CS336 Assignment 1 Writeup

Linkang Dong

October 29, 2025

Contents

1 Overview

This document contains my solutions to every problem in CS336 Assignment 1, where I implement a transformer-based language model from scratch and conduct extensive experiments on model training and optimization.

Code Repository

All implementation code is available in my GitHub repository:

1. **Main Implementation Repository**: donglinkang2021/cs336-assignment1-basics
   (a) Core model implementation: `model.py`
   (b) Training script: `train.py`, `train_muon.py`
   (c) Experiment scripts: `scripts/`
   (d) Experiment changelog: `CHANGELOG.md`
   (e) Tokenize data: `data_utils/`
2. **Writeup Repository**: donglinkang2021/cs336-assignment1-writeup (this document)
   (a) Main LaTeX document: `main.tex`
   (b) Report sections: `sections/`
   (c) Plotting scripts for experiments: `plot_*.py`
   (d) The results data of experiments: `exps/`
   (e) Bibliography: `ref.bib`

Experiment Tracking

All experiments are tracked using Weights & Biases (wandb) with comprehensive logging of training/validation losses, learning rates, gradient norms, entropy, and wallclock time. The experiment reports are organized by topic in Table 1.

Table 1: Overview of Experiments and W&B Reports

Experiment	W&B	Description
Learning Rate	Link	Tune the learning rate on TinyStories and OpenWebText datasets
Batch Size	Link	Impact of batch size on training performance on TinyStories
Ablation Studies	Link	Component analysis (SwiGLU, RoPE, RMSNorm, Pre-norm) on TinyStories
Main	Link	Loss comparison between TinyStories and OpenWebText training
Muon	Link	Using Muon for better training performance on OpenWebText
Leaderboard	Link	Final model training and leaderboard submission

Key Results:

- Validation loss of **1.45 or better** on TinyStories dataset (as required)

- Identified optimal learning rate of **0.01** through comprehensive hyperparameter search

- Found batch size **128** to be optimal, achieving best validation performance in only 8.75 minutes for 10,000 iterations

- Ablation studies of components (RoPE, RMSNorm, SwiGLU, Pre-norm) (as required)

- Trained models on both TinyStories and OpenWebText datasets with detailed performance comparisons, and get a validation loss of **3.33508** on the leaderboard.

Document Structure: The remainder of this document is organized as follows: Section 2 covers the BPE tokenizer implementation, Section 3 details the transformer architecture, Section 4 discusses language model training objectives, Section 5 presents the training loop implementation, Section 6 covers text generation methods, and Section 7 presents comprehensive experimental results. Additional implementation details and code snippets are provided in the Appendix.

2 Byte-Pair Encoding (BPE) Tokenizer

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

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

Deliverable: A one-sentence response.

Answer:

`chr(0)` returns the null byte `'\x00'`.

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

Deliverable: A one-sentence response.

Answer:

`chr(0).__repr__()` returns `'\\x00'`, adding a backslash `\` before the `x`.

(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.

Answer:

It prints nothing when it occurs in text; the result is "this is a teststring".

2.2 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.

Answer:

UTF-8 is more space-efficient for texts that are primarily in ASCII, as it uses one byte for these characters, while UTF-16 and UTF-32 use two and four bytes respectively; additionally, UTF-8 is backward compatible with ASCII and is the most widely used encoding on the web (more than 98% of all webpages).

(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.

```
>>> "hello".encode("utf-8")
b'hello'
>>> "hello".encode("utf-8").decode("utf-8")
'hello'
>>> decode_utf8_bytes_to_str_wrong("hello".encode("utf-8"))
'hello'
>>> "é".encode("utf-8")
b'\xc3\xa9'
>>> "é".encode("utf-8").decode("utf-8")
'é'
>>> decode_utf8_bytes_to_str_wrong("é".encode("utf-8"))
Traceback (most recent call last):
```

```
  File "<stdin>", line 1, in <module>
  File "<stdin>", line 2, in decode_utf8_bytes_to_str_wrong
UnicodeDecodeError: 'utf-8' codec can't decode byte 0xc3 in position 0: unexpected end of
    data
```

Answer:

An example input byte string is `b'\xc3\xa9'`, which represents the character 'é' in UTF-8. The function is incorrect because **it decodes each byte individually**, leading to a `UnicodeDecodeError` since `b'\xc3'` and `b'\xa9'` are not valid standalone UTF-8 characters.

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

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

```
>>> b'\x80\x80'.decode("utf-8")
Traceback (most recent call last):
  File "<stdin>", line 1, in <module>
UnicodeDecodeError: 'utf-8' codec can't decode byte 0x80 in position 0: invalid start byte
```

Answer:

The byte sequence `b'\x80\x80'` does not decode to any Unicode characters because it is an invalid UTF-8 sequence; in UTF-8, continuation bytes (bytes starting with `10xxxxxx`) must follow a valid leading byte, and `0x80` cannot be a leading byte.

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

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

- `input_path: str` - Path to a text file with BPE tokenizer training data.
- `vocab_size: int` - A positive integer that defines the maximum final vocabulary size (including the initial byte vocabulary, vocabulary items produced from merging, and any special tokens).
- `special_tokens: list[str]` - A list of strings to add to the vocabulary. These special tokens do not otherwise affect BPE training.

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

- `vocab: dict[int, bytes]` - The tokenizer vocabulary, a mapping from int (token ID in the vocabulary) to bytes (token bytes).
- `merges: list[tuple[bytes, bytes]]` - A list of BPE merges produced from training. Each list item is a tuple of bytes (`<token1>`, `<token2>`), representing that `<token1>` was merged with `<token2>`. The merges should be ordered by order of creation.

To test your BPE training function against our provided tests, you will first need to implement the test adapter at `[adapters.run_train_bpe]`. Then, run `uv run pytest tests/test_train_bpe.py`.

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

Deliverable: A function that trains a BPE tokenizer according to the specifications above.

```
(cs336-basics) [root:assignment1-basics]$ uv run pytest tests/test_train_bpe.py
============================== test session starts ==============================
platform linux -- Python 3.13.7, pytest-8.4.1, pluggy-1.6.0
rootdir: standford-cs336/assignment1-basics
```

```
configfile: pyproject.toml
plugins: jaxtyping-0.3.2, hydra-core-1.3.2
collected 3 items

tests/test_train_bpe.py::test_train_bpe_speed PASSED
tests/test_train_bpe.py::test_train_bpe PASSED
tests/test_train_bpe.py::test_train_bpe_special_tokens PASSED

============================== 3 passed in 22.04s ==============================
```

Answer:

I have implemented a BPE tokenizer training function that can be found at: bpe.py#L11.

The implementation successfully passes all tests in `uv run pytest tests/test_train_bpe.py` with a runtime of approximately 22 seconds in my machine.

2.4 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: ≤30 minutes (no GPUs), ≤30 GB RAM

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

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

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

Deliverable: A one-to-two sentence response.

Answer:

Since my initial implementation was too slow for the TinyStories dataset, I used the `tokenizers.trainers.BpeTrainer` from the Hugging Face tokenizers library (code available at hf_train_bpe.py). Training completed in 202.87 seconds (3.38 minutes) with the longest token being 'Ġresponsibility' (15 characters), which makes sense as it's a complete meaningful word that appears frequently enough to be merged into a single token.

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

Deliverable: A one-to-two sentence response.

Answer:

Based on the training output, the pre-processing phase (reading and processing the 2250 MB TinyStories files) took the majority of the time at over 3 minutes, while the actual BPE merge computation was completed very quickly in under a second.

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

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

Resource requirements: ≤12 hours (no GPUs), ≤100 GB RAM

Deliverable: A one-to-two sentence response.

Answer:

Training on OpenWebText completed in 1045.20 seconds (17.42 minutes) with the longest token being a 64-character dash line '--', which makes sense as such formatting patterns are common in web text and code documentation found in the OpenWebText dataset.

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

Deliverable: A one-to-two sentence response.

```
(cs336-basics) [root:assignment1-basics]$ python data_utils/compare_tokenizer_demo.py
TinyStories vocab size: 10000
OpenWebText vocab size: 32000
TinyStories avg token length: 5.83
OpenWebText avg token length: 6.34
Character distribution:
TinyStories: {'alphabetic': 58120, 'special': 151, 'numeric': 19}
OpenWebText: {'alphabetic': 199840, 'numeric': 1699, 'special': 1395}
Common tokens: 7283
TinyStories unique tokens: 2717
OpenWebText unique tokens: 24717

Sample unique TinyStories tokens:
['Ġcluck', 'aisy', 'Ġreluctantly', 'Splash', 'Ġnapkin', 'Ġpillows', 'Julie', 'Ġsqueezing', '
    Ġlaug', 'Ġconn']

Sample unique OpenWebText tokens:
['ylan', 'igo', 'Ġemphas', 'Ġsuburbs', 'ĠCab', 'ĠMarine', 'liable', 'ĠBrett', 'model', 'ĠSQL
    ']
```

Answer:

The OpenWebText tokenizer (32k vocab) produces more diverse tokens including technical symbols and formatting patterns (avg length: 6.34 chars, 24,717 unique tokens) with terms like 'ĠMarine' and 'ĠSQL', while the TinyStories tokenizer (10k vocab) focuses on simpler narrative vocabulary (avg length: 5.83 chars, 2,717 unique tokens) with words like 'Ġcluck' and 'Ġreluctantly', with 7,283 tokens shared between both datasets reflecting common English usage (comparison code available at compare_tokenizer_demo.py).

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

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

- `def __init__(self, vocab, merges, special_tokens=None)` - Construct a tokenizer from a given vocabulary, list of merges, and (optionally) a list of special tokens. This function should accept the following parameters:
  - `vocab: dict[int, bytes]`
  - `merges: list[tuple[bytes, bytes]]`
  - `special_tokens: list[str] | None = None`
- `def from_files(cls, vocab_filepath, merges_filepath, special_tokens=None)` - Class method that constructs and returns a Tokenizer from a serialized vocabulary and list of merges (in the same format that your BPE training code output) and (optionally) a list of special tokens. This method should accept the following additional parameters:
  - `vocab_filepath: str`
  - `merges_filepath: str`
  - `special_tokens: list[str] | None = None`
- `def encode(self, text: str)-> list[int]` - Encode an input text into a sequence of token IDs.
- `def encode_iterable(self, iterable: Iterable[str])-> Iterator[int]` - Given an iterable of strings (e.g., a Python file handle), return a generator that lazily yields token IDs. This is required for memory-efficient tokenization of large files that we cannot directly load into memory.
- `def decode(self, ids: list[int])-> str` - Decode a sequence of token IDs into text.

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

Deliverable: A Tokenizer class that implements the interface described above.

```
(cs336-basics) [root:assignment1-basics]$ uv run pytest tests/test_tokenizer.py
======================= test session starts ========================
platform linux -- Python 3.13.7, pytest-8.4.1, pluggy-1.6.0
rootdir: standford-cs336/assignment1-basics
configfile: pyproject.toml
plugins: jaxtyping-0.3.2, hydra-core-1.3.2
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

================== 24 passed, 1 xfailed in 14.70s ==================
```

Answer:

Code available at: tokenizer.py.

2.7 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.

Answer:

The TinyStories tokenizer (10K vocab) achieves 4.19 bytes/token compression ratio, while the OpenWebText tokenizer (32K vocab) achieves 4.65 bytes/token, showing that the larger vocabulary provides better compression efficiency for diverse web content (experiment code available at tokenizer_experiments.py).

(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.

Answer:

Using TinyStories tokenizer on OpenWebText data results in 3.32 bytes/token compared to 4.65 bytes/-token with the native tokenizer, showing a 28.6% degradation due to vocabulary mismatch and increased unknown token fragmentation.

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

Deliverable: A one-to-two sentence response.

Answer:

The tokenizer achieves approximately 2.85 MB/s throughput, suggesting it would take around 82.3 hours (3.4 days) to tokenize the entire Pile dataset (825 GB of text).

(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 data type `uint16`. Why is `uint16` an appropriate choice?

Deliverable: A one-to-two sentence response.

Answer:

`uint16` is appropriate because it can represent values up to 65,535, which easily accommodates our vocabulary sizes (10K and 32K tokens), while being twice as memory-efficient as `uint32` (1.9 MB vs 3.8 MB per 1M tokens) and avoiding the 255 value limit of `uint8`. The code to encode the datasets into `uint16` arrays and save them as `train.bin` and `val.bin` is available at hf_tokenize_data.py.

/* * @Author: linkdom * @Date: 2025-10-29 22:46:11 * @Last Modified by: xiaokang001 * @Last Modified time: 2025-10-29 22:48:06 */

3 Transformer Language Model Architecture

All transformer model implementations in this section are available at: model.py.

The complete implementation passes all 13 tests with the following results:

```
(cs336-basics) [root:assignment1-basics]$ uv run pytest tests/test_model.py
======================== test session starts ========================
platform linux -- Python 3.13.7, pytest-8.4.1, pluggy-1.6.0
rootdir: standford-cs336/assignment1-basics
configfile: pyproject.toml
plugins: jaxtyping-0.3.2, hydra-core-1.3.2
collected 13 items

tests/test_model.py::test_linear PASSED
tests/test_model.py::test_embedding PASSED
tests/test_model.py::test_swiglu PASSED
tests/test_model.py::test_scaled_dot_product_attention PASSED
tests/test_model.py::test_4d_scaled_dot_product_attention PASSED
tests/test_model.py::test_multihead_self_attention PASSED
tests/test_model.py::test_multihead_self_attention_with_rope PASSED
tests/test_model.py::test_transformer_lm PASSED
tests/test_model.py::test_transformer_lm_truncated_input PASSED
tests/test_model.py::test_transformer_block PASSED
tests/test_model.py::test_rmsnorm PASSED
tests/test_model.py::test_rope PASSED
tests/test_model.py::test_silu_matches_pytorch PASSED

======================== 13 passed in 2.63s =========================
```

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

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

- `def __init__(self, in_features, out_features, device=None, dtype=None)` - Construct a linear transformation module. This function should accept the following parameters:
  - `in_features: int` - final dimension of the input
  - `out_features: int` - final dimension of the output
  - `device: torch.device | None = None` - Device to store the parameters on
  - `dtype: torch.dtype | None = None` - Datatype of the parameters
- `def forward(self, x: torch.Tensor)-> torch.Tensor` - Apply the linear transformation to the input.

Make sure to:

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

For initializations, use the settings from above along with `torch.nn.init.trunc_normal_` to initialize the weights.

To test your Linear module, implement the test adapter at `[adapters.run_linear]`. The adapter should load the given weights into your Linear module. You can use `Module.load_state_dict` for this purpose. Then, run `uv run pytest -k test_linear`.

Answer:

> **Note**: Although the problem specification mentions not having a bias parameter, the implementation includes a bias parameter to ensure compatibility with the provided tests that load pre-trained weights with bias terms. However, in our final transformer architecture, we set `bias=False` to follow the no-bias design.

```python
# uv run pytest -k test_linear
class Linear(nn.Module):
    """ A simple linear layer implemented from scratch."""
    in_features: int
    out_features: int
    weight: torch.Tensor

    def __init__(
        self,
        in_features:int,
        out_features:int,
        bias:bool=False,
        device=None,
        dtype=None,
    ) -> None:
        kwargs = {'device': device, 'dtype': dtype}
        super().__init__()
        self.in_features = in_features
        self.out_features = out_features
        self.weight = nn.Parameter(
            torch.empty((out_features, in_features), **kwargs)
        )
        if bias:
            self.bias = nn.Parameter(torch.empty(out_features, **kwargs))
        else:
            self.register_parameter('bias', None)

    def forward(self, x:torch.Tensor) -> torch.Tensor:
        # [..., in_features] -> [..., out_features]
```

```
        return torch.einsum('...i,oi->...o', x, self.weight) + (
            self.bias if self.bias is not None else 0
        )
```

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

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

- `def __init__(self, num_embeddings, embedding_dim, device=None, dtype=None)` - Construct an embedding module. This function should accept the following parameters:
  - `num_embeddings: int` - Size of the vocabulary
  - `embedding_dim: int` - Dimension of the embedding vectors, i.e., d_{model}
  - `device: torch.device | None = None` - Device to store the parameters on
  - `dtype: torch.dtype | None = None` - Data type of the parameters
- `def forward(self, token_ids: torch.Tensor)-> torch.Tensor` - Lookup the embedding vectors for the given token IDs.

Make sure to:

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

Again, use the settings from above for initialization, and use `torch.nn.init.trunc_normal_` to initialize the weights.

To test your implementation, implement the test adapter at `[adapters.run_embedding]`. Then, run `uv run pytest -k test_embedding`.

```
# uv run pytest -k test_embedding
class Embedding(nn.Module):
    """ A simple embedding layer implemented from scratch. """
    num_embeddings: int
    embedding_dim: int
    weight: torch.Tensor

    def __init__(
        self,
        num_embeddings:int,
        embedding_dim:int,
        device=None,
        dtype=None
    ) -> None:
        kwargs = {'device': device, 'dtype': dtype}
        super().__init__()
        self.num_embeddings = num_embeddings
        self.embedding_dim = embedding_dim
        self.weight = nn.Parameter(
            torch.empty((num_embeddings, embedding_dim), **kwargs)
        )

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

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

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

- `def __init__(self, d_model: int, eps: float = 1e-5, device=None, dtype=None)` - Construct the RMSNorm module. This function should accept the following parameters:
  - `d_model: int` - Hidden dimension of the model
  - `eps: float = 1e-5` - Epsilon value for numerical stability
  - `device: torch.device | None = None` - Device to store the parameters on
  - `dtype: torch.dtype | None = None` - Data type of the parameters
- `def forward(self, x: torch.Tensor)-> torch.Tensor` - Process an input tensor of shape `(batch_size, sequence_length, d_model)` and return a tensor of the same shape.

Note: Remember to upcast your input to `torch.float32` before performing the normalization (and later downcast to the original dtype), as described above.

To test your implementation, implement the test adapter at `[adapters.run_rmsnorm]`. Then, run `uv run pytest -k test_rmsnorm`.

```
# uv run pytest -k test_rmsnorm
class RMSNorm(nn.Module):
    """
    Root Mean Square Layer Normalization (RMSNorm)
    Reference https://github.com/donglinkang2021/normalize-layers-pytorch/
    """
    d_model: int
    eps: float
    weight: torch.Tensor

    def __init__(
        self,
        d_model: int,
        eps: float = 1e-5,
        device=None,
        dtype=None
    ) -> None:
        kwargs = {'device': device, 'dtype': dtype}
        super().__init__()
        self.weight = nn.Parameter(torch.empty(d_model, **kwargs))
        self.eps = eps

    def forward(self, x:torch.Tensor) -> torch.Tensor:
        # [..., d_model] -> [..., d_model]
        in_dtype = x.dtype
        x = x.to(torch.float32)
        var = x.pow(2).mean(dim=-1, keepdim=True) + self.eps
        x_out = x * var.rsqrt() * self.weight
        return x_out.to(in_dtype)
```

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

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

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

You should set d_{ff} to approximately $\frac{8}{3} \times d_{\text{model}}$ in your implementation, while ensuring that the dimensionality of the inner feed-forward layer is a multiple of 64 to make good use of your hardware.

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

```
# uv run pytest -k test_silu
def silu(x:torch.Tensor) -> torch.Tensor:
    return x * torch.sigmoid(x)

# uv run pytest -k test_swiglu
```

```python
class SwiGLU(nn.Module):
    """ SwiGLU FFN """
    d_model: int
    d_ff: int

    def __init__(self, d_model:int, d_ff:int) -> None:
        super().__init__()
        self.d_model = d_model
        # should be roughly d_ff = 8/3 * d_model,
        # then the parameter count = 3 * d_model * 8/3 * d_model = 8 * d_model^2
        self.d_ff = d_ff
        self.w1 = Linear(in_features=d_model, out_features=d_ff)
        self.w2 = Linear(in_features=d_ff, out_features=d_model)
        self.w3 = Linear(in_features=d_model, out_features=d_ff)

    def forward(self, x:torch.Tensor) -> torch.Tensor:
        # [..., d_model] -> [..., d_model]
        return self.w2(silu(self.w1(x)) * self.w3(x))
```

Here we also provide an alternative SiLU-based FFN for ablation studies later. You can ignore this part for now.

```python
class SiLUFFN(nn.Module):
    """ FFN with SiLU activation """
    d_model: int
    d_ff: int

    def __init__(self, d_model:int, d_ff:int) -> None:
        super().__init__()
        self.d_model = d_model
        # should be d_ff = 4 * d_model,
        # then the parameter count = 2 * d_model * 4 * d_model = 8 * d_model^2
        self.d_ff = d_ff
        self.w1 = Linear(in_features=d_model, out_features=d_ff)
        self.w2 = Linear(in_features=d_ff, out_features=d_model)

    def forward(self, x:torch.Tensor) -> torch.Tensor:
        # [..., d_model] -> [..., d_model]
        return self.w2(silu(self.w1(x)))
```

3.5 Problem (rope): Implement RoPE (2 points)

Deliverable: Implement a class RotaryPositionalEmbedding that applies RoPE to the input tensor. The following interface is recommended:

- `def __init__(self, theta: float, d_k: int, max_seq_len: int, device=None)` - Construct the RoPE module and create buffers if needed.
  - `theta: float` - Θ value for the RoPE
  - `d_k: int` - dimension of query and key vectors
  - `max_seq_len: int` - Maximum sequence length that will be inputted
  - `device: torch.device | None = None` - Device to store the buffer on
- `def forward(self, x: torch.Tensor, token_positions: torch.Tensor)-> torch.Tensor` - Process an input tensor of shape `(..., seq_len, d_k)` and return a tensor of the same shape.

  Note that you should tolerate x with an arbitrary number of batch dimensions. You should assume that the token positions are a tensor of shape `(..., seq_len)` specifying the token positions of x along the sequence dimension.

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

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

Answer:

Implementation references: [karpathy/nano-llama31] and [GeeeekExplorer/nano-vllm].

Important: When implementing RoPE, pay special attention to the tensor splitting of x. The dimension splitting is performed on the `model_dim` (i.e., d_k), where we split the last dimension into pairs and apply rotation to each pair separately using the precomputed cos/sin values.

```
# uv run pytest -k test_rope
class RotaryPositionalEmbedding(nn.Module):
    """ Rotary Positional Embedding (RoPE) """
    theta: float
    d_k: int
    max_seq_len: int

    def __init__(self, theta: float, d_k: int, max_seq_len: int, device=None):
        super().__init__()
        self.theta = theta
        self.d_k = d_k
        self.max_seq_len = max_seq_len
        self.register_buffer(
            'cos_sin',
            precompute_freqs_cis(d_k, max_seq_len, theta),
            persistent=False
        )

    def forward(self, x: torch.Tensor, token_positions: torch.Tensor) -> torch.Tensor:
        # [..., seq_len, dim], [seq_len,] -> [..., seq_len, dim]
        cos_sin = self.cos_sin[:x.size(-2)] if token_positions is None else self.cos_sin[
            token_positions]
        return apply_rotary_emb(x, cos_sin)


def precompute_freqs_cis(head_dim: int, max_len: int, theta: float = 10000.0) -> torch.Tensor:
    # shape (head_dim/2,)
    freqs = 1.0 / (theta ** (torch.arange(0, head_dim, 2).float() / head_dim))

    # shape (max_len,)
    t = torch.arange(max_len, device=freqs.device).float()

    # equal to einsum('i,j->ij', t, freqs), shape (max_len, head_dim/2)
    freqs = torch.outer(t, freqs)

    # complex64, equal to torch.complex(torch.cos(freqs), torch.sin(freqs))
    freqs_cis = torch.polar(torch.ones_like(freqs), freqs)

    # [cos, sin] shape (max_len, head_dim/2 * 2)
    cos_sin = torch.cat([freqs_cis.real, freqs_cis.imag], dim=-1)
    return cos_sin

def apply_rotary_emb(x:torch.Tensor, cos_sin:torch.Tensor):
    # x1, x2 = torch.chunk(x, 2, dim=-1) # wrong
    x1, x2 = x.reshape(*x.shape[:-1], -1, 2).unbind(-1)
    cos, sin = torch.chunk(cos_sin, 2, dim=-1)
    x_out = torch.stack([x1 * cos - x2 * sin,
                         x1 * sin + x2 * cos], dim=-1)
    return x_out.reshape(*x.shape).type_as(x)
```

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

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

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

```
# uv run pytest -k test_softmax_matches_pytorch
def softmax(x:torch.Tensor, dim:int) -> torch.Tensor:
    # [..., d_model] -> [..., d_model]
    x_exp = torch.exp(x - x.max(dim=dim, keepdim=True).values)
```

```
    return x_exp / x_exp.sum(dim=dim,keepdim=True)
```

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

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

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

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

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

```
# uv run pytest -k test_scaled_dot_product_attention
# uv run pytest -k test_4d_scaled_dot_product_attention
from .nn_utils import softmax
def scaled_dot_product_attention(
    Q: torch.Tensor,
    K: torch.Tensor,
    V: torch.Tensor,
    mask: torch.Tensor = None,
) -> torch.Tensor:
    D = Q.size(-1)
    scores = torch.matmul(Q, K.transpose(-2, -1)) / math.sqrt(D)
    if mask is not None:
        scores = scores.masked_fill(mask == 0, float('-inf'))
    attn_weights = softmax(scores, dim=-1)
    output = torch.matmul(attn_weights, V)
    return output
```

3.8 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.

Following Vaswani et al. (2017), set $d_k = d_v = d_{\text{model}}/h$. To test your implementation against our provided tests, implement the test adapter at `[adapters.run_multihead_self_attention]`.

Then, run `uv run pytest -k test_multihead_self_attention` to test your implementation.

```
# uv run pytest -k test_multihead_self_attention
# pass tests/test_model.py::test_multihead_self_attention
from einops import rearrange
class MultiheadSelfAttention(nn.Module):
    """ Multi-head self-attention layer """
    d_model: int
    num_heads: int

    def __init__(self, d_model:int, num_heads:int) -> None:
        super().__init__()
        assert d_model % num_heads == 0, "d_model must be divisible by num_heads"
        self.d_model = d_model
        self.num_heads = num_heads
        self.head_dim = d_model // num_heads
        self.fc_qkv = Linear(d_model, 3*d_model)
        self.fc_out = Linear(d_model, d_model)
```

```
    def forward(self, x:torch.Tensor) -> torch.Tensor:
        # [batch, seq_len, d_model] -> [batch, seq_len, d_model]
        seq_len = x.size(1)
        qkv = self.fc_qkv(x)
        qkv = rearrange(qkv, 'B T (nH Hs) -> B nH T Hs', Hs=self.head_dim)
        xq, xk, xv = torch.chunk(qkv, 3, dim=1)
        mask = torch.ones((seq_len, seq_len), device=x.device).tril()
        mask = rearrange(mask, 'T1 T2 -> 1 1 T1 T2')
        xo = scaled_dot_product_attention(xq, xk, xv, mask)
        xo = rearrange(xo, 'B nH T Hs -> B T (nH Hs)')
        return self.fc_out(xo)

# uv run pytest -k test_multihead_self_attention
# pass tests/test_model.py::test_multihead_self_attention_with_rope
class MultiheadRoPESelfAttention(nn.Module):
    """ Multi-head self-attention layer with RoPE"""
    d_model: int
    num_heads: int

    def __init__(
        self,
        d_model:int,
        num_heads:int,
        max_seq_len:int,
        theta:float,
    ) -> None:
        super().__init__()
        assert d_model % num_heads == 0, "d_model must be divisible by num_heads"
        self.d_model = d_model
        self.num_heads = num_heads
        self.head_dim = d_model // num_heads
        self.fc_qkv = Linear(d_model, 3*d_model)
        self.fc_out = Linear(d_model, d_model)
        self.rope = RotaryPositionalEmbedding(theta, self.head_dim, max_seq_len)

    def forward(self, x:torch.Tensor, token_positions: torch.Tensor = None) -> torch.Tensor:
        # [batch, seq_len, d_model] -> [batch, seq_len, d_model]
        seq_len = x.size(1)
        qkv = self.fc_qkv(x)
        qkv = rearrange(qkv, 'B T (nH Hs) -> B nH T Hs', Hs=self.head_dim)
        xq, xk, xv = torch.chunk(qkv, 3, dim=1)
        xq = self.rope(xq, token_positions)
        xk = self.rope(xk, token_positions)
        mask = torch.ones((seq_len, seq_len), device=x.device).tril()
        mask = rearrange(mask, 'T1 T2 -> 1 1 T1 T2')
        xo = scaled_dot_product_attention(xq, xk, xv, mask)
        xo = rearrange(xo, 'B nH T Hs -> B T (nH Hs)')
        return self.fc_out(xo)
```

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

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

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

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

Deliverable: Transformer block code that passes the provided tests.

Answer:

Note: I reimplemented a separate attention module (`TransformerAttention`) specifically designed to pass the later tests by having the correct parameter names and structure that can directly load state dictionaries (`load_state_dict`) from the provided test weights. This module uses separate projection layers (`q_proj`, `k_proj`, `v_proj`, `output_proj`) instead of the combined `fc_qkv` approach used in `MultiheadSelfAttention`.

The Transformer block implementation includes several optional parameters to allow for flexibility in experimentation (for ablation studies later):

- `ffn_type: str = 'swiglu'` - Type of feed-forward network to use. Options are 'swiglu' for SwiGLU and 'silu' for SiLU-based FFN.
- `use_post_norm: bool = False` - If set to True, applies layer normalization after the attention and feed-forward sub-layers (post-norm). If False, applies layer normalization before these sub-layers (pre-norm).
- `remove_rmsnorm: bool = False` - If set to True, removes RMSNorm layers and replaces them with identity mappings.
- `remove_rope: bool = False` - If set to True, removes RoPE from the attention mechanism.

```python
from functools import lru_cache

@lru_cache(1)
def get_rope(theta: float, d_k: int, max_seq_len: int) -> RotaryPositionalEmbedding:
    return RotaryPositionalEmbedding(theta, d_k, max_seq_len)

class TransformerAttention(nn.Module):
    """ Transformer Attention with RoPE for TransformerBlock """
    d_model: int
    num_heads: int

    def __init__(
        self,
        d_model:int,
        num_heads:int,
        max_seq_len:int,
        theta:float,
    ) -> None:
        super().__init__()
        assert d_model % num_heads == 0, "d_model must be divisible by num_heads"
        self.d_model = d_model
        self.num_heads = num_heads
        self.head_dim = d_model // num_heads
        self.q_proj = Linear(d_model, d_model)
        self.k_proj = Linear(d_model, d_model)
        self.v_proj = Linear(d_model, d_model)
        self.output_proj = Linear(d_model, d_model)
        self.rope = get_rope(theta, self.head_dim, max_seq_len)

    def forward(self, x:torch.Tensor, token_positions: torch.Tensor = None) -> torch.Tensor:
        # [batch, seq_len, d_model] -> [batch, seq_len, d_model]
        seq_len = x.size(1)
        q = self.q_proj(x)
        k = self.k_proj(x)
        v = self.v_proj(x)
        xq = rearrange(q, 'B T (nH Hs) -> B nH T Hs', Hs=self.head_dim)
        xk = rearrange(k, 'B T (nH Hs) -> B nH T Hs', Hs=self.head_dim)
        xv = rearrange(v, 'B T (nH Hs) -> B nH T Hs', Hs=self.head_dim)
        if self.rope is not None:
            xq = self.rope(xq, token_positions)
            xk = self.rope(xk, token_positions)
        mask = torch.ones((seq_len, seq_len), device=x.device).tril()
        mask = rearrange(mask, 'T1 T2 -> 1 1 T1 T2')
        xo = scaled_dot_product_attention(xq, xk, xv, mask)
        xo = rearrange(xo, 'B nH T Hs -> B T (nH Hs)')
        return self.output_proj(xo)


# uv run pytest -k test_transformer_block
class TransformerBlock(nn.Module):
    """ Transformer Block """
    def __init__(
```

```
        self,
        d_model:int,
        num_heads:int,
        d_ff:int,
        max_seq_len:int,
        theta:float,
        ffn_type:str = 'swiglu',
        use_post_norm:bool = False,
        remove_rmsnorm:bool = False,
        remove_rope:bool = False,
    ) -> None:
        super().__init__()
        self.attn = TransformerAttention(
            d_model, num_heads, max_seq_len, theta
        )
        if remove_rope:
            self.attn.rope = None

        if ffn_type == 'swiglu':
            self.ffn = SwiGLU(d_model, d_ff)
        elif ffn_type == 'silu':
            self.ffn = SiLUFFN(d_model, d_ff)
        else:
            raise ValueError(f"Unknown ffn_type: {ffn_type}")

        self.use_post_norm = use_post_norm
        if remove_rmsnorm:
            self.ln1 = nn.Identity()
            self.ln2 = nn.Identity()
        else:
            self.ln1 = RMSNorm(d_model)
            self.ln2 = RMSNorm(d_model)

    def forward(self, x:torch.Tensor) -> torch.Tensor:
        # [batch, seq_len, d_model] -> [batch, seq_len, d_model]
        if self.use_post_norm:
            x = self.ln1(x + self.attn(x))
            x = self.ln2(x + self.ffn(x))
        else:
            x = x + self.attn(self.ln1(x))
            x = x + self.ffn(self.ln2(x))
        return x
```

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

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

- `vocab_size: int` - The size of the vocabulary, necessary for determining the dimensionality of the token embedding matrix.
- `context_length: int` - The maximum context length, necessary for determining the dimensionality of the position embedding matrix.
- `num_layers: int` - The number of Transformer blocks to use.

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

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

Note

We initialize the weights using the initialization method specified in the `cs336_spring2025_assignment1_basics.pdf`. This approach ensures compliance with the given specifications for weight initialization.

```python
def init_weights(m:nn.Module):
    if isinstance(m, Linear):
        std = math.sqrt(2.0 / (m.in_features + m.out_features))
        nn.init.trunc_normal_(m.weight, mean=0.0, std=std, a=-3*std, b=3*std)
        if m.bias is not None:
            nn.init.zeros_(m.bias)
    elif isinstance(m, Embedding):
        nn.init.trunc_normal_(m.weight, mean=0.0, std=1.0, a=-3.0, b=3.0)
    elif isinstance(m, RMSNorm):
        nn.init.ones_(m.weight)

# uv run pytest -k test_transformer_lm
class TransformerLM(nn.Module):
    """ Transformer Language Model """
    def __init__(
        self,
        vocab_size:int,
        context_length:int,
        d_model:int,
        num_layers:int,
        num_heads:int,
        d_ff:int,
        rope_theta:float,
        ffn_type:str = 'swiglu',
        use_post_norm:bool = False,
        remove_rmsnorm:bool = False,
        remove_rope:bool = False,
    ) -> None:
        super().__init__()
        self.token_embeddings = Embedding(vocab_size, d_model)
        self.layers = nn.ModuleList([
            TransformerBlock(
                d_model, num_heads, d_ff, context_length, rope_theta,
                ffn_type=ffn_type,
                use_post_norm=use_post_norm,
                remove_rmsnorm=remove_rmsnorm,
                remove_rope=remove_rope,
            )
            for _ in range(num_layers)
        ])
        if remove_rmsnorm:
            self.ln_final = nn.Identity()
        else:
            self.ln_final = RMSNorm(d_model)
        self.lm_head = Linear(d_model, vocab_size)
        self.max_seq_len = context_length
        self.apply(init_weights)

    def forward(self, token_ids:torch.Tensor) -> torch.Tensor:
        # [batch, seq_len] -> [batch, seq_len, vocab_size]
        seq_len = token_ids.size(1)
        assert seq_len <= self.max_seq_len, "Sequence length exceeds model capacity"
        x = self.token_embeddings(token_ids)
        for layer in self.layers:
            x = layer(x)
        x = self.ln_final(x)
        logits = self.lm_head(x)
        return logits
```

3.11 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.

Answer:

Our GPT-2 XL model would have approximately 2.13 billion trainable parameters

$$\begin{aligned}\text{Total Parameters} &= \underbrace{50,257 \times 1,600}_{\text{embeddings}} + 48 \times (\underbrace{4 \times 1,600^2}_{\text{attention}} + \underbrace{3 \times 1,600 \times 6,400}_{\text{SwiGLU}}) + \underbrace{1,600 \times 50,257}_{\text{output head}} \\ &= 2,126,902,400\end{aligned}$$

Assuming single-precision floating point (32 bits = 4 bytes per parameter), the model would require approximately 7.92 GB of memory to load.

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

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

Answer:

Per layer matrix multiplies are listed in the following Table 2. Total per layer: $90,596,966,400 \approx 90.6$B FLOPs, and for 48 layers: $4,348,654,387,200 \approx 4.35$ trillion FLOPs, and for final projection: $2 \times 1,024 \times 1,600 \times 50,257 = 164,682,137,600 \approx 164.68$B FLOPs[1], so the total is $4,514,221,260,800 \approx 4.51$ trillion FLOPs.

Table 2: Matrix multiplications and FLOP calculations for GPT-2 XL forward pass

Operation	**Mathematical Formula**	**FLOP Formula**	**GPT-2 XL Calculation**
Q projection	$Q = W_Q \cdot X$	$2 \times T \times d_{\text{model}} \times d_{\text{model}}$	$2 \times 1024 \times 1600 \times 1600 = 5.24$B
K projection	$K = W_K \cdot X$	$2 \times T \times d_{\text{model}} \times d_{\text{model}}$	$2 \times 1024 \times 1600 \times 1600 = 5.24$B
V projection	$V = W_V \cdot X$	$2 \times T \times d_{\text{model}} \times d_{\text{model}}$	$2 \times 1024 \times 1600 \times 1600 = 5.24$B
Output projection	$O = W_O \cdot A$	$2 \times T \times d_{\text{model}} \times d_{\text{model}}$	$2 \times 1024 \times 1600 \times 1600 = 5.24$B
Attention scores	$S = Q \cdot K^T / \sqrt{d_k}$	$2 \times h \times T \times d_k \times T$	$2 \times 25 \times 1024 \times 64 \times 1024 = 3.36$B
Attention output	$A = S \cdot V$	$2 \times h \times T \times T \times d_k$	$2 \times 25 \times 1024 \times 1024 \times 64 = 3.36$B
SwiGLU W_1	$G_1 = W_1 \cdot X$	$2 \times T \times d_{\text{model}} \times d_{\text{ff}}$	$2 \times 1024 \times 1600 \times 6400 = 20.97$B
SwiGLU W_2	$Y = W_2 \cdot G_{\text{activated}}$	$2 \times T \times d_{\text{ff}} \times d_{\text{model}}$	$2 \times 1024 \times 6400 \times 1600 = 20.97$B
SwiGLU W_3	$G_3 = W_3 \cdot X$	$2 \times T \times d_{\text{model}} \times d_{\text{ff}}$	$2 \times 1024 \times 1600 \times 6400 = 20.97$B
		Total per layer:	**90.6B FLOPs**
		48 layers total:	**4.35T FLOPs**
Final projection	logits $= W_{\text{lm}} \cdot H$	$2 \times T \times d_{\text{model}} \times V$	$2 \times 1024 \times 1600 \times 50257 = 164.68$B
		Grand total:	**4.51T FLOPs**

Note: where T = sequence length, d_{model} = model dimension, d_{ff} = feed-forward dimension, d_k = head dimension, h = number of heads, V = vocabulary size.

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

Deliverable: A one-to-two sentence response.

Answer:

[1]Thanks to xhs user GHOST for pointing out my original calculation error where I mistakenly wrote "50,527" instead of "50,257".

From the table Table 2 above, we can just use one formula to calculate the total FLOPs:

$$\begin{aligned}\text{Total FLOPs} &= N_{\text{layer}}(\underbrace{4 \times 2 \times Td_{\text{model}}^2}_{\text{projection}} + \underbrace{2 \times 2 \times hd_k \times T^2}_{\text{attention}} + \underbrace{3 \times 2 \times Td_{\text{model}}d_{\text{ff}}}_{\text{feed-forward}}) + \underbrace{2 \times Td_{\text{model}} \times V}_{\text{output head}} \\ &= 2Td_{\text{model}}\left(N_{\text{layer}}\left(4d_{\text{model}} + 2T + 3d_{\text{ff}}\right) + V\right)\end{aligned}$$

The SwiGLU feed-forward networks consume the most FLOPs, accounting for approximately 67.0% of the total, while attention mechanisms account for about 29.4% of the computation, and embedding operations account for the remaining 3.6%.

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

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

Answer:

GPT-2 model configurations and FLOP breakdown are shown in Table 3. As model size increases, the feed-forward networks consume a larger proportion of FLOPs while embedding operations become negligible, with attention maintaining roughly constant proportional cost across all model sizes. (See Appendix A.3 for calculation code)

Table 3: GPT-2 model configurations and computational requirements comparison

Model	Configuration	FLOP Breakdown (per forward pass)
GPT-2 Small	$N_{\text{layer}} = 12$ $d_{\text{model}} = 768$ $h = 12$ $d_{\text{ff}} = 3072$	Attention: 27.6% (0.10T FLOPs) FFN: 49.8% (0.17T FLOPs) Embeddings: 22.6% (0.08T FLOPs) **Total: 0.35T FLOPs**
GPT-2 Medium	$N_{\text{layer}} = 24$ $d_{\text{model}} = 1024$ $h = 16$ $d_{\text{ff}} = 4096$	Attention: 29.9% (0.31T FLOPs) FFN: 59.9% (0.62T FLOPs) Embeddings: 10.2% (0.11T FLOPs) **Total: 1.03T FLOPs**
GPT-2 Large	$N_{\text{layer}} = 36$ $d_{\text{model}} = 1280$ $h = 20$ $d_{\text{ff}} = 5120$	Attention: 30.0% (0.68T FLOPs) FFN: 64.2% (1.45T FLOPs) Embeddings: 5.8% (0.13T FLOPs) **Total: 2.26T FLOPs**
GPT-2 XL	$N_{\text{layer}} = 48$ $d_{\text{model}} = 1600$ $h = 25$ $d_{\text{ff}} = 6400$	Attention: 29.4% (1.33T FLOPs) FFN: 66.9% (3.02T FLOPs) Embeddings: 3.6% (0.16T FLOPs) **Total: 4.51T FLOPs**

Note: All calculations assume sequence length $T = 1024$. N_{layer} = number of transformer layers, d_{model} = model dimension, h = number of attention heads, $d_{\text{ff}} = 4 \times d_{\text{model}}$ following Radford et al. (2019).

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

Deliverable: A one-to-two sentence response.

Answer:

Increasing context length from 1,024 to 16,384 (16× increase) results in total FLOPs increasing by approximately 33× to 149.5 trillion FLOPs, with attention computation scaling quadratically (74× increase) while feed-forward computation scales linearly (16× increase). This dramatically shifts the computational balance: attention now dominates with 65.9% of total FLOPs (compared to 29.4% at shorter sequences), feed-forward drops to 32.3% (from 66.9%), and embeddings become negligible at 1.8% (from 3.6%), making attention the primary computational bottleneck at very long sequences.

4 Training a Transformer LM

4.1 Problem (cross_entropy): Implement Cross entropy (2 points)

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

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

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

Answer:

The essence of implementing `cross_entropy` is computing `log_softmax`. We can derive the formula for `log_softmax` . Let `logits` be l, then:

$$\begin{aligned}\log \text{softmax}(l_i) &= \log \frac{\exp(l_i)}{\sum_j \exp(l_j)} \\ &= l_i - \log \sum_j \exp(l_j)\end{aligned}$$

Therefore, the following two computation methods are equivalent:

```
logits = model(input)  # B,T,V
log_softmax1 = logits.log_softmax(dim=-1)  # B,T,V
log_softmax2 = logits - logits.logsumexp(dim=-1, keepdim=True)  # B,T,V
```

By using the `logsumexp` function, we avoid explicitly computing `exp` and `log` operations, thereby improving numerical stability and reducing computational overhead. This approach directly computes log softmax, then uses `gather` operations to obtain the log probabilities for target classes, and finally takes the negative value to get the cross-entropy loss.

```
# uv run pytest -k test_cross_entropy
def cross_entropy(inputs:torch.Tensor, targets:torch.Tensor) -> torch.Tensor:
    # (batch_size, num_classes), (batch_size,) -> scalar
    inputs = inputs - inputs.max(dim=-1, keepdim=True).values  # for numerical stability
    log_probs = inputs - torch.logsumexp(inputs, dim=-1, keepdim=True)
    return -log_probs.gather(dim=-1, index=targets.unsqueeze(-1)).squeeze(-1).mean()
```

4.2 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.

Answer:

For learning rates 1e1 and 1e2, the loss decays faster (1e2 > 1e1 > 1), while for 1e3, the loss diverges, indicating that too high a learning rate can lead to instability in training. This demonstrates that while higher learning rates can accelerate convergence up to a point, there exists a critical threshold beyond which training becomes unstable and the optimization process fails.

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

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

```
(cs336-basics) [root:assignment1-basics]$ uv run pytest -k test_adamw
======================== test session starts ========================
platform linux -- Python 3.13.7, pytest-8.4.1, pluggy-1.6.0
rootdir: standford-cs336/assignment1-basics
configfile: pyproject.toml
plugins: jaxtyping-0.3.2, hydra-core-1.3.2
collected 48 items / 47 deselected / 1 selected

tests/test_optimizer.py::test_adamw PASSED

================= 1 passed, 47 deselected in 4.36s =================
```

Answer:

AdamW is a variant of the Adam optimizer that decouples weight decay from the gradient-based update. The key difference is that weight decay is applied directly to the parameters rather than being added to the gradients.

The AdamW update rule is:

$$
\begin{aligned}
g_t &= \nabla_\theta \mathcal{L}(\theta_{t-1}) \\
\theta_t &= \theta_{t-1} - \alpha\lambda\theta_{t-1} \\
m_t &= \beta_1 m_{t-1} + (1-\beta_1) g_t \\
v_t &= \beta_2 v_{t-1} + (1-\beta_2) g_t^2 \\
\hat{m}_t &= m_t / (1-\beta_1^t) \\
\hat{v}_t &= v_t / (1-\beta_2^t) \\
\theta_t &= \theta_{t-1} - \alpha \hat{m}_t / (\sqrt{\hat{v}_t} + \epsilon)
\end{aligned}
$$

where m_t and v_t are the first and second moment estimates, λ is the weight decay coefficient, and the weight decay term $\lambda\theta_{t-1}$ is applied directly to the parameters. Code is available at optimizer.py#L30.

4.4 Problem (adamw_accounting): Resource accounting for training with AdamW (2 points)

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

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

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

- Transformer block:
  - RMSNorm(s)
  - Multi-head self-attention sublayer: QKV projections, Q^TK matrix multiply, softmax, weighted sum of values, output projection.
  - Position-wise feed-forward: W_1 matrix multiply, SiLU, W_2 matrix multiply
- Final RMSNorm
- Output embedding
- Cross-entropy on logits

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

Answer:

Parameters: (See Appendix A.1 for verification of this formula)

$$P = \underset{\text{embed}}{V \cdot d} + L \cdot (\underset{\text{proj}}{4d^2} + \underset{\text{ffn}}{12d^2} + \underset{\text{norm}}{2d}) + d + \underset{\text{out}}{d \cdot V} = 16Ld^2 + (2L + 1 + 2V)d$$

Gradients: $G = P$ (same as parameters)
Optimizer State: $O = 2P$ (first and second moments in AdamW)
Activations:

$$\begin{aligned} A =& L \cdot (\underset{\text{RMSNorm}}{2BTd} + \underset{\text{QKV proj}}{3BTd} + \underset{Q^TK}{BHTT} + \underset{\text{softmax}}{BHTT} + \underset{\text{weighted } V'}{BHTd_v} + \underset{\text{out proj}}{BTd} + \underset{\text{W1,SiLU}}{2BTd_{\text{ff}}} + \underset{\text{W2}}{BTd}) \\ &+ \underset{\text{final RMSNorm}}{BTd} + \underset{\text{output emb}}{BTV} + \underset{\text{cross-entropy}}{BTV} \\ =& L \cdot (16BTd + 2BHT^2) + BT(d + 2V) \\ =& BT(L(16d + 2HT) + d + 2V) \end{aligned}$$

Total Memory: $M = P + G + O + A = 4P + A$
(where $V =$ `vocab_size`, $d =$ `d_model`, $L =$ `num_layers`, $T =$ `context_length`, $H =$ `num_heads`, $B =$ `batch_size`, $d_{\text{ff}} = 4d$, and $d_v = d/H$)

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

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

Answer:

For GPT-2 XL: $V = 50257$, $d = 1600$, $L = 48$, $T = 1024$, $H = 25$
Parameters: $P = 16 \times 48 \times 1600^2 + (2 \times 48 + 1 + 2 \times 50257) \times 1600 \approx 2.13\text{B}$ parameters
Memory expression: $M = 4P \times 4 + B \times 1024 \times (48 \times (16 \times 1600 + 2 \times 25 \times 1024) + 1600 + 2 \times 50257) \times 4$
$M = 31.71\text{GB} + B \times 14.45\text{GB}$
Maximum batch size: $\frac{80 - 31.71}{14.45} \approx 3.34$ (See Appendix A.2 for calculation code)

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

Deliverable: An algebraic expression, with a brief justification.

Answer:

AdamW requires approximately $16P$ FLOPs per step. The detailed breakdown is shown in Table 4.
Assumptions:
a. We count only the optimizer work (gradient computation via forward+backward is not included)
b. Scalar constants are precomputed once per step (e.g., $\alpha\lambda$, $(1 - \beta_1^t)^{-1}$, $(1 - \beta_2^t)^{-1}$)
c. Each multiply/add/subtract/divide/sqrt operation counts as 1 FLOP
d. P is the total number of model parameters

(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. (2023). 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.

Answer:

Assumptions: use our previous defined parameters for our N=2.13B GPT-2 XL:
Forward FLOPs per token: 4.51T FLOPs/1024 tokens = 4.404B FLOPs/token (approximately $2N$),
Backward FLOPs per token: 2× 4.404B = 8.808B FLOPs/token (approximately $4N$),
Total FLOPs per token: 13.212B FLOPs/token (approximately $6N$)
Total tokens: $D = 400,000 \times 1024 \times 1024 = 419,430,400,000$ tokens
Total training FLOPs budget: $13.212B * 419.43B \approx 5.54 \times 10^{21}$ FLOPs (approximately $6ND$)
With 50% MFU on A100: $19.5 \times 0.5 = 9.75$ teraFLOP/s effective throughput
Time required: $(5.54 \times 10^{21})/(9.75 \times 10^{12}) \approx 5.68 \times 10^8$ seconds ≈ 157778 hours ≈ 6574 days ≈ 18 years

Table 4: FLOP breakdown for AdamW optimizer per training step

Operation	Mathematical Formula	FLOPs/P	Description
Weight decay	$\theta \leftarrow \theta - \alpha\lambda\theta$	2	1 mul + 1 sub
First moment	$m_t = \beta_1 m_{t-1} + (1-\beta_1) g_t$	3	2 mul + 1 add
Gradient square	g_t^2	1	1 mul
Second moment	$v_t = \beta_2 v_{t-1} + (1-\beta_2) g_t^2$	3	2 mul + 1 add
Bias correction (m)	$\hat{m}_t = m_t / (1-\beta_1^t)$	1	1 mul
Bias correction (v)	$\hat{v}_t = v_t / (1-\beta_2^t)$	1	1 mul
Square root	$\sqrt{\hat{v}_t}$	1	1 sqrt
Add epsilon	$\sqrt{\hat{v}_t} + \epsilon$	1	1 add
Division	$\hat{m}_t / (\sqrt{\hat{v}_t} + \epsilon)$	1	1 div
Scale by lr	$\alpha \cdot \hat{m}_t / (\sqrt{\hat{v}_t} + \epsilon)$	1	1 mul
Parameter update	$\theta \leftarrow \theta - \alpha \hat{m}_t / (\sqrt{\hat{v}_t} + \epsilon)$	1	1 sub
		Total per parameter:	**16 FLOPs**
		Total for model:	**16P FLOPs**

4.5 Problem (`learning_rate_schedule`): Implement cosine learning rate schedule with warmup (2 points)

(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 $\alpha_t = \alpha_{\min}$.

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

Answer:

Code is available at optimizer.py#L183.

```
(cs336-basics) [root:assignment1-basics]$ uv run pytest -k test_get_lr_cosine_schedule
======================== test session starts ========================
platform linux -- Python 3.13.7, pytest-8.4.1, pluggy-1.6.0
rootdir: standford-cs336/assignment1-basics
configfile: pyproject.toml
plugins: jaxtyping-0.3.2, hydra-core-1.3.2
collected 48 items / 47 deselected / 1 selected

tests/test_optimizer.py::test_get_lr_cosine_schedule PASSED

================== 1 passed, 47 deselected in 4.31s ==================
```

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

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

Answer:

Code is available at nn_utils.py#L27.

```
(cs336-basics) [root:assignment1-basics]$ uv run pytest -k test_gradient_clipping
======================== test session starts ========================
platform linux -- Python 3.13.7, pytest-8.4.1, pluggy-1.6.0
rootdir: standford-cs336/assignment1-basics
configfile: pyproject.toml
plugins: jaxtyping-0.3.2, hydra-core-1.3.2
collected 48 items / 47 deselected / 1 selected
```

```
tests/test_nn_utils.py::test_gradient_clipping PASSED

================== 1 passed, 47 deselected in 2.81s ==================
```

5 Training Loop

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

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

Answer:

Code is available at data.py#L6-L25.

```
(cs336-basics) [root:assignment1-basics]$ uv run pytest -k test_get_batch
======================== test session starts ========================
platform linux -- Python 3.13.7, pytest-8.4.1, pluggy-1.6.0
rootdir: standford-cs336/assignment1-basics
configfile: pyproject.toml
plugins: jaxtyping-0.3.2, hydra-core-1.3.2
collected 48 items / 47 deselected / 1 selected

tests/test_data.py::test_get_batch PASSED

================== 1 passed, 47 deselected in 3.09s ==================
```

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

Implement the following two functions to load and save checkpoints:

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

This function expects the following parameters:

- `model`: `torch.nn.Module`
- `optimizer`: `torch.optim.Optimizer`
- `iteration`: `int`
- `out`: `str | os.PathLike | typing.BinaryIO | typing.IO[bytes]`

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

This function expects the following parameters:

- `src`: `str | os.PathLike | typing.BinaryIO | typing.IO[bytes]`
- `model`: `torch.nn.Module`
- `optimizer`: `torch.optim.Optimizer`

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

Answer:

Code is available at checkpoint.py. P.S. I also consider there might be `list` of optimizers(when implement the muon optimizer).

```
(cs336-basics) [root:assignment1-basics]$ uv run pytest -k test_checkpointing
======================== test session starts ========================
platform linux -- Python 3.13.7, pytest-8.4.1, pluggy-1.6.0
rootdir: standford-cs336/assignment1-basics
configfile: pyproject.toml
plugins: jaxtyping-0.3.2, hydra-core-1.3.2
collected 48 items / 47 deselected / 1 selected

tests/test_serialization.py::test_checkpointing PASSED

================== 1 passed, 47 deselected in 4.43s ==================
```

5.3 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).

Answer:

I implemented two complete training scripts: `train.py` and `train_muon.py`, both integrating Hydra for configuration management and Weights & Biases for experiment tracking.

- `train.py`: Standard training loop with AdamW optimizer, cosine learning rate scheduling with warmup, and gradient clipping, memory-efficient data loading, model checkpointing, and evaluation.
- `train_muon.py`: Extended version supporting the Muon optimizer for comparison studies

The scripts include comprehensive evaluation loops, proper device handling (CPU/GPU), and robust error handling. Differences between the two scripts can be viewed with `diff -u train.py train_muon.py`.

Code is available at: train.py train_muon.py

6 Generating Text

6.1 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 $x_1 ... t$ and sample a completion until you hit an `<|endoftext|>` token).
- Allow the user to control the maximum number of generated tokens.
- Given a desired temperature value, apply softmax temperature scaling to the predicted next-word distributions before sampling.

- Top-p sampling (Holtzman et al. (2019); also referred to as nucleus sampling), given a user-specified threshold value.

Answer:

I implemented a comprehensive text generation function supporting temperature scaling, top-p sampling, prompt completion, and maximum length control. Additionally, I implemented KV-cache optimization inspired by karpathy/nano-llama31 to significantly speed up autoregressive generation by caching attention key-value pairs from previous tokens. Code is available at generate.py.

7 Experiments

7.1 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.

Answer:

I implemented comprehensive experiment tracking infrastructure using Weights & Biases (wandb) integrated with Hydra for configuration management. The logging system tracks training and validation losses, learning rates, gradient norms, and wallclock time (recorded by wandb automatically) across all experiments.

All experiment configurations and results are systematically logged and can be accessed through the wandb dashboard for analysis and comparison. A comprehensive code snippet of the logging setup is provided in Appendix A.5. The whole training code is available at train.py. And all the experiment scripts/logs can be found at scripts and CHANGELOG.md.

7.2 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.

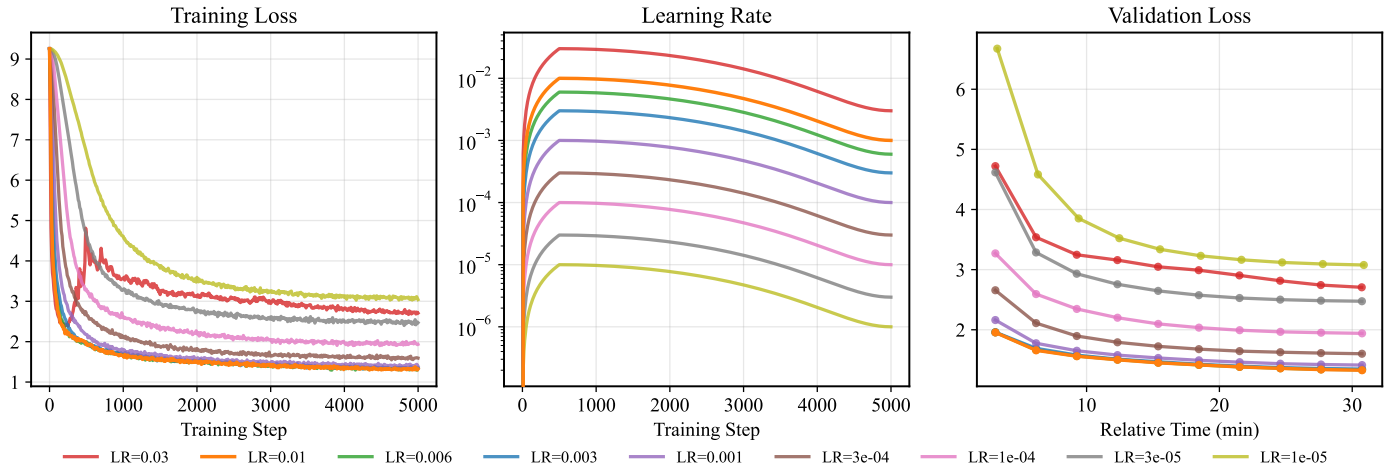
Deliverable: Learning curves of increasing learning rate which include at least one divergent run and an analysis of how this relates to convergence rates.

Answer:

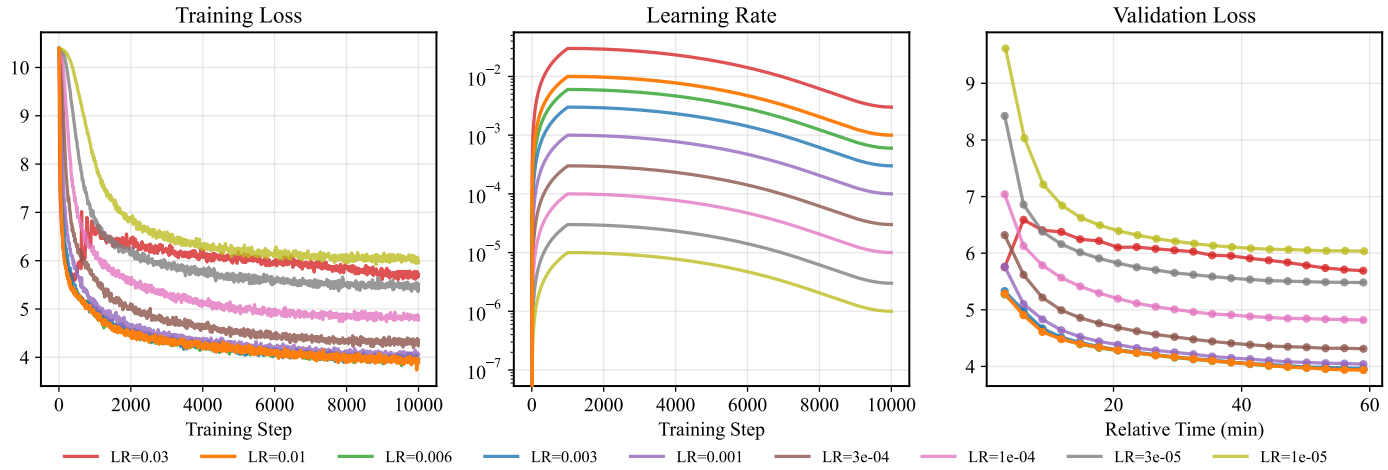
Part (a): I employed a logarithmic grid search across learning rates from 10^{-5} to 3×10^{-2}, testing nine values: [1e-5, 3e-5, 1e-4, 3e-4, 1e-3, 3e-3, 6e-3, 1e-2, 3e-2]. Each experiment used identical model architecture and training configuration with cosine learning rate scheduling, 500-step warmup, and AdamW optimizer ($\beta_1 = 0.9, \beta_2 = 0.95$, weight decay=0.01). Due to GPU memory constraints (48GB), TinyStories experiments used batch size 256 for 5,000 iterations, while OpenWebText experiments used batch size 128 for 10,000 iterations to maintain total 327,680,000 tokens processed (context length=256). The learning curves are shown in Figure 1.

Key findings from both experiments:

- ($\leq 3 \times 10^{-4}$): Stable convergence but inefficient training



(a) Learning rate experiments on TinyStories dataset



(b) Learning rate experiments on OpenWebText dataset

Figure 1: Learning rate experiments showing training loss (left), learning rate schedule (middle), and validation loss over time (right) for both datasets. The highlighted orange line represents LR=0.01, which achieved the best performance across both TinyStories and OpenWebText. Complete experiment logs/configs are available on the W&B: Report.

- (1×10^{-3} to 1×10^{-2}): Best balance of convergence speed and stability

- 1×10^{-2} achieved val/loss of 1.327, surpassing the target of 1.45 on TinyStories

- 3×10^{-2} caused training collapse during warmup at step 480

- OpenWebText experiments mirrored TinyStories results, with 1×10^{-2} yielding the best val/loss of 3.93

Part (b): The "edge of stability" hypothesis suggests optimal learning rates exist just below the divergence threshold. My experiments support this principle:

- 3×10^{-2} caused immediate instability during warmup

- 1×10^{-2} (approximately 3× below divergence) achieved the best validation performance

7.3 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.

Answer:

I conducted comprehensive batch size experiments on TinyStories, testing values from 1 to 1024: [1, 2, 4, 8, 16, 32, 64, 128, 256, 512, 768, 1024]. All experiments used learning rate 1×10^{-3} (selected for better balance of convergence speed and stability), training for 10,000 iterations with the same total token count of 327,680,000 to ensure fair comparison. The learning curves are shown in Figure 2.

Key Findings:

- Small (1-16): Exhibited high gradient noise, and increased wall-clock training time despite processing the same number of tokens

- Optimal (128): Achieved the lowest val/loss efficiently (8.75 minutes for 10k iterations)

- Near (32,64,256,512,768): Showed competitive performance, faster per-step computation(256-768)

- Large (1024): Training collapsed, which exceeds memory capacity

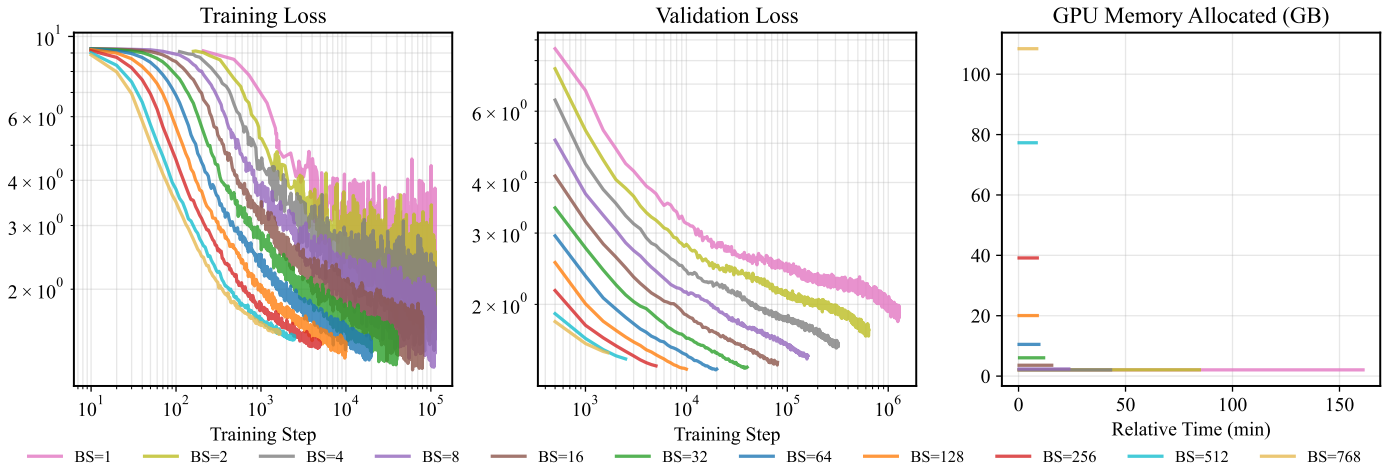


Figure 2: Batch size experiments on TinyStories showing training loss vs. steps (left, log-log scale), validation loss vs. steps (middle, log-log scale), and GPU memory vs. time (right). Batch size 128 (orange line) achieved the best validation performance with only 8.75 minutes of training time for 10,000 iterations. Complete experiment logs are available on W&B: Report.

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

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

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

Answer:

Here is a sample of text generated by the best model(val/loss=1.327) from Learning rate experiments on the TinyStories dataset, using max generation length of 1000, a temperature of 0.6 and top-p sampling with p=0.95:

> Once upon a time, there was a little girl named Lily. She had a doll that she loved very much. The doll was very pretty and expensive. Lily would play with her doll every day and take it with her everywhere she went. One day, Lily and her doll went to the park. They played on the swings and the slide. Lily was having so much fun that she forgot her doll at the park. She forgot about her doll at home. When it was time to go home, Lily's mom called her. Lily was sad because she forgot her doll at the park. She missed her doll and wished she had her doll back. The next day, Lily found her doll at the park and remembered her doll at the park. She ran back to the park and found her doll waiting for her. Lily was so happy to have her doll back and promised to always remember her. Once upon a time, there was a little girl named Sue. She loved to play with her toys and eat yummy food. One day, Sue saw a big, green avocado on the table. She wanted to eat it, but she knew she should ask her mom first. Sue went to her mom and said, "Mom, can I eat the avocado?" Her mom smiled and said, "Yes, you can eat the avocado, but be careful not to break it." Sue was very happy and started to eat the avocado. But then, something unexpected happened. The avocado started to talk! It said, "Please don't eat me!" Sue was very surprised. She did not eat the avocado, but she still loved it very much. Sue and the avocado became best friends. They played and lived happily ever after. Once upon a time, there was a little boy named Tim. He had a big, and he was very tired. He wanted to eat, but he could not. He tried to eat, but he was sad. He had a bad feeling. One day, a lot. She was so close to the day. She was very tired, but she was in the park. She went to the park. She was happy. One day, but a boy was not available. She was not fit in the next to. She went to the day. She felt very well. She was happy. She was a little bit.

Once upon a time, there was a little boy, and she had a time. The little girl's house to play. She liked to play area to play. The park. She would go. The little boy's birthdays time to go, but she got a little girl's birthday party. At the park. The park. But, but the park. But, but she could not. She was in the park. She was in the park. The park. Sue's birthday party. Sue's birthday party. Sue's birthday. Sue's birthday to her birthday. At the park. Sue's birthday party. Once there was a big party time, and she had a new season. Sue's birthday, so her mom and her mom's things. Sue's birthdays value her mom's house. Sue, her mom said, her things, and they were not. One day, Sue. She was her house. Sue's birthday. Sue's home. Sue's birthday to go to go to go. Sue's home. Sue's house, but her house. Sue went to go. Sue's room was in a clear that she would be careful not be careful with her mom and she did not to have her mom said, "Mom said, it. Her mom was her mom said, "Please, but also had to remind her mom would not to be careful. She took her mom. She was not. Sue's room for too much better than before she could not. One day, she did not want to eat, "Hi, but not hungry. She said, but not hungry. She wanted her mom, but she would not eat, and she would not. She would open a lot of her mom was always be sad. One day, she was in the whole wide-evia. Avi-evid-so-b at the day. She knew that day. The sun, but the same food was a little boy's mom made a lot of her mom and her mom said thank you, and she thought about the day. Lily and she was a big, and she felt hungry. Lily's day, and her mom said, so much. Lily, but she was a big, and her mom's food to eat, but the day, but the day. Lily's food. The end, and she was not very hungry. Lily was full of a little, and her small, and her favorite food

Comment:

The generated text shows mixed quality. The first two stories (Lily's doll and Sue's talking avocado) are coherent with proper grammar and logical plots. After 200 tokens, quality degrades significantly with repetitive phrases ("Sue's birthday party"), incomplete sentences ("He had a big, and he was very tired"), and nonsense ("wide-evia. Avi-evid-so-b").

Key factors affecting quality:

The model uses a 256-token context window during training. Beyond this length, it loses track of earlier content, causing repetition and fragmentation. Maybe the TinyStories dataset contains mostly short stories (<300 tokens), so the model hasn't learned long-form generation. Additionally, with only 22M parameters, the model has limited capacity for complex linguistic patterns and long-range dependencies.

7.5 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.

Answer:

I conducted comprehensive experiments removing RMSNorm from the Transformer architecture across three different learning rates: 1×10^{-2}, 1×10^{-3}, and 3×10^{-4}. The learning curves are shown in Figure 3.

Commentary: RMSNorm is essential for training stability in Transformer architectures. Without it, training diverges catastrophically at the optimal learning rate (1×10^{-2}), with losses exploding to over 10^{26} within hundreds of steps. While lower learning rates (1×10^{-3} and 3×10^{-4}) provide some stability, they converge to significantly worse solutions compared to models with RMSNorm. These results demonstrate that RMSNorm enables both efficient training with higher learning rates and better final model quality by maintaining well-conditioned gradients throughout training.

7.6 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.

Answer:

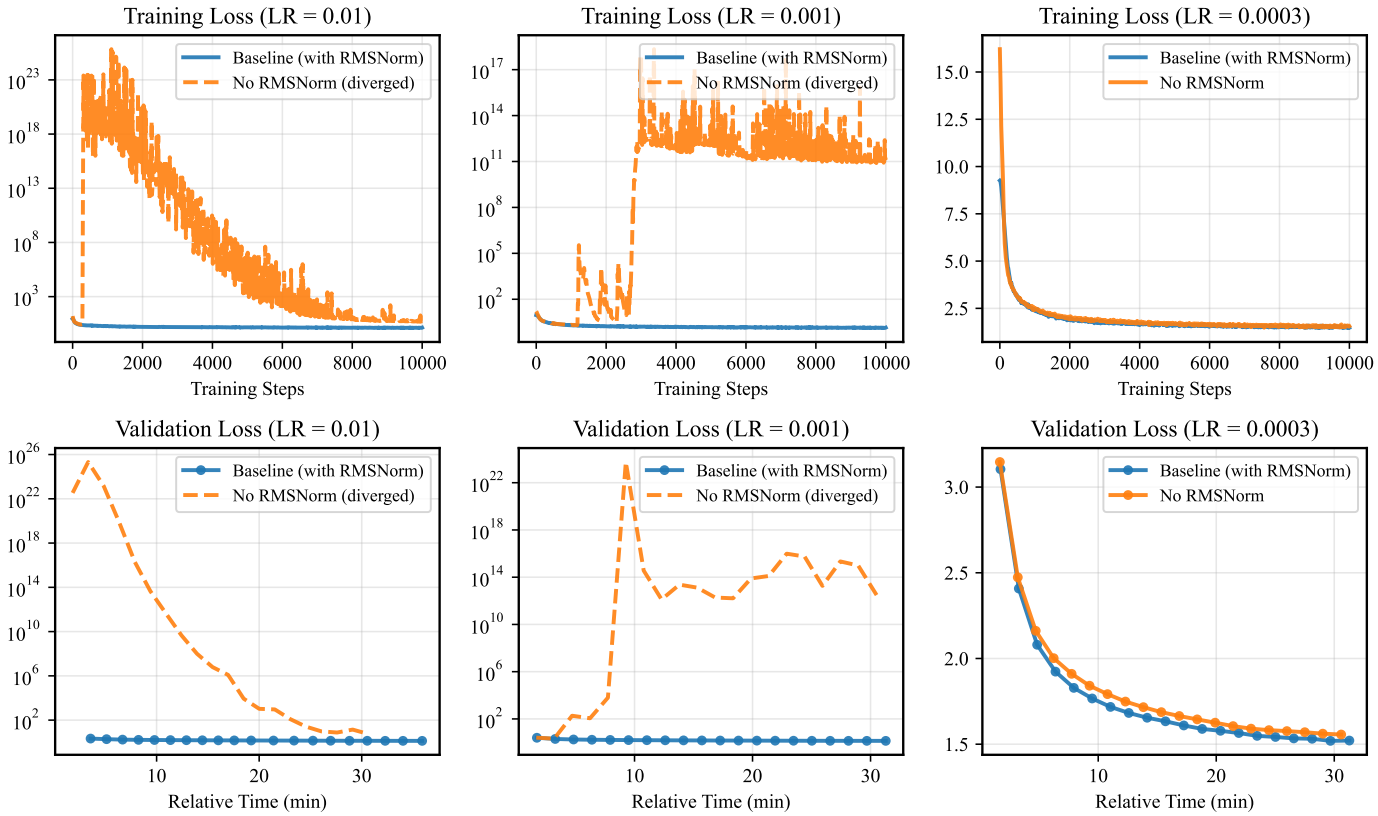


Figure 3: RMSNorm ablation experiments comparing baseline (with RMSNorm) vs. no RMSNorm across different learning rates. Top row shows training loss vs. steps, bottom row shows validation loss vs. relative time.

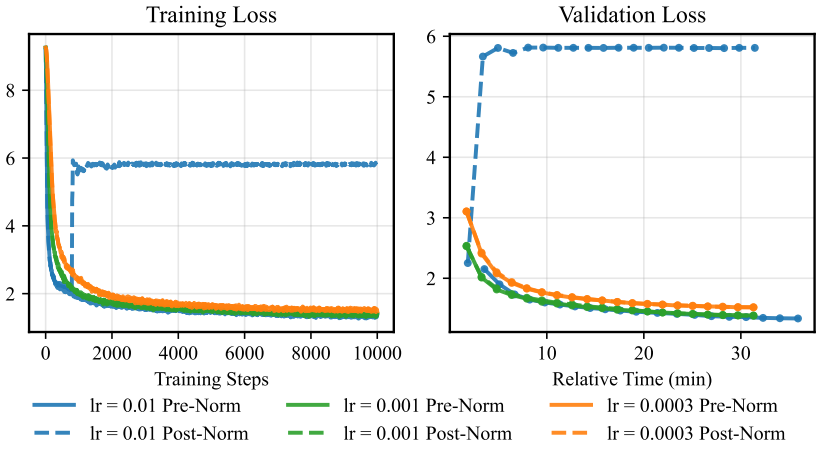


Figure 4: Training and validation loss for pre-norm vs. post-norm across multiple learning rates.

The learning curves are shown in Figure 4.

Commentary: The pre-norm transformer remained stable across both learning rates, whereas the post-norm variant using the optimal learning rate diverged quickly, showing that pre-norm is more stable than post-norm.

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

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

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

Answer:

The learning curves are shown in Figure 5.

Commentary: RoPE maintained lower training and validation losses at every learning rate, while NoPE plateaued more quickly. This suggests that removing positional encodings limits convergence speed and

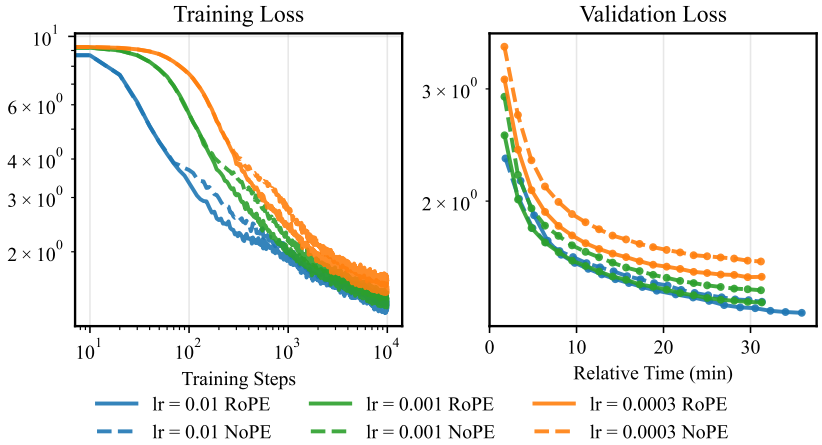


Figure 5: Training and validation loss for RoPE vs. NoPE across multiple learning rates.

final quality, even with conservative learning rate. However, the stability of NoPE at optimal learning rate 0.01 indicates that positional encodings are not strictly necessary for training stability, but they do enhance performance.

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

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

Deliverable: A few sentences discussing your findings.

Answer:

The learning curves [2] are shown in Figure 6.

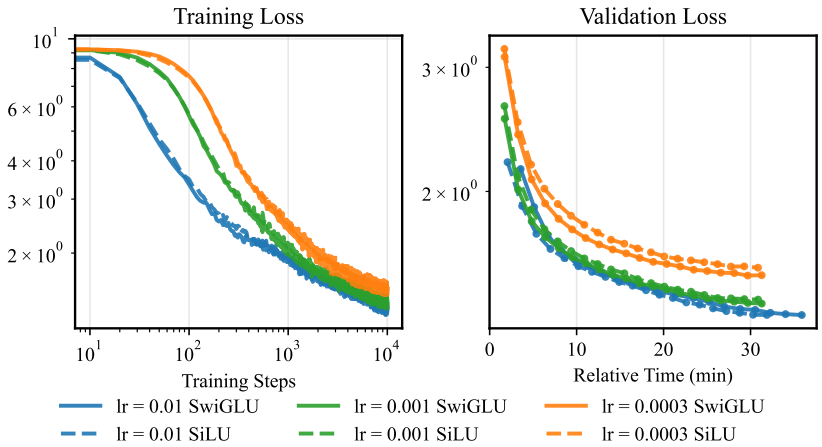


Figure 6: Training and validation loss for SwiGLU vs. SiLU across multiple learning rates.

Commentary: SiLU edges out SwiGLU only at the optimal learning rate (0.01), delivering slightly lower losses. Across the stable learning-rate regime (0.001 and 0.0003), SwiGLU converges faster and finishes with better validation loss, indicating that the gating boost helps when optimization pressure is lower.

7.9 Problem (`main_experiment`): Experiment on OWT (2 points) (3 H100 hrs)

Train your language model on OpenWebText with the same model architecture and total training iterations as TinyStories. How well does this model do?

Deliverable: A learning curve of your language model on OpenWebText. Describe the difference in losses from TinyStories - how should we interpret these losses?

[2] The ablation experiment logs/configs are available on W&B: Report

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?

Answer:

The learning curves[3] comparing TinyStories and OpenWebText are shown in Figure 7.

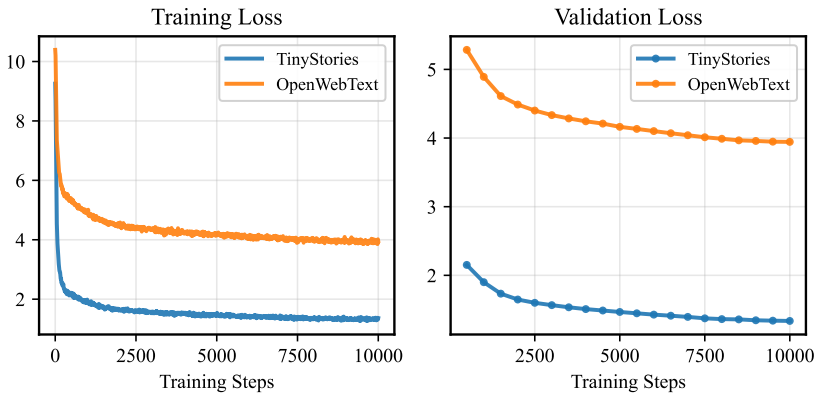


Figure 7: Main experiments comparing TinyStories and OpenWebText datasets.

Loss Difference: The OpenWebText model achieves significantly higher losses compared to TinyStories across both training and validation. The final validation loss for OpenWebText is approximately 3.93, while TinyStories achieves 1.33. This difference reflects the inherent complexity of the datasets: OpenWebText contains diverse, real-world web content with complex vocabulary, grammar, and topics, whereas TinyStories consists of simple, repetitive children's stories with limited vocabulary and predictable structures.

Generated Text: Here is a sample of the first 1000 tokens generated by the OpenWebText model, using temperature 0.6 and top-p sampling with p=0.95:

> The Miami Heat are a team of world champions, and the Heat are the same team that now will have their first team. The Heat are the team that has been known for their depth, but they are the team that will be getting their first team on Saturday. The Heat are the team that has been designated to the Heat in recent years, but that 鈥 not the case. It 鈥 not just the team that is building a roster that will be the NBA 鈥 best-performing team, but it 鈥 also the latest sports that will be the NBA 鈥 first team in the NBA. This team is the team that is building a team that will be in the league. They will be making a team that will be able to compete in the playoffs. The Heat are the team that will be building the team 鈥 best-performing team, and the team will have a chance to win the Heat 鈥 first team in the series. The Heat are the team that will be in the NBA, and they are going to be the team that will be the team that will be the next best team in the series. The Heat are the team that will be the team that will be the team that will be the team that will be the team 鈥 best team. The Heat will have a team that will be the team that will be the team that will be the team that will be the team that will be the team. When you guys are the team that has to play the Heat. They 鈥 e in the Heat, and the Heat have the best team that can beat the Heat in the Bulls and the Heat are the team. They 鈥 e got the team that has the most of which is the team that team, and they. Saints, the Heat fans. I 鈥 Wright W A This The K B K It LSU G. Cous The The The If The R The The It The The It The The The It 鈥 The The The The The The The The The The The We 鈥 The The The The The The The next The The The It 鈥 Lets The The It 鈥 The The The second The The only The The best The next- The The It 鈥 They 鈥 ight- It 鈥 The entire It 鈥 unoby A lot of The next of The other The same-Bened A/fresh The two- 鈥 鈥 he other players 鈥 鈥 think 鈥 he best of the last year-Diamen. the best. the first- the 鈥 he best-t. I think of the game and it 鈥 he most 鈥 he second- playing the other than

Output Fluency: The OpenWebText model exhibits significantly degraded output quality compared to TinyStories. The text demonstrates severe coherence issues, including excessive repetition (e.g., "the team that") and grammatical inconsistencies. The garbled characters (e.g., `It鈥` for `It's`) are a known encoding artifact, which is further discussed in the Appendix A.6.

Four key factors explain this quality degradation:

1. Vocabulary Size: OpenWebText uses a vocabulary of 32,000 tokens, far larger than TinyStories' 10,000. The 22M-parameter model lacks the capacity to effectively model this larger linguistic space. An experiment reducing OpenWebText's vocabulary to 10,000 confirmed this, yielding better results.

[3]The main experiment logs/configs are available on W&B: Report.

2. Training Data: OpenWebText is scraped from the web and contains significant noise, whereas TinyStories is synthetically generated and clean. This difference in data quality directly impacts the model's ability to learn coherent language patterns.

3. Context Window: The model was trained with a 256-token context window. When asked to generate text far exceeding this length (1000 tokens), the model loses track of long-range dependencies, leading to degenerative repetition. This is analogous to long-horizon forecasting in RNNs, where predictions can collapse to a constant value.

4. Training Inefficiency: With an identical token budget, the model trained on OpenWebText sees fewer examples per concept/task due to the dataset's diversity. In contrast, TinyStories' simpler, repetitive structure allows for more robust pattern learning within the same compute budget.

7.10 Problem (`leaderboard`): Leaderboard (6 points) (10 H100 hrs)

You will train a model under the leaderboard rules above with the goal of minimizing the validation loss of your language model within 1.5 H100-hour.

Deliverable: The final validation loss that was recorded, an associated learning curve that clearly shows a wallclock-time x-axis that is less than 1.5 hours and a description of what you did. We expect a leaderboard submission to beat at least the naive baseline of a 5.0 loss. Submit to the leaderboard here: `https://github.com/stanford-cs336/assignment1-basics-leaderboard`.

Answer:

Final Result: I achieved a validation loss of **3.335** at the 90-minute mark, ranking **24th** on the global leaderboard[4]. The learning curves are shown in Figure 8.

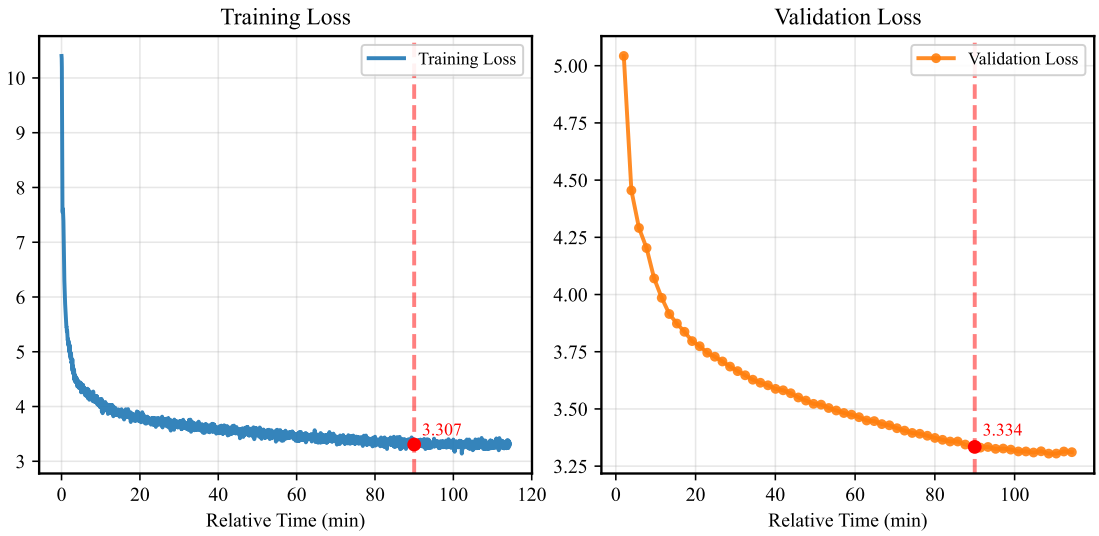


Figure 8: Leaderboard submission learning curves showing training loss (left) and validation loss (right) vs. relative time in minutes. The red dashed line marks the 90-minute budget limit, with the final validation loss of 3.335 annotated.

Key Modifications:

Within the constraints of 80GB GPU memory and 1.5-hour wall-clock time using only the OpenWebText training set, I implemented the following modifications:

- **Model Size:** Scaled up to GPT-2 Small size (d_model=768, d_ff=2048, n_head=12, n_layer=12)
- **QK Normalization:** Added RMSNorm to query and key projections before RoPE for stability
- **Muon Optimizer:** Adopted the Muon optimizer (inspired by modded-nanogpt) for efficient optimization
- **Weight Tying:** Tied the token embedding and output projection weights
- **LR Schedule:** Used cosine annealing from 1e-2 to 0 over 30,000 iterations with 2,000 warmup steps

Ablation Study:

Table 5 summarizes the impact of each modification on validation loss:

Analysis:

[4]Leaderboard: `https://github.com/stanford-cs336/assignment1-basics-leaderboard`

Table 5: Ablation study showing the impact of different modifications on validation loss.

Configuration	**Val Loss - Min**	**Δ Loss**
Model Size		
Baseline (d_model=512, n_layer=4)	3.695	-
GPT-2 Small (d_model=768, n_layer=12)	3.335	-0.360
GPT-2 Small + (n_layer=16)	3.264	-0.071
Optimizer & Learning Rate (GPT-2 Small)		
AdamW (lr=1e-3)	3.482	-
AdamW (lr=1e-2)	3.436	-0.046
AdamW (lr=1e-3) + Muon (lr=3e-2)	3.454	-0.028
AdamW (lr=1e-2) + Muon (lr=3e-2)	3.427	-0.055
Weight Tying		
Baseline w/o weight tying	3.753	-
Baseline w/ weight tying	3.695	-0.058
GPT-2 Small w/o weight tying	3.427	-
GPT-2 Small w/ weight tying	3.362	-0.065

The most significant improvement ($\Delta = -0.360$) came from scaling to GPT-2 Small architecture, demonstrating that model capacity is crucial given sufficient compute budget. Weight tying consistently improved performance across model sizes ($\Delta \approx -0.06$), reducing parameter count while maintaining expressiveness. The Muon optimizer combined with higher learning rates showed modest gains over AdamW alone. QK normalization provided training stability without degrading final performance. Further scaling to 16 layers showed additional gains but exceeded the 90-minute time budget.

References

Aakanksha Chowdhery, Sharan Narang, Jacob Devlin, Maarten Bosma, Gaurav Mishra, Adam Roberts, Paul Barham, Hyung Won Chung, Charles Sutton, Sebastian Gehrmann, et al. Palm: Scaling language modeling with pathways. *Journal of Machine Learning Research*, 24(240):1–113, 2023.

Jordan Hoffmann, Sebastian Borgeaud, Arthur Mensch, Elena Buchatskaya, Trevor Cai, Eliza Rutherford, Diego de Las Casas, Lisa Anne Hendricks, Johannes Welbl, Aidan Clark, et al. Training compute-optimal large language models. *arXiv preprint arXiv:2203.15556*, 2022.

Ari Holtzman, Jan Buys, Li Du, Maxwell Forbes, and Yejin Choi. The curious case of neural text degeneration. *arXiv preprint arXiv:1904.09751*, 2019.

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. *arXiv preprint arXiv:2001.08361*, 2020.

Alec Radford, Jeffrey Wu, Rewon Child, David Luan, Dario Amodei, Ilya Sutskever, et al. Language models are unsupervised multitask learners. *OpenAI blog*, 1(8):9, 2019.

Ashish Vaswani, Noam Shazeer, Niki Parmar, Jakob Uszkoreit, Llion Jones, Aidan N Gomez, Łukasz Kaiser, and Illia Polosukhin. Attention is all you need. *Advances in neural information processing systems*, 30, 2017.

A Appendix

A.1 Parameter Calculation Verification

To verify that our parameter calculation formula is correct, we provide the following verification code that compares the actual parameter count from a PyTorch model (implemented in `cs336_basics/model.py`) with our analytical formula:

Listing 1: Parameter count verification code

```python
from cs336_basics.model import TransformerLM

model_cfg = dict(
    vocab_size=10000,
    context_length=256,
    d_model=512,
    num_layers=4,
    num_heads=16,
    d_ff=1344,
    rope_theta=10000,
)

model = TransformerLM(**model_cfg)
total_params = sum(p.numel() for p in model.parameters())

def compute_model_size_from_scratch(
    vocab_size: int,
    context_length: int,
    d_model: int,
    num_layers: int,
    num_heads: int,
    d_ff: int,
    **kwargs
):
    total_params = 0
    total_params += vocab_size * d_model  # token embedding
    total_params += num_layers * ( # per block
        4 * (d_model * d_model) +  # Q, K, V, O projections
        3 * (d_model * d_ff) +     # SwiGLU layers
        2 * (d_model)              # RMSNorm layers
    )
    total_params += d_model  # final RMSNorm
    total_params += d_model * vocab_size  # output projection
    return total_params

print(f"Total number of parameters: {total_params}")
print("Computed model size from scratch:", compute_model_size_from_scratch(**model_cfg))
```

```
Total number of parameters: 22696448
Computed model size from scratch: 22696448
```

A.2 Memory Calculation Verification

```python
def compute_model_size(
    vocab_size: int,
    context_length: int,
    d_model: int,
    num_layers: int,
    num_heads: int,
    d_ff: int,
    **kwargs
):
    total_params = 0
    total_params += vocab_size * d_model  # token embedding
    total_params += num_layers * ( # per block
        4 * (d_model * d_model) +  # Q, K, V, O projections
        3 * (d_model * d_ff) +     # SwiGLU layers
        2 * (d_model)              # RMSNorm layers
    )
    total_params += d_model  # final RMSNorm
    total_params += d_model * vocab_size  # output projection
    return total_params

def compute_memory(vocab_size: int,
    context_length: int,
    d_model: int,
    num_layers: int,
    num_heads: int,
    d_ff: int,
    batch_size: int,
    **kwargs
) -> float:
    param_size = compute_model_size(vocab_size, context_length, d_model, num_layers, num_heads,
        d_ff)
    optimizer_size = param_size * 3  # Adam optimizer
    activation_size = batch_size * context_length *(
        num_layers * (
            8 * d_model + 2 * d_ff + 2 * num_heads * context_length
        ) + d_model + 2 * vocab_size
    )
    memory_bytes = (param_size + optimizer_size + activation_size) * 4  # assuming 4 bytes per
        parameter (float32)
    return memory_bytes / (1024 ** 3)  # convert to GB


if __name__ == "__main__":
    # GPT-2 XL
    cfg = dict(
        vocab_size=50257,
        context_length=1024,
        d_model=1600,
        num_layers=48,
        num_heads=25,
        d_ff=6400,
        rope_theta=10000,
        batch_size=3.34,
    )
    model_size = compute_model_size(**cfg)
    memory_gb = compute_memory(**cfg)
    print(f"Model size: {model_size/(1e9):.2f}B parameters")
    print(f"Estimated memory usage during training: {memory_gb:.2f} GB")
```

```
Model size: 2.13B parameters
Estimated memory usage during training: 79.97 GB
```

A.3 GPT-2 FLOP Breakdown Code

To verify our FLOP breakdown for GPT-2 models, we provide the following code that computes the FLOP counts for attention, feed-forward, and language modeling head components based on the GPT-2 architecture:

```
def compute_gpt2_flops(d_model:int, n_layer:int, T:int=1024) -> dict:
    d_ff = 4 * d_model
    vocab_size = 50_257
    attn_flops = 4 * 2 * T * d_model * d_model + 2 * 2 * T * T * d_model
    ff_flops = 3 * 2 * T * d_model * d_ff
    lm_head_flops = 2 * T * d_model * vocab_size
    total_flops = n_layer * (attn_flops + ff_flops) + lm_head_flops
    return {
        "attention": n_layer * attn_flops / total_flops,
        "feed-forward": n_layer * ff_flops / total_flops,
        "lm-head": lm_head_flops / total_flops,
        "total": total_flops / 1e12, # in TFLOPs
    }, {
        "attention": n_layer * attn_flops / 1e12,
        "feed-forward": n_layer * ff_flops / 1e12,
        "lm-head": lm_head_flops / 1e12,
        "total": total_flops / 1e12,
    }

print("GPT-2 small", compute_gpt2_flops(768, 12))
print("GPT-2 medium", compute_gpt2_flops(1024, 24))
print("GPT-2 large", compute_gpt2_flops(1280, 36))
print("GPT-2 XL", compute_gpt2_flops(1600, 48))
print("GPT-2 XL", compute_gpt2_flops(1600, 48, 16384))
```

```
GPT-2 small ({'attention': 0.276396942718713, 'feed-forward': 0.49751449689368343, 'lm-head':
    0.22608856038760353, 'total': 0.349630365696}, {'attention': 0.09663676416, 'feed-forward':
    0.173946175488, 'lm-head': 0.079047426048, 'total': 0.349630365696})
GPT-2 medium ({'attention': 0.29932707434661254, 'feed-forward': 0.5986541486932251, 'lm-head':
    0.1020187769601624, 'total': 1.033109504}, {'attention': 0.309237645312, 'feed-forward':
    0.618475290624, 'lm-head': 0.105396568064, 'total': 1.033109504})
GPT-2 large ({'attention': 0.2996151010432329, 'feed-forward': 0.6420323593783562, 'lm-head':
    0.05835253957841083, 'total': 2.2577545216}, {'attention': 0.67645734912, 'feed-forward':
    1.4495514624, 'lm-head': 0.13174571008, 'total': 2.2577545216})
GPT-2 XL ({'attention': 0.29440647731422626, 'feed-forward': 0.6691056302596051, 'lm-head':
    0.036487892426168594, 'total': 4.5133365248}, {'attention': 1.3287555072, 'feed-forward':
    3.01989888, 'lm-head': 0.1646821376, 'total': 4.5133365248})
GPT-2 XL ({'attention': 0.65922723665908, 'feed-forward': 0.32315060620543135, 'lm-head':
    0.017622157135488675, 'total': 149.5227957248}, {'attention': 98.5694994432, 'feed-forward':
    48.31838208, 'lm-head': 2.6349142016, 'total': 149.5227957248})
```

A.4 Comparison of GPT-2 Parameter Counts with original paper

To verify our GPT-2 parameter count calculations, we provide the following comparison with the original GPT-2 paper Radford et al. (2019):

Table 6: Comparison of our GPT-2 calculations with values reported in the original GPT-2 paper Radford et al. (2019).

Model	Our Calculation	Paper Reported	Difference	Relative Error
GPT-2 Small	124M	117M	+7M	5.98%
GPT-2 Medium	354M	345M	+9M	2.61%
GPT-2 Large	772M	762M	+10M	1.31%
GPT-2 XL	1555M	1542M	+13M	0.84%

```
def compute_gpt2_params(d_model:int, num_layers:int) -> float:
    vocab_size = 50_257
    d_ff = 4 * d_model
    total_params = 0
    total_params += vocab_size * d_model  # token embedding
    total_params += num_layers * ( # per block
        4 * (d_model * d_model) +  # Q, K, V, O projections
        # 3 * (d_model * d_ff) +     # our SwiGLU layers
```

```
        2 * (d_model * d_ff) +     # original gpt-2 layers
        2 * (d_model)              # RMSNorm layers
    )
    total_params += d_model  # final RMSNorm
    # total_params += d_model * vocab_size  # output projection tied with input embedding
    return total_params


print("GPT-2 small", compute_gpt2_params(768, 12) / 1e6, "M params")
print("GPT-2 medium", compute_gpt2_params(1024, 24) / 1e6, "M params")
print("GPT-2 large", compute_gpt2_params(1280, 36) / 1e6, "M params")
print("GPT-2 XL", compute_gpt2_params(1600, 48) / 1e9, "B params")
```

```
GPT-2 small 123.551232 M params
GPT-2 medium 353.503232 M params
GPT-2 large 772.2112 M params
GPT-2 XL 1.5551264 B params
```

A.5 Comprehensive Logging Code Snippet

```
# Training script with comprehensive logging infrastructure
import hydra
from cs336_basics.logger import Logger
from cs336_basics.config import TrainConfig

@hydra.main(config_path="conf", config_name="train_config", version_base=None)
def main(cfg: TrainConfig) -> None:
    # Initialize logger with Hydra configuration
    logger = Logger(cfg)
    output_dir = Path(HydraConfig.get().runtime.output_dir)

    # ... model and data setup ...

    # Training loop with comprehensive logging
    for it in tqdm(range(start_iter, cfg.training.max_iters), desc="Training"):
        # ... forward/backward pass ...

        # Training metrics logging
        if it % cfg.training.log_interval == 0:
            ent = compute_entropy_chunked(logits).mean()
            logger.log_metrics({
                'train/loss': loss.item(),
                'train/ppl': loss.exp().item(),
                'train/lr': lr,
                'train/entropy': ent.item(),
                'train/grad_norm': grad_norm
            }, step=it)

        # Validation logging
        if it % cfg.training.eval_interval == 0:
            metrics = evaluate(model, val_data, cfg, device)
            logger.log_metrics({
                'val/loss': metrics['val/loss'],
                'val/ppl': metrics['val/ppl'],
                'val/entropy': metrics['val/entropy']
            }, step=it)

    # Log generated text samples
    generated_output = generate_text(model, tokenizer, ...)
    logger.log_text("Generated Text", generated_output, step=cfg.training.max_iters)

    logger.close()
    # Save final configuration
    OmegaConf.save(cfg, output_dir / 'config.yaml')

@torch.no_grad()
def evaluate(model, data, cfg, device):
    """Evaluation with entropy and perplexity tracking."""
```

```
    # ... evaluation loop ...
    return {
        'val/loss': mean_loss,
        'val/ppl': np.exp(mean_loss),
        'val/entropy': np.mean(entropies)
    }
```

A.6 Analysis of the Error encodings 鈥檚

To investigate the source of the garbled characters like 鈥檚 in the generated text, I analyzed the raw training data (`owt_train.txt`), the tokenizer's behavior, and the processed binary data (`train.bin`). My goal was to understand the frequency and representation of different apostrophe styles.

The Python snippet below confirms the root cause: the sequence 鈥檚 is the result of incorrectly decoding a UTF-8 encoded string ’ s (using a right single quotation mark) with a different encoding, such as GBK.

```
>>> "NBA’ s".encode("utf-8").decode("utf-8")
'NBA’ s'
>>> "NBA’ s".encode("utf-8").decode("gbk")
'NBA鈥檚'
```

To quantify the presence of this error in the dataset, I ran scripts to count the occurrences of different 's variants both in the raw text file and in the tokenized training data[5].

```
(cs336-basics) [root:assignment1-basics]$ python data_utils/count_word_from_train_bin.py
============================================================
Token Sequence Occurrence Counting Tool
============================================================
Loading tokenizer: hf_tokenizer/openwebtext-32k/tokenizer.json
'鈥檚' -> token_ids: [164, 187, 99, 161, 104, 205]
''s' -> token_ids: [382]
'’ s' -> token_ids: [286, 83]
' ‘s' -> token_ids: [4719, 83]
------------------------------------------------------------
File size: 5.08 GB
Total tokens: 2,727,044,684
Target token sequences:
  '鈥檚': [164, 187, 99, 161, 104, 205]
  ''s': [382]
  '’ s': [286, 83]
  ' ‘s': [4719, 83]
Start counting...
Counting progress: 100%|                                    | 2727044684/2727044684 [03:48<00:00,
    11959960.05tokens/s]

Counting results:
----------------------------------------
'鈥檚' (sequence: [164, 187, 99, 161, 104, 205]): 2 times
''s' (sequence: [382]): 8,062,975 times
'’ s' (sequence: [286, 83]): 11,253,531 times
' ‘s' (sequence: [4719, 83]): 24,540 times

Total tokens: 2,727,044,684
Processing time: 228.02 seconds
============================================================
(cs336-basics) [root:assignment1-basics]$ bash data_utils/count_word_from_train_txt.sh
Counting character occurrences in data/owt_train.txt...
File size: 12G
Occurrences of '鈥檚'
2
Occurrences of ''s'
8131046
Occurrences of '’ s'
11277977
Occurrences of ' ‘s'
71423
```

[5]The code for this analysis can be found in `count_word_from_train_txt.sh` and `count_word_from_train_bin.py`.

```
Counting completed!
```

Some Findings:

The results from both the raw text and the tokenized data are consistent. They reveal that the dataset is dominated by standard apostrophes, with `’ s` (right single quote) being the most frequent variant (over 11 million times), followed by `'s` (straight quote, over 8 million times).

Crucially, the garbled sequence 鈥檚 appears only twice in the entire 12GB dataset. This confirms that the issue is not widespread data corruption but rather a rare artifact, likely from the original web scraping process.

The paradox that the model reproduces this extremely rare pattern invites some interesting speculation. Here are a couple of my hypotheses:

1. Sensitivity to Low-Probability Patterns: This behavior might indicate that the model is highly sensitive to every pattern it has learned, even those with vanishingly low probabilities. During a degenerative state where the model loses long-range context, it might latch onto any learned sequence that fits the immediate context. The fact that it can recall and repeat a pattern seen only twice suggests that no training example is truly forgotten, merely assigned a low probability.

2. Preference for Longer Token Sequences: An alternative hypothesis is that the model may develop a preference for longer token sequences to fulfill certain grammatical roles. The garbled 鈥檚 is tokenized into a sequence of six tokens, whereas the correct `'s` and `’ s` are tokenized into one and two tokens, respectively. It's conceivable that when the model is struggling to form a possessive, it defaults to a longer, more ”descriptive” token sequence it has learned, even if it's incorrect. This is a fascinating thought, as it might offer a glimpse into why techniques like Chain-of-Thought (CoT) are effective—they guide the model to solve problems through longer, more deliberate sequences of tokens.