Stanford CS336 1. Basics

This note references the following lectures and notes:

- Stanford CS336 Lecture 1: Overview and Tokenization
- Stanford CS336 Lecture 2: Pytorch, Resource Accounting
- Stanford CS336 Lecture 3: Architectures, Hyperparameters
- weiruihhh/cs336 note & hw
- 李理的博客
- 周鑫的博客

Corrections and constructive criticism are welcome!

1.1 Byte-Pair Encoding (BPE) Tokenizer

1.1.1 Unicode Standard

Before discussing tokenization algorithms, we must clarify how raw text is encoded.
Unicode is a text encoding standard that maps characters to integer code points.
In Python, the `ord()` function converts a Unicode character into its integer representation.
The `chr()` function converts an integer code point into its corresponding character.

```
>>> ord('雍')
38605
>>> chr(38605)
'雍'
```

Problem (`unicode1`): Understanding Unicode

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

  Solution: `chr(0)` returns the NULL character.

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

  Solution: Its string representation shows an escape sequence (`'\x00'`), while printing it produces no visible output.

  ```
  >>> chr(0)
  '\x00'
  >>> print(chr(0))
  ```

- (c) What happens when this character occurs in text?

  Solution: When this character occurs in text, it is preserved in the string and acts as a null control character, while printing appears to skip over it without displaying anything.

```
>>> "This is a test" + chr(0) + "string."
'This is a test\x00string.'
>>> print("This is a test" + chr(0) + "string.")
This is a teststring.
```

1.1.2 UTF-8 Encoding

It is impractical to train tokenizers directly on Unicode code points, since the vocabulary would be prohibitively large (around 150K items) and sparse (since many characters are quite rare); instead, we use **UTF-8**, a variable-length Unicode encoding that represents each Unicode character as a sequence of **1–4 bytes**.

Bits	Byte 1	Byte 2	Byte 3	Byte 4	Range
7	0xxx_xxxx				$\text{U+0000}\ (0) \sim \text{U+007F}\ (127)$
11	110x_xxxx	10xx_xxxx			$\text{U+0080}\ (128) \sim \text{U+07FF}\ (2,047)$
16	1110_xxxx	10xx_xxxx	10xx_xxxx		$\text{U+0800}\ (2,048) \sim \text{U+FFFF}\ (65,535)$
21	1111_0xxx	10xx_xxxx	10xx_xxxx	10xx_xxxx	$\text{U+10000}\ (65,536) \sim \text{U+10FFFF}\ (1,114,111)$

If a byte starts with `0` (i.e., `110xxxxx`), it represents a single-byte code point.
Otherwise, it is part of a multi-byte code point.
If a byte starts with `10` (i.e., `10xxxxxx`), it is a **continuation byte**, not a **leading byte**.
Otherwise, the number of leading `1`s in the leading byte indicates the **total number of bytes**.

- `110xxxxx` $\rightarrow 2$ bytes total (1 continuation byte)
- `1110xxxx` $\rightarrow 3$ bytes total (2 continuation byte)
- `11110xxx` $\rightarrow 4$ bytes total (3 continuation byte)

In Python, the `encode()` function encodes a Unicode string into a UTF-8 byte string.
The `decode()` function decodes a UTF-8 byte string into a Unicode string.

```
>>> test_string = "孩子们, Carpe Diem."
>>> utf8_encoding = test_string.encode("utf-8")
>>> list(utf8_encoding)
[229, 173, 169, 229, 173, 144, 228, 187, 172, 44, 32, 67, 97, 114, 112, 101, 32, 68, 105,
101, 109, 46]
>>> print(utf8_encoding)
b'\xe5\xad\xa9\xe5\xad\x90\xe4\xbb\xac, Carpe Diem.'
>>> print(utf8_encoding.decode("utf-8"))
孩子们, Carpe Diem.
```

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

- (a) Why UTF-8 is better than UTF-16 or UTF-32?

  Solution: UTF-8 is compatible with ASCII and more storage-efficient than UTF-16 and UTF-32.

- (b) Why is this function incorrect?

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

  Solution: The fuction decodes each byte independently, but UTF-8 is a variable-length encoding in which multi-byte characters such as `é` must be decoded jointly.

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

  Solution: A valid UTF-8 two-byte sequence must have the form `110x xxxx 10xx xxxx`; however, byte sequences in the range `1100 0000 1000 0000` to `1100 0001 1011 1111` are invalid because they represent overlong encodings.

  ```
  >>> utf8_encoding = b'\xc0\x00'
  >>> utf8_encoding.decode("utf-8")
  Traceback (most recent call last):
    File "<stdin>", line 1, in <module>
  UnicodeDecodeError: 'utf-8' codec can't decode byte 0xc0 in position 0: invalid start
  byte
  ```

1.1.3 Subword Tokenization

Byte-level tokenization results in extremely long input sequences, thereby slowing down model training.
Processing these longer sequences requires more computation at each step of the model.
Furthermore, longer input sequences create long-term dependencies in the data.
Subword tokenization is a midpoint between word-level tokenizers and byte-level tokenizers.
A subword tokenizer trades-off a larger vocabulary size for better compression of the input byte sequence.

In this project, we use **byte-pair encoding** (BPE) to select subword units.
BPE iteratively merges the most frequent pair of bytes with a single, new unused index.
As a result, common sequences are represented as single tokens,
while rare sequences are encoded as combinations of multiple tokens.

The BPE tokenizer training procedure consists of three main steps.

- **① Initialization**
  Build the initial vocabulary with the set of all bytes.
  Since there are 256 possible byte values, our initial vocabulary is of size 256.
- **② Pre-tokenization**
  Count how often bytes occur next to each other and merge the most frequent pair of bytes.
  However, it is computationally expensive taking a full pass over the corpus each time we merge.
  Moreover, directly merging may result in tokens that differ only in punctuation (e.g., `me!` vs. `me.`), which receive different token IDs despite their high semantic similarity.

  To mitigate these issues, we apply **pre-tokenization**,
  a coarse-grained tokenization step that efficiently count how often character pairs occur.
  For example, if the word `MAGA` appears as a single pre-token 10 times,
  then the adjacent characters `M` and `G` contribute 10 counts at once,
  rather than requiring repeated scans through the corpus.

The original BPE implementation pre-tokenizes by splitting on whitespace.
In this project, we instead use a **regex-based pre-tokenizer** defined as:

```
>>> PAT = r"""'(?:[sdmt]|ll|ve|re)| ?\p{L}+| ?\p{N}+| ?[^\s\p{L}\p{N}]+|\s+
(?!\S)|\s+"""
>>> import regex as re
>>> re.findall(PAT, "I'll pre-tokenize some text about the kill line in American
society.")
['I', "'ll", ' pre', '-', 'tokenize', ' some', ' text', ' about', ' the', ' kill', '
line', ' in', ' American', ' society', '.']
```

It matches (in order):
common English contractions (e.g., `'s`, `'ll`), sequences of letters, sequences of digits,
runs of non-alphanumeric symbols (punctuation), and different kinds of whitespace,
optionally including a leading space to preserve word boundaries.
We can use `re.finditer` to count pre-tokens streamingly without storing all pre-tokens in memory.
As a result, input text is segmented into pre-tokens, each represented as a sequence of UTF-8 bytes.

- ③ **Merge**
  Iteratively counts all byte pairs and identifies the pair with the highest frequency.
  Every occurrence of this most frequent pair (e.g. `A` & `B`) is then merged into a new token (e.g. `AB`).
  For efficiency during BPE training, pairs that cross pre-token boundaries are ignored.
  When multiple symbol pairs share the same highest frequency,
  ties are broken deterministically by selecting the lexicographically greatest pair.

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

- ④ **Special tokens**
  Some strings (e.g., `<|endoftext|>`) are used to encode metadata (e.g., boundaries between documents).
  These strings are treated as "special tokens" that should never be split into multiple tokens.
  These special tokens must be added to the vocabulary, so they have a corresponding fixed token ID.

Example (`bpe_example`): BPE training example

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

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

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

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

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

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

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

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

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

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

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

1.1.4 BPE Tokenizer Training

This assignment will use two pre-processed datasets: TinyStories and OpenWebText.
Optimization tips:

- **① Parallelizing pre-tokenization**
  Speed up pre-tokenization by using Python's built-in `multiprocessing` library.
  Specifically, split the corpus into chunks such that boundaries align with the start of a pre-token.
  For this assignment, it is sufficient to split only at `<|endoftext|>` boundaries.
  (Example: assignment1-basics/cs336_basics/pretokenization_example.py)
  Then, distribute the chunks across processes to parallelize pre-tokenization.
- **② Removing special tokens**
  Remove special tokens before applying the pre-tokenization regex,
  ensuring no merges occur across special-token boundaries.
- **③ Optimizing the merging step**
  Naive BPE implementation is slow because each merge scans all byte pairs to find the most frequent one.
  To speed up training, we only change pair counts that overlap with merged pair after each merge.
- **④ Using profiling tools**
  Use profiling tools such as `cProfile` to identify bottlenecks in the implementation.
  Focus optimization efforts accordingly.

- ⑤ **Downscaling**
  First use the Tiny Stories validation set as a "debug dataset".

Problem (`train_bpe`)
Write a function that trains a byte-level BPE tokenizer given the path to an input text file.
The function should support at least the following parameters:

- `input_path: str`
  Path to a text file containing BPE training data.
- `vocab_size: int`
  Maximum size of the final vocabulary.
- `special_tokens: list[str]`
  Special tokens to add to the vocabulary.

The function should return:

- `vocab: dict[int, bytes]`
  A mapping from token ID to token bytes.
- `merges: list[tuple[bytes, bytes]]`
  An ordered list of BPE merges.
  Each entry is a tuple `(token1, token2)`, indicating that `token1` was merged with `token2`.

To test your implementation against the provided tests:

- Implement the test adapter at `adapters.run_train_bpe`.
- Run `uv run pytest tests/test_train_bpe.py`.

Futhermore, we can implement performance-critical parts of training in $\mathrm{C{+}{+}}$ via `cppyy`.
Be mindful of operations that copy data from Python memory.
Provide build instructions, or ensure the project builds using only `pyproject.toml`.

Note that the GPT-2 regex is not well supported or performant in most regex engines.
The Python `regex` package is fast and recommended.

Solution:
The core implementation lives in cs336_basics/BPE_Tokenizer/train_bpe.py.
The BPE training procedure follows these steps:

- ① Initialize the vocabulary with all single-byte tokens
  and append any user-specified special tokens.
- ② Pretokenize the input text using a GPT-2–style regex
  and count word frequencies in parallel using multiprocessing.
- ③ Encode each unique word as a sequence of byte-level token IDs.
- ④ Count all adjacent token pairs across the corpus, while maintaining a mapping
  from each pair to the words in which it appears (with per-word occurrence counts).
- ⑤ Build a max-heap over token pairs keyed by frequency,
  enabling efficient selection of the most frequent pair at each merge step.
- ⑥ Repeatedly merge the most frequent token pair, updating:

    - encodings of affected words,
    - pair counts,
    - pair-to-word mappings, and
    - the vocabulary,

  until the target vocabulary size is reached.

Run the standard test suite by `uv run pytest tests/test_train_bpe.py`:

```
========================= test session starts ==========================
platform linux -- Python 3.11.13, pytest-8.4.1, pluggy-1.6.0
rootdir: /root/Share/Stanford-CS336-spring25/assignment1-basics
configfile: pyproject.toml
plugins: jaxtyping-0.3.2
collected 3 items


tests/test_train_bpe.py::test_train_bpe_speed train_bpe: args =
Namespace(max_num_counters=64)
_pretokenize_and_count_words: 0.10451829433441162 sec for 4787 unique words
_count_pairs: 0.02 sec for 1032 unique pairs
_build_heap: 0.00 sec for 1032 heap size
_merge: 0.13969143945723772 sec for 243 merges

PASSED
tests/test_train_bpe.py::test_train_bpe train_bpe: args = Namespace(max_num_counters=64)
_pretokenize_and_count_words: 0.11133486218750477 sec for 4787 unique words
_count_pairs: 0.02 sec for 1032 unique pairs
_build_heap: 0.00 sec for 1032 heap size
_merge: 0.1400450374931097 sec for 243 merges

PASSED
tests/test_train_bpe.py::test_train_bpe_special_tokens train_bpe: args =
Namespace(max_num_counters=64)
_pretokenize_and_count_words: 0.4600677276030183 sec for 8126 unique words
_count_pairs: 0.04 sec for 796 unique pairs
_build_heap: 0.00 sec for 796 heap size
_merge: 0.23012215178459883 sec for 743 merges

PASSED

========================= 3 passed in 1.31s ==========================
```

Problem (`train_bpe_tinystories`): BPE Training on TinyStories

Train a 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.
Resource requirements: ≤ 30 minutes (no GPUs), ≤ 30 GB RAM.

Run extended test by `uv run pytest cs336_basics/BPE_Tokenizer/tests/test_01_train_bpe.py`:

```
_pretokenize_and_count_words: 14.782620964571834 sec for 59941 unique words
_count_pairs: 0.44 sec for 2080 unique pairs
_build_heap: 0.01 sec for 2080 heap size
_merge: 4.146655489690602 sec for 9743 merges

Longest token:
    -id       = 7160
    -bytes    = b' accomplishment'
    -length   = 15
    -utf8_str =  accomplishment
RSS memory (resident): 0.11 GB
VMS memory (virtual):  2.38 GB

========== cProfile Results (TinyStoriesV2-GPT4-train.txt) ==========
         3129278 function calls in 19.411 seconds

   Ordered by: cumulative time
   List reduced from 207 to 25 due to restriction <25>

   ncalls  tottime  percall  cumtime  percall filename:lineno(function)
        1    0.050    0.050   19.411   19.411 train_bpe.py:63(train_bpe)
        1    0.029    0.029   14.780   14.780 train_bpe.py:168(_pretokenize_and_count_mp)
       65    0.001    0.000   13.775    0.212 pool.py:853(next)
       67    0.001    0.000   13.774    0.206 threading.py:295(wait)
      271   13.773    0.051   13.773    0.051 {method 'acquire' of '_thread.lock' objects}
     9743    0.114    0.000    4.135    0.000 train_bpe.py:354(_merge_a_pair)
     9743    2.959    0.000    3.779    0.000
train_bpe.py:441(_update_pair_count_of_affected_words)
    69421    0.561    0.000    0.836    0.000 __init__.py:660(update)
        1    0.405    0.405    0.439    0.439 train_bpe.py:300(_count_pairs)
   173286    0.286    0.000    0.427    0.000 max_heapq.py:23(heappush_max)
   958608    0.274    0.000    0.274    0.000 {method 'get' of 'dict' objects}
   277707    0.256    0.000    0.256    0.000
train_bpe.py:521(_merge_pair_and_count_pair_difference)
    69357    0.179    0.000    0.190    0.000 __init__.py:587(__init__)
    74329    0.023    0.000    0.152    0.000 max_heapq.py:32(heappop_max)
    74329    0.129    0.000    0.129    0.000 {built-in method _heapq._heappop_max}
   173286    0.112    0.000    0.112    0.000 heapq.py:280(_siftdown_max)
        1    0.000    0.000    0.072    0.072 context.py:115(Pool)
        1    0.000    0.000    0.072    0.072 pool.py:183(__init__)
        1    0.000    0.000    0.071    0.071 pool.py:305(_repopulate_pool)
        1    0.002    0.002    0.071    0.071 pool.py:314(_repopulate_pool_static)
       64    0.002    0.000    0.064    0.001 process.py:110(start)
   748760    0.062    0.000    0.062    0.000 __init__.py:601(__missing__)
       65    0.000    0.000    0.062    0.001 util.py:208(__call__)
        1    0.000    0.000    0.061    0.061 pool.py:738(__exit__)
        1    0.000    0.000    0.061    0.061 pool.py:654(terminate)
```

- (a) What is the longest token in the vocabulary? Does it make sense?

  Solution:
  The longest token is `" accomplishment"` (15 bytes, including a leading space), which makes sense because BPE often learns frequent words as single tokens.

- (b) What part of the tokenizer training process takes the most time?

  Solution:
  The pre-tokenization and word counting step dominates runtime,
  largely due to multiprocessing overhead and thread synchronization.

Problem (`train_bpe_expts_owt`): BPE Training on OpenWebText
Train a 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.
Resource requirements: ≤ 12 hours (no GPUs), ≤ 100 GB RAM.

Run extended test by `uv run pytest cs336_basics/BPE_Tokenizer/tests/test_01_train_bpe.py`:

```
_pretokenize_and_count_words: 93.87730549834669 sec for 6424740 unique words
_count_pairs: 78.31 sec for 16219 unique pairs
_build_heap: 0.03 sec for 16219 heap size
_merge: 918.4185373773798 sec for 31743 merges

Longest token:
    -id       = 25806
    -bytes    =
b'\xc3\x83\xc3\x82\xc3\x83\xc3\x82\xc3\x83\xc3\x82\xc3\x83\xc3\x82\xc3\x83\xc3\x82\xc3\x83\x
c3\x82\xc3\x83\xc3\x82\xc3\x83\xc3\x82\xc3\x83\xc3\x82\xc3\x83\xc3\x82\xc3\x83\xc3\x82\xc3\x
83\xc3\x82\xc3\x83\xc3\x82\xc3\x83\xc3\x82\xc3\x83\xc3\x82\xc3\x83\xc3\x82'
    -length   = 64
    -utf8_str = ÃÂÃÂÃÂÃÂÃÂÃÂÃÂÃÂÃÂÃÂÃÂÃÂÃÂÃÂÃÂÃÂ
RSS memory (resident): 5.62 GB
VMS memory (virtual):  7.93 GB

========== cProfile Results (owt_train.txt) ==========
         220924824 function calls in 1102.543 seconds

   Ordered by: cumulative time
   List reduced from 207 to 25 due to restriction <25>

   ncalls  tottime  percall  cumtime  percall filename:lineno(function)
        1   13.247   13.247 1102.543 1102.543 train_bpe.py:63(train_bpe)
    31743    3.645    0.000  917.100    0.029 train_bpe.py:354(_merge_a_pair)
    31743  777.103    0.024  904.996    0.029
train_bpe.py:441(_update_pair_count_of_affected_words)
        1    0.652    0.652   93.867   93.867 train_bpe.py:168(_pretokenize_and_count_mp)
        1   73.874   73.874   78.307   78.307 train_bpe.py:300(_count_pairs)
       65    0.002    0.000   65.045    1.001 pool.py:853(next)
       67    0.001    0.000   65.043    0.971 threading.py:295(wait)
      271   65.041    0.240   65.041    0.240 {method 'acquire' of '_thread.lock' objects}
 35172372   58.831    0.000   58.831    0.000
train_bpe.py:521(_merge_pair_and_count_pair_difference)
  7482501   42.360    0.000   48.832    0.000 max_heapq.py:23(heappush_max)
  3530131   13.286    0.000   28.126    0.000 __init__.py:660(update)
  3530067   16.105    0.000   16.625    0.000 __init__.py:587(__init__)
 29896522   14.822    0.000   14.822    0.000 {method 'get' of 'dict' objects}
109230858    8.905    0.000    8.905    0.000 __init__.py:601(__missing__)
```

```
   1102306    0.325    0.000    6.975    0.000 max_heapq.py:32(heappop_max)
   1102306    6.650    0.000    6.650    0.000 {built-in method _heapq._heappop_max}
   7482501    5.306    0.000    5.306    0.000 heapq.py:280(_siftdown_max)
  13907264    1.044    0.000    1.044    0.000 {built-in method builtins.len}
   7514323    0.552    0.000    0.552    0.000 {method 'append' of 'list' objects}
        65    0.000    0.000    0.248    0.004 util.py:208(__call__)
         1    0.000    0.000    0.248    0.248 pool.py:738(__exit__)
         1    0.000    0.000    0.248    0.248 pool.py:654(terminate)
         1    0.000    0.000    0.247    0.247 pool.py:680(_terminate_pool)
         1    0.220    0.220    0.220    0.220 {built-in method gc.collect}
      2151    0.001    0.000    0.206    0.000 popen_fork.py:24(poll)
```

- (a) What is the longest token in the vocabulary? Does it make sense?

  Solution:
  The longest token is a 64-byte sequence decoding to `"ÃÂÃÂÃÂÃÂÃÂÃÂÃÂÃÂÃÂÃÂÃÂÃÂÃÂÃÂÃÂÃÂ"`,
  which make sense for byte-level BPE on noisy web data.

- (b) Compare the tokenizer trained on TinyStories versus OpenWebText.

  Solution:
  Compared to TinyStories, the OpenWebText tokenizer takes longer to train,
  uses significantly more memory, and includes longer, less interpretable tokens.

1.1.5 Encoding & Decoding

After implementing a BPE tokenizer trainer,
we will implement a BPE tokenizer that loads a provided vocabulary and list of merges,
and use them to encode and decode text to/from token IDs.

The process of encoding text by BPE mirrors how we train the BPE vocabulary:

- ① **Pre-tokenization**
  We first pre-tokenize the sequence and represent each pre-token as a sequence of UTF-8 bytes.
  We will be merging these bytes within each pre-token into vocabulary elements,
  handling each pre-token independently (no merges across pre-token boundaries).
- ② **Apply the merges**
  Then we merge these bytes within each pre-token into vocabulary elements,
  handling each pre-token independently (no merges across pre-token boundaries).

Note that our tokenizer should be able to properly handle user-defined special tokens.
To efficiently tokenize this large file that we cannot fit in memory,
we need to break it up into manageable chunks and process each chunk in turn.

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,
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 should replace the malformed bytes with the official Unicode replacement character $\mathrm{U{+}FFFD}$ (�),

using the errors argument `errors = 'replace'` of `bytes.decode()`.

Example (`bpe_encoding`): BPE encoding example

For example, suppose our input string is `'the cat ate'`, our vocabulary is `{0: b' ', 1: b'a', 2: b'c', 3: b'e', 4: b'h', 5: b't', 6: b'th', 7: b' c', 8: b' a', 9: b'the', 10: b' at'}`, and our learned merges are `[(b't', b'h'), (b' ', b'c'), (b' ', 'a'), (b'th', b'e'), (b' a', b't')]`. First, our pre-tokenizer would split this string into `['the', ' cat', ' ate']`. Then, we'll look at each pre-token and apply the BPE merges.
The first pre-token `'the'` is initially represented as `[b't', b'h', b'e']`. Looking at our list of merges, we identify the first applicable merge to be `(b't', b'h')`, and use that to transform the pre-token into `[b'th', b'e']`. Then, we go back to the list of merges and identify the next applicable merge to be `(b'th', b'e')`, which transforms the pre-token into `[b'the']`. Finally, looking back at the list of merges, we see that there are no more that apply to the string (since the entire pre-token has been merged into a single token), so we are done applying the BPE merges. The corresponding integer sequence is `[9]`.
Repeating this process for the remaining pre-tokens, we see that the pre-token `' cat'` is represented as `[b' c', b'a', b't']` after applying the BPE merges, which becomes the integer sequence `[7, 1, 5]`. The final pre-token `' ate'` is `[b' at', b'e']` after applying the BPE merges, which becomes the integer sequence `[10, 3]`. Thus, the final result of encoding our input string is `[9, 7, 1, 5, 10, 3]`.

Problem (`tokenizer`)

Implement a Tokenizer class that, given a vocabulary, a list of merges and special tokens, encodes text into integer IDs and decodes integer IDs into text.
We recommend the following interface:

- `def __init__(self, vocab, merges, special_tokens)`:
  Construct a tokenizer from a given vocabulary, list of merges and special tokens.
- `def from_files(cls, vocab_path, merges_path, special_tokens)`:
  Class method that constructs and return a Tokenizer.
- `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.
- `def decode(self, ids: list[int]) -> str`
  Decode a sequence of token IDs into text.

Solution:

Run the standard test suite by `uv run pytest tests/test_tokenizer.py`:

```
========================= test session starts ==========================
platform linux -- Python 3.11.13, pytest-8.4.1, pluggy-1.6.0
rootdir: /root/Share/Stanford-CS336-spring25/assignment1-basics
configfile: pyproject.toml
plugins: jaxtyping-0.3.2
collected 25 items


tests/test_tokenizer.py::test_roundtrip_empty PASSED
tests/test_tokenizer.py::test_empty_matches_tiktoken PASSED
tests/test_tokenizer.py::test_roundtrip_single_character PASSED
tests/test_tokenizer.py::test_single_character_matches_tiktoken PASSED
```

```
tests/test_tokenizer.py::test_roundtrip_single_unicode_character PASSED
tests/test_tokenizer.py::test_single_unicode_character_matches_tiktoken PASSED
tests/test_tokenizer.py::test_roundtrip_ascii_string PASSED
tests/test_tokenizer.py::test_ascii_string_matches_tiktoken PASSED
tests/test_tokenizer.py::test_roundtrip_unicode_string PASSED
tests/test_tokenizer.py::test_unicode_string_matches_tiktoken PASSED
tests/test_tokenizer.py::test_roundtrip_unicode_string_with_special_tokens PASSED
tests/test_tokenizer.py::test_unicode_string_with_special_tokens_matches_tiktoken PASSED
tests/test_tokenizer.py::test_overlapping_special_tokens PASSED
tests/test_tokenizer.py::test_address_roundtrip PASSED
tests/test_tokenizer.py::test_address_matches_tiktoken PASSED
tests/test_tokenizer.py::test_german_roundtrip PASSED
tests/test_tokenizer.py::test_german_matches_tiktoken PASSED
tests/test_tokenizer.py::test_tinystories_sample_roundtrip PASSED
tests/test_tokenizer.py::test_tinystories_matches_tiktoken PASSED
tests/test_tokenizer.py::test_encode_special_token_trailing_newlines PASSED
tests/test_tokenizer.py::test_encode_special_token_double_newline_non_whitespace PASSED
tests/test_tokenizer.py::test_encode_iterable_tinystories_sample_roundtrip PASSED
tests/test_tokenizer.py::test_encode_iterable_tinystories_matches_tiktoken PASSED
tests/test_tokenizer.py::test_encode_iterable_memory_usage PASSED
tests/test_tokenizer.py::test_encode_memory_usage XFAIL (Tokenizer.encode is expected to
take more memory than allotted (1MB).)

========================= 24 passed, 1 xfailed in 16.28s =========================
```

1.1.6 Experiments

Problem (`tokenizer_experiments`)
Run extended test by `uv run pytest cs336_basics/BPE_Tokenizer/tests/test_02_tokenizer.py`.

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

  Solution:
  TinyStories tokenizer compresses TinyStories docs to 4.12 bytes/token ratio.
  OpenWebText tokenizer compresses OpenWebText docs to 4.37 bytes/token ratio.

  ```
  [test_02_tinystories_valid_profile_tokenizer]
  [OK] First-100-character encode/decode check passed
  [OK] Calculating statistics of tokenizer:
    - vocab size : 10000
    - merges     : 9743
    - bytes in file: 22502601
    - token count: 5465883
    - compression ratio (avg over 10 docs, seed = 51): 4.1157 bytes/token
    - elapsed time: 2.78 sec
    - throughput: 7.73 MB/sec

  [test_03_tinystories_train_profile_tokenizer]
  ```

```
[OK] First-100-character encode/decode check passed
[OK] Calculating statistics of tokenizer:
  - vocab size : 10000
  - merges     : 9743
  - bytes in file: 2227753162
  - token count: 541229357
  - compression ratio (avg over 10 docs, seed = 51): 4.1157 bytes/token
  - elapsed time: 204.11 sec
  - throughput: 10.41 MB/sec

[test_05_OpenWebText_valid_profile_tokenizer]
[OK] First-100-character encode/decode check passed
[OK] Calculating statistics of tokenizer:
  - vocab size : 32000
  - merges     : 31743
  - bytes in file: 289998753
  - token count: 66406936
  - compression ratio (avg over 10 docs, seed = 51): 4.4054 bytes/token
  - elapsed time: 32.23 sec
  - throughput: 8.58 MB/sec

[test_06_OpenWebText_train_profile_tokenizer]
[OK] First-100-character encode/decode check passed
[OK] Calculating statistics of tokenizer:
  - vocab size : 32000
  - merges     : 31743
  - bytes in file: 11920511059
  - token count: 2727321299
  - compression ratio (avg over 10 docs, seed = 51): 4.3716 bytes/token
  - elapsed time: 4585.96 sec
  - throughput: 2.48 MB/sec
```

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

  Solution:
  TinyStories tokenizer is not trained on OpenWebText, so it produces smaller subword tokens.
  This increases the number of tokens per byte, which explains the lower bytes/token ratio.
  The tokenizer is less efficient on out-of-domain text, fragmenting words more aggressively.

```
[test_04_OpenWebText_with_TinyStories_tokenizer]
[Experiment] OpenWebText encoded with TinyStories tokenizer
  - total bytes : 28992792
  - total tokens : 9017790
  - avg bytes/token : 3.2151

[Experiment] TinyStories encoded with TinyStories tokenizer
  - total bytes : 9919203
  - total tokens : 2410116
  - avg bytes/token : 4.1157
```

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

Solution:
It would take $825 \times 1024 \text{ MB} \div 8 \text{ MB/sec} \div 3600 \text{ sec/hour} \approx 29.3$ hours.

- (d) Using the previously-trained TinyStories and OpenWebText tokenizers,
  encode the respective training and validation 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?

  Solution:

  - TinyStories vocab size $= 10,000$
  - OpenWebText vocab size $= 32,000$

  A `uint16` can represent integers $0 \sim 65,535$, which is larger than the largest token ID.
  Using `uint16` arrays is both memory-efficient and well-suited for training.

1.2 Transformer

1.2.1 Architecture

Given a sequence of token IDs,
the Transformer language model uses an input embedding to convert token IDs to dense vectors.
Each embedding layer takes in a tensor of integers of shape `(batch_size, sequence_length)`
and produces a sequence of vectors of shape `(batch_size, sequence_length, d_model)`.

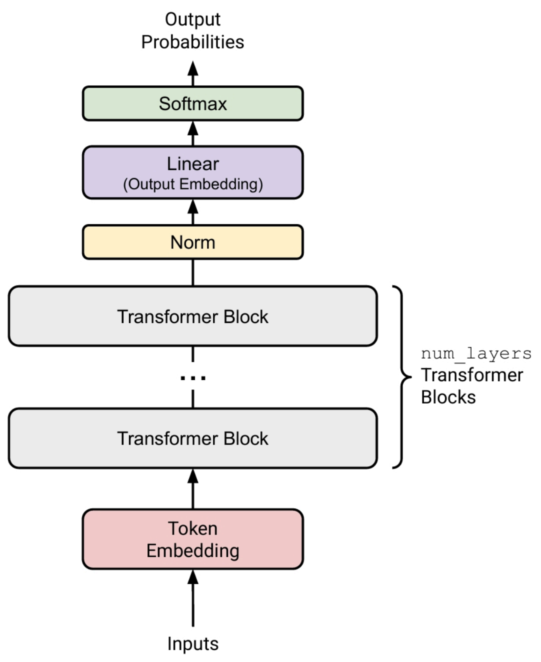


Figure 1: An overview of our Transformer language model.

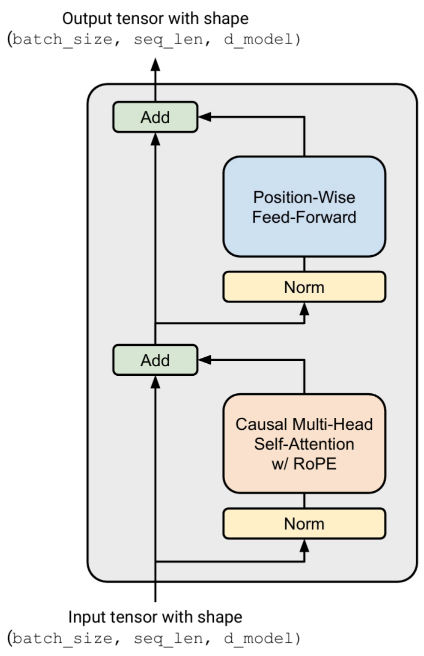


Figure 2: A pre-norm Transformer block.

Then the embedded tokens are passed through `num_layers` 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.

At the end of Transformer blocks, we use layer normalization to ensure its outputs are properly scaled, after which we use a linear transformation to convert the output into predicted next-token logits.

1.2.2 Basic Building Blocks

(1) Linear Module

Problem (`linear`)
Implement a `Linear` class that inherits from `torch.nn.Module`
and performs a linear transformation $y = Wx$, which does not include a bias term.

- `def __init__(self, in_features, out_features, device=None, dtype=None)`
  Construct a linear transformation module.
  Construct and store the parameter as W (not W^{T}), putting it in an `nn.Parameter`.
  For initializations, use $\mathcal{N}(\mu = 0, \sigma^2 = 2/(d_{\text{in}} + d_{\text{out}}))$ truncated at $[-3\sigma, 3\sigma]$.
- `def forward(self, x: torch.Tensor) -> torch.Tensor`
  Apply the linear transformation to the input.

For tests, implement the test adapter at `adapters.run_linear`.

Solution:
Run the standard test suite by `uv run pytest -k test_linear`:

```
========================= test session starts ==========================
platform linux -- Python 3.11.13, pytest-8.4.1, pluggy-1.6.0
rootdir: /root/Share/Stanford-CS336-spring25/assignment1-basics
configfile: pyproject.toml
plugins: jaxtyping-0.3.2
collected 57 items / 56 deselected / 1 selected


tests/test_model.py::test_linear PASSED

========================= 1 passed, 56 deselected in 0.95s =========================
```

(2) Embedding Module

Problem (`embedding`)
Implement a `Embedding` class that inherits from `torch.nn.Module` and performs an embedding lookup.

- `def __init__(self, num_embeddings, embedding_dim, device=None, dtype=None)`
  Construct an embedding module.
  Construct and store the embedding matrix of shape `(vocab_size,d_model)`.
  For initializations, use $\mathcal{N}(\mu = 0, \sigma^2 = 1))$ truncated at $[-3, 3]$.
- `def forward(self, token_ids: torch.Tensor) -> torch.Tensor`
  Lookup the embedding vectors for the given token IDs of shape `(batch_size,sequence_length)`.

For tests, implement the test adapter at `adapters.run_embedding`.

Solution:
Run the standard test suite by `uv run pytest -k test_embedding`:

```
========================= test session starts ==========================
platform linux -- Python 3.11.13, pytest-8.4.1, pluggy-1.6.0
rootdir: /root/Share/Stanford-CS336-spring25/assignment1-basics
configfile: pyproject.toml
plugins: jaxtyping-0.3.2
collected 57 items / 56 deselected / 1 selected


tests/test_model.py::test_embedding PASSED

========================= 1 passed, 56 deselected in 0.15s =========================
```

(3) RMSNorm

We use root mean square layer normalization (RMSNorm) for layer normalization.
Given a vector $a \in \mathbb{R}^{d_{\text{model}}}$ of activations, RMSNorm will rescale each activation a_i as follows:

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

where g_i $(i = 1, \ldots, d_{\text{model}})$ are learnable "gain" parameters (initialized as 1) and

$$\text{RMS}(a) = \sqrt{\frac{1}{d_{\text{model}}} \sum_{i=1}^{d_{\text{model}}} a_i^2 + \varepsilon}.$$

Here, ε is a hyperparameter that is often fixed at 10^{-5}.

Problem (`rmsnorm`)
Implement `RMSNorm` as a `torch.nn.Module`.

- `def __init__(self, d_model: int, eps: float = 1e-5, device=None, dtype=None)`
  Construct the RMSNorm module.
- `def forward(self, x: torch.Tensor) -> torch.Tensor`
  Process an input tensor of shape `(batch_size, sequence_length, d_model)`
  and return a tensor of the same shape.
  Upcast the input to `torch.float32` to prevent overflow during squaring.

For tests, implement the test adapter at `adapters.run_rmsnorm`.

Solution:
Run the standard test suite by `uv run pytest -k test_rmsnorm`:

```
========================= test session starts =========================
platform linux -- Python 3.11.13, pytest-8.4.1, pluggy-1.6.0
rootdir: /root/Share/Stanford-CS336-spring25/assignment1-basics
configfile: pyproject.toml
plugins: jaxtyping-0.3.2
collected 57 items / 56 deselected / 1 selected


tests/test_model.py::test_rmsnorm PASSED

========================= 1 passed, 56 deselected in 0.10s =========================
```

(4) SwiGLU

We use SwiGLU function which combines the Swish activation with a GLU, omitting bias terms.
The Swish activation is defined as follows:

$$\mathrm{SiLU}(x) := x \cdot \sigma(x) = \frac{x}{1 + \mathrm{e}^{-x}}.$$

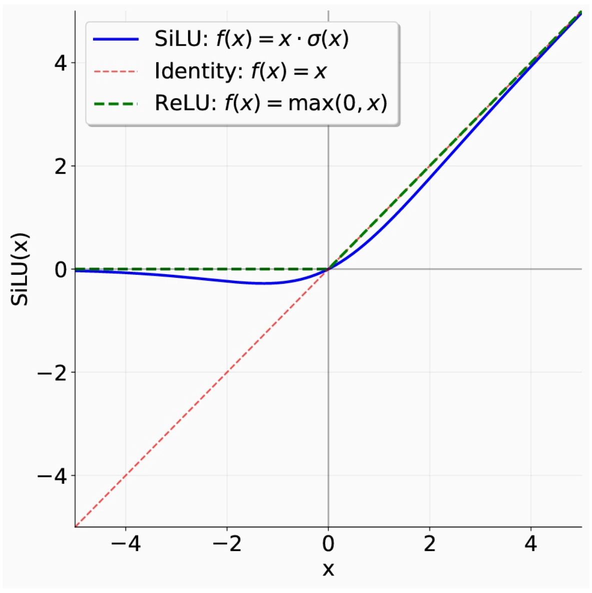


Figure 3: Comparing the SiLU (aka Swish) and ReLU activation functions.

GLU (Gated Linear Units) is defined as follows:

$$\mathrm{GLU}(x, W_1, W_2) := \sigma(W_1 x) \odot (W_2 x),$$

where $\odot$ represents Hadamard product.

Putting the Swish and GLU together, we get the SwiGLU feed-forward network:

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

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

Problem (`swiglu`)

Implement the `SwiGLU` feed-forward network.
Set the dimensionality of the hidden layer is a multiple of 64 to make good use of your hardware.
For tests, implement the test adapter at `adapters.run_swiglu`.

Solution:

Run the standard test suite by `uv run pytest -k test_swiglu`:

```
========================= test session starts ==========================
platform linux -- Python 3.11.13, pytest-8.4.1, pluggy-1.6.0
rootdir: /root/Share/Stanford-CS336-spring25/assignment1-basics
configfile: pyproject.toml
plugins: jaxtyping-0.3.2
collected 57 items / 56 deselected / 1 selected


tests/test_model.py::test_swiglu PASSED

========================= 1 passed, 56 deselected in 1.05s ==========================
```

(5) RoPE

To inject positional information into the model, we use RoPE (Rotary Position Embeddings).
For a given query token $q^{(i)} = W_q x^{(i)} \in \mathbb{R}^d$ at position i, we calculate $q'^{(i)} = R^{(i)} q^{(i)}$,
where $R^{(i)}$ will rotate $q^{(i)}_{2k-1:2k}$ by $\theta_{i,k} = i/\Theta^{(2k-2)/d}$ $(k \in \{1, \dots, d/2\})$ (Θ is a constant).
The full rotation matrix is:

$$R^{(i)} = R^{(i)}_1 \oplus R^{(i)}_2 \oplus \cdots \oplus R^{(i)}_{d/2},$$
$$\text{where } R^{(i)}_k = \begin{bmatrix} \cos(\theta_{i,k}) & -\sin(\theta_{i,k}) \\ \sin(\theta_{i,k}) & \cos(\theta_{i,k}) \end{bmatrix}.$$

Note that we can reuse the values of $\cos(\theta_{i,k})$ and $\sin(\theta_{i,k})$ across different layers and batches.
We may use a single `RoPE` module referenced by all layers,
initialized with `self.register_buffer(persistent=False)`,
instead of a `nn.Parameter` (because cosine and sine values are not learnable).
The exact same rotation process we did for $q^{(i)}$ is then done for $k^{(j)}$.

Problem (`rope`)

Implement `RoPE` module as a `torch.nn.Module`.

- `def __init__(self, theta: float, d_k: int, max_seq_len: int, device=None)`
  Construct the RoPE module and create buffers if needed.
- `def forward(self, x: torch.Tensor, token_positions: torch.Tensor) -> torch.Tensor`
  Process an input tensor of shape `(..., seq_len, d_k)` and return a tensor of the same shape.
  Note that we should tolerate `x` with an arbitrary number of batch dimensions.
  We assume that the token positions are a tensor of shape `(..., seq_len)`,
  specifying the token positions of `x` along the sequence dimension.
  Use the token positions to slice cos and sin tensors along the sequence dimension.

For tests, implement the test adapter at `adapters.run_rope`.

Solution:
Run the standard test suite by `uv run pytest -k test_rope`:

```
========================= test session starts ==========================
platform linux -- Python 3.11.13, pytest-8.4.1, pluggy-1.6.0
rootdir: /root/Share/Stanford-CS336-spring25/assignment1-basics
configfile: pyproject.toml
plugins: jaxtyping-0.3.2
collected 57 items / 56 deselected / 1 selected


tests/test_model.py::test_rope PASSED

========================= 1 passed, 56 deselected in 0.10s =========================
```

(6) Softmax

Problem (`softmax`)
Implement `softmax` function to apply the softmax operation on a specific dimension of a tensor.
The function should take two parameters: a tensor and a dimension i.
The output tensor should have the same shape as the input tensor,
but its i-th dimension will have a normalized probability distribution.

Use the trick of subtracting the maximum value in the i-th dimension
from all elements of the i-th dimension to avoid numerical stability issues.

For tests, implement the test adapter at `adapters.run_softmax`.

Solution:
Run the standard test suite by `uv run pytest -k test_softmax`:

```
========================= test session starts ==========================
platform linux -- Python 3.11.13, pytest-8.4.1, pluggy-1.6.0
rootdir: /root/Share/Stanford-CS336-spring25/assignment1-basics
configfile: pyproject.toml
plugins: jaxtyping-0.3.2
collected 57 items / 56 deselected / 1 selected


tests/test_nn_utils.py::test_softmax_matches_pytorch PASSED

========================= 1 passed, 56 deselected in 0.10s =========================
```

(7) Scaled Dot-Product Attention

The scaled dot-product attention operation is as follows:

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

where $Q \in \mathbb{R}^{n\times d_k}$, $K \in \mathbb{R}^{m\times d_k}$, and $V \in \mathbb{R}^{m\times d_v}$.

It is sometimes convenient to mask the output of an attention operation.
A mask should have the shape $M \in \{\text{True}, \text{False}\}^{n\times m}$,
and each row i of this boolean matrix indicates which keys the query i should attend to.

Canonically, a value of `True` at position (i, j) indicates that the query i attends to the key j,
and a value of `False` indicates that the query does not attend to the key.
In other words, "information flows" at (i, j) pairs with value `True`.
For example, consider a 1×3 mask matrix with entries:

$$[[\text{True}, \text{True}, \text{False}]]$$

The single query vector attends only to the first two keys.

It will be much more efficient to use masking than to compute attention on subsequences.
We can do this by taking the pre-softmax values (QK^T)

$$\left(\frac{QK^\text{T}}{\sqrt{d_k}}\right)$$

and adding a $-\infty$ in any entry of the mask matrix that is `False`.

Problem (`scaled_dot_product_attention`)
Implement scaled dot-product attention, which should handle:

- keys and queries of shape `(batch_size, ..., seq_len, d_k)`
- values of shape `(batch_size, ..., seq_len, d_v)`

The implementation should return an output with the shape `(batch_size, ..., d_v)`.
It should also support an optional user-provided boolean mask of shape `(seq_len, seq_len)`.
The attention probabilities of positions with a mask value of `True` should collectively sum to 1,
and the attention probabilities of positions with a mask value of `False` should be 0.

For tests, implement the test adapter at `adapters.run_scaled_dot_product_attention`.

Solution:
Run the standard test suite by `uv run pytest -k test_scaled_dot_product_attention`:

```
========================= test session starts ==========================
platform linux -- Python 3.11.13, pytest-8.4.1, pluggy-1.6.0
rootdir: /root/Share/Stanford-CS336-spring25/assignment1-basics
configfile: pyproject.toml
plugins: jaxtyping-0.3.2
collected 57 items / 56 deselected / 1 selected


tests/test_model.py::test_scaled_dot_product_attention PASSED

========================= 1 passed, 56 deselected in 1.01s =========================
```

Run the standard test suite by `uv run pytest -k test_4d_scaled_dot_product_attention`:

```
========================= test session starts ==========================
platform linux -- Python 3.11.13, pytest-8.4.1, pluggy-1.6.0
rootdir: /root/Share/Stanford-CS336-spring25/assignment1-basics
configfile: pyproject.toml
plugins: jaxtyping-0.3.2
collected 57 items / 56 deselected / 1 selected


tests/test_model.py::test_4d_scaled_dot_product_attention PASSED

========================= 1 passed, 56 deselected in 1.09s =========================
```

(8) Causal Multi-Head Self-Attention

The multi-head attention is defined as follows:

$$\mathrm{MultiHead}(Q, K, V) := \mathrm{Concat}(\mathrm{head}_1, \ldots, \mathrm{head}_h),$$

where $\mathrm{head}_i = \mathrm{Attention}(Q_i, K_i, V_i)$ with Q_i, K_i, V_i being the i-th slice of Q, K, V.
From this we can form the multi-head self-attention operation:

$$\mathrm{MultiHeadSelfAttention}(x) := W_O \cdot \mathrm{MultiHead}(W_Q x, W_K x, W_V x),$$

where $W_Q, W_K \in \mathbb{R}^{hd_k \times d_{\text{model}}},\ W_V \in \mathbb{R}^{hd_v \times d_{\text{model}}},\ W_O \in \mathbb{R}^{d_{\text{model}} \times hd_v}$.
We can combine W_Q, W_K, W_V into a single weight matrix
and use a single matrix multiply to compute the query, key and value projections.

The implementation should prevent the model from attending to future tokens in the sequence.
For an input token sequence $t_1, \ldots, t_n$, we apply **causal masking**,
which allows token i to attend only to positions $j \leq i$.
We can use `torch.tril` to construct a lower-triangular mask and filling those entries with `True`.

RoPE should be applied to the query and key vectors, but not the value vectors.
The same RoPE rotation should be applied to the query and key vectors for each head.

Problem (`multihead_self_attention`):
Implement causal multi-head self-attention as a `torch.nn.Module`,
which should accept (at least) the following parameters:

- `d_model`: dimensionality of the Transformer block inputs.
- `num_heads`: number of heads to use in multi-head self-attention

Set $d_k = d_v = d_{\text{model}}/h$.
For tests, implement the test adapters at `adapters.run_multihead_self_attention`
and `adapters.run_multihead_self_attention_with_rope`.

Solution:
Run the standard test suite by `uv run pytest -k test_multihead_self_attention`:

```
========================== test session starts ==========================
platform linux -- Python 3.11.13, pytest-8.4.1, pluggy-1.6.0
rootdir: /root/Share/Stanford-CS336-spring25/assignment1-basics
configfile: pyproject.toml
plugins: jaxtyping-0.3.2
collected 57 items / 55 deselected / 2 selected


tests/test_model.py::test_multihead_self_attention PASSED
tests/test_model.py::test_multihead_self_attention_with_rope PASSED

========================== 2 passed, 55 deselected in 1.54s ==========================
```

(9) Transformer Block

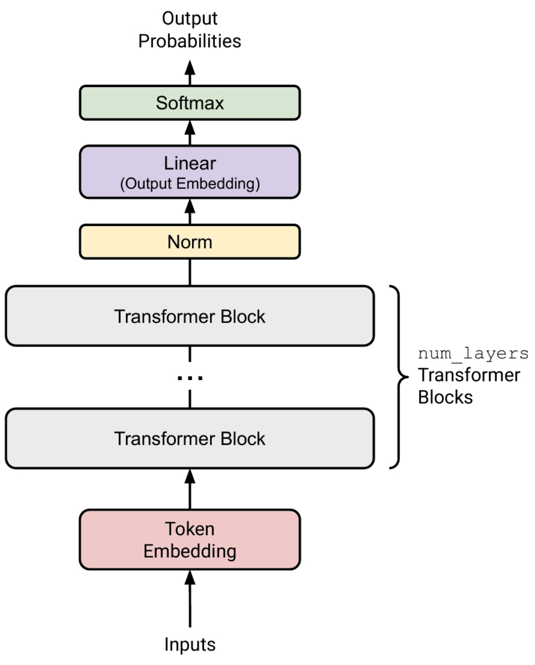


Figure 1: An overview of our Transformer language model.

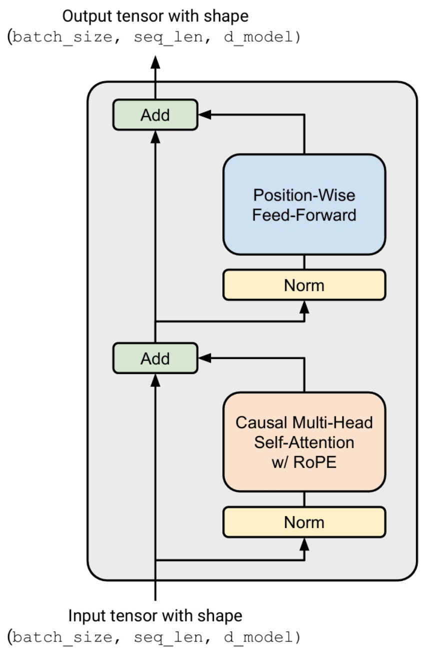


Figure 2: A pre-norm Transformer block.

A Transformer block contains two "sublayers",
one for the multihead self attention, and another for the feed-forward network.
In each sublayer, we first perform RMSNorm,
then the main operation (MHA/FF), finally adding in the residual connection.

The first "sub-layer" of the Transformer block is

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

The second "sub-layer" of the Transformer block is

$$z = y + \text{SwiGLU}(\text{RMSNorm}(y)).$$

Problem (`transformer_block`)
Implement the pre-norm Transformer block which accept (at least) the following parameters:

- `d_model`: dimensionality of the Transformer block inputs.
- `num_heads`: number of heads to use in multi-head self-attention.
- `d_ff`: dimensionality of the position-wise feed-forward inner layer.

For tests, implement the test adapter at `adapters.run_transformer_block`.

Solution:
Run the standard test suite by `uv run pytest -k test_transformer_block`:

```
========================= test session starts =========================
platform linux -- Python 3.11.13, pytest-8.4.1, pluggy-1.6.0
rootdir: /root/Share/Stanford-CS336-spring25/assignment1-basics
configfile: pyproject.toml
plugins: jaxtyping-0.3.2
collected 57 items / 56 deselected / 1 selected


tests/test_model.py::test_transformer_block PASSED

========================= 1 passed, 56 deselected in 1.01s =========================
```

1.2.3 Put Together

Problem (`transformer_lm`)
Implement the Transformer language model.
It should accept all the aforementioned construction parameters for the Transformer block,
as well as these additional parameters:

- `vocab_size`:
  The vocabulary size determines the dimensionality of the token embedding matrix.
- `context_length`:
  The maximum context length determines the dimensionality of the position embedding matrix.

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

For tests, implement the test adapter at `adapters.run_transformer_lm`.

Solution:
Run the standard test suite by `uv run pytest -k test_transformer_lm`:

```
========================= test session starts ==========================
platform linux -- Python 3.11.13, pytest-8.4.1, pluggy-1.6.0
rootdir: /root/Share/Stanford-CS336-spring25/assignment1-basics
configfile: pyproject.toml
plugins: jaxtyping-0.3.2
collected 57 items / 55 deselected / 2 selected


tests/test_model.py::test_transformer_lm PASSED
tests/test_model.py::test_transformer_lm_truncated_input PASSED

========================= 2 passed, 55 deselected in 1.09s ==========================
```

Problem (`transformer_accounting`)
The vast majority of FLOPs in a Transformer are matrix multiplies, so our approach is simple:
Find all the matrix multiplies in a Transformer forward pass and estimate the required FLOPs.
We evaluate multiple GPT-2 style configurations:

```
configs = {
    "GPT2-small": dict(context_length=1024, num_layers=12, d_model=768,  num_heads=12,
d_ff=3072),
    "GPT2-medium": dict(context_length=1024, num_layers=24, d_model=1024, num_heads=16,
d_ff=4096),
    "GPT2-large": dict(context_length=1024, num_layers=36, d_model=1280, num_heads=20,
d_ff=5120),
    "GPT2-XL": dict(context_length=1024, num_layers=48, d_model=1600, num_heads=25,
d_ff=6400),
    "GPT2-XL (longer context)": dict(context_length=16384, num_layers=48, d_model=1600,
num_heads=25, d_ff=6400)
}
```

Run `python cs336_basics/Transformer/transformer_lm.py`:

```
================================================================================
Model: GPT2-small
--------------------------------------------------------------------------------
Trainable parameters: 190,460,160
Parameter memory (bytes): 761,840,640
Parameter memory (MB): 726.55 MB
Parameter memory (GB): 0.71 GB
Total FLOPs (forward): 349,630,365,696
```

```
FLOPs breakdown:
  lm_head                                    79,047,426,048   ( 22.61%)
  layers.ffn_w1                              57,982,058,496   ( 16.58%)
  layers.ffn_w3                              57,982,058,496   ( 16.58%)
  layers.ffn_w2                              57,982,058,496   ( 16.58%)
  layers.qkv_proj                            43,486,543,872   ( 12.44%)
  layers.attn_scores(QK^T)                   19,327,352,832   (  5.53%)
  layers.attn_weighted_sum(AV)               19,327,352,832   (  5.53%)
  layers.output_proj                         14,495,514,624   (  4.15%)
================================================================================

================================================================================
Model: GPT2-medium
--------------------------------------------------------------------------------
Trainable parameters: 505,629,696
Parameter memory (bytes): 2,022,518,784
Parameter memory (MB): 1928.82 MB
Parameter memory (GB): 1.88 GB
Total FLOPs (forward): 1,033,109,504,000

FLOPs breakdown:
  layers.ffn_w1                             206,158,430,208   ( 19.96%)
  layers.ffn_w3                             206,158,430,208   ( 19.96%)
  layers.ffn_w2                             206,158,430,208   ( 19.96%)
  layers.qkv_proj                           154,618,822,656   ( 14.97%)
  lm_head                                   105,396,568,064   ( 10.20%)
  layers.attn_scores(QK^T)                   51,539,607,552   (  4.99%)
  layers.attn_weighted_sum(AV)               51,539,607,552   (  4.99%)
  layers.output_proj                         51,539,607,552   (  4.99%)
================================================================================

================================================================================
Model: GPT2-large
--------------------------------------------------------------------------------
Trainable parameters: 1,072,469,760
Parameter memory (bytes): 4,289,879,040
Parameter memory (MB): 4091.15 MB
Parameter memory (GB): 4.00 GB
Total FLOPs (forward): 2,257,754,521,600

FLOPs breakdown:
  layers.ffn_w1                             483,183,820,800   ( 21.40%)
  layers.ffn_w3                             483,183,820,800   ( 21.40%)
  layers.ffn_w2                             483,183,820,800   ( 21.40%)
  layers.qkv_proj                           362,387,865,600   ( 16.05%)
  lm_head                                   131,745,710,080   (  5.84%)
  layers.output_proj                        120,795,955,200   (  5.35%)
  layers.attn_scores(QK^T)                   96,636,764,160   (  4.28%)
  layers.attn_weighted_sum(AV)               96,636,764,160   (  4.28%)
================================================================================

================================================================================
Model: GPT2-XL
```

```
--------------------------------------------------------------------------------
Trainable parameters: 2,127,057,600
Parameter memory (bytes): 8,508,230,400
Parameter memory (MB): 8114.08 MB
Parameter memory (GB): 7.92 GB
Total FLOPs (forward): 4,513,336,524,800

FLOPs breakdown:
  layers.ffn_w1                          1,006,632,960,000   ( 22.30%)
  layers.ffn_w3                          1,006,632,960,000   ( 22.30%)
  layers.ffn_w2                          1,006,632,960,000   ( 22.30%)
  layers.qkv_proj                          754,974,720,000   ( 16.73%)
  layers.output_proj                       251,658,240,000   (  5.58%)
  lm_head                                  164,682,137,600   (  3.65%)
  layers.attn_scores(QK^T)                 161,061,273,600   (  3.57%)
  layers.attn_weighted_sum(AV)             161,061,273,600   (  3.57%)
================================================================================

================================================================================
Model: GPT2-XL (longer context)
--------------------------------------------------------------------------------
Trainable parameters: 2,127,057,600
Parameter memory (bytes): 8,508,230,400
Parameter memory (MB): 8114.08 MB
Parameter memory (GB): 7.92 GB
Total FLOPs (forward): 149,522,795,724,800

FLOPs breakdown:
  layers.attn_scores(QK^T)              41,231,686,041,600   ( 27.58%)
  layers.attn_weighted_sum(AV)          41,231,686,041,600   ( 27.58%)
  layers.ffn_w1                         16,106,127,360,000   ( 10.77%)
  layers.ffn_w3                         16,106,127,360,000   ( 10.77%)
  layers.ffn_w2                         16,106,127,360,000   ( 10.77%)
  layers.qkv_proj                       12,079,595,520,000   (  8.08%)
  layers.output_proj                     4,026,531,840,000   (  2.69%)
  lm_head                                2,634,914,201,600   (  1.76%)
================================================================================
```

- (a) Construct our model using GPT-2 XL configuration.
  How many trainable parameters does the model have?
  How much memory is required to load it in `torch.float32`?

  Solution:
  GPT-2 XL has $2,127,057,600$ trainable parameters.
  Stored in `torch.float32`, it requires $8,508,230,400$ bytes ≈ 7.92 GB of memory to load.

- (b) Identify the major matrix multiplications in a forward pass and
  compute total FLOPs for one sequence of `context_length` tokens.

  Solution:
  Major matrix multiplications are:

  - QKV projection: $Q = W_Q x,\ K = W_K x,\ V = W_V x.$
  - Attention score computation: $A = \mathrm{softmax}(QK^{\mathrm{T}}/\sqrt{d_k})$

    - Attention-weighted value mixing: $s = AV$
    - SwiGLU: $z = W_2(\mathrm{SiLU}(W_1 s) \odot (W_3 s))$
    - LM head projection: $y = W_{\text{out}} z$

  Total FLOPs (forward) $= 4,513,336,524,800 \text{ FLOPs} \approx 4.51 \text{ TFLOPs}$.

- (c) Which parts of the model require the most FLOPs?

  Solution:
  For GPT-2 XL (1024 context), the FFN matmuls dominate computation,
  with W_1, W_2, W_3 collectively accounting for about 66.9% of total FLOPs.
  The attention-related matmuls contribute much less (about 7.1% total).

- (d) Repeat the above analysis with other configurations of different model sizes.
  How do the relative contribution of FLOPs of the model components change?

  Solution:
  As model size increases,
  the FFN layers take up a larger proportion of FLOPs, rising from 50% (small) to 67% (XL).
  Meanwhile, the LM head becomes proportionally cheaper (dropping from 22% to 4%),
  and attention matmuls shrink slightly in relative contribution (dropping from 10% to 7%).

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

  Solution:
  FLOPs change:

    - GPT-2 XL (1024): $4,513,336,524,800 \text{ FLOPs}$
    - GPT-2 XL ($16,384$): $149,522,795,724,800 \text{ FLOPs}$

  Ratio:

$$\frac{149.52 \times 10^{12}}{4.51 \times 10^{12}} \approx 33.1$$

  Increasing context length to $16,384$ causes total FLOPs to rise by about 33 times,
  and the dominant cost shifts from the FFN ($67\% \to 32\%$) to attention ($7\% \to 55\%$).

1.2.4 Generating Text

The last piece we need is the ability to generate text from our model.
Recall that a language model takes in a integer sequence of length `sequence_length`
and produces a matrix of size (`sequence_length` $\times$ `vocab size`),
where each column 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)**
  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.
  We take the last column of $\mathrm{TransformerLM}(x_{[1:t]}; \theta) \in \mathbb{R}^{t \times \text{vocab\_size}}$ and calculate

$$p_\theta(x_{t+1} \mid x_{[1:t]}) = \mathrm{softmax}(\mathrm{logits}_\theta)[x_{t+1}],$$
$$\text{where } \mathrm{logits}_\theta = \mathrm{TransformerLM}(x_{[1:t]}; \theta)[t] \in \mathbb{R}^{\text{vocab\_size}}.$$

In temperature scaling, we modify our softmax with a temperature parameter τ:

$$p_\theta(x_{t+1} \mid x_{[1:t]}) = \mathrm{softmax}(\mathrm{logits}_\theta[t]/\tau)[x_{t+1}].$$

Setting $\tau \to 0$ makes the largest element of logits_θ dominates,
and the output of the softmax becoms a one-hot vector concentrated at this maximal element.

- **(Decoding)**
  To generate text (decode) from our model,
  we will provide the model with a sequence of prefix tokens (the "prompt"),
  and ask it to produce a next-word probability distribution.
  Then we sample from this distribution to determine the next output token.
  This gives us a basic decoder by repeatedly sampling from these one-step conditionals,
  util we generate the end-of-sequence token `<|endoftext|>` (or reach a maximum number).

  In nucleus sampling, we modify the sampling distribution by truncating low-probability words.
  Let q be a probability distribution that we get from a (temperature-scaled) softmax.
  Nucleus sampling with hyperparameter α produces the next token according to

$$p(x_{t+1} = i \mid q) = \begin{cases} q_i / \sum_{j=1}^{\mathrm{vocab\_size}} q_j, & \text{if } i \in V(\alpha), \\ 0, & \text{otherwise}, \end{cases}$$

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

1.3 Training

1.3.1 Cross-entropy Loss

Given a training set D_{train} consisting of sequences of length m,
we define the standard cross-entropy loss function:

$$l(\theta; D_{\mathrm{train}}) := -\frac{1}{m|D_{\mathrm{train}}|} \sum_{x \in D_{\mathrm{train}}} \sum_{i=1}^{m} \log p_\theta(x_{i+1} \mid x_{[1:i]}).$$

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

$$p_\theta(x_{i+1} \mid x_{[1:i]}) = \mathrm{softmax}(\mathrm{logits}_\theta)[x_{i+1}], \quad \text{where } \mathrm{logits}_\theta \in \mathbb{R}^{\mathrm{vocab\_size}}.$$

So that

$$\begin{aligned} l_\theta[i] &= -\log \mathrm{softmax}(\mathrm{logits}_\theta)[x_{i+1}] \\ &= -\log \frac{\exp(\mathrm{logits}_\theta[x_{i+1}])}{\sum_{x=1}^{\mathrm{vocab\_size}} \exp(\mathrm{logits}_\theta[x])} \\ &= -\mathrm{logits}_\theta[x_{i+1}] + \log \sum_{x=1}^{\mathrm{vocab\_size}} \exp(\mathrm{logits}_\theta[x]). \end{aligned}$$

Shifting $\mathrm{logits}_\theta \leftarrow \mathrm{logits}_\theta - \max(\mathrm{logits}_\theta)\mathbb{1}_{\mathrm{vocab\_size}}$ keeps the value identical but prevents overflow.

Problem (`cross_entropy`)
Implement `cross_entropy` function which takes in logits_θ and target x_{i+1},
and computes the cross entropy $l_\theta[i] = -\log \text{softmax}(\text{logits}_\theta)[x_{i+1}]$.
The function should handle the following:

- Substract 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.

When we evaluate the model, we also want to report perplexity.
For a sequence of length m where we suffer cross-entropy losses $l_1, \ldots, l_m$:

$$\text{perplexity} = \exp(\frac{1}{m}\sum_{i=1}^{m} l_i).$$

For tests, implement the test adapter at `adapters.run_cross_entropy`.

Solution:
Run the standard test suite by `uv run pytest -k test_cross_entropy`:

```
========================= test session starts ==========================
platform linux -- Python 3.11.13, pytest-8.4.1, pluggy-1.6.0
rootdir: /root/Share/Stanford-CS336-spring25/assignment1-basics
configfile: pyproject.toml
plugins: jaxtyping-0.3.2
collected 57 items / 56 deselected / 1 selected


tests/test_nn_utils.py::test_cross_entropy PASSED

========================= 1 passed, 56 deselected in 0.08s =========================
```

1.3.2 The SGD Optimizer

The simplest gradient-based optimizer is Stochastic Gradient Descent (SGD):

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

where $B(t)$ is a random batch of data sampled from the training dataset,
and α_t is learning rate (default is $\alpha_t := \alpha_0/\sqrt{t+1}$).

Implement the `SGD` optimizer as a subclass of `torch.optim.Optimizer`:

- `def __init__(self, params, ...)` should initialize your optimizer.
  Here, `params` will be a collection of parameter groups to be optimized.
  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,
  and pass them to the base class constructor as a dictionary,
  where keys are the 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,
  based on the gradient `p.grad` (if it exists).

1.3.3 AdamW

Modern language models are typically trained with more sophisticated optimizers, instead of SGD.
For each parameter, AdamW 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 weight decay rate λ,
and a pair of hyperparameters (β_1, β_2) that control the updates to the moment estimates.
Typical applications set (β_1, β_2) to $(0.9, 0.999)$, but LLMs 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}):

Algorithm 1 AdamW Optimizer

```
init($\theta$)    (Initialize learnable parameters)
$m \leftarrow 0$    (Initial value of the first moment vector; same shape as $\theta$)
$v \leftarrow 0$    (Initial value of the second moment vector; same shape as $\theta$)
for $t = 1, \ldots, T$ do
    Sample batch of data $B_t$
    $g \leftarrow \nabla_\theta l(\theta; B_t)$    (Compute the gradient)
    $m \leftarrow \beta_1 m + (1 - \beta_1) g$    (Update the first moment estimate)
    $v \leftarrow \beta_2 v + (1 - \beta_2) g \odot g$    (Update the second moment estimate)
    $\alpha_t \leftarrow \alpha \sqrt{1 - (\beta_2)^t} / (1 - (\beta_1)^t)$
    $\theta \leftarrow \theta - \alpha_t m \oslash (\sqrt{v} + \varepsilon)$    (Update the parameter)
    $\theta \leftarrow (1 - \alpha\lambda)\theta$    (Apply weight decay)
end for
```

Note that t starts at 1.

Problem (`adamw`)
Implement the `AdamW` optimizer as a subclass of `torch.optim.Optimizer`.
The class takes α, β, ε and λ hyperparameters in `__init__`.
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 necessary information.
For tests, implement the test adapter at `adapters.get_adamw_cls`.

Solution:
Run the standard test suite by `uv run pytest -k test_adamw`:

```
========================= test session starts ==========================
platform linux -- Python 3.11.13, pytest-8.4.1, pluggy-1.6.0
rootdir: /root/Share/Stanford-CS336-spring25/assignment1-basics
configfile: pyproject.toml
plugins: jaxtyping-0.3.2
collected 57 items / 56 deselected / 1 selected


tests/test_optimizer.py::test_adamw PASSED

========================= 1 passed, 56 deselected in 1.71s =========================
```

1.3.4 Learning Rate Scheduling

A schedular is simply a function that takes the current step t and other relevant parameters.
and returns the learning rate to use for the gradient update at step t.

In this assignment, we will implement the cosine annealing scheduler used to train LLaMA.
It takes:

- the current step t,
- the maximum learning rate $\alpha_{\max}$,
- the minimum learning rate $\alpha_{\min}$,
- the number of warm-up iterations T_w,
- and the number of cosine annealing iterations T_c.

The learning rate at step t is defined as:

- (Warm-up)
  If $t < T_w$, then $\alpha_t = t\alpha_{\max}/T_w$.
- (Cosine annealing)
  If $T_w \leq t \leq T_c$, then $\alpha_t = \alpha_{\min} + (1/2) \cdot (1 + \cos((t - T_w)\pi/(T_c - T_w)))(\alpha_{\max} - \alpha_{\min})$.
- (Post-annealing)
  If $t > T_c$, then $\alpha_t = \alpha_{\min}$.

Problem (`learning_rate_schedule`)
Write function `cosine_annealing_sheduler` 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.
For tests, implement the test adapter at `adapters.get_lr_cosine_schedule`.

Solution:
Run the standard test suite by `uv run pytest -k test_get_lr_cosine_schedule`:

```
========================= test session starts ==========================
platform linux -- Python 3.11.13, pytest-8.4.1, pluggy-1.6.0
rootdir: /root/Share/Stanford-CS336-spring25/assignment1-basics
configfile: pyproject.toml
plugins: jaxtyping-0.3.2
collected 57 items / 56 deselected / 1 selected


tests/test_optimizer.py::test_get_lr_cosine_schedule PASSED

========================= 1 passed, 56 deselected in 0.08s =========================
```

1.3.5 Gradient Clipping

We sometimes hit training examples that yield large gradients, which destabilizes 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 before taking an optimizer step.
Given the gradient (for all parameters) g, we compute its ℓ_2-norm $||g||_2$.
If this norm is less than a maximum value M, then we leave g as is;
otherwise, we scale g down by a factor of $M/(||g||_2 + \varepsilon)$ (where $\varepsilon = 10^{-6}$ is added for numeric stability).

Problem (`gradient_clipping`)
Write function `gradient_clipping` that takes a list of parameters and a maximum l_2-norm.
It should modify each parameter gradient in place.
For tests, implement the test adapter at `adapters.run_gradient_clipping`.

Solution:
Run the standard test suite by `uv run pytest -k test_gradient_clipping`:

```
========================= test session starts ==========================
platform linux -- Python 3.11.13, pytest-8.4.1, pluggy-1.6.0
rootdir: /root/Share/Stanford-CS336-spring25/assignment1-basics
configfile: pyproject.toml
plugins: jaxtyping-0.3.2
collected 57 items / 56 deselected / 1 selected


tests/test_nn_utils.py::test_gradient_clipping PASSED

========================= 1 passed, 56 deselected in 0.13s =========================
```

1.3.6 Data Loader

The tokenized data (prepared in `tokenizer_experiments`) is a single sequence of tokens.
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.

Problem (`data_loading`)
Write `data_loader` function that takes a numpy array `x` (integer array with token IDs),
a `batch_size`, a `context_length` and a PyTorch `device` string,
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.
For tests, implement the test adapter at `adapters.run_get_batch`.

Solution:
Run the standard test suite by `uv run pytest -k test_get_batch`:

```
========================== test session starts ==========================
platform linux -- Python 3.11.13, pytest-8.4.1, pluggy-1.6.0
rootdir: /root/Share/Stanford-CS336-spring25/assignment1-basics
configfile: pyproject.toml
plugins: jaxtyping-0.3.2
collected 57 items / 56 deselected / 1 selected


tests/test_data.py::test_get_batch PASSED

========================== 1 passed, 56 deselected in 0.52s ==========================
```

1.3.7 Checkpointing

Problem (`checkpointing`)
In addition to loading data, we will also need to save models as we train.
A checkpoint should have all the states that we need to resume training.
If using a stateful optimizer (such as AdamW), we will need to save the optimizer's state.
To resume the learning rate shedule, we need to save the iteration number we stopped at.

Use `state_dict()` to save learnable weights of `nn.Module` and `nn.optim.optimizer`.
We can restore these weights later with `load_state_dict()`.
Use `torch.save()` to dump an object to a file, which can be loaded with `torch.load()`.
Implement the following two functions to load and save checkpointing:

- `def save_checkpoint(model, optimizer, iteration, out)`
  Dump all the state from the first three parameters into the file-like object `out`.
- `def load_checkpoint(src, model, optimizer)`

Load a checkpoint from `src` and recover the model and optimizer states.

For tests, implement `adapters.run_save_checkpoint` and `adapters.run_load_checkpoint`.

Solution:
Run the standard test suite by `uv run pytest -k test_load_checkpoint`:

```
========================= test session starts ==========================
platform linux -- Python 3.11.13, pytest-8.4.1, pluggy-1.6.0
rootdir: /root/Share/Stanford-CS336-spring25/assignment1-basics
configfile: pyproject.toml
plugins: jaxtyping-0.3.2
collected 57 items / 56 deselected / 1 selected


tests/test_serialization.py::test_checkpointing PASSED

========================= 1 passed, 56 deselected in 1.12s =========================
```

1.3.8 Training Loop

Problem (`training_together`)
Write a script that runs a training loop to train your model on user-provided input.
We recommend that your training script allow for (at least) the following:

- Ability to configure the model and optimizer hyperparameters using command line.
- Memory-efficient loading of training and validation large datasets.
- Serializing checkpoints to a user-provided path.
- Periodically logging training and validation performance.

See assignment1-basics/cs336_basics/Trainer/trainer.py.

1.4 Experiments

1.4.1 Running on TinyStories

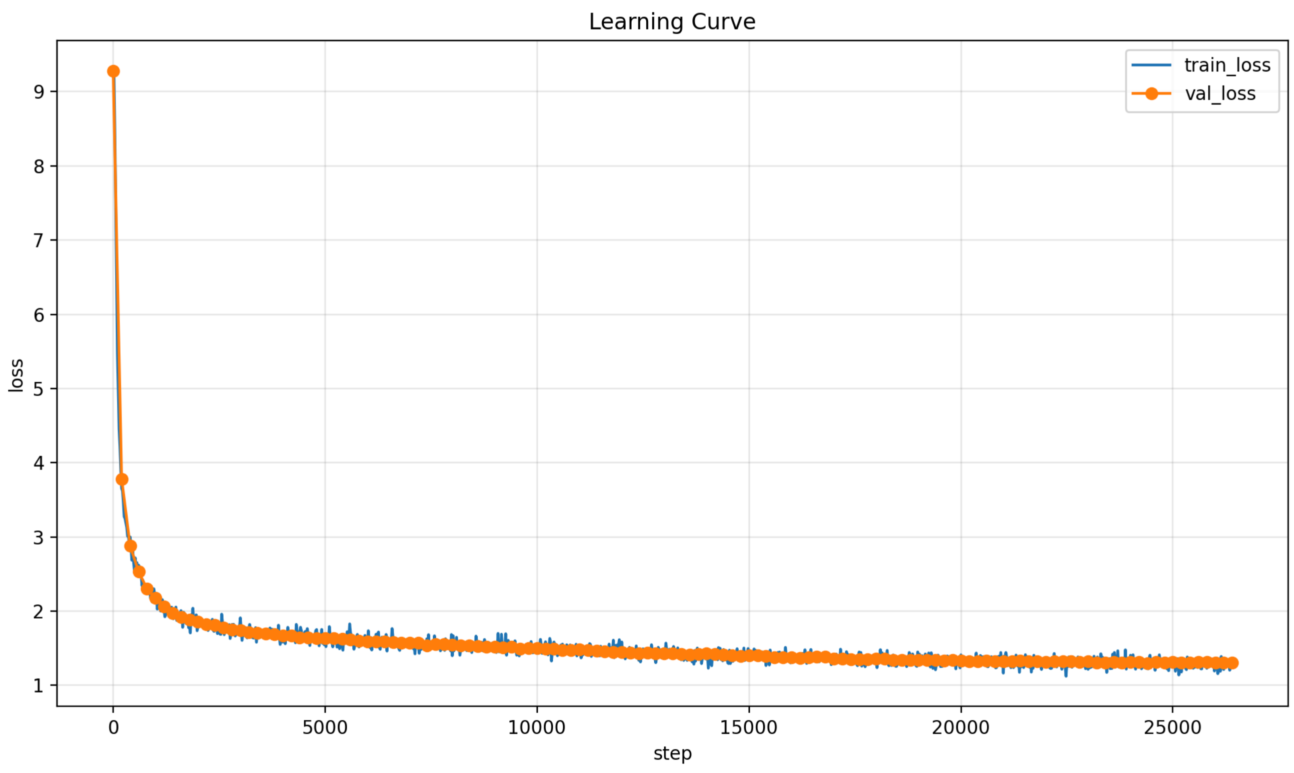
Configuration: a minor modification of GPT-small

```
========== Training Config (Final) ==========
batch_size                : 32
betas1                    : 0.9
betas2                    : 0.95
checkpoint_path           : ./scripts/checkpoints/train_on_gpu_TinyStories_20260209_011538.pt
context_length            : 256
cosine_steps              : 30000
d_ff                      : 2688
d_model                   : 1024
device                    : cuda
```

```
dtype                        : bfloat16
early_stop_min_delta         : 0.001
early_stop_patience          : 10
eps                          : 1e-08
eval_batches                 : 100
eval_every                   : 200
grad_clip                    : 1.0
log_every                    : 20
lr_max                       : 0.0005
lr_min                       : 3e-05
max_steps                    : 30000
num_heads                    : 16
num_layers                   : 20
resume                       : False
rope_theta                   : 10000.0
seed                         : 51
train_tokens                 : ./cs336_basics/BPE_Tokenizer/tests/TinyStoriesV2-GPT4-train.npy
valid_tokens                 : ./cs336_basics/BPE_Tokenizer/tests/TinyStoriesV2-GPT4-valid.npy
vocab_size                   : 10000
warmup_steps                 : 1000
weight_decay                 : 0.1
=============================================
```

- $32 \times 256 = 8,192$ tokens per step
- $26,400$ steps $\approx 216,268,800$ tokens trained ≈ 0.4 epoch
  (total token count of TinyStories training dataset is $541,229,357$)
- 9344.46 seconds on a NVIDIA GeForce RTX 4090

Learning Curve:



Prompt:

```
Once upon a time, there was a man called Snape. He was a professor at a magic school.
```

Generated text:

```
Once upon a time, there was a man called Snape. He was a professor at a magic school. The
school was filled with children who loved to play.

One day, a little three year old boy, Tim, had a fun idea. He thought if he could collect
some of the children from school to the next day, they would help him.

So, Jul acting like he was a big boy. He worked hard to make sure all the children were
successful at finding the most delicious treats. Everyone was impressed with the boy's idea.

When it was time to go home, Tim was very pleased with himself. He had collected lots of
delicious things and everyone was happy to see him having so much fun. From then on, Tim was
known as the hate person who collected the most delicious treats!
<|endoftext|>
```

1.4.2 Running on OpenWebText

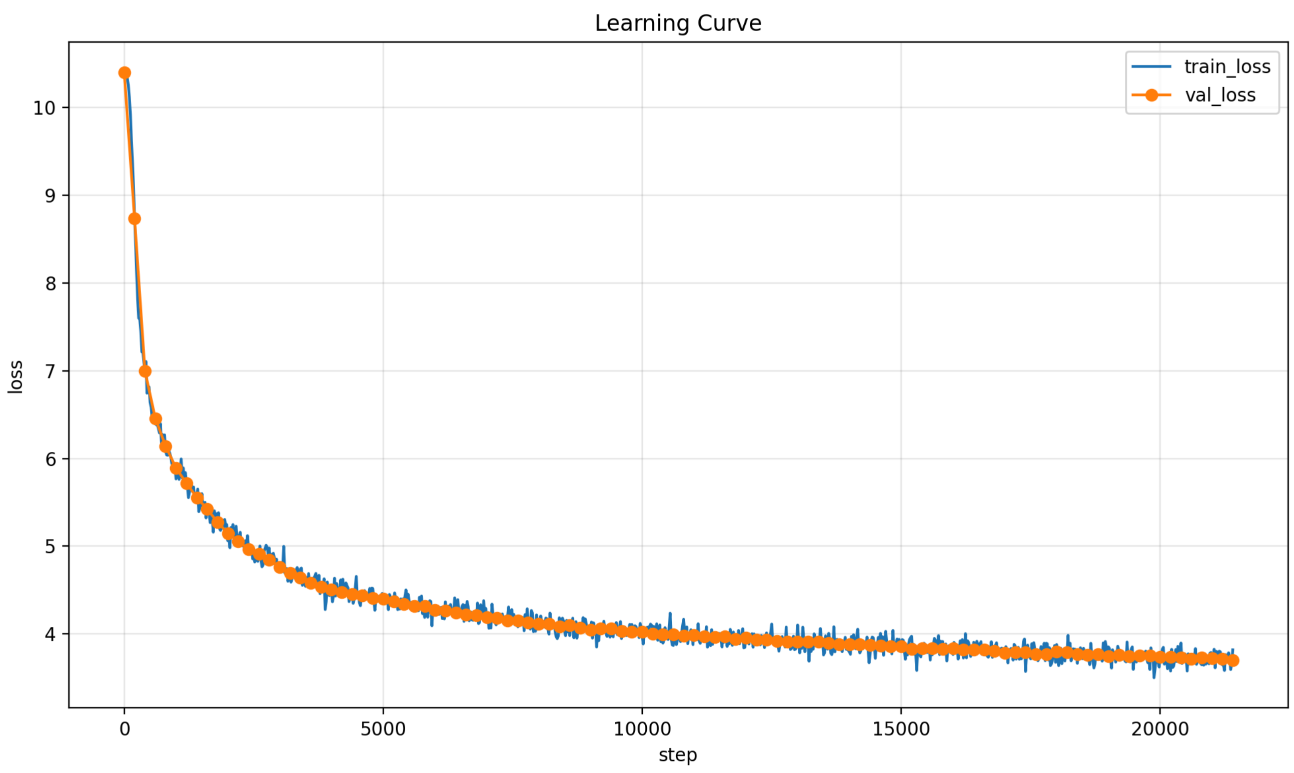
Configuration: a minor modification of GPT-medium

```
========== Training Config (Final) ==========
batch_size                : 30
betas1                    : 0.9
betas2                    : 0.95
checkpoint_path           : ./scripts/checkpoints/train_on_gpu_OpenWebText_20260209_085954.pt
context_length            : 512
cosine_steps              : 50000
d_ff                      : 2720
d_model                   : 1024
device                    : cuda
dtype                     : bfloat16
early_stop_min_delta      : 0.0001
early_stop_patience       : 20
eps                       : 1e-08
eval_batches              : 100
eval_every                : 200
grad_clip                 : 1.0
log_every                 : 20
lr_max                    : 0.0006
lr_min                    : 3e-05
max_steps                 : 50000
num_heads                 : 16
num_layers                : 24
resume                    : False
rope_theta                : 10000.0
seed                      : 51
train_tokens              : ./cs336_basics/BPE_Tokenizer/tests/owt_train.npy
valid_tokens              : ./cs336_basics/BPE_Tokenizer/tests/owt_valid.npy
```

```
vocab_size                  : 32000
warmup_steps                : 5000
weight_decay                : 0.01
=============================================
```

- $30 \times 512 = 15360$ tokens per step
- 50000 steps $\approx 768,000,000$ tokens trained ≈ 0.3 epoch
  (total token count of OpenWebText training dataset is $2727,321,299$)
- 24000 seconds on a NVIDIA GeForce RTX 4090

Learning Curve:



Prompt:

```
I have a dream, that one day this nation will
```

Generated Text:

```
I have a dream, that one day this nation will finally find a new woman, a new voice.

The question will be of particular concern if Donald Trump is the nominee of the United
States. In fact, he would be the first, first lady of the United States, to announce that
she will vote for his candidacy. Hillary Clinton is the first woman president. We are the
only ones who are willing to work hard, not with Trump. We are the only ones who will win
the 2016 election. Hillary Clinton would be the first woman President. She would be the
first woman president.
<|endoftext|>
```

The End