CS336 Assignment 1 (basics): Building a Transformer LM

Version 0.1.3

Spring 2024

1 Assignment Overview

In this assignment, you will build all of 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 the dataset 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`
- `torch.nn.Dropout` and `torch.nn.functional.dropout`
- `torch.nn.Linear` and `torch.nn.Embedding`
- 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 post on Slack.

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

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

`github.com/stanford-cs336/spring2024-assignment1-basics`

Please `git clone` the repository. If there are any updates, we will notify you and 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! So you can do whatever you want from scratch!
2. `adapters.py`: There is a set of functionality that your code must have (e.g., scaled dot product attention). 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 this file.

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/spring2024-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`.

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 15.1 (released in September 2023), the standard defines 149,813 characters across 161 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 149,812) 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.

```
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 string 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:

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

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 used for 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"), and ("B", "A") all have the highest frequency, we'd merge ("B", "A"):

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

[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.

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 a 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 When we pretokenize and count, we end up with the frequency table.

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

It is often convenient to represent this as a `dict[tuple[byte]: int]`, e.g. `{(l,o,w):5 ...}`

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]`.

This naïve implementation of BPE training 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.

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

Deliverable: Write a function that, given 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 non-negative 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 `pytest tests/test_train_bpe.py`. Your implementation should be able to pass all tests.

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.

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: N hours (no GPUs), N GB RAM

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?

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.

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 the 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, we recommend replacing 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__(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 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)` Construct 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 `pytest tests/test_tokenizer.py`. Your implementation should be able to pass all tests.

2.7 Experiments

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

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 tokenizer 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.

3 Transformer language model architecture

A language model takes as input a batched sequence of integer token IDs (i.e., a PyTorch `LongTensor` of shape `(batch_size, sequence_length)`), and returns a (batched) normalized probability distribution over the vocabulary (i.e., a PyTorch `FloatTensor` 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 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.), adding the generated token to the input sequence and repeating.

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 and inject positional information, passes the embedded tokens through `num_layers` Transformer blocks, and then applies a learned linear projection (the "output embedding") to produce the predicted next-token logits. See Figure 1 for a schematic representation.

3.1.1 Token and Positional Embeddings.

In the very first step, the Transformer *embeds* the (batched) sequence of token IDs into a sequence of vectors containing information on both the token identity and position (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. To inject positional information into the model, we use absolute

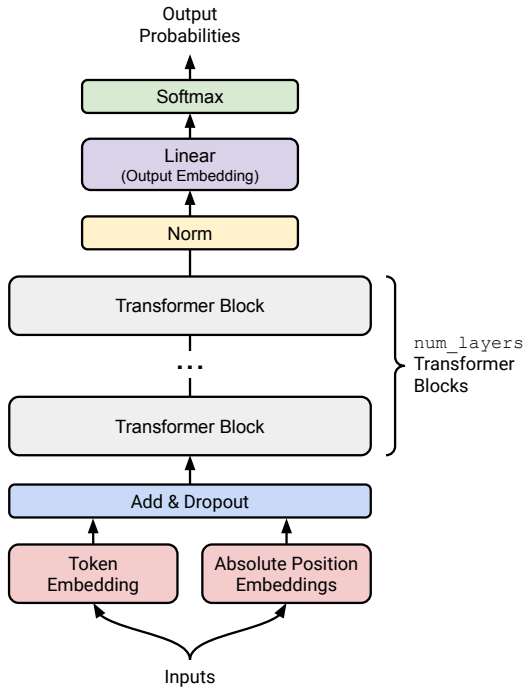

Figure 1: An overview of a Transformer language model.

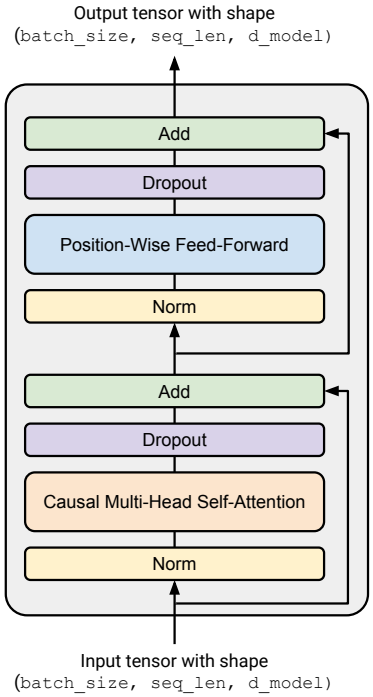

Figure 2: A pre-norm Transformer block.

learned position embeddings and add them to the token embeddings (e.g., as used in GPT-2; Radford et al., 2018, equation 2). For a token i in position j, you would return an embedding vector

$$\text{emb}(\text{position} = j, \text{token} = i) = v_i + u_j,$$

where v and u are parameter matrices of size `vocab_size` $\times$ `d_model` and `context_length` $\times$ `d_model`. The **context length** of a model is the maximum sequence length that a model can process. In our case, since each position in the sequence gets its own parameter, the size of the position embedding matrix determines the maximum allowable sequence length [4].

In addition, we apply dropout to the result of the embedding function (embedding dropout; Vaswani et al., 2017, section 5.4). 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).

[4] Of course, we must also train these parameters, so processing long contexts is not merely a matter of making this matrix big and using more compute.

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 §3.4), 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). Following Vaswani et al. [2017] (section 3.4) and Chowdhery et al. [2022] (section 2), we tie the weights between the input and output embeddings.

3.3 Remark: batching 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 head**: the same key and value parameters are applied to different heads of 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. As a reminder, here is a useful PyTorch fact.

Many PyTorch operations can take in excess "batch-like" dimensions at the start of a tensor and repeat / broadcast the operation across these dimensions automatically and efficiently. As an example, 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 X `(d_model, d_model)`. We can simply write `D @ X`, and this operation would perform a vector-matrix multiply that is repeated across the first two `(batch_size, sequence_length)` dimensions. **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.** Operations such as PyTorch's `view` or `reshape` functions can be helpful in achieving this.

3.4 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]. 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. Dropout is applied to the output of each Transformer block sub-layer, before it is added to the sub-layer input via the residual connection (Vaswani et al., 2017, section 5.4). See Figure 2 for a visual representation of the pre-norm Transformer block. We will walk through each of the components of a pre-norm Transformer block, implementing them in sequence.

3.4.1 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:

$$\text{RMSNorm}(a_i) = \frac{a_i}{\text{RMS}(a)} g_i, \tag{1}$$

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), typically initialized to 1 and ε is a hyperparameter that is often fixed at 1e-5.

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

Deliverable: Implement RMSNorm as a `torch.nn.Module`. To test your implementation against our provided test, you will first need to implement the test adapter at `[adapters.run_rmsnorm]`. Then, run `pytest -k test_rmsnorm` to test your implementation.

3.4.2 Position-Wise Feed-Forward Network

As originally described in section 3.3 of Vaswani et al. [2017], the Transformer feed-forward network consists of two linear transformations with a ReLU activate between them. The dimensionality of the inner feed-forward layer is typically 4x the input dimensionality.

Following the GPT and GPT-2 architecture Radford et al. [2018], we will use the GELU activation function [Hendrycks and Gimpel, 2016] instead of the ReLU activation function:

$$\text{GELU}(x) = x \cdot \frac{1}{2}\left[1 + \text{erf}(x/\sqrt{2})\right] \tag{2}$$

In addition, following recent models like PaLM [Chowdhery et al., 2022] and LLaMA [Touvron et al., 2023], we omit the biases in the feed-forward network linear transformations. Thus, our feed-forward network is defined as follows:

$$\text{FFN}(x) = \text{GELU}(xW_1)W_2, \tag{3}$$

where $x \in \mathbb{R}^{d_{\text{model}}}$, $W_1 \in \mathbb{R}^{d_{\text{model}} \times d_{\text{ff}}}$, and $W_2 \in \mathbb{R}^{d_{\text{ff}} \times d_{\text{model}}}$, with $d_{\text{ff}} = 4d_{\text{model}}$.

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

(a) **Deliverable**: Implement the GELU activation function. To test your implementation against our provided tests, you will need to implement the test adapter at `[adapters.run_gelu]`. Then, run `pytest -k test_gelu` to test your implementation.

(b) **Deliverable**: Implement the position-wise feed-forward network. To test your implementation, implement the test adapter at `[adapters.run_positionwise_feedforward]`. Then, run `pytest -k test_positionwise_feedforward` to test your implementation.

3.4.3 Scaled Dot-Product Attention

We now implement scaled dot-product attention as described in Vaswani et al. [2017] (section 3.2.1). Mathematically, define the Attention operation as follows:

$$\text{Attention}(Q, K, V) = \text{softmax}\left(\frac{QK^\top}{\sqrt{d_k}}\right)V \tag{4}$$

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. In addition, apply attention dropout (mentioned in section 6.3 of Vaswani et al., 2017) to the softmax-normalized attention scores.

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 of this boolean matrix indicates which keys are 'valid' targets to attend to. For example, consider a 1×3 mask matrix, and the first row has entries [`False`, `False`, `True`]. Running attention with this mask should have the following property: running attention with a mask is equivalent to running attention where we omit the masked positions.

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 ($\frac{QK^\top}{\sqrt{d_k}}$) and adding a $-\infty$ in any entry of the mask matrix that is True.

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 the masked positions should be zero, and the relative probabilities on the non-masked positions should remain the same.

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

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

3.4.4 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:

$$\text{MultiHead}(Q, K, V) = \text{Concat}(\text{head}_1, \ldots, \text{head}_h)W^O \tag{5}$$

where $\text{head}_i = \text{Attention}(QW_i^Q, KW_i^K, VW_i^V)$, with Attention being the scaled dot-product attention operation defined in §3.4.3. Here, the learnable parameters are $W_i^Q \in \mathbb{R}^{d_{\text{model}} \times d_k}$ (the query projection of the ith head), $W_i^K \in \mathbb{R}^{d_{\text{model}} \times d_k}$ (the key projection of the ith head), $W_i^V \in \mathbb{R}^{d_{\text{model}} \times d_v}$ (the value projection of the ith head), and $W^O \in \mathbb{R}^{hd_v \times d_{\text{model}}}$. Given this definition of multi-head attention, the multi-head *self*-attention operation is defined as $\text{MultiHeadSelfAttention}(x) = \text{MultiHead}(x, x, x)$.

Rather than iterating over each head and explicitly applying the key, query, and value projections ($h \times 3$ matrix multiplies in total), we recommend combining all the projection operations across the heads into a single matrix. tThis will require you to think carefully about batch-like dimensions and PyTorch shapes (See

Section 3.3). When you have this working, you should be computing the key, value, and query projections in a total of three matrix multiplies.[5]

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 $t_1, \ldots, t_n$ and we want to calculate the next-word predictions for the prefix $t_1, \ldots, t_i$ (where $i < n$), the model should *not* be able to access (attend to) the token representations at positions $t_{i+1}, \ldots, t_n$ 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 $t_1, \ldots, t_n$ 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 eliminates the contribution of token j for the output of self-attention layers at token i when $j > i$. You may find `torch.triu` to be useful for constructing the mask, and you should take advantage of the fact that your scaled dot-product attention implementation from §3.4.3 already supports attention masking.

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.

`attn_pdrop: float | None = None` Dropout rate for softmax-normalized attention probabilities.

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 `pytest -k test_multihead_self_attention` to test your implementation.

3.5 The Full Transformer LM

Let's begin by assembling the Transformer block. Recall that our diagram of the Transformer block in Figure 2. A Transformer block contains two 'sublayers', one for the multihead self attention, and another for the feed-forward network. In each block, we first perform RMSNorm, then the main operation (MHA/FF), then dropout, finally adding in the residual connection.

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

$$y = x + \text{Dropout}(\text{MultiHeadSelfAttention}(\text{RMSNorm}(x))). \tag{6}$$

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

Implement the pre-norm Transformer block as described in §3.4 and illustrated in Figure 2. Your Transformer block should accept (at least) the following parameters.

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

[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.

`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.

`attn_pdrop: float | None = None` Dropout rate for softmax-normalized attention probabilities.

`residual_pdrop: float | None = None` Dropout rate for embeddings and Transformer block sub-layer outputs.

To test your implementation, implement the adapter `[adapters.run_transformer_block]`. Then 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 of 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 `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

`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.

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.

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

Deliverable: A one-to-two sentence response.

(c) 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.

(d) 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.

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 of the supporting infrastructure that loads data, saves checkpoints, and manages training.

4.1 Cross-entropy loss

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:

$$\ell(\theta; D) = \frac{1}{|D|m} \sum_{x \in D} \sum_{i=1}^{m} -\log p_\theta(x_{i+1} \mid x_{1:i}). \tag{7}$$

(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]

$$p(x_{i+1} \mid x_{1:i}) = \text{softmax}(o_i)[x_{i+1}] = \frac{\exp(o_i[x_{i+1}])}{\sum_{a=1}^{\texttt{vocab\_size}} \exp(o_i[a])}. \tag{8}$$

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. Note that $\exp(o_{i,j})$ can become `inf` for large values (then, `inf/inf` = `NaN`). We can avoid this by noticing that the *softmax* operation (the $\exp(\cdot)/\sum \exp(\cdot)$ operation inside the log) 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 the cross-entropy loss, using this trick for numerical stability.

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 $-\log \text{softmax}(o_i)[x_{i+1}]$. Your function should handle the ollowing:

- 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 `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 $\ell_1, \ldots, \ell_m$:

$$\text{perplexity} = \exp\left(\frac{1}{m} \sum_{i=1}^{m} \ell_i\right). \tag{9}$$

[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 $\text{softmax}(o_i)$ 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:

$$\theta_{t+1} \leftarrow \theta_t - \alpha \nabla L(\theta_t; B_t), \tag{10}$$

where B_t is a random batch of data sampled from the dataset D.

where 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 our initial learning rate and taking successively smaller steps over time:

$$\theta_{t+1} = \theta_t - \frac{\alpha}{\sqrt{t+1}} \nabla L(\theta_t; B_t) \tag{11}$$

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

```
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.
```

```
            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
```

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 11. 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 numeric 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)
$t \leftarrow 0$ (Initial time step)
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

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 `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 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 model hyperparameters (`num_layers`, `d_model`, `context_length`). Assume `d_ff` $= 4 \times$ `d_model`.

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

(b) Instantiate your answer for GPT-2 XL 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 $a \cdot$ `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 Trasnformers, 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 $T_w \leq t \leq T_c$, 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 $\alpha_t = \alpha_{\min}$.

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 `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 a gradient 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.

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

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 `pytest -k test_gradient_clipping`.

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 `pytest -k test_get_batch` to test your implementation.

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`, which will return a numpy array-like object, where the entries are loaded on demand as you access them. **When sampling from your dataset (i.e., a numpy array) during training, be sure load the dataset with `np.memmap`.**

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:

`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]
```

`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 `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

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` $\times$ `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,

$$\text{softmax}(v)_i = \frac{\exp(v_i)}{\sum_{j=1}^{|\text{vocab\_size}|} \exp(v_j)}. \tag{12}$$

We have already seen this in the computation of cross-entropies, and the same ideas and tricks apply here.

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 same trick as your cross-entropy loss calculation to avoid numerical stability issues.

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

Decoding. To generate text (decode) from our model, we will provide the model with 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 x_{t+1} via the

following equation,

$$P(x_{t+1} = i \mid x_{1\ldots t}) = \frac{\exp(v_i)}{\sum_j \exp(v_j)}$$
$$v = \text{TransformerLM}(x_{1\ldots t})_t$$

where TransformerLM is our model which takes as input a sequence of `sequence_length` and produces a matrix of size (`sequence_length` $\times$ `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

$$\text{softmax}(v, \tau)_i = \frac{\exp(v_i/\tau)}{\sum_{j=1}^{|\text{vocab\_size}|} \exp(v_j/\tau)}. \tag{13}$$

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

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

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 a `<|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.

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 and simple dataset – TinyStories and a 17M parameter model. 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` 2048. Many models use the rule of thumb of 4`d_model`.

`number of layers and heads` 4 layers, 16 heads. Together, this will give 17.70M non-embedding parameters which is a fairly small Transformer.

`total tokens processed` 12,800,000 (your batch size $\times$ total step count should add up to roughly this value).

You should do some trial and error to find good defaults for the following other hyperparameters: `dropout`, `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 20-minute runtime on 1 H100 GPU. If you have runtimes that are much longer, please check and make sure your checkpointing / validation loss code is not bottlenecking your runtimes and that your implementation is properly batched.

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

(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.

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

Problem (`generate`): Generate text (1 point)

Use your decoder and your trained checkpoint, and see the text that comes out from 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.

Architecture Modification 1: Parallel Layers We will now make a simple change to the architecture and see how this architecture works. **Parallel layers** [Wang and Komatsuzaki, 2021, Chowdhery et al., 2022] are used in several modern Transformer variants as a way to trade off expressive power for speed.

In equations, we can view the Transformer block as performing the following serial computation (ignoring dropout terms) that takes input x and produces output y,

$$z = x + \text{MultiHeadedSelfAttention}(\text{RMSNorm}(x))$$
$$y = z + \text{FFN}(\text{RMSNorm}(z)).$$

This is serial and the computation of y must wait on the computation of z. We might be able to parallelize this by instead computing

$$y = x + \text{FFN}(\text{RMSNorm}(x)) + \text{MultiHeadedSelfAttention}(\text{RMSNorm}(x)). \tag{14}$$

Now instead of composing the FFN and the MultiHeadedSelfAttention, we simply add the two together. This will let us parallelize the operations, at the potential cost of expressive power.

Problem (`parallel_layers`): Experiment with parallel layers (1 point) (1 H100 hr)

Implement parallel layers, as defined in Eq 14. Train a model with parallel layers, logging your training curves.

Deliverable: Two training curves associated with your parallel layer model. One where the x-axis is the number of gradient updates and y-axis is validation perplexity. Another is where the x-axis is wallclock time.

Deliverable: Discuss any differences between the two training curves in a few sentences. Are the

results surprising?

Ablation: 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

$$z = x + \text{MultiHeadedSelfAttention}(\text{RMSNorm}(x))$$
$$y = z + \text{FFN}(\text{RMSNorm}(z)).$$

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

$$z = \text{RMSNorm}(x + \text{MultiHeadedSelfAttention}(x))$$
$$y = \text{RMSNorm}(z + \text{FFN}(z)).$$

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.

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 model architecture and total training iterations. 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 stacks 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 try turning your Transformer into the LLaMA architecture. To do this, you can implement RoPE embeddings [Su et al., 2021] and SwiGLU activations [Shazeer, 2020]. You may also want to try tying the weights of the input and output embeddings together, as was done in the original Transformer paper and PaLM. You will want to test these on either a small subset of OpenwebText or on TinyStories before trying the full 1.5-hour run.

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 one 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 leaderboard submission to beat at least the naive baseline of a 5.0 loss.

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.

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.

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.

Dan Hendrycks and Kevin Gimpel. Bridging nonlinearities and stochastic regularizers with gaussian error linear units, 2016. arXiv:1606.08415.

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.

Ben Wang and Aran Komatsuzaki. GPT-J-6B: A 6 Billion Parameter Autoregressive Language Model. `https://github.com/kingoflolz/mesh-transformer-jax`, May 2021.

Jianlin Su, Yu Lu, Shengfeng Pan, Bo Wen, and Yunfeng Liu. Roformer: Enhanced transformer with rotary position embedding, 2021.

Noam Shazeer. GLU variants improve transformer, 2020. arXiv:2002.05202.