Assignment 5: Transformers

Instructor: Noah Smith   *CSE 517 and 447 – Winter 2025*

In this assignment, you will implement multi-head attention in transformers from scratch and use your implementation to train transformer-based language models.

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.

Required Deliverables

- **Code Notebook**: The assignment is associated with Jupyter notebook. (CSE447_Assignment5.ipynb) Unlike previous homeworks, this time we ask you to **submit the .ipynb notebooks and not .py scripts**. On Google Colab you can do so by File → Download → Download .ipynb. **Please comment out any additional code you had written to solve the write-up exercises before submitting on gradescope to avoid timeouts.**

- **Write-up**: For written answers and open-ended reports, produce a single PDF 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.

Recommended Reading

The homework is based on Lectures in week 7 (transformers), so the lecture slides should be your best resource. For more detailed reading we recommend checking Chapters 9 and 10 of Jurafsky and Martin. For decoding algorithms, check the lecture slides and section 13.4 in the textbook. We also recommend Patrick von Platen's blog post.

Required Compute

Except §1.1, all of the exercises will require you to use a GPU to run your code. We have tested the reference implementations on the free tier T4 GPU on colab and you should be able to use it to solve the exercises. If you run into any issues with the compute please contact the staff.

Acknowledgement

1 Building Your Own Mini Transformer (70 pts)

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.

Deliverables:

1. **Coding Exercises (§1.1):** You should complete the code blocks denoted by YOUR CODE HERE in the Python notebook. **We will only grade the codes you wrote for §1.1. §1.2 codes are not graded but will be useful for you to write the report.**

2. **Write-up (§1.2):** Your report for §1.2 should be **no more than five pages**. **We will only grade the write-up for §1.2.**

1.1 Implementing Attention from Scratch (30 pts)

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:

- 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 `d_model` features to `d_model` features. `d_model` is the embedding dimension of the token representations in the transformer. Attention does allow some (W_Q, W_K, W_V) 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: *(10 pts)* **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 $A = QK^\top$.

  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:

$$\frac{1}{\sqrt{d\_model/n\_head}}$$

- `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: *(10 pts)* **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: *(10 pts)* **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, d_model), where B is the batch size, n_tok is the sequence length, d_model 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, d_head) where n_heads is the number of heads and d_head is the embedding dimensionality of each head (d_model / n_heads). A single reshaping cannot convert from a tensor of shape (B, n_tok, d_model) to (B, n_heads, n_tok, d_head). Moreover, we want n_heads and d_head to be split from d_model and leave B and n_tok essentially untouched.

  To make the steps clear:

  First, reshape from (B, n_tok, d_model) to (B, n_tok, n_heads, d_head), where d_model = n_heads*d_head.

  Then, transpose the n_tok and n_heads dimensions from (B, n_tok, n_heads, d_head) to (B, n_heads, n_tok, d_head).

- `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, d_head) to (B, n_tok, n_heads, d_head).

  Then, reshape from (B, n_tok, n_heads, d_head) to (B, n_tok, d_model).

  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.

1.2 Experiment with Your Implementation of Attention (40 pts)

Check section 1 of CSE447_Assignment5_Writeup.pdf for detail.

Starter code: We have provided some starter code for you to train the tiny GPT model using the building blocks you implemented in §1.1. 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 §1.1 was completely guided, the code for §1.2 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 §1.2 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 §1.2, 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.