HW2: Attention and Transformers

CSCI 662: Fall 2025

out: Sep 29, 2025
due: Oct 17, 2025

Acknowledgement: This homework is based on Assignment 2 for Yejin Choi's CSE 447/517 class at UW, which was written by Yegor Kuznetsov, Liwei Jiang, and Jaehun Jung, with feedback from Alisa Liu, Melanie Sclar, Gary Liu, and Taylor Sorensen. This homework uses Andrej Karpathy's minGPT implementation of a GPT-style transformer.

Overview

This assignment has four parts. In the first part, which should be **very brief**, you'll do some hands-on practice with query, key, and value matrices. In the second part, you will implement the attention mechanism of your own mini-Transformer. In Part 3, you will pretrain your mini-Transformer on language modeling, then finetune it for the classification datasets from HW1. Finally, in part 4, you will experiment with how changes to the attention implementation impact model performance.

Part 1: Attention Practice (10%)

For this section of the assignment, you will interact with the attention operation and understand its behavior in a simple example. The code for this section is in `part1.py`, and you will fill in your answers in the functions `problem_1()`, `problem_2()`, and `problem_3()`. The rest of the file includes testing functions for you to check your answers. You can modify these functions as desired, and you are strongly encouraged to use them to check your answers before uploading to Gradescope.

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 will receive full credit.
- 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.

Selection and Averaging via Attention

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 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.
2. Define a *query* matrix Q ($\in \mathbb{R}^{4\times 3}$) which results in an identity mapping – select all the *value* vectors.
3. Define a *query* vector q ($\in \mathbb{R}^3$) which averages all the *value* vectors.

Part 2: Implementing Attention for your Mini-Transformer (10%)

In this part, you will implement multi-head scaled dot product self-attention. 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:

- "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 `multi_head_attention.py`. 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. After you implement each step, you should fill in the corresponding part of `self_attention()` and run the corresponding tests. You can call the functions `test_step_1()`, `test_step_2()`, `test_step_3()`. You can also run `python multi_head_attention.py` to run all tests. It will raise `NotImplementedError` for parts you haven't implemented yet - this is expected and OK.

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.**

Code to implement (in `multi_head_attention.py`): Here, we provide high-level explanations of what each function does in `multi_head_attention.py` **In `multi_head_attention.py`, you will complete code blocks denoted by `TODO:`.**

Step 0: Set up the projections for attention.

- `init_qkv_proj()`: **This is already implemented for you.**
  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 test functions 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'. You do not need to implement your own softmax from scratch - you can use pytorch functions.
- `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()`, test this portion of your implementation with `test_step_1()`.

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 shape `(B, n_tok, n_embd)` to `(B, n_heads, n_tok, n_embd_per_head)`. Moreover, we want `n_heads` and `n_embd_per_head` to be split from `n_embd` and leave `B` and `n_tok` essentially untouched.

  To make the steps clear:

  First, reshape from `(B, n_tok, n_embd)` to `(B, n_tok, n_heads, n_embd_per_head)`, where `n_embd = n_heads*n_embd_per_head`.

  Then, transpose the `n_tok` and `n_heads` dimensions from `(B, n_tok, n_heads, n_embd_per_head)` to `(B, n_heads, n_tok, n_embd_per_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, 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 pass with very low error now.

Part 3: Training your Mini-Transformer (20%)

Now that you have a working implementation of *Causal Multi-Head Scaled Dot Product Self-Attention*,[2], you will pretrain your mini-Transformers for language modeling, then finetune them for several classification tasks. We have provided modified versions of `model.py`, `bpe.py`, `trainer.py`, and `utils.py` from Andrej Karpathy's minGPT implementation of a GPT-style transformer. We have also provided `train.py` and `evaluate.py`, which are very similar to the `train.py` and `classify.py` scripts from HW1. We also provide the same classification datasets from HW1, plus training and dev data for language model pretraining: `1b_benchmark.train.tokens` and `1b_benchmark.dev.tokens`. You are allowed to use additional pretraining or finetuning data, as long as you don't attempt to locate our specific test sets online.

Once you have implemented attention as described in part 2, you should be able to do basic training and evaluation with little to no changes to the starter code. You are expected to experiment thoroughly with

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

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

the attention mechanism and/or with other parts of your mini-Transformer, as described below. 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 it has not been tuned in any way beyond getting it to run, and is nowhere near optimal. Part of your task is to improve it for your experiment if necessary.

You will need to submit a minimum of 5 models for autograding: one pretrained on LM objective, and four with classification heads for specific classification tasks. The starter code is set up to allow LM pretraining followed by task finetuning, but you are not required to train your models this way. Remember that the questions dataset is non-English, so English pretraining may be unhelpful or even harmful. You can submit as many models as you want, named `id.task.model`, where `task` $\in$ `[lm, questions, products, 4dim, news]`. Your best model for each task should be named `best.task.model` - only `best` models will be autograded for code correctness and posted to the leaderboard. Others will be evaluated but not scored, like the additonal models you may have submitted with HW1.

Compute Requirements: A single pretraining epoch of a `gpt-nano`-size model on `1b_benchmark.train.tokens` took about 15 minutes on the TA's CPU. Finetuning time (per epoch) for the same model size and same CPU was under 3 minutes for all datasets. All evaluations took about 1 minute per dataset. You probably want more epochs, and you may want to use GPU for pretraining. You are allowed to use models larger than `gpt-nano`, but it's not required to get full points for this assignment. If you choose to use larger models, you are responsible for ensuring that the autograder can evaluate your models within 40 minutes on 4 CPUs with 6GB RAM. For reference, a full autograding run of 7 `gpt-nano`-sized models took 12 minutes in our testing.

Part 4: Experiment with Your Mini-Transformer

You are required to implement two meaningful changes to the internals of the basic transformer provided to you. At least one of your modifications must be a change to the attention mechanism you implemented in `multi_head_attention.py`. The other can be a different change to attention, or it can be a change to any other part of the transformer implementation. You are encouraged to tune hyperparameters; however, changing hyperparameters is not enough to constitute a "meaningful change."

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. You are encouraged to report on how your experiments affect training efficiency, dev set perplexity, and classification performance. 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?

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?

'Extra Mile' ideas

This is not meant to be comprehensive and you do not have to do any of the things here (nor should you do all of them). But an 'extra mile' component is 10% of your grade (**not extra credit!**).

- Replicate the main change(s) to attention in Attention Free Transformer
- Replicate the main change(s) to attention in Fast Attention
- **Thorough** ablation or grid search for hyperparameter tuning.
- Dig into the ACL/NeurIPS/ICLR/etc. archives and find ideas for other architectures, augmentations, or approaches; try them out, analyze performance.

Your Report

Write a report about your experimentation with your mini-Transformer. It's perfectly okay (and even expected) if your experiments did not improve performance.

Your report should at a *minimum*:

- Explain the setup of the experiment, including the motivations for your two changes and relevant background.
- Describe technical details: any pre/post-processing steps, compute, hyperparameter tuning, changes to provided code, etc.
- 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.
- Interpret your results. What does this mean for using attention in language modeling? Do your results support some aspect of the design of attention?

Use the ACL style files : `https://github.com/acl-org/acl-style-files`

Your report should be at least two pages long, including references, and not more than five pages long, not including references (i.e. you can have up to five pages of text if you need to). Just like a conference paper or journal article it should contain an abstract, introduction, experimental results, and conclusion sections (as well as other sections as deemed necessary). Unlike a conference paper/journal article, a complete related works section is not obligatory (but you may include it if it is relevant to what you do).

Files to Submit

You must submit the following files:

1. `part1.py`
2. `multi_head_attention.py`
3. `train.py`
4. `evaluate.py`
5. modified versions of `bpe.py`, `utils.py`, `trainer.py`, `model.py` if you modified these files
6. at least 5 model files: `best.[lm, questions,products,4dim,news].model`.
7. `requirements.txt`, if you need packages beyond the preinstalled ones listed below
8. `README.md`
9. `hw2_report.pdf`

Coding Recommendations

- See CARC stater guide for advice on GPU access, if you need it.
- For development purposes, start small and see that the training loop works as expected (i.e. loss drops, you can overfit to the training set, etc.). Training the model on the full dataset will take some time, so only run training on the full dataset once you are confident that your set up is correct. Use the `-d` flag provided in `train.py` and `evaluate.py` to debug on dummy data.
- You are more than welcome to use any packages out there to help you train your model. If you are having trouble getting the basic training going please reach out for help; this shouldn't be where you spend most of your time.
- Autograder improvements for HW2:
  - The following packages (and their dependencies) will be pre-installed for you: `torch, scikit-learn, tqdm, numpy, requests, regex`. No need to include them in your `requirements.txt`, and no need to waste your autograder time on installing them! We will install the latest versions of these packages that are compatible with Python 3.10. If you want different versions, put them in your `requirements.txt`.
  - The autograder will now use uv pip instead of vanilla pip for faster package installs. Just submit your `requirements.txt` as normal, and we will install dependencies from it using We will install your dependencies using `uv pip install -r requirements.txt`.
  - `requirements.txt` is now optional. You only need to submit it if you want packages besides the pre-installed ones listed above.

Grading

Grading will be roughly broken down as follows:

- 10% – (Part 1) Did you do the calculations right? You can check with the provided tests, so you should be able to get this perfect.
- 10% – (Part 2) Is your attention implementation in `multi_head_attention.py` correct? You can check this with the provided tests, so you should be able to get full credit here.
- 15% – (Part 3) Did you train and submit the 5 required models? Do they perform above baseline thresholds?
- 5% – (Code Quality, all Parts) Is your code well-written, documented, and robust? Will it run from a different directory than the one you ran it in? Does it rely on hard-codes? Is it commented and structured such that we can read it and understand what you are doing?
- 50% – (Part 4 and Report) Did you implement 2 meaningful changes to the attention mechanism or other parts of the model? Does your report read like a research paper? Did you clearly communicate your description of what you implemented, how you implemented it, what your experiments were, and what conclusions you drew from them? This includes appropriate use of graphics and tables where warranted that clearly explain your point. This also includes well written explanations that tell a compelling story. Grammar and syntax are a small part of this (maybe 5% of the grade, so 10% of this section) but much more important is the narrative you tell. Also a part of this is that you clearly acknowledged your sources and influences with appropriate bibliography and, where relevant, cited influencing prior work.
- 10% – Did you go the extra mile in Part 4? Did you push beyond what was asked for in the assignment, trying (well-justified) new models, features, or approaches? Did you use motivation (and document appropriately) from another researcher trying the same problem or from an unrelated but transferable other paper?

Rules

- This is an individual assignment. You may not work in teams or collaborate with other students. You must be the sole author of 100% of the code you turn in.
- Depending on need and class interest, we may collaborate *in class* or *publicly on Slack* if you get stuck; this kind of collaboration is okay.
- You may not look for coded solutions on the web, or use code you find online or anywhere else. You can and are encouraged to read material beyond what you have been given in class (see above) but should not copy code.
- Generative language, code, and vision models (e.g. ChatGPT, Llama 2, Midjourney, Github Copilot, etc.; if you are unsure, ask and don't assume!!) can be used (to aid in report writing/coding, not to actually do the classification tasks) with the following caveats:
  - You must declare your use of the tools in your submitted artifact. If you don't declare the tool usage but you did use these tools, we will consider that plagiarism
  - For code and image generation, you must indicate the prompt used and output generated
  - For text generation you must provide either a link to the chat session you used to help write the content or an equivalent readout of the inputs you provided and outputs received from the system. You will lose credit if "the AI" is doing the work rather than you.
- Failure to follow the above rules is considered a violation of academic integrity, and is grounds for failure of the assignment, or in serious cases failure of the course.
- We use plagiarism detection software to identify similarities between student assignments, and between student assignments and known solutions on the web. Any attempt to fool plagiarism detection, for example the modification of code to reduce its similarity to the source, will result in an automatic failing grade for the course.
- If you have questions about what is and isn't allowed, post them to Slack!