digging into why in Part 3.



```
(py37_pytorch) azureuser@cs224na5:~/student-new$ 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
data has 418352 characters, 256 unique.
number of parameters: 3323392
500it [01:13,  6.84it/s]
Correct: 7.0 out of 500.0: 1.4000000000000001%
```

Accuracy: `1.4%`

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 your write-up. You should be able to leverage existing code such that the file is only a few lines long.



```
(py37_pytorch) azureuser@cs224na5:~/student-new$ python src/london_baseline.py
Correct: 25.0 out of 500.0: 5.0%
```

Accuracy: `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 `CharCorruptionDatas` 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 your whether 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 birth-place 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 three four commands: (Don't be concerned if the loss appears to plateau in the middle of pretraining; it will eventually go back down.)

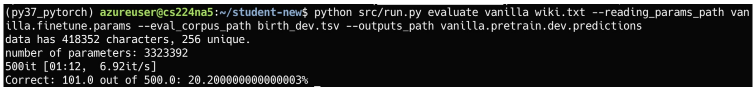
```
# Pretrain the model
python src/run.py pretrain vanilla wiki.txt \
      --writing_params_path vanilla.pretrain.params

# 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.



```
(py37_pytorch) azureuser@cs224na5:~/student-new$ python src/run.py evaluate vanilla wiki.txt --reading_params_path van
illa.finetune.params --eval_corpus_path birth_dev.tsv --outputs_path vanilla.pretrain.dev.predictions
data has 418352 characters, 256 unique.
number of parameters: 3323392
500it [01:12,  6.92it/s]
Correct: 101.0 out of 500.0: 20.200000000000003% _
```

Accuracy: `20.2%`

(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

$$Y_i = \text{softmax}\left(\frac{(XQ_i)(XK_i)^\top}{\sqrt{d/h}}\right)(XV_i) \tag{9}$$

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:

$$Y = [Y_1; \ldots; Y_h]A \tag{10}$$

were $A \in \mathbb{R}^{d \times d}$ 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 9, implement the following in `SynthesizerAttention`:

$$Y_i = \text{softmax}\big(\text{ReLU}(XA_i + b_1)B_i + b_2\big)(XV_i), \tag{11}$$

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.

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
```

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

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

```
# 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 birth-place prediction on `birth_dev.tsv` after pretraining and fine-tuning.

Answer:

i. (8 points) We'll score your model as to whether it gets at least 5% accuracy on the held-out test set; we'll get the model's predictions on the held-out test set using the bash script above.

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:

- The term $(XQ_i)\,(XK_i)^\top$ in key-query-value self-attention is an $\ell \times \ell$ matrix of attention scores, computed as all pairs of dot products between word embeddings. It thus computes scores for a query that are dependent on the values of the other vectors in the input thereby accounting for pairwise interactions between words. These interactions account for pairwise attention which are missing in equation for *synthesizer* self-attention, which results in inability of synthesizer self-attention to compute pairwise interactions between words.
- In other words, with key-query-value attention, we can say "word x, attend to words that have property y." Synthesizer attention on the other hand can only tell a word to attend to indices, based on the content of that word − so, it can't perform a function like "word x, attend to words that have property y."

3. Considerations in pretrained knowledge (5 points)

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

Answer:

- The pretrained model allows us to achieve better performance because it has learned both the low-level (semantic) and high-level (syntactic) features of language on a relatively larger dataset and then transfer learned the task at hand by fine-tuning on a smaller dataset. The non-pretrained model only has access to a small dataset and thus is limited in it's learning of language features beyond basic grammar/syntax/linguistic information, while the pretrained dataset contains specific world knowledge that the finetuning dataset lacks.

- For instance, pretraining, with some probability, masks out the name of a person while providing the birth place, or masks out the birth place while providing the name – this teaches the model to associate the names with the birthplaces. At finetuning time, this information can be accessed, since it has been encoded in the parameters (which are initialized to the pretrained model's). Without pretraining, there's no way for the model to for the model to have any knowledge of the birth places of people that weren't in the finetuning training set, so it can't get above a simple heuristic baseline.

(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 reasons why this indeterminacy of model behavior may cause concern for such applications.

Answer:

There is a large space of possible reasons indeterminacy of model behavior may cause concern for user-facing systems that involve pretrained NLP components, such as:

1. Users will always get outputs that look valid (if the user doesn't know the real answer) and so won't be able to perform quality estimation themselves (like one sometimes can when, e.g., a translation seems nonsensical). As such, system designers wouldn't have a way of filtering outputs for low-confidence predictions. This means we cannot even incorporate other techniques (say, another algorithm or simply, a human-in-the-loop) if the model's prediction is wrong since we cannot discern low-confidence predictions.
2. Models will not indicate that they simply do not know the birth place of a person (unlike a relational database or similar, which will return that the knowledge is not contained in it). This means the system cannot indicate a question is unanswerable. Since we cannot ascertain whether the model retrieved the correct birth place, or made up an incorrect birth place, we do not know when to trust and when not to trust the model. Once users realize the system can output plausible but incorrect answers, they may stop trusting the system, therefore making it useless.
3. Made up answers could be biased or offensive.
4. There is little avenue for recourse if users believe an answer is wrong, as it's impossible to determine the reasoning of the model is retrieving some gold standard knowledge (in which case the user's request to change the knowledge should be rejected), or just making up something (in which case the user's request to change the knowledge should be granted).
5. If the system makes downstream decisions on the basis of the model's output, the system can go wrong with its decisions owing to a made up prediction that was incorrect. This would cause the system to do more harm than good.

(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.

Answer:

- The model is probably doing a closest-similarity match in terms of features and is looking up the birth place for the target person using another person whose name bears the closest resemblance in terms of the said features. As such, the model could use character-level phonetic-like (sound-like) information to make judgments about where a person was born based on how their name "sounds", likely leading to racist outputs. The model could also learn that certain names or types of names tend to be of people born in richer cities, leading to classist outputs that predict a birth place simply based on whether the names are like that of rich people or poorer people.

- This would be a concern because this would lead to a form of "aliasing" where another person's birth place might be incorrectly assigned to the target person.

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).