Decoder Only!

CS336 Assignment 1 (basics): Building a Transformer LM

Version 1.0.4

CS336 Staff

Spring 2025

1 Assignment Overview

In this assignment, you will build all the components needed to train a standard Transformer language model (LM) from scratch and train some models.

What you will implement

1. Byte-pair encoding (BPE) tokenizer (§2)
2. Transformer language model (LM) (§3)
3. The cross-entropy loss function and the AdamW optimizer (§4)
4. The training loop, with support for serializing and loading model and optimizer state (§5)

What you will run

1. Train a BPE tokenizer on the TinyStories dataset.
2. Run your trained tokenizer on the dataset to convert it into a sequence of integer IDs.
3. Train a Transformer LM on the TinyStories dataset.
4. Generate samples and evaluate perplexity using the trained Transformer LM.
5. Train models on OpenWebText and submit your attained perplexities to a leaderboard.

What you can use We expect you to build these components from scratch. In particular, you may *not* use any definitions from `torch.nn`, `torch.nn.functional`, or `torch.optim` except for the following:

- `torch.nn.Parameter`
- Container classes in `torch.nn` (e.g., `Module`, `ModuleList`, `Sequential`, etc.)[1]
- The `torch.optim.Optimizer` base class

You may use any other PyTorch definitions. If you would like to use a function or class and are not sure whether it is permitted, feel free to ask on Slack. When in doubt, consider if using it compromises the "from-scratch" ethos of the assignment.

[1] See PyTorch.org/docs/stable/nn.html#containers for a full list.

Statement on AI tools Prompting LLMs such as ChatGPT is permitted for low-level programming questions or high-level conceptual questions about language models, but using it directly to solve the problem is prohibited.

We strongly encourage you to disable AI autocomplete (e.g., Cursor Tab, GitHub CoPilot) in your IDE when completing assignments (though non-AI autocomplete, e.g., autocompleting function names is totally fine). We have found that AI autocomplete makes it much harder to engage deeply with the content.

What the code looks like All the assignment code as well as this writeup are available on GitHub at:

`github.com/stanford-cs336/assignment1-basics`

Please `git clone` the repository. If there are any updates, we will notify you so you can `git pull` to get the latest.

1. `cs336_basics/*`: This is where you write your code. Note that there's no code in here—you can do whatever you want from scratch!

2. `adapters.py`: There is a set of functionality that your code must have. For each piece of functionality (e.g., scaled dot product attention), fill out its implementation (e.g., `run_scaled_dot_product_attention`) by simply invoking your code. Note: your changes to `adapters.py` should not contain any substantive logic; this is glue code.

3. `test_*.py`: This contains all the tests that you must pass (e.g., `test_scaled_dot_product_attention`), which will invoke the hooks defined in `adapters.py`. Don't edit the test files.

How to submit You will submit the following files to Gradescope:

- `writeup.pdf`: Answer all the written questions. Please typeset your responses.
- `code.zip`: Contains all the code you've written.

To submit to the leaderboard, submit a PR to:

`github.com/stanford-cs336/assignment1-basics-leaderboard`

See the `README.md` in the leaderboard repository for detailed submission instructions.

Where to get datasets This assignment will use two pre-processed datasets: TinyStories [Eldan and Li, 2023] and OpenWebText [Gokaslan et al., 2019]. Both datasets are single, large plaintext files. If you are doing the assignment with the class, you can find these files at `/data` of any non-head node machine.

If you are following along at home, you can download these files with the commands inside the `README.md`.

Low-Resource/Downscaling Tip: Init

Throughout the course's assignment handouts, we will give advice for working through parts of the assignment with fewer or no GPU resources. For example, we will sometimes suggest **downscaling** your dataset or model size, or explain how to run training code on a MacOS integrated GPU or CPU. You'll find these "low-resource tips" in a blue box (like this one). Even if you are an enrolled Stanford student with access to the course machines, these tips may help you iterate faster and save time, so we recommend you to read them!

Low-Resource/Downscaling Tip: Assignment 1 on Apple Silicon or CPU

With the staff solution code, we can train an LM to generate reasonably fluent text on an Apple M3 Max chip with 36 GB RAM, in under 5 minutes on Metal GPU (MPS) and about 30 minutes using the CPU. If these words don't mean much to you, don't worry! Just know that if you have a reasonably up-to-date laptop and your implementation is correct and efficient, you will be able to train a small LM that generates simple children's stories with decent fluency.

Later in the assignment, we will explain what changes to make if you are on CPU or MPS.

2 Byte-Pair Encoding (BPE) Tokenizer

In the first part of the assignment, we will train and implement a byte-level byte-pair encoding (BPE) tokenizer [Sennrich et al., 2016, Wang et al., 2019]. In particular, we will represent arbitrary (Unicode) strings as a sequence of bytes and train our BPE tokenizer on this byte sequence. Later, we will use this tokenizer to encode text (a string) into tokens (a sequence of integers) for language modeling.

2.1 The Unicode Standard

Unicode is a text encoding standard that maps characters to integer *code points*. As of Unicode 16.0 (released in September 2024), the standard defines 154,998 characters across 168 scripts. For example, the character "s" has the code point 115 (typically notated as `U+0073`, where `U+` is a conventional prefix and `0073` is 115 in hexadecimal), and the character "牛" has the code point 29275. In Python, you can use the `ord()` function to convert a single Unicode character into its integer representation. The `chr()` function converts an integer Unicode code point into a string with the corresponding character.

```
>>> ord('牛')
29275
>>> chr(29275)
'牛'
```

Problem (`unicode1`): Understanding Unicode (1 point)

(a) What Unicode character does `chr(0)` return?

Deliverable: A one-sentence response.

(b) How does this character's string representation (`__repr__()`) differ from its printed representation?

Deliverable: A one-sentence response.

(c) What happens when this character occurs in text? It may be helpful to play around with the following in your Python interpreter and see if it matches your expectations:

```
>>> chr(0)
>>> print(chr(0))
>>> "this is a test" + chr(0) + "string"
>>> print("this is a test" + chr(0) + "string")
```

Deliverable: A one-sentence response.

2.2 Unicode Encodings

While the Unicode standard defines a mapping from characters to code points (integers), it's impractical to train tokenizers directly on Unicode codepoints, since the vocabulary would be prohibitively large (around 150K items) and sparse (since many characters are quite rare). Instead, we'll use a Unicode encoding, which converts a Unicode character into a sequence of bytes. The Unicode standard itself defines three encodings: UTF-8, UTF-16, and UTF-32, with UTF-8 being the dominant encoding for the Internet (more than 98% of all webpages).

To encode a Unicode string into UTF-8, we can use the `encode()` function in Python. To access the underlying byte values for a Python `bytes` object, we can iterate over it (e.g., call `list()`). Finally, we can use the `decode()` function to decode a UTF-8 byte string into a Unicode string.

```
>>> test_string = "hello! こんにちは!"
>>> utf8_encoded = test_string.encode("utf-8")
>>> print(utf8_encoded)
b'hello! \xe3\x81\x93\xe3\x82\x93\xe3\x81\xab\xe3\x81\xa1\xe3\x81\xaf!'
>>> print(type(utf8_encoded))
<class 'bytes'>
>>> # Get the byte values for the encoded string (integers from 0 to 255).
>>> list(utf8_encoded)
[104, 101, 108, 108, 111, 33, 32, 227, 129, 147, 227, 130, 147, 227, 129, 171, 227, 129,
161, 227, 129, 175, 33]
>>> # One byte does not necessarily correspond to one Unicode character!
>>> print(len(test_string))
13
>>> print(len(utf8_encoded))
23
>>> print(utf8_encoded.decode("utf-8"))
hello! こんにちは!
```

By converting our Unicode codepoints into a sequence of bytes (e.g., via the UTF-8 encoding), we are essentially taking a sequence of codepoints (integers in the range 0 to 154,997) and transforming it into a sequence of byte values (integers in the range 0 to 255). The 256-length byte vocabulary is *much* more manageable to deal with. When using byte-level tokenization, we do not need to worry about out-of-vocabulary tokens, since we know that *any* input text can be expressed as a sequence of integers from 0 to 255.

Problem (`unicode2`): Unicode Encodings (3 points)

(a) What are some reasons to prefer training our tokenizer on UTF-8 encoded bytes, rather than UTF-16 or UTF-32? It may be helpful to compare the output of these encodings for various input strings.

Deliverable: A one-to-two sentence response.

(b) Consider the following (incorrect) function, which is intended to decode a UTF-8 byte string into a Unicode string. Why is this function incorrect? Provide an example of an input byte string that yields incorrect results.

bytes with an integer return a bytestring of that integer length,
while bytes with an iterable return the bytestring of that object

```
def decode_utf8_bytes_to_str_wrong(bytestring: bytes):
    return "".join([bytes([b]).decode("utf-8") for b in bytestring])

>>> decode_utf8_bytes_to_str_wrong("hello".encode("utf-8"))
'hello'
```

Deliverable: An example input byte string for which `decode_utf8_bytes_to_str_wrong` produces incorrect output, with a one-sentence explanation of why the function is incorrect.

(c) Give a two byte sequence that does not decode to any Unicode character(s).

Deliverable: An example, with a one-sentence explanation.

2.3 Subword Tokenization

While byte-level tokenization can alleviate the out-of-vocabulary issues faced by word-level tokenizers, tokenizing text into bytes results in extremely long input sequences. This slows down model training, since a sentence with 10 words might only be 10 tokens long in a word-level language model, but could be 50 or more tokens long in a character-level model (depending on the length of the words). Processing these longer sequences requires more computation at each step of the model. Furthermore, language modeling on byte sequences is difficult because the longer input sequences create long-term dependencies in the data.

Subword tokenization is a midpoint between word-level tokenizers and byte-level tokenizers. Note that a byte-level tokenizer's vocabulary has 256 entries (byte values are 0 to 225). A subword tokenizer trades-off a larger vocabulary size for better compression of the input byte sequence. For example, if the byte sequence `b'the'` often occurs in our raw text training data, assigning it an entry in the vocabulary would reduce this 3-token sequence to a single token.

How do we select these subword units to add to our vocabulary? Sennrich et al. [2016] propose to use byte-pair encoding (BPE; Gage, 1994), a compression algorithm that iteratively replaces ("merges") the most frequent pair of bytes with a single, new unused index. Note that this algorithm adds subword tokens to our vocabulary to maximize the compression of our input sequences—if a word occurs in our input text enough times, it'll be represented as a single subword unit.

Subword tokenizers with vocabularies constructed via BPE are often called BPE tokenizers. In this assignment, we'll implement a byte-level BPE tokenizer, where the vocabulary items are bytes or merged sequences of bytes, which give us the best of both worlds in terms of out-of-vocabulary handling and manageable input sequence lengths. The process of constructing the BPE tokenizer vocabulary is known as "training" the BPE tokenizer.

2.4 BPE Tokenizer Training

The BPE tokenizer training procedure consists of three main steps.

Vocabulary initialization The tokenizer vocabulary is a one-to-one mapping from bytestring token to integer ID. Since we're training a byte-level BPE tokenizer, our initial vocabulary is simply the set of all bytes. Since there are 256 possible byte values, our initial vocabulary is of size 256.

Pre-tokenization Once you have a vocabulary, you could, in principle, count how often bytes occur next to each other in your text and begin merging them starting with the most frequent pair of bytes. However, this is quite computationally expensive, since we'd have to go take a full pass over the corpus each time we merge. In addition, directly merging bytes across the corpus may result in tokens that differ only in punctuation (e.g., `dog!` vs. `dog.`). These tokens would get completely different token IDs, even though they are likely to have high semantic similarity (since they differ only in punctuation).

To avoid this, we *pre-tokenize* the corpus. You can think of this as a coarse-grained tokenization over the corpus that helps us count how often pairs of characters appear. For example, the word `'text'` might be a pre-token that appears 10 times. In this case, when we count how often the characters 't' and 'e' appear next to each other, we will see that the word 'text' has 't' and 'e' adjacent and we can increment their count by 10 instead of looking through the corpus. Since we're training a byte-level BPE model, each pre-token is represented as a sequence of UTF-8 bytes.

The original BPE implementation of Sennrich et al. [2016] pre-tokenizes by simply splitting on whitespace (i.e., `s.split(" ")`). In contrast, we'll use a regex-based pre-tokenizer (used by GPT-2; Radford et al., 2019) from github.com/openai/tiktoken/pull/234/files:

```
>>> PAT = r"""'(?:[sdmt]|ll|ve|re)| ?\p{L}+| ?\p{N}+| ?[^\s\p{L}\p{N}]+|\s+(?!\S)|\s+"""
```

It may be useful to interactively split some text with this pre-tokenizer to get a better sense of its behavior:

‘(?:[sdmt] | ll | ve | re) is apostrophe followed by s, d, m, t, ll, ve, or re

?\p{L}+ is one or more unicode letters from any language with potential leading space

```
>>> # requires `regex` package
>>> import regex as re
>>> re.findall(PAT, "some text that i'll pre-tokenize")
['some', ' text', ' that', ' i', "'ll", ' pre', '-', 'tokenize']
```


?\p{N}+ is one or more unicode numbers with potentially leading space

?[^\s\p{L}\p{N}]+ is anything that is not a space or unicode number or word, e.g. punctuation

\s+(?!\S) matches one or more whitespace characters and ?!\S says don't match unless followed by a whitespace character (i.e. should be trailing whitespace)

When using it in your code, however, you should use `re.finditer` to avoid storing the pre-tokenized words as you construct your mapping from pre-tokens to their counts.

Compute BPE merges Now that we've converted our input text into pre-tokens and represented each pre-token as a sequence of UTF-8 bytes, we can compute the BPE merges (i.e., train the BPE tokenizer). At a high level, the BPE algorithm iteratively counts every pair of bytes and identifies the pair with the highest frequency ("A", "B"). Every occurrence of this most frequent pair ("A", "B") is then *merged*, i.e., replaced with a new token "AB". This new merged token is added to our vocabulary; as a result, the final vocabulary after BPE training is the size of the initial vocabulary (256 in our case), plus the number of BPE merge operations performed during training. For efficiency during BPE training, we do not consider pairs that cross pre-token boundaries.[2] When computing merges, deterministically break ties in pair frequency by *preferring the lexicographically greater pair*. For example, if the pairs ("A", "B"), ("A", "C"), ("B", "ZZ"), and ("BA", "A") all have the highest frequency, we'd merge ("BA", "A"):

```
>>> max([("A", "B"), ("A", "C"), ("B", "ZZ"), ("BA", "A")])
('BA', 'A')
```

Special tokens Often, some strings (e.g., `<|endoftext|>`) are used to encode metadata (e.g., boundaries between documents). When encoding text, it's often desirable to treat some strings as "special tokens" that should never be split into multiple tokens (i.e., will always be preserved as a single token). For example, the end-of-sequence string `<|endoftext|>` should always be preserved as a single token (i.e., a single integer ID), so we know when to stop generating from the language model. These special tokens must be added to the vocabulary, so they have a corresponding fixed token ID.

Algorithm 1 of Sennrich et al. [2016] contains an inefficient implementation of BPE tokenizer training (essentially following the steps that we outlined above). As a first exercise, it may be useful to implement and test this function to test your understanding.

Example (`bpe_example`): BPE training example

Here is a stylized example from Sennrich et al. [2016]. Consider a corpus consisting of the following text

```
low low low low low
lower lower widest widest widest
newest newest newest newest newest newest
```

and the vocabulary has a special token `<|endoftext|>`.

Vocabulary We initialize our vocabulary with our special token `<|endoftext|>` and the 256 byte values.

Pre-tokenization For simplicity and to focus on the merge procedure, we assume in this example that pretokenization simply splits on whitespace. When we pretokenize and count, we end up with the frequency table.

```
{low: 5, lower: 2, widest: 3, newest: 6}
```

[2]Note that the original BPE formulation [Sennrich et al., 2016] specifies the inclusion of an end-of-word token. We do not add an end-of-word-token when training byte-level BPE models because all bytes (including whitespace and punctuation) are included in the model's vocabulary. Since we're explicitly representing spaces and punctuation, the learned BPE merges will naturally reflect these word boundaries.

It is convenient to represent this as a `dict[tuple[bytes], int]`, e.g. {`(l,o,w)`: 5 …}. Note that even a single byte is a `bytes` object in Python. There is no `byte` type in Python to represent a single byte, just as there is no `char` type in Python to represent a single character.

Merges We first look at every successive pair of bytes and sum the frequency of the words where they appear {`lo`: 7, `ow`: 7, `we`: 8, `er`: 2, `wi`: 3, `id`: 3, `de`: 3, `es`: 9, `st`: 9, `ne`: 6, `ew`: 6}. The pair `('es')` and `('st')` are tied, so we take the lexicographically greater pair, `('st')`. We would then merge the pre-tokens so that we end up with {`(l,o,w)`: 5, `(l,o,w,e,r)`: 2, `(w,i,d,e,st)`: 3, `(n,e,w,e,st)`: 6}.

In the second round, we see that `(e, st)` is the most common pair (with a count of 9) and we would merge into {`(l,o,w)`: 5, `(l,o,w,e,r)`: 2, `(w,i,d,est)`: 3, `(n,e,w,est)`: 6}. Continuing this, the sequence of merges we get in the end will be `['s t', 'e st', 'o w', 'l ow', 'w est', 'n e', 'ne west', 'w i', 'wi d', 'wid est', 'low e', 'lowe r']`.

If we take 6 merges, we have `['s t', 'e st', 'o w', 'l ow', 'w est', 'n e']` and our vocabulary elements would be `[<|endoftext|>, [...256 BYTE CHARS], st, est, ow, low, west, ne]`.

With this vocabulary and set of merges, the word `newest` would tokenize as `[ne, west]`.

2.5 Experimenting with BPE Tokenizer Training

Let's train a byte-level BPE tokenizer on the TinyStories dataset. Instructions to find / download the dataset can be found in Section 1. Before you start, we recommend taking a look at the TinyStories dataset to get a sense of what's in the data.

Parallelizing pre-tokenization You will find that a major bottleneck is the pre-tokenization step. You can speed up pre-tokenization by parallelizing your code with the built-in library `multiprocessing`. Concretely, we recommend that in parallel implementations of pre-tokenization, you chunk the corpus while ensuring your chunk boundaries occur at the beginning of a special token. You are free to use the starter code at the following link verbatim to obtain chunk boundaries, which you can then use to distribute work across your processes:

https://github.com/stanford-cs336/assignment1-basics/blob/main/cs336_basics/pretokenization_example.py

This chunking will always be valid, since we never want to merge across document boundaries. For the purposes of the assignment, you can always split in this way. Don't worry about the edge case of receiving a very large corpus that does not contain `<|endoftext|>`.

Removing special tokens before pre-tokenization Before running pre-tokenization with the regex pattern (using `re.finditer`), you should strip out all special tokens from your corpus (or your chunk, if using a parallel implementation). Make sure that you **split** on your special tokens, so that no merging can occur across the text they delimit. For example, if you have a corpus (or chunk) like `[Doc 1]<|endoftext|>[Doc 2]`, you should split on the special token `<|endoftext|>`, and pre-tokenize `[Doc 1]` and `[Doc 2]` separately, so that no merging can occur across the document boundary. This can be done using `re.split` with `"|" .join(special_tokens)` as the delimiter (with careful use of `re.escape` since `|` may occur in the special tokens). The test `test_train_bpe_special_tokens` will test for this.

Optimizing the merging step The naïve implementation of BPE training in the stylized example above is slow because for every merge, it iterates over all byte pairs to identify the most frequent pair. However, the only pair counts that change after each merge are those that overlap with the merged pair. Thus, BPE training speed can be improved by indexing the counts of all pairs and incrementally updating these counts, rather than explicitly iterating over each pair of bytes to count pair frequencies. You can get significant speedups with this caching procedure, though we note that the merging part of BPE training is *not* parallelizable in Python.

Low-Resource/Downscaling Tip: Profiling

You should use profiling tools like `cProfile` or `scalene` to identify the bottlenecks in your implementation, and focus on optimizing those.

Low-Resource/Downscaling Tip: "Downscaling"

Instead of jumping to training your tokenizer on the full TinyStories dataset, we recommend you first train on a small subset of the data: a "debug dataset". For example, you could train your tokenizer on the TinyStories validation set instead, which is 22K documents instead of 2.12M. This illustrates a general strategy of downscaling whenever possible to speed up development: for example, using smaller datasets, smaller model sizes, etc. Choosing the size of the debug dataset or hyperparameter config requires careful consideration: you want your debug set to be large enough to have the same bottlenecks as the full configuration (so that the optimizations you make will generalize), but not so big that it takes forever to run.

Problem (`train_bpe`): BPE Tokenizer Training (15 points)

Deliverable: Write a function that, given a path to an input text file, trains a (byte-level) BPE tokenizer. Your BPE training function should handle (at least) the following input parameters:

`input_path`: `str` Path to a text file with BPE tokenizer training data.

`vocab_size`: `int` A positive integer that defines the maximum final vocabulary size (including the initial byte vocabulary, vocabulary items produced from merging, and any special tokens).

`special_tokens`: `list[str]` A list of strings to add to the vocabulary. These special tokens do not otherwise affect BPE training.

Your BPE training function should return the resulting vocabulary and merges:

`vocab`: `dict[int, bytes]` The tokenizer vocabulary, a mapping from `int` (token ID in the vocabulary) to `bytes` (token bytes).

`merges`: `list[tuple[bytes, bytes]]` A list of BPE merges produced from training. Each list item is a `tuple` of `bytes` (`<token1>`, `<token2>`), representing that `<token1>` was merged with `<token2>`. The merges should be ordered by order of creation.

To test your BPE training function against our provided tests, you will first need to implement the test adapter at [adapters.run_train_bpe]. Then, run `uv run pytest tests/test_train_bpe.py`. Your implementation should be able to pass all tests. Optionally (this could be a large time-investment), you can implement the key parts of your training method using some systems language, for instance C++ (consider `cppyy` for this) or Rust (using PyO3). If you do this, be aware of which operations require copying vs reading directly from Python memory, and make sure to leave build instructions, or make sure it builds using only `pyproject.toml`. Also note that the GPT-2 regex is not well-supported in most regex engines and will be too slow in most that do. We have verified that Oniguruma is reasonably fast and supports negative lookahead, but the `regex` package in Python is, if anything, even faster.

Problem (train_bpe_tinystories): BPE Training on TinyStories (2 points)

(a) Train a byte-level BPE tokenizer on the TinyStories dataset, using a maximum vocabulary size of 10,000. Make sure to add the TinyStories `<|endoftext|>` special token to the vocabulary. Serialize the resulting vocabulary and merges to disk for further inspection. How many hours and memory did training take? What is the longest token in the vocabulary? Does it make sense?

Resource requirements: $\leq$ 30 minutes (no GPUs), $\leq$ 30GB RAM

Hint You should be able to get under 2 minutes for BPE training using `multiprocessing` during pretokenization and the following two facts:

(a) The `<|endoftext|>` token delimits documents in the data files.

(b) The `<|endoftext|>` token is handled as a special case before the BPE merges are applied.

Deliverable: A one-to-two sentence response.

(b) Profile your code. What part of the tokenizer training process takes the most time?

Deliverable: A one-to-two sentence response.

Next, we'll try training a byte-level BPE tokenizer on the OpenWebText dataset. As before, we recommend taking a look at the dataset to better understand its contents.

Problem (train_bpe_expts_owt): BPE Training on OpenWebText (2 points)

(a) Train a byte-level BPE tokenizer on the OpenWebText dataset, using a maximum vocabulary size of 32,000. Serialize the resulting vocabulary and merges to disk for further inspection. What is the longest token in the vocabulary? Does it make sense?

Resource requirements: $\leq$ 12 hours (no GPUs), $\leq$ 100GB RAM

Deliverable: A one-to-two sentence response.

(b) Compare and contrast the tokenizer that you get training on TinyStories versus OpenWebText.

Deliverable: A one-to-two sentence response.

2.6 BPE Tokenizer: Encoding and Decoding

In the previous part of the assignment, we implemented a function to train a BPE tokenizer on input text to obtain a tokenizer vocabulary and a list of BPE merges. Now, we will implement a BPE tokenizer that loads a provided vocabulary and list of merges and uses them to encode and decode text to/from token IDs.

2.6.1 Encoding text

The process of encoding text by BPE mirrors how we train the BPE vocabulary. There are a few major steps.

Step 1: Pre-tokenize. We first pre-tokenize the sequence and represent each pre-token as a sequence of UTF-8 bytes, just as we did in BPE training. We will be merging these bytes within each pre-token into vocabulary elements, handling each pre-token independently (no merges across pre-token boundaries).

Step 2: Apply the merges. We then take the sequence of vocabulary element merges created during BPE training, and apply it to our pre-tokens *in the same order of creation.*

Example (`bpe_encoding`): BPE encoding example

For example, suppose our input string is `'the cat ate'`, our vocabulary is `{0: b' ', 1: b'a', 2: b'c', 3: b'e', 4: b'h', 5: b't', 6: b'th', 7: b' c', 8: b' a', 9: b'the', 10: b' at'}`, and our learned merges are `[(b't', b'h'), (b' ', b'c'), (b' ', 'a'), (b'th', b'e'), (b' a', b't')]`. First, our pre-tokenizer would split this string into `['the', ' cat', ' ate']`. Then, we'll look at each pre-token and apply the BPE merges.

The first pre-token `'the'` is initially represented as `[b't', b'h', b'e']`. Looking at our list of merges, we identify the first applicable merge to be `(b't', b'h')`, and use that to transform the pre-token into `[b'th', b'e']`. Then, we go back to the list of merges and identify the next applicable merge to be `(b'th', b'e')`, which transforms the pre-token into `[b'the']`. Finally, looking back at the list of merges, we see that there are no more that apply to the string (since the entire pre-token has been merged into a single token), so we are done applying the BPE merges. The corresponding integer sequence is `[9]`.

Repeating this process for the remaining pre-tokens, we see that the pre-token `' cat'` is represented as `[b' c', b'a', b't']` after applying the BPE merges, which becomes the integer sequence `[7, 1, 5]`. The final pre-token `' ate'` is `[b' at', b'e']` after applying the BPE merges, which becomes the integer sequence `[10, 3]`. Thus, the final result of encoding our input string is `[9, 7, 1, 5, 10, 3]`.

Special tokens. Your tokenizer should be able to properly handle user-defined special tokens when encoding text (provided when constructing the tokenizer).

Memory considerations. Suppose we want to tokenize a large text file that we cannot fit in memory. To efficiently tokenize this large file (or any other stream of data), we need to break it up into manageable chunks and process each chunk in-turn, so that the memory complexity is constant as opposed to linear in the size of the text. In doing so, we need to make sure that a token doesn't cross chunk boundaries, else we'll get a different tokenization than the naïve method of tokenizing the entire sequence in-memory.

2.6.2 Decoding text

To decode a sequence of integer token IDs back to raw text, we can simply look up each ID's corresponding entries in the vocabulary (a byte sequence), concatenate them together, and then decode the bytes to a Unicode string. Note that input IDs are not guaranteed to map to valid Unicode strings (since a user could input any sequence of integer IDs). In the case that the input token IDs do not produce a valid Unicode string, you should replace the malformed bytes with the official Unicode replacement character `U+FFFD`.[3] The `errors` argument of `bytes.decode` controls how Unicode decoding errors are handled, and using `errors='replace'` will automatically replace malformed data with the replacement marker.

Problem (`tokenizer`): Implementing the tokenizer (15 points)

Deliverable: Implement a `Tokenizer` class that, given a vocabulary and a list of merges, encodes text into integer IDs and decodes integer IDs into text. Your tokenizer should also support user-provided special tokens (appending them to the vocabulary if they aren't already there). We recommend the following interface:

`def __init__(self, vocab, merges, special_tokens=None)` Construct a tokenizer from a given vocabulary, list of merges, and (optionally) a list of special tokens. This function should accept

[3] See en.wikipedia.org/wiki/Specials_(Unicode_block)#Replacement_character for more information about the Unicode replacement character.

the following parameters:

`vocab: dict[int, bytes]`

`merges: list[tuple[bytes, bytes]]`

`special_tokens: list[str] | None = None`

`def from_files(cls, vocab_filepath, merges_filepath, special_tokens=None)` Class method that constructs and return a `Tokenizer` from a serialized vocabulary and list of merges (in the same format that your BPE training code output) and (optionally) a list of special tokens. This method should accept the following additional parameters:

`vocab_filepath: str`

`merges_filepath: str`

`special_tokens: list[str] | None = None`

`def encode(self, text: str) -> list[int]` Encode an input text into a sequence of token IDs.

`def encode_iterable(self, iterable: Iterable[str]) -> Iterator[int]` Given an iterable of strings (e.g., a Python file handle), return a generator that lazily yields token IDs. This is required for memory-efficient tokenization of large files that we cannot directly load into memory.

`def decode(self, ids: list[int]) -> str` Decode a sequence of token IDs into text.

To test your `Tokenizer` against our provided tests, you will first need to implement the test adapter at `[adapters.get_tokenizer]`. Then, run `uv run pytest tests/test_tokenizer.py`. Your implementation should be able to pass all tests.

2.7 Experiments

Problem (`tokenizer_experiments`): Experiments with tokenizers (4 points)

(a) Sample 10 documents from TinyStories and OpenWebText. Using your previously-trained TinyStories and OpenWebText tokenizers (10K and 32K vocabulary size, respectively), encode these sampled documents into integer IDs. What is each tokenizer's compression ratio (bytes/token)?

Deliverable: A one-to-two sentence response.

(b) What happens if you tokenize your OpenWebText sample with the TinyStories tokenizer? Compare the compression ratio and/or qualitatively describe what happens.

Deliverable: A one-to-two sentence response.

(c) Estimate the throughput of your tokenizer (e.g., in bytes/second). How long would it take to tokenize the Pile dataset (825GB of text)?

Deliverable: A one-to-two sentence response.

(d) Using your TinyStories and OpenWebText tokenizers, encode the respective training and development datasets into a sequence of integer token IDs. We'll use this later to train our language model. We recommend serializing the token IDs as a NumPy array of datatype `uint16`. Why is `uint16` an appropriate choice?

Deliverable: A one-to-two sentence response.

~/Dropbox/Eric/Machine Learning/cs336/cs336-a1-main-brandon-snider/cs336_basics/train_bpe.py

```python
import os
import heapq
from typing import BinaryIO
import regex as re
import collections
import multiprocessing as mp
import time
import pickle
from functools import reduce

# Regex for coarse tokenization
PAT = re.compile(r"""'(?:[sdmt]|ll|ve|re)| ?\p{L}+| ?\p{N}+| ?[^\s\p{L}\p{N}]+|\s+(?!\S)|\s+""")


class ReverseLexOrderPair:
    """
    Encapsulates (bytes, bytes) so that in a min-heap, the "largest in normal lex order"
    is treated as the smallest. Ensures that tie frequencies pop in reverse lex order.
    """

    def __init__(self, pair: tuple[bytes, bytes]):
        self.pair = pair

    def __lt__(self, other: "ReverseLexOrderPair") -> bool:
        # Invert normal order: self < other if self is > other (so larger lex sorts first).
        return self.pair > other.pair

    def __eq__(self, other: "ReverseLexOrderPair") -> bool:
        return self.pair == other.pair


def find_chunk_boundaries(file: BinaryIO, desired_num_chunks: int, split_special_token: bytes) -> list[int]:
    """
    Find chunk boundaries by reading forward from guessed positions
    until split_special_token is found (or EOF). Ensures alignment.
    """
    assert isinstance(split_special_token, bytes), "Must represent special token as a bytestring"

    file.seek(0, os.SEEK_END)
    file_size = file.tell()
    file.seek(0)
    chunk_size = file_size // desired_num_chunks
```

```python

    # Initial boundary guesses (uniformly spaced); force last boundary at EOF
    chunk_boundaries = [i * chunk_size for i in range(desired_num_chunks + 1)]
    chunk_boundaries[-1] = file_size

    # This seems inefficient since it does extra work when the special token is not found in the mini chunk...
    # but I guess we never expect this to fail
    mini_chunk_size = 4096
    for bi in range(1, len(chunk_boundaries) - 1):
        pos = chunk_boundaries[bi]
        file.seek(pos)
        while True:
            mini_chunk = file.read(mini_chunk_size)
            if not mini_chunk:
                # If EOF is reached before finding split token
                chunk_boundaries[bi] = file_size
                break
            found_at = mini_chunk.find(split_special_token)
            if found_at != -1:
                # Found the split token; adjust boundary precisely
                chunk_boundaries[bi] = pos + found_at
                break
            pos += mini_chunk_size

    return sorted(set(chunk_boundaries)) #sorted turns a set -> list (that is sorted)


def pre_tokenize_chunk(chunk: str, special_pattern: re.Pattern | None) -> dict[tuple[bytes], int]:
    """Regex tokenizes the chunk. Splits first on special tokens, then uses PAT."""
    freqs: dict[tuple[bytes], int] = {}
    sub_chunks = special_pattern.split(chunk) if special_pattern else [chunk]

    for sub_chunk in sub_chunks:
        for match in PAT.finditer(sub_chunk):
            match_bytes = tuple(bytes([b]) for b in match.group().encode("UTF-8"))
            freqs[match_bytes] = freqs.get(match_bytes, 0) + 1

    return freqs


def merge_freq_dicts(dict1: dict[tuple[bytes], int], dict2: dict[tuple[bytes], int]) -> dict[tuple[bytes], int]:
    """Adds frequencies from dict2 into dict1."""
    result = dict1.copy()
    for key, value in dict2.items():
        result[key] = result.get(key, 0) + value
    return result


```

```python
def pre_tokenize(input_path: str, special_tokens: list[str]) -> dict[tuple[bytes], int]:
    """
    Splits a file into chunks aligned with <|endoftext|>, then tokenizes each chunk
    in parallel. Returns aggregated frequency dict.
    """
    num_processes = mp.cpu_count()
    pool = mp.Pool(processes=num_processes)
    chunk_freqs = []
    special_pattern = re.compile("|".join(re.escape(tok) for tok in special_tokens)) if special_tokens else None

    with open(input_path, "rb") as f:
        # Divide file into number of chunks matching number of cpu cores
        boundaries = find_chunk_boundaries(f, num_processes, b"<|endoftext|>")

        # Read each chunk in bytes, decode, then apply_async for parallel tokenization
        for start, end in zip(boundaries[:-1], boundaries[1:]):
            f.seek(start)
            chunk_bytes = f.read(end - start)
            chunk_str = chunk_bytes.decode("utf-8", errors="ignore")
            chunk_freqs.append(pool.apply_async(pre_tokenize_chunk, (chunk_str, special_pattern)))

    pool.close()
    pool.join()

    # Collect and merge partial results
    freq_dicts = [res.get() for res in chunk_freqs]
    combined_freqs = reduce(merge_freq_dicts, freq_dicts, {})
    return combined_freqs


def get_pair_freqs(
    freqs: dict[tuple[bytes], int],
) -> tuple[dict[tuple[bytes, bytes], int], dict[tuple[bytes, bytes], set[tuple[bytes]]]]:
    """
    Builds a pair-frequency table and reverse mapping (pair -> set of keys).
    """
    pair_freqs: dict[tuple[bytes, bytes], int] = collections.defaultdict(int)
    pairs_to_keys: dict[tuple[bytes, bytes], set[tuple[bytes]]] = collections.defaultdict(set)

    for symbols, freq in freqs.items():
        for i in range(len(symbols) - 1):
            pair = (symbols[i], symbols[i + 1])
            pair_freqs[pair] += freq
            pairs_to_keys[pair].add(symbols)

```

```python
    return pair_freqs, pairs_to_keys


def build_new_repr(old_repr: tuple[bytes], pair: tuple[bytes, bytes]) -> tuple[bytes]:
    """Replaces every occurrence of pair=(x,y) in old_repr with the merged symbol x+y."""
    new_symbols = []
    i = 0
    while i < len(old_repr):
        if i < len(old_repr) - 1 and old_repr[i] == pair[0] and old_repr[i + 1] == pair[1]:
            new_symbols.append(old_repr[i] + old_repr[i + 1])  # merges, e.g. b'A' + b'B' => b'AB'
            i += 2
        else:
            new_symbols.append(old_repr[i])
            i += 1
    return tuple(new_symbols)


def merge(
    freqs: dict[tuple[bytes], int],
    pair_freqs: dict[tuple[bytes, bytes], int],
    pairs_to_keys: dict[tuple[bytes, bytes], set[tuple[bytes]]],
    pair: tuple[bytes, bytes],
) -> set[tuple[bytes, bytes]]:
    """Merges 'pair' into freqs and updates pair_freqs & pairs_to_keys for all affected old/new keys."""
    changed_pairs = set()
    keys_to_modify = pairs_to_keys[pair].copy()

    for old_key in keys_to_modify:
        old_freq = freqs.pop(old_key)
        new_key = build_new_repr(old_key, pair)

        # Decrement frequencies in pair_freqs for old_key's adjacencies
        for i in range(len(old_key) - 1):
            left, right = old_key[i], old_key[i + 1]
            pair_freqs[left, right] -= old_freq
            changed_pairs.add((left, right))
            if pair_freqs[left, right] <= 0:
                del pair_freqs[left, right]
            pairs_to_keys[left, right].discard(old_key)

        # Increment frequencies for new_key's adjacencies
        for i in range(len(new_key) - 1):
            left, right = new_key[i], new_key[i + 1]
            pair_freqs[left, right] += old_freq
            changed_pairs.add((left, right))
            pairs_to_keys[left, right].add(new_key)
```

```python

        # Put new_key back with updated freq
        freqs[new_key] = freqs.get(new_key, 0) + old_freq

    pairs_to_keys[pair] = set()
    return changed_pairs


def write_merges(merges, outpath):
    """Pickle the merges list to a binary file."""
    os.makedirs(os.path.dirname(outpath), exist_ok=True)
    with open(outpath, "wb") as f:
        pickle.dump(merges, f)
    print(f"Saved {len(merges)} merges to {outpath}")


def write_vocab(vocab, outpath):
    """Pickle the vocab dict to a binary file."""
    os.makedirs(os.path.dirname(outpath), exist_ok=True)
    with open(outpath, "wb") as f:
        pickle.dump(vocab, f)
    print(f"Saved vocabulary with {len(vocab)} tokens to {outpath}")


def train_bpe(
    input_path: str,
    vocab_size: int,
    special_tokens: list[str],
    merges_outpath: str = None,
    vocab_outpath: str = None,
) -> tuple[dict[int, bytes], list[tuple[bytes, bytes]]]:
    """
    Trains byte-level BPE on a text file, returning:
      - vocab: dict[int, bytes]
      - merges: list of merged pairs
    """
    train_start_time = time.time()

    # Initialize special tokens and single-byte tokens
    initial_tokens = [tok.encode("UTF-8") for tok in special_tokens] + [bytes([i]) for i in range(256)]
    vocab = {i: token for i, token in enumerate(initial_tokens)}
    merges = []

    print("Pre-tokenize: start")
    start_time = time.time()
    freqs = pre_tokenize(input_path, special_tokens)
    print(f"Pre-tokenize: finished in {time.time() - start_time:.2f}s")

    print("Initial pair frequencies: start")
```

```python
    start_time = time.time()
    pair_freqs, pairs_to_keys = get_pair_freqs(freqs)

    # Build a max-heap by pushing negative frequencies, ReverseLexOrderPair resolves ties with max lexicographic order (hence the reverse)
    pair_heap = []
    for p, f in pair_freqs.items():
        if f > 0:
            heapq.heappush(pair_heap, (-f, ReverseLexOrderPair(p), p))

    print(f"Initial pair frequencies: finished in {time.time() - start_time:.2f}s")

    n_initial_tokens = len(initial_tokens)
    n_merges = vocab_size - n_initial_tokens

    print("Merge: start")
    start_time = time.time()

    for i in range(n_initial_tokens, n_initial_tokens + n_merges):
        if not pair_heap:
            break

        # Pop until we find the top pair that still matches pair_freqs
        while pair_heap:
            neg_freq, _, top_pair = heapq.heappop(pair_heap)
            freq = -neg_freq
            if pair_freqs.get(top_pair, 0) == freq:
                pair = top_pair
                break
            if top_pair in pair_freqs and pair_freqs[top_pair] > 0:
                heapq.heappush(pair_heap, (-pair_freqs[top_pair], ReverseLexOrderPair(top_pair), top_pair))
        else:
            # If pair_heap is empty after the loop, we are done
            break

        if pair_freqs.get(pair, 0) <= 0:
            break

        # Add this new merge token to vocab and record the merge
        vocab[i] = pair[0] + pair[1]
        merges.append(pair)

        # Merge in freqs, then update the heap for pairs changed by this merge
        changed_pairs = merge(freqs, pair_freqs, pairs_to_keys, pair)
        for cp in changed_pairs:
            if cp in pair_freqs and pair_freqs[cp] > 0:
                heapq.heappush(pair_heap, (-pair_freqs[cp], ReverseLexOrderPair(cp), cp))

        # Print progress every 100 merges or at the last iteration
```

```python
        if ((i > n_initial_tokens) and ((i - n_initial_tokens + 1) % 100 == 0)) or (
            i == n_initial_tokens + n_merges - 1
        ):
            print(
                f"{i - n_initial_tokens + 1}/{n_merges} merges completed (merge runtime: {time.time() - start_time:.2f}s)"
            )

    print(f"Merges completed in {time.time() - start_time:.2f}s")
    print(f"Training completed in {time.time() - train_start_time:.2f}s")

    # Optionally save merges and vocab
    if merges_outpath:
        write_merges(merges, merges_outpath)
    if vocab_outpath:
        write_vocab(vocab, vocab_outpath)

    return vocab, merges


if __name__ == "__main__":
    (vocab, merges) = train_bpe(
        input_path="./data/TinyStoriesV2-GPT4-valid.txt",
        vocab_size=10000,
        special_tokens=["<|endoftext|>"],
        merges_outpath="./out/ts-valid-merges-2.txt",
        vocab_outpath="./out/ts-valid-vocab-2.txt",
    )

```

~/Dropbox/Eric/Machine Learning/cs336/cs336-a1-main-brandon-snider/cs336_basics/tokenizer.py

```python
from collections.abc import Iterable, Iterator
import regex as re
import pickle

from cs336_basics import train_bpe


class Tokenizer:
    def __init__(
        self, vocab: dict[int, bytes], merges: list[tuple[bytes, bytes]], special_tokens: list[str] | None = None
    ):
        """
        Constructs a tokenizer from a vocab, list of merges, and (optionally) list of special tokens.
        """
        self.vocab = vocab
        self.vocab_inv = {v: k for k, v in vocab.items()}
        self.merges = merges
        self.merges_dict = {merge: i for i, merge in enumerate(merges)}
        self.encode_cache = {}
        self.cache_hits = 0

        self.pretokenize_pattern = re.compile(train_bpe.PAT)

        # If there are extra special tokens than those used to construct the vocabulary initially
        if special_tokens:
            self.special_tokens = sorted(special_tokens, key=len, reverse=True)
            self.special_pattern = "(" + "|".join(re.escape(k) for k in self.special_tokens) + ")"

            next_id = max(self.vocab.keys()) + 1
            for token in special_tokens:
                token_bytes = token.encode("UTF-8")
                if token_bytes not in self.vocab_inv:
                    self.vocab[next_id] = token_bytes
                    self.vocab_inv[token_bytes] = next_id
                    next_id += 1
        else:
            self.special_tokens = None
            self.special_pattern = None

    @classmethod
    def from_files(cls, vocab_filepath: str, merges_filepath: str, special_tokens: list[str] | None = None):
        """
```

```
        Constructs a Tokenizer from a serialized vocab, list of merges, and (optionally) list of special tokens.
        """
        with open(vocab_filepath, "rb") as f:
            vocab = pickle.load(f)

        with open(merges_filepath, "rb") as f:
            merges = pickle.load(f)

        # This calls the regular constructor, called as Tokenizer.from_files(vocab_filepath, merges_filepath, special_tokens)
        return cls(vocab, merges, special_tokens)

    def encode(self, text: str) -> list[int]:
        """Encodes an input text into a sequence of token IDs, handling special tokens."""
        if not self.special_tokens:
            return self._encode_chunk(text)

        # If we have special tokens, split on them, keeping delimiters
        special_chunks = re.split(self.special_pattern, text)

        ids = []
        for part in special_chunks:
            if part in self.special_tokens:
                # this is a special token
                ids.append(self.vocab_inv[part.encode("UTF-8")])
            else:
                # this is ordinary text
                ids.extend(self._encode_chunk(part))
        return ids

    def _encode_chunk(self, text: str) -> list[int]:
        """Encodes an input text chunk into a sequence of token IDs."""
        pretokens = self._pretokenize(text)
        pretoken_reprs: dict[str, list[bytes]] = {}

        ids = []

        # Merge each pretoken using the BPE rules, in ascending rank order
        for p in pretokens:
            # Check if we have already merged this subword previously
            if p in self.encode_cache:
                ids.extend(self.encode_cache[p])
                self.cache_hits += 1
            else:
                # Each character → single bytes: e.g. "abc" -> [b'a', b'b', b'c']
                if p not in pretoken_reprs:
                    match_bytes = list(bytes([b]) for b in p.encode("UTF-8"))
                    pretoken_reprs[p] = match_bytes

```

```
                merged = self._merge_subword(pretoken_reprs[p])
                token_ids = [self.vocab_inv[subword] for subword in merged]
                self.encode_cache[p] = token_ids
                ids.extend(token_ids)

        return ids

    def _merge_subword(self, rep: list[bytes]) -> list[bytes]:
        """
        Given a list of subword units (bytes), repeatedly merges adjacent pairs
        in ascending rank order until no more merges are found.
        """
        while True:
            best_rank = float("inf")
            best_idx = None

            # Scan adjacent pairs in current rep, finding the earliest merge
            for i in range(len(rep) - 1):
                pair = (rep[i], rep[i + 1])
                rank = self.merges_dict.get(pair)
                if rank is not None and rank < best_rank:
                    best_rank = rank
                    best_idx = i

            # If no merges found, we're done
            if best_idx is None:
                return rep

            # Merge the best pair
            merged = rep[best_idx] + rep[best_idx + 1]  # Concatenate bytes
            rep = rep[:best_idx] + [merged] + rep[best_idx + 2 :]

    def encode_iterable(self, iterable: Iterable[str]) -> Iterator[int]:
        """Yields token IDs lazily from an iterable of strings (e.g., a file handle)."""
        for text in iterable:
            yield from self.encode(text)

    def decode(self, ids: list[int]) -> str:
        """Decodes a sequence of token IDs into text."""
        text = b"".join(self.vocab[id] for id in ids)
        return text.decode("UTF-8", errors="replace")

    def _pretokenize(self, text: str) -> list[str]:
        """Splits text into 'pretokens' and builds an initial byte representation for each."""
        pretokens: list[str] = []

        for match in self.pretokenize_pattern.finditer(text):
            match_str = match.group()
            pretokens.append(match_str)
```

```
        return pretokens

```

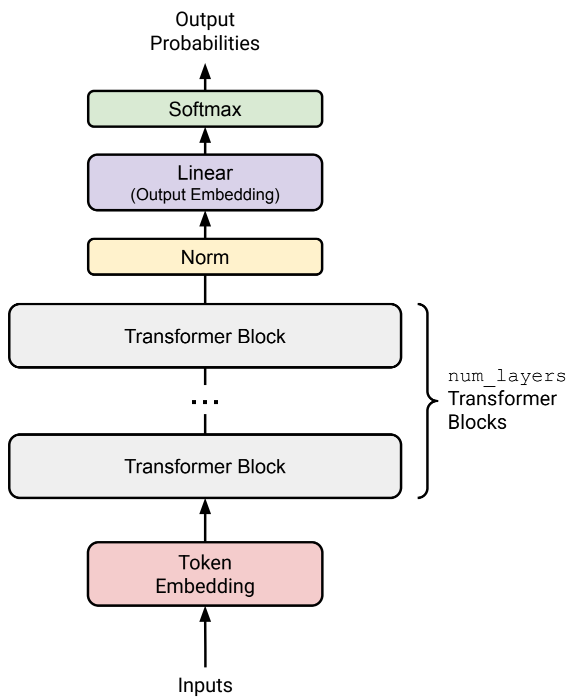

Figure 1: An overview of our Transformer language model.

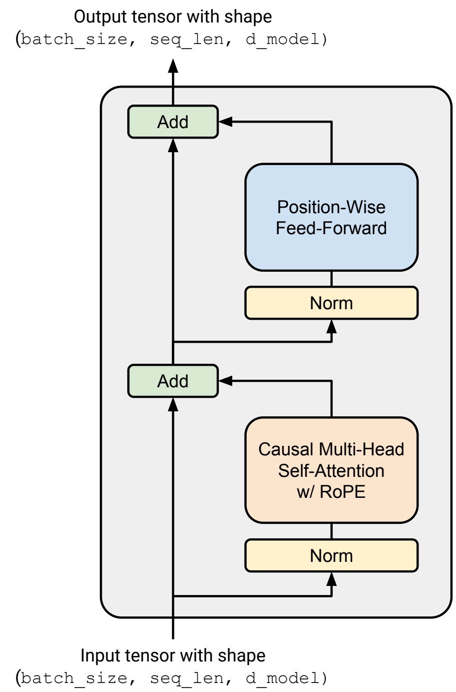

Figure 2: A pre-norm Transformer block.

3 Transformer Language Model Architecture

A language model takes as input a batched sequence of integer token IDs (i.e., `torch.Tensor` of shape `(batch_size, sequence_length)`), and returns a (batched) normalized probability distribution over the vocabulary (i.e., a PyTorch Tensor of shape `(batch_size, sequence_length, vocab_size)`), where the predicted distribution is over the next word for each input token. When training the language model, we use these next-word predictions to calculate the cross-entropy loss between the actual next word and the predicted next word. When generating text from the language model during inference, we take the predicted next-word distribution from the final time step (i.e., the last item in the sequence) to generate the next token in the sequence (e.g., by taking the token with the highest probability, sampling from the distribution, etc.), add the generated token to the input sequence, and repeat.

In this part of the assignment, you will build this Transformer language model from scratch. We will begin with a high-level description of the model before progressively detailing the individual components.

3.1 Transformer LM

Given a sequence of token IDs, the Transformer language model uses an input embedding to convert token IDs to dense vectors, passes the embedded tokens through `num_layers` Transformer blocks, and then applies a learned linear projection (the "output embedding" or "LM head") to produce the predicted next-token logits. See Figure 1 for a schematic representation.

3.1.1 Token Embeddings

In the very first step, the Transformer *embeds* the (batched) sequence of token IDs into a sequence of vectors containing information on the token identity (red blocks in Figure 1).

More specifically, given a sequence of token IDs, the Transformer language model uses a token embedding layer to produce a sequence of vectors. Each embedding layer takes in a tensor of integers of shape (batch_size, sequence_length) and produces a sequence of vectors of shape (batch_size, sequence_length, d_model).

3.1.2 Pre-norm Transformer Block

After embedding, the activations are processed by several identically structured neural net layers. A standard decoder-only Transformer language model consists of num_layers identical layers (commonly called Transformer "blocks"). Each Transformer block takes in an input of shape (batch_size, sequence_length, d_model) and returns an output of shape (batch_size, sequence_length, d_model). Each block aggregates information across the sequence (via self-attention) and non-linearly transforms it (via the feed-forward layers).

3.2 Output Normalization and Embedding

After num_layers Transformer blocks, we will take the final activations and turn them into a distribution over the vocabulary.

We will implement the "pre-norm" Transformer block (detailed in §3.5), which additionally requires the use of layer normalization (detailed below) after the final Transformer block to ensure its outputs are properly scaled.

After this normalization, we will use a standard learned linear transformation to convert the output of the Transformer blocks into predicted next-token logits (see, e.g., Radford et al. [2018] equation 2).

3.3 Remark: Batching, Einsum and Efficient Computation

Throughout the Transformer, we will be performing the same computation applied to many batch-like inputs. Here are a few examples:

- **Elements of a batch**: we apply the same Transformer forward operation on each batch element.
- **Sequence length**: the "position-wise" operations like RMSNorm and feed-forward operate identically on each position of a sequence.
- **Attention heads**: the attention operation is batched across attention heads in a "multi-headed" attention operation.

It is useful to have an ergonomic way of performing such operations in a way that fully utilizes the GPU, and is easy to read and understand. Many PyTorch operations can take in excess "batch-like" dimensions at the start of a tensor and repeat/broadcast the operation across these dimensions efficiently.

For instance, say we are doing a position-wise, batched operation. We have a "data tensor" D of shape (batch_size, sequence_length, d_model), and we would like to do a batched vector-matrix multiply against a matrix A of shape (d_model, d_model). In this case, D @ A will do a batched matrix multiply, which is an efficient primitive in PyTorch, where the (batch_size, sequence_length) dimensions are batched over.

Because of this, it is helpful to assume that your functions may be given additional batch-like dimensions and to keep those dimensions at the start of the PyTorch shape. To organize tensors so they can be batched in this manner, they might need to be shaped using many steps of view, reshape and transpose. This can be a bit of a pain, and it often gets hard to read what the code is doing and what the shapes of your tensors are.

A more ergonomic option is to use *einsum notation* within torch.einsum, or rather use framework agnostic libraries like einops or einx. The two key ops are einsum, which can do tensor contractions with arbitrary dimensions of input tensors, and rearrange, which can reorder, concatenate, and split arbitrary dimensions. It turns out almost all operations in machine learning are some combination of dimension juggling and tensor contraction with the occasional (usually pointwise) nonlinear function. This means that a lot of your code can be more readable and flexible when using einsum notation.

We **strongly** recommend learning and using einsum notation for the class. Students who have not been exposed to einsum notation before should use `einops` (docs here), and students who are already comfortable with `einops` should learn the more general `einx` (here).[4] Both packages are already installed in the environment we've supplied.

Here we give some examples of how einsum notation can be used. These are a supplement to the documentation for einops, which you should read first.

Example (`einstein_example1`): Batched matrix multiplication with `einops.einsum`

```
import torch
from einops import rearrange, einsum

## Basic implementation
Y = D @ A.T
# Hard to tell the input and output shapes and what they mean.
# What shapes can D and A have, and do any of these have unexpected behavior?

## Einsum is self-documenting and robust
#                          D                 A     ->          Y
Y = einsum(D, A, "batch sequence d_in, d_out d_in -> batch sequence d_out")

## Or, a batched version where D can have any leading dimensions but A is constrained.
Y = einsum(D, A, "... d_in, d_out d_in -> ... d_out")
```

Example (`einstein_example2`): Broadcasted operations with `einops.rearrange`

We have a batch of images, and for each image we want to generate 10 dimmed versions based on some scaling factor:

```
images = torch.randn(64, 128, 128, 3)  # (batch, height, width, channel)
dim_by = torch.linspace(start=0.0, end=1.0, steps=10)

## Reshape and multiply
dim_value = rearrange(dim_by,    "dim_value              -> 1 dim_value 1 1 1")
images_rearr = rearrange(images, "b height width channel -> b 1 height width channel")
dimmed_images = images_rearr * dim_value

## Or in one go:
dimmed_images = einsum(
    images, dim_by,
    "batch height width channel, dim_value -> batch dim_value height width channel"
)
```

[4]It's worth noting that while `einops` has a great amount of support, `einx` is not as battle-tested. You should feel free to fall back to using `einops` with some more plain PyTorch if you find any limitations or bugs in `einx`.

Example (`einstein_example3`): Pixel mixing with `einops.rearrange`

Suppose we have a batch of images represented as a tensor of shape `(batch, height, width, channel)`, and we want to perform a linear transformation across all pixels of the image, but this transformation should happen independently for each channel. Our linear transformation is represented as a matrix B of shape `(height` $\times$ `width, height` $\times$ `width)`.

```
channels_last = torch.randn(64, 32, 32, 3)  # (batch, height, width, channel)
B = torch.randn(32*32, 32*32)

## Rearrange an image tensor for mixing across all pixels
channels_last_flat = channels_last.view(
    -1, channels_last.size(1) * channels_last.size(2), channels_last.size(3)
)
channels_first_flat = channels_last_flat.transpose(1, 2)

channels_first_flat_transformed = channels_first_flat @ B.T

channels_last_flat_transformed = channels_first_flat_transformed.transpose(1, 2)

channels_last_transformed = channels_last_flat_transformed.view(*channels_last.shape)
```

Instead, using `einops`:

```
height = width = 32
## Rearrange replaces clunky torch view + transpose
channels_first = rearrange(
    channels_last,
    "batch height width channel -> batch channel (height width)"
)
channels_first_transformed = einsum(
    channels_first, B,
    "batch channel pixel_in, pixel_out pixel_in -> batch channel pixel_out"
)
channels_last_transformed = rearrange(
    channels_first_transformed,
    "batch channel (height width) -> batch height width channel",
    height=height, width=width
)
```

Or, if you're feeling crazy: all in one go using `einx.dot` (`einx` equivalent of `einops.einsum`)

```
height = width = 32
channels_last_transformed = einx.dot(
    "batch row_in col_in channel, (row_out col_out) (row_in col_in)"
    "-> batch row_out col_out channel",
    channels_last, B,
    col_in=width, col_out=width
)
```

The first implementation here could be improved by placing comments before and after to indicate what the input and output shapes are, but this is clunky and susceptible to bugs. With einsum notation, documentation *is* implementation!

Einsum notation can handle arbitrary input batching dimensions, but also has the key benefit of being *self-documenting*. It's much clearer what the relevant shapes of your input and output tensors are in code that uses einsum notation. For the remaining tensors, you can consider using Tensor type hints, for instance using the `jaxtyping` library (not specific to Jax).

We will talk more about the performance implications of using einsum notation in assignment 2, but for now know that they're almost always better than the alternative!

3.3.1 Mathematical Notation and Memory Ordering

Many machine learning papers use row vectors in their notation, which result in representations that mesh well with the row-major memory ordering used by default in NumPy and PyTorch. With row vectors, a linear transformation looks like

$$y = xW^\top, \tag{1}$$

for row-major $W \in \mathbb{R}^{d_{\text{out}} \times d_{\text{in}}}$ and row-vector $x \in \mathbb{R}^{1 \times d_{\text{in}}}$.

In linear algebra it's generally more common to use column vectors, where linear transformations look like

$$y = Wx, \tag{2}$$

given a row-major $W \in \mathbb{R}^{d_{\text{out}} \times d_{\text{in}}}$ and column-vector $x \in \mathbb{R}^{d_{\text{in}}}$. **We will use column vectors** for mathematical notation in this assignment, as it is generally easier to follow the math this way. You should keep in mind that if you want to use plain matrix multiplication notation, you will have to apply matrices using the row vector convention, since PyTorch uses row-major memory ordering. If you use `einsum` for your matrix operations, this should be a non-issue.

3.4 Basic Building Blocks: Linear and Embedding Modules

3.4.1 Parameter Initialization

Training neural networks effectively often requires careful initialization of the model parameters—bad initializations can lead to undesirable behavior such as vanishing or exploding gradients. Pre-norm transformers are unusually robust to initializations, but they can still have a siginificant impact on training speed and convergence. Since this assignment is already long, we will save the details for assignment 3, and instead give you some approximate initializations that should work well for most cases. For now, use:

- Linear weights: $\mathcal{N}\left(\mu = 0, \sigma^2 = \frac{2}{d_{\text{in}} + d_{\text{out}}}\right)$ truncated at $[-3\sigma, 3\sigma]$.
- Embedding: $\mathcal{N}\left(\mu = 0, \sigma^2 = 1\right)$ truncated at $[-3, 3]$
- RMSNorm: $\mathbb{1}$

You should use `torch.nn.init.trunc_normal_` to initialize the truncated normal weights.

3.4.2 Linear Module

Linear layers are a fundamental building block of Transformers and neural nets in general. First, you will implement your own `Linear` class that inherits from `torch.nn.Module` and performs a linear transformation:

$$y = Wx. \tag{3}$$

Note that we do not include a bias term, following most modern LLMs.

Problem (linear): Implementing the linear module (1 point)

Deliverable: Implement a `Linear` class that inherits from `torch.nn.Module` and performs a linear transformation. Your implementation should follow the interface of PyTorch's built-in `nn.Linear` module, except for not having a `bias` argument or parameter. We recommend the following interface:

`def __init__(self, in_features, out_features, device=None, dtype=None)` Construct a linear transformation module. This function should accept the following parameters:

`in_features: int` final dimension of the input

`out_features: int` final dimension of the output

`device: torch.device | None = None` Device to store the parameters on

`dtype: torch.dtype | None = None` Data type of the parameters

`def forward(self, x: torch.Tensor) -> torch.Tensor` Apply the linear transformation to the input.

Make sure to:

- subclass `nn.Module`
- call the superclass constructor
- construct and store your parameter as W (not $W^\top$) for memory ordering reasons, putting it in an `nn.Parameter`
- of course, don't use `nn.Linear` or `nn.functional.linear`

For initializations, use the settings from above along with `torch.nn.init.trunc_normal_` to initialize the weights.
To test your `Linear` module, implement the test adapter at `[adapters.run_linear]`. The adapter should load the given weights into your `Linear` module. You can use `Module.load_state_dict` for this purpose. Then, run `uv run pytest -k test_linear`.

3.4.3 Embedding Module

As discussed above, the first layer of the Transformer is an embedding layer that maps integer token IDs into a vector space of dimension `d_model`. We will implement a custom `Embedding` class that inherits from `torch.nn.Module` (so you should not use `nn.Embedding`). The `forward` method should select the embedding vector for each token ID by indexing into an embedding matrix of shape `(vocab_size, d_model)` using a `torch.LongTensor` of token IDs with shape `(batch_size, sequence_length)`.

Problem (embedding): Implement the embedding module (1 point)

Deliverable: Implement the `Embedding` class that inherits from `torch.nn.Module` and performs an embedding lookup. Your implementation should follow the interface of PyTorch's built-in `nn.Embedding` module. We recommend the following interface:

`def __init__(self, num_embeddings, embedding_dim, device=None, dtype=None)` Construct an embedding module. This function should accept the following parameters:

`num_embeddings: int` Size of the vocabulary `embedding_dim: int` Dimension of the embedding vectors, i.e., d_{model}

`device: torch.device | None = None` Device to store the parameters on

`dtype: torch.dtype | None = None` Data type of the parameters

`def forward(self, token_ids: torch.Tensor) -> torch.Tensor` Lookup the embedding vectors for the given token IDs.

Make sure to:

- subclass `nn.Module`
- call the superclass constructor
- initialize your embedding matrix as a `nn.Parameter`
- store the embedding matrix with the `d_model` being the final dimension
- of course, don't use `nn.Embedding` or `nn.functional.embedding`

Again, use the settings from above for initialization, and use `torch.nn.init.trunc_normal_` to initialize the weights.
To test your implementation, implement the test adapter at `[adapters.run_embedding]`. Then, run `uv run pytest -k test_embedding`.

3.5 Pre-Norm Transformer Block

Each Transformer block has two sub-layers: a multi-head self-attention mechanism and a position-wise feed-forward network (Vaswani et al., 2017, section 3.1).

In the original Transformer paper, the model uses a residual connection around each of the two sub-layers, followed by layer normalization. This architecture is commonly known as the "post-norm" Transformer, since layer normalization is applied to the sublayer output. However, a variety of work has found that moving layer normalization from the output of each sub-layer to the input of each sub-layer (with an additional layer normalization after the final Transformer block) improves Transformer training stability [Nguyen and Salazar, 2019, Xiong et al., 2020]—see Figure 2 for a visual representation of this "pre-norm" Transformer block. The output of each Transformer block sub-layer is then added to the sub-layer input via the residual connection (Vaswani et al., 2017, section 5.4). An intuition for pre-norm is that there is a clean "residual stream" without any normalization going from the input embeddings to the final output of the Transformer, which is purported to improve gradient flow. This pre-norm Transformer is now the standard used in language models today (e.g., GPT-3, LLaMA, PaLM, etc.), so we will implement this variant. We will walk through each of the components of a pre-norm Transformer block, implementing them in sequence.

3.5.1 Root Mean Square Layer Normalization

The original Transformer implementation of Vaswani et al. [2017] uses layer normalization [Ba et al., 2016] to normalize activations. Following Touvron et al. [2023], we will use root mean square layer normalization (RMSNorm; Zhang and Sennrich, 2019, equation 4) for layer normalization. Given a vector $a \in \mathbb{R}^{d_{\text{model}}}$ of activations, RMSNorm will rescale each activation a_i as follows:

$$\text{RMSNorm}(a_i) = \frac{a_i}{\text{RMS}(a)} g_i, \tag{4}$$

where $\text{RMS}(a) = \sqrt{\frac{1}{d_{\text{model}}} \sum_{i=1}^{d_{\text{model}}} a_i^2 + \varepsilon}$. Here, g_i is a learnable "gain" parameter (there are `d_model` such parameters total), and ε is a hyperparameter that is often fixed at 1e-5.

You should upcast your input to torch.float32 to prevent overflow when you square the input. Overall, your forward method should look like:

```
in_dtype = x.dtype
x = x.to(torch.float32)

# Your code here performing RMSNorm
...
result = ...

# Return the result in the original dtype
return result.to(in_dtype)
```

Problem (rmsnorm): Root Mean Square Layer Normalization (1 point)

Deliverable: Implement RMSNorm as a torch.nn.Module. We recommend the following interface:

def __init__(self, d_model: int, eps: float = 1e-5, device=None, dtype=None)
Construct the RMSNorm module. This function should accept the following parameters:

- **d_model: int** Hidden dimension of the model
- **eps: float = 1e-5** Epsilon value for numerical stability
- **device: torch.device | None = None** Device to store the parameters on
- **dtype: torch.dtype | None = None** Data type of the parameters

def forward(self, x: torch.Tensor) -> torch.Tensor Process an input tensor of shape (batch_size, sequence_length, d_model) and return a tensor of the same shape.

Note: Remember to upcast your input to torch.float32 before performing the normalization (and later downcast to the original dtype), as described above.
To test your implementation, implement the test adapter at [adapters.run_rmsnorm]. Then, run uv run pytest -k test_rmsnorm.

3.5.2 Position-Wise Feed-Forward Network

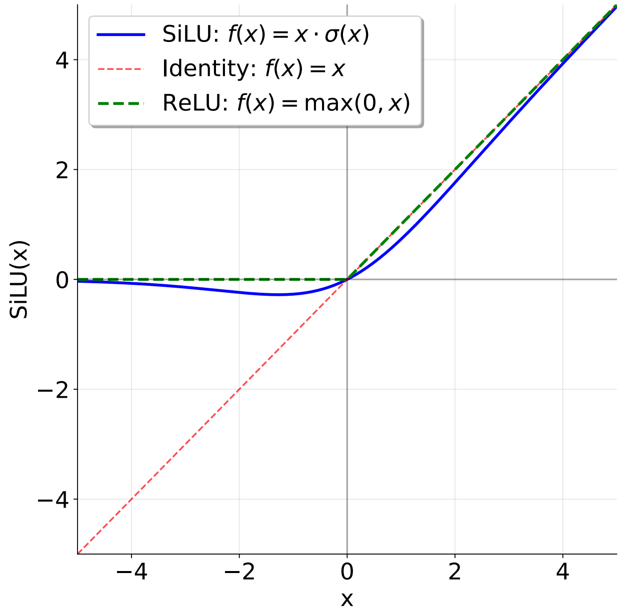

Figure 3: Comparing the SiLU (aka Swish) and ReLU activation functions.

In the original Transformer paper (section 3.3 of Vaswani et al. [2017]), the Transformer feed-forward network consists of two linear transformations with a ReLU activation ($\text{ReLU}(x) = \max(0, x)$) between them. The dimensionality of the inner feed-forward layer is typically 4x the input dimensionality.

However, modern language models tend to incorporate two main changes compared to this original design: they use another activation function and employ a gating mechanism. Specifically, we will implement the "SwiGLU" activation function adopted in LLMs like Llama 3 [Grattafiori et al., 2024] and Qwen 2.5 [Yang et al., 2024], which combines the SiLU (often called Swish) activation with a gating mechanism called a Gated Linear Unit (GLU). We will also omit the bias terms sometimes used in linear layers, following most modern LLMs since PaLM [Chowdhery et al., 2022] and LLaMA [Touvron et al., 2023].

The SiLU or Swish activation function [Hendrycks and Gimpel, 2016, Elfwing et al., 2017] is defined as follows:

$$\text{SiLU}(x) = x \cdot \sigma(x) = \frac{x}{1 + e^{-x}} \tag{5}$$

As can be seen in Figure 3, the SiLU activation function is similar to the ReLU activation function, but is smooth at zero.

Gated Linear Units (GLUs) were originally defined by Dauphin et al. [2017] as the element-wise product of a linear transformation passed through a sigmoid function and another linear transformation:

$$\text{GLU}(x, W_1, W_2) = \sigma(W_1 x) \odot W_2 x, \tag{6}$$

where $\odot$ represents element-wise multiplication. Gated Linear Units are suggested to "reduce the vanishing gradient problem for deep architectures by providing a linear path for the gradients while retaining non-linear capabilities."

Putting the SiLU/Swish and GLU together, we get the SwiGLU, which we will use for our feed-forward networks:

$$\text{FFN}(x) = \text{SwiGLU}(x, W_1, W_2, W_3) = W_2(\text{SiLU}(W_1 x) \odot W_3 x), \tag{7}$$

where $x \in \mathbb{R}^{d_{\text{model}}}$, $W_1, W_3 \in \mathbb{R}^{d_{\text{ff}} \times d_{\text{model}}}$, $W_2 \in \mathbb{R}^{d_{\text{model}} \times d_{\text{ff}}}$, and canonically, $d_{\text{ff}} = \frac{8}{3} d_{\text{model}}$.

Shazeer [2020] first proposed combining the SiLU/Swish activation with GLUs and conducted experiments showing that SwiGLU outperforms baselines like ReLU and SiLU (without gating) on language modeling tasks. Later in the assignment, you will compare SwiGLU and SiLU. Though we've mentioned some heuristic arguments for these components (and the papers provide more supporting evidence), it's good to keep an empirical perspective: a now famous quote from Shazeer's paper is

> We offer no explanation as to why these architectures seem to work; we attribute their success, as all else, to divine benevolence.

Problem (`positionwise_feedforward`): Implement the position-wise feed-forward network (2 points)

Deliverable: Implement the SwiGLU feed-forward network, composed of a SiLU activation function and a GLU.

Note: in this particular case, you should feel free to use `torch.sigmoid` in your implementation for numerical stability.

You should set d_{ff} to approximately $\frac{8}{3} \times d_{\text{model}}$ in your implementation, while ensuring that the dimensionality of the inner feed-forward layer is a multiple of 64 to make good use of your hardware. To test your implementation against our provided tests, you will need to implement the test adapter at `[adapters.run_swiglu]`. Then, run `uv run pytest -k test_swiglu` to test your implementation.

3.5.3 Relative Positional Embeddings

To inject positional information into the model, we will implement Rotary Position Embeddings [Su et al., 2021], often called RoPE. For a given query token $q^{(i)} = W_q x^{(i)} \in \mathbb{R}^d$ at token position i, we will apply a pairwise rotation matrix R^i, giving us $q'^{(i)} = R^i q^{(i)} = R^i W_q x^{(i)}$. Here, R^i will rotate pairs of embedding elements $q^{(i)}_{2k-1:2k}$ as 2d vectors by the angle $\theta_{i,k} = \frac{i}{\Theta^{2k/d}}$ for $k \in \{1, \ldots, d/2\}$ and some constant Θ. Thus, we can consider R^i to be a block-diagonal matrix of size $d \times d$, with blocks R^i_k for $k \in \{1, \ldots, d/2\}$, with

$$R^i_k = \begin{bmatrix} \cos(\theta_{i,k}) & -\sin(\theta_{i,k}) \\ \sin(\theta_{i,k}) & \cos(\theta_{i,k}) \end{bmatrix}. \tag{8}$$

Thus we get the full rotation matrix

Should k start with 0 or with 1?

It probably doesn't matter…

d = d_k

$$R^i = \begin{bmatrix} R^i_1 & 0 & 0 & \ldots & 0 \\ 0 & R^i_2 & 0 & \ldots & 0 \\ 0 & 0 & R^i_3 & \ldots & 0 \\ \vdots & \vdots & \vdots & \ddots & \vdots \\ 0 & 0 & 0 & \ldots & R^i_{d/2} \end{bmatrix}, \tag{9}$$

where 0s represent 2×2 zero matrices. While one could construct the full $d \times d$ matrix, a good solution should use the properties of this matrix to implement the transformation more efficiently. Since we only care about the relative rotation of tokens within a given sequence, we can reuse the values we compute for $\cos(\theta_{i,k})$ and $\sin(\theta_{i,k})$ across layers, and different batches. If you would like to optimize it, you may use a single RoPE module referenced by all layers, and it can have a 2d pre-computed buffer of sin and cos values created during init with `self.register_buffer(persistent=False)`, instead of a `nn.Parameter` (because we do not want to learn these fixed cosine and sine values). The exact same rotation process we did for our $q^{(i)}$ is then done for $k^{(j)}$, rotating by the corresponding R^j. Notice that this layer has no learnable parameters.

Problem (rope): Implement RoPE (2 points)

Deliverable: Implement a class `RotaryPositionalEmbedding` that applies RoPE to the input tensor.

The following interface is recommended:

`def __init__(self, theta: float, d_k: int, max_seq_len: int, device=None)` Construct the RoPE module and create buffers if needed.

`theta: float` Θ value for the RoPE

`d_k: int` dimension of query and key vectors

`max_seq_len: int` Maximum sequence length that will be inputted

`device: torch.device | None = None` Device to store the buffer on

`def forward(self, x: torch.Tensor, token_positions: torch.Tensor) -> torch.Tensor` Process an input tensor of shape `(..., seq_len, d_k)` and return a tensor of the same shape. Note that you should tolerate x with an arbitrary number of batch dimensions. You should assume that the token positions are a tensor of shape `(..., seq_len)` specifying the token positions of x along the sequence dimension.

You should use the token positions to slice your (possibly precomputed) cos and sin tensors along the sequence dimension.

To test your implementation, complete [adapters.run_rope] and make sure it passes `uv run pytest -k test_rope`.

3.5.4 Scaled Dot-Product Attention

We will now implement scaled dot-product attention as described in Vaswani et al. [2017] (section 3.2.1). As a preliminary step, the definition of the Attention operation will make use of softmax, an operation that takes an unnormalized vector of scores and turns it into a normalized distribution:

$$\mathrm{softmax}(v)_i = \frac{\exp(v_i)}{\sum_{j=1}^{n} \exp(v_j)}. \tag{10}$$

Note that $\exp(v_i)$ can become `inf` for large values (then, `inf`/`inf` = `NaN`). We can avoid this by noticing that the softmax operation is invariant to adding any constant c to all inputs. We can leverage this property for numerical stability—typically, we will subtract the largest entry of o_i from all elements of o_i, making the new largest entry 0. You will now implement softmax, using this trick for numerical stability.

Problem (`softmax`): Implement softmax (1 point)

Deliverable: Write a function to apply the softmax operation on a tensor. Your function should take two parameters: a tensor and a *dimension* i, and apply softmax to the i-th dimension of the input tensor. The output tensor should have the same shape as the input tensor, but its i-th dimension will now have a normalized probability distribution. Use the trick of subtracting the maximum value in the i-th dimension from all elements of the i-th dimension to avoid numerical stability issues.

To test your implementation, complete [adapters.run_softmax] and make sure it passes `uv run pytest -k test_softmax_matches_pytorch`.

We can now define the Attention operation mathematically as follows:

$$\mathrm{Attention}(Q, K, V) = \mathrm{softmax}\left(\frac{Q^\top K}{\sqrt{d_k}}\right) V \tag{11}$$

where $Q \in \mathbb{R}^{n \times d_k}$, $K \in \mathbb{R}^{m \times d_k}$, and $V \in \mathbb{R}^{m \times d_v}$. Here, Q, K and V are all inputs to this operation—note that these are not the learnable parameters. If you're wondering why this isn't $QK^\top$, see 3.3.1.

Masking: It is sometimes convenient to *mask* the output of an attention operation. A mask should have the shape $M \in \{\texttt{True}, \texttt{False}\}^{n \times m}$, and each row i of this boolean matrix indicates which keys the query i should attend to. Canonically (and slightly confusingly), a value of `True` at position (i, j) indicates that the query i *does* attend to the key j, and a value of `False` indicates that the query *does not* attend to the key. In other words, "information flows" at (i, j) pairs with value `True`. For example, consider a 1×3 mask matrix with entries [[`True`, `True`, `False`]]. The single query vector attends only to the first two keys.

Computationally, it will be much more efficient to use masking than to compute attention on subsequences, and we can do this by taking the pre-softmax values $\left(\frac{Q^\top K}{\sqrt{d_k}}\right)$ and adding a $-\infty$ in any entry of the mask matrix that is False.

Problem (`scaled_dot_product_attention`): Implement scaled dot-product attention (5 points)

Deliverable: Implement the scaled dot-product attention function. Your implementation should handle keys and queries of shape `(batch_size, ..., seq_len, d_k)` and values of shape `(batch_size, ..., seq_len, d_v)`, where `...` represents any number of other batch-like dimensions (if provided). The implementation should return an output with the shape `(batch_size, ..., d_v)`. See section 3.3 for a discussion on batch-like dimensions.

Your implementation should also support an optional user-provided boolean mask of shape `(seq_len, seq_len)`. The attention probabilities of positions with a mask value of `True` should collectively sum to 1, and the attention probabilities of positions with a mask value of `False` should be zero.

To test your implementation against our provided tests, you will need to implement the test adapter at `[adapters.run_scaled_dot_product_attention]`.

`uv run pytest -k test_scaled_dot_product_attention` tests your implementation on third-order input tensors, while `uv run pytest -k test_4d_scaled_dot_product_attention` tests your implementation on fourth-order input tensors.

3.5.5 Causal Multi-Head Self-Attention

We will implement multi-head self-attention as described in section 3.2.2 of Vaswani et al. [2017]. Recall that, mathematically, the operation of applying multi-head attention is defined as follows:

$$\text{MultiHead}(Q, K, V) = \text{Concat}(\text{head}_1, \ldots, \text{head}_h) \tag{12}$$

$$\text{for } \text{head}_i = \text{Attention}(Q_i, K_i, V_i) \tag{13}$$

with Q_i, K_i, V_i being slice number $i \in \{1, \ldots, h\}$ of size d_k or d_v of the embedding dimension for Q, K, and V respectively. With Attention being the scaled dot-product attention operation defined in §3.5.4. From this we can form the multi-head *self*-attention operation:

$$\text{MultiHeadSelfAttention}(x) = W_O \text{MultiHead}(W_Q x, W_K x, W_V x) \tag{14}$$

Here, the learnable parameters are $W_Q \in \mathbb{R}^{hd_k \times d_{\text{model}}}$, $W_K \in \mathbb{R}^{hd_k \times d_{\text{model}}}$, $W_V \in \mathbb{R}^{hd_v \times d_{\text{model}}}$, and $W_O \in \mathbb{R}^{d_{\text{model}} \times hd_v}$. Since the Qs, K, and Vs are sliced in the multi-head attention operation, we can think of W_Q, W_K and W_V as being separated for each head along the output dimension. When you have this working, you should be computing the key, value, and query projections in a total of three matrix multiplies.[5]

[5] As a stretch goal, try combining the key, query, and value projections into a single weight matrix so you only need a single matrix multiply.

Causal masking. Your implementation should prevent the model from attending to future tokens in the sequence. In other words, if the model is given a token sequence $t_1, \dots, t_n$, and we want to calculate the next-word predictions for the prefix $t_1, \dots, t_i$ (where $i < n$), the model should *not* be able to access (attend to) the token representations at positions $t_{i+1}, \dots, t_n$ since it will not have access to these tokens when generating text during inference (and these future tokens leak information about the identity of the true next word, trivializing the language modeling pre-training objective). For an input token sequence $t_1, \dots, t_n$ we can naively prevent access to future tokens by running multi-head self-attention n times (for the n unique prefixes in the sequence). Instead, we'll use causal attention masking, which allows token i to attend to all positions $j \leq i$ in the sequence. You can use `torch.triu` or a broadcasted index comparison to construct this mask, and you should take advantage of the fact that your scaled dot-product attention implementation from §3.5.4 already supports attention masking.

Applying RoPE. RoPE should be applied to the query and key vectors, but not the value vectors. Also, the head dimension should be handled as a batch dimension, because in multi-head attention, attention is being applied independently for each head. This means that precisely the same RoPE rotation should be applied to the query and key vectors for each head.

Problem (`multihead_self_attention`): Implement causal multi-head self-attention (5 points)

Deliverable: Implement causal multi-head self-attention as a `torch.nn.Module`. Your implementation should accept (at least) the following parameters:

`d_model`: `int` Dimensionality of the Transformer block inputs.

`num_heads`: `int` Number of heads to use in multi-head self-attention.

Folllowing Vaswani et al. [2017], set $d_k = d_v = d_{\text{model}}/h$. To test your implementation against our provided tests, implement the test adapter at `[adapters.run_multihead_self_attention]`. Then, run `uv run pytest -k test_multihead_self_attention` to test your implementation.

3.6 The Full Transformer LM

Let's begin by assembling the Transformer block (it will be helpful to refer back to Figure 2). A Transformer block contains two 'sublayers', one for the multihead self attention, and another for the feed-forward network. In each sublayer, we first perform RMSNorm, then the main operation (MHA/FF), finally adding in the residual connection.

To be concrete, the first half (the first 'sub-layer') of the Transformer block should be implementing the following set of updates to produce an output y from an input x,

$$y = x + \text{MultiHeadSelfAttention}(\text{RMSNorm}(x)). \tag{15}$$

Problem (`transformer_block`): Implement the Transformer block (3 points)

Implement the pre-norm Transformer block as described in §3.5 and illustrated in Figure 2. Your Transformer block should accept (at least) the following parameters.

`d_model`: `int` Dimensionality of the Transformer block inputs.

`num_heads`: `int` Number of heads to use in multi-head self-attention.

`d_ff`: `int` Dimensionality of the position-wise feed-forward inner layer.

To test your implementation, implement the adapter [adapters.run_transformer_block]. Then run `uv run pytest -k test_transformer_block` to test your implementation.

Deliverable: Transformer block code that passes the provided tests.

Now we put the blocks together, following the high level diagram in Figure 1. Follow our description of the embedding in Section 3.1.1, feed this into `num_layers` Transformer blocks, and then pass that into the three output layers to obtain a distribution over the vocabulary.

Problem (`transformer_lm`): Implementing the Transformer LM (3 points)

Time to put it all together! Implement the Transformer language model as described in §3.1 and illustrated in Figure 1. At minimum, your implementation should accept all the aforementioned construction parameters for the Transformer block, as well as these additional parameters:

`vocab_size:` `int` The size of the vocabulary, necessary for determining the dimensionality of the token embedding matrix.

`context_length:` `int` The maximum context length, necessary for determining the dimensionality of the position embedding matrix.

`num_layers:` `int` The number of Transformer blocks to use.

To test your implementation against our provided tests, you will first need to implement the test adapter at [adapters.run_transformer_lm]. Then, run `uv run pytest -k test_transformer_lm` to test your implementation.

Deliverable: A Transformer LM module that passes the above tests.

Resource accounting. It is useful to be able to understand how the various parts of the Transformer consume compute and memory. We will go through the steps to do some basic "FLOPs accounting." The *vast* majority of FLOPS in a Transformer are matrix multiplies, so our core approach is simple:

1. Write down all the matrix multiplies in a Transformer forward pass.
2. Convert each matrix multiply into FLOPs required.

For this second step, the following fact will be useful:

Rule: Given $A \in \mathbb{R}^{m\times n}$ and $B \in \mathbb{R}^{n\times p}$, the matrix-matrix product AB requires $2mnp$ FLOPs.

To see this, note that $(AB)[i,j] = A[i,:] \cdot B[:,j]$, and that this dot product requires n additions and n multiplications ($2n$ FLOPs). Then, since the matrix-matrix product AB has $m \times p$ entries, the total number of FLOPS is $(2n)(mp) = 2mnp$.

Now, before you do the next problem, it can be helpful to go through each component of your Transformer block and Transformer LM, and list out all the matrix multiplies and their associated FLOPs costs.

Problem (`transformer_accounting`): Transformer LM resource accounting (5 points)

(a) Consider GPT-2 XL, which has the following configuration:

`vocab_size` : 50,257

`context_length` : 1,024

`num_layers` : 48

`d_model` : 1,600

M_emb = d_v x d_model = 50,257 x 1600
M_q = d_model x d_quer (tot) = d_model x d_model = 1600 x 1600
M_k = d_model x d_model = 1600 x 1600
M_v = d_model x d_model = 1600 x 1600
M_W1 = d_model x d_ff = 1600 x 6400
M_W2 = d_model x d_ff = 1600 x 6400
M_W3 = d_model x d_ff = 1600 x 6400

M_rot = 2x2xcontext_length/2 = 2context_length = 2048, Not Trainable!

num_heads : 25

d_ff : 6,400

Suppose we constructed our model using this configuration. How many trainable parameters would our model have? Assuming each parameter is represented using single-precision floating point, how much memory is required to just load this model?

Deliverable: A one-to-two sentence response.

(b) Identify the matrix multiplies required to complete a forward pass of our GPT-2 XL-shaped model. How many FLOPs do these matrix multiplies require in total? Assume that our input sequence has `context_length` tokens.

Deliverable: A list of matrix multiplies (with descriptions), and the total number of FLOPs required.

(c) Based on your analysis above, which parts of the model require the most FLOPs?

Deliverable: A one-to-two sentence response.

(d) Repeat your analysis with GPT-2 small (12 layers, 768 `d_model`, 12 heads), GPT-2 medium (24 layers, 1024 `d_model`, 16 heads), and GPT-2 large (36 layers, 1280 `d_model`, 20 heads). As the model size increases, which parts of the Transformer LM take up proportionally more or less of the total FLOPs?

Deliverable: For each model, provide a breakdown of model components and its associated FLOPs (as a proportion of the total FLOPs required for a forward pass). In addition, provide a one-to-two sentence description of how varying the model size changes the proportional FLOPs of each component.

(e) Take GPT-2 XL and increase the context length to 16,384. How does the total FLOPs for one forward pass change? How do the relative contribution of FLOPs of the model components change?

Deliverable: A one-to-two sentence response.

~/Dropbox/Eric/Machine Learning/cs336/cs336-a1-main-brandon-snider/cs336_basics/model.py

```
import torch
import math
from einops import einsum, rearrange, reduce


def softmax(x: torch.Tensor, dim: int) -> torch.Tensor:
    x_max = x.max(dim=dim, keepdim=True).values
    x_exp = torch.exp(x - x_max)
    return x_exp / x_exp.sum(dim=dim, keepdim=True)


def scaled_dot_product_attention(Q: torch.Tensor, K: torch.Tensor, V: torch.Tensor,
mask: torch.Tensor):
    d_k = Q.shape[-1]

    attention_scores = einsum(Q, K, "... seq_q d, ... seq_k d -> ... seq_q seq_k")
    attention_scores = attention_scores / math.sqrt(d_k)
    attention_scores = torch.where(mask, attention_scores, float("-inf"))

    attention_weights = softmax(attention_scores, dim=-1)
    output = einsum(attention_weights, V, "... seq_q seq_k, ... seq_k d -> ... seq_q
d")

    return output


class Linear(torch.nn.Module):
    def __init__(
        self, in_features: int, out_features: int, device: torch.device | None =
None, dtype: torch.dtype | None = None
    ):
        super().__init__()

        mean = 0
        std = math.sqrt(2 / (out_features + in_features))
        lower = -3 * std
        upper = 3 * std

        w = torch.empty((out_features, in_features), device=device, dtype=dtype)
        torch.nn.init.trunc_normal_(w, mean=mean, std=std, a=lower, b=upper)

        self.weight = torch.nn.Parameter(w)

    def forward(self, x: torch.Tensor) -> torch.Tensor:
        return einsum(self.weight, x, "d_out d_in, ... d_in -> ... d_out")


class RotaryPositionalEmbedding(torch.nn.Module):
```

```
    def __init__(
        self,
        theta: float,
        d_k: int,
        max_seq_len: int,
        device: torch.device | None = None,
        dtype: torch.dtype | None = None,
    ):
        super().__init__()

        positions = torch.arange(max_seq_len, device=device).unsqueeze(1)
        freqs = torch.arange(0, d_k, 2, device=device) / d_k # should this start with a 1?
        inv_freq = 1.0 / (theta**freqs)
        angles = positions * inv_freq

        self.register_buffer("cos", angles.cos().to(dtype), persistent=False)
        self.register_buffer("sin", angles.sin().to(dtype), persistent=False)

    def forward(self, x: torch.Tensor, token_positions: torch.Tensor) -> torch.Tensor:
        cos_pos = self.cos[token_positions]
        sin_pos = self.sin[token_positions]

        x_even = x[..., 0::2]
        x_odd = x[..., 1::2]

        x_rot_even = x_even * cos_pos - x_odd * sin_pos
        x_rot_odd = x_even * sin_pos + x_odd * cos_pos

        x_rot = rearrange([x_rot_even, x_rot_odd], "two ... -> ... two")
        x_out = rearrange(x_rot, "... d1 d2 -> ... (d1 d2)")

        return x_out


class CausalMultiHeadSelfAttention(torch.nn.Module):
    def __init__(self, d_model: int, num_heads: int, device=None, dtype=None, **kwargs):
        super().__init__()

        self.wqkv = Linear(d_model, 3 * d_model, device, dtype)
        self.output_proj = Linear(d_model, d_model, device, dtype)

        self.num_heads = num_heads
        self.d_model = d_model
        self.d_head = d_model // num_heads

    def forward(
        self,
        x: torch.Tensor,
```

```
        rope: RotaryPositionalEmbedding | None = None,
        token_positions: torch.Tensor | None = None,
    ) -> torch.Tensor:
        batch_size, seq_len, _ = x.shape
        qkv = self.wqkv(x)

        # Split into separate q, k, v tensors
        q, k, v = qkv.split(self.d_model, dim=2)

        # Reshape from (batch, seq_len, dim) to (batch, heads, seq_len, head_dim)
        q = rearrange(q, "b s (h d) -> b h s d", h=self.num_heads)
        k = rearrange(k, "b s (h d) -> b h s d", h=self.num_heads)
        v = rearrange(v, "b s (h d) -> b h s d", h=self.num_heads)

        if rope is not None:
            if token_positions is None:
                token_positions = torch.arange(seq_len, device=x.device)
            q = rope(q, token_positions)
            k = rope(k, token_positions)

        # Create causal mask for self-attention
        mask = ~torch.triu(torch.ones((seq_len, seq_len), device=x.device, dtype=torch.bool), diagonal=1)

        y = scaled_dot_product_attention(q, k, v, mask)
        y = rearrange(y, "b h s d -> b s (h d)")
        return self.output_proj(y)


class RMSNorm(torch.nn.Module):
    def __init__(self, d_model: int, eps: float = 1e-5, device=None, dtype=None):
        super().__init__()

        self.eps = eps
        self.weight = torch.nn.Parameter(torch.ones(d_model, device=device, dtype=dtype))

    def forward(self, x: torch.Tensor) -> torch.Tensor:
        in_dtype = x.dtype
        x = x.to(torch.float32)

        rms = torch.sqrt(reduce(x**2, "... d -> ... 1", "mean") + self.eps)
        result = x * self.weight / rms

        return result.to(in_dtype)


def silu_activation(x: torch.Tensor) -> torch.Tensor:
    return x * torch.sigmoid(x)


```

```
class SwiGLU(torch.nn.Module):
    def __init__(self, d_model: int, d_ff: int, device: torch.device | None = None,
dtype: torch.dtype | None = None):
        super().__init__()

        self.w1 = Linear(d_model, d_ff, device, dtype)
        self.w2 = Linear(d_ff, d_model, device, dtype)
        self.w3 = Linear(d_model, d_ff, device, dtype)

    def forward(self, x: torch.Tensor) -> torch.Tensor:
        a1 = self.w1(x)
        silu = silu_activation(a1)
        return self.w2(silu * self.w3(x))


class SiLU(torch.nn.Module):
    def __init__(self, d_model: int, d_ff: int, device: torch.device | None = None,
dtype: torch.dtype | None = None):
        super().__init__()

        self.w1 = Linear(d_model, d_ff, device, dtype)
        self.w2 = Linear(d_ff, d_model, device, dtype)

    def forward(self, x: torch.Tensor) -> torch.Tensor:
        a1 = self.w1(x)
        silu = silu_activation(a1)
        return self.w2(silu)


class Block(torch.nn.Module):
    def __init__(
        self,
        d_model: int,
        num_heads: int,
        d_ff: int,
        rope: RotaryPositionalEmbedding | None = None,
        device=None,
        dtype=None,
        **kwargs,
    ):
        super().__init__()

        self.rope = rope

        self.ln1 = RMSNorm(d_model, device=device, dtype=dtype)
        self.attn = CausalMultiHeadSelfAttention(d_model, num_heads, device, dtype,
**kwargs)

        self.ln2 = RMSNorm(d_model, device=device, dtype=dtype)

        ffn_type = kwargs.get("ffn_type", "swiglu")
```

```python

        if ffn_type == "silu":
            self.ffn = SiLU(d_model, d_ff, device, dtype)
        elif ffn_type == "swiglu":
            self.ffn = SwiGLU(d_model, d_ff, device, dtype)
        else:
            raise ValueError(f"Unsupported ffn_type: {ffn_type}")

    def forward(self, x: torch.Tensor):
        x = x + self.attn(self.ln1(x), self.rope)
        x = x + self.ffn(self.ln2(x))
        return x


class Embedding(torch.nn.Module):
    def __init__(
        self,
        num_embeddings: int,
        embedding_dim: int,
        device: torch.device | None = None,
        dtype: torch.dtype | None = None,
        **kwargs,
    ):
        super().__init__()

        mean = 0
        std = 1
        lower = -3
        upper = 3

        if kwargs.get("embedding_std", None) is not None:
            std = kwargs.get("embedding_std")

        w = torch.empty((num_embeddings, embedding_dim), device=device, dtype=dtype)
        torch.nn.init.trunc_normal_(w, mean=mean, std=std, a=lower, b=upper)

        self.weight = torch.nn.Parameter(w)

    def forward(self, token_ids: torch.Tensor) -> torch.Tensor:
        return self.weight[token_ids]


class Transformer(torch.nn.Module):
    def __init__(
        self,
        d_model: int,
        num_heads: int,
        d_ff: int,
        vocab_size: int,
        context_length: int,
```

```
        num_layers: int,
        rope_theta: float = 10000.0,
        device=None,
        dtype=None,
        **kwargs,
    ):
        super().__init__()

        self.context_length = context_length
        self.token_embeddings = Embedding(vocab_size, d_model, device, dtype, **kwargs)

        if d_model % num_heads != 0:
            raise ValueError("d_model must be divisible by num_heads")

        d_head = d_model // num_heads
        rope = RotaryPositionalEmbedding(rope_theta, d_head, context_length, device=device, dtype=dtype)

        self.layers = torch.nn.ModuleList(
            [Block(d_model, num_heads, d_ff, rope, device, dtype, **kwargs) for _ in range(num_layers)]
        )

        self.ln_final = RMSNorm(d_model, device=device, dtype=dtype)
        self.lm_head = Linear(d_model, vocab_size, device, dtype)

        if kwargs.get("weight_tying", False):
            self.lm_head.weight = self.token_embeddings.weight

    def forward(self, x: torch.Tensor) -> torch.Tensor:
        batch_size, seq_len = x.shape

        if seq_len > self.context_length:
            raise ValueError(f"Input sequence length ({seq_len}) exceeds model context length ({self.context_length})")

        x = self.token_embeddings(x)

        for layer in self.layers:
            x = layer(x)

        x = self.ln_final(x)
        x = self.lm_head(x)

        return x

```

4 Training a Transformer LM

We now have the steps to preprocess the data (via tokenizer) and the model (Transformer). What remains is to build all of the code to support training. This consists of the following:

- **Loss:** we need to define the loss function (cross-entropy).
- **Optimizer:** we need to define the optimizer to minimize this loss (AdamW).
- **Training loop:** we need all the supporting infrastructure that loads data, saves checkpoints, and manages training.

4.1 Cross-entropy loss

Confusing

Recall that the Transformer language model defines a distribution $p_\theta(x_{i+1} \mid x_{1:i})$ for each sequence x of length $m+1$ and $i = 1, \ldots, m$. Given a training set D consisting of sequences of length m, we define the standard cross-entropy (negative log-likelihood) loss function:

$$\ell(\theta; D) = \frac{1}{|D|m} \sum_{x \in D} \sum_{i=1}^{m} -\log p_\theta(x_{i+1} \mid x_{1:i}). \tag{16}$$

(Note that a single forward pass in the Transformer yields $p_\theta(x_{i+1} \mid x_{1:i})$ for *all* $i = 1, \ldots, m$.)

In particular, the Transformer computes logits $o_i \in \mathbb{R}^{\texttt{vocab\_size}}$ for each position i, which results in:[6]

$$p(x_{i+1} \mid x_{1:i}) = \text{softmax}(o_i)[x_{i+1}] = \frac{\exp(o_i[x_{i+1}])}{\sum_{a=1}^{\texttt{vocab\_size}} \exp(o_i[a])}. \tag{17}$$

The cross entropy loss is generally defined with respect to the vector of logits $o_i \in \mathbb{R}^{\texttt{vocab\_size}}$ and target x_{i+1}.[7]

Implementing the cross entropy loss requires some care with numerical issues, just like in the case of softmax.

Problem (`cross_entropy`): Implement Cross entropy

Deliverable: Write a function to compute the cross entropy loss, which takes in predicted logits (o_i) and targets (x_{i+1}) and computes the cross entropy $\ell_i = -\log \text{softmax}(o_i)[x_{i+1}]$. Your function should handle the following:

- Subtract the largest element for numerical stability.
- Cancel out log and exp whenever possible.
- Handle any additional batch dimensions and return the *average* across the batch. As with section 3.3, we assume batch-like dimensions always come first, before the vocabulary size dimension.

Implement [adapters.run_cross_entropy], then run `uv run pytest -k test_cross_entropy` to test your implementation.

Perplexity Cross entropy suffices for training, but when we evaluate the model, we also want to report perplexity. For a sequence of length m where we suffer cross-entropy losses $\ell_1, \ldots, \ell_m$:

$$\text{perplexity} = \exp\left(\frac{1}{m}\sum_{i=1}^{m} \ell_i\right). \tag{18}$$

[6] Note that $o_i[k]$ refers to value at index k of the vector o_i.

[7] This corresponds to the cross entropy between the Dirac delta distribution over x_{i+1} and the predicted softmax(o_i) distribution.

4.2 The SGD Optimizer

Now that we have a loss function, we will begin our exploration of optimizers. The simplest gradient-based optimizer is Stochastic Gradient Descent (SGD). We start with randomly initialized parameters θ_0. Then for each step $t = 0, \ldots, T-1$, we perform the following update:

$$\theta_{t+1} \leftarrow \theta_t - \alpha_t \nabla L(\theta_t; B_t), \tag{19}$$

where B_t is a random batch of data sampled from the dataset D, and the *learning rate* α_t and *batch size* $|B_t|$ are hyperparameters.

4.2.1 Implementing SGD in PyTorch

To implement our optimizers, we will subclass the PyTorch `torch.optim.Optimizer` class. An `Optimizer` subclass must implement two methods:

`def __init__(self, params, ...)` should initialize your optimizer. Here, `params` will be a collection of parameters to be optimized (or parameter groups, in case the user wants to use different hyperparameters, such as learning rates, for different parts of the model). Make sure to pass `params` to the `__init__` method of the base class, which will store these parameters for use in `step`. You can take additional arguments depending on the optimizer (e.g., the learning rate is a common one), and pass them to the base class constructor as a dictionary, where keys are the names (strings) you choose for these parameters.

`def step(self)` should make one update of the parameters. During the training loop, this will be called after the backward pass, so you have access to the gradients on the last batch. This method should iterate through each parameter tensor `p` and modify them *in place*, i.e. setting `p.data`, which holds the tensor associated with that parameter based on the gradient `p.grad` (if it exists), the tensor representing the gradient of the loss with respect to that parameter.

The PyTorch optimizer API has a few subtleties, so it's easier to explain it with an example. To make our example richer, we'll implement a slight variation of SGD where the learning rate decays over training, starting with an initial learning rate α and taking successively smaller steps over time:

$$\theta_{t+1} = \theta_t - \frac{\alpha}{\sqrt{t+1}} \nabla L(\theta_t; B_t) \tag{20}$$

Let's see how this version of SGD would be implemented as a PyTorch `Optimizer`:

```python
from collections.abc import Callable, Iterable
from typing import Optional
import torch
import math

class SGD(torch.optim.Optimizer):
    def __init__(self, params, lr=1e-3):
        if lr < 0:
            raise ValueError(f"Invalid learning rate: {lr}")
        defaults = {"lr": lr}
        super().__init__(params, defaults)

    def step(self, closure: Optional[Callable] = None):
        loss = None if closure is None else closure()
        for group in self.param_groups:
            lr = group["lr"]  # Get the learning rate.
```

this is inherited
from torch.optim.Optimizer

```
            for p in group["params"]:
                if p.grad is None:
                    continue

                state = self.state[p]  # Get state associated with p.
                t = state.get("t", 0)  # Get iteration number from the state, or initial value.
                grad = p.grad.data  # Get the gradient of loss with respect to p.
                p.data -= lr / math.sqrt(t + 1) * grad  # Update weight tensor in-place.
                state["t"] = t + 1  # Increment iteration number.

        return loss
```

state is a dict for each parameter p

In `__init__`, we pass the parameters to the optimizer, as well as default hyperparameters, to the base class constructor (the parameters might come in groups, each with different hyperparameters). In case the parameters are just a single collection of `torch.nn.Parameter` objects, the base constructor will create a single group and assign it the default hyperparameters. Then, in `step`, we iterate over each parameter group, then over each parameter in that group, and apply Eq 20. Here, we keep the iteration number as a state associated with each parameter: we first read this value, use it in the gradient update, and then update it. The API specifies that the user might pass in a callable `closure` to re-compute the loss before the optimizer step. We won't need this for the optimizers we'll use, but we add it to comply with the API.

To see this working, we can use the following minimal example of a *training loop*:

```
weights = torch.nn.Parameter(5 * torch.randn((10, 10)))
opt = SGD([weights], lr=1)

for t in range(100):
    opt.zero_grad()  # Reset the gradients for all learnable parameters.
    loss = (weights**2).mean() # Compute a scalar loss value.
    print(loss.cpu().item())
    loss.backward() # Run backward pass, which computes gradients.
    opt.step() # Run optimizer step.
```

This is the typical structure of a training loop: in each iteration, we will compute the loss and run a step of the optimizer. When training language models, our learnable parameters will come from the model (in PyTorch, `m.parameters()` gives us this collection). The loss will be computed over a sampled batch of data, but the basic structure of the training loop will be the same.

Problem (`learning_rate_tuning`): Tuning the learning rate (1 point)

As we will see, one of the hyperparameters that affects training the most is the learning rate. Let's see that in practice in our toy example. Run the SGD example above with three other values for the learning rate: 1e1, 1e2, and 1e3, for just 10 training iterations. What happens with the loss for each of these learning rates? Does it decay faster, slower, or does it diverge (i.e., increase over the course of training)?

Deliverable: A one-two sentence response with the behaviors you observed.

4.3 AdamW

Modern language models are typically trained with more sophisticated optimizers, instead of SGD. Most optimizers used recently are derivatives of the Adam optimizer [Kingma and Ba, 2015]. We will use AdamW [Loshchilov and Hutter, 2019], which is in wide use in recent work. AdamW proposes a modification to Adam that improves regularization by adding *weight decay* (at each iteration, we pull the parameters towards 0), in a way that is decoupled from the gradient update. We will implement AdamW as described in algorithm 2 of Loshchilov and Hutter [2019].

AdamW is *stateful*: for each parameter, it keeps track of a running estimate of its first and second moments. Thus, AdamW uses additional memory in exchange for improved stability and convergence. Besides the learning rate α, AdamW has a pair of hyperparameters (β_1, β_2) that control the updates to the moment estimates, and a weight decay rate λ. Typical applications set (β_1, β_2) to $(0.9, 0.999)$, but large language models like LLaMA [Touvron et al., 2023] and GPT-3 [Brown et al., 2020] are often trained with $(0.9, 0.95)$. The algorithm can be written as follows, where ϵ is a small value (e.g., 10^{-8}) used to improve numerical stability in case we get extremely small values in v:

Algorithm 1 AdamW Optimizer

init(θ) (Initialize learnable parameters)
$m \leftarrow 0$ (Initial value of the first moment vector; same shape as θ)
$v \leftarrow 0$ (Initial value of the second moment vector; same shape as θ)
for $t = 1, \ldots, T$ **do**
  Sample batch of data B_t
  $g \leftarrow \nabla_\theta \ell(\theta; B_t)$ (Compute the gradient of the loss at the current time step)
  $m \leftarrow \beta_1 m + (1 - \beta_1) g$ (Update the first moment estimate)
  $v \leftarrow \beta_2 v + (1 - \beta_2) g^2$ (Update the second moment estimate)
  $\alpha_t \leftarrow \alpha \frac{\sqrt{1-(\beta_2)^t}}{1-(\beta_1)^t}$ (Compute adjusted α for iteration t)
  $\theta \leftarrow \theta - \alpha_t \frac{m}{\sqrt{v}+\epsilon}$ (Update the parameters)
  $\theta \leftarrow \theta - \alpha\lambda\theta$ (Apply weight decay)
end for

Note that t starts at 1. You will now implement this optimizer.

Problem (`adamw`): Implement AdamW (2 points)

Deliverable: Implement the AdamW optimizer as a subclass of `torch.optim.Optimizer`. Your class should take the learning rate α in `__init__`, as well as the β, ϵ and λ hyperparameters. To help you keep state, the base `Optimizer` class gives you a dictionary `self.state`, which maps `nn.Parameter` objects to a dictionary that stores any information you need for that parameter (for AdamW, this would be the moment estimates). Implement `[adapters.get_adamw_cls]` and make sure it passes `uv run pytest -k test_adamw`.

Problem (`adamwAccounting`): Resource accounting for training with AdamW (2 points)

Let us compute how much memory and compute running AdamW requires. Assume we are using float32 for every tensor.

(a) How much peak memory does running AdamW require? Decompose your answer based on the memory usage of the parameters, activations, gradients, and optimizer state. Express your answer in terms of the `batch_size` and the model hyperparameters (`vocab_size`, `context_length`, `num_layers`, `d_model`, `num_heads`). Assume `d_ff` $= 4 \times$ `d_model`.

For simplicity, when calculating memory usage of activations, consider only the following components:

- Transformer block
  - RMSNorm(s)

  - Multi-head self-attention sublayer: QKV projections, $Q^\top K$ matrix multiply, softmax, weighted sum of values, output projection.
  - Position-wise feed-forward: W_1 matrix multiply, SiLU, W_2 matrix multiply
- final RMSNorm
- output embedding
- cross-entropy on logits

Deliverable: An algebraic expression for each of parameters, activations, gradients, and optimizer state, as well as the total.

(b) Instantiate your answer for a GPT-2 XL-shaped model to get an expression that only depends on the `batch_size`. What is the maximum batch size you can use and still fit within 80GB memory?

Deliverable: An expression that looks like $a \cdot \texttt{batch\_size} + b$ for numerical values a, b, and a number representing the maximum batch size.

(c) How many FLOPs does running one step of AdamW take?

Deliverable: An algebraic expression, with a brief justification.

(d) Model FLOPs utilization (MFU) is defined as the ratio of observed throughput (tokens per second) relative to the hardware's theoretical peak FLOP throughput [Chowdhery et al., 2022]. An NVIDIA A100 GPU has a theoretical peak of 19.5 teraFLOP/s for float32 operations. Assuming you are able to get 50% MFU, how long would it take to train a GPT-2 XL for 400K steps and a batch size of 1024 on a single A100? Following Kaplan et al. [2020] and Hoffmann et al. [2022], assume that the backward pass has twice the FLOPs of the forward pass.

Deliverable: The number of days training would take, with a brief justification.

4.4 Learning rate scheduling

The value for the learning rate that leads to the quickest decrease in loss often varies during training. In training Transformers, it is typical to use a learning rate *schedule*, where we start with a bigger learning rate, making quicker updates in the beginning, and slowly decay it to a smaller value as the model trains[8]. In this assignment, we will implement the cosine annealing schedule used to train LLaMA [Touvron et al., 2023].

A scheduler is simply a function that takes the current step t and other relevant parameters (such as the initial and final learning rates), and returns the learning rate to use for the gradient update at step t. The simplest schedule is the constant function, which will return the same learning rate given any t.

The cosine annealing learning rate schedule takes (i) the current iteration t, (ii) the maximum learning rate $\alpha_{\max}$, (iii) the minimum (final) learning rate $\alpha_{\min}$, (iv) the number of *warm-up* iterations T_w, and (v) the number of cosine annealing iterations T_c. The learning rate at iteration t is defined as:

(Warm-up) If $t < T_w$, then $\alpha_t = \frac{t}{T_w}\alpha_{\max}$.

(Cosine annealing) If $T_w \le t \le T_c$, then $\alpha_t = \alpha_{\min} + \frac{1}{2}\left(1 + \cos\left(\frac{t - T_w}{T_c - T_w}\pi\right)\right)(\alpha_{\max} - \alpha_{\min})$.

(Post-annealing) If $t > T_c$, then $\alpha_t = \alpha_{\min}$.

[8] It's sometimes common to use a schedule where the learning rate rises back up (restarts) to help get past local minima.

Problem (learning_rate_schedule): Implement cosine learning rate schedule with warmup

Write a function that takes t, $\alpha_{\max}$, $\alpha_{\min}$, T_w and T_c, and returns the learning rate α_t according to the scheduler defined above. Then implement [adapters.get_lr_cosine_schedule] and make sure it passes `uv run pytest -k test_get_lr_cosine_schedule`.

4.5 Gradient clipping

During training, we can sometimes hit training examples that yield large gradients, which can destabilize training. To mitigate this, one technique often employed in practice is *gradient clipping*. The idea is to enforce a limit on the norm of the gradient after each backward pass before taking an optimizer step.

Given the gradient (for all parameters) g, we compute its ℓ_2-norm $\|g\|_2$. If this norm is less than a maximum value M, then we leave g as is; otherwise, we scale g down by a factor of $\frac{M}{\|g\|_2+\epsilon}$ (where a small ϵ, like 10^{-6}, is added for numeric stability). Note that the resulting norm will be just under M.

Problem (gradient_clipping): Implement gradient clipping (1 point)

Write a function that implements gradient clipping. Your function should take a list of parameters and a maximum ℓ_2-norm. It should modify each parameter gradient in place. Use $\epsilon = 10^{-6}$ (the PyTorch default). Then, implement the adapter [adapters.run_gradient_clipping] and make sure it passes `uv run pytest -k test_gradient_clipping`.

~/Dropbox/Eric/Machine Learning/cs336/cs336-a1-main-brandon-snider/cs336_basics/loss.py

```python
import torch


def cross_entropy_loss_naive(logits: torch.Tensor, targets: torch.Tensor) -> torch.Tensor:
    """
    Computes the cross entropy loss between logits and target indices.

    Args:
        logits (torch.Tensor): Logits with shape (..., vocab_size)
        targets (torch.Tensor): Target indices with shape (...)

    Returns:
        torch.Tensor: Average cross entropy loss across the batch
    """
    max_logits = logits.max(dim=-1, keepdim=True).values
    logits_shifted = logits - max_logits
    sum_exp = torch.exp(logits_shifted).sum(dim=-1, keepdim=True)
    log_sum_exp = torch.log(sum_exp)
    log_probs = logits_shifted - log_sum_exp

    target_log_probs = log_probs.gather(dim=-1, index=targets.unsqueeze(-1)).squeeze(-1)

    return -target_log_probs.mean()


@torch.compile
def cross_entropy_loss(logits: torch.Tensor, targets: torch.Tensor) -> torch.Tensor:
    max_vals, _ = logits.max(dim=-1, keepdim=True)
    target_logits = logits.gather(dim=-1, index=targets.unsqueeze(-1))
    shifted = logits - max_vals
    sum_exp = shifted.exp().sum(dim=-1, keepdim=True)
    log_sum_exp = sum_exp.log()
    return -((target_logits - max_vals) - log_sum_exp).mean()

```

~/Dropbox/Eric/Machine Learning/cs336/cs336-a1-main-brandon-snider/cs336_basics/adamw.py

```python
import torch
import math


class AdamW(torch.optim.Optimizer):
    def __init__(
        self,
        params,
        lr: float = 1e-3,
        betas: tuple[float, float] = (0.9, 0.95),
        eps: float = 1e-8,
        weight_decay: float = 0.1,
        **kwargs,
    ):
        defaults = {"lr": lr, "betas": betas, "eps": eps, "weight_decay": weight_decay}
        super().__init__(params, defaults)

        self.lr = lr
        self.betas = betas
        self.eps = eps
        self.weight_decay = weight_decay

    @torch.no_grad()
    def step(self, closure=None):
        loss = None if closure is None else closure()

        for group in self.param_groups:
            b1, b2 = group["betas"]
            lr = group["lr"]
            eps = group["eps"]
            wd = group["weight_decay"]

            for p in group["params"]:
                if p.grad is None:
                    continue

                state = self.state[p]

                m = state.get("m", torch.zeros_like(p.data))
                v = state.get("v", torch.zeros_like(p.data))
                t = state.get("t", 1)

                grad = p.grad

                state["m"] = b1 * m + (1 - b1) * grad
                state["v"] = b2 * v + (1 - b2) * grad.pow(2)

```

```
                step_size = lr * (math.sqrt(1 - b2**t) / (1 - b1**t))

                # must use addcdiv to update in place! otherwise extra memory is allotted
                p.data.addcdiv_(state["m"], torch.sqrt(state["v"]) + eps, value=-step_size)

                if wd != 0:
                    p.data.add_(p.data, alpha=-lr * wd)

                state["t"] = t + 1

        return loss

```

~/Dropbox/Eric/Machine Learning/cs336/cs336-a1-main-brandon-snider/cs336_basics/lr_schedule.py

```python
import math

from torch.optim import Optimizer


class LRSchedule:
    def __init__(
        self,
        lr_max: float,
        warmup_iters: int,
        schedule: list[dict],
        optimizer: Optimizer | None = None,
        param_groups: list[dict] | None = None,
    ):
        """
        A steppable learning rate scheduler supporting arbitrary multi-phase decay schedules.

        Args:
            lr_max: The maximum learning rate.
            warmup_iters: The number of iterations for linear warmup from zero to lr_max
            schedule: A list of dictionaries, each containing the following keys:
                - until_iter: The iteration at which the phase should end
                - to_lr: The learning rate at which the phase should end
                - type: The type of decay to use for the phase (linear, cosine, exp)
            optimizer (optional): The optimizer whose param groups will be updated
            param_groups (optional): The param groups to update the learning rate for
                - If provided, will use these param groups instead of optimizer.param_groups
        """
        self.lr_max = lr_max
        self.warmup_iters = warmup_iters
        self.schedule = schedule
        self.optimizer = optimizer
        self.param_groups = param_groups
        self.reset()

    def step(self):
        self.it += 1
        self.lr = lr_schedule(self.it, self.lr_max, self.warmup_iters, self.schedule)

        param_groups = self.param_groups if self.param_groups is not None else self.optimizer.param_groups

        for param_group in param_groups:
            param_group["lr"] = self.lr

```

```
    def reset(self):
        self.it = -1
        self.step()

    def __repr__(self):
        return f"LRSchedule(lr_max={self.lr_max}, warmup_iters={self.warmup_iters}, schedule={self.schedule}, optimizer={self.optimizer}, param_groups={self.param_groups})"


def lr_schedule(
    it: int,
    lr_max: float,
    warmup_iters: int,
    schedule: list[dict],
):
    """A learning rate scheduler supporting arbitrary multi-phase decay schedules.

    Args:
        it: The current iteration.
        lr_max: The maximum learning rate.
        warmup_iters: The number of iterations for linear warmup from zero to lr_max
        schedule: A list of dictionaries, each containing the following keys:
            - until_iter: The iteration at which the phase should end
            - to_lr: The learning rate at which the phase should end
            - type: The type of decay to use for the phase (linear, cosine, exp)
    """
    if it < warmup_iters:
        return (it / warmup_iters) * lr_max

    phase_max_lr = lr_max
    phase_start_iter = warmup_iters

    for phase in schedule:
        phase_min_lr = phase["to_lr"]

        if it <= phase["until_iter"]:
            decay_step = it - phase_start_iter
            decay_steps = phase["until_iter"] - phase_start_iter

            if phase["type"] == "linear":
                return phase_max_lr - (decay_step / decay_steps) * (phase_max_lr - phase_min_lr)
            elif phase["type"] == "cosine":
                cos = math.cos((decay_step / decay_steps) * math.pi)
                return phase_min_lr + 1 / 2 * (1 + cos) * (phase_max_lr - phase_min_lr)
            elif phase["type"] == "exp":
                return phase_max_lr * (phase_min_lr / phase_max_lr) ** (decay_step / decay_steps)

```

```python
        phase_start_iter = phase["until_iter"]
        phase_max_lr = phase["to_lr"]

    return schedule[-1]["to_lr"]


def lr_linear_schedule(it: int, lr_max: float, lr_min: float, warmup_iters: int,
linear_cycle_iters: int):
    if it < warmup_iters:
        return (it / warmup_iters) * lr_max

    if it <= linear_cycle_iters:
        decay_step = it - warmup_iters
        decay_steps = linear_cycle_iters - warmup_iters
        return lr_max - (decay_step / decay_steps) * (lr_max - lr_min)

    return lr_min


def lr_cosine_schedule(it: int, lr_max: float, lr_min: float, warmup_iters: int,
cosine_cycle_iters: int):
    if it < warmup_iters:
        return (it / warmup_iters) * lr_max

    if it <= cosine_cycle_iters:
        decay_step = it - warmup_iters
        decay_steps = cosine_cycle_iters - warmup_iters
        cos = math.cos((decay_step / decay_steps) * math.pi)
        return lr_min + 1 / 2 * (1 + cos) * (lr_max - lr_min)

    return lr_min


def lr_double_schedule(
    it: int,
    lr_max: float,
    lr_inter: int,
    lr_min: float,
    warmup_iters: int,
    phase_one_iters: int,
    phase_two_iters: int,
    phase_two_type: str,
):
    """
    A double-decay learning rate schedule.

    Args:
        it: The current iteration.
        lr_max: Max. LR, to which we warm up linearly.
        lr_inter: LR to which we decay exponentially from lr_max.
        lr_min: Min. LR, to which we decay from lr_inter, linearly or cosine.
```

```
        warmup_iters: The number of iters for linear warmup from zero to lr_max
        exp_decay_iters: The iter at which the exponential decay phase should end
        phase_two_iters: The iter at which the second decay phase (linear or cosine) should end
        phase_two_type: The type of decay to use for the second phase (linear or cosine)

    Note:
        - exp_decay_iters is NOT the number of iterations for the exponential decay phase.
          It is the iter at which the exponential decay should end.
        - phase_two_iters is NOT the number of iterations for the second decay phase.
          It is the iter at which the second decay should end.
    Example:
        - Want: warmup for 1000 iters, exp decay for 1000 iters, linear decay for 1000 iters
        - Set:
            warmup_iters = 1000
            exp_decay_iters = 2000
            phase_two_iters = 3000
            phase_two_type = "linear"
    """
    if it < warmup_iters:
        # We're in the warmup phase
        return (it / warmup_iters) * lr_max

    if it <= phase_one_iters:
        # We're in the exponential decay phase
        decay_step = it - warmup_iters
        decay_steps = phase_one_iters - warmup_iters
        return lr_max * (lr_inter / lr_max) ** (decay_step / decay_steps)

    # We're in phase two (linear or cosine decay from lr_inter to lr_min)
    it2 = it - phase_one_iters
    phase_two_decay_steps = phase_two_iters - phase_one_iters

    if phase_two_type == "linear":
        # The second decay phase of the schedule is linear
        return lr_linear_schedule(
            it2, lr_max=lr_inter, lr_min=lr_min, warmup_iters=0, linear_cycle_iters=phase_two_decay_steps
        )

    if phase_two_type == "cosine":
        # The second decay phase of the schedule is cosine
        return lr_cosine_schedule(
            it2, lr_max=lr_inter, lr_min=lr_min, warmup_iters=0, cosine_cycle_iters=phase_two_decay_steps
        )

    return lr_min
```

```python


def seq_len_schedule(
    it: int,
    seq_len_min: int,
    schedule: list[dict],
):
    """A sequence length scheduler supporting arbitrary multi-phase decay schedules.

    Args:
        it: The current iteration.
        seq_len_min: The minimum sequence length.
        schedule: A list of dictionaries, each containing the following keys:
            - until_iter: The iteration at which the phase should end
            - to_seq_len: The sequence length at which the phase should end
    """
    phase_min_seq_len = seq_len_min
    phase_start_iter = 0

    for phase in schedule:
        phase_max_seq_len = phase["to_seq_len"]

        if it <= phase["until_iter"]:
            growth_step = it - phase_start_iter
            growth_steps = phase["until_iter"] - phase_start_iter

            return int(phase_min_seq_len + (growth_step / growth_steps) * (phase_max_seq_len - phase_min_seq_len))

        phase_start_iter = phase["until_iter"]
        phase_min_seq_len = phase["to_seq_len"]

    return schedule[-1]["to_seq_len"]


def batch_size_schedule(it: int, batch_size_max: int, schedule: list[dict]):
    """A batch size scheduler supporting arbitrary multi-phase decay schedules.

    Args:
        it: The current iteration.
        batch_size_max: The maximum batch size.
        schedule: A list of dictionaries, each containing the following keys:
            - until_iter: The iteration at which the phase should end
            - to_batch_size: The batch size at which the phase should end
    """
    phase_max_batch_size = batch_size_max
    phase_start_iter = 0

    for phase in schedule:
        phase_min_batch_size = phase["to_batch_size"]
```

```python

        if it <= phase["until_iter"]:
            decay_step = it - phase_start_iter
            decay_steps = phase["until_iter"] - phase_start_iter

            return int(
                phase_max_batch_size - (decay_step / decay_steps) *
(phase_max_batch_size - phase_min_batch_size)
            )

        phase_start_iter = phase["until_iter"]
        phase_max_batch_size = phase["to_batch_size"]

    return schedule[-1]["to_batch_size"]

```

~/Dropbox/Eric/Machine Learning/cs336/cs336-a1-main-brandon-snider/cs336_basics/gradient_clip.py

```python
from collections.abc import Iterable
import torch


@torch.compile
def gradient_clip(parameters: Iterable[torch.nn.Parameter], max_l2_norm: float):
    grads = [p.grad for p in parameters if p.grad is not None]

    if not grads:
        return torch.tensor(0.0)

    stacked_grads = torch.stack([torch.norm(g.detach(), p=2) for g in grads])
    total_norm = torch.norm(stacked_grads, p=2)

    if total_norm > max_l2_norm:
        scale = max_l2_norm / (total_norm + 1e-6)
        for grad in grads:
            grad.detach().mul_(scale)

    return total_norm

```

5 Training loop

We will now finally put together the major components we've built so far: the tokenized data, the model, and the optimizer.

5.1 Data Loader

The tokenized data (e.g., that you prepared in `tokenizer_experiments`) is a single sequence of tokens $x = (x_1, \ldots, x_n)$. Even though the source data might consist of separate documents (e.g., different web pages, or source code files), a common practice is to concatenate all of those into a single sequence of tokens, adding a delimiter between them (such as the `<|endoftext|>` token).

A *data loader* turns this into a stream of *batches*, where each batch consists of B sequences of length m, paired with the corresponding next tokens, also with length m. For example, for $B = 1, m = 3$, $([x_2, x_3, x_4], [x_3, x_4, x_5])$ would be one potential batch.

Loading data in this way simplifies training for a number of reasons. First, any $1 \leq i < n - m$ gives a valid training sequence, so sampling sequences are trivial. Since all training sequences have the same length, there's no need to pad input sequences, which improves hardware utilization (also by increasing batch size B). Finally, we also don't need to fully load the full dataset to sample training data, making it easy to handle large datasets that might not otherwise fit in memory.

Problem (`data_loading`): Implement data loading (2 points)

Deliverable: Write a function that takes a numpy array x (integer array with token IDs), a `batch_size`, a `context_length` and a PyTorch device string (e.g., `'cpu'` or `'cuda:0'`), and returns a pair of tensors: the sampled input sequences and the corresponding next-token targets. Both tensors should have shape (`batch_size`, `context_length`) containing token IDs, and both should be placed on the requested device. To test your implementation against our provided tests, you will first need to implement the test adapter at `[adapters.run_get_batch]`. Then, run `uv run pytest -k test_get_batch` to test your implementation.

Low-Resource/Downscaling Tip: Data loading on CPU or Apple Silicon

If you are planning to train your LM on CPU or Apple Silicon, you need to move your data to the correct device (and similarly, you should use the same device for your model later on).

If you are on CPU, you can use the `'cpu'` device string, and on Apple Silicon (M* chips), you can use the `'mps'` device string.

For more on MPS, checkout these resources:

- https://developer.apple.com/metal/pytorch/
- https://pytorch.org/docs/main/notes/mps.html

What if the dataset is too big to load into memory? We can use a Unix systemcall named `mmap` which maps a file on disk to virtual memory, and lazily loads the file contents when that memory location is accessed. Thus, you can "pretend" you have the entire dataset in memory. Numpy implements this through `np.memmap` (or the flag `mmap_mode='r'` to `np.load`, if you originally saved the array with `np.save`), which will return a numpy array-like object that loads the entries on-demand as you access them. **When sampling from your dataset (i.e., a numpy array) during training, be sure load the dataset in memory-mapped mode** (via `np.memmap` or the flag `mmap_mode='r'` to `np.load`, depending on how you saved the array). Make sure you also specify a `dtype` that matches the array that you're loading. It may be helpful to explicitly verify that the memory-mapped data looks correct (e.g., doesn't contain values beyond the expected vocabulary size).

~/Dropbox/Eric/Machine Learning/cs336/cs336-a1-main-brandon-snider/cs336_basics/data_loader.py

```python
import numpy as np
import torch


def get_batch(x: np.ndarray, batch_size: int, context_length: int, device: str):
    """
    Takes a numpy array of token IDs, batches them, and returns a pair of tensors:
    - (batch_size, context_length) - the actual batch
    - (batch_size, context_length) - the next token ID for each sample in the batch

    Args:
      x: np.ndarray - integer array of training data
      batch_size: int - number of samples per batch
      context_length: int - length of each sequence in batch
      device: str - device to load the data on
    """
    # Maximum valid starting index to ensure we have enough tokens for a full sequence plus one
    max_start_idx = len(x) - context_length - 1

    if max_start_idx < 0:
        raise ValueError(f"Input array length {len(x)} is too short for context_length {context_length}")

    # Sample batch_size random starting positions all at once
    start_indices = np.random.randint(0, max_start_idx + 1, size=batch_size)

    # Prepare arrays to hold our sequences
    x_sequences = np.zeros((batch_size, context_length), dtype=np.int64)
    y_sequences = np.zeros((batch_size, context_length), dtype=np.int64)

    # Fill the arrays with the appropriate sequences
    for i, start_idx in enumerate(start_indices):
        x_sequences[i] = x[start_idx : start_idx + context_length]
        y_sequences[i] = x[start_idx + 1 : start_idx + context_length + 1]

    x_batch = torch.from_numpy(x_sequences)
    y_batch = torch.from_numpy(y_sequences)

    if device.startswith("cuda"):
        # Pin memory if on GPU
        x_batch, y_batch = (
            x_batch.pin_memory().to(device, non_blocking=True),
            y_batch.pin_memory().to(device, non_blocking=True),
        )
    else:
        x_batch, y_batch = x_batch.to(device), y_batch.to(device)
```

```
    return x_batch, y_batch
```

5.2 Checkpointing

In addition to loading data, we will also need to save models as we train. When running jobs, we often want to be able to resume a training run that for some reason stopped midway (e.g., due to your job timing out, machine failure, etc). Even when all goes well, we might also want to later have access to intermediate models (e.g., to study training dynamics post-hoc, take samples from models at different stages of training, etc).

A checkpoint should have all the states that we need to resume training. We of course want to be able to restore model weights at a minimum. If using a stateful optimizer (such as AdamW), we will also need to save the optimizer's state (e.g., in the case of AdamW, the moment estimates). Finally, to resume the learning rate schedule, we will need to know the iteration number we stopped at. PyTorch makes it easy to save all of these: every `nn.Module` has a `state_dict()` method that returns a dictionary with all learnable weights; we can restore these weights later with the sister method `load_state_dict()`. The same goes for any `nn.optim.Optimizer`. Finally, `torch.save(obj, dest)` can dump an object (e.g., a dictionary containing tensors in some values, but also regular Python objects like integers) to a file (path) or file-like object, which can then be loaded back into memory with `torch.load(src)`.

Problem (`checkpointing`): Implement model checkpointing (1 point)

Implement the following two functions to load and save checkpoints:

`def save_checkpoint(model, optimizer, iteration, out)` should dump all the state from the first three parameters into the file-like object `out`. You can use the `state_dict` method of both the model and the optimizer to get their relevant states and use `torch.save(obj, out)` to dump `obj` into out (PyTorch supports either a path or a file-like object here). A typical choice is to have `obj` be a dictionary, but you can use whatever format you want as long as you can load your checkpoint later.

This function expects the following parameters:

`model: torch.nn.Module`

`optimizer: torch.optim.Optimizer`

`iteration: int`

`out: str | os.PathLike | typing.BinaryIO | typing.IO[bytes]`

`def load_checkpoint(src, model, optimizer)` should load a checkpoint from `src` (path or file-like object), and then recover the model and optimizer states from that checkpoint. Your function should return the iteration number that was saved to the checkpoint. You can use `torch.load(src)` to recover what you saved in your `save_checkpoint` implementation, and the `load_state_dict` method in both the model and optimizers to return them to their previous states.

This function expects the following parameters:

`src: str | os.PathLike | typing.BinaryIO | typing.IO[bytes]`

`model: torch.nn.Module`

`optimizer: torch.optim.Optimizer`

Implement the `[adapters.run_save_checkpoint]` and `[adapters.run_load_checkpoint]` adapters, and make sure they pass `uv run pytest -k test_checkpointing`.

5.3 Training loop

Now, it's finally time to put all of the components you implemented together into your main training script. It will pay off to make it easy to start training runs with different hyperparameters (e.g., by taking them as command-line arguments), since you will be doing these many times later to study how different choices impact training.

Problem (`training_together`): Put it together (4 points)

Deliverable: Write a script that runs a training loop to train your model on user-provided input. In particular, we recommend that your training script allow for (at least) the following:

- Ability to configure and control the various model and optimizer hyperparameters.
- Memory-efficient loading of training and validation large datasets with `np.memmap`.
- Serializing checkpoints to a user-provided path.
- Periodically logging training and validation performance (e.g., to console and/or an external service like Weights and Biases).[a]

[a] wandb.ai

~/Dropbox/Eric/Machine Learning/cs336/cs336-a1-main-brandon-snider/cs336_basics/checkpointing.py

```python
from typing import IO, BinaryIO
import torch
import os


def save_checkpoint(
    model: torch.nn.Module,
    optimizers: list[torch.optim.Optimizer] | torch.optim.Optimizer,
    iteration: int,
    out: str | os.PathLike | BinaryIO | IO[bytes],
):
    # Extract the original model from a compiled module if present
    orig_model = model._orig_mod if hasattr(model, "_orig_mod") else model

    if isinstance(optimizers, torch.optim.Optimizer):
        optimizers = [optimizers]

    torch.save(
        {
            "model": orig_model.state_dict(),
            "optimizer": [optimizer.state_dict() for optimizer in optimizers],
            "iteration": iteration,
        },
        out,
    )


def load_checkpoint(
    src: str | os.PathLike | BinaryIO | IO[bytes],
    model: torch.nn.Module | None = None,
    optimizers: list[torch.optim.Optimizer] | torch.optim.Optimizer | None = None,
):
    checkpoint = torch.load(src)

    if model is not None:
        model.load_state_dict(checkpoint["model"])

    if optimizers is not None:
        if isinstance(optimizers, torch.optim.Optimizer):
            optimizers = [optimizers]

        for optimizer, state_dict in zip(optimizers, checkpoint["optimizer"]):
            optimizer.load_state_dict(state_dict)

    return checkpoint["iteration"]
```

~/Dropbox/Eric/Machine Learning/cs336/cs336-a1-main-brandon-snider/cs336_basics/train.py

```python
import torch
import os
import time
import numpy as np
import argparse
import json
import wandb.wandb_run
import yaml
import math
import wandb

from cs336_basics.adamw import AdamW
from cs336_basics.checkpointing import save_checkpoint, load_checkpoint
from cs336_basics.lr_schedule import lr_linear_schedule
from cs336_basics.model import Transformer
from cs336_basics.data_loader import get_batch
from cs336_basics.loss import cross_entropy_loss
from cs336_basics.gradient_clip import gradient_clip
from cs336_basics.lr_schedule import lr_cosine_schedule, lr_double_schedule


class Logger:
    """Logger that handles console, file, and wandb logging"""

    def __init__(self, log_file: str | None = None, wandb_run: wandb.wandb_run.Run | None = None, resume: bool = False):
        self.log_file = log_file
        self.wandb_run = wandb_run

        # Initialize log file
        if self.log_file:
            os.makedirs(os.path.dirname(self.log_file), exist_ok=True)
            # Only clear the file if not resuming from checkpoint
            if not resume:
                with open(self.log_file, "w") as _:  # Clear the file
                    pass

    def log_info(self, message: str | dict, console=True):
        """Log a message to console and/or file"""
        if isinstance(message, dict):
            message = self.format_metrics(message)

        if console:
            print(message)

        if self.log_file:
            with open(self.log_file, "a") as f:
                f.write(message + "\n")
```

```python

    def log_metrics(self, metrics: dict):
        """Log metrics to wandb"""
        if self.wandb_run:
            self.wandb_run.log(metrics)

    def format_metrics(self, metrics_dict: dict) -> str:
        """Format metrics dictionary into a readable string"""
        return " | ".join(f"{key}: {value}" for key, value in metrics_dict.items())


class Config(dict):
    """Config object that allows attribute-style access to dictionary keys

        e.g. config = Config({"training": {"lr": 0.001,"batch_size": 64}})

        can be called as config["training"]["lr"] or config.training.lr

    """

    def __init__(self, *args, **kwargs):
        super().__init__(*args, **kwargs)
        self.__dict__ = self

        # Convert nested dictionaries to Config objects
        for key, value in self.items():
            if isinstance(value, dict):
                self[key] = Config(value)


def load_config_from_file(config_path: str) -> dict:
    """Load configuration from a file and return as a dictionary"""
    ext = os.path.splitext(config_path)[1].lower()
    with open(config_path) as f:
        if ext == ".json":
            return json.load(f)
        elif ext in [".yaml", ".yml"]: # yaml is a nested dictionar
            return yaml.safe_load(f)
        else:
            raise ValueError(f"Unsupported config file format: {ext}")


def load_config(config_path: str | None = None, base_config: dict | None = None) -> Config:
    """Load configuration from a file and detect runtime device"""
    # Start with base_config if provided (e.g. when resuming), otherwise load default
    if base_config is None:
        default_config_path = os.path.join(os.path.dirname(__file__), "./configs/default.yml")
        config = load_config_from_file(default_config_path)
    else:
```

```python
        config = base_config

    # If user specified a config, override defaults
    if config_path:
        user_config = load_config_from_file(config_path)

        # Deep update the config
        for section, section_config in user_config.items():
            if section in config:
                config[section].update(section_config)
            else:
                config[section] = section_config

    config["run"]["run_id"] = config["run"]["run_id"].replace("<timestamp>", f"{int(time.time())}")

    # Detect device and dtype at runtime
    device = "cpu"
    if torch.cuda.is_available():
        if config["training"].get("device", None) is not None:
            device = config["training"]["device"]
        else:
            device = "cuda"
    elif hasattr(torch.backends, "mps") and torch.backends.mps.is_available():
        device = "mps"

    print(f"Using device: {device}")

    dtype = torch.bfloat16 if torch.cuda.is_available() else torch.float32

    config["device"] = device
    config["dtype"] = str(dtype)  # Convert dtype to string for JSON serialization

    # Convert nested dictionary to Config object
    return Config(config)


def train(config: Config | None = None):
    """Train a transformer model with the given configuration"""
    if config is None:
        config = load_config()

    # Check if we're resuming from a checkpoint
    resuming = config.training.get("resume", False)
    start_step = 1

    run_dir = os.path.join(config.run.out_dir, config.run.run_id)
    config_outfile = os.path.join(run_dir, "config.json")
    log_file = os.path.join(run_dir, "log.txt")
    checkpoint_dir = os.path.join(run_dir, "checkpoints")
```

```python

    os.makedirs(run_dir, exist_ok=True)
    os.makedirs(checkpoint_dir, exist_ok=True)

    # Create symlink pointing `latest` to run_dir (remove `latest` if it exists)
    latest_symlink = os.path.join(config.run.out_dir, "latest")
    if os.path.islink(latest_symlink) or os.path.exists(latest_symlink):
        os.remove(latest_symlink)
    os.symlink(os.path.abspath(run_dir), latest_symlink, target_is_directory=True)

    # Initialize wandb and logger
    wandb_run = (
        None
        if not config.run.wandb_project
        else wandb.init(
            project=config.run.wandb_project,
            id=config.run.run_id,  # Use run_id as the wandb id
            resume="must" if resuming else None,  # Set resume conditionally
            name=config.run.run_id,
            config=config,
            dir=run_dir,
            tags=config.run.wandb_tags,
        )
    )

    logger = Logger(log_file=log_file, wandb_run=wandb_run, resume=resuming)

    # Save configuration (only if not resuming)
    if resuming:
        logger.log_info(f"Resuming training from existing config: {config_outfile}")
    else:
        with open(config_outfile, "w") as f:
            json.dump(config, f, indent=2, default=lambda o: list(o) if isinstance(o, tuple) else o.__dict__)
        logger.log_info(f"Saved config to: {config_outfile}")

    device = config.device
    dtype = getattr(torch, config.dtype.split(".")[-1])  # Convert string back to torch dtype

    train_data = np.memmap(config.data.train_data_path, dtype=np.uint16, mode="r")
    valid_data = np.memmap(config.data.valid_data_path, dtype=np.uint16, mode="r")

    # Initialize model
    model = Transformer(**config.model, device=device, dtype=dtype)
    model.to(device)

    print(f"Trainable params: {sum(p.numel() for p in model.parameters() if p.requires_grad)}")

    # Only decay 2D parameters (i.e. not layernorms)
```

```
    param_dict = {pn: p for pn, p in model.named_parameters() if p.requires_grad}
    decay_params = [p for n, p in param_dict.items() if p.dim() >= 2]
    nodecay_params = [p for n, p in param_dict.items() if p.dim() < 2]
    optim_groups = [
        {"params": decay_params, **config.optimizer},
        {"params": nodecay_params, **config.optimizer, "weight_decay": 0.0},
    ]
    num_decay_params = sum(p.numel() for p in decay_params)
    num_nodecay_params = sum(p.numel() for p in nodecay_params)
    print(f"Decayed parameter tensors: {len(decay_params)}, with {num_decay_params:,} parameters")
    print(f"Non-decayed parameter tensors: {len(nodecay_params)}, with {num_nodecay_params:,} parameters")

    optimizer = AdamW(optim_groups, **config.optimizer)
    # optimizer = AdamW(model.parameters(), **config.optimizer)

    # Load checkpoint if resuming
    if resuming:
        checkpoint_path = config.training.resume_checkpoint
        logger.log_info(f"Loading checkpoint from: {checkpoint_path}")
        start_step = load_checkpoint(checkpoint_path, model, optimizer) + 1
        logger.log_info(f"Resuming training from step {start_step}")

    # Compile + AMP on GPU, AOT on MPS
    use_compile = True
    if use_compile and device != "mps":
        model = torch.compile(model)
        torch.set_float32_matmul_precision("high")
    elif use_compile and device == "mps":
        model = torch.compile(model, backend="aot_eager") # this does not accelerate, but does check for errors via compilation before running

    max_steps = config.training.max_steps
    batch_size = config.training.batch_size
    max_l2_norm = config.training.max_l2_norm
    eval_interval = config.training.eval_interval
    checkpoint_interval = config.training.checkpoint_interval
    grad_accum_steps = config.training.grad_accum_steps

    lr_max = config.training.lr_max
    lr_inter = config.training.lr_inter
    lr_min = config.training.lr_min
    warmup_ratio = config.training.warmup_ratio
    warmup_iters = config.training.warmup_iters
    phase_one_iters = config.training.phase_one_iters
    phase_two_iters = config.training.phase_two_iters
    phase_two_type = config.training.phase_two_type
    cosine_cycle_iters = config.training.cosine_cycle_iters
    linear_cycle_iters = config.training.linear_cycle_iters

```

```
    if warmup_iters is False or warmup_iters is None:
        warmup_iters = int(warmup_ratio * max_steps)

    if cosine_cycle_iters is False or cosine_cycle_iters is None:
        cosine_cycle_iters = max_steps

    if linear_cycle_iters is False or linear_cycle_iters is None:
        linear_cycle_iters = max_steps

    if phase_two_iters is False or phase_two_iters is None:
        phase_two_iters = max_steps

    def evaluate(step: int, is_last_step: bool):
        n_eval_steps = config.training.eval_steps

        if is_last_step:
            n_eval_steps = n_eval_steps * 3

        model.eval()

        with torch.no_grad():
            val_loss = 0.0
            for _ in range(n_eval_steps):
                x, y = get_batch(valid_data, config.training.eval_batch_size,
config.model.context_length, device)
                with torch.autocast(device_type=device, dtype=dtype):
                    logits = model(x)
                loss = cross_entropy_loss(logits, y)
                val_loss += loss.item()
            val_loss /= n_eval_steps

        progress_str = get_progress_str(step, max_steps)

        # WandB metrics
        metrics = {
            "eval/loss": val_loss,
            "eval/perplexity": get_perplexity(val_loss),
            "eval/peak_memory": get_peak_memory(device),
            "step": step,
        }

        # Console + local file metrics
        display_metrics = {
            "step": progress_str,
            "v_loss": f"{val_loss:.4f}",
            "v_ppl": f"{get_perplexity(val_loss):.2f}",
            "mem": f"{get_peak_memory(device):.1f}MB",
        }

        logger.log_info(display_metrics)
```

```
        logger.log_metrics(metrics)

        # Restore to training mode
        model.train()

    # Only evaluate before training if not resuming
    if config.training.eval_before_training and not resuming:
        evaluate(0)

    # Load the first batch
    x, y = get_batch(train_data, batch_size, config.model.context_length, device)

    model.train()

    for step in range(start_step, max_steps + 1):
        t0 = time.time()
        is_last_step = step == max_steps
        loss_accum = 0.0

        # Gradient accumulation loop
        for _ in range(grad_accum_steps):
            with torch.autocast(device_type=device, dtype=dtype):
                logits = model(x)

            loss = cross_entropy_loss(logits, y) / grad_accum_steps

            x, y = get_batch(train_data, batch_size, config.model.context_length,
device)

            loss_accum += loss.detach()
            loss.backward()

        norm = gradient_clip(model.parameters(), max_l2_norm)

        if config.training.lr_schedule == "linear":
            lr = lr_linear_schedule(step, lr_max, lr_min, warmup_iters,
linear_cycle_iters)
        elif config.training.lr_schedule == "cosine":
            lr = lr_cosine_schedule(step, lr_max, lr_min, warmup_iters,
cosine_cycle_iters)
        elif config.training.lr_schedule == "double":
            lr = lr_double_schedule(
                step, lr_max, lr_inter, lr_min, warmup_iters, phase_one_iters,
phase_two_iters, phase_two_type
            )

        for param_group in optimizer.param_groups:
            param_group["lr"] = lr

        optimizer.step()
        optimizer.zero_grad(set_to_none=True)
```

```
        if device == "cuda":
            torch.cuda.synchronize()
        elif device == "mps":
            torch.mps.synchronize()

        t1 = time.time()
        dt = t1 - t0
        tokens_per_sec = config.model.context_length * batch_size * grad_accum_steps
/ dt
        train_loss = loss_accum.item()
        progress_str = get_progress_str(step, max_steps)

        # WandB metrics
        metrics = {
            "train/loss": train_loss,
            "train/perplexity": get_perplexity(train_loss),
            "train/lr": lr,
            "train/grad_norm": norm,
            "train/tokens_per_sec": tokens_per_sec,
            "train/peak_memory": get_peak_memory(device),
            "step": step,
        }

        # Console + local file metrics
        display_metrics = {
            "step": progress_str,
            "t_loss": f"{train_loss:.4f}",
            "t_ppl": f"{get_perplexity(train_loss):.2f}",
            "lr": f"{lr:.4e}",
            "grad_norm": f"{norm:.2f}",
            "mem": f"{get_peak_memory(device):.1f}MB",
            "tok/sec": f"{int(tokens_per_sec):,}",
            "dt": f"{dt * 1000:.2f}ms",
        }

        logger.log_info(display_metrics)
        logger.log_metrics(metrics)

        if step % eval_interval == 0 or is_last_step:
            evaluate(step, is_last_step)

        if step % checkpoint_interval == 0 or is_last_step:
            checkpoint_path = os.path.join(checkpoint_dir, f"checkpoint_{step}.pt")
            save_checkpoint(model, optimizer, step, checkpoint_path)

            # Create symlink pointing `latest` to checkpoint_path (remove `latest` if
it exists)
            latest_symlink = os.path.join(checkpoint_dir, "latest.pt")
            if os.path.islink(latest_symlink) or os.path.exists(latest_symlink):
                os.remove(latest_symlink)
```

```python
            os.symlink(os.path.abspath(checkpoint_path), latest_symlink)

            logger.log_info(f"Saved checkpoint to: {checkpoint_path}")

    wandb.finish()


def get_peak_memory(device):
    """Get peak memory usage in MB on the current device"""
    if device != "cuda":
        return 0

    peak_memory = torch.cuda.max_memory_allocated() / (1024 * 1024)
    torch.cuda.reset_peak_memory_stats()
    return peak_memory


def get_perplexity(loss):
    """Calculate perplexity from cross-entropy loss"""
    return math.exp(min(loss, 20))  # Cap at 20 to avoid overflow


def get_progress_str(step, max_steps):
    return f"step {step}/{max_steps} ({step / max_steps * 100:.2f}%)"


def parse_value(value_str: str):
    """Convert argparse arg to list, int, float, bool, or leave as string."""
    if value_str.strip().startswith("[") and value_str.strip().endswith("]"):
        content = value_str.strip()[1:-1].strip()
        return [parse_value(v.strip()) for v in content.split(",")] if content else []

    try:
        return int(value_str)
    except ValueError:
        pass

    try:
        return float(value_str)
    except ValueError:
        pass

    lower = value_str.strip().lower()
    if lower in ("true", "false"):
        return lower == "true"

    return value_str

```

```
def deep_set(config_dict, key_path: str, value):
    """Deeply set dot-separated key in a config dictionary"""
    keys = key_path.split(".")
    d = config_dict
    for k in keys[:-1]:
        if k not in d or not isinstance(d[k], dict):
            d[k] = {}
        d = d[k]
    d[keys[-1]] = value


if __name__ == "__main__":
    parser = argparse.ArgumentParser(description="Train a transformer model")
    parser.add_argument("--config", type=str, help="Path to config file (optional)")
    parser.add_argument("--resume-from", type=str, help="Path to run directory to
resume from")
    parser.add_argument("--override-config", type=str, help="Path to config file with
values to override when resuming")
    parser.add_argument(
        "--override-param",
        action="append",
        default=[],
        help="Override a config param, e.g. model.d_model=512 (can be repeated)",
    )
    args = parser.parse_args()

    if args.resume_from:
        # Load config from the previous run
        resume_config_path = os.path.join(args.resume_from, "config.json")
        base_config = load_config_from_file(resume_config_path)
        base_config["training"]["resume"] = True
        base_config["training"]["resume_checkpoint"] = os.path.join(args.resume_from,
"checkpoints/latest.pt")

        # Use the override config if provided, or None otherwise
        config = load_config(args.override_config, base_config=base_config)
    else:
        config = load_config(args.config)  # Will use default if args.config is None

    for override_str in args.override_param:
        if "=" not in override_str:
            raise ValueError(f"Invalid override: {override_str}, must be like
key=val")

        key, raw_value = override_str.split("=", 1)
        value = parse_value(raw_value)
        deep_set(config, key, value)

    train(config)
```

6 Generating text

Now that we can train models, the last piece we need is the ability to generate text from our model. Recall that a language model takes in a (possibly batched) integer sequence of length (`sequence_length`) and produces a matrix of size (`sequence_length` × `vocab size`), where each element of the sequence is a probability distribution predicting the next word after that position. We will now write a few functions to turn this into a sampling scheme for new sequences.

Softmax By standard convention, the language model output is the output of the final linear layer (the "logits") and so we have to turn this into a normalized probability via the *softmax* operation, which we saw earlier in Eq 10.

Decoding To generate text (decode) from our model, we will provide the model with a sequence of prefix tokens (the "prompt"), and ask it to produce a probability distribution over the vocabulary that predicts the next word in the sequence. Then, we will sample from this distribution over the vocabulary items to determine the next output token.

Concretely, one step of the decoding process should take in a sequence $x_{1\ldots t}$ and return a token x_{t+1} via the following equation,

$$P(x_{t+1} = i \mid x_{1\ldots t}) = \frac{\exp(v_i)}{\sum_j \exp(v_j)}$$
$$v = \text{TransformerLM}(x_{1\ldots t})_t \in \mathbb{R}^{\texttt{vocab\_size}}$$

where TransformerLM is our model which takes as input a sequence of `sequence_length` and produces a matrix of size (`sequence_length` × `vocab_size`), and we take the last element of this matrix, as we are looking for the next word prediction at the t-th position.

This gives us a basic decoder by repeatedly sampling from these one-step conditionals (appending our previously-generated output token to the input of the next decoding timestep) until we generate the end-of-sequence token `<|endoftext|>` (or a user-specified maximum number of tokens to generate).

Decoder tricks We will be experimenting with small models, and small models can sometimes generate very low quality texts. Two simple decoder tricks can help fix these issues. First, in *temperature scaling* we modify our softmax with a temperature parameter τ, where the new softmax is

$$\text{softmax}(v, \tau)_i = \frac{\exp(v_i/\tau)}{\sum_{j=1}^{|\text{vocab\_size}|} \exp(v_j/\tau)}. \tag{24}$$

Note how setting $\tau \to 0$ makes it so that the largest element of v dominates, and the output of the softmax becomes a one-hot vector concentrated at this maximal element.

Second, another trick is *nucleus* or *top-p* sampling, where we modify the sampling distribution by truncating low-probability words. Let q be a probability distribution that we get from a (temperature-scaled) softmax of size (`vocab_size`). Nucleus sampling with hyperparameter p produces the next token according to the equation

$$P(x_{t+1} = i|q) = \begin{cases} \frac{q_i}{\sum_{j\in V(p)} q_j} & \text{if } i \in V(p) \\ 0 & \text{otherwise} \end{cases}$$

where $V(p)$ is the *smallest* set of indices such that $\sum_{j\in V(p)} q_j \geq p$. You can compute this quantity easily by first sorting the probability distribution q by magnitude, and selecting the largest vocabulary elements until you reach the target level of α.

Problem (`decoding`): Decoding (3 points)

Deliverable: Implement a function to decode from your language model. We recommend that you support the following features:

- Generate completions for a user-provided prompt (i.e., take in some $x_{1\ldots t}$ and sample a completion until you hit an `<|endoftext|>` token).
- Allow the user to control the maximum number of generated tokens.
- Given a desired temperature value, apply softmax temperature scaling to the predicted next-word distributions before sampling.
- Top-p sampling (Holtzman et al., 2020; also referred to as nucleus sampling), given a user-specified threshold value.

~/Dropbox/Eric/Machine Learning/cs336/cs336-a1-main-brandon-snider/cs336_basics/decoding.py

```python
import torch
from cs336_basics.tokenizer import Tokenizer
from cs336_basics.model import Transformer


def decode(
    model: Transformer,
    tokenizer: Tokenizer,
    prompt: str,
    max_new_tokens: int = 32,
    temperature: float = 0.7,
    top_p: float = 0.9,
):
    end_id = tokenizer.encode("<|endoftext|>")[0]
    input_ids = tokenizer.encode(prompt)
    device = next(model.parameters()).device
    context_length = model.context_length

    with torch.no_grad():
        for _ in range(max_new_tokens):
            window_input_ids = input_ids[-context_length:] if len(input_ids) >= context_length else input_ids
            x = torch.tensor([window_input_ids], dtype=torch.long, device=device)

            logits = model(x)
            next_logits = logits[0, -1, :]

            scaled = next_logits / temperature
            stable = scaled - scaled.max()
            exp_vals = stable.exp()
            probs = exp_vals / exp_vals.sum()

            sorted_probs, sorted_idxs = torch.sort(probs, descending=True)
            cumsum = torch.cumsum(sorted_probs, dim=0)
            cutoff_idx = torch.searchsorted(cumsum, top_p)
            trimmed_probs = sorted_probs[: cutoff_idx + 1]
            trimmed_idxs = sorted_idxs[: cutoff_idx + 1]
            trimmed_probs /= trimmed_probs.sum()

            next_token = trimmed_idxs[torch.multinomial(trimmed_probs, 1).item()]
            if next_token.item() == end_id:
                break
            input_ids.append(next_token.item())

    return tokenizer.decode(input_ids)
```

7 Experiments

Now it is time to put everything together and train (small) language models on a pretaining dataset.

7.1 How to Run Experiments and Deliverables

The best way to understand the rationale behind the architectural components of a Transformer is to actually modify it and run it yourself. There is no substitute for hands-on experience.

To this end, it's important to be able to experiment **quickly, consistently, and keep records** of what you did. To experiment quickly, we will be running many experiments on a small scale model (17M parameters) and simple dataset (TinyStories). To do things consistently, you will ablate components and vary hyperparameters in a systematic way, and to keep records we will ask you to submit a log of your experiments and learning curves associated with each experiment.

To make it possible to submit loss curves, **make sure to periodically evaluate validation losses and record both the number of steps and wallclock times**. You might find logging infrastructure such as Weights and Biases helpful.

Problem (`experiment_log`): Experiment logging (3 points)

For your training and evaluation code, create experiment tracking infrastructure that allows you to track your experiments and loss curves with respect to gradient steps and wallclock time.

Deliverable: Logging infrastructure code for your experiments and an experiment log (a document of all the things you tried) for the assignment problems below in this section.

7.2 TinyStories

We are going to start with a very simple dataset (TinyStories; Eldan and Li, 2023) where models will train quickly, and we can see some interesting behaviors. The instructions for getting this dataset is at section 1. An example of what this dataset looks like is below.

Example (`tinystories_example`): One example from TinyStories

Once upon a time there was a little boy named Ben. Ben loved to explore the world around him. He saw many amazing things, like beautiful vases that were on display in a store. One day, Ben was walking through the store when he came across a very special vase. When Ben saw it he was amazed! He said, “Wow, that is a really amazing vase! Can I buy it?” The shopkeeper smiled and said, “Of course you can. You can take it home and show all your friends how amazing it is!” So Ben took the vase home and he was so proud of it! He called his friends over and showed them the amazing vase. All his friends thought the vase was beautiful and couldn't believe how lucky Ben was. And that's how Ben found an amazing vase in the store!

Hyperparameter tuning We will tell you some very basic hyperparameters to start with and ask you to find some settings for others that work well.

`vocab_size` 10000. Typical vocabulary sizes are in the tens to hundreds of thousands. You should vary this and see how the vocabulary and model behavior changes.

`context_length` 256. Simple datasets such as TinyStories might not need long sequence lengths, but for the later OpenWebText data, you may want to vary this. Try varying this and seeing the impact on both the per-iteration runtime and the final perplexity.

`d_model` 512. This is slightly smaller than the 768 dimensions used in many small Transformer papers, but this will make things faster.

`d_ff` 1344. This is roughly $\frac{8}{3}$`d_model` while being a multiple of 64, which is good for GPU performance.

`RoPE theta parameter` Θ 10000.

`number of layers and heads` 4 layers, 16 heads. Together, this will give about 17M non-embedding parameters which is a fairly small Transformer.

`total tokens processed` 327,680,000 (your batch size $\times$ total step count $\times$ context length should equal roughly this value).

You should do some trial and error to find good defaults for the following other hyperparameters: `learning rate`, `learning rate warmup`, `other AdamW hyperparameters` ($\beta_1, \beta_2, \epsilon$), and `weight decay`. You can find some typical choices of such hyperparameters in Kingma and Ba [2015].

Putting it together Now you can put everything together by getting a trained BPE tokenizer, tokenizing the training dataset, and running this in the training loop that you wrote. **Important note:** If your implementation is correct and efficient, the above hyperparameters should result in a roughly 30-40 minute runtime on 1 H100 GPU. If you have runtimes that are much longer, please check and make sure your dataloading, checkpointing, or validation loss code is not bottlenecking your runtimes and that your implementation is properly batched.

Tips and tricks for debugging model architectures We highly recommend getting comfortable with your IDE's built-in debugger (e.g., VSCode/PyCharm), which will save you time compared to debugging with print statements. If you use a text editor, you can use something more like `pdb`. A few other good practices when debugging model architectures are:

- A common first step when developing any neural net architecture is to overfit to a single minibatch. If your implementation is correct, you should be able to quickly drive the training loss to near-zero.
- Set debug breakpoints in various model components, and inspect the shapes of intermediate tensors to make sure they match your expectations.
- Monitor the norms of activations, model weights, and gradients to make sure they are not exploding or vanishing.

Problem (`learning_rate`): Tune the learning rate (3 points) (4 H100 hrs)

The learning rate is one of the most important hyperparameters to tune. Taking the base model you've trained, answer the following questions:

(a) Perform a hyperparameter sweep over the learning rates and report the final losses (or note divergence if the optimizer diverges).

Deliverable: Learning curves associated with multiple learning rates. Explain your hyperparameter search strategy.

Deliverable: A model with validation loss (per-token) on TinyStories of at most 1.45

Low-Resource/Downscaling Tip: Train for few steps on CPU or Apple Silicon

If you are running on `cpu` or `mps`, you should instead reduce the total tokens processed count to $40,000,000$, which will be sufficient to produce reasonably fluent text. You may also increase the target validation loss from 1.45 to 2.00.

Running our solution code with a tuned learning rate on an M3 Max chip and 36 GB of RAM, we use batch size $\times$ total step count $\times$ context length $= 32 \times 5000 \times 256 = 40,960,000$ tokens, which takes 1 hour and 22 minutes on `cpu` and 36 minutes on `mps`. At step 5000, we achieve a validation loss of 1.80.

Some additional tips:

- When using X training steps, we suggest adjusting the cosine learning rate decay schedule to terminate its decay (i.e., reach the minimum learning rate) at precisely step X.
- When using `mps`, do **not** use TF32 kernels, i.e., do **not** set

```
torch.set_float32_matmul_precision('high')
```

  as you might with `cuda` devices. We tried enabling TF32 kernels with `mps` (`torch` version 2.6.0) and found the backend will use silently broken kernels that cause unstable training.
- You can speed up training by JIT-compiling your model with `torch.compile`. Specifically:
  - On `cpu`, compile your model with

```
model = torch.compile(model)
```

  - On `mps`, you can somewhat optimize the backward pass using

```
model = torch.compile(model, backend="aot_eager")
```

    Compilation with Inductor is not supported on `mps` as of `torch` version 2.6.0.

(b) Folk wisdom is that the best learning rate is "at the edge of stability." Investigate how the point at which learning rates diverge is related to your best learning rate.

Deliverable: Learning curves of increasing learning rate which include at least one divergent run and an analysis of how this relates to convergence rates.

Now let's vary the batch size and see what happens to training. Batch sizes are important – they let us get higher efficiency from our GPUs by doing larger matrix multiplies, but is it true that we always want batch sizes to be large? Let's run some experiments to find out.

Problem (`batch_size_experiment`): Batch size variations (1 point) (2 H100 hrs)

Vary your batch size all the way from 1 to the GPU memory limit. Try at least a few batch sizes in between, including typical sizes like 64 and 128.

Deliverable: Learning curves for runs with different batch sizes. The learning rates should be optimized again if necessary.

Deliverable: A few sentences discussing of your findings on batch sizes and their impacts on training.

With your decoder in hand, we can now generate text! We will generate from the model and see how good it is. As a reference, you should get outputs that look at least as good as the example below.

Example (ts_generate_example): Sample output from a TinyStories language model

Once upon a time, there was a pretty girl named Lily. She loved to eat gum, especially the big black one. One day, Lily's mom asked her to help cook dinner. Lily was so excited! She loved to help her mom. Lily's mom made a big pot of soup for dinner. Lily was so happy and said, "Thank you, Mommy! I love you." She helped her mom pour the soup into a big bowl. After dinner, Lily's mom made some yummy soup. Lily loved it! She said, "Thank you, Mommy! This soup is so yummy!" Her mom smiled and said, "I'm glad you like it, Lily." They finished cooking and continued to cook together. The end.

Low-Resource/Downscaling Tip: Generate text on CPU or Apple Silicon

If instead you used the low-resource configuration with 40M tokens processed, you should see generations that still resemble English but are not as fluent as above. For example, our sample output from a TinyStories language model trained on 40M tokens is below:

Once upon a time, there was a little girl named Sue. Sue had a tooth that she loved very much. It was his best head. One day, Sue went for a walk and met a ladybug! They became good friends and played on the path together.

"Hey, Polly! Let's go out!" said Tim. Sue looked at the sky and saw that it was difficult to find a way to dance shining. She smiled and agreed to help the talking!"

As Sue watched the sky moved, what it was. She

Here is the precise problem statement and what we ask for:

Problem (generate): Generate text (1 point)

Using your decoder and your trained checkpoint, report the text generated by your model. You may need to manipulate decoder parameters (temperature, top-p, etc.) to get fluent outputs.

Deliverable: Text dump of at least 256 tokens of text (or until the first <|endoftext|> token), and a brief comment on the fluency of this output and at least two factors which affect how good or bad this output is.

7.3 Ablations and architecture modification

The best way to understand the Transformer is to actually modify it and see how it behaves. We will now do a few simple ablations and modifications.

Ablation 1: layer normalization It is often said that layer normalization is important for the stability of Transformer training. But perhaps we want to live dangerously. Let's remove RMSNorm from each of our Transformer blocks and see what happens.

Problem (layer_norm_ablation): Remove RMSNorm and train (1 point) (1 H100 hr)

Remove all of the RMSNorms from your Transformer and train. What happens at the previous optimal learning rate? Can you get stability by using a lower learning rate?

Deliverable: A learning curve for when you remove RMSNorms and train, as well as a learning curve for the best learning rate.

Deliverable: A few sentence commentary on the impact of RMSNorm.

Let's now investigate another layer normalization choice that seems arbitrary at first glance. *Pre-norm* Transformer blocks are defined as

$$
\begin{aligned}
z &= x + \mathrm{MultiHeadedSelfAttention}(\mathrm{RMSNorm}(x)) \\
y &= z + \mathrm{FFN}(\mathrm{RMSNorm}(z)).
\end{aligned}
$$

This is one of the few 'consensus' modifications to the original Transformer architecture, which used a *post-norm* approach as

$$
\begin{aligned}
z &= \mathrm{RMSNorm}(x + \mathrm{MultiHeadedSelfAttention}(x)) \\
y &= \mathrm{RMSNorm}(z + \mathrm{FFN}(z)).
\end{aligned}
$$

Let's revert back to the *post-norm* approach and see what happens.

Problem (`pre_norm_ablation`): Implement post-norm and train (1 point) (1 H100 hr)

Modify your pre-norm Transformer implementation into a post-norm one. Train with the post-norm model and see what happens.

Deliverable: A learning curve for a post-norm transformer, compared to the pre-norm one.

We see that layer normalization has a major impact on the behavior of the transformer, and that even the position of the layer normalization is important.

Ablation 2: position embeddings We will next investigate the impact of the position embeddings on the performance of the model. Specifically, we will compare our base model (with RoPE) with not including position embeddings at all (NoPE). It turns out that decoder-only transformers, i.e., those with a causal mask as we have implemented, can in theory infer relative or absolute position information without being provided with position embeddings explicitly [Tsai et al., 2019, Kazemnejad et al., 2023]. We will now test empirically how NoPE performs compare to RoPE.

Problem (`no_pos_emb`): Implement NoPE (1 point) (1 H100 hr)

Modify your Transformer implementation with RoPE to remove the position embedding information entirely, and see what happens.

Deliverable: A learning curve comparing the performance of RoPE and NoPE.

Ablation 3: SwiGLU vs. SiLU Next, we will follow Shazeer [2020] and test the importance of gating in the feed-forward network, by comparing the performance of SwiGLU feed-forward networks versus feed-forward networks using SiLU activations but no gated linear unit (GLU):

$$
\mathrm{FFN}_{\mathrm{SiLU}}(x) = W_2 \mathrm{SiLU}(W_1 x). \tag{25}
$$

Recall that in our SwiGLU implementation, we set the dimensionality of the inner feed-forward layer to be roughly $d_{\mathrm{ff}} = \frac{8}{3} d_{\mathrm{model}}$ (while ensuring that $d_{\mathrm{ff}} \mod 64 = 0$, to make use of GPU tensor cores). In your $\mathrm{FFN}_{\mathrm{SiLU}}$ implementation you should set $d_{\mathrm{ff}} = 4 \times d_{\mathrm{model}}$, to approximately match the parameter count of the SwiGLU feed-forward network (which has three instead of two weight matrices).

Problem (`swiglu_ablation`): SwiGLU vs. SiLU (1 point) (1 H100 hr)

Deliverable: A learning curve comparing the performance of SwiGLU and SiLU feed-forward networks, with approximately matched parameter counts.

Deliverable: A few sentences discussing your findings.

Low-Resource/Downscaling Tip: Online students with limited GPU resources should test modifications on TinyStories

In the remainder of the assignment, we will move to a larger-scale, noisier web dataset (OpenWebText), experimenting with architecture modifications and (optionally) making a submission to the course leaderboard.

It takes a long time to train an LM to fluency on OpenWebText, so we suggest that online students with limited GPU access continue testing modifications on TinyStories (using validation loss as a metric to evaluate performance).

7.4 Running on OpenWebText

We will now move to a more standard pretraining dataset created from a webcrawl. A small sample of OpenWebText [Gokaslan et al., 2019] is also provided as a single text file: see section 1 for how to access this file.

Here is an example from OpenWebText. Note how the text is much more realistic, complex, and varied. You may want to look through the training dataset to get a sense of what training data looks like for a webscraped corpus.

Example (`owt_example`): One example from OWT

Baseball Prospectus director of technology Harry Pavlidis took a risk when he hired Jonathan Judge.

Pavlidis knew that, as Alan Schwarz wrote in The Numbers Game, “no corner of American culture is more precisely counted, more passionately quantified, than performances of baseball players.” With a few clicks here and there, you can findout that Noah Syndergaard’s fastball revolves more than 2,100 times per minute on its way to the plate, that Nelson Cruz had the game’s highest average exit velocity among qualified hitters in 2016 and myriad other tidbits that seem ripped from a video game or science fiction novel. The rising ocean of data has empowered an increasingly important actor in baseball’s culture: the analytical hobbyist.

That empowerment comes with added scrutiny – on the measurements, but also on the people and publications behind them. With Baseball Prospectus, Pavlidis knew all about the backlash that accompanies quantitative imperfection. He also knew the site’s catching metrics needed to be reworked, and that it would take a learned mind – someone who could tackle complex statistical modeling problems – to complete the job.

“He freaks us out.” Harry Pavlidis

Pavlidis had a hunch that Judge “got it” based on the latter’s writing and their interaction at a site-sponsored ballpark event. Soon thereafter, the two talked over drinks. Pavlidis’ intuition was validated. Judge was a fit for the position – better yet, he was a willing fit. “I spoke to a lot of people,” Pavlidis said, “he was the only one brave enough to take it on.” [...]

Note: You may have to re-tune your hyperparameters such as learning rate or batch size for this experiment.

Problem (`main_experiment`): Experiment on OWT (2 points) (3 H100 hrs)

Train your language model on OpenWebText with the same model architecture and total training iterations as TinyStories. How well does this model do?

Deliverable: A learning curve of your language model on OpenWebText. Describe the difference in losses from TinyStories – how should we interpret these losses?

Deliverable: Generated text from OpenWebText LM, in the same format as the TinyStories outputs. How is the fluency of this text? Why is the output quality worse even though we have the same model and compute budget as TinyStories?

7.5 Your own modification + leaderboard

Congratulations on getting to this point. You're almost done! You will now try to improve upon the Transformer architecture, and see how your hyperparameters and architecture stack up against other students in the class.

Rules for the leaderboard There are no restrictions other than the following:

`Runtime` Your submission can run for at most 1.5 hours on an H100. You can enforce this by setting `--time=01:30:00` in your slurm submission script.

`Data` You may only use the OpenWebText training dataset that we provide.

Otherwise, you are free to do whatever your heart desires.

If you are looking for some ideas on what to implement, you can checkout some of these resources:

- State-of-the-art open-source LLM families, such as Llama 3 [Grattafiori et al., 2024] or Qwen 2.5 [Yang et al., 2024].
- The NanoGPT speedrun repository (`https://github.com/KellerJordan/modded-nanogpt`), where community members post many interesting modifications for "speedrunning" small-scale language model pretraining. For example, a common modification that dates back to the original Transformer paper is to tie the weights of the input and output embeddings together (see Vaswani et al. [2017] (Section 3.4) and Chowdhery et al. [2022] (Section 2)). If you do try weight tying, you may have to decrease the standard deviation of the embedding/LM head init.

You will want to test these on either a small subset of OpenWebText or on TinyStories before trying the full 1.5-hour run.

As a caveat, we do note that some of the modifications you may find working well in this leaderboard may not generalize to larger-scale pretraining. We will explore this idea further in the scaling laws unit of the course.

Problem (`leaderboard`): Leaderboard (6 points) (10 H100 hrs)

You will train a model under the leaderboard rules above with the goal of minimizing the validation loss of your language model within 1.5 H100-hour.

Deliverable: The final validation loss that was recorded, an associated learning curve that clearly shows a wallclock-time x-axis that is less than 1.5 hours and a description of what you did. We expect a leaderboard submission to beat at least the naive baseline of a 5.0 loss. Submit to the leaderboard here: https://github.com/stanford-cs336/assignment1-basics-leaderboard.

References

Ronen Eldan and Yuanzhi Li. TinyStories: How small can language models be and still speak coherent English?, 2023. arXiv:2305.07759.

Aaron Gokaslan, Vanya Cohen, Ellie Pavlick, and Stefanie Tellex. OpenWebText corpus. `http://Skylion007.github.io/OpenWebTextCorpus`, 2019.

Rico Sennrich, Barry Haddow, and Alexandra Birch. Neural machine translation of rare words with subword units. In *Proc. of ACL*, 2016.

Changhan Wang, Kyunghyun Cho, and Jiatao Gu. Neural machine translation with byte-level subwords, 2019. arXiv:1909.03341.

Philip Gage. A new algorithm for data compression. *C Users Journal*, 12(2):23–38, February 1994. ISSN 0898-9788.

Alec Radford, Jeff Wu, Rewon Child, David Luan, Dario Amodei, and Ilya Sutskever. Language models are unsupervised multitask learners, 2019.

Alec Radford, Karthik Narasimhan, Tim Salimans, and Ilya Sutskever. Improving language understanding by generative pre-training, 2018.

Ashish Vaswani, Noam Shazeer, Niki Parmar, Jakob Uszkoreit, Llion Jones, Aidan N Gomez, Ł ukasz Kaiser, and Illia Polosukhin. Attention is all you need. In *Proc. of NeurIPS*, 2017.

Toan Q. Nguyen and Julian Salazar. Transformers without tears: Improving the normalization of self-attention. In *Proc. of IWSWLT*, 2019.

Ruibin Xiong, Yunchang Yang, Di He, Kai Zheng, Shuxin Zheng, Chen Xing, Huishuai Zhang, Yanyan Lan, Liwei Wang, and Tie-Yan Liu. On layer normalization in the Transformer architecture. In *Proc. of ICML*, 2020.

Jimmy Lei Ba, Jamie Ryan Kiros, and Geoffrey E. Hinton. Layer normalization, 2016. arXiv:1607.06450.

Hugo Touvron, Thibaut Lavril, Gautier Izacard, Xavier Martinet, Marie-Anne Lachaux, Timothée Lacroix, Baptiste Rozière, Naman Goyal, Eric Hambro, Faisal Azhar, Aurelien Rodriguez, Armand Joulin, Edouard Grave, and Guillaume Lample. Llama: Open and efficient foundation language models, 2023. arXiv:2302.13971.

Biao Zhang and Rico Sennrich. Root mean square layer normalization. In *Proc. of NeurIPS*, 2019.

Aaron Grattafiori, Abhimanyu Dubey, Abhinav Jauhri, Abhinav Pandey, Abhishek Kadian, Ahmad Al-Dahle, Aiesha Letman, Akhil Mathur, Alan Schelten, Alex Vaughan, Amy Yang, Angela Fan, Anirudh Goyal, Anthony Hartshorn, Aobo Yang, Archi Mitra, Archie Sravankumar, Artem Korenev, Arthur Hinsvark, Arun Rao, Aston Zhang, Aurelien Rodriguez, Austen Gregerson, Ava Spataru, Baptiste Roziere, Bethany Biron, Binh Tang, Bobbie Chern, Charlotte Caucheteux, Chaya Nayak, Chloe Bi, Chris Marra, Chris McConnell, Christian Keller, Christophe Touret, Chunyang Wu, Corinne Wong, Cristian Canton Ferrer, Cyrus Nikolaidis, Damien Allonsius, Daniel Song, Danielle Pintz, Danny Livshits, Danny Wyatt, David Esiobu, Dhruv Choudhary, Dhruv Mahajan, Diego Garcia-Olano, Diego Perino, Dieuwke Hupkes, Egor Lakomkin, Ehab AlBadawy, Elina Lobanova, Emily Dinan, Eric Michael Smith, Filip Radenovic, Francisco Guzmán, Frank Zhang, Gabriel Synnaeve, Gabrielle Lee, Georgia Lewis Anderson, Govind Thattai, Graeme Nail, Gregoire Mialon, Guan Pang, Guillem Cucurell, Hailey Nguyen, Hannah Korevaar, Hu Xu, Hugo Touvron, Iliyan Zarov, Imanol Arrieta Ibarra, Isabel Kloumann, Ishan Misra, Ivan Evtimov, Jack Zhang, Jade Copet, Jaewon Lee, Jan Geffert, Jana Vranes, Jason Park, Jay Mahadeokar, Jeet Shah, Jelmer van der Linde, Jennifer Billock, Jenny Hong, Jenya Lee, Jeremy Fu, Jianfeng Chi, Jianyu

Huang, Jiawen Liu, Jie Wang, Jiecao Yu, Joanna Bitton, Joe Spisak, Jongsoo Park, Joseph Rocca, Joshua Johnstun, Joshua Saxe, Junteng Jia, Kalyan Vasuden Alwala, Karthik Prasad, Kartikeya Upasani, Kate Plawiak, Ke Li, Kenneth Heafield, Kevin Stone, Khalid El-Arini, Krithika Iyer, Kshitiz Malik, Kuenley Chiu, Kunal Bhalla, Kushal Lakhotia, Lauren Rantala-Yeary, Laurens van der Maaten, Lawrence Chen, Liang Tan, Liz Jenkins, Louis Martin, Lovish Madaan, Lubo Malo, Lukas Blecher, Lukas Landzaat, Luke de Oliveira, Madeline Muzzi, Mahesh Pasupuleti, Mannat Singh, Manohar Paluri, Marcin Kardas, Maria Tsimpoukelli, Mathew Oldham, Mathieu Rita, Maya Pavlova, Melanie Kambadur, Mike Lewis, Min Si, Mitesh Kumar Singh, Mona Hassan, Naman Goyal, Narjes Torabi, Nikolay Bashlykov, Nikolay Bogoychev, Niladri Chatterji, Ning Zhang, Olivier Duchenne, Onur Çelebi, Patrick Alrassy, Pengchuan Zhang, Pengwei Li, Petar Vasic, Peter Weng, Prajjwal Bhargava, Pratik Dubal, Praveen Krishnan, Punit Singh Koura, Puxin Xu, Qing He, Qingxiao Dong, Ragavan Srinivasan, Raj Ganapathy, Ramon Calderer, Ricardo Silveira Cabral, Robert Stojnic, Roberta Raileanu, Rohan Maheswari, Rohit Girdhar, Rohit Patel, Romain Sauvestre, Ronnie Polidoro, Roshan Sumbaly, Ross Taylor, Ruan Silva, Rui Hou, Rui Wang, Saghar Hosseini, Sahana Chennabasappa, Sanjay Singh, Sean Bell, Seohyun Sonia Kim, Sergey Edunov, Shaoliang Nie, Sharan Narang, Sharath Raparthy, Sheng Shen, Shengye Wan, Shruti Bhosale, Shun Zhang, Simon Vandenhende, Soumya Batra, Spencer Whitman, Sten Sootla, Stephane Collot, Suchin Gururangan, Sydney Borodinsky, Tamar Herman, Tara Fowler, Tarek Sheasha, Thomas Georgiou, Thomas Scialom, Tobias Speckbacher, Todor Mihaylov, Tong Xiao, Ujjwal Karn, Vedanuj Goswami, Vibhor Gupta, Vignesh Ramanathan, Viktor Kerkez, Vincent Gonguet, Virginie Do, Vish Vogeti, Vítor Albiero, Vladan Petrovic, Weiwei Chu, Wenhan Xiong, Wenyin Fu, Whitney Meers, Xavier Martinet, Xiaodong Wang, Xiaofang Wang, Xiaoqing Ellen Tan, Xide Xia, Xinfeng Xie, Xuchao Jia, Xuewei Wang, Yaelle Goldschlag, Yashesh Gaur, Yasmine Babaei, Yi Wen, Yiwen Song, Yuchen Zhang, Yue Li, Yuning Mao, Zacharie Delpierre Coudert, Zheng Yan, Zhengxing Chen, Zoe Papakipos, Aaditya Singh, Aayushi Srivastava, Abha Jain, Adam Kelsey, Adam Shajnfeld, Adithya Gangidi, Adolfo Victoria, Ahuva Goldstand, Ajay Menon, Ajay Sharma, Alex Boesenberg, Alexei Baevski, Allie Feinstein, Amanda Kallet, Amit Sangani, Amos Teo, Anam Yunus, Andrei Lupu, Andres Alvarado, Andrew Caples, Andrew Gu, Andrew Ho, Andrew Poulton, Andrew Ryan, Ankit Ramchandani, Annie Dong, Annie Franco, Anuj Goyal, Aparajita Saraf, Arkabandhu Chowdhury, Ashley Gabriel, Ashwin Bharambe, Assaf Eisenman, Azadeh Yazdan, Beau James, Ben Maurer, Benjamin Leonhardi, Bernie Huang, Beth Loyd, Beto De Paola, Bhargavi Paranjape, Bing Liu, Bo Wu, Boyu Ni, Braden Hancock, Bram Wasti, Brandon Spence, Brani Stojkovic, Brian Gamido, Britt Montalvo, Carl Parker, Carly Burton, Catalina Mejia, Ce Liu, Changhan Wang, Changkyu Kim, Chao Zhou, Chester Hu, Ching-Hsiang Chu, Chris Cai, Chris Tindal, Christoph Feichtenhofer, Cynthia Gao, Damon Civin, Dana Beaty, Daniel Kreymer, Daniel Li, David Adkins, David Xu, Davide Testuggine, Delia David, Devi Parikh, Diana Liskovich, Didem Foss, Dingkang Wang, Duc Le, Dustin Holland, Edward Dowling, Eissa Jamil, Elaine Montgomery, Eleonora Presani, Emily Hahn, Emily Wood, Eric-Tuan Le, Erik Brinkman, Esteban Arcaute, Evan Dunbar, Evan Smothers, Fei Sun, Felix Kreuk, Feng Tian, Filippos Kokkinos, Firat Ozgenel, Francesco Caggioni, Frank Kanayet, Frank Seide, Gabriela Medina Florez, Gabriella Schwarz, Gada Badeer, Georgia Swee, Gil Halpern, Grant Herman, Grigory Sizov, Guangyi, Zhang, Guna Lakshminarayanan, Hakan Inan, Hamid Shojanazeri, Han Zou, Hannah Wang, Hanwen Zha, Haroun Habeeb, Harrison Rudolph, Helen Suk, Henry Aspegren, Hunter Goldman, Hongyuan Zhan, Ibrahim Damlaj, Igor Molybog, Igor Tufanov, Ilias Leontiadis, Irina-Elena Veliche, Itai Gat, Jake Weissman, James Geboski, James Kohli, Janice Lam, Japhet Asher, Jean-Baptiste Gaya, Jeff Marcus, Jeff Tang, Jennifer Chan, Jenny Zhen, Jeremy Reizenstein, Jeremy Teboul, Jessica Zhong, Jian Jin, Jingyi Yang, Joe Cummings, Jon Carvill, Jon Shepard, Jonathan McPhie, Jonathan Torres, Josh Ginsburg, Junjie Wang, Kai Wu, Kam Hou U, Karan Saxena, Kartikay Khandelwal, Katayoun Zand, Kathy Matosich, Kaushik Veeraraghavan, Kelly Michelena, Keqian Li, Kiran Jagadeesh, Kun Huang, Kunal Chawla, Kyle Huang, Lailin Chen, Lakshya Garg, Lavender A, Leandro Silva, Lee Bell, Lei Zhang, Liangpeng Guo, Licheng Yu, Liron Moshkovich, Luca Wehrstedt, Madian Khabsa, Manav Avalani, Manish Bhatt, Martynas Mankus, Matan Hasson, Matthew Lennie, Matthias Reso, Maxim Groshev, Maxim Naumov, Maya Lathi, Meghan Keneally, Miao Liu, Michael L. Seltzer, Michal Valko, Michelle Restrepo, Mihir Patel, Mik Vyatskov, Mikayel Samvelyan, Mike Clark, Mike Macey, Mike Wang, Miquel Jubert Hermoso, Mo Metanat, Moham-

mad Rastegari, Munish Bansal, Nandhini Santhanam, Natascha Parks, Natasha White, Navyata Bawa, Nayan Singhal, Nick Egebo, Nicolas Usunier, Nikhil Mehta, Nikolay Pavlovich Laptev, Ning Dong, Norman Cheng, Oleg Chernoguz, Olivia Hart, Omkar Salpekar, Ozlem Kalinli, Parkin Kent, Parth Parekh, Paul Saab, Pavan Balaji, Pedro Rittner, Philip Bontrager, Pierre Roux, Piotr Dollar, Polina Zvyagina, Prashant Ratanchandani, Pritish Yuvraj, Qian Liang, Rachad Alao, Rachel Rodriguez, Rafi Ayub, Raghotham Murthy, Raghu Nayani, Rahul Mitra, Rangaprabhu Parthasarathy, Raymond Li, Rebekkah Hogan, Robin Battey, Rocky Wang, Russ Howes, Ruty Rinott, Sachin Mehta, Sachin Siby, Sai Jayesh Bondu, Samyak Datta, Sara Chugh, Sara Hunt, Sargun Dhillon, Sasha Sidorov, Satadru Pan, Saurabh Mahajan, Saurabh Verma, Seiji Yamamoto, Sharadh Ramaswamy, Shaun Lindsay, Shaun Lindsay, Sheng Feng, Shenghao Lin, Shengxin Cindy Zha, Shishir Patil, Shiva Shankar, Shuqiang Zhang, Shuqiang Zhang, Sinong Wang, Sneha Agarwal, Soji Sajuyigbe, Soumith Chintala, Stephanie Max, Stephen Chen, Steve Kehoe, Steve Satterfield, Sudarshan Govindaprasad, Sumit Gupta, Summer Deng, Sungmin Cho, Sunny Virk, Suraj Subramanian, Sy Choudhury, Sydney Goldman, Tal Remez, Tamar Glaser, Tamara Best, Thilo Koehler, Thomas Robinson, Tianhe Li, Tianjun Zhang, Tim Matthews, Timothy Chou, Tzook Shaked, Varun Vontimitta, Victoria Ajayi, Victoria Montanez, Vijai Mohan, Vinay Satish Kumar, Vishal Mangla, Vlad Ionescu, Vlad Poenaru, Vlad Tiberiu Mihailescu, Vladimir Ivanov, Wei Li, Wenchen Wang, Wenwen Jiang, Wes Bouaziz, Will Constable, Xiaocheng Tang, Xiaojian Wu, Xiaolan Wang, Xilun Wu, Xinbo Gao, Yaniv Kleinman, Yanjun Chen, Ye Hu, Ye Jia, Ye Qi, Yenda Li, Yilin Zhang, Ying Zhang, Yossi Adi, Youngjin Nam, Yu, Wang, Yu Zhao, Yuchen Hao, Yundi Qian, Yunlu Li, Yuzi He, Zach Rait, Zachary DeVito, Zef Rosnbrick, Zhaoduo Wen, Zhenyu Yang, Zhiwei Zhao, and Zhiyu Ma. The llama 3 herd of models, 2024. URL https://arxiv.org/abs/2407.21783.

An Yang, Baosong Yang, Beichen Zhang, Binyuan Hui, Bo Zheng, Bowen Yu, Chengyuan Li, Dayiheng Liu, Fei Huang, Haoran Wei, Huan Lin, Jian Yang, Jianhong Tu, Jianwei Zhang, Jianxin Yang, Jiaxi Yang, Jingren Zhou, Junyang Lin, Kai Dang, Keming Lu, Keqin Bao, Kexin Yang, Le Yu, Mei Li, Mingfeng Xue, Pei Zhang, Qin Zhu, Rui Men, Runji Lin, Tianhao Li, Tingyu Xia, Xingzhang Ren, Xuancheng Ren, Yang Fan, Yang Su, Yichang Zhang, Yu Wan, Yuqiong Liu, Zeyu Cui, Zhenru Zhang, and Zihan Qiu. Qwen2.5 technical report. *arXiv preprint arXiv:2412.15115*, 2024.

Aakanksha Chowdhery, Sharan Narang, Jacob Devlin, Maarten Bosma, Gaurav Mishra, Adam Roberts, Paul Barham, Hyung Won Chung, Charles Sutton, Sebastian Gehrmann, Parker Schuh, Kensen Shi, Sasha Tsvyashchenko, Joshua Maynez, Abhishek Rao, Parker Barnes, Yi Tay, Noam Shazeer, Vinodkumar Prabhakaran, Emily Reif, Nan Du, Ben Hutchinson, Reiner Pope, James Bradbury, Jacob Austin, Michael Isard, Guy Gur-Ari, Pengcheng Yin, Toju Duke, Anselm Levskaya, Sanjay Ghemawat, Sunipa Dev, Henryk Michalewski, Xavier Garcia, Vedant Misra, Kevin Robinson, Liam Fedus, Denny Zhou, Daphne Ippolito, David Luan, Hyeontaek Lim, Barret Zoph, Alexander Spiridonov, Ryan Sepassi, David Dohan, Shivani Agrawal, Mark Omernick, Andrew M. Dai, Thanumalayan Sankaranarayana Pillai, Marie Pellat, Aitor Lewkowycz, Erica Moreira, Rewon Child, Oleksandr Polozov, Katherine Lee, Zongwei Zhou, Xuezhi Wang, Brennan Saeta, Mark Diaz, Orhan Firat, Michele Catasta, Jason Wei, Kathy Meier-Hellstern, Douglas Eck, Jeff Dean, Slav Petrov, and Noah Fiedel. PaLM: Scaling language modeling with pathways, 2022. arXiv:2204.02311.

Dan Hendrycks and Kevin Gimpel. Bridging nonlinearities and stochastic regularizers with gaussian error linear units, 2016. arXiv:1606.08415.

Stefan Elfwing, Eiji Uchibe, and Kenji Doya. Sigmoid-weighted linear units for neural network function approximation in reinforcement learning, 2017. URL https://arxiv.org/abs/1702.03118.

Yann N. Dauphin, Angela Fan, Michael Auli, and David Grangier. Language modeling with gated convolutional networks, 2017. URL https://arxiv.org/abs/1612.08083.

Noam Shazeer. GLU variants improve transformer, 2020. arXiv:2002.05202.

Jianlin Su, Yu Lu, Shengfeng Pan, Bo Wen, and Yunfeng Liu. Roformer: Enhanced transformer with rotary position embedding, 2021.

Diederik P. Kingma and Jimmy Ba. Adam: A method for stochastic optimization. In *Proc. of ICLR*, 2015.

Ilya Loshchilov and Frank Hutter. Decoupled weight decay regularization. In *Proc. of ICLR*, 2019.

Tom B. Brown, Benjamin Mann, Nick Ryder, Melanie Subbiah, Jared Kaplan, Prafulla Dhariwal, Arvind Neelakantan, Pranav Shyam, Girish Sastry, Amanda Askell, Sandhini Agarwal, Ariel Herbert-Voss, Gretchen Krueger, Tom Henighan, Rewon Child, Aditya Ramesh, Daniel M. Ziegler, Jeffrey Wu, Clemens Winter, Christopher Hesse, Mark Chen, Eric Sigler, Mateusz Litwin, Scott Gray, Benjamin Chess, Jack Clark, Christopher Berner, Sam McCandlish, Alec Radford, Ilya Sutskever, and Dario Amodei. Language models are few-shot learners. In *Proc. of NeurIPS*, 2020.

Jared Kaplan, Sam McCandlish, Tom Henighan, Tom B. Brown, Benjamin Chess, Rewon Child, Scott Gray, Alec Radford, Jeffrey Wu, and Dario Amodei. Scaling laws for neural language models, 2020. arXiv:2001.08361.

Jordan Hoffmann, Sebastian Borgeaud, Arthur Mensch, Elena Buchatskaya, Trevor Cai, Eliza Rutherford, Diego de Las Casas, Lisa Anne Hendricks, Johannes Welbl, Aidan Clark, Tom Hennigan, Eric Noland, Katie Millican, George van den Driessche, Bogdan Damoc, Aurelia Guy, Simon Osindero, Karen Simonyan, Erich Elsen, Jack W. Rae, Oriol Vinyals, and Laurent Sifre. Training compute-optimal large language models, 2022. arXiv:2203.15556.

Ari Holtzman, Jan Buys, Li Du, Maxwell Forbes, and Yejin Choi. The curious case of neural text degeneration. In *Proc. of ICLR*, 2020.

Yao-Hung Hubert Tsai, Shaojie Bai, Makoto Yamada, Louis-Philippe Morency, and Ruslan Salakhutdinov. Transformer dissection: An unified understanding for transformer's attention via the lens of kernel. In Kentaro Inui, Jing Jiang, Vincent Ng, and Xiaojun Wan, editors, *Proceedings of the 2019 Conference on Empirical Methods in Natural Language Processing and the 9th International Joint Conference on Natural Language Processing (EMNLP-IJCNLP)*, pages 4344–4353, Hong Kong, China, November 2019. Association for Computational Linguistics. doi: 10.18653/v1/D19-1443. URL https://aclanthology.org/D19-1443/.

Amirhossein Kazemnejad, Inkit Padhi, Karthikeyan Natesan, Payel Das, and Siva Reddy. The impact of positional encoding on length generalization in transformers. In *Thirty-seventh Conference on Neural Information Processing Systems*, 2023. URL https://openreview.net/forum?id=Drrl2gcjzl.