COSC490LLMs

Homework 5: Transformers/Self-Attention

Name: Dustin O'Brien
Collaborators, if any: None
Sources used for your homework, if any: Currently None

This assignment focuses on basic understanding of Transformers for language modeling. We will also touch upon Tokenization.

Homework goals: After completing this homework, you should be comfortable with:

- thinking about Self-Attention module and implementing a simple Transformer language model.

1 Concepts, intuitions and big picture

Each question might have multiple correct answers. (Check all that apply).

1. Which of these types of models would you use for completing prompts with generated text?
   ☐ An encoder model
   ✓ A decoder model
   Answer:

2. Which of these types of models would you use for classifying text inputs according to certain labels?
   ✓ An encoder model
   ☐ A decoder model
   Answer:

3. To which of these tasks would you apply a many-to-one RNN architecture?
   ☐ Speech recognition (input an audio clip and output a transcript)
   ✓ Sentiment classification (input a piece of text and output a 0/1 to denote positive or negative sentiment)
   ✓ Gender recognition from speech (input an audio clip and output a label indicating the speaker's gender)
   Answer:

4. What possible source can the bias observed in a model have?
   ✓ The model is a fine-tuned version of a pretrained model and it picked up its bias from it.
   ✓ The data the model was trained on is biased.
   ✓ The metric the model was optimizing for is biased.
   Answer:

5. How many dimensions does the tensor output by a decoder Transformer model have, and what are they?
   ☐ 2: The sequence length and the batch size
   ☐ 2: The sequence length and the hidden size
   ✓ 3: The sequence length, the batch size, and the hidden size
   Answer:

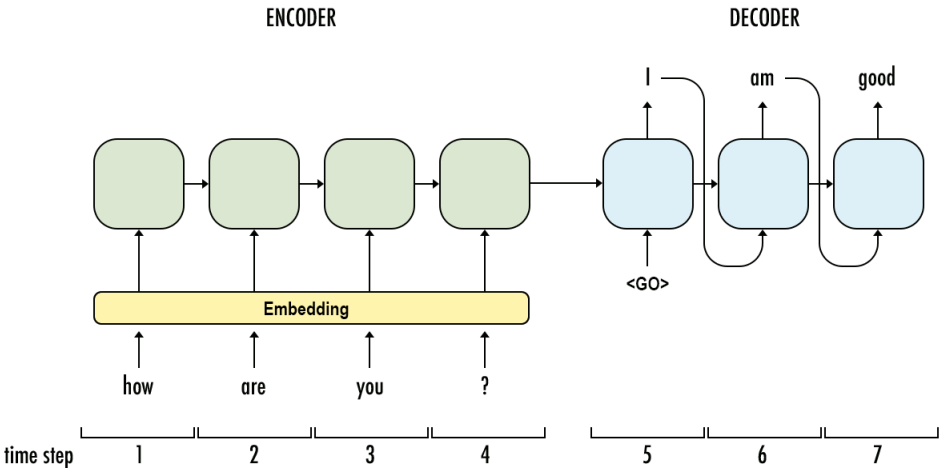
6. You are training an RNN language model. At the tth time step, what is the RNN doing? Choose the best answer.
   ☐ Estimating $P(y_1, y_2, ..., y_{t-1})$
   ☐ Estimating $P(y_1)$
   ✓ Estimating $P(y_t|y_1, y_2, ..., y_{t-1})$
   ☐ Estimating $P(y_t|y_1, y_2, ..., y_t)$
   Answer:

7. You are training an RNN, and find that your weights and activations are all taking on the value of `NaN` ("Not a Number"). Which of these is the most likely cause of this problem?
   ☐ Vanishing gradient problem.
   ✓ Exploding gradient problem.
   ☐ ReLU activation function g(.) used to compute g(z), where z is too large.
   ☐ Sigmoid activation function g(.) used to compute g(z), where z is too large.
   Answer:

8. You're done training an RNN language model. You're usig it to sample random sentences as follows. What are you doing at each time step t?
   ☐ (i) Use the probabilities output by the RNN to pick the highest probability word for that time-step as y_t. (ii) Then pass the ground-truth word from the training set to the next time-step.
   ☐ (i) Use the probabilities output by the RNN to randomly sample a chosen word for that time-step as y_t. (ii) Then pass the ground-truth word from the training set to the next time-step.
   ☐ (i) Use the probabilities output by the RNN to pick the highest probability word for that time-step as y_t. (ii) Then pass this selected word to the next time-step.
   ✓ (i) Use the probabilities output by the RNN to randomly sample a chosen word for that time-step as y_t. (ii) Then pass this selected word to the next time-step.
   Answer:

9. You have a pet crab whose mood is heavily dependent on the current and past few days' weather. You've collected data for the past 42 days on the weather, which you represent as a sequence as $x_1, ..., x_{42}$. You've also collected data on your crab's mood, which you represent as $y_1, ..., y_{42}$. You'd like to build a model to map from $x \rightarrow y$. Should you use a Unidirectional RNN or Bidirectional RNN for this problem?
☐ Bidirectional RNN, because this allows the prediction of mood on day t to consider more information.
☐ Bidirectional RNN, because this allows backpropagation to compute more accurate gradients.
✓ Unidirectional RNN, because the value of y_t depends only on $x_1, ..., x_t$ but not on $x_{t+1}, ..., x_{42}$ ☐ Unidirectional RNN, because the value of y_t depends only on x, and not other days' weather.
Answer:

10. Consider this encoder-decoder model. This model is a "conditional language model" in the sense that the



encoder portion (shown in green) is modeling the probability of the input sentence x.
☐ True ✓ False
Answer:

11. Compared to the RNN-LMs and fixed-window-LMs, we expect the attention-based LMs to have the greatest advantage when:
✓ The input sequence length is large.
☐ The input sequence length is small.
Answer:

12. What is a model head?
☐ A component of the base Transformer network that redirects tensors to their correct layers
☐ Also known as the self-attention mechanism, it adapts the representation of a token according to the other tokens of the sequence
✓ An additional component, usually made up of one or a few layers, to convert the transformer predictions to a task-specific output
Answer:

13. What are the techniques to be aware of when batching sequences of different lengths together?
✓ Truncating ☐ Returning tensors ✓ Padding ✓ Attention masking
Answer:

2 Self-Attention and Transformers

> In Transformers, patterns align and flow,
> Each token learns where meanings grow.
> Context builds with layers deep,
> A dance of words, a bond to keep.
>
> And on this day, as time moves on,
> Self-attention still makes us strong.
> Not just in models, cold and bright,
> But in our hearts, it sheds new light.
>
> To know ourselves, we must attend,
> To every thought, each truth we send.
> With care and kindness, let us see,
> The love within, wild and free.
>
> So as this day of life unfolds,
> Let self-reflection make us bold.
> May we embrace both heart and mind,
> And in their depths, true love we find.
>
> –ChatGPT Feb 25, 2025

Recall that the transformer architecture uses scaled dot-product attention to compute *attention weights*:

$$\boldsymbol{\alpha}^{(t)} = \text{softmax}\left(\frac{\mathbf{q}_t \mathbf{K}^\top}{\sqrt{h}}\right) \in [0,1]^n$$

The resulting embedding in the output of attention at position t are:

$$\text{Attention}(\mathbf{Q},\mathbf{K},\mathbf{V})_t = \sum_{t'=1}^{n} \alpha_{t'}^{(t)} \mathbf{v}_{t'} \in \mathbb{R}^{1\times h},$$

where $\boldsymbol{\alpha}^{(t)} = [\alpha_0^{(t)}, \ldots, \alpha_n^{(t)}]$. The same idea can be stated in a matrix form,

$$\text{Attention}(\mathbf{Q},\mathbf{K},\mathbf{V}) = \text{softmax}\left(\frac{\mathbf{Q}\mathbf{K}^\top}{\sqrt{h}}\right)\mathbf{V} \in \mathbb{R}^{n\times h}.$$

In the above equations:

- h is the hidden state dimension and n is the input sequence length;
- $\mathbf{X} \in \mathbb{R}^{n\times h}$ is the input to the attention;
- $\mathbf{x}_t \in \mathbb{R}^{1\times h}$ is the slice of $\mathbf{X}$ at position t, i.e. vector representation (embedding) of the input token at position t;
- $\mathbf{W}_q, \mathbf{W}_k, \mathbf{W}_v \in \mathbb{R}^{h\times h}$ are the projection matrices to build query, key and value representations;
- $\mathbf{Q} = \mathbf{X}\mathbf{W}_q \in \mathbb{R}^{n\times h}, \mathbf{K} = \mathbf{X}\mathbf{W}_k \in \mathbb{R}^{n\times h}, \mathbf{V} = \mathbf{X}\mathbf{W}_v \in \mathbb{R}^{n\times h}$ are the query, key and value representations;
- $\mathbf{q}_t = \mathbf{x}_t\mathbf{W}_q \in \mathbb{R}^{1\times h}$ is the slice of $\mathbf{Q}$ at position t. Similarly, $\mathbf{k}_t = \mathbf{x}_t\mathbf{W}_k \in \mathbb{R}^{1\times h}$ and $\mathbf{v}_t = \mathbf{x}_t\mathbf{W}_v \in \mathbb{R}^{1\times h}$.

Now answer the following questions:

2.1 Complexity

What is the computational complexity of self-attention layer in terms of n and h? In particular, show that the complexity of a self-attention layer at test-time scales quadratically with the sequence length n. Lastly, explain (no more than 5 sentences) why this can be computed efficiently on GPUs, despite being a quadratic function of sequence length n.

Answer: *By default the time complexity is $O(n^2d)$ despite being quadratic this attention is more parallelizable since it of the $O(n)$ is simplicity each layer of the output tensor that is computed by independent neural networks each of which can be trained on there own set of threads and don't require a sort of sequential input from eachother in essence assuming enough threads exist we can divide the time complex by n decreasing it to $O(nd)$ as an effective run time on well paralized threads*

2.2 Masking in Self-Attention

Suppose we are are using token-making objective to create a language generation model that uses self-attention. For example, suppose we want to mask $t = 3$ position. Then, it does not make sense for $\mathbf{q}_t : \forall t \in [0 \ldots n]$ to look at $\mathbf{k}_3$ and $\mathbf{v}_3$. Describe one way we can go about implementing such masking.

Answer: *we can do this by implementing something similar to causual mask except where instead it following a sort of diagonal the 3rd or tth column is a o vector and therefore when matrix multiplication is done all the multiplications in that column will = o and no information will be given to the model*

3 Programming

In this programming homework, we will

- implement our own GPT - Transformer-based language model.
- learn about how to train and evaluate the GPT LM on GPUs.

Skeleton Code and Structure: The code base for this homework can be found at MyClasses/Files under the `hw5` directory. Your task is to fill in the missing parts in the skeleton code, following the requirements, guidance, and tips provided in this pdf and the comments in the corresponding .py files. The code base has the following structure:

- `gpt` defines the Python module of the GPT LM
  - `bpe.py` creates a Byte-Pair Encoding tokenizer from scratch.
  - `model.py` implements our GPT model architecture from scratch.
  - `trainer.py` is the script for training, evaluating the GPT model, and visualizing the results.
  - `utils.py` defines helper functions.
- `main.py` provides the entry point to run your implementations of `gpt` module.
- `hw5.md` provides instructions on how to setup the environment and run each part of the homework in `main.py`.
- `generate.py` provides helper functions for sampling from GPT LMs.
- `data.py` convert WikiText Data into LM training data.

TODOs — Your tasks include 1) generate plots and/or write short answers based on the results of running the code; 2) fill in the blanks in the skeleton to complete the code. We will explicitly mark these plotting, written answer, and filling-in-the-blank tasks as **TODOs** in the following descriptions, as well as a `# TODO` at the corresponding blank in the code.

Submission: Your submission should contain two parts: 1) plots and short answers under the corresponding questions below; and 2) your completion of the skeleton code base, in a `.zip` file

3.1 A GPT Language Model from Scratch

3.1.1 Introduction: Self-attention and Transformer

What do BERT, RoBERTa, GPT4, ... all have in common? *Self-attention and Transformer-based architecture*! Transformer-based architectures, which are primarily used in modeling language understanding tasks, eschew the use of recurrence in neural networks (RNNs) and instead trust entirely on self-attention mechanisms to draw global dependencies between inputs and outputs.

Some useful documentation: As the architecture is so popular, there already exists a Pytorch module for `nn.Transformer` and a tutorial on how to use it for the next token prediction. However, we will implement it here ourselves, to get through to the smallest details.

There are of course many more tutorials out there about attention and Transformers. Below, we list a few that are worth exploring if you are interested in the topic and might want yet another perspective on the topic after this one:

- Transformer: A Novel Neural Network Architecture for Language Understanding (Jakob Uszkoreit, 2017) – The original Google blog post about the Transformer paper, focusing on the application in machine translation.
- The Illustrated Transformer (Jay Alammar, 2018) – A very popular and great blog post intuitively explaining the Transformer architecture with many nice visualizations.
- Attention? Attention! (Lilian Weng, 2018) – A nice blog post summarizing attention mechanisms in many domains including vision.

- Illustrated: Self-Attention (Raimi Karim, 2019) – A nice visualization of the steps of self-attention. Recommended going through if the explanation below is too abstract for you.
- The Transformer family (Lilian Weng, 2020) – A very detailed blog post reviewing more variants of Transformers besides the original one.

3.1.2 What is Attention?

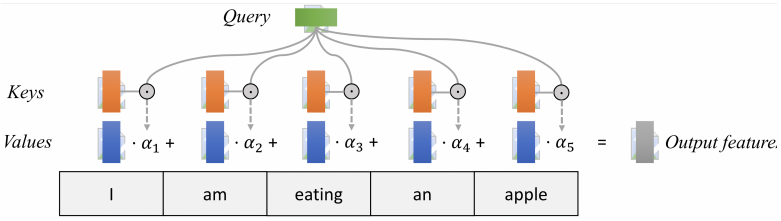
The attention mechanism describes a recent new group of layers in neural networks that has attracted a lot of interest in the past few years, especially in sequence tasks. There are a lot of different possible definitions of "attention" in the literature, but the one we will use here is the following: *the attention mechanism describes a weighted average of (sequence) elements with the weights dynamically computed based on an input query and elements' keys*. So what does this exactly mean? The goal is to take an average of the features of multiple elements. However, instead of weighting each element equally, we want to weight them depending on their actual values. In other words, we want to dynamically decide on which inputs we want to "attend" more than others. In particular, an attention mechanism has usually four parts we need to specify:

- **Query**: The query is a feature vector that describes what we are looking for in the sequence, i.e. what would we maybe want to pay attention to.
- **Keys**: For each input element, we have a key which is again a feature vector. This feature vector roughly describes what the element is "offering", or when it might be important. The keys should be designed such that we can identify the elements we want to pay attention to based on the query.
- **Values**: For each input element, we also have a value vector. This feature vector is the one we want to average over.
- **Score function**: To rate which elements we want to pay attention to, we need to specify a score function f_{attn}. The score function takes the query and a key as input, and outputs the score/attention weight of the query-key pair. It is usually implemented by simple similarity metrics like a dot product.

The weights of the average are calculated by a Softmax overall score function outputs. Hence, we assign those value vectors a higher weight whose corresponding key is most similar to the query. If we try to describe it with pseudo-math, we can write:

$$\alpha_i = \frac{\exp\left(f_{attn}\left(\text{key}_i, \text{query}\right)\right)}{\sum_j \exp\left(f_{attn}\left(\text{key}_j, \text{query}\right)\right)}, \qquad \text{out} = \sum_i \alpha_i \cdot \text{value}_i$$

Visually, we can show the attention over a sequence of words as follows. For every word, we have one key and one value vector. The query is compared to all keys with a score function (in this case the dot product) to determine the weights. The Softmax is not visualized for simplicity. Finally, the value vectors of all words are averaged using the attention weights.



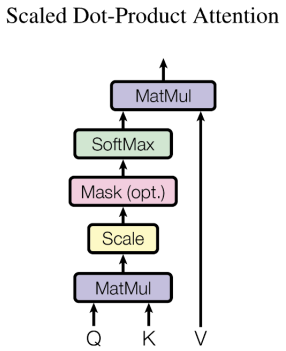
Most attention mechanisms differ in terms of what queries they use, how the key and value vectors are defined, and what score function is used. The attention applied inside the Transformer architecture is called **self-attention**. In self-attention, each sequence element provides a key, value, and query. For each element, we perform an attention layer where based on its query, we check the similarity of all sequence elements' keys and returned a different, averaged value vector for each element. We will now go into a bit more detail by first looking at the specific implementation of the attention mechanism which is in the Transformer case the scaled dot product attention.

3.1.3 Scaled Dot Product Attention

The core concept behind self-attention is the scaled dot product attention. Our goal is to have an attention mechanism with which any element in a sequence can attend to any other while still being efficient to compute. The dot product attention takes as input a set of queries $Q \in \mathbb{R}^{T \times d_k}$, keys $K \in \mathbb{R}^{T \times d_k}$ and values $V \in \mathbb{R}^{T \times d_v}$ where T is the sequence length, and d_k and d_v are the hidden dimension for queries/keys and values respectively. For simplicity, we neglect the batch dimension for now. The attention value from element i to j is based on its similarity of the query Q_i and key K_j, using the dot product as the similarity metric. In math, we calculate the dot product attention as follows:

$$\text{Attention}(Q, K, V) = \text{softmax}\left(\frac{QK^T}{\sqrt{d_k}}\right)V$$

The matrix multiplication QK^T performs the dot product for every possible pair of queries and keys, resulting in a matrix of the shape $T \times T$. Each row represents the attention logits for a specific element i to all other elements in the sequence. On these, we apply a Softmax and multiply with the value vector to obtain a weighted mean (the weights being determined by the attention). Another perspective on this attention mechanism offers the computation graph which is visualized below (figure credit: **?**). There is also a nice visualization of these values are computed here.



One aspect we haven't discussed yet is the scaling factor of $1/\sqrt{d_k}$. This scaling factor is crucial to maintain an appropriate variance of attention values after initialization. Remember that we initialize our layers to have equal variance throughout the model, and hence, Q and K might also have a variance close to 1. However, performing a dot product over two vectors with a variance σ results in a scalar having d_k-times higher variance:

$$q_i \sim \mathcal{N}(0, \sigma), k_i \sim \mathcal{N}(0, \sigma) \rightarrow \text{Var}\left(\sum_{i=1}^{d_k} q_i \cdot k_i\right) = \sigma \cdot d_k$$

If we do not scale down the variance back to σ, the softmax over the logits will already saturate to 1 for one random element and 0 for all others. The gradients through the softmax will be close to zero so we can't learn the parameters appropriately.

3.1.4 Multi-Head Attention

The scaled dot product attention allows a network to attend over a sequence. However, often there are multiple different aspects a sequence element wants to attend to, and a single weighted average is not a good option for it. This is why we extend the attention mechanisms to multiple heads, i.e. multiple different query-key-value triplets on the same features. Specifically, given a query, key, and value matrix, we transform those into h sub-queries, sub-keys, and sub-values, which we pass through the scaled dot product attention independently. Afterward, we concatenate the heads and combine them with a final weight matrix. Mathematically, we can express this operation as:

$$\text{Multihead}(Q, K, V) = \text{Concat}(\text{head}_1, ..., \text{head}_h)W^O \tag{1}$$

$$\text{where head}_i = \text{Attention}(QW_i^Q, KW_i^K, VW_i^V) \tag{2}$$

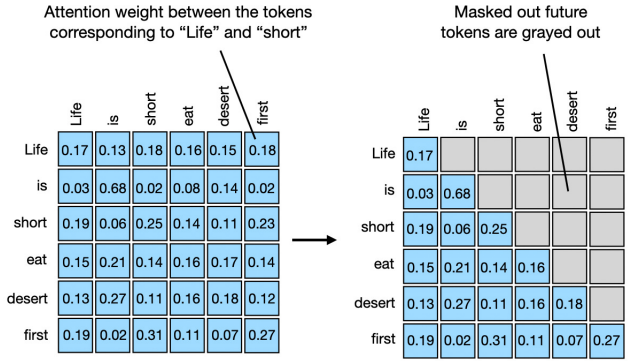
We refer to this as Multi-Head Attention layer with the learnable parameters $W^Q_{1...h} \in \mathbb{R}^{D \times d_k}$, $W^K_{1...h} \in \mathbb{R}^{D \times d_k}$, $W^V_{1...h} \in \mathbb{R}^{D \times d_v}$, and $W^O \in \mathbb{R}^{h \cdot d_k \times d_{out}}$ (D being the input dimension). Expressed in a computational graph, we can visualize it below (figure credit: **?**).

3.1.5 Causal Attention Masking

For GPT-like LMs, we train the model to generate one token at a time from left to right (autoregressive). When predicting the next token, the autoregressive setup constraints the model to only "see", i.e. attend to, the preceding tokens on the left-hand side. For instance, given a sample training text: "Life is short eat dessert first", the self-attention should be applied only within the tokens on the left-hand side of the $\rightarrow$, in order to predict the token on the right-hand side:

- "Life" $\rightarrow$ "is"
- "Life is" $\rightarrow$ "short"
- "Life is short" $\rightarrow$ "eat"
- "Life is short eat" $\rightarrow$ "dessert"
- "Life is short eat dessert" $\rightarrow$ "first"

To achieve this in practice, one straightforward approach is to mask out future tokens for each timestamp by applying a mask to the attention weight matrix (causal attention masking). For the example above, we can apply the mask:



TODOs: Read and complete the missing lines in the `__init__` function of the `CausalSelfAttention` class in `gpt/model.py`, create the causal attention mask.
Hint: Follow the requirements and hints specified in the code comments.

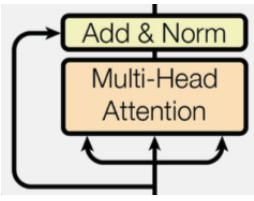
3.1.6 Putting it Together (so far): Multi-Head Self Attention with Causal Masking

Now that we have delved into the details of all important components, it's time to build the causal self-attention block for our own GPT!

TODOs: Read and complete the missing lines in the `forward` function of the `CausalSelfAttention` class in `gpt/model.py`.
Hint: Follow the requirements and hints specified in the code comments.

3.1.7 Residual connection and normalization

Notice that there is a residual connection and normalization on top of the multi-head attention layer.



Taking as input x, it is first passed through a Multi-Head Attention block as we have implemented above. The output is added to the original input using a residual connection, and we apply a consecutive Layer Normalization on the sum. Overall, it calculates $\text{LayerNorm}(x + \text{Multihead}(x, x, x))$ (x being Q, K and V input to the attention layer). The residual connection is crucial in the Transformer architecture for two reasons:

1. Similar to ResNets, Transformers are designed to be very deep. Some models contain more than 24 blocks in the encoder. Hence, the residual connections are crucial for enabling a smooth gradient flow through the model.

2. Without the residual connection, the information about the original sequence is lost. Remember that the Multi-Head Attention layer ignores the position of elements in a sequence, and can only learn it based on the input features. Removing the residual connections would mean that this information is lost after the first attention layer (after initialization), and with a randomly initialized query and key vector, the output vectors for position i have no relation to its original input. All outputs of the attention are likely to represent similar/same information, and there is no chance for the model to distinguish which information came from which input element. An alternative option to residual connection would be to fix at least one head to focus on its original input, but this is very inefficient and does not have the benefit of the improved gradient flow.

The Layer Normalization also plays an important role in the Transformer architecture as it enables faster training and provides small regularization. Additionally, it ensures that the features are in a similar magnitude among the elements in the sequence. We are not using Batch Normalization because it depends on the batch size which is often small with Transformers (they require a lot of GPU memory), and BatchNorm has shown to perform particularly badly in language as the features of words tend to have a much higher variance (there are many, very rare words which need to be considered for a good distribution estimate).

3.1.8 Putting it Together: Transformer Architecture

Originally, the Transformer model was designed for machine translation. Hence, it has an encoder-decoder structure where the encoder takes as input the sentence in the original language and generates an attention-based representation. *This is different from our GPT implementation, which is a decoder-only architecture.* On the other hand, the decoder attends to the encoded information and generates the translated sentence in an auto-regressive manner, as in a standard RNN. The full Transformer architecture looks as follows (figure credit: **?**):

The encoder consists of N identical blocks. These blocks contain Self-Attention, Layer Normalization, and Residual connections. Additionally, a small fully connected feed-forward network is added to the model, which is applied to each position separately and identically. Specifically, the model uses a Linear→ReLU→Linear MLP. The full transformation including the residual connection can be expressed as:

$$\text{FFN}(x) = \max(0, xW_1 + b_1)W_2 + b_2 \tag{3}$$

$$x = \text{LayerNorm}(x + \text{FFN}(x)) \tag{4}$$

This MLP adds extra complexity to the model and allows transformations on each sequence element separately. You can imagine as this allows the model to "post-process" the new information added by the previous Multi-Head Attention, and prepare it for the next attention block. Usually, the inner dimension of the MLP is 2-8$\times$ larger than d_{model}, i.e. the dimension of the original input x. The general advantage of a wider layer instead of a narrow, multi-layer MLP is the faster, parallelizable execution.

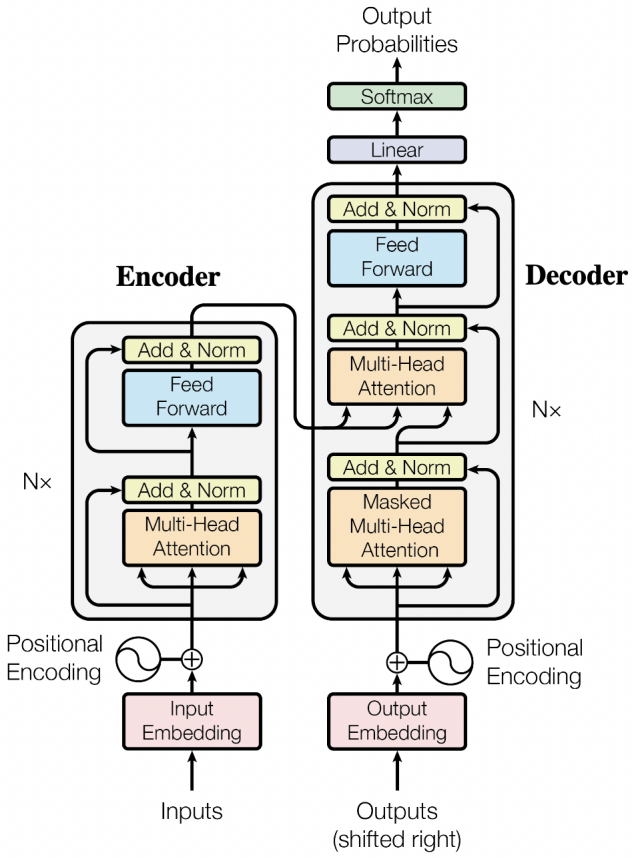
3.1.9 Train and Evaluate GPT LM on GPUs

Now let's train our GPT LM!

From this homework on, we will be using GPUs. **Why GPUs?** A crucial feature of PyTorch is the support of GPUs, short for Graphics Processing Unit. A GPU can perform many thousands of small operations in parallel, making it very well suitable for performing large matrix operations in neural networks. When comparing GPUs to CPUs, we can list the following main differences:

CPUs and GPUs have both different advantages and disadvantages, which is why many computers contain both components and use them for different tasks. In case you are not familiar with GPUs, you can read up more details in this NVIDIA blog post or here.

GPUs can accelerate the training of your network up to a factor of 100 which is essential for large neural networks. PyTorch implements a lot of functionality to support GPUs (mostly those of NVIDIA due to the libraries CUDA and cuDNN).



To get a more direct comparison, let's run our model for 50 training steps with both CPU and GPU.

TODOs Run the `cpu_gpu_comparison` in `main.py`, and fill in the running time below (will be printed out on the console):
Answer:
CPU - Trainer Time Elapsed: 50.22 seconds
GPU - Trainer Time Elapsed: 3.34 seconds

Finally, let's run the full training and evaluation on GPU.

TODOs Run the `gpu_full_run` function in `main.py` to train the GPT language model, paste the two generated plots here. This function will also sample from the trained GPT LM. paste the completion to the prefix after the plots.
your plots
Answer:
iter 20000: validation loss 0.14691 validation ppl 1.159
———— Trainer Time Elapsed: 1228.19s ————

The best perks of living on the east east best best best best best best best best best best best best best best best best best best best best beststicksIOIO best FreeBSD of living on the

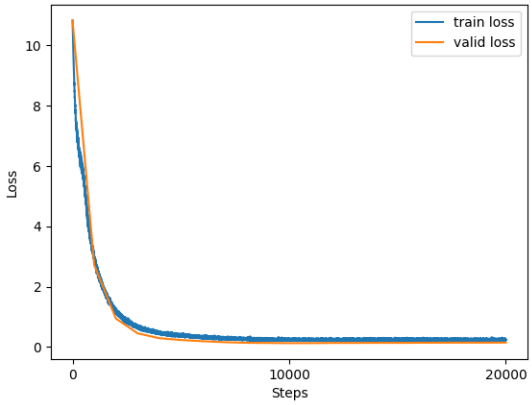
---

It is a truth universally acknowledged report vision vision too too acknowledged is a truth universally

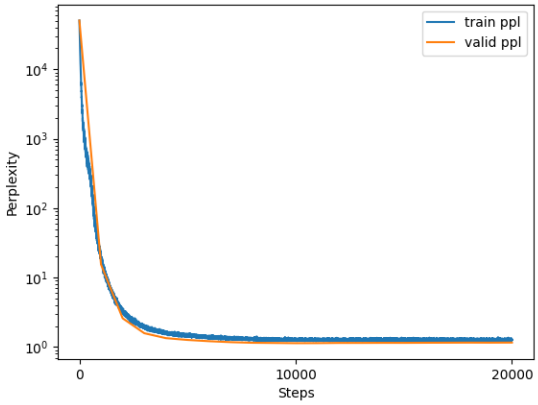
---

The best way to learn chance chance chance best way to

According to the latest research research research research research to from from from from from from from from from from from



(a) train and dev loss of the GPT LM



(b) train and dev ppl of the MLP LM

Figure 1: loss and ppl of the GPT LM

from from from from from from research research research research from to the latest