CSE599-O Assignment 1: Building a Transformer LM (Part 1)

Version 1.0
CSE 599-O Staff
Fall 2025

Acknowledgment: This assignment is adapted from Assignment 1 of Stanford CS336 (Spring 2025).

1 Assignment Overview

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

What you will implement

1. Transformer language model (LM) (§3)
2. The cross-entropy loss function and the AdamW optimizer (§4)
3. The training loop, with support for serializing and loading model and optimizer state (§**??**)

What you will run

1. Run the tokenizer on the dataset to convert it into a sequence of integer IDs.
2. Train a Transformer LM on the TinyStories dataset.
3. Generate samples and evaluate perplexity using the trained Transformer LM.

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

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

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

[1] See `http://pytorch.org/docs/stable/nn.html#containers` for a full list.

Statement on AI tools Prompting LLMs such as ChatGPT is permitted for low-level programming questions or high-level conceptual questions about language models, but using it directly to solve the problem is not encouraged.

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

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

`https://github.com/uw-syfi/assignment1-basics`

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

1. cse599o_basics/*: This is where you write your code. Note that there's no code in here-you can do whatever you want from scratch!
2. adapters.py: There is a set of functionality that your code must have. For each piece of functionality (e.g., scaled dot product attention), fill out its implementation (e.g., run_scaled_dot_product_attention) by simply invoking your code. Note: your changes to `adapters.py` should not contain any substantive logic; this is glue code.
3. test_*.py: This contains all the tests that you must pass (e.g., test_scaled_dot_product_attention), which will invoke the hooks defined in `adapters.py`. Don't edit the test files.

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

- code.zip: Include all the code you have written, along with the scripts needed to run the training loop.
- An output figure of the learning curve.
- Answers to analytical questions.

Grading (Total: 100% credits)

- **Test cases (70%)**: 20 tests in total, each worth 3.5 credits.
  - 18 tests for the transformer implementation. (Part 1 of HW1; not necessary to access a GPU)
  - 2 tests for data loading and checkpointing.
  - Note: Tokenization-related tests are not counted toward the grade.
- **Training and outputs (15%)**: Scripts to run the training loop, generate the learning curve, and produce sample text outputs.
- **Profiling (15%)**: Profiling your implementation and answer analytical questions.

Where to get datasets This assignment will use two pre-processed datasets: TinyStories Eldan and Li (2023) and OpenWebText Gokaslan et al. (2019). Both datasets are single, large plaintext files. You can download these files with the commands inside the README.md.

2 Byte-Pair Encoding (BPE) Tokenizer

In the first part of the assignment, we provide a tokenizer based on the off-the-shelf OpenAI tiktoken (https://github.com/openai/tiktoken), which already passes the relevant tests. You can verify this by running:

```
$ uv run pytest
```

Expected result: All tests in `tests/test_tokenizer.py` pass, while tests from other files (e.g., `test_model.py`) fail with `NotImplementedError`.

You can use this tokenizer directly and apply your customizations. Alternatively, you can choose to train and implement your own BPE tokenizer, but it must pass all the required tests. Note: Tokenization-related test items do not count toward the grade.

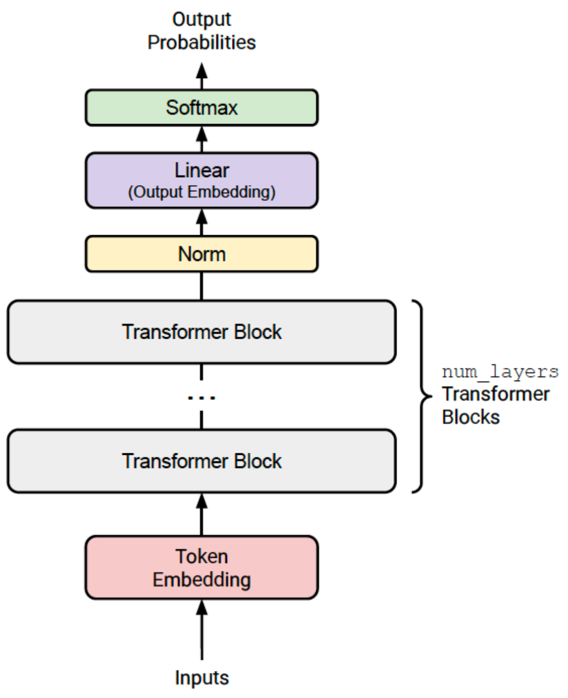

Figure 1: An overview of our Transformer language model.

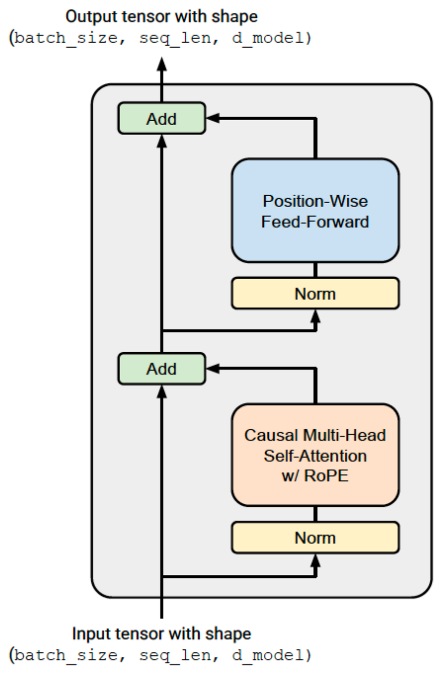

Figure 2: A pre-norm Transformer block.

3 Transformer Language Model Architecture

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

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

3.1 Transformer LM

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

3.1.1 Token Embeddings

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

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

3.1.2 Pre-norm Transformer Block

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

3.2 Output Normalization and Embedding

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

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

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

3.3 Remark: Batching, Einsum and Efficient Computation

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

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

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

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

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

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

We strongly recommend learning and using einsum notation for the class. Students who have not been exposed to einsum notation before should use einops (docs here), and students who are already comfortable with einops should learn the more general einx (here) [2]. Both packages are already installed in the environment we've supplied.

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

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

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

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

## Rearrange an image tensor for mixing across all pixels
channels_last_flat = channels_last.view(
    -1, channels_last.size(1) * channels_last.size(2), channels_last.size(3)
)
channels_first_flat = channels_last_flat.transpose(1, 2)
channels_first_flat_transformed = channels_first_flat @ B.T
channels_last_flat_transformed = channels_first_flat_transformed.transpose(1, 2)
channels_last_transformed =
↪   channels_last_flat_transformed.view(*channels_last.shape)
```

Instead, using `einops`:

```
height = width = 32
```

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

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

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

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

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

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

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

3.3.1 Mathematical Notation and Memory Ordering

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

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

for row-major $W \in \mathbb{R}^{d_{\text{out}} \times d_{\text{in}}}$ and row-vector $x \in \mathbb{R}^{1 \times d_{\text{in}}}$.
In linear algebra it's generally more common to use column vectors, where linear transformations look like

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

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

3.4 Basic Building Blocks: Linear and Embedding Modules

3.4.1 Parameter Initialization

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

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

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

3.4.2 Linear Module

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

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

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

Problem (linear): Implementing the linear module

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

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

    in_features: int
```

```
        Final dimension of the input
    out_features: int
        Final dimension of the output
    device: torch.device | None = None
        Device to store the parameters on
    dtype: torch.dtype | None = None
        Data type of the parameters
    """

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

Make sure to: - subclass `nn.Module` - call the superclass constructor - construct and store your parameter as `W` (not `WT̂`) for memory ordering reasons, putting it in an `nn.Parameter` - of course, don’ t use `nn.Linear` or `nn.functional.linear`
For initializations, use the settings from above along with `torch.nn.init.trunc_normal_` to initialize the weights.
To test your `Linear` module, implement the test adapter at `adapters.run_linear`. The adapter should load the given weights into your `Linear` module. You can use `Module.load_state_dict` for this purpose. Then, run:

```
uv run pytest -k test_linear
```

3.4.3 Embedding Module

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

Problem (embedding): Implement the embedding module

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

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

    num_embeddings: int
        Size of the vocabulary
    embedding_dim: int
```

```
        Dimension of the embedding vectors, i.e., d_model
    device: torch.device | None = None
        Device to store the parameters on
    dtype: torch.dtype | None = None
        Data type of the parameters
    """

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

Make sure to: - subclass `nn.Module` - call the superclass constructor - initialize your embedding matrix as a `nn.Parameter` - store the embedding matrix with the d_{model} being the final dimension - of course, don' t use `nn.Embedding` or `nn.functional.embedding`
Again, use the settings from above for initialization, and use `torch.nn.init.trunc_normal_` to initialize the weights.
To test your implementation, implement the test adapter at `adapters.run_embedding`. Then, run:

```
uv run pytest -k test_embedding
```

3.5 Pre-Norm Transformer Block

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

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

3.5.1 Root Mean Square Layer Normalization

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

$$\mathrm{RMSNorm}\left(a_i\right) = \frac{a_i}{\mathrm{RMS}(a)} g_i \tag{4}$$

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

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

```
in_dtype = x.dtype
x = x.to(torch.float32)
# Your code here performing RMSNorm
result = ...
# Return the result in the original dtype
return result.to(in_dtype)
```

Problem (rmsnorm): Root Mean Square Layer Normalization

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   # Device to store the parameters on
    dtype: torch.dtype  | None    # Data type of the parameters

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

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

```
uv run pytest -k test_rmsnorm
```

3.5.2 Position-Wise Feed-Forward Network

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

However, modern language models tend to incorporate two main changes compared to this original design: they use another activation function and employ a gating mechanism. Specifically, we will implement the ”SwiGLU” activation function adopted in LLMs like Llama 3 [Grattafiori et al., 2024] and Qwen 2.5

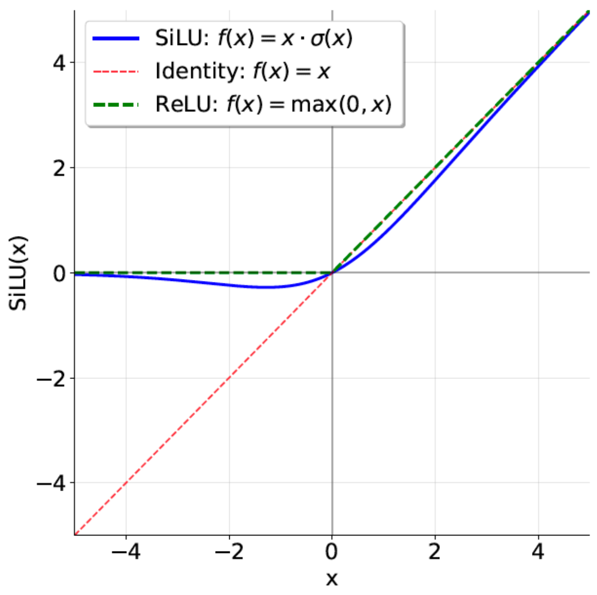

Figure 3: Comparing the SiLU (aka Swish) and ReLU activation functions.

[Yang et al., 2024], which combines the SiLU (often called Swish) activation with a gating mechanism called a Gated Linear Unit (GLU). We will also omit the bias terms sometimes used in linear layers, following most modern LLMs since PaLM [Chowdhery et al., 2022] and LLaMA [Touvron et al., 2023].

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

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

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

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

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

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

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

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

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

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

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

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

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

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

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

3.5.3 Relative Positional Embeddings

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

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

Thus we get the full rotation matrix

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

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

Problem (rope): Implement RoPE

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.

    Args:
        theta: Theta value for RoPE.
        d_k: Dimension of query/key vectors (should be even).
        max_seq_len: Maximum sequence length that will be inputted.
        device: torch.device | None. Device to store the buffers on.
    """
    ...

def forward(self, x: torch.Tensor, token_positions: torch.Tensor) -> torch.Tensor:
    """
    Apply RoPE to an input tensor of shape (... , seq_len, d_k) and
    return a tensor of the same shape.

    Notes:
        - Accept x with an arbitrary number of batch dimensions.
        - token_positions has shape (... , seq_len) and gives absolute
          positions per token along the sequence dimension.
        - Use token_positions to slice (precomputed) cos/sin tensors
          along the sequence dimension.
    """
    ...
```

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

3.5.4 Scaled Dot-Product Attention

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

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

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

Problem (softmax): Implement softmax

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

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

We can now define the Attention operation mathematically as follows:

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

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

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

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

Problem (scaled_dot_product_attention): Implement scaled dot-product attention

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

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

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

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

3.5.5 Causal Multi-Head Self-Attention

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

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

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

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

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

Here, the learnable parameters are $W_Q \in \mathbb{R}^{hd_k \times d_{\text{model}}}$, $W_K \in \mathbb{R}^{hd_k \times d_{\text{model}}}$, $W_V \in \mathbb{R}^{hd_v \times d_{\text{model}}}$, and $W_O \in \mathbb{R}^{d_{\text{model}} \times hd_v}$. Since the Q s, K, and V s are sliced in the multi-head attention operation, we can think of W_Q, W_K and W_V as being separated for each head along the output dimension. When you have this working, you should be computing the key, value, and query projections in a total of three matrix multiplies [3].

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

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

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

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
```

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

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
```

3.6 The Full Transformer LM

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

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

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

Problem (transformer_block): Implement the Transformer block

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
```

Deliverable: Transformer block code that passes the provided tests.

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

Problem (transformer_lm): Implementing the Transformer LM

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
```

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

4 Training a Transformer LM

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

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

4.1 Cross-entropy loss

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

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

(Note that a single forward pass in the Transformer yields $p_\theta(x_{i+1} \mid x_{1:i})$ for all $i = 1, \ldots, m$. .) In particular, the Transformer computes logits $o_i \in \mathbb{R}^{vocab\_size}$ for each position i, which results in: 6

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

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

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

Problem (cross_entropy): Implement Cross Entropy

Deliverable: Write a function to compute the cross entropy loss, which takes in predicted logits o_i and targets x_{i+1}, and computes the cross entropy

$$\ell_i = -\log \text{softmax}(o_i)[x_{i+1}].$$

Your function should handle the following:

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

To test your implementation, implement `adapters.run_cross_entropy`, then run:

```
uv run pytest -k test_cross_entropy
```

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

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

4.2 The SGD Optimizer

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

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

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

4.2.1 Implementing SGD in PyTorch

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

```
def __init__(self, params, ... ):
    """
    Initialize your optimizer.

    Arguments:
        params: iterable
            A collection of parameters to optimize, or parameter groups
            (for applying different hyperparameters to different parts of the model).
        ...
```

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

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

```
            Additional arguments depending on the optimizer
            (e.g., learning rate is common).
        """
        # Make sure to call the base class constructor, passing params
        # and a dictionary of hyperparameters (keys are string names).
        super().__init__(params, defaults)

    def step(self):
        """
        Make one update of the parameters.

        This is called after the backward pass in training, so gradients
        are available for each parameter.

        For each parameter tensor p:
            - Access its gradient in p.grad (if it exists).
            - Update p.data in-place based on p.grad.
        """
        for group in self.param_groups:
            for p in group['params']:
                if p.grad is None:
                    continue
                # Example update rule (to be implemented for your optimizer)
                p.data = p.data - group['lr'] * p.grad
```

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

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

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

```
from collections.abc import Callable, Iterable
from typing import Optional
import torch
import math
class SGD(torch.optim.Optimizer):
    def __init__(self, params, lr=1e-3):
        if lr < 0:
            raise ValueError(f"Invalid learning rate: {lr}")
        defaults = {"lr": lr}
        super().__init__(params, defaults)
    def step(self, closure: Optional[Callable] = None):
        loss = None if closure is None else closure()
        for group in self.param_groups:
            lr = group["lr"] # Get the learning rate.
```

```
for p in group["params"]:
    if p.grad is None:
        continue
    state = self.state[p] # Get state associated with p.
    t = state.get("t", 0) # Get iteration number from the state, or initial value.
    grad = p.grad.data # Get the gradient of loss with respect to p.
    p.data -= lr / math.sqrt(t + 1) * grad # Update weight tensor in-place.
    state["t"] = t + 1 # Increment iteration number.
return loss
```

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

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

```
weights = torch.nn.Parameter(5 * torch.randn((10, 10)))
opt = SGD([weights], lr=1)
for t in range(100):
    opt.zero_grad() # Reset the gradients for all learnable parameters.
    loss = (weights**2).mean() # Compute a scalar loss value.
    print(loss.cpu().item())
    loss.backward() # Run backward pass, which computes gradients.
    opt.step() # Run optimizer step.
```

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

4.3 AdamW

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

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

Algorithm 1 AdamW Optimizer

```
1:  init($\theta$)                                   ▷ Initialize learnable parameters
2:  $m \leftarrow 0$                                 ▷ Initial value of the first moment vector; same shape as $\theta$
3:  $v \leftarrow 0$                                 ▷ Initial value of the second moment vector; same shape as $\theta$
4:  for $t = 1, \ldots, T$ do
5:      Sample batch of data $B_t$
6:      $g \leftarrow \nabla_\theta \ell(\theta; B_t)$                ▷ Gradient of the loss at time $t$
7:      $m \leftarrow \beta_1 m + (1 - \beta_1) g$                  ▷ Update first moment estimate
8:      $v \leftarrow \beta_2 v + (1 - \beta_2) g^2$                ▷ Update second moment estimate
9:      $\alpha_t \leftarrow \alpha \frac{\sqrt{1-\beta_2^t}}{1-\beta_1^t}$                ▷ Compute adjusted $\alpha$
10:     $\theta \leftarrow \theta - \alpha_t \frac{m}{\sqrt{v}+\epsilon}$                ▷ Update parameters
11:     $\theta \leftarrow \theta - \alpha \lambda \theta$                      ▷ Apply weight decay
12: end for
```

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

Problem (adamw): Implement AdamW

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 provides 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).

To test your implementation, implement `adapters.get_adamw_cls` and make sure it passes:

```
uv run pytest -k test_adamw
```

4.4 Learning rate scheduling

The value for the learning rate that leads to the quickest decrease in loss often varies during training. In training Transformers, it is typical to use a learning rate schedule, where we start with a bigger learning rate, making quicker updates in the beginning, and slowly decay it to a smaller value as the model trains[4]. In this assignment, we will implement the cosine annealing schedule used to train LLaMA [Touvron et al., 2023].

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

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

The cosine annealing learning rate schedule takes (i) the current iteration t, (ii) the maximum learning rate $\alpha_{\max}$, (iii) the minimum (final) learning rate $\alpha_{\min}$, (iv) the number of warm-up iterations T_w, and (v) the number of cosine annealing iterations T_c. The learning rate at iteration t is defined as:
(Warm-up) If $t < T_w$, then $\alpha_t = \frac{t}{T_w}\alpha_{\max}$.
(Cosine annealing) If $T_w \leq t \leq T_c$, then $\alpha_t = \alpha_{\min} + \frac{1}{2}\left(1 + \cos\left(\frac{t-T_w}{T_c-T_w}\pi\right)\right)(\alpha_{\max} - \alpha_{\min})$.
(Post-annealing) If $t > T_c$, then $\alpha_t = \alpha_{\min}$.

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

Write a function that takes $t, \alpha_{\max}, \alpha_{\min}, T_w, T_c$ 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
```

Deliverable: A function that computes the cosine learning rate schedule with warmup and passes the provided tests.

4.5 Gradient clipping

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

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

Problem (gradient_clipping): Implement gradient clipping

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
```

Deliverable: A gradient clipping function that passes the provided tests.

4.6 Expected Output So far

At this current stage, we expect to pass the following 18 tests.

```
uv run pytest
```

```
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
tests/test_nn_utils.py::test_softmax_matches_pytorch PASSED
tests/test_nn_utils.py::test_cross_entropy PASSED
tests/test_nn_utils.py::test_gradient_clipping PASSED
tests/test_optimizer.py::test_adamw PASSED
tests/test_optimizer.py::test_get_lr_cosine_schedule PASSED
```

References

R. Eldan and Y. Li. Tinystories: How small can language models be and still speak coherent english? arXiv preprint arXiv:2305.07759, 2023.

A. Gokaslan, V. Cohen, E. Pavlick, and S. Tellex. Openwebtext corpus. `https://skylion007.github.io/OpenWebTextCorpus`, 2019.

A. Vaswani, N. Shazeer, N. Parmar, J. Uszkoreit, L. Jones, A. N. Gomez, Ł. Kaiser, and I. Polosukhin. Attention is all you need. Advances in neural information processing systems, 30, 2017.