Practical 4: Self-Attention, Transformers, Pretraining

Deep Natural Language Processing Class, 2023

Report due date - 30.04.2023

The practical report (pdf + code) should be sent via Moodle/Pegaz with a standard due dates policy.

1 Transformers

We will start this practical from the analysis of self-attention mechanism and its benefits. Then, we will move to the pretraining idea. The final section touches on the problem of hallucination of pre-trained large language models. We recommend to read the original paper of Transformers while you are working on the practical.

This practical was greatly inspired by the CS224 class from Stanford University.

2 Attention exploration

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) (1.5 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 $\{v_1, ..., v_n\}, v_i \in \mathbb{R}^d$, and a set of key vectors $\{k_1, ..., k_n\}, k_i \in \mathbb{R}^d$, specified as follows:

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

$$\alpha_i = \frac{\exp(k_i^T q)}{\sum_{j=1}^{n} \exp(k_j^T q)},$$

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 will 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 $j \in \{1, ..., n\}$. Specifically, what must be true about the query q, the values $\{v_1, ..., v_n\}$ and/or the keys $\{k_1, ..., k_n\}$?

b) (2 points)

An average of two: Consider a set of key vectors $\{k_1, ..., k_n\}$ 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 $\{v_1, ..., v_n\}$ be a set of arbitrary value vectors. Let $v_a, v_b \in \{v_1, ..., v_n\}$ 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, $\frac{1}{2}(v_a + v_b)$. Note that you can reference the corresponding key vector of v_a and v_b as k_a and k_b.

c) (3 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 $\{k_1, ..., k_n\}$ 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; $\mu_i^T \mu_j = 0$ if $i \neq j$, and unit norm, $||\mu_u|| = 1$.

1. 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.

2. 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 an example, let us consider a covariance for item a as $\Sigma_a = \alpha I + \frac{1}{2}(\mu_a \mu_a^T)$ for vanishingly small α. Further, let $\Sigma_i = \alpha I$ for all $i \neq a$. When you sample $\{k_1, ..., k_n\}$ multiple times, and use the q vector that you defined in part 1., what qualitatively do you expect the vector c will look like for different samples?

d) (1.5 points)

Benefits of multi-headed attention. Let's consider a simple version of multi-headed attention which is identical to single-headed self-attention 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 $\{k_1, ..., k_n\}$ 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; $\mu_i^T \mu_j = 0$ if $i \neq j$, and unit norm, $||\mu_i|| = 1$.

1. 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 $\frac{1}{2}(v_a + v_b)$.

2. Assume that the covariance matrices are $\Sigma_a = \alpha I + \frac{1}{2}(\mu_a \mu_a^T)$ 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 $q_i^T k_a < 0$.

e) (4 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 $\{x_1, ..., x_n\}$ 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

$$v_i = V x_i \quad i \in \{1, ..., n\} \tag{1}$$
$$k_i = K x_i, \quad i \in \{1, ..., n\} \tag{2}$$
$$q_i = Q x_i, \quad i \in \{1, ..., n\} \tag{3}$$

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 $\alpha_{ij} = \frac{\exp(k_j^T q_i)}{\sum_{l=1}^{n} \exp(k_l^T q_i)}$. Note that this is a 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.)

1. 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 will 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:

$$x_1 = u_d + u_b \tag{4}$$
$$x_2 = u_a \tag{5}$$
$$x_3 = u_c + u_b \tag{6}$$

   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).

2. 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 1., 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 if you first find V such that $v_1 = u_b$ and $v_3 = u_b - u_c$, and then work on Q and K. Some outer product properties of matrix multiplications may be helpful.

3 Pretraining Transformer-based Generative Model

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

a)

In the `mingptdemo` folder there 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.

b)

Read through `NameDataset`, the 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 `mingptdemo` won't be changed or evaluated for this assignment. In `dataset.py`, you'll find 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: `python src/dataset.py namedata`, it'll load your `NameDataset` on the training set `birth_places_train.tsv` and print out a few examples.

c)

Take a look at `run.py`. It has some skeleton code specifying flags you will 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 will focus on the finetuning function, in the case without pretraining. Taking inspiration from the training code in the `mingptdemo/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 will 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". 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.

d) (3 points)

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

Training will should take 75 epochs. Report your model's accuracy on the dev set (as printed by the second command above). 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.

e) (8 points)

In the file `src/dataset.py`, implement the `__getitem__` function for the dataset class `CharCorruptionDataset`. Follow the instructions provided in the comments in `dataset.py`. Span corruption is explored in the T5 paper. It randomly selects spans of text in a document and replaces them with unique tokens (noising). Models take this noised text, and are required to output a pattern of each unique sentinel followed by the tokens that were replaced by that sentinel in the input. In this question, you will implement a simplification that only masks out a single sequence of characters. We will instantiate the `CharCorruptionDataset` with our own data, and draw examples from it. To help you debug, if you run the following code: `python src/dataset.py charcorruption` , it wil sample a few examples from your `CharCorruptionDataset` on the pretraining dataset `wiki.txt` and print them out for you.

f) (8 points)

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 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
```

Report the accuracy on the dev set (printed by the third command above). We expect the dev accuracy will be at least 10%.

g) (3 points)

As was discussed, the regular Transformer scales quadratically with the number of tokens L in the input sequence, which is prohibitively expensive for large L and precludes its usage in settings with limited computational resources even for moderate values of L. At the lecture we discussed two methods that approached this problem, i.e Linformer and Big Bird models.

Question 1: Explain in no more than 3 sentences the idea behind Linformer.

Question 2: Explain in no more than 3 sentences the idea behind BigBird.

Question 3: In what respects BigBird is as powerful and expressive as full-attention mechanisms? Explain in up to 3 sentences.

*h) (2 points)

The attention operation used in Transformer could be also approximated by feature maps (see this paper), as shown in Figure 1. Given query, keys and values matrices $\mathbf{Q}, \mathbf{K}, \mathbf{V} \in \mathbb{R}^{L \times d}$ where d is the hidden dimension (dimension of the latent representation), the original attention representation:

$$\mathbf{A} = \exp(\mathbf{Q}\mathbf{K}^t)/\sqrt{d}$$

is approximated by randomized variant (omitting here normalization):

$$\mathbf{A} = \exp(\mathbf{Q}'\mathbf{K}'^t).$$

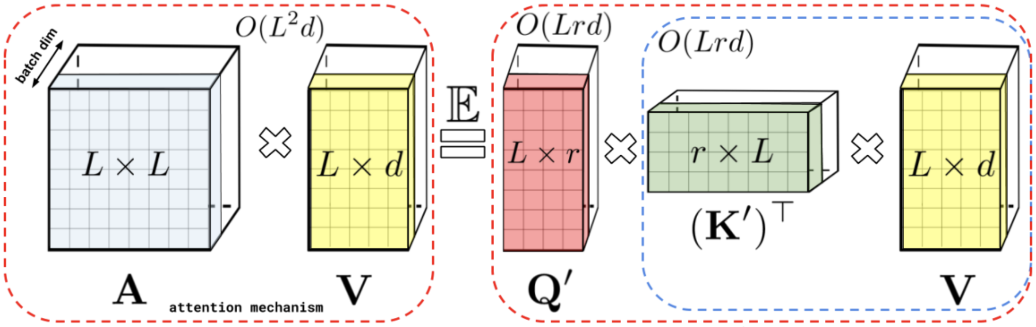

Figure 1: Approximation of the regular attention mechanism AV (before renormalization) via (random) feature maps. Dashed-blocks indicate order of computation with corresponding time complexities attached [Choromański et al., 2021].

Question 1: Explain in less than 3 sentences high-level idea of attention approximation from the paper to speed up computations? What are expected speedups?

Question 2: What problem might arise when the the softmax-kernel which defines regular attention matrix is approximated naively by trigonometric functions?

Question 3: How do the authors of the paper avoid the problem from question 2? What approximation can help here?

Question 4: What do they achieve by sampling orthogonal random features?

4 Pretrained Models as Source of Knowledge

a) (2 points)

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

b) (3 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.

c) (3 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.