

DS GA 1011 Assignment 2: Building a Transformer LM

Fall 2026

This assignment is a modified version of Assignment 1 from Stanford CS336,¹ used with permission. All credit for the original development and the original text of this document goes to the Stanford course staff. **If you have questions about the assignment, do NOT contact the Stanford creators; only contact the NYU course staff.**

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) (§2)
2. The cross-entropy loss function and the AdamW optimizer (§3)
3. A batch data loader (§4)

This assignment does not have you build a BPE tokenizer or tokenize raw text yourself. We instead provide a small amount of pre-tokenized TinyStories data and the vocabulary used to produce it that you'll use directly.

What you will run

1. Train your Transformer LM on the provided (pre-tokenized) TinyStories data and watch the loss go down.
2. Generate samples using the trained Transformer LM.
3. Perform ablations of the architecture

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 (which you may use):

- `torch.nn.Parameter`
- Container classes in `torch.nn` (e.g., `Module`, `ModuleList`, `Sequential`, etc.)²
- The `torch.optim.Optimizer` base class

You may use any other PyTorch definitions. When in doubt, consider if using it compromises the “from-scratch” ethos of the assignment.

¹<https://stanford-cs336.github.io/spring2025/>

²See [PyTorch.org/docs/stable/nn.html#containers](https://pytorch.org/docs/stable/nn.html#containers) for a full list.

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

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/gregdurrett/nyu-dsga1011-a2>

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

1. **student/*:** This is where you write your code. Note that there's just one (optional) starter utility here—you can do whatever you want from scratch!
2. **tests/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. **tests/test_*.py:** This contains all the tests that you must pass (e.g., `test_scaled_dot_product_attention`), which will invoke the hooks defined in `adapters.py`. Don't edit the test files.

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

- **writeup.pdf:** Answer all the written questions. Please typeset your responses.
- **code.zip:** Contains all the code you've written. This should be a zipped file containing: (1) `student/` containing your implementation; (2) `tests/` containing just a single file `adapters.py` with your adapter code. **The autograder will check that the tests pass, but you should include all code that you wrote for the assignment, even code that is not tested by the autograder parts.**

Where to get datasets This assignment will use TinyStories [Eldan and Li, 2023] as the training and test dataset. You can download the dataset files with the command inside the `README.md`.

Low-Resource/Downscaling Tip: Init

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

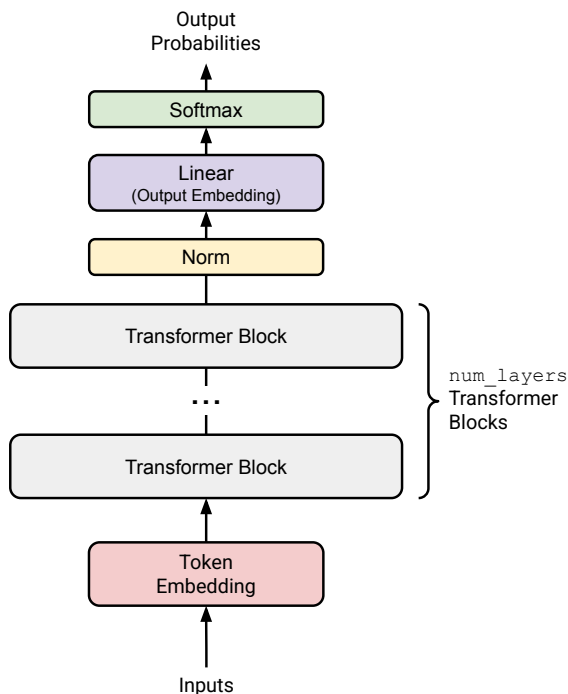


Figure 1: An overview of our Transformer language model.

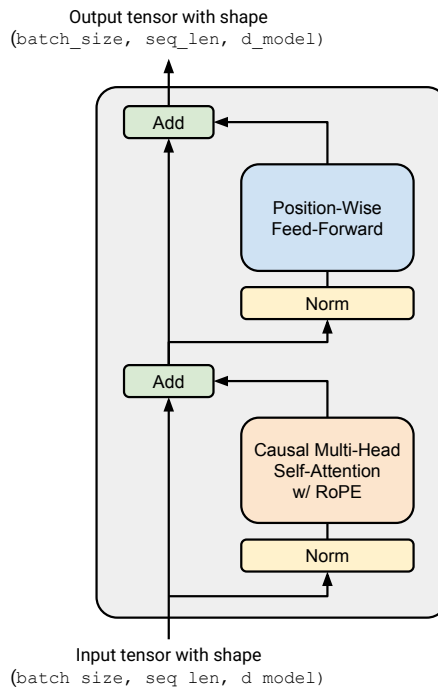


Figure 2: A pre-norm Transformer block.

2 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.

2.1 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.

2.1.1 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)`.

2.1.2 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).

2.2 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 §2.5), 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).

2.3 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).³ Both packages are already installed in the environment we've supplied.

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

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

```
import torch
from einops import rearrange, einsum

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

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

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

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

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

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

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

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

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

Suppose we have a batch of images represented as a tensor of shape (batch, height, width,

³It'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`.

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 $(\text{height} \times \text{width}, \text{height} \times \text{width})$.

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

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

channels_first_flat_transformed = channels_first_flat @ B.T

channels_last_flat_transformed = channels_first_flat_transformed.transpose(1, 2)

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

Instead, using einops:

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

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

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

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

Einsum notation can handle arbitrary input batching dimensions, but also has the key benefit of being *self-documenting*. It's much clearer what the relevant shapes of your input and output tensors are in code that uses einsum notation. For the remaining tensors, you can consider using Tensor type hints, for instance

using the `jaxtyping` library (not specific to Jax).

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

2.3.1 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

$$y = xW^\top, \quad (1)$$

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

$$y = Wx, \quad (2)$$

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.

2.4 Basic Building Blocks: Linear and Embedding Modules

2.4.1 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 significant 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: $\mathcal{N}(\mu = 0, \sigma^2 = 1)$ truncated at $[-3, 3]$
- RMSNorm: 1

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

2.4.2 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:

$$y = Wx. \quad (3)$$

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

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

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

```
def __init__(self, in_features, out_features, device=None, dtype=None) Construct a
```

linear transformation module. This function should accept the following parameters:

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

```
def forward(self, x: torch.Tensor) -> torch.Tensor
```

 Apply the linear transformation to the input.

Make sure to:

- subclass `nn.Module`
- call the superclass constructor
- construct and store your parameter as W (not W^T) 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`.

2.4.3 Embedding Module

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

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

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

```
def __init__(self, num_embeddings, embedding_dim, device=None, dtype=None)
```

 Construct an embedding module. This function should accept the following parameters:

num_embeddings: `int` Size of the vocabulary
embedding_dim: `int` Dimension of the embedding vectors, i.e., d_{model}
device: `torch.device` | `None` = `None` Device to store the parameters on
dtype: `torch.dtype` | `None` = `None` Data type of the parameters

```
def forward(self, token_ids: torch.Tensor) -> torch.Tensor
```

 Lookup the embedding vectors for the given token IDs.

Make sure to:

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

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

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

2.5 Pre-Norm Transformer Block

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

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

2.5.1 Root Mean Square Layer Normalization

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

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

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

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

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

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

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

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

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

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

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

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

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

2.5.2 Position-Wise Feed-Forward Network

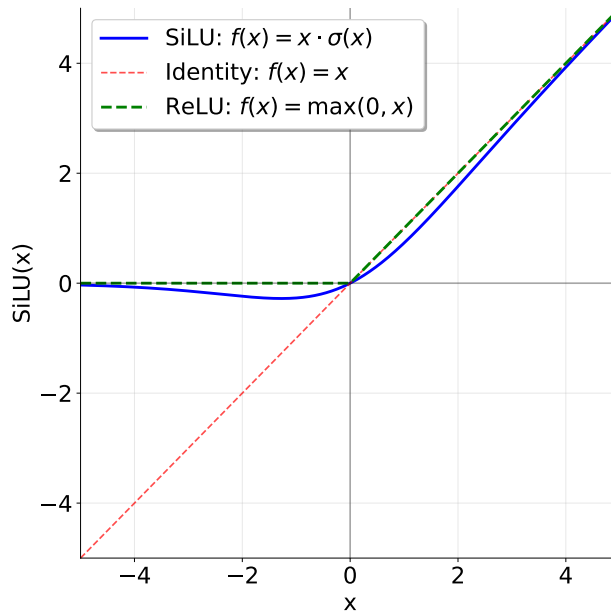


Figure 3: Comparing the SiLU (aka Swish) and ReLU activation functions.

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

dimensionality of the inner feed-forward layer is typically 4x the input dimensionality.

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

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

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

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

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

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

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

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

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

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

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

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

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

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

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

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

2.5.3 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 (note the 1-indexing used in the mathematical formulas here!), 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 $q_{2k-1:2k}^{(i)}$ as 2d vectors by the angle $\theta_{i,k} = \frac{i}{\Theta(2k-2)/d}$ for $k \in \{1, \dots, d/2\}$ and some constant Θ . Thus, we can consider R^i to be a block-diagonal matrix of size $d \times d$, with blocks R_k^i for $k \in \{1, \dots, d/2\}$, with

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

Thus we get the full rotation matrix

$$R^i = \begin{bmatrix} R_1^i & 0 & 0 & \dots & 0 \\ 0 & R_2^i & 0 & \dots & 0 \\ 0 & 0 & R_3^i & \dots & 0 \\ \vdots & \vdots & \vdots & \ddots & \vdots \\ 0 & 0 & 0 & \dots & R_{d/2}^i \end{bmatrix}, \quad (9)$$

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

Problem (rope): Implement RoPE (2 points)

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

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

    theta: float  $\Theta$  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  $(\dots, \text{seq\_len}, d_k)$  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  $(\dots, \text{seq\_len})$  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`.

2.5.4 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:

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

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

Problem (softmax): Implement softmax (1 point)

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

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

We can now define the Attention operation mathematically as follows:

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

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 2.3.1.

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

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

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

Deliverable: Implement the scaled dot-product attention function. Your implementation should handle keys and queries of shape `(batch_size, ..., seq_len, d_k)` and values of shape `(batch_size, ..., seq_len, d_v)`, where `...` represents any number of other batch-like dimensions (if provided). The implementation should return an output with the shape `(batch_size, ..., d_v)`. See section 2.3 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.

2.5.5 Causal Multi-Head Self-Attention

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

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

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

with Q_i, K_i, V_i being slice number $i \in \{1, \dots, h\}$ 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 §2.5.4. From this we can form the multi-head *self*-attention operation:

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

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.⁴

Causal masking. Your implementation should prevent the model from attending to future tokens in the sequence. In other words, if the model is given a token sequence $t_1, \dots, t_n$, and we want to calculate the next-word predictions for the prefix $t_1, \dots, t_i$ (where $i < n$), the model should *not* be able to access (attend to) the token representations at positions $t_{i+1}, \dots, t_n$ since it will not have access to these tokens when generating text during inference (and these future tokens leak information about the identity of the true next word, trivializing the language modeling pre-training objective). For an input token sequence $t_1, \dots, t_n$ we can naively prevent access to future tokens by running multi-head self-attention n times (for the n unique prefixes in the sequence). Instead, we'll use causal attention masking, which allows token i to attend to all positions $j \leq i$ in the sequence. You can use `torch.triu` or a broadcasted index comparison to construct this mask, and you should take advantage of the fact that your scaled dot-product attention implementation from §2.5.4 already supports attention masking.

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

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

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

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

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

Following Vaswani et al. [2017], set $d_k = d_v = d_{\text{model}}/h$. To test your implementation against our

⁴As a stretch goal, try combining the key, query, and value projections into a single weight matrix so you only need a single matrix multiply.

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.

2.6 The Full Transformer LM

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

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

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

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

Implement the pre-norm Transformer block as described in §2.5 and illustrated in Figure 2. Your Transformer block should accept (at least) the following parameters.

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

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

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

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

Deliverable: Transformer block code that passes the provided tests.

Now we put the blocks together, following the high level diagram in Figure 1. Follow our description of the embedding in Section 2.1.1, feed this into `num_layers` Transformer blocks, and then pass that into the three output layers to obtain a distribution over the vocabulary.

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

Time to put it all together! Implement the Transformer language model as described in §2.1 and illustrated in Figure 1. At minimum, your implementation should accept all the aforementioned construction parameters for the Transformer block, as well as these additional parameters:

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

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

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

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

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

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

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

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

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

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

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

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

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

```
vocab_size : 50,257
context_length : 1,024
num_layers : 48
d_model : 1,600
num_heads : 25
d_ff : 6,400
```

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

Deliverable: A one-to-two sentence response.

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

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

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

Deliverable: A one-to-two sentence response.

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

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

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

Deliverable: A one-to-two sentence response.

3 Training a Transformer LM

We now have the model (Transformer) and pre-tokenized data to train it on. 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 a way to load batches of data and run a training run end to end.

3.1 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 $i = 1, \dots, m$. Given a training set D consisting of sequences of length m , we define the standard cross-entropy (negative log-likelihood) loss function:

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

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

In particular, the Transformer computes logits $o_i \in \mathbb{R}^{\text{vocab_size}}$ for each position i , which results in:⁵

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

The cross entropy loss is generally defined with respect to the vector of logits $o_i \in \mathbb{R}^{\text{vocab_size}}$ and target x_{i+1} .⁶

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

Problem (cross_entropy): Implement Cross entropy (1 point)

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 2.3, we assume batch-like dimensions always come first, before the vocabulary size dimension.

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

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

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

⁵Note that $o_i[k]$ refers to value at index k of the vector o_i .

⁶This corresponds to the cross entropy between the Dirac delta distribution over x_{i+1} and the predicted $\text{softmax}(o_i)$ distribution.

3.2 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 $t = 0, \dots, T - 1$, we perform the following update:

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

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.

3.2.1 Implementing SGD in PyTorch

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

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

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

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

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

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.

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.

3.3 AdamW

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

in a way that is decoupled from the gradient update. We will implement AdamW as described in algorithm 2 of Loshchilov and Hutter [2019].

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

Algorithm 1 AdamW Optimizer

```

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

```

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

Problem (adamw): Implement AdamW (2 points)

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

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

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

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

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

- Transformer block
 - RMSNorm(s)

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

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

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

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

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

Deliverable: An algebraic expression, with a brief justification.

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

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

3.4 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 trains⁷. In this assignment, we will implement the cosine annealing schedule used to train LLaMA [Touvron et al., 2023].

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

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

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

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

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

Problem (learning_rate_schedule): Implement cosine learning rate schedule with warmup (1 point)

⁷It's sometimes common to use a schedule where the learning rate rises back up (restarts) to help get past local minima.

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`.

3.5 Gradient clipping

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

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

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

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

4 Data Loading and a Small Training Run

We will now finally put together the model and the optimizer. We will train the model on some tokenized data: we provide a pre-tokenized TinyStories validation split and the vocabulary used to produce it (details in §4.2).

4.1 Data Loader

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

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

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

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

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

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

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

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

For more on MPS, checkout these resources:

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

What if the dataset is too big to load into memory? We can use a Unix systemcall named `mmap` which maps a file on disk to virtual memory, and lazily loads the file contents when that memory location is accessed. Thus, you can “pretend” you have the entire dataset in memory. Numpy implements this through `np.memmap` (or the flag `mmap_mode='r'` to `np.load`, if you originally saved the array with `np.save`), which will return a numpy array-like object that loads the entries on-demand as you access them. **When sampling from your dataset (i.e., a numpy array) during training, be sure load the dataset in memory-mapped mode** (via `np.memmap` or the flag `mmap_mode='r'` to `np.load`, depending on how you saved the array). Make sure you also specify a `dtype` that matches the array that you’re loading. It may be helpful

to explicitly verify that the memory-mapped data looks correct (e.g., doesn't contain values beyond the expected vocabulary size).

4.2 Putting it together: a small training run

Now we combine the Transformer LM, cross-entropy loss, AdamW, and `get_batch` and watch a real language model train.

The data we give you is a small amount of data from TinyStories, labeled as a validation set, already tokenized with a 10,000-token byte-level BPE vocabulary (`data/tinystories_valid.bin + .meta`, about 5.4M tokens), the vocabulary and merges used to produce it (`data/tinystories_vocab_merges.pt`), and a short `data_utils.py` with two functions you can use:

- `load_memmap_dataset(bin_path, meta_path)`, which memory-maps the `.bin` file (using the dtype and length recorded in the `.meta` file) and returns a 1-D array of token IDs you can pass straight to your `get_batch`.
- `decode(ids, vocab)`, which turns a list of token IDs back into text, for when you want to eyeball what your (untrained or lightly trained) model is generating.

Problem (first_training_run): Put it together (3 points)

Deliverable: Write a short script (a notebook is fine too) that:

1. Loads `tinystories_valid.bin` with `load_memmap_dataset`.
2. Constructs your Transformer LM, AdamW optimizer, and (optionally) a learning rate schedule.
3. Runs a training loop: repeatedly sample a batch with your `get_batch`, compute the loss with your cross-entropy implementation, and take an optimizer step. A few hundred to a couple thousand steps at a small model size (e.g., `d_model=128`, `d_ff=384`, 2–4 layers, `context_length=64--128`, batch size 16–32) is enough, which should run in just a few minutes even on modest hardware.
4. Periodically prints (or plots) the training loss.

Deliverable: A learning curve (loss on the y-axis, batches processed on the x-axis) showing loss going down. Write 1-2 sentences on what you observed: does it decrease smoothly? Feel free to also use `decode` (implemented next) to look at what your model outputs before vs. after training, though we aren't expecting fluent text out of a model this small / undertrained.)

5 Generating text

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

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

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

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

$$P(x_{t+1} = i \mid x_{1\dots t}) = \frac{\exp(v_i)}{\sum_j \exp(v_j)}$$

$$v = \text{TransformerLM}(x_{1\dots t})_t \in \mathbb{R}^{\text{vocab_size}}$$

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

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

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

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

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

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

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

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

Problem (decoding): Decoding (3 points)

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

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

Indicate in your writeup where this script lives.

References

- Ronen Eldan and Yuanzhi Li. TinyStories: How small can language models be and still speak coherent English?, 2023. arXiv:2305.07759.
- Alec Radford, Karthik Narasimhan, Tim Salimans, and Ilya Sutskever. Improving language understanding by generative pre-training, 2018.
- Ashish Vaswani, Noam Shazeer, Niki Parmar, Jakob Uszkoreit, Llion Jones, Aidan N Gomez, Łukasz Kaiser, and Illia Polosukhin. Attention is all you need. In *Proc. of NeurIPS*, 2017.
- Toan Q. Nguyen and Julian Salazar. Transformers without tears: Improving the normalization of self-attention. In *Proc. of IWSWLT*, 2019.
- Ruibin Xiong, Yunchang Yang, Di He, Kai Zheng, Shuxin Zheng, Chen Xing, Huishuai Zhang, Yanyan Lan, Liwei Wang, and Tie-Yan Liu. On layer normalization in the Transformer architecture. In *Proc. of ICML*, 2020.
- Jimmy Lei Ba, Jamie Ryan Kiros, and Geoffrey E. Hinton. Layer normalization, 2016. arXiv:1607.06450.
- Hugo Touvron, Thibaut Lavril, Gautier Izacard, Xavier Martinet, Marie-Anne Lachaux, Timothée Lacroix, Baptiste Rozière, Naman Goyal, Eric Hambro, Faisal Azhar, Aurelien Rodriguez, Armand Joulin, Edouard Grave, and Guillaume Lample. Llama: Open and efficient foundation language models, 2023. arXiv:2302.13971.
- Biao Zhang and Rico Sennrich. Root mean square layer normalization. In *Proc. of NeurIPS*, 2019.
- Aaron Grattafiori, Abhimanyu Dubey, Abhinav Jauhri, Abhinav Pandey, Abhishek Kadian, Ahmad Al-Dahle, Aiesha Letman, Akhil Mathur, Alan Schelten, Alex Vaughan, Amy Yang, Angela Fan, Anirudh Goyal, Anthony Hartshorn, Aobo Yang, Archi Mitra, Archie Sravankumar, Artem Korenev, Arthur Hinsvark, Arun Rao, Aston Zhang, Aurelien Rodriguez, Austen Gregerson, Ava Spataru, Baptiste Roziere, Bethany Biron, Binh Tang, Bobbie Chern, Charlotte Caucheteux, Chaya Nayak, Chloe Bi, Chris Marra, Chris McConnell, Christian Keller, Christophe Touret, Chunyang Wu, Corinne Wong, Cristian Canton Ferrer, Cyrus Nikolaidis, Damien Allonsius, Daniel Song, Danielle Pintz, Danny Livshits, Danny Wyatt, David Esiobu, Dhruv Choudhary, Dhruv Mahajan, Diego Garcia-Olano, Diego Perino, Dieuwke Hupkes, Egor Lakomkin, Ehab AlBadawy, Elina Lobanova, Emily Dinan, Eric Michael Smith, Filip Radenovic, Francisco Guzmán, Frank Zhang, Gabriel Synnaeve, Gabrielle Lee, Georgia Lewis Anderson, Govind Thattai, Graeme Nail, Gregoire Mialon, Guan Pang, Guillem Cucurell, Hailey Nguyen, Hannah Korevaar, Hu Xu, Hugo Touvron, Iliyan Zarov, Imanol Arrieta Ibarra, Isabel Kloumann, Ishan Misra, Ivan Evtimov, Jack Zhang, Jade Copet, Jaewon Lee, Jan Geffert, Jana Vranes, Jason Park, Jay Mahadeokar, Jeet Shah, Jelmer van der Linde, Jennifer Billock, Jenny Hong, Jenya Lee, Jeremy Fu, Jianfeng Chi, Jianyu Huang, Jiawen Liu, Jie Wang, Jiecao Yu, Joanna Bitton, Joe Spisak, Jongsoo Park, Joseph Rocca, Joshua Johnstun, Joshua Saxe, Junteng Jia, Kalyan Vasuden Alwala, Karthik Prasad, Kartikeya Upasani, Kate Plawiak, Ke Li, Kenneth Heafield, Kevin Stone, Khalid El-Arini, Krithika Iyer, Kshitiz Malik, Kuenley Chiu, Kunal Bhalla, Kushal Lakhotia, Lauren Rantala-Yeary, Laurens van der Maaten, Lawrence Chen, Liang Tan, Liz Jenkins, Louis Martin, Lovish Madaan, Lubo Malo, Lukas Blecher, Lukas Landzaat, Luke de Oliveira, Madeline Muzzi, Mahesh Pasupuleti, Mannat Singh, Manohar Paluri, Marcin Kardas, Maria Tsimpoukelli, Mathew Oldham, Mathieu Rita, Maya Pavlova, Melanie Kambadur, Mike Lewis, Min Si, Mitesh Kumar Singh, Mona Hassan, Naman Goyal, Narjes Torabi, Nikolay Bashlykov, Nikolay Bogoychev, Niladri Chatterji, Ning Zhang, Olivier Duchenne, Onur Çelebi, Patrick Alrassy, Pengchuan Zhang, Pengwei Li, Petar Vasic, Peter Weng, Prajjwal Bhargava, Pratik Dubal, Praveen Krishnan, Punit Singh Koura, Puxin Xu, Qing He, Qingxiao Dong, Ragavan Srinivasan, Raj Ganapathy, Ramon Calderer, Ricardo Silveira Cabral, Robert Stojnic, Roberta Raileanu, Rohan Maheswari, Rohit Girdhar, Rohit Patel, Romain Sauvestre, Ronnie Polidoro, Roshan Sumbaly, Ross Taylor, Ruan Silva, Rui Hou, Rui Wang, Saghar Hosseini, Sahana Chennabasappa, Sanjay Singh, Sean Bell, Seohyun Sonia Kim, Sergey Edunov, Shao-liang

Nie, Sharan Narang, Sharath Raparthy, Sheng Shen, Shengye Wan, Shruti Bhosale, Shun Zhang, Simon Vandenhende, Soumya Batra, Spencer Whitman, Sten Sootla, Stephane Collot, Suchin Gururangan, Sydney Borodinsky, Tamar Herman, Tara Fowler, Tarek Sheasha, Thomas Georgiou, Thomas Scialom, Tobias Speckbacher, Todor Mihaylov, Tong Xiao, Ujjwal Karn, Vedanuj Goswami, Vibhor Gupta, Vignesh Ramanathan, Viktor Kerkez, Vincent Gonguet, Virginie Do, Vish Vogeti, Vitor Albiero, Vladan Petrovic, Weiwei Chu, Wenhan Xiong, Wenxin Fu, Whitney Meers, Xavier Martinet, Xiaodong Wang, Xiaofang Wang, Xiaoqing Ellen Tan, Xide Xia, Xinfeng Xie, Xuchao Jia, Xuwei Wang, Yaelle Goldschlag, Yashesh Gaur, Yasmine Babaei, Yi Wen, Yiwen Song, Yuchen Zhang, Yue Li, Yuning Mao, Zacharie Delpierre Coudert, Zheng Yan, Zhengxing Chen, Zoe Papakipos, Aaditya Singh, Aayushi Srivastava, Abha Jain, Adam Kelsey, Adam Shajnfeld, Adithya Gangidi, Adolfo Victoria, Ahuva Goldstand, Ajay Menon, Ajay Sharma, Alex Boesenberg, Alexei Baevski, Allie Feinstein, Amanda Kallet, Amit Sangani, Amos Teo, Anam Yunus, Andrei Lupu, Andres Alvarado, Andrew Caples, Andrew Gu, Andrew Ho, Andrew Poulton, Andrew Ryan, Ankit Ramchandani, Annie Dong, Annie Franco, Anuj Goyal, Aparajita Saraf, Arkabandhu Chowdhury, Ashley Gabriel, Ashwin Bharambe, Assaf Eisenman, Azadeh Yazdan, Beau James, Ben Maurer, Benjamin Leonhardi, Bernie Huang, Beth Loyd, Beto De Paola, Bhargavi Paranjape, Bing Liu, Bo Wu, Boyu Ni, Braden Hancock, Bram Wasti, Brandon Spence, Brani Stojkovic, Brian Gamido, Britt Montalvo, Carl Parker, Carly Burton, Catalina Mejia, Ce Liu, Changan Wang, Changkyu Kim, Chao Zhou, Chester Hu, Ching-Hsiang Chu, Chris Cai, Chris Tindal, Christoph Feichtenhofer, Cynthia Gao, Damon Civin, Dana Beaty, Daniel Kreymer, Daniel Li, David Adkins, David Xu, Davide Testuggine, Delia David, Devi Parikh, Diana Liskovich, Didem Foss, Dingkang Wang, Duc Le, Dustin Holland, Edward Dowling, Eissa Jamil, Elaine Montgomery, Eleonora Presani, Emily Hahn, Emily Wood, Eric-Tuan Le, Erik Brinkman, Esteban Arcaute, Evan Dunbar, Evan Smothers, Fei Sun, Felix Kreuk, Feng Tian, Filippos Kokkinos, Firat Ozgenel, Francesco Caggioni, Frank Kanayet, Frank Seide, Gabriela Medina Florez, Gabriella Schwarz, Gada Badeer, Georgia Swee, Gil Halpern, Grant Herman, Grigory Sizov, Guangyi, Zhang, Guna Lakshminarayanan, Hakan Inan, Hamid Shojanazeri, Han Zou, Hannah Wang, Hanwen Zha, Haroun Habeeb, Harrison Rudolph, Helen Suk, Henry Aspegren, Hunter Goldman, Hongyuan Zhan, Ibrahim Damla, Igor Molybog, Igor Tufanov, Ilias Leontiadis, Irina-Elena Veliche, Itai Gat, Jake Weissman, James Geboski, James Kohli, Janice Lam, Japhet Asher, Jean-Baptiste Gaya, Jeff Marcus, Jeff Tang, Jennifer Chan, Jenny Zhen, Jeremy Reizenstein, Jeremy Teboul, Jessica Zhong, Jian Jin, Jingyi Yang, Joe Cummings, Jon Carvill, Jon Shepard, Jonathan McPhie, Jonathan Torres, Josh Ginsburg, Junjie Wang, Kai Wu, Kam Hou U, Karan Saxena, Kartikay Khandelwal, Katayoun Zand, Kathy Matosich, Kaushik Veeraraghavan, Kelly Michelena, Keqian Li, Kiran Jagadeesh, Kun Huang, Kunal Chawla, Kyle Huang, Lailin Chen, Lakshya Garg, Lavender A, Leandro Silva, Lee Bell, Lei Zhang, Liangpeng Guo, Licheng Yu, Liron Moshkovich, Luca Wehrstedt, Madian Khabsa, Manav Avalani, Manish Bhatt, Martynas Mankus, Matan Hasson, Matthew Lennie, Matthias Reso, Maxim Groshev, Maxim Naumov, Maya Lathi, Meghan Keneally, Miao Liu, Michael L. Seltzer, Michal Valko, Michelle Restrepo, Mihir Patel, Mik Vyatskov, Mikayel Samvelyan, Mike Clark, Mike Macey, Mike Wang, Miquel Jubert Hermoso, Mo Metanat, Mohammad Rastegari, Munish Bansal, Nandhini Santhanam, Natascha Parks, Natasha White, Navyata Bawa, Nayan Singhal, Nick Egebo, Nicolas Usunier, Nikhil Mehta, Nikolay Pavlovich Laptev, Ning Dong, Norman Cheng, Oleg Chernoguz, Olivia Hart, Omkar Salpekar, Ozlem Kalinli, Parkin Kent, Parth Parekh, Paul Saab, Pavan Balaji, Pedro Rittner, Philip Bontrager, Pierre Roux, Piotr Dollar, Polina Zvyagina, Prashant Ratanchandani, Pritish Yuvraj, Qian Liang, Rachad Alao, Rachel Rodriguez, Rafi Ayub, Raghotham Murthy, Raghu Nayani, Rahul Mitra, Rangaprabhu Parthasarathy, Raymond Li, Rebekkah Hogan, Robin Battey, Rocky Wang, Russ Howes, Ruty Rinott, Sachin Mehta, Sachin Siby, Sai Jayesh Bondu, Samyak Datta, Sara Chugh, Sara Hunt, Sargun Dhillon, Sasha Sidorov, Satadru Pan, Saurabh Mahajan, Saurabh Verma, Seiji Yamamoto, Sharadh Ramaswamy, Shaun Lindsay, Shaun Lindsay, Sheng Feng, Shenghao Lin, Shengxin Cindy Zha, Shishir Patil, Shiva Shankar, Shuqiang Zhang, Shuqiang Zhang, Sinong Wang, Sneha Agarwal, Soji Sajuyigbe, Soumith Chintala, Stephanie Max, Stephen Chen, Steve Kehoe, Steve Satterfield, Sudarshan Govindaprasad, Sumit Gupta, Summer Deng, Sungmin Cho, Sunny Virk, Suraj Subramanian, Sy Choudhury, Sydney Goldman, Tal Remez, Tamar Glaser, Tamara Best, Thilo Koehler, Thomas Robinson, Tianhe Li, Tianjun Zhang, Tim Matthews, Timothy Chou, Tzook Shaked,

- Varun Vontimitta, Victoria Ajayi, Victoria Montanez, Vijai Mohan, Vinay Satish Kumar, Vishal Mangla, Vlad Ionescu, Vlad Poenaru, Vlad Tiberiu Mihailescu, Vladimir Ivanov, Wei Li, Wenchen Wang, Wenwen Jiang, Wes Bouaziz, Will Constable, Xiaocheng Tang, Xiaojuan Wu, Xiaolan Wang, Xilun Wu, Xinbo Gao, Yaniv Kleinman, Yanjun Chen, Ye Hu, Ye Jia, Ye Qi, Yenda Li, Yilin Zhang, Ying Zhang, Yossi Adi, Youngjin Nam, Yu, Wang, Yu Zhao, Yuchen Hao, Yundi Qian, Yunlu Li, Yuzi He, Zach Rait, Zachary DeVito, Zef Rosnbrick, Zhaoduo Wen, Zhenyu Yang, Zhiwei Zhao, and Zhiyu Ma. The llama 3 herd of models, 2024. URL <https://arxiv.org/abs/2407.21783>.
- An Yang, Baosong Yang, Beichen Zhang, Binyuan Hui, Bo Zheng, Bowen Yu, Chengyuan Li, Dayiheng Liu, Fei Huang, Haoran Wei, Huan Lin, Jian Yang, Jianhong Tu, Jianwei Zhang, Jianxin Yang, Jiayi Yang, Jingren Zhou, Junyang Lin, Kai Dang, Keming Lu, Keqin Bao, Kexin Yang, Le Yu, Mei Li, Mingfeng Xue, Pei Zhang, Qin Zhu, Rui Men, Runji Lin, Tianhao Li, Tingyu Xia, Xingzhang Ren, Xuancheng Ren, Yang Fan, Yang Su, Yichang Zhang, Yu Wan, Yuqiong Liu, Zeyu Cui, Zhenru Zhang, and Zihan Qiu. Qwen2.5 technical report. *arXiv preprint arXiv:2412.15115*, 2024.
- Aakanksha Chowdhery, Sharan Narang, Jacob Devlin, Maarten Bosma, Gaurav Mishra, Adam Roberts, Paul Barham, Hyung Won Chung, Charles Sutton, Sebastian Gehrmann, Parker Schuh, Kensen Shi, Sasha Tsvyashchenko, Joshua Maynez, Abhishek Rao, Parker Barnes, Yi Tay, Noam Shazeer, Vinodkumar Prabhakaran, Emily Reif, Nan Du, Ben Hutchinson, Reiner Pope, James Bradbury, Jacob Austin, Michael Isard, Guy Gur-Ari, Pengcheng Yin, Toju Duke, Anselm Levskaya, Sanjay Ghemawat, Sunipa Dev, Henryk Michalewski, Xavier Garcia, Vedant Misra, Kevin Robinson, Liam Fedus, Denny Zhou, Daphne Ippolito, David Luan, Hyeontaek Lim, Barret Zoph, Alexander Spiridonov, Ryan Sepassi, David Dohan, Shivani Agrawal, Mark Omernick, Andrew M. Dai, Thanumalayan Sankaranarayanan Pillai, Marie Pellat, Aitor Lewkowycz, Erica Moreira, Rewon Child, Oleksandr Polozov, Katherine Lee, Zongwei Zhou, Xuezhi Wang, Brennan Saeta, Mark Diaz, Orhan Firat, Michele Catasta, Jason Wei, Kathy Meier-Hellstern, Douglas Eck, Jeff Dean, Slav Petrov, and Noah Fiedel. PaLM: Scaling language modeling with pathways, 2022. arXiv:2204.02311.
- Dan Hendrycks and Kevin Gimpel. Bridging nonlinearities and stochastic regularizers with gaussian error linear units, 2016. arXiv:1606.08415.
- Stefan Elfving, Eiji Uchibe, and Kenji Doya. Sigmoid-weighted linear units for neural network function approximation in reinforcement learning, 2017. URL <https://arxiv.org/abs/1702.03118>.
- Yann N. Dauphin, Angela Fan, Michael Auli, and David Grangier. Language modeling with gated convolutional networks, 2017. URL <https://arxiv.org/abs/1612.08083>.
- Noam Shazeer. GLU variants improve transformer, 2020. arXiv:2002.05202.
- Jianlin Su, Yu Lu, Shengfeng Pan, Bo Wen, and Yunfeng Liu. Roformer: Enhanced transformer with rotary position embedding, 2021.
- Diederik P. Kingma and Jimmy Ba. Adam: A method for stochastic optimization. In *Proc. of ICLR*, 2015.
- Ilya Loshchilov and Frank Hutter. Decoupled weight decay regularization. In *Proc. of ICLR*, 2019.
- Tom B. Brown, Benjamin Mann, Nick Ryder, Melanie Subbiah, Jared Kaplan, Prafulla Dhariwal, Arvind Nee-lakantan, Pranav Shyam, Girish Sastry, Amanda Askell, Sandhini Agarwal, Ariel Herbert-Voss, Gretchen Krueger, Tom Henighan, Rewon Child, Aditya Ramesh, Daniel M. Ziegler, Jeffrey Wu, Clemens Winter, Christopher Hesse, Mark Chen, Eric Sigler, Mateusz Litwin, Scott Gray, Benjamin Chess, Jack Clark, Christopher Berner, Sam McCandlish, Alec Radford, Ilya Sutskever, and Dario Amodei. Language models are few-shot learners. In *Proc. of NeurIPS*, 2020.
- Jared Kaplan, Sam McCandlish, Tom Henighan, Tom B. Brown, Benjamin Chess, Rewon Child, Scott Gray, Alec Radford, Jeffrey Wu, and Dario Amodei. Scaling laws for neural language models, 2020. arXiv:2001.08361.

Jordan Hoffmann, Sebastian Borgeaud, Arthur Mensch, Elena Buchatskaya, Trevor Cai, Eliza Rutherford, Diego de Las Casas, Lisa Anne Hendricks, Johannes Welbl, Aidan Clark, Tom Hennigan, Eric Noland, Katie Millican, George van den Driessche, Bogdan Damoc, Aurelia Guy, Simon Osindero, Karen Simonyan, Erich Elsen, Jack W. Rae, Oriol Vinyals, and Laurent Sifre. Training compute-optimal large language models, 2022. arXiv:2203.15556.

Ari Holtzman, Jan Buys, Li Du, Maxwell Forbes, and Yejin Choi. The curious case of neural text degeneration. In *Proc. of ICLR*, 2020.