Assignment 2: Attention and Transformers

Instructor: Yejin Choi *CSE 517/447 Win 24*

Due at 11:59pm PT, Feb 9, 2024
100 pt for 447 (+ 5 extra credit) / 110 pt for 517, 15% towards the final grade

In this assignment, you will explore the behavior of the attention operation, implement the attention module from scratch within a transformer, and become familiar with fine-tuning a Huggingface model end-to-end.

You will submit both your **code** and **writeup** (as PDF) via Gradescope. Remember to **specify your collaborators** (including AI tools like ChatGPT) and **how they contribute** to the completion of your assignment at the beginning of your writeup. If you work on the assignment independently, please specify so, too. **NOT properly acknowledging your collaborators will result in -2 % of your overall score on this assignment.** Please adhere to the assignment collaboration policy specified on the course website.

Required Deliverables

- **Code Notebook**: Each question has an associated Python notebook. You need to submit the notebooks for all of §1-3. Please download all three notebooks as Python files (`.py`) and submit them in Gradescope.
- **Write-up**:
  - For written answers and open-ended reports, produce a single PDF for §1-3 and submit it in Gradescope. We recommend using Overleaf to typeset your answers in LaTeX, but other legible typed formats are acceptable. We do not accept hand-written solutions because grading hand-written reports is incredibly challenging.
  - The suggested page limit for each section is to make sure the reports do not get too long. We would not penalize shorter reports as long as they contain all necessary grading components. Longer reports do not directly result in higher scores. On the other hand, concise and on-point reports will be more favorable.

Acknowledgement

This assignment is primarily designed by Yegor Kuznetsov, Liwei Jiang, Jaehun Jung, with invaluable feedback from Alisa Liu, Melanie Sclar, Gary Liu, and Taylor Sorensen.

1 Understanding Attentions (20%)

As an introduction to this assignment, you will interact with the attention operation and play with its capabilities/behavior in a simplified context. Our goal for this problem is to impart a basic intuitive understanding of the mechanisms involved in attention.

Notebook: We have designed this question with the following Python notebook: A2S1.ipynb.

Deliverables:

1. **Coding Exercises:** You should complete the code blocks denoted by `TODO:` in the Python notebook. To submit your code, download your notebook as a Python file (`A2S1.py`).
2. **Write-up:** Your report for §1 should be **no more than three pages**. However, you will most likely be able to answer all questions within two pages.

1.1 Background on Self-Attention

Multi-head scaled dot product self-attention is the core building block of all transformer architectures. It can be confusing for people seeing it for the first time, despite the motivations behind the design choices being intuitive. For this problem, we will ignore scaling and multiple heads to focus on developing an intuition for the behavior of dot product self-attention.

Recall that the attention operation requires computing three matrices Q, K, V.

- Q is a set of *query* vectors $q_i \in \mathbb{R}^d$.
- K is a set of *key* vectors $k_i \in \mathbb{R}^d$.
- V is a set of *value* vectors $v_i \in \mathbb{R}^d$.

We can simplify this by considering a **single** query vector. Each part within this question will clarify if we're asking for a single query vector q or a query matrix Q.

Dot product self-attention follows the following steps:

1. Pairwise similarities are computed to create pre-softmax attention scores A:

$$\alpha_{i,j} = q_i k_j \qquad\qquad A = QK^T$$

2. Softmax is applied across the last dimension as a normalization to produce the attention matrix A':

$$\alpha'_{i,j} = \frac{\exp(\alpha_{i,j})}{\sum_j \exp(\alpha_{i,j})} \qquad\qquad A' = \text{softmax(A)}$$

3. Each output vector $b_i \in \mathbb{R}^d$ is computed as a weighted sum of values using attention.

$$b_i = \sum_j \alpha'_{i,j} v_j \qquad\qquad O = A'V$$

Notes for the Following Exercises:

- Most of the operations in this problem cannot be represented *exactly*, and there may be small deviations between your crafted vs. target vectors or matrices. This is acceptable and expected. Solutions within a **0.05** error (as reported in the notebook) will receive full credit.
- There are many different possible solutions to the following problems. And there may be shortcuts to getting an answer without applying attention computation (like random guessing). However, in this exercise, we ask you to devise a solution by thinking of the working mechanism of attention. Your rationales of how you derive your solution should reflect such an understanding. **Rationales that do not involve any aspects of the internal mechanisms of attention are not eligible for points.**
- Note that your solutions for this exercise don't have to be a generalizable solution that handles all kinds of K, V. They can be *ad hoc* to this specific example. But you're also welcome to propose generalizable solutions. You only need to give one solution for each question in this exercise.
- Please answer the following questions in your **write-up**.

1.2 Selection via Attention (10%)

Suppose we have the following K and V matrices with $d = 3$ and $n = 4$, produced from 4 tokens. K consists of 4 vectors $k_i \in \mathbb{R}^3$, and V consists of 4 vectors $v_i \in \mathbb{R}^3$.

$$K = \begin{bmatrix} 0.47 & 0.65 & 0.60 \\ 0.64 & 0.50 & -0.59 \\ -0.03 & -0.48 & -0.88 \\ 0.43 & -0.83 & 0.35 \end{bmatrix} \quad V = \begin{bmatrix} -0.07 & -0.88 & 0.47 \\ 0.37 & -0.93 & -0.07 \\ -0.25 & -0.75 & 0.61 \\ 0.94 & 0.20 & 0.28 \end{bmatrix}$$

We will ask you to define a few *query* vectors that satisfy some conditions. For any requested *query* vectors or matrices (q or Q), you may provide either numerical values, or an expression in terms of K, V or the vectors contained within them. In this exercise, vectors such as v_i are 0-indexed.

When we ask you to provide a *query* that does something, this means that the output vectors from performing attention using the *query* you provide along with the given K, V would result in that operation having been performed.

Hint: For one of the versions of the solutions, you may find it useful to define a "large number," S for finding a solution! Also, you can try to think of what matrix A do you need. But again, there are many different possible solutions.

1. **Define a *query* vector q ($\in \mathbb{R}^3$) to "select" (i.e., return) the first *value* vector v_0. Briefly explain how you get your solution.**

   We want to find the $q = [q_0, q_1, q_2]$ such that:

   $$O = A'V = \text{softmax}(A)V = \text{softmax}(qK^\top)V = v_0$$

   In other words, we want $\text{softmax}(qK^\top)V = v_0$. Since v_0 is the first row of V, we want $\text{softmax}(qK^\top)$ to be something like $[1, 0, 0, 0]$ so that when it's multiplied by V, we get v_0.

   In order for $\text{softmax}(qK^\top)$ to be $[1, 0, 0, 0]$, we want the first element of $\text{softmax}(qK^\top)$ to be much larger than the rest. This is because the softmax function will make the largest element even larger, and the rest even smaller.

   Since $qK^\top$ is simply a dot product between q and each row of K, we want the dot product between q and k_0 (the first row of K) to be much larger than the rest. To maximize the dot product between q and k_0, we just want q to be parallel to k_0, and we can scale q by a large number S to make the dot product even larger. So we can set $q = S \cdot k_0$, where $S = 1,000,000$, for example.

In this example, this yields $\boxed{q = S \cdot k_0 = [470000, 650000, 600000]}$, and we get $O = A'V = v_0$.

2. **Define a *query* matrix Q ($\in \mathbb{R}^{4\times3}$) which results in an identity mapping – select all the *value* vectors. Briefly explain how you get your solution.**

   We want to find $Q = [q_0, q_1, q_2, q_3]^\top$ where $q_i \in \mathbb{R}^3$ is the i-th row of Q, such that:

   $$O = A'V = \text{softmax}(A)V = \text{softmax}(QK^\top)V = V$$

   In other words, we want $\text{softmax}(QK^\top)V$ to be V. For this to be true, we need $\text{softmax}(QK^\top)$ to be the identity matrix $\mathbf{I}^{4\times4}$, that way when it's multiplied by V (which lives in $\mathbb{R}^{4\times3}$), we get V.

   To do this, we will follow a similar strategy to 1.2.1. We want the vectors in Q (the rows) to be parallel to the rows of K, and we can scale each row of Q by a large number S to make the dot product yield a large value on the diagonal and smaller values everywhere else. Applying softmax to this matrix will then yield the identity matrix.

   In this example, this yields $\boxed{Q = S \cdot K}$, where $S = 1{,}000{,}000$, and we get $O = A'V = V$.

3. **What does attention's ability to copy / select from input tokens when creating outputs imply for language modeling? In other words, why might this be desirable? (1-3 sentences)**

   Attention's ability to copy / select from input tokens when creating outputs is like having a spotlight that the model can shine on specific words it has already seen when it's trying to figure out what word comes next. This is really useful because it helps the model remember and use important words from earlier in the sentence, which helps it make better guesses. It's like a human who can remember context from much earlier in the conversation and use it to understand the current sentence better.

1.3 Averaging via Attention (10%)

Continue using the same K, V matrices for this section.

Hint: You can try to think of what matrix A do you need. But again, there are many different possible solutions.

1. **Define a *query* vector q ($\in \mathbb{R}^3$) which averages all the *value* vectors. Briefly explain how you get your solution.**

   We want to find the $q = [q_0, q_1, q_2]$ such that:

   $$O = A'V = \text{softmax}(A)V = \text{softmax}(qK^\top)V = \frac{v_0 + v_1 + v_2 + v_3}{4}$$

   In other words, we want $\text{softmax}(qK^\top)V$ to be the average of the rows of V. Note that the formula $\frac{v_0+v_1+v_2+v_3}{4}$ can be rewritten to give us a better sense of what our $qK^\top$ should be:

   $$\frac{v_0 + v_1 + v_2 + v_3}{4} = 0.25 \cdot (v_0 + v_1 + v_2 + v_3) = 0.25v_0 + 0.25v_1 + 0.25v_2 + 0.25v_3$$

   Thus, we want $\text{softmax}(qK^\top)$ to be $[0.25, 0.25, 0.25, 0.25]$, so that when it's multiplied by V, we get the average of the rows of V.

   In order for $\text{softmax}(qK^\top)$ to be $[0.25, 0.25, 0.25, 0.25]$, we want the dot product between q and each row of K to be the same for each row of K. To do this, we can set q to be the average of the rows of K. The intuition behind this is that if K has rows that are not orthogonal, taking the average might "smooth out" the differences and lead to the dot products being equal. However, this is a heuristic and might not work if K has rows that are very different in magnitude or direction.

Anyways, we can set $\boxed{q = \frac{1}{4}\sum_{i=0}^{3} k_i}$, where $k_i \in \mathbb{R}^3$ is the i-th row of K, and we get $O = A'V = \frac{v_0+v_1+v_2+v_3}{4}$.

In this case, $\boxed{q = [0.3775, -0.0400, -0.1300]}$.

2. **Define a *query* vector q ($\in \mathbb{R}^3$) which averages the first two *value* vectors. Briefly explain how you get your solution.**

   We want to find $q = [q_0, q_1, q_2]$ such that:

$$O = A'V = \text{softmax}(A)V = \text{softmax}(qK^\top)V = \frac{v_0 + v_1}{2} = 0.5v_0 + 0.5v_1 + 0v_2 + 0v_3$$

   This implies that we want $\text{softmax}(qK^\top) = [0.5, 0.5, 0, 0]$. That way, when we multiply it by V, we are just averaging the first two value vectors.

   To do this, we can set q to be the average of the first two rows of K. This is because the dot product between q and each row of K will be larger for the first two rows of K than the last two rows of K, and applying softmax to this will yield a vector that is close to $[0.5, 0.5, 0, 0]$, which is what we want. We will still need to scale q by a (smaller) large number S to make the dot product large enough to yield the desired softmax.

   Empirically, I chose $S = 10$ as I got a very low error with this value, but this was determined through trial and error. In this case, $\boxed{q = S \cdot \sum_{i=0}^{1} k_i}$, where $k_i \in \mathbb{R}^3$ is the i-th row of K and $S = 10$, and we get $O = A'V = \frac{v_0+v_1}{2}$ (to an error of 0.0029).

3. **What does the ability to average / aggregate (in some cases selectively) imply for language modeling? In other words, why might this be desirable? (1-3 sentences)**

   The ability to average / aggregate (in some cases selectively) is desirable for language modeling because it allows the model to combine information from multiple words in the sentence to make a prediction. This is useful because it allows the model to understand the sentence as a whole, rather than just looking at individual words. I guess the human analog would be being able to distinguish ideas from long-ranging sentences, rather than each word as a separate entity.

1.4 Interactions within Attention (10% for 517, 5% extra credit for 447)

Unlike the tasks listed in §1.2 and §1.3, averaging just the first two *value* vectors is not reliably possible (i.e. generalizable). Without changing your *query* q from §1.3.2 or the rest of K, change only the third *key* vector k_2 for each of the following cases.

1. **Come up with a replacement for only the third *key* vector k_2 such that the result of attention with the same unchanged *query* q from §1.3.2 averages the first three *value* vectors. Briefly explain how you get your solution.**

   Given that the query q from 1.3.2 averages the first two value vectors, to extend this to the first three value vectors without altering q, we need to change k_2 such that it makes the dot product with q equivalent to that of the first two keys. Since q is the average of the first two keys k_0 and k_1, $\boxed{\text{we could set } k_2 \text{ to this same value}}$. This will result in the dot product of q and k_2 being similar to the dot product of q with k_0 and k_1. Thus, the softmax will treat them similarly, and the output will be the average of the first three value vectors.

2. **Come up with a replacement for only the third *key* vector k_2 such that the result of attention with the same unchanged *query* q from §1.3.2 returns the third *value* vector v_2. However, there is the condition that k_2 should have length $= 1$. This is not usually a restriction in attention, but is only for this problem. Briefly explain how you get your solution.**

   For the softmax to return v_2, we need the dot product of q with k_2 to be significantly higher than its dot product with other key vectors. Given the condition that k_2 should have length $= 1$, we can achieve this by aligning k_2 in the direction of q and scaling it to have a unit length. This means k_2 would become the normalized version of q, that is, $\boxed{k_2 = \frac{q}{|||q|||}}$.

3. **Why is altering k_2 able to impact an output which previously only considered the first two tokens? (2-4 sentences)**

   Altering k_2 affects the output because the attention mechanism relies on the dot products between the query and key vectors to determine the weighting of the value vectors. Changing k_2 alters these dot products and thus changes the weights assigned to the corresponding value vectors during the attention calculation. This shows the sensitivity of attention to the input key vectors, and how small changes can lead to significant differences in the output.

2 Building Your Own Mini Transformer (40%)

In this part, you will implement multi-head scaled dot product self-attention and use it to train a tiny decoder-only transformer using a modified fork of Andrej Karpathy's minGPT implementation of a GPT-style transformer. Finally, you will run a small experiment of your choosing and write a mini-report summarizing your experiment and interpreting your results.

Notebook: We have designed this question with the following Python notebook: A2S2.ipynb

Deliverables:

1. **Coding Exercises (§2.1):** You should complete the code blocks denoted by `TODO:` in the Python notebook. To submit your code, download your notebook as a Python file (`A2S2.py`). **We will only grade the codes you wrote for §2.1. §2.2 codes are not graded but will be useful for you to write the report.**

2. **Write-up (§2.2):** Your report for §2.2 should be **no more than five pages**. **We will only grade the write-up for §2.2.**

2.1 Implementing Attention from Scratch (20%)

We have provided a very decomposed scaffold for implementing attention, and after filling in the implementation details, you should check your implementation against the one built into PyTorch. The intent for this first part is to assist with *understanding* implementations of attention, primarily for working with research code.

Useful resources that may help with this section include, but are not limited to:

- "Lecture 5: Attention & Transformers" slides.
- PyTorch's documentation for `torch.nn.functional.scaled_dot_product_attention`: lacks multi-head attention, but is otherwise most excellent.
- The attention implementation in `mingpt/model.py` in the original `minGPT` repository.

Code style: This exercise has four steps, matched with corresponding functions in the notebook. This style of excessively decomposing and separating out details would normally be bad design but is done this way here to provide a step-by-step scaffold.

Code efficiency: Attention is a completely vectorizable operation. In order to make it fast, avoid using any loops whatsoever. We will not grade down for using loops in your implementation, but it would likely make the solution far slower and more complicated in most cases. **In the staff solution, each function except for `self_attention()` is a single line of code.**

Coding exercises (in the Python notebook): Here, we provide high-level explanations of what each function does in the Python notebook. **In the notebook, you will complete code blocks denoted by `TODO:`.**

Step 0: Set up the projections for attention.

- `init_qkv_proj()`: You do NOT need to implement this function.

  Initialize the projection matrices W_Q, W_K, W_V. Each of these can be defined as an `nn.Linear` from `n_embd` features to `n_embd` features. Attention does allow some of these to be different, but this particular model (i.e., minGPT) has the same output features dimension for all three. Do NOT disable bias. This function is passed into the modified model on initialization, and so does not need to be used in your implementation of `self_attention()`.

  This function should return a tuple of three PyTorch Modules. Internally, your W_Q, W_K, W_V will be used to project the input tokens a into the Q, K, V. Each row of Q is one of the q_i.

- `self_attention()`: As you work on Step 1-3, integrate the functions from each section into this function and test the behaviors you expect to work.

  Stitch together all the required functions as you work on this section within this function. Start with a minimal implementation of scaled dot product attention without causal masking or multiple heads.

  As you gradually transform it into a complete causal multi-head scaled dot-product self-attention operation, there are several provided cells comparing your implementation with pytorch's built-in implementation `multi_head_attention_forward` with various features enabled. If you see close to 0 error relative to the expected output, it's extremely likely that your implementation is correct.

  While it is allowed, we do not recommend looking into the internals of `multi_head_attention_forward` as it is extremely optimized for performance and features over readability, and is several hundred lines of confusing variables and various forms of input handling. Instead, see the above listed "useful resources."

Step 1: Implement the core components of attention.

- `pairwise_similarities()`: Implement this function.

  Dot product attention is computed via the dot product between each query and each key. Computing the dot product for all $\alpha_{i,j} = k_j q_i$ is equivalent to multiplying the matrices with a transpose. One possible matrix representation for this operation is $A = QK^T$.

  Hint: PyTorch's default way to transpose a matrix fails with more than two dimensions, which we have due to the batch dimension. As such, you can specify to torch.transpose the last two dimensions.

- `attn_scaled()`: Implement this function.
  Attention is defined with a scale factor on the pre-softmax scores. This factor is calculated as follows:

$$\frac{1}{\sqrt{n\_embd/n\_head}}$$

- `attn_softmax()`: Implement this function.
  A now contains an unnormalized "relevancy" score from each token to each other token. Attention involves a `softmax` along one dimension. There are multiple ways to implement this, but we recommend taking a look at torch.nn.functional.softmax. You will have to specify along which dimension the softmax is done, but we leave figuring that out to you. This step will give us the scaled and normalized attention A'.

- `compute_outputs()`: Implement this function.
  Recall that we compute output for each word or token as weighted sum of values, weighed by attention. Once again, we can actually express this as a matrix multiplication $O = A'V$.

Test 1: Once you implement functions from Step 1 and integrate them in `self_attention()`, we have provided a cell for you to test this portion of your implementation.

Step 2: Implement causal masking for language modeling.

This requires preventing tokens from attending to tokens in the future via a triangular mask. Enable causal language modeling when the `causal` **flag in the parameters of** `self_attention` is set to True.

- `make_causal_mask()`: Implement this function.
  The causal mask used in a language model is a matrix used to mask out elements in the attention matrix. Each token is allowed to attend to itself and to all previous tokens. This leads the causal mask to be a triangular matrix containing ones for valid attention and zeros when attention would go backwards in the sequence. We suggest looking into documentation of torch.tril.
- `apply_causal_mask()`: Implement this function.
  Entries in the attention matrix can be masked out by overwriting entries with $-\infty$ before the softmax. Make sure it's clear why this results in the desired masking behavior; consider why it doesn't work to mask attention entries to 0 after the softmax. You may find torch.where helpful, though there are many other ways to implement this part.

Test 2: Test causal masking in your attention implementation. Also, make sure your changes didn't break the first test.

Step 3: Implement multi-head attention.

Split and reshape each of Q, K, V at the start, and merge the heads back together for the output.

In order to match `multi_head_attention_forward`, we omit the transformation we would usually apply at the end from this function. Therefore when it is used later, an output projection needs to be applied to the attention's output. This is already implemented in our modified minGPT.

- `split_heads_qkv()`: You do NOT need to implement this function.
  We have provided a very short utility function for applying `split_heads` to all three of Q, K, V. No implementation is necessary for this function, and you may choose not to use it.
- `split_heads()`: Implement this function.
  Before splitting into multiple heads, each of Q, K, V has shape `(B, n_tok, n_embd)`, where `B` is the batch size, `n_tok` is the sequence length, `n_embd` is the embedding dimensionality. Note that PyTorch's matrix multiplication is batched – only multiplying using the last two dimensions. Thus, the matrix multiplication still works with the additional batch dimension of Q, K, V.[1]

  Since we want all heads to do attention separately, we want the head dimension to be before the last two dimensions. A sensible shape for this would be `(B, n_heads, n_tok, n_embd_per_head)`, where `n_heads` is the number of heads and `n_embd_per_head` is the embedding dimensionality of each head (`n_embd` / `n_heads`). A single reshaping cannot convert from a tensor of
- `merge_heads()`: Implement this function.
  When merging, you want to reverse/undo the operations done for splitting.

  First, transpose from `(B, n_heads, n_tok, n_embd_per_head)` to `(B, n_tok, n_heads, n_embd_per_head)`.

  Then, reshape from `(B, n_tok, n_heads, n_embd_per_head)` to `(B, n_tok, n_embd)`.

  Note that you can let PyTorch infer one dimension's size if you enter -1 for it.

Test 3: All three testing cells should result in matching outputs now.

[1] If you're interested, see details on batched matrix multiplication in `https://pytorch.org/docs/stable/generated/torch.bmm.html`.

2.2 Experiment with Your Implementation of Attention (20%)

Now that you have a working implementation of *Causal Multi-Head Scaled Dot Product Self-Attention*,[2], you will experiment with the mini transformer that you built out and write a report on your exploration.

Here's a list of suggested exploration topics/directions for modifying attention:

- Change dot-product to a different, custom operation which also takes two vectors and returns a number.
- Why do we need all three of (query, key, value)? See what happens if the projection used to create them is shared between two (or all three). Which versions of this are capable of learning anything, and which ones aren't?
- minGPT uses learned positional embeddings, and we truncate all sequences to 100 tokens during training, so it's expected to do poorly with tokens outside that limit when tested. Implement a mathematical positional encoding (e.g., sinusoidal positional encoding) and see if it makes it work properly with longer sequences.
- What actually happens if we try the naive masking approach of setting attention values to 0 after the softmax instead of setting to $-\infty$ before the softmax?
- Currently, W_Q, W_K, W_V are simply projection matrices. Why not make them more interesting, like turning each into a small fully connected network? Alternatively, what if we put a small nonlinearity on one of them – does it cause anything interesting?
- (If you want a bigger challenge for no extra credit) Replicate the main change(s) to attention in Attention Free Transformer
- (If you want a bigger challenge for no extra credit) Replicate the main change(s) to attention in Fast Attention

Your explorations could be based on the above suggested directions, but you are also welcome to explore other exciting aspects of the internal mechanisms of attention. You are also not limited to only exploring changes to the implementation of attention – you can fork our provided git repo and change any internal details of the overall transformer model. By changing the cloned repository to your fork, you can have persistent changes to any part of the architecture.

This exploration is fairly open-ended, but we want you to focus on ablating, changing, or otherwise testing/evaluating some aspect of attention or transformers as a whole. For most experiment choices, you are encouraged to report on the training performance and validation perplexity and use it to evaluate/interpret the model's behavior. You can also consider a qualitative inspection; for example, if your chosen experiment completely ruins performance, are there any particular patterns in the sampled text?

Overall, try to develop interesting ways to make changes to attention and transformers, and try NOT to simply toy with hyperparameters like `n_heads`.

Write a mini-report for the results of your experimentation. The report does NOT have to be "research quality" or answer completely new questions – simply pick any aspect you're interested in learning more about, and use this as an opportunity to explore the internals of a transformer, even if your experiment is simply breaking part of the architecture. Please limit your report to **no more than five pages.** Write for an audience familiar with NLP but not the internals of this question.

Specifically, we will be looking for and grading on the following aspects:

1. Explain the setup of the experiment, including the motivating idea and relevant background.

[2] This isn't an official name – just wanted to stress what all the components are.

2. Report your results. This will likely include graph(s) and/or table(s), as well as explanations of results. We expect at least one relevant graph/table which shows your results at a glance.
3. Interpret your results. What does this mean for using attention in language modeling? Do your results support some aspect of the design of attention?

§2.2 will be graded entirely on your report. If your experiment is simply varying a hyperparameter (such as embedding dimension, number of heads, or scaling factor) it is still possible to get full points, but the report will be held to significantly higher standards (and thus we encourage you to come up with experiments beyond this level of modification). On the other hand, if your report replicates the core changes from a research paper which (non-trivially) modifies the attention mechanism or something else that's fundamental to transformers, we will be considerably more lenient in our evaluation of your report.

Starter code: We have provided some starter code for you to train the tiny GPT model using the building blocks you implemented in §2.1. Specifically, we set it up to train on the same data used in Assignment 1 Question 1, with a similar tokenization scheme. Although we will not ask questions about the given training setup, take some time to read through it and make sure you understand it. Looking through minGPT code could also help you understanding how different components of your model are connected.

While §2.1 was completely guided, the code for §2.2 is merely a start; you are encouraged to rewrite any and all portions of the code we provide as you see fit. We make no guarantees about the provided code and training code for this question has not been tuned in any way beyond getting it to run, and is nowhere near optimal. Part of your task for §2.2 is to work past this detail and improve it for your experiment if necessary.

Code efficiency: Even the smallest usable transformer configuration in minGPT is painfully slow on the CPU available in a Colab notebook. As such, for experiments in §2.2, you should switch the notebook to use GPU – everything is already configured to train the model on GPU if it is available. The train runner will print whether it is using cpu or cuda. Training on cpu will take hours; training on cuda will take minutes.

Setup of the experiment.

Motivating idea. I chose to experiment by trying the naive masking approach of setting attention values to 0 *after* the softmax instead of setting to $-\infty$ *before* the softmax.

Causal attention masking is done to prevent the model from "cheating" by looking at future tokens when making predictions. The traditional way to do this is to set the attention scores (above the main diagonal) to $-\infty$ before applying the softmax across the rows. This way, the softmax will assign a probability of 0 to the future tokens, and the model will not use them when making predictions.

However, I was curious to see what would happen if we tried to set the attention scores to 0 after the softmax. Conceptually, this approach is about the same, although could be somewhat easier to understand. It's simple to think about the attention scores and "just set the ones you don't care about to 0", instead of having to think about $-\infty$ and the softmax.

Relevant background. The reason we set the attention scores to $-\infty$ before the softmax is to prevent the model from using future tokens when making predictions. The softmax function will use what its given: if we provide it with the vector $[2.2, 1.2, -\infty, -\infty]$, it will assign a high probability to the first two tokens and a probability of 0 to the last two tokens. That's because the $-\infty$ doesn't contribute to the sum in the denominator of the softmax, and so the softmax will assign a probability of 0 to the last two tokens.

However, if we provide softmax with the attention scores $[-20, -12, 20, 25]$, it will assign a *lower* probability to the first two tokens and a really high probability to the last two tokens. This is because the last two tokens have much higher scores than the first two tokens.

Suppose the result of applying softmax to this vector was $[0.01, 0.02, 0.48, 0.49]$. In the naive masking algorithm, we would actually end up masking the last two values here, and end up with $[0.01, 0.02, 0, 0]$. Whereas if we had done the original masking algorithm, we would have ended up with something like $[0.4, 0.6, 0, 0]$. Clearly, these are very different distributions! Which means that the model will be using very different information when making predictions.

Hypothesis. My hypothesis was that I actually had no idea what would happen: good or bad. So, I divided up what I thought could happen into two categories: (1) What could go wrong, and (2) What could go right, and tried to use that as a compass to guide my expectations.

What could go wrong.

- The naive approach might disrupt the intended probability distribution of the sequence. This would lead to the model using the wrong information when making predictions, and would likely lead to a higher perplexity.
- Maybe this would lead to a model that performs well on the training data, but does not generalize well to new data. That's because this approach might not let a model properly handle the causal relationships in the sequence, which are obviously critical.

What could go right.

- The naive approach might actually act as a form of **regularization**. By directly setting the attention scores to 0, regardless of their true value, we might be avoiding overconfidence in certain predictions and end up preventing overfitting to the training data.
- Perhaps the model learns an alternate pathway for prediction, which would lead to a more robust model that does not overly rely on specific tokens.

- Depending on the dataset, the naive approach might actually work better than the traditional approach. For example, if the dataset has a lot of noise, the traditional approach might be too aggressive in masking out future tokens while applying too much probability mass to current and past tokens, whereas the naive approach (which could act as a regularizer) might actually work better.

Results.

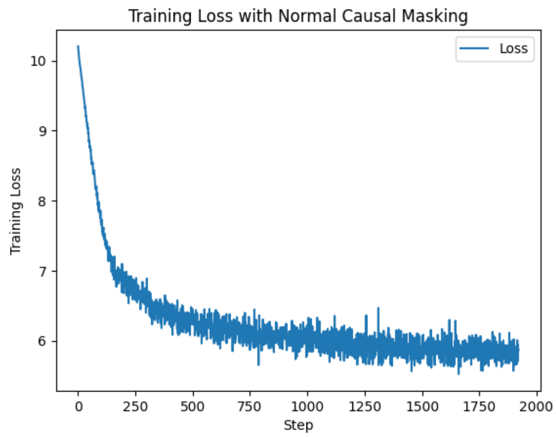

Figure 1: Training loss with normal causal masking.

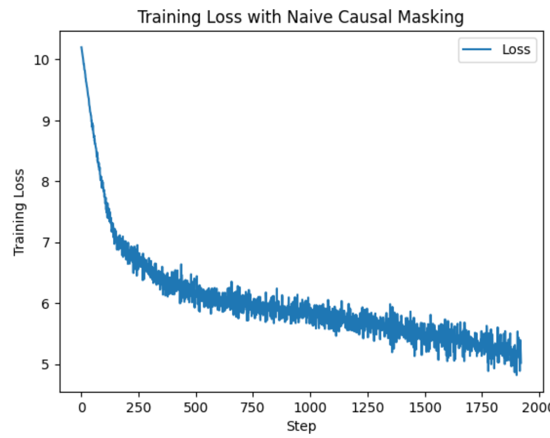

Figure 2: Training loss with naive causal masking.

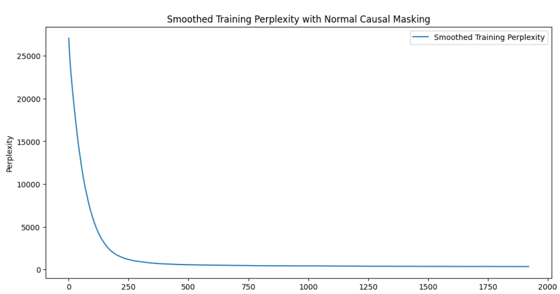

Figure 3: Smoothed training perplexity with normal causal masking.

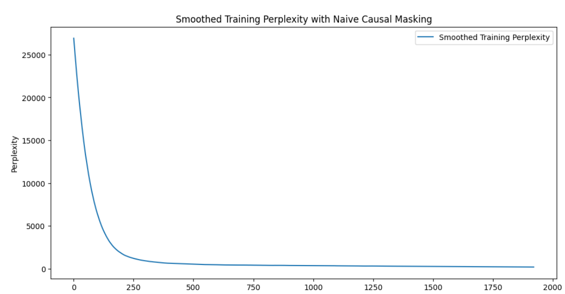

Figure 4: Smoothed training perplexity with naive causal masking.

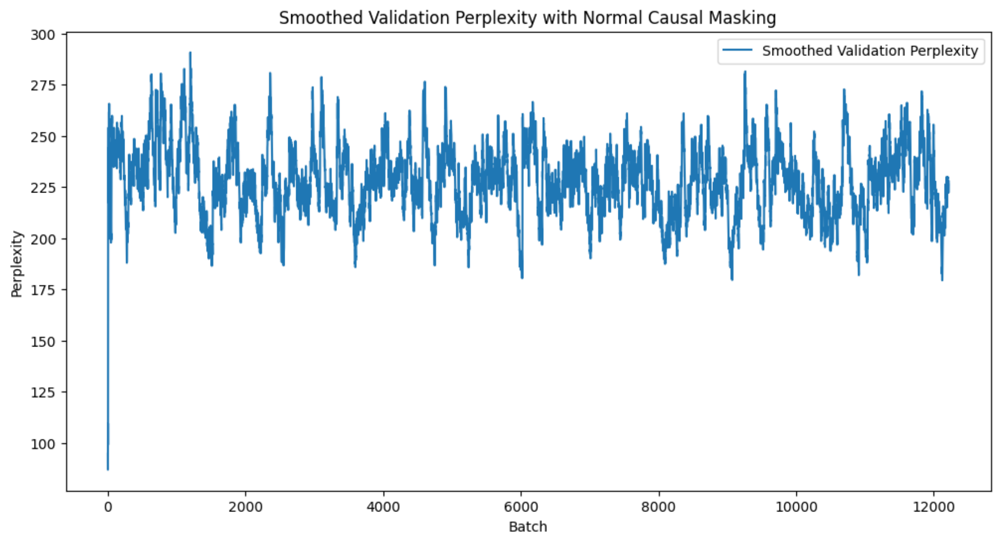

Figure 5: Smoothed validation perplexity with normal causal masking.

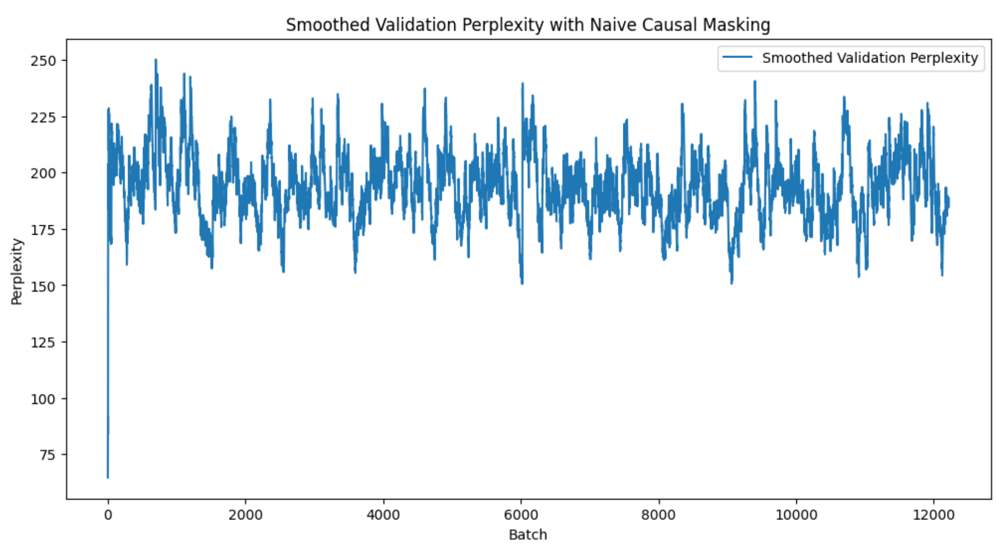

Figure 6: Smoothed validation perplexity with naive causal masking.

Masking Approach	**Training Perplexity**	**Validation Perplexity**
Normal Causal Masking	949.59	228.15
Naive Causal Masking	894.52	228.15

Table 1: Average Training and Validation Perplexities

Interpretation of results.
The results were quite interesting! The training loss, training perplexity, and validation perplexity were all *lower* with naive causal masking than with normal causal masking. This was quite surprising to me, as I had expected the naive approach to perform worse than the traditional approach.

My guess for this is twofold:

- The naive approach might have acted as a form of regularization. By directly setting the attention scores to 0, regardless of their true value, we might have been avoiding overconfidence in certain predictions and ended up preventing overfitting to the training data.
- The naive approach might have actually worked better than the traditional approach, but only on this specific dataset. For example, if the dataset has a lot of noise, the traditional approach might be too aggressive in masking out future tokens while applying too much probability mass to current and past tokens, whereas the naive approach (which could act as a regularizer) might actually work better. Then again, I can't be sure of this without further experimentation on different datasets.

My gut tells me that the naive approach is not as good as the normal approach, I mean it's literally called the "naive" approach. But the results are the results, and I can't argue with them. I would like to do further experimentation on different datasets to see if the naive approach is actually better than the normal approach, or if it was just a fluke on this specific dataset. I would also like to experiment with different hyperparameters and see if the naive approach is still better than the normal approach, as I just used the same hyperparameters for both approaches in this experiment.

Edited code. Plotting functions have been omitted.

```python
    """
    assert Q.shape == K.shape == V.shape
    B, n_tok, n_embd = Q.size()

    # TODO: Step 3 -- split heads.
    if n_heads > 1:
        Q, K, V = split_heads_qkv(Q, K, V, n_heads)

    # TODO: Step 1 -- calculate raw attetion.
    # Hint: you need two lines here.
    A = pairwise_similarities(Q, K)
    A = attn_scaled(A, n_embd, n_heads)

    # # TODO: Step 2 -- create and apply the causal mask to attention.
    # if causal:
    #     mask = make_causal_mask(n_tok)
    #     A = apply_causal_mask(mask, A)

    # TODO: Step 1 -- softmax the raw attention and use it to get outputs.
    # Hint: you need two lines here.
    A = attn_softmax(A)

    # Experiment code: Apply the mask after softmax, naively setting things to 0.
    if causal:
        mask = make_causal_mask(n_tok)
        A = apply_naive_causal_mask(mask, A)

    y = compute_outputs(A, V)

    # TODO: Step 3 -- merge heads.
    if n_heads > 1:
        y = merge_heads(y)

    # output should have the same shape as input
    assert y.shape == (B, n_tok, n_embd)
    return y
```

Figure 7: The edited code in the self_attention function. Notice the causal masking occurring after the softmax function.

```python
def make_causal_mask(n_tok:int):
    """
    Create a mask matrix that masks future context for the attention.
    :return: A mask matrix which is a tensor of shape (n_tok, n_tok)
    """
    # Hint: In order for it to run properly later, you'll need to put `.to(DEVICE)` at
    # the end of your expression for this. This will not be relevant until section 2.2.

    # Create a n_tok by n_tok matrix of booleans (all True initially). This mask
    # is n_tok by n_tok as it's a mask for A: which is the pairwise attention
    # score for the ith token with the jth token.
    booleans = torch.ones(n_tok, n_tok, dtype=bool)

    # We want everything on and below the main diagonal to be True and
    # everything else should be False. By default, all elements on and below
    # the main diagonal are retained in torch.tril().
    return torch.tril(booleans).to(DEVICE)

def apply_causal_mask(mask, A):
    """
    Apply mask to attention.
    :return: A masked attention matrix.
    """
    # A is the attention scores. mask is the mask matrix. Because masked_fill_
    # will fill in values marked as True, we apply logical_not() to it to
    # coerce the values we want to be -infinity (everything above
    # the main diagonal) to be True, so they are the values that get filled in.
    # A.masked_fill_(torch.logical_not(mask), float('-inf'))
    # return A

    return torch.where(mask, A, float('-inf'))

def apply_naive_causal_mask(mask, A):
    return torch.where(mask, A, 0)
```

Figure 8: The new naive causal masking function. This was called instead in this experiment to compare the regular approach with the naive one.

3 HuggingFace (40%)

In this part of the assignment, you will complete a codebase used to finetune a pretrained language model (RoBERTa-base) end-to-end on a sentiment analysis task (SST-2) using the convenient infrastructure and tools provided by HuggingFace. With your implementation, you'll write a short report to answer questions of your implementation, and analyze the behaviors of your trained model. **We will grade both the code and the report.**

Notebook: You will use the following Python notebook for this exercise: A2S3.ipynb.

Deliverables:

1. **Coding Exercises:** You should complete the code blocks denoted by `TODO:` in the Python notebook. To submit your code, download your notebook as a Python file (`A2S3.py`).
2. **Write-up:** Your report for this part should be **no more than four pages**, and should answer all questions listed in §3.2.

3.1 Background

The pretrain-then-finetune pipeline is a common recipe for large language model (LLM) applications and research. Essentially, the *pretraining* step leverages the vast amount of raw data gathered through the internet to train a model in an unsupervised way to capture the underlying knowledge patterns and structure of language. Next, the *finetuning* step customizes pretrained models to specific applications and tasks, building on top of their existing capabilities. In this exercise, you will complete a codebase that is used to finetune a `RoBERTa-base` model on a sentiment analysis task (SST-2) using the HuggingFace library.

Pretrained Model `RoBERTa` is a Transformers-based bidirectional encoder-only language model pretrained with masked language modeling objective[3]. Encoder-only models like `RoBERTa` are useful for encoding input sequences for classification tasks, rather than open-ended text generation. In this exercise, you will finetune a pretrained `RoBERTa-base` model that is already provided for you by HuggingFace.

Dataset/Task The Stanford Sentiment Treebank (SST-2) is a corpus labeled for the sentence-level sentiment analysis task, consisting of 11,855 single sentences extracted from movie reviews.[4] Each data point in the dataset consists of an input sentence and an output sentiment label (1 for positive and 0 for negative).

3.2 Finetuning Your Own `RoBERTa` Classifier

The provided Python notebook breaks the task down into the following six steps:

- Step 0: Preperation
- Step 1: Defining PyTorch Dataset and Dataloader
- Step 2: Load Data
- Step 3: Training and Evaluation
- Step 4: Main Training Loop
- Step 5: Testing the Final Model

[3]The `RoBERTa` paper: `https://arxiv.org/abs/1907.11692`

[4]See details of SST-2 at `https://huggingface.co/datasets/sst2`.

For each of Step 1-5, you will complete the following two tasks:

1. **Coding Exercises** in the **notebook**: You will complete the the code blocks denoted by `TODO:`.
2. **Questions to Answer** in the **report/write-up**: You will answer questions denoted by `Q:`.

For the full context of this exercise, please refer to the notebook. You should be able to answer the questions once you finish implementing the code blocks. As the **Coding Exercises** and **Questions to Answer** are interleaving and interdependent, **Questions to Answer** are best understood in conjunction with **Coding Exercises** in the notebook. However, to streamline your report/write-up, we list the questions below as a checklist for your report.

Step 1: Defining PyTorch Dataset and Dataloader

Q1.1: Explain the usages of the following arguments when you encode the input texts: `padding`, `max_length`, `truncation`, `return_tensors`.

- `padding` — When batching sequences together, each sequence needs to be the same length. Padding is applied to make sure all sequences in a batch are the same length for the model to process them in parallel. If `padding = True | "longest"`, the tokenizer will add a special token to the sequences shorter than the longest sequence in the batch to match its length.
- `truncation` — If `truncation = True`, sequences longer than `max_length` will be truncated down to that length.
- `max_length` — This is the maximum length of the sequence after tokenization. If a sequence is longer than `max_length`, it will be truncated to this length.
- `return_tensors` — Setting `return_tensors = "pt"` tells the tokenizer to return the sequences as PyTorch tensors. Other options include `"tf"` for TensorFlow tensors or `"np"` for NumPy arrays.

`https://huggingface.co/docs/transformers/v4.37.2/en/main_classes/tokenizer#transformers.PreTrainedTokenizerFast.encode` was very helpful for understanding these arguments.

Q1.2: For the above arguments, explain what are the potential advantages of setting them to the default values we provide.

- `padding=True` — We're ensuring that all sequences in a batch have the same length. This uniformity is required for batch processing on GPUs, which significantly speeds up training and evaluation since matrix operations on a GPU are highly parallelized. Without padding, batch processing would not be possible, and each sequence would need to be processed individually, which is way more inefficient.
- `truncation=True` — By enabling truncation, we're making sure that any sequence longer than 512 tokens is cut down, preventing errors from sequences that are too long for the model to handle.
- `max_length=512` — I looked it up, and it seems like many transformer models (like BERT) are trained on a maximum sequence length of 512 tokens. I guess this ensures we're using the model in the way it was designed and trained for, and also manages memory usage by keeping the sequences bounded.
- `return_tensors="pt"` — We can feed the sequences directly into the model without any further processing, since they're already in PyTorch tensor format. This is convenient and saves time and memory.

Step 2: Loading Data

Q2.1: What are the lengths of train, validation, test datasets?

* **Training set**: 6,920
* **Validation set**: 872
* **Test set**: 1,821

Q2.2: Explain the role of each of the following parameters: `batch_size`, `shuffle`, `collate_fn`, `num_workers` given to the `DataLoader` in the above code block.

* `batch_size` — The number of samples we want to pass into the training loop at each iteration.
* `shuffle` — Set to `True` if we want to shuffle the data at every epoch.
* `collate_fn` — The function used to collate the samples into a batch. In this case, we are passing `SST2Dataset.collate_fn` as our function, which does the tokenization and returns a dictionary of tensors for the input sequences and the labels.
* `num_workers` — The number of parallel subprocesses we want to use when we're loading all data. This just speeds up the data loading process.

Q2.3: Write the type and shape (if the type is tensor) of `input_ids`, `attention_mask`, and `label_encoding` in `batch` and explain what these elements represent.

All are tensors. Note that the batch size equals 64 and the sequence length equals 45.

* `input_ids` (shape: `[64, 45]`) — Each number in `input_ids` is a token ID that corresponds to a particular token in the tokenizer's vocabulary. The model uses these IDs to look up the embedding for each token and process the input sequence.
* `attention_mask` (shape: `[64, 45]`) — The attention mask designates which tokens are actual tokens versus padding tokens in `input_ids`. A value of 1 indicates a real token, and 0 indicates a padding token. This mask makes sure that the model does not attend to padding positions when processing the input.
* `label_encoding` (shape: `[64]`) — This is the label for each sequence in the batch. A value of 1 indicates a positive sentiment, and 0 indicates a negative sentiment. These are like the truth values we're trying to predict, which the model will compare against its predictions to calculate the loss during training.

Step 3: Training and Evaluation

Q3.1: For the three lines of code you implemented for computing gradients and updating parameters using optimizer, explain what each of the lines does, respectively.

1. `optimizer.zero_grad()` — This line clears the gradients from the previous batch. This is because in PyTorch, gradients are accumulated by default, so we need to clear them because otherwise our gradients would mix across batches.
2. `loss.backward()` — This line computes the gradient of the loss with respect to all the parameters in the model (technically, all the parameters that have `requires_grad=True`). This is just backpropagation.
3. `optimizer.step()` — This line actually performs the update based on the gradients we just computed in the last line. We update the parameters by taking a step in the direction of the negative gradient, scaled by the learning rate (direction of steepest descent).

Q3.2: Explain what setting the model to training and evaluation modes do, respectively.

It turns out there are some layers like `Dropout` and `BatchNorm` that are used during training but not during evaluation. For example, `Dropout` randomly zeroes some of the input tensor elements with a probability of p at each step during training which apparently is effective for regularization. Batch normalization uses batch statistics to normalize the input tensor, which is also different between training and evaluation. So, setting the model to training mode turns on these layers, and setting the model to evaluation mode turns them off. Evaluation mode means the model behaves more consistently, and we can use it to make predictions on new data.

Q3.3: Explain what `with torch.no_grad()` does in the `evaluation()` function.

We don't need to compute gradients during evaluation because we're not updating the model's parameters. Thus, if we want to save memory and compute time, we can disable gradient computation, which is what `torch.no_grad()` does. It's specifically a context manager, which allows the code inside the block to run without gradient computation.

Read more: `https://realpython.com/python-with-statement/`

Step 4: Main Training Loop

Q4.1: With the following default hyperparameters we provide, plot both training and validation loss curves across 10 epochs in a single plot (x-axis: num of the epoch; y-axis: acc). You can draw this plot with a Python script or other visualization tools like Google Sheets. (`batch_size = 64`, `learning_rate = 5e-5`, `num_epochs = 20`, `model_name = "roberta-base"`)

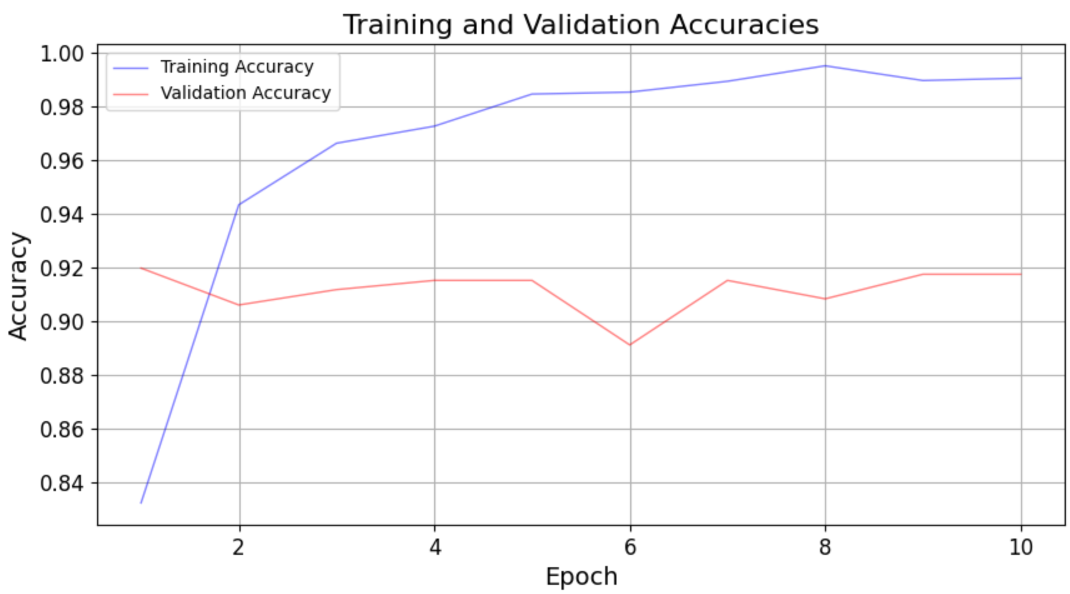

Figure 9: Training and validation accuracies across 10 epochs with optimizer `AdamW`, $\eta = 5 \cdot 10^{-5}$, and $\varepsilon = 1 \cdot 10^{-8}$.

Q4.2: Describe the behaviors of the training and validation loss curves you plotted above. At which epoch does the model achieve the best accuracy on the training dataset? What about the validation dataset? Do training and validation curves have the same trend? Why does the current trend happen?

The training accuracy starts off low and increases rapidly, while the validation accuracy starts off high and doesn't really change much. The model achieves the best accuracy on the training dataset in the 3rd-to-last epoch, however, the training accuracy in the last 3 epochs was almost at 1.0 anyways.

The validation accuracy surprisingly was highest in the first epoch, and it even decreased after that (although not by much).

The training and validation curves don't have the same trend. The training accuracy increases rapidly, while the validation accuracy doesn't really change much. This is likely because the model is overfitting to the training data, which is why the training accuracy keeps increasing while the validation accuracy doesn't. The model is learning the training data too well, and it's not generalizing well to the validation data.

Q4.3: Why do you shuffle the training data but not the validation data?

We shuffle the training data because we want to randomize the order of the training samples at the beginning of each epoch. This is important because we want the model to learn to generalize well to the entire training dataset, not just the order of the samples.

We don't shuffle the validation data because we don't need to. We're not updating the model's parameters during validation, so the order of the samples doesn't matter.

Q4.4: Explain the functionality of optimizers.

Optimizers are algorithms that are used to update the parameters of the model based on the gradients of the loss function. They are used to minimize the loss function by computing its gradient with respect to the model's parameters and updating the parameters in the direction of the negative gradient, scaled by the learning rate. This allows the parameters to flow towards the minimum of the loss function, which is where the model is most accurate.

Q4.5: Experiment with two other optimizers defined in `torch.optim` for the training the model with the default hyperparameters we give you. What is the difference between `AdamW` and these two new optimizers? Back up your claims with empirical evidence.

I experimented with `SGD` and `Adagrad`.

* `SGD` — The training accuracy and evaluation accuracy is much lower than with `AdamW`. It started out near 50% for both and didn't really change much! The model is not learning as well with `SGD` as it is with `AdamW`. I did not bother to plot the training and validation accuracies across 10 epochs with `SGD` because it was bad and didn't change much.
* `Adagrad` — With `Adagrad`, the training and validation accuracy were actually *higher* than with `AdamW`. The training accuracy started out above 87% and climbed to 98% by the end, while the validation accuracy started out at 91% or so and even got to 93% by the end!

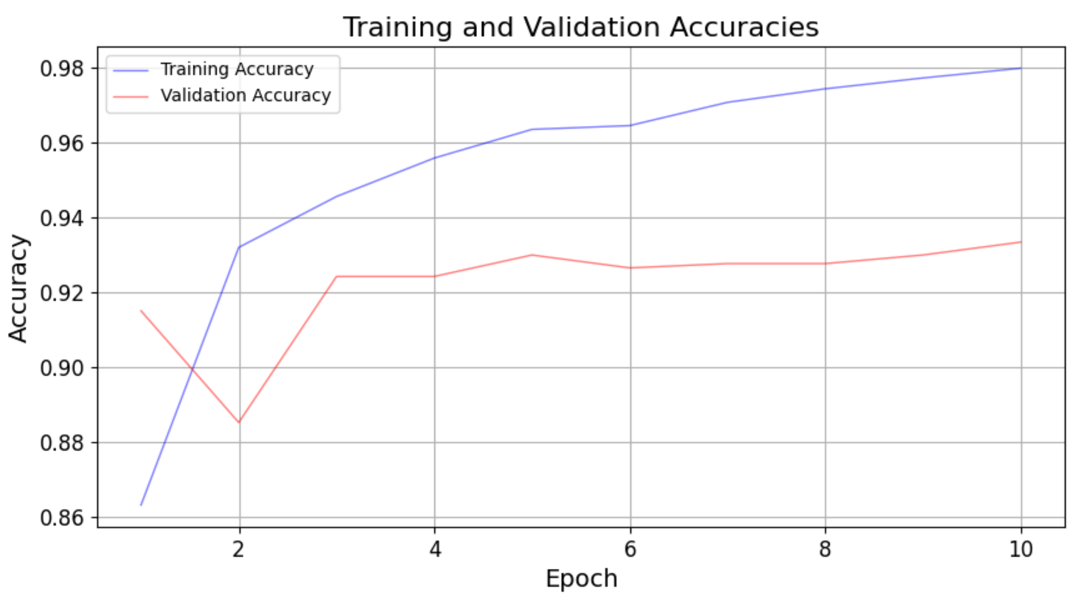

Figure 10: Training and validation accuracies across 10 epochs with optimizer `Adagrad`, $\eta = 5 \cdot 10^{-5}$, and $\varepsilon = 1 \cdot 10^{-8}$.

Q4.6: Experiment with different combinations of `batch_size`, `learning_rate`, and `num_epochs`. Your goal is to pick the final, best model checkpoint based on the validation dataset accuracy. Describe the strategy you used to try different combinations of hyperparameters. Why did you use this strategy?

I tried different combinations of `batch_size`, `learning_rate`, and `num_epochs` by using a grid search. I tried a few different values for each hyperparameter and trained the model with each combination. I then picked the combination that gave the highest validation accuracy.

I used this strategy because it's a simple and systematic way to try different combinations of hyperparameters. I wanted to make sure I didn't miss any good combination. A grid search is a good way to do this because it's easy to implement and it's easy to understand.

Q4.7: What are the `batch_size`, `learning_rate`, and `num_epochs` of the best model checkpoint that you picked? What are the training accuracy and validation accuracy?

The best configuration of hyperparameters I found was:

* `batch_size` $= 64$
* `learning_rate` $= 2 \cdot 10^{-6}$
* `num_epochs` $= 10$

Step 5: Testing the Final Model

Q5.1: What's the test set accuracy of the best model?

The test set accuracy I got on my best model was $\boxed{0.9330038440417353}$.