[1.1] - Transformers from scratch

Introduction

This is a clean, first principles implementation of GPT-2 in PyTorch. The architectural choices closely follow those used by the TransformerLens library (which you'll be using a lot more in later exercises).

The exercises are written to accompany Neel Nanda's TransformerLens library for doing mechanistic interpretability research on GPT-2 style language models. We'll be working with this library extensively in this chapter of the course.

Each exercise will have a difficulty and importance rating out of 5, as well as an estimated maximum time you should spend on these exercises and sometimes a short annotation. You should interpret the ratings & time estimates relatively (e.g. if you find yourself spending about 50% longer on the exercises than the time estimates, adjust accordingly). Please do skip exercises / look at solutions if you don't feel like they're important enough to be worth doing, and you'd rather get to the good stuff!

Content & Learning Objectives

1️⃣ Understanding Inputs & Outputs of a Transformer

In this section, we'll take a first look at transformers - what their function is, how information moves inside a transformer, and what inputs & outputs they take.

> Learning objectives
>
> - Understand what a transformer is used for
> - Understand causal attention, and what a transformer's output representsalgebra operations on tensors
> - Learn what tokenization is, and how models do it
> - Understand what logits are, and how to use them to derive a probability distribution over the vocabulary

2️⃣ Clean Transformer Implementation

Here, we'll implement a transformer from scratch, using only PyTorch's tensor operations. This will give us a good understanding of how transformers work, and how to use them. We do this by going module-by-module, in an experience which should feel somewhat similar to last week's ResNet exercises. Much like with ResNets, you'll conclude by loading in pretrained weights and verifying that your model works as expected.

> Learning objectives
>
> - Understand that a transformer is composed of attention heads and MLPs, with each one performing operations on the residual stream
> - Understand that the attention heads in a single layer operate independently, and that they have the role of calculating attention patterns (which determine where information is moved to & from in the residual stream)
> - Learn about & implement the following transformer modules:
>   - LayerNorm (transforming the input to have zero mean and unit variance)
>   - Positional embedding (a lookup table from position indices to residual stream vectors)
>   - Attention (the method of computing attention patterns for residual stream vectors)

- MLP (the collection of linear and nonlinear transformations which operate on each residual stream vector in the same way)
- Embedding (a lookup table from tokens to residual stream vectors)
- Unembedding (a matrix for converting residual stream vectors into a distribution over tokens)

3️⃣ Training a Transformer

Next, you'll learn how to train your transformer from scratch. This will be quite similar to the training loops you wrote for ResNet in your first week.

> Learning objectives
>
> - Understand how to train a transformer from scratch
> - Write a basic transformer training loop with PyTorch Lightning
> - Interpret the transformer's falling cross entropy loss with reference to features of the training data (e.g. bigram frequencies)

4️⃣ Sampling from a Transformer

Lastly, you'll learn how to sample from a transformer. This will involve implementing a few different sampling methods, and writing a caching system which can reuse computations from previous forward passes to improve your model's text generation speed.

The second half of this section is less important, and you can skip it if you want.

> Learning objectives
>
> - Learn how to sample from a transformer
>   - This includes basic methods like greedy search or top-k, and more advanced methods like beam search
> - Learn how to cache the output of a transformer, so that it can be used to generate text more efficiently
>   - Optionally, rewrite your sampling functions to make use of your caching methods

Setup (don't read, just run!)

[ ] ↳ *3 cells hidden*

1️⃣ Understanding Inputs & Outputs of a Transformer

> ### Learning Objectives
>
> - Understand what a transformer is used for
> - Understand causal attention, and what a transformer's output represents
> - Learn what tokenization is, and how models do it
> - Understand what logits are, and how to use them to derive a probability distribution over the vocabulary

What is the point of a transformer?

Transformers exist to model text!

We're going to focus GPT-2 style transformers. Key feature: They generate text! You feed in language, and the model generates a probability distribution over tokens. And you can repeatedly sample from this to generate text!

(To explain this in more detail - you feed in a sequence of length N, then sample from the probability distribution over the $N+1$-th word, use this to construct a new sequence of length $N+1$, then feed this new sequence into the model to get a probability distribution over the $N+2$-th word, and so on.)

How is the model trained?

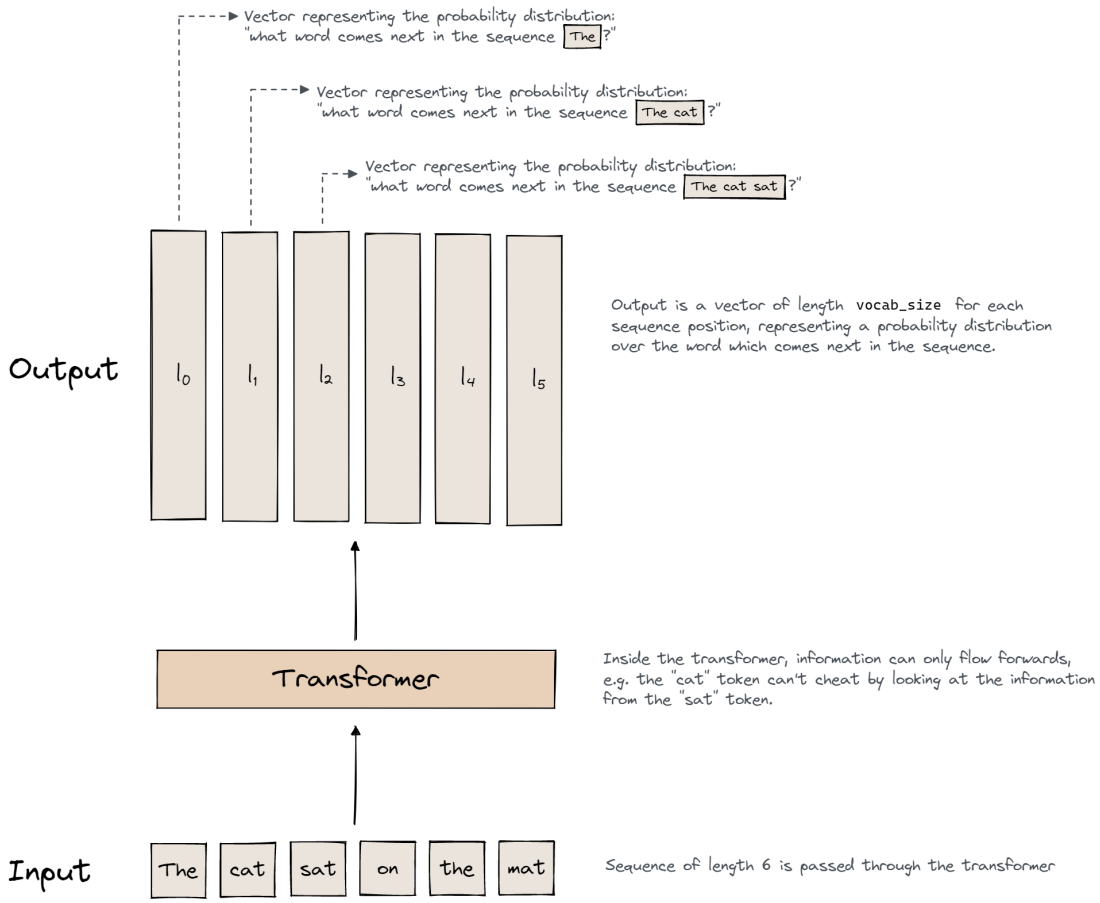
You give it a bunch of text, and train it to predict the next token.

Importantly, if you give a model 100 tokens in a sequence, it predicts the next token for *each* prefix, i.e. it produces 100 logit vectors (= probability distributions) over the set of all words in our vocabulary, with the `i`-th logit vector representing the probability distribution over the token *following* the `i`-th token in the sequence.

▶ Aside - logits

How do we stop the transformer by "cheating" by just looking at the tokens it's trying to predict? Answer - we make the transformer have *causal attention* (as opposed to *bidirectional attention*). Causal attention only allows information to move forwards in the sequence, never backwards.

The prediction of what comes after token 50 is only a function of the first 50 tokens, *not* of token 51. We say the transformer is **autoregressive**, because it only predicts future words based on past data.



Tokens - Transformer Inputs

Our tranformer's input is natural language (i.e. a sequence of characters, strings, etc). But ML models generally take vectors as input, not langage. How do we convert language to vectors?

We can factor this into 2 questions:

1. How do we split up language into small sub-units?
2. How do we convert these sub-units into vectors?

Let's start with the second of these questions.

Converting sub-units to vectors

We basically make a massive lookup table, which is called an **embedding**. It has one vector for each possible sub-unit of language we might get (we call this set of all sub-units our **vocabulary**). We label every element in our vocabulary with an integer (this labelling never changes), and we use this integer to index into the embedding.

A key intuition is that one-hot encodings let you think about each integer independently. We don't bake in any relation between words when we perform our embedding, because every word has a completely separate embedding vector.

▶ Aside - one-hot encodings

Now, let's answer the first question - how do we split language into sub-units?

Splitting language into sub-units

We need to define a standard way of splitting up language into a series of substrings, where each substring is a member of our **vocabulary** set.

Could we use a dictionary, and have our vocabulary be the set of all words in the dictionary? No, because this couldn't handle arbitrary text (e.g. URLs, punctuation, etc). We need a more general way of splitting up language.

Could we just use the 256 ASCII characters? This fixes the previous problem, but it loses structure of language - some sequences of characters are more meaningful than others. For example, "language" is a lot more meaningful than "hjksdfiu". We want "language" to be a single token, but not "hjksdfiu" - this is a more efficient use of our vocab.

What actually happens? The most common strategy is called **Byte-Pair encodings**.

We begin with the 256 ASCII characters as our tokens, and then find the most common pair of tokens, and merge that into a new token. Note that we do have a space character as one of our 256 tokens, and merges using space are very common. For instance, here are the five first merges for the tokenizer used by GPT-2 (you'll be able to verify this below).

```
" t"
" a"
"he"
"in"
"re"
```

▶ Fun (totally optional) exercise - can you guess what the first-formed 3/4/5/6/7-letter encodings in GPT-2's vocabulary are?

Note - you might see the character `Ġ` in front of some tokens. This is a special token that indicates that the token begins with a space. Tokens with a leading space vs not are different.

You can run the code below to see some more of GPT-2's tokenizer's vocabulary:

```
sorted_vocab = sorted(list(reference_gpt2.tokenizer.vocab.items()), key=lambda n: n[1])
print(sorted_vocab[:20])
print()
print(sorted_vocab[250:270])
print()
print(sorted_vocab[990:1010])
print()
```

```
[('!', 0), ('"', 1), ('#', 2), ('$', 3), ('%', 4), ('&', 5), ("'", 6), ('(', 7), (')', 8), ('*', 9), ('+', 10), (',', 11)

[('Ľ', 250), ('Ł', 251), ('ł', 252), ('Ń', 253), ('ń', 254), ('Ň', 255), ('Ġt', 256), ('Ġa', 257), ('he', 258), ('in', 25

[('Ġprodu', 990), ('Ġstill', 991), ('led', 992), ('ah', 993), ('Ġhere', 994), ('Ġworld', 995), ('Ġthough', 996), ('Ġnum',
```

As you get to the end of the vocabulary, you'll be producing some pretty weird-looking esoteric tokens (because you'll already have exhausted all of the short frequently-occurring ones):

```
print(sorted_vocab[-20:])
```

```
[('Revolution', 50237), ('Ġsnipers', 50238), ('Ġreverted', 50239), ('Ġconglomerate', 50240), ('Terry', 50241), ('794', 50
```

Transformers in the `transformer_lens` library have a `to_tokens` method that converts text to numbers. It also prepends them with a special token called BOS (beginning of sequence) to indicate the start of a sequence. You can disable this with the `prepend_bos=False` argument.

▶ Aside - BOS token

Some tokenization annoyances

There are a few funky and frustrating things about tokenization, which causes it to behave differently than you might expect. For instance:

Whether a word begins with a capital or space matters!

```
print(reference_gpt2.to_str_tokens("Ralph"))
print(reference_gpt2.to_str_tokens(" Ralph"))
print(reference_gpt2.to_str_tokens(" ralph"))
print(reference_gpt2.to_str_tokens("ralph"))
```

```
['<|endoftext|>', 'R', 'alph']
['<|endoftext|>', ' Ralph']
['<|endoftext|>', ' r', 'alph']
['<|endoftext|>', 'ral', 'ph']
```

Arithmetic is a mess.

Length is inconsistent, common numbers bundle together.

```
print(reference_gpt2.to_str_tokens("56873+3184623=123456789-1000000000"))
```

```
['<|endoftext|>', '568', '73', '+', '318', '46', '23', '=', '123', '45', '67', '89', '-', '1', '000000', '000']
```

> Key Takeaways
>
> - We learn a dictionary of vocab of tokens (sub-words).
> - We (approx) losslessly convert language to integers via tokenizing it.
> - We convert integers to vectors via a lookup table.
> - Note: input to the transformer is a sequence of *tokens* (ie integers), not vectors

Text generation

Now that we understand the basic ideas here, let's go through the entire process of text generation, from our original string to a new token which we can append to our string and plug back into the model.

Step 1: Convert text to tokens

The sequence gets tokenized, so it has shape `[batch, seq_len]`. Here, the batch dimension is just one (because we only have one sequence).

```
reference_text = "I am an amazing autoregressive, decoder-only, GPT-2 style transformer. One day I will exceed human level
tokens = reference_gpt2.to_tokens(reference_text).to(device)
print(tokens)
print(tokens.shape)
print(reference_gpt2.to_str_tokens(tokens))
```

```
tensor([[50256,    40,   716,   281,  4998,  1960,   382, 19741,    11,   875,
         12342,    12,  8807,    11,   402, 11571,    12,    17,  3918, 47385,
            13,  1881,  1110,   314,   481,  7074,  1692,  1241,  4430,   290,
          1011,   625,   262,   995,     0]], device='cuda:0')
torch.Size([1, 35])
['<|endoftext|>', 'I', ' am', ' an', ' amazing', ' aut', 'ore', 'gressive', ',', ' dec', 'oder', '-', 'only', ',', ' G',
```

Step 2: Map tokens to logits

From our input of shape `[batch, seq_len]`, we get output of shape `[batch, seq_len, vocab_size]`. The `[i, j, :]`-th element of our output is a vector of logits representing our prediction for the `j+1`-th token in the `i`-th sequence.

```
logits, cache = reference_gpt2.run_with_cache(tokens)
print(logits.shape)
```

```
torch.Size([1, 35, 50257])
```

(`run_with_cache` tells the model to cache all intermediate activations. This isn't important right now; we'll look at it in more detail later.)

Step 3: Convert the logits to a distribution with a softmax

This doesn't change the shape, it is still `[batch, seq_len, vocab_size]`.

```
probs = logits.softmax(dim=-1)
print(probs.shape)
```

```
torch.Size([1, 35, 50257])
```

Bonus step: What is the most likely next token at each position?

```
most_likely_next_tokens = reference_gpt2.tokenizer.batch_decode(logits.argmax(dim=-1)[0])

print(list(zip(reference_gpt2.to_str_tokens(tokens), most_likely_next_tokens)))
```

```
[('<|endoftext|>', '\n'), ('I', "'m"), (' am', ' a'), (' an', ' avid'), (' amazing', ' person'), (' aut', 'od'), ('ore',
```

We can see that, in a few cases (particularly near the end of the sequence), the model accurately predicts the next token in the sequence. We might guess that `"take over the world"` is a common phrase that the model has seen in training, which is why the model can predict it.

Step 4: Map distribution to a token

```
next_token = logits[0, -1].argmax(dim=-1)
next_char = reference_gpt2.to_string(next_token)
print(repr(next_char))
```

```
' I'
```

Note that we're indexing `logits[0, -1]`. This is because logits have shape `[1, sequence_length, vocab_size]`, so this indexing returns the vector of length `vocab_size` representing the model's prediction for what token follows the **last** token in the input sequence.

We can see the model predicts the line break character `\n`, since this is common following the end of a sentence.

Step 5: Add this to the end of the input, re-run

There are more efficient ways to do this (e.g. where we cache some of the values each time we run our input, so we don't have to do as much calculation each time we generate a new value), but this doesn't matter conceptually right now.

```
print(f"Sequence so far: {reference_gpt2.to_string(tokens)[0]!r}")

for i in range(10):
    print(f"{tokens.shape[-1]+1}th char = {next_char!r}")
    # Define new input sequence, by appending the previously generated token
    tokens = t.cat([tokens, next_token[None, None]], dim=-1)
    # Pass our new sequence through the model, to get new output
    logits = reference_gpt2(tokens)
    # Get the predicted token at the end of our sequence
    next_token = logits[0, -1].argmax(dim=-1)
    # Decode and print the result
    next_char = reference_gpt2.to_string(next_token)
```

```
Sequence so far: '<|endoftext|>I am an amazing autoregressive, decoder-only, GPT-2 style transformer. One day I will exce
36th char = ' I'
37th char = ' am'
38th char = ' a'
39th char = ' very'
40th char = ' talented'
41th char = ' and'
42th char = ' talented'
43th char = ' person'
44th char = ','
45th char = ' and'
```

Key takeaways

- Transformer takes in language, predicts next token (for *each* token in a causal way)
- We convert language to a sequence of integers with a tokenizer.
- We convert integers to vectors with a lookup table.
- Output is a vector of logits (one for each input token), we convert to a probability distn with a softmax, and can then convert this to a token (eg taking the largest logit, or sampling).
- We append this to the input + run again to generate more text (Jargon: *autoregressive*)
- Meta level point: Transformers are sequence operation models, they take in a sequence, do processing in parallel at each position, and use attention to move information between positions!

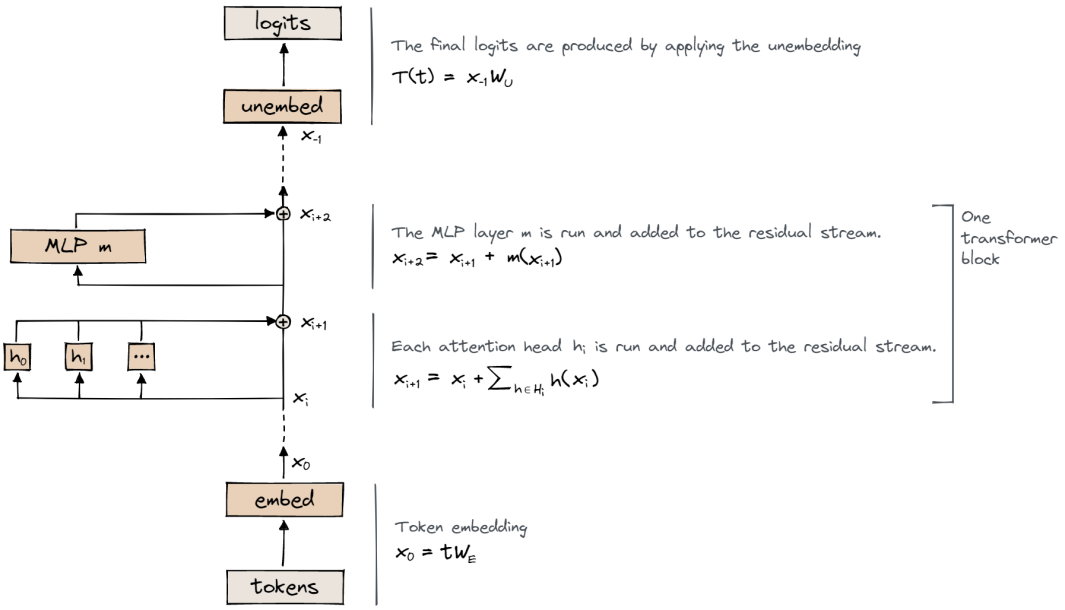
2️⃣ Clean Transformer Implementation

> Learning objectives
>
> - Understand that a transformer is composed of attention heads and MLPs, with each one performing operations on the residual stream
> - Understand that the attention heads in a single layer operate independently, and that they have the role of calculating attention patterns (which determine where information is moved to & from in the residual stream)
> - Learn about & implement the following transformer modules:
>   - LayerNorm (transforming the input to have zero mean and unit variance)
>   - Positional embedding (a lookup table from position indices to residual stream vectors)
>   - Attention (the method of computing attention patterns for residual stream vectors)
>   - MLP (the collection of linear and nonlinear transformations which operate on each residual stream vector in the same way)
>   - Embedding (a lookup table from tokens to residual stream vectors)
>   - Unembedding (a matrix for converting residual stream vectors into a distribution over tokens)

High-Level architecture

Go watch Neel's Transformer Circuits walkthrough if you want more intuitions!

(Diagram is bottom to top.)



Tokenization & Embedding

The input tokens t are integers. We get them from taking a sequence, and tokenizing it (like we saw in the previous section).

The token embedding is a lookup table mapping tokens to vectors, which is implemented as a matrix W_E. The matrix consists of a stack of token embedding vectors (one for each token).

Residual stream

The residual stream is the sum of all previous outputs of layers of the model, is the input to each new layer. It has shape `[batch, seq_len, d_model]` (where `d_model` is the length of a single embedding vector).

The initial value of the residual stream is denoted x_0 in the diagram, and x_i are later values of the residual stream (after more attention and MLP layers have been applied to the residual stream).

The residual stream is *really* fundamental. It's the central object of the transformer. It's how model remembers things, moves information between layers for composition, and it's the medium used to store the information that attention moves between positions.

▶ Aside - **logit lens**

Transformer blocks

Then we have a series of `n_layers` **transformer blocks** (also sometimes called **residual blocks**).

Note - a block contains an attention layer *and* an MLP layer, but we say a transformer has k layers if it has k blocks (i.e. $2k$ total layers).

Attention

First we have attention. This moves information from prior positions in the sequence to the current token.

We do this for *every* token in parallel using the same parameters. The only difference is that we look backwards only (to avoid "cheating"). This means later tokens have more of the sequence that they can look at.

Attention layers are the only bit of a transformer that moves information between positions (i.e. between vectors at different sequence positions in the residual stream).

Attention layers are made up of `n_heads` heads - each with their own parameters, own attention pattern, and own information how to copy things from source to destination. The heads act independently and additively, we just add their outputs together, and back to the stream.

Each head does the following:

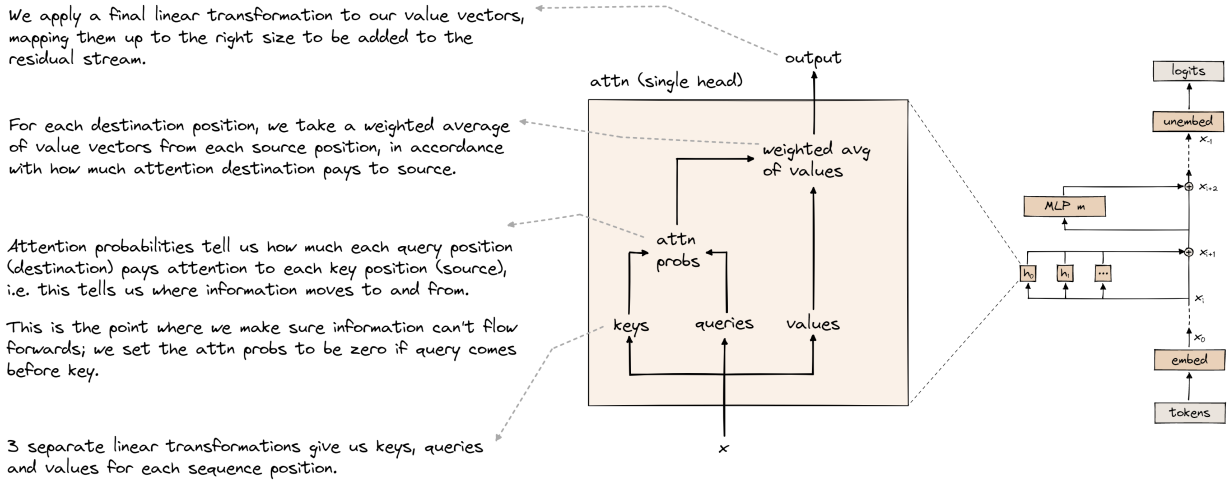
- Produces an **attention pattern** for each destination token, a probability distribution of prior source tokens (including the current one) weighting how much information to copy.
- Moves information (via a linear map) in the same way from each source token to each destination token.

A few key points:

- What information we copy depends on the source token's *residual stream*, but this doesn't mean it only depends on the value of that token, because the residual stream can store more information than just the token identity (the purpose of the attention heads is to move information between vectors at different positions in the residual stream!)
- We can think of each attention head as consisting of two different **circuits**:
    - One circuit determines **where to move information to and from** (this is a function of the residual stream for the source and destination tokens)
    - The other circuit determines **what information to move** (this is a function of only the source token's residual stream)
    - For reasons which will become clear later, we refer to the first circuit as the **QK circuit**, and the second circuit as the **OV circuit**

▶ Key intuition - attention as generalized convolution

Below is a schematic diagram of the attention layers. Don't worry if you don't follow this right now, we'll go into more detail during implementation.



MLP

The MLP layers are just a standard neural network, with a singular hidden layer and a nonlinear activation function. The exact activation isn't conceptually important (GELU seems to perform best).

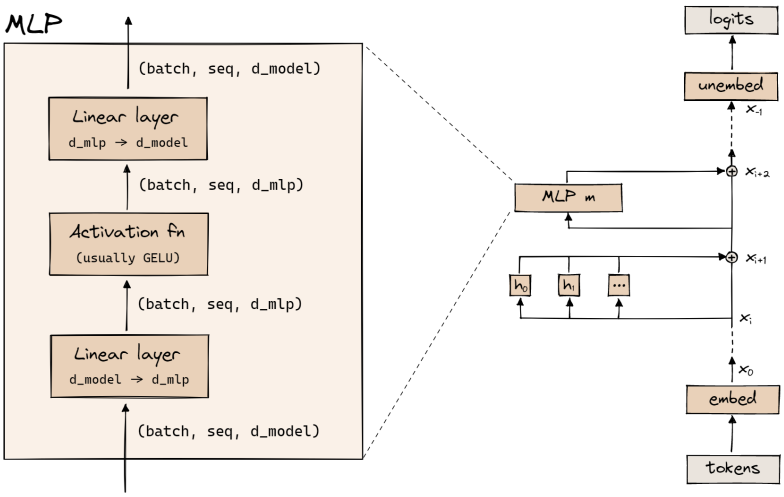
Our hidden dimension is normally `d_mlp = 4 * d_model`. Exactly why the ratios are what they are isn't super important (people basically cargo-cult what GPT did back in the day!).

Importantly, **the MLP operates on positions in the residual stream independently, and in exactly the same way**. It doesn't move information between positions.

Intuition - once attention has moved relevant information to a single position in the residual stream, MLPs can actually do computation, reasoning, lookup information, etc. *What the hell is going on inside MLPs* is a pretty big open problem in transformer mechanistic interpretability - see the Toy Model of Superposition Paper for more on why this is hard.

▶ Key intuition - MLPs as key-value pairs

▶ Key intuition - MLPs as knowledge storage



Unembedding

Finally, we unembed!

This just consists of applying a linear map W_U, going from final residual stream to a vector of logits - this is the output.

▶ Aside - tied embeddings

Bonus things - less conceptually important but key technical details

LayerNorm

- Simple normalization function applied at the start of each layer (i.e. before each MLP, attention layer, and before the unembedding)
- Converts each input vector (independently in parallel for each batch x position residual stream vector) to have mean zero and variance 1.
- Then applies an elementwise scaling and translation
- Cool maths tangent: The scale & translate is just a linear map. LayerNorm is only applied immediately before another linear map. Linear compose linear = linear, so we can just fold this into a single effective linear layer and ignore it.
  - `fold_ln=True` flag in `from_pretrained` does this for you.
- LayerNorm is annoying for interpertability - the scale part is not linear, so you can't think about different bits of the input independently. But it's *almost* linear - if you're changing a small part of the input it's linear, but if you're changing enough to alter the norm substantially it's not linear.

Positional embeddings

- **Problem:** Attention operates over all pairs of positions. This means it's symmetric with regards to position - the attention calculation from token 5 to token 1 and token 5 to token 2 are the same by default
  - This is dumb because nearby tokens are more relevant.
- There's a lot of dumb hacks for this.
- We'll focus on **learned, absolute positional embeddings**. This means we learn a lookup table mapping the index of the position of each token to a residual stream vector, and add this to the embed.
  - Note that we *add* rather than concatenate. This is because the residual stream is shared memory, and likely under significant superposition (the model compresses more features in there than the model has dimensions)
  - We basically never concatenate inside a transformer, unless doing weird shit like generating text efficiently.
- This connects to **attention as generalized convolution**
  - We argued that language does still have locality, and so it's helpful for transformers to have access to the positional information so they "know" two tokens are next to each other (and hence probably relevant to each other).

Actual Code!

Key (for the results you get when running the code immediately below)

```
batch = 1
position = 35
d_model = 768
n_heads = 12
n_layers = 12
```

```
d_mlp = 3072 (= 4 * d_model)
d_head = 64 (= d_model / n_heads)
```

Parameters and Activations

It's important to distinguish between parameters and activations in the model.

- **Parameters** are the weights and biases that are learned during training.
  - These don't change when the model input changes.
- **Activations** are temporary numbers calculated during a forward pass, that are functions of the input.
  - We can think of these values as only existing for the duration of a single forward pass, and disappearing afterwards.
  - We can use hooks to access these values during a forward pass (more on hooks later), but it doesn't make sense to talk about a model's activations outside the context of some particular input.
  - Attention scores and patterns are activations (this is slightly non-intuitve because they're used in a matrix multiplication with another activation).

Print All Activation Shapes of Reference Model

Run the following code to print all the activation shapes of the reference model:

```
for activation_name, activation in cache.items():
    # Only print for first layer
    if ".0." in activation_name or "blocks" not in activation_name:
        print(f"{activation_name:30} {tuple(activation.shape)}")
```

```
hook_embed                     (1, 35, 768)
hook_pos_embed                 (1, 35, 768)
blocks.0.hook_resid_pre        (1, 35, 768)
blocks.0.ln1.hook_scale        (1, 35, 1)
blocks.0.ln1.hook_normalized   (1, 35, 768)
blocks.0.attn.hook_q           (1, 35, 12, 64)
blocks.0.attn.hook_k           (1, 35, 12, 64)
blocks.0.attn.hook_v           (1, 35, 12, 64)
blocks.0.attn.hook_attn_scores (1, 12, 35, 35)
blocks.0.attn.hook_pattern     (1, 12, 35, 35)
blocks.0.attn.hook_z           (1, 35, 12, 64)
blocks.0.hook_attn_out         (1, 35, 768)
blocks.0.hook_resid_mid        (1, 35, 768)
blocks.0.ln2.hook_scale        (1, 35, 1)
blocks.0.ln2.hook_normalized   (1, 35, 768)
blocks.0.mlp.hook_pre          (1, 35, 3072)
blocks.0.mlp.hook_post         (1, 35, 3072)
blocks.0.hook_mlp_out          (1, 35, 768)
blocks.0.hook_resid_post       (1, 35, 768)
ln_final.hook_scale            (1, 35, 1)
ln_final.hook_normalized       (1, 35, 768)
```

Print All Parameters Shapes of Reference Model

```
for name, param in reference_gpt2.named_parameters():
    # Only print for first layer
    if ".0." in name or "blocks" not in name:
        print(f"{name:18} {tuple(param.shape)}")
```

```
embed.W_E          (50257, 768)
pos_embed.W_pos    (1024, 768)
blocks.0.ln1.w     (768,)
blocks.0.ln1.b     (768,)
blocks.0.ln2.w     (768,)
blocks.0.ln2.b     (768,)
blocks.0.attn.W_Q  (12, 768, 64)
blocks.0.attn.W_K  (12, 768, 64)
blocks.0.attn.W_V  (12, 768, 64)
blocks.0.attn.W_O  (12, 64, 768)
blocks.0.attn.b_Q  (12, 64)
blocks.0.attn.b_K  (12, 64)
blocks.0.attn.b_V  (12, 64)
blocks.0.attn.b_O  (768,)
blocks.0.mlp.W_in  (768, 3072)
blocks.0.mlp.b_in  (3072,)
blocks.0.mlp.W_out (3072, 768)
blocks.0.mlp.b_out (768,)
ln_final.w         (768,)
ln_final.b         (768,)
unembed.W_U        (768, 50257)
unembed.b_U        (50257,)
```

This diagram shows the name of all activations and parameters in a fully general transformer model from transformerlens (except for a few at the start and end, like the embedding and unembedding). Lots of this won't make sense at first, but you can return to this diagram later and check that you understand most/all parts of it.

Config

The config object contains all the hyperparameters of the model. We can print the config of the reference model to see what it contains:

```
# As a reference - note there's a lot of stuff we don't care about in here, to do with library internals or other architect
print(reference_gpt2.cfg)
```

```
HookedTransformerConfig:
{'act_fn': 'gelu_new',
 'attention_dir': 'causal',
 'attn_only': False,
 'attn_types': None,
 'checkpoint_index': None,
 'checkpoint_label_type': None,
 'checkpoint_value': None,
 'd_head': 64,
 'd_mlp': 3072,
 'd_model': 768,
 'd_vocab': 50257,
 'd_vocab_out': 50257,
 'device': 'cuda',
 'eps': 1e-05,
 'final_rms': False,
 'from_checkpoint': False,
 'gated_mlp': False,
 'init_mode': 'gpt2',
 'init_weights': False,
 'initializer_range': 0.02886751345948129,
 'model_name': 'gpt2',
 'n_ctx': 1024,
 'n_devices': 1,
 'n_heads': 12,
 'n_layers': 12,
 'n_params': 84934656,
 'normalization_type': 'LN',
 'original_architecture': 'GPT2LMHeadModel',
 'parallel_attn_mlp': False,
 'positional_embedding_type': 'standard',
 'rotary_dim': None,
 'scale_attn_by_inverse_layer_idx': False,
 'seed': None,
 'tokenizer_name': 'gpt2',
 'use_attn_result': False,
 'use_attn_scale': True,
 'use_hook_tokens': False,
 'use_local_attn': False,
 'use_split_qkv_input': False,
 'window_size': None}
```

We define a stripped down config for our model:

```
@dataclass
class Config:
    d_model: int = 768
    debug: bool = True
    layer_norm_eps: float = 1e-5
    d_vocab: int = 50257
    init_range: float = 0.02
    n_ctx: int = 1024
    d_head: int = 64
    d_mlp: int = 3072
    n_heads: int = 12
    n_layers: int = 12

cfg = Config()
print(cfg)
```

```
Config(d_model=768, debug=True, layer_norm_eps=1e-05, d_vocab=50257, init_range=0.02, n_ctx=1024, d_head=64, d_mlp=3072,
```

Tests

Tests are great, write lightweight ones to use as you go!

Naive test: Generate random inputs of the right shape, input to your model, check whether there's an error and print the correct output.

```
def rand_float_test(cls, shape):
    cfg = Config(debug=True)
    layer = cls(cfg).to(device)
    random_input = t.randn(shape).to(device)
    print("Input shape:", random_input.shape)
    output = layer(random_input)
    if isinstance(output, tuple): output = output[0]
    print("Output shape:", output.shape, "\n")

def rand_int_test(cls, shape):
    cfg = Config(debug=True)
    layer = cls(cfg).to(device)
    random_input = t.randint(100, 1000, shape).to(device)
    print("Input shape:", random_input.shape)
    output = layer(random_input)
    if isinstance(output, tuple): output = output[0]
    print("Output shape:", output.shape, "\n")

def load_gpt2_test(cls, gpt2_layer, input):
    cfg = Config(debug=True)
    layer = cls(cfg).to(device)
    layer.load_state_dict(gpt2_layer.state_dict(), strict=False)
    print("Input shape:", input.shape)
    output = layer(input)
    if isinstance(output, tuple): output = output[0]
    print("Output shape:", output.shape)
    try: reference_output = gpt2_layer(input)
    except: reference_output = gpt2_layer(input, input, input)
    print("Reference output shape:", reference_output.shape, "\n")
    comparison = t.isclose(output, reference_output, atol=1e-4, rtol=1e-3)
    print(f"{comparison.sum()/comparison.numel():.2%} of the values are correct\n")
```

LayerNorm

Difficulty: 🟠🟠🟠⚪⚪
Importance: 🟠🟠🟠⚪⚪

You should spend up to 10-15 minutes on this exercise.

You should fill in the code below, and then run the tests to verify that your layer is working correctly.

Your LayerNorm should do the following:

- Make mean 0
- Normalize to have variance 1
- Scale with learned weights
- Translate with learned bias

You can use the PyTorch LayerNorm documentation as a reference. A few more notes:

- Your layernorm implementation always has `affine=True`, i.e. you do learn parameters `w` and `b` (which are represented as γ and β respectively in the PyTorch documentation).
- Remember that, after the centering and normalization, each vector of length `d_model` in your input should have mean 0 and variance 1.
- As the PyTorch documentation page says, your variance should be computed using `unbiased=False`.
- The `layer_norm_eps` argument in your config object corresponds to the ϵ term in the PyTorch documentation (it is included to avoid division-by-zero errors).
- We've given you a `debug` argument in your config. If `debug=True`, then you can print output like the shape of objects in your `forward` function to help you debug (this is a very useful trick to improve your coding speed).

Fill in the function, where it says `pass` (this will be the basic pattern for most other exercises in this section).

```
class LayerNorm(nn.Module):
    def __init__(self, cfg: Config):
        super().__init__()
        self.cfg = cfg
        self.w = nn.Parameter(t.ones(cfg.d_model))
        self.b = nn.Parameter(t.zeros(cfg.d_model))

    def forward(self, residual: Float[Tensor, "batch posn d_model"]) -> Float[Tensor, "batch posn d_model"]:
        # SOLUTION
        residual_mean = residual.mean(dim=-1, keepdim=True)
        residual_std = (residual.var(dim=-1, keepdim=True, unbiased=False) + self.cfg.layer_norm_eps).sqrt()

        residual = (residual - residual_mean) / residual_std
        return residual * self.w + self.b
```

```
15
16
17 rand_float_test(LayerNorm, [2, 4, 768])
18 load_gpt2_test(LayerNorm, reference_gpt2.ln_final, cache["resid_post", 11])
```

```
Input shape: torch.Size([2, 4, 768])
Output shape: torch.Size([2, 4, 768])

Input shape: torch.Size([1, 35, 768])
Output shape: torch.Size([1, 35, 768])
Reference output shape: torch.Size([1, 35, 768])

100.00% of the values are correct
```

▶ Solution

Embedding

Difficulty: 🟠🟠⚪⚪⚪
Importance: 🟠🟠🟠⚪⚪

You should spend up to 5-10 minutes on this exercise.

This is basically a lookup table from tokens to residual stream vectors.

(Hint - you can implement this in just one line, without any complicated functions.)

```
1 class Embed(nn.Module):
2     def __init__(self, cfg: Config):
3         super().__init__()
4         self.cfg = cfg
5         self.W_E = nn.Parameter(t.empty((cfg.d_vocab, cfg.d_model)))
6         nn.init.normal_(self.W_E, std=self.cfg.init_range)
7
8     def forward(self, tokens: Int[Tensor, "batch position"]) -> Float[Tensor, "batch position d_model"]:
9         # SOLUTION
10        return self.W_E[tokens]
11
12
13 rand_int_test(Embed, [2, 4])
14 load_gpt2_test(Embed, reference_gpt2.embed, tokens)
```

```
Input shape: torch.Size([2, 4])
Output shape: torch.Size([2, 4, 768])

Input shape: torch.Size([1, 45])
Output shape: torch.Size([1, 45, 768])
Reference output shape: torch.Size([1, 45, 768])

100.00% of the values are correct
```

▶ Help - I keep getting `RuntimeError: CUDA error: device-side assert triggered`.
▶ Solution

Positional Embedding

Difficulty: 🟠🟠⚪⚪⚪
Importance: 🟠🟠🟠⚪⚪

You should spend up to 10-15 minutes on this exercise.

Positional embedding can also be thought of as a lookup table, but rather than the indices being our token IDs, the indices are just the numbers `0, 1, 2, ..., seq_len-1` (i.e. the position indices of the tokens in the sequence).

```
1 class PosEmbed(nn.Module):
2     def __init__(self, cfg: Config):
3         super().__init__()
4         self.cfg = cfg
5         self.W_pos = nn.Parameter(t.empty((cfg.n_ctx, cfg.d_model)))
6         nn.init.normal_(self.W_pos, std=self.cfg.init_range)
7
8     def forward(self, tokens: Int[Tensor, "batch position"]) -> Float[Tensor, "batch position d_model"]:
```

```
        # SOLUTION
        batch, seq_len = tokens.shape
        return einops.repeat(self.W_pos[:seq_len], "seq d_model -> batch seq d_model", batch=batch)

rand_int_test(PosEmbed, [2, 4])
load_gpt2_test(PosEmbed, reference_gpt2.pos_embed, tokens)
```

```
Input shape: torch.Size([2, 4])
Output shape: torch.Size([2, 4, 768])

Input shape: torch.Size([1, 45])
Output shape: torch.Size([1, 45, 768])
Reference output shape: torch.Size([1, 45, 768])

100.00% of the values are correct
```

▶ Solution

Attention

Difficulty: 🟠🟠🟠🟠⚪
Importance: 🟠🟠🟠🟠🟠

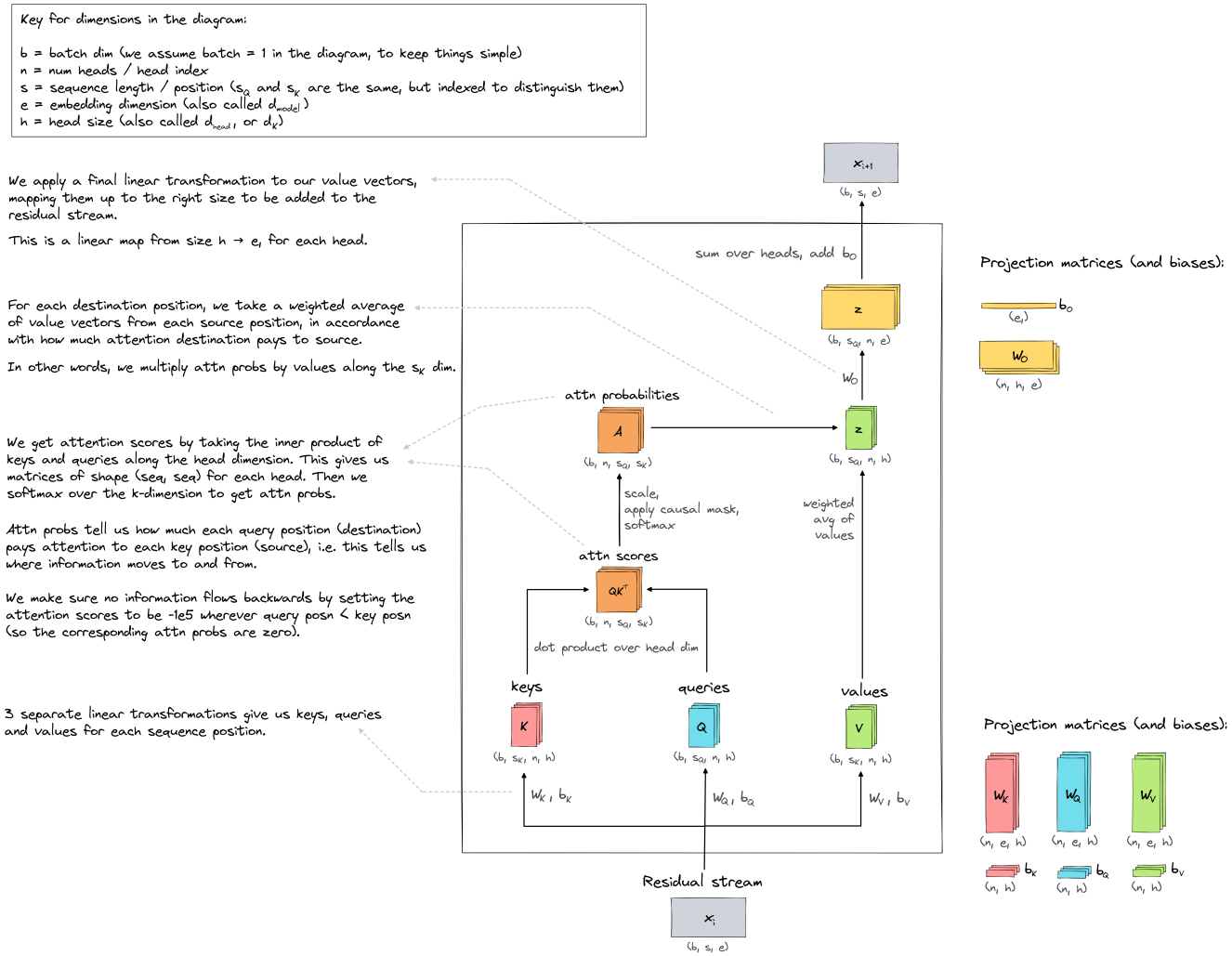
You should spend up to 30-45 minutes on this exercise.

- **Step 1:** Produce an attention pattern - for each destination token, probability distribution over previous tokens (including current token)
  - Linear map from input -> query, key shape `[batch, seq_posn, head_index, d_head]`
  - Dot product every *pair* of queries and keys to get attn_scores `[batch, head_index, query_pos, key_pos]` (query = dest, key = source)
  - **Scale** and mask `attn_scores` to make it lower triangular, i.e. causal
  - Softmax along the `key_pos` dimension, to get a probability distribution for each query (destination) token - this is our attention pattern!
- **Step 2:** Move information from source tokens to destination token using attention pattern (move = apply linear map)
  - Linear map from input -> value `[batch, key_pos, head_index, d_head]`
  - Mix along the `key_pos` with attn pattern to get `z`, which is a weighted average of the value vectors `[batch, query_pos, head_index, d_head]`
  - Map to output, `[batch, position, d_model]` (position = query_pos, we've summed over all heads)

Note - when we say **scale**, we mean dividing by `sqrt(d_head)`. The purpose of this is to avoid vanishing gradients (which is a big problem when we're dealing with a function like softmax - if one of the values is much larger than all the others, the probabilities will be close to 0 or 1, and the gradients will be close to 0).

Below is a much larger, more detailed version of the attention head diagram from earlier. This should give you an idea of the actual tensor operations involved. A few clarifications on this diagram:

- Whenever there is a third dimension shown in the pictures, this refers to the `head_index` dimension. We can see that all operations within the attention layer are done independently for each head.
- The objects in the box are activations; they have a batch dimension (for simplicity, we assume the batch dimension is 1 in the diagram). The objects to the right of the box are our parameters (weights and biases); they have no batch dimension.
- We arrange the keys, queries and values as `(batch, seq_pos, head_idx, d_head)`, because the biases have shape `(head_idx, d_head)`, so this makes it convenient to add the biases (recall the rules of array broadcasting!).

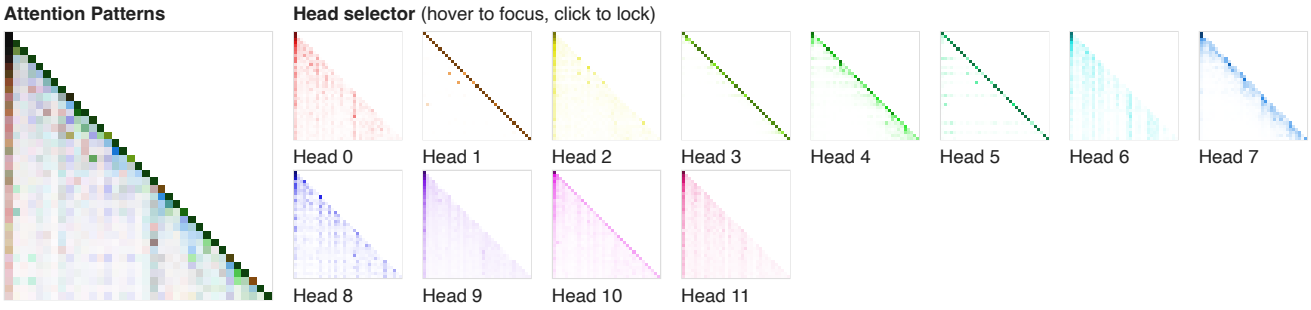


▶ **A few extra notes on attention (optional)**

First, it's useful to visualize and play around with attention patterns - what exactly are we looking at here? (Click on a head to lock onto just showing that head's pattern, it'll make it easier to interpret)

```
import circuitsvis as cv
from IPython.display import display

html = cv.attention.attention_patterns(
    tokens=reference_gpt2.to_str_tokens(reference_text),
    attention=cache["pattern", 0][0]
)
display(html)
```



You can also use the `attention_heads` function, which has similar syntax but presents the information in a different (sometimes more helpful) way.

▶ Help - my `attention_heads` plots are behaving weirdly.

Note - don't worry if you don't get 100% accuracy here; the tests are pretty stringent. Even things like having your `einsum` input arguments in a different order might result in the output being very slightly different. You should be getting at least 99% accuracy though, so if the value is lower then this it probably means you've made a mistake somewhere.

Also, this implementation will probably be the most challenging exercise on this page, so don't worry if it takes you some time! You should look at parts of the solution if you're stuck.

A couple of notes / hints:

- Don't forget the attention score scaling (this should come before the masking).
- You can use `torch.where`, or the `torch.masked_fill` function when masking the attention scores.
- The `"IGNORE"` buffer is a very large negative number. This is the value you should mask your attention scores with (i.e. set them to this number wherever you want the probabilities to be zero). We indicate the existence of a `self.IGNORE` attribute to VSCode's typechecker via the line `IGNORE: Float[Tensor, ""]` in the second line of the code below.

▶ Question - why do you think we mask the attention scores by setting them to a large negative number, rather than the attention probabilities by setting them to zero?

```
class Attention(nn.Module):
    IGNORE: Float[Tensor, ""]

    def __init__(self, cfg: Config):
        super().__init__()
        self.cfg = cfg
        self.W_Q = nn.Parameter(t.empty((cfg.n_heads, cfg.d_model, cfg.d_head)))
        self.W_K = nn.Parameter(t.empty((cfg.n_heads, cfg.d_model, cfg.d_head)))
        self.W_V = nn.Parameter(t.empty((cfg.n_heads, cfg.d_model, cfg.d_head)))
        self.W_O = nn.Parameter(t.empty((cfg.n_heads, cfg.d_head, cfg.d_model)))
        self.b_Q = nn.Parameter(t.zeros((cfg.n_heads, cfg.d_head)))
        self.b_K = nn.Parameter(t.zeros((cfg.n_heads, cfg.d_head)))
        self.b_V = nn.Parameter(t.zeros((cfg.n_heads, cfg.d_head)))
        self.b_O = nn.Parameter(t.zeros((cfg.d_model)))
        nn.init.normal_(self.W_Q, std=self.cfg.init_range)
        nn.init.normal_(self.W_K, std=self.cfg.init_range)
        nn.init.normal_(self.W_V, std=self.cfg.init_range)
        nn.init.normal_(self.W_O, std=self.cfg.init_range)
        self.register_buffer("IGNORE", t.tensor(-1e5, dtype=t.float32, device=device))

    def forward(
        self, normalized_resid_pre: Float[Tensor, "batch posn d_model"]
    ) -> Float[Tensor, "batch posn d_model"]:
        # SOLUTION
        # Calculate query, key and value vectors
        q = einops.einsum(
            normalized_resid_pre, self.W_Q,
            "batch posn d_model, nheads d_model d_head -> batch posn nheads d_head",
        ) + self.b_Q
        k = einops.einsum(
            normalized_resid_pre, self.W_K,
            "batch posn d_model, nheads d_model d_head -> batch posn nheads d_head",
        ) + self.b_K
        v = einops.einsum(
            normalized_resid_pre, self.W_V,
            "batch posn d_model, nheads d_model d_head -> batch posn nheads d_head",
        ) + self.b_V

        # Calculate attention scores, then scale and mask, and apply softmax to get probabilities
        attn_scores = einops.einsum(
            q, k,
            "batch posn_Q nheads d_head, batch posn_K nheads d_head -> batch nheads posn_Q posn_K",
        )
        attn_scores_masked = self.apply_causal_mask(attn_scores / self.cfg.d_head ** 0.5)
        attn_pattern = attn_scores_masked.softmax(-1)

        # Take weighted sum of value vectors, according to attention probabilities
        z = einops.einsum(
            v, attn_pattern,
            "batch posn_K nheads d_head, batch nheads posn_Q posn_K -> batch posn_Q nheads d_head",
        )

        # Calculate output (by applying matrix W_O and summing over heads, then adding bias b_O)
        attn_out = einops.einsum(
            z, self.W_O,
            "batch posn_Q nheads d_head, nheads d_head d_model -> batch posn_Q d_model",
        ) + self.b_O
```

```
        return attn_out

    def apply_causal_mask(
        self, attn_scores: Float[Tensor, "batch n_heads query_pos key_pos"]
    ) -> Float[Tensor, "batch n_heads query_pos key_pos"]:
        '''
        Applies a causal mask to attention scores, and returns masked scores.
        '''
        # SOLUTION
        # Define a mask that is True for all positions we want to set probabilities to zero for
        all_ones = t.ones(attn_scores.size(-2), attn_scores.size(-1), device=attn_scores.device)
        mask = t.triu(all_ones, diagonal=1).bool()
        # Apply the mask to attention scores, then return the masked scores
        attn_scores.masked_fill_(mask, self.IGNORE)
        return attn_scores


rand_float_test(Attention, [2, 4, 768])
load_gpt2_test(Attention, reference_gpt2.blocks[0].attn, cache["normalized", 0, "ln1"])
```

```
Input shape: torch.Size([2, 4, 768])
Output shape: torch.Size([2, 4, 768])

Input shape: torch.Size([1, 35, 768])
Output shape: torch.Size([1, 35, 768])
Reference output shape: torch.Size([1, 35, 768])

100.00% of the values are correct
```

▶ Hint (pseudocode for both functions)

▶ Solution

MLP

Difficulty: 🟠🟠⚪⚪⚪
Importance: 🟠🟠🟠🟠⚪

You should spend up to 10-15 minutes on this exercise.

Next, you should implement the MLP layer, which consists of:

- A linear layer, with weight `W_in`, bias `b_in`
- A nonlinear functino (we usually use GELU; the function `gelu_new` has been imported for this purpose)
- A linear layer, with weight `W_out`, bias `b_out`

```
class MLP(nn.Module):
    def __init__(self, cfg: Config):
        super().__init__()
        self.cfg = cfg
        self.W_in = nn.Parameter(t.empty((cfg.d_model, cfg.d_mlp)))
        self.W_out = nn.Parameter(t.empty((cfg.d_mlp, cfg.d_model)))
        self.b_in = nn.Parameter(t.zeros((cfg.d_mlp)))
        self.b_out = nn.Parameter(t.zeros((cfg.d_model)))
        nn.init.normal_(self.W_in, std=self.cfg.init_range)
        nn.init.normal_(self.W_out, std=self.cfg.init_range)

    def forward(
        self, normalized_resid_mid: Float[Tensor, "batch posn d_model"]
    ) -> Float[Tensor, "batch posn d_model"]:
        # SOLUTION
        pre = einops.einsum(
            normalized_resid_mid, self.W_in,
            "batch position d_model, d_model d_mlp -> batch position d_mlp",
        ) + self.b_in
        post = gelu_new(pre)
        mlp_out = einops.einsum(
            post, self.W_out,
            "batch position d_mlp, d_mlp d_model -> batch position d_model",
        ) + self.b_out
        return mlp_out


rand_float_test(MLP, [2, 4, 768])
load_gpt2_test(MLP, reference_gpt2.blocks[0].mlp, cache["normalized", 0, "ln2"])
```

```
Input shape: torch.Size([2, 4, 768])
Output shape: torch.Size([2, 4, 768])
```

```
Input shape: torch.Size([1, 35, 768])
Output shape: torch.Size([1, 35, 768])
Reference output shape: torch.Size([1, 35, 768])

100.00% of the values are correct
```

▶ Solution

Transformer Block

Difficulty: 🟠🟠⚪⚪⚪
Importance: 🟠🟠🟠⚪⚪

You should spend up to ~10 minutes on this exercise.

Now, we can put together the attention, MLP and layernorms into a single transformer block. Remember to implement the residual connections correctly!

```
class TransformerBlock(nn.Module):
    def __init__(self, cfg: Config):
        super().__init__()
        self.cfg = cfg
        self.ln1 = LayerNorm(cfg)
        self.attn = Attention(cfg)
        self.ln2 = LayerNorm(cfg)
        self.mlp = MLP(cfg)

    def forward(
        self, resid_pre: Float[Tensor, "batch position d_model"]
    ) -> Float[Tensor, "batch position d_model"]:
        # SOLUTION
        resid_mid = self.attn(self.ln1(resid_pre)) + resid_pre
        resid_post = self.mlp(self.ln2(resid_mid)) + resid_mid
        return resid_post


rand_float_test(TransformerBlock, [2, 4, 768])
load_gpt2_test(TransformerBlock, reference_gpt2.blocks[0], cache["resid_pre", 0])
```

```
Input shape: torch.Size([2, 4, 768])
Output shape: torch.Size([2, 4, 768])

Input shape: torch.Size([1, 35, 768])
Output shape: torch.Size([1, 35, 768])
Reference output shape: torch.Size([1, 35, 768])

100.00% of the values are correct
```

▶ Solution

Unembedding

Difficulty: 🟠🟠⚪⚪⚪
Importance: 🟠🟠🟠⚪⚪

You should spend up to ~10 minutes on this exercise.

The unembedding is jus a linear layer (with weight `W_U` and bias `b_U`).

```
class Unembed(nn.Module):
    def __init__(self, cfg):
        super().__init__()
        self.cfg = cfg
        self.W_U = nn.Parameter(t.empty((cfg.d_model, cfg.d_vocab)))
        nn.init.normal_(self.W_U, std=self.cfg.init_range)
        self.b_U = nn.Parameter(t.zeros((cfg.d_vocab), requires_grad=False))

    def forward(
        self, normalized_resid_final: Float[Tensor, "batch position d_model"]
    ) -> Float[Tensor, "batch position d_vocab"]:
```

```
        # SOLUTION
        return einops.einsum(
            normalized_resid_final, self.W_U,
            "batch posn d_model, d_model d_vocab -> batch posn d_vocab",
        ) + self.b_U
        # Or, could just do `normalized_resid_final @ self.W_U + self.b_U`


rand_float_test(Unembed, [2, 4, 768])
load_gpt2_test(Unembed, reference_gpt2.unembed, cache["ln_final.hook_normalized"])
```

```
Input shape: torch.Size([2, 4, 768])
Output shape: torch.Size([2, 4, 50257])

Input shape: torch.Size([1, 35, 768])
Output shape: torch.Size([1, 35, 50257])
Reference output shape: torch.Size([1, 35, 50257])

100.00% of the values are correct
```

▶ Solution

Full Transformer

Difficulty: 🟠🟠⚪⚪⚪
Importance: 🟠🟠🟠⚪⚪

You should spend up to ~10 minutes on this exercise.

```
class DemoTransformer(nn.Module):
    def __init__(self, cfg: Config):
        super().__init__()
        self.cfg = cfg
        self.embed = Embed(cfg)
        self.pos_embed = PosEmbed(cfg)
        self.blocks = nn.ModuleList([TransformerBlock(cfg) for _ in range(cfg.n_layers)])
        self.ln_final = LayerNorm(cfg)
        self.unembed = Unembed(cfg)

    def forward(self, tokens: Int[Tensor, "batch position"]) -> Float[Tensor, "batch position d_vocab"]:
        # SOLUTION
        residual = self.embed(tokens) + self.pos_embed(tokens)
        for block in self.blocks:
            residual = block(residual)
        logits = self.unembed(self.ln_final(residual))
        return logits


rand_int_test(DemoTransformer, [2, 4])
load_gpt2_test(DemoTransformer, reference_gpt2, tokens)
```

```
Input shape: torch.Size([2, 4])
Output shape: torch.Size([2, 4, 50257])

Input shape: torch.Size([1, 45])
Output shape: torch.Size([1, 45, 50257])
Reference output shape: torch.Size([1, 45, 50257])

100.00% of the values are correct
```

▶ Solution

Try it out!

```
demo_gpt2 = DemoTransformer(Config(debug=False)).to(device)
demo_gpt2.load_state_dict(reference_gpt2.state_dict(), strict=False)

demo_logits = demo_gpt2(tokens)
```

Let's take a test string, and calculate the loss!

We're using the formula for **cross-entropy loss**. The cross entropy loss between a modelled distribution Q and target distribution P is:

$$-\sum_x P(x)\log Q(x)$$

In the case where P is just the empirical distribution from target classes (i.e. $P(x^*) = 1$ for the correct class x^*) then this becomes:

$$-\log Q(x^*)$$

in other words, the negative log prob of the true classification.

```
def get_log_probs(
    logits: Float[Tensor, "batch posn d_vocab"],
    tokens: Int[Tensor, "batch posn"]
) -> Float[Tensor, "batch posn-1"]:

    log_probs = logits.log_softmax(dim=-1)
    # Get logprobs the first seq_len-1 predictions (so we can compare them with the actual next tokens)
    log_probs_for_tokens = log_probs[:, :-1].gather(dim=-1, index=tokens[:, 1:].unsqueeze(-1)).squeeze(-1)

    return log_probs_for_tokens


pred_log_probs = get_log_probs(demo_logits, tokens)
print(f"Avg cross entropy loss: {-pred_log_probs.mean():.4f}")
print(f"Avg cross entropy loss for uniform distribution: {math.log(demo_gpt2.cfg.d_vocab):4f}")
print(f"Avg probability assigned to correct token: {pred_log_probs.exp().mean():4f}")
```

```
Avg cross entropy loss: 4.0442
Avg cross entropy loss for uniform distribution: 10.824905
Avg probability assigned to correct token: 0.098628
```

We can also greedily generate text, by taking the most likely next token and continually appending it to our prompt before feeding it back into the model:

```
test_string = '''The Total Perspective Vortex derives its picture of the whole Universe on the principle of'''
for i in tqdm(range(100)):
    test_tokens = reference_gpt2.to_tokens(test_string).to(device)
    demo_logits = demo_gpt2(test_tokens)
    test_string += reference_gpt2.tokenizer.decode(demo_logits[-1, -1].argmax())

print(test_string)
```

```
100%                                                  100/100 [00:03<00:00, 27.90it/s]
The Total Perspective Vortex derives its picture of the whole Universe on the principle of the total perspective. The tot
```

In later sections, we'll learn to generate text in slightly more interesting ways than just argmaxing the output.

3️⃣ Training a Transformer

> Learning objectives
>
> - Understand how to train a transformer from scratch
> - Write a basic transformer training loop with PyTorch Lightning
> - Interpret the transformer's falling cross entropy loss with reference to features of the training data (e.g. bigram frequencies)

Now that we've built our transformer, and verified that it performs as expected when we load in weights, let's try training it from scratch!

This is a lightweight demonstration of how you can actually train your own GPT-2 with this code! Here we train a tiny model on a tiny dataset, but it's fundamentally the same code for training a larger/more real model (though you'll need beefier GPUs and data parallelism to do it remotely efficiently, and fancier parallelism for much bigger ones).

For our purposes, we'll train 2L 4 heads per layer model, with context length 256, for 1000 steps of batch size 8, just to show what it looks like (and so the notebook doesn't melt your colab lol).

Create Model

```
model_cfg = Config(
    debug=False,
    d_model=256,
    n_heads=4,
```

```
    d_head=64,
    d_mlp=1024,
    n_layers=2,
    n_ctx=256,
    d_vocab=reference_gpt2.cfg.d_vocab
)
model = DemoTransformer(model_cfg)
```

Training Args

Note, for this optimization we'll be using **weight decay**.

```
@dataclass
class TransformerTrainingArgs():
    batch_size = 8
    max_epochs = 1
    max_steps = 1000
    log_every = 10
    lr = 1e-3
    weight_decay = 1e-2
    log_dir: str = os.getcwd() + "/logs"
    log_name: str = "day1-transformer"
    run_name: Optional[str] = None
    log_every_n_steps: int = 1

args = TransformerTrainingArgs()
```

Create Data

We load in a tiny dataset I made, with the first 10K entries in the Pile (inspired by Stas' version for OpenWebText!)

```
dataset = datasets.load_dataset("NeelNanda/pile-10k", split="train").remove_columns("meta")
print(dataset)
print(dataset[0]['text'][:100])
```

```
Downloading metadata: 100%  921/921 [00:00<00:00, 68.5kB/s]
Downloading readme: 100%  373/373 [00:00<00:00, 28.3kB/s]
Downloading and preparing dataset None/None (download: 31.72 MiB, generated: 58.43 MiB, post-processed: Unknown size, tot
Downloading data files: 100%  1/1 [00:01<00:00, 1.44s/it]
Downloading data: 100%  33.3M/33.3M [00:00<00:00, 54.4MB/s]
Extracting data files: 100%  1/1 [00:00<00:00, 56.55it/s]

Dataset parquet downloaded and prepared to /root/.cache/huggingface/datasets/NeelNanda___parquet/NeelNanda--pile-10k-72f5
Dataset({
    features: ['text'],
    num_rows: 10000
})
It is done, and submitted. You can play “Survival of the Tastiest” on Android, and on the web. Playi
```

`tokenize_and_concatenate` is a useful function which takes our dataset of strings, and returns a dataset of token IDs ready to feed into the model. We then create a dataloader from this tokenized dataset.

```
tokenized_dataset = tokenize_and_concatenate(dataset, reference_gpt2.tokenizer, streaming=False, max_length=model.cfg.n_ctx
data_loader = DataLoader(tokenized_dataset, batch_size=args.batch_size, shuffle=True, num_workers=4, pin_memory=True)
```

```
Token indices sequence length is longer than the specified maximum sequence length for this model (80023 > 1024). Running
Token indices sequence length is longer than the specified maximum sequence length for this model (101051 > 1024). Runnir
Token indices sequence length is longer than the specified maximum sequence length for this model (155995 > 1024). Runnir
Token indices sequence length is longer than the specified maximum sequence length for this model (229134 > 1024). Runnir
```

When we iterate through `data_loader`, we will find dictionaries with the single key `"tokens"`, which maps to a tensor of token IDs with shape `(batch, seq_len)`.

```
first_batch = data_loader.dataset[:args.batch_size]

```

```
print(first_batch.keys())
print(first_batch['tokens'].shape)
```

```
dict_keys(['tokens'])
torch.Size([8, 256])
```

Training Loop

If you did the material on PyTorch Lightning during the first week, this should all be familiar to you. If not, a little refresher:

▶ Click here for a basic refresher on PyTorch Lightning & Weights and Biases

Exercise - write training loop

```
Difficulty: 🟠🟠⚪⚪⚪
Importance: 🟠🟠🟠🟠⚪

You should spend up to 10-15 minutes on this exercise.
```

You should fill in the methods below. Some guidance:

- Remember we were able to calculate cross entropy loss using the `get_log_probs` function in the previous section.
- You should use the optimizer `t.optim.AdamW` (Adam with weight decay), and with hyperparameters `lr` and `weight_decay` taken from your `TransformerTrainingArgs` dataclass instance.

If you've not encountered `train_dataloader` before, this function returns a dataloader (or else something which you iterate through to get the batches used in the `training_step` function). Also, the `forward` function of the model overrides the default behaviour when you call `self(input)` within another method. Neither of these are strictly necessary, but they're useful Lightning features for keeping your code clean.

```
class LitTransformer(pl.LightningModule):
    def __init__(self, args: TransformerTrainingArgs, model: DemoTransformer, data_loader: DataLoader):
        super().__init__()
        self.model = model
        self.cfg = model.cfg
        self.args = args
        self.data_loader = data_loader

    def forward(self, tokens: Int[Tensor, "batch position"]) -> Float[Tensor, "batch position d_vocab"]:
        logits = self.model(tokens)
        return logits

    def training_step(self, batch: Dict[str, Tensor], batch_idx: int) -> Float[Tensor, ""]:
        '''
        Here you compute and return the training loss and some additional metrics for e.g.
        the progress bar or logger.
        '''
        # SOLUTION
        tokens = batch["tokens"].to(device)
        logits = self.model(tokens)
        loss = -get_log_probs(logits, tokens).mean()
        self.log("train_loss", loss)
        return loss

    def configure_optimizers(self):
        '''
        Choose what optimizers and learning-rate schedulers to use in your optimization.
        '''
        # SOLUTION
        optimizer = t.optim.AdamW(self.model.parameters(), lr=self.args.lr, weight_decay=self.args.weight_decay)
        return optimizer

    def train_dataloader(self):
        return self.data_loader
```

▶ Solution

```
litmodel = LitTransformer(args, model, data_loader)
logger = WandbLogger(save_dir=args.log_dir, project=args.log_name, name=args.run_name)

trainer = pl.Trainer(
    max_epochs=args.max_epochs,
    logger=logger,
    log_every_n_steps=args.log_every_n_steps
)
```

```
trainer.fit(model=litmodel, train_dataloaders=litmodel.data_loader)
wandb.finish()
```

```
wandb: Appending key for api.wandb.ai to your netrc file: /root/.netrc
wandb: WARNING Path /content/chapter1_transformers/exercises/logs/wandb/ wasn't writable, using system temp directory.
Tracking run with wandb version 0.15.3
Run data is saved locally in /tmp/wandb/run-20230529_214038-jr088lu1
Syncing run chocolate-feather-13 to Weights & Biases (docs)
View project at https://wandb.ai/callum-mcdougall/day1-transformer
View run at https://wandb.ai/callum-mcdougall/day1-transformer/runs/jr088lu1
INFO:pytorch_lightning.utilities.rank_zero:GPU available: True (cuda), used: True
INFO:pytorch_lightning.utilities.rank_zero:TPU available: False, using: 0 TPU cores
INFO:pytorch_lightning.utilities.rank_zero:IPU available: False, using: 0 IPUs
INFO:pytorch_lightning.utilities.rank_zero:HPU available: False, using: 0 HPUs
INFO:pytorch_lightning.utilities.rank_zero:You are using a CUDA device ('NVIDIA A100-SXM4-40GB') that has Tensor Cores. T
INFO:pytorch_lightning.accelerators.cuda:LOCAL_RANK: 0 - CUDA_VISIBLE_DEVICES: [0]
INFO:pytorch_lightning.callbacks.model_summary:
  | Name  | Type            | Params
-----------------------------------------
0 | model | DemoTransformer | 27.4 M
-----------------------------------------
27.4 M    Trainable params
0         Non-trainable params
27.4 M    Total params
109.710   Total estimated model params size (MB)
Epoch 0: 100%                                                    8506/8506 [05:22<00:00, 26.39it/s, v_num=8lu1]
INFO:pytorch_lightning.utilities.rank_zero:`Trainer.fit` stopped: `max_epochs=1` reached.
Waiting for W&B process to finish... (success).
0.002 MB of 0.002 MB uploaded (0.000 MB deduped)

Run history:                                  Run summary:

epoch                                         epoch 0
train_loss                                    train_loss 3.19731
trainer/global_step                           trainer/global_step 8505

View run chocolate-feather-13 at: https://wandb.ai/callum-mcdougall/day1-transformer/runs/jr088lu1
Synced 5 W&B file(s), 0 media file(s), 0 artifact file(s) and 0 other file(s)
Find logs at: ./logs/wandb/run-20230529_214038-jr088lu1/logs
```

> Note - to see these patterns more clearly in Weights and Biases, you can click on the **edit panel** of your plot (the small pencil symbol at the top-right), then move the **smoothing** slider to the right.

A note on this loss curve (optional)

What's up with the shape of our loss curve? It seems like we start at around 10-11, drops down very fast, but then levels out. It turns out, this is all to do with the kinds of algorithms the model learns during training.

When it starts out, your model will be outputting random noise, which might look a lot like "predict each token with approximately uniform probability", i.e. $Q(x) = 1/d_{\text{vocab}}$ for all x. This gives us a cross entropy loss of $\log(d_{\text{vocab}})$.

```
d_vocab = model.cfg.d_vocab

print(f"d_vocab = {d_vocab}")
print(f"Cross entropy loss on uniform distribution = {math.log(d_vocab)}")
```

```
d_vocab = 50257
Cross entropy loss on uniform distribution = 10.82490511970208
```

The next thing we might expect the model to learn is the frequencies of words in the english language. After all, small common tokens like `" and"` or `" the"` might appear much more frequently than others. This would give us an average cross entropy loss of:

$$-\sum_x p_x \log p_x$$

where p_x is the actual frequency of the word in our training data.

We can evaluate this quantity as follows:

```
toks = tokenized_dataset[:]["tokens"].flatten()

```

```
d_vocab = model.cfg.d_vocab
freqs = t.bincount(toks, minlength=d_vocab)
probs = freqs.float() / freqs.sum()

distn = t.distributions.categorical.Categorical(probs=probs)
entropy = distn.entropy()

print(f"Entropy of training data = {entropy}")
```

```
Entropy of training data = 7.349369525909424
```

After unigram frequencies, the next thing our model usually learns is **bigram frequencies** (i.e. the frequency of pairs of adjacent tokens in the training data). For instance, `"I"` and `" am"` are common tokens, but their bigram frequency is much higher than it would be if they occurred independently. Bigram frequencies actually take you pretty far, since they also help with:

- Some simple grammatical rules (e.g. a full stop being followed by a capitalized word)
- Weird quirks of tokenization (e.g. `" manip"` being followed by `"ulative"`)
- Common names (e.g. `"Barack"` being followed by `" Obama"`)

After approximating bigram frequencies, we need to start using smarter techniques, like trigrams (which can only be implemented using attention heads), **induction heads** (which we'll learn a lot more about in the next set of exercises!), and fact memorization or more basic grammar and syntax rules. Marginal improvement starts getting a lot harder around here, leading to a flattening of our loss curve.

4️⃣ Sampling from a Transformer

> Learning objectives
>
> - Learn how to sample from a transformer
>   - This includes basic methods like greedy search or top-k, and more advanced methods like beam search
> - Learn how to cache the output of a transformer, so that it can be used to generate text more efficiently
>   - Optionally, rewrite your sampling functions to make use of your caching methods

One obvious method to sample tokens from a distribution would be to always take the token assigned the highest probability. But this can lead to some boring and repetitive outcomes, and at worst it can lock our transformer's output into a loop.

First, you should read HuggingFace's blog post How to generate text: using different decoding methods for language generation with Transformers.

Once you've done that, you can work through the `TransformerSampler` class below, and implement the different sampling methods. Each method will come with its own tests, and demo code for you to run.

```
model_cfg = Config()
model = DemoTransformer(model_cfg).to(device)
model.load_state_dict(reference_gpt2.state_dict(), strict=False)

tokenizer = reference_gpt2.tokenizer
```

```
class TransformerSampler:

    def __init__(self, model: DemoTransformer, tokenizer: GPT2TokenizerFast):
        self.model = model
        self.cfg = model.cfg
        self.tokenizer = tokenizer

    @t.inference_mode()
    def sample(self, prompt: str, max_tokens_generated=100, verbose=False, **kwargs):
        '''
        Returns a string of autoregressively generated text, starting from the prompt.

        Sampling terminates at max_tokens_generated, or when the model generates an
        end-of-sequence token.

        kwargs are passed to sample_next_token, to give detailed instructions on how
        new tokens are chosen.
        '''
        # SOLUTION
        self.model.eval()
        input_ids = self.tokenizer.encode(prompt, return_tensors="pt").to(device)[0]

        for i in range(max_tokens_generated):
```

```
            # Get new logits (make sure we don't pass in more tokens than the model's context length)
            logits = self.model(input_ids[None, -self.cfg.n_ctx:])
            # We only take logits for the last token, because this is what we're sampling
            logits = logits[0, -1]
            # Get next token (as a tensor of size (1, 1) so we can concat it to input_ids)
            next_token = t.tensor([TransformerSampler.sample_next_token(input_ids, logits, **kwargs)], device=device)
            # Create new input ids string, with shape (1, old_seq_len + 1)
            input_ids = t.cat([input_ids, next_token], dim=-1)
            # Print out results, if required
            if verbose:
                print(self.tokenizer.decode(input_ids), end="\r")
            # If our new token was the end-of-text token, stop
            if next_token == getattr(self.tokenizer, "eos_token_id", None):
                break

        return self.tokenizer.decode(input_ids)



    @t.inference_mode()
    def beam_search(
        self,
        prompt: str,
        num_return_sequences: int,
        num_beams: int,
        max_new_tokens: int,
        no_repeat_ngram_size: int = 0,
        verbose=False
    ) -> List[Tuple[float, t.Tensor]]:
        '''
        Returns a string of autoregressively generated text, starting from the prompt.

        Sampling terminates at max_tokens_generated, or when the model generates an
        end-of-sequence token.

        kwargs are passed to sample_next_token, to give detailed instructions on how
        new tokens are chosen.
        '''
        pass


    @staticmethod
    def sample_next_token(
        input_ids: Int[Tensor, "seq_len"],
        logits: Float[Tensor, "seq_len d_vocab"],
        temperature=1.0,
        top_k=0,
        top_p=0.0,
        frequency_penalty=0.0,
        seed=None
    ):
        assert input_ids.ndim == 1, "input_ids should be a 1D sequence of token ids"
        assert temperature >= 0, "Temperature should be non-negative"
        assert 0 <= top_p <= 1.0, "Top-p must be a probability"
        assert 0 <= top_k, "Top-k must be non-negative"
        assert not (top_p != 0 and top_k != 0), "At most one of top-p and top-k supported"

        # Set random seeds for reproducibility
        if seed is not None:
            t.manual_seed(seed)
            np.random.seed(seed)

        # Apply all the specialized sampling methods
        if temperature == 0:
            return TransformerSampler.greedy_search(logits)
        elif temperature != 1.0:
            logits = TransformerSampler.apply_temperature(logits, temperature)
        if frequency_penalty != 0.0:
            logits = TransformerSampler.apply_frequency_penalty(input_ids, logits, frequency_penalty)
        if top_k > 0:
            return TransformerSampler.sample_top_k(logits, top_k)
        if top_p > 0.0:
            return TransformerSampler.sample_top_p(logits, top_p)
        return TransformerSampler.sample_basic(logits)


    @staticmethod
    def greedy_search(logits: Float[Tensor, "d_vocab"]) -> int:
        '''
        Returns the most likely token (as an int).
        '''
        out = logits.argmax().item()
```

```
        return out


    @staticmethod
    def apply_temperature(logits: Float[Tensor, "d_vocab"], temperature: float) -> Float[Tensor, "d_vocab"]:
        '''
        Applies temperature scaling to the logits.
        '''
        # SOLUTION
        return logits / temperature



    @staticmethod
    def apply_frequency_penalty(input_ids: Int[Tensor, "seq_len"], logits: Float[Tensor, "d_vocab"], freq_penalty: float) -
        '''
        Applies a frequency penalty to the logits.
        '''
        # SOLUTION
        d_vocab = logits.size(0)
        id_freqs = t.bincount(input_ids, minlength=d_vocab)
        return logits - freq_penalty * id_freqs



    @staticmethod
    def sample_basic(logits: Float[Tensor, "d_vocab"]) -> int:
        '''
        Samples from the distribution defined by the logits.
        '''
        # SOLUTION
        sampled_token = t.distributions.categorical.Categorical(logits=logits).sample()
        return sampled_token.item()



    @staticmethod
    def sample_top_k(logits: Float[Tensor, "d_vocab"], k: int) -> int:
        '''
        Samples from the top k most likely tokens.
        '''
        # SOLUTION
        top_k_logits, top_k_token_ids = logits.topk(k)
        # Get sampled token (which is an index corresponding to the list of top-k tokens)
        sampled_token_idx = t.distributions.categorical.Categorical(logits=top_k_logits).sample()
        # Get the actual token id, as an int
        return top_k_token_ids[sampled_token_idx].item()



    @staticmethod
    def sample_top_p(logits: Float[Tensor, "d_vocab"], top_p: float, min_tokens_to_keep: int = 1) -> int:
        '''
        Samples from the most likely tokens which make up at least p cumulative probability.
        '''
        # SOLUTION
        # Sort logits, and get cumulative probabilities
        logits_sorted, indices = logits.sort(descending=True, stable=True)
        cumul_probs = logits_sorted.softmax(-1).cumsum(-1)
        # Choose which tokens to keep, in the set we sample from
        n_keep = t.searchsorted(cumul_probs, top_p, side="left").item() + 1
        n_keep = max(n_keep, min_tokens_to_keep)
        keep_idx = indices[:n_keep]
        keep_logits = logits[keep_idx]
        # Perform the sampling
        sample = t.distributions.categorical.Categorical(logits=keep_logits).sample()
        return keep_idx[sample].item()
```

Main Sampling Function

The first thing you should do is implement the `sample` method.

Exercise - implement `sample`

```
Difficulty: 🟠🟠🟠⚪⚪
Importance: 🟠🟠🟠⚪⚪

You should spend up to 20-25 minutes on this exercise.
```

This function takes in a prompt (in the form of a string), encodes it as a sequence of token ids using `self.tokenizer.encode`, and then continually generates new tokens by repeating the following steps:

1. Passing the tokenized prompt through the model to get logits,
2. Taking the logit vector corresponding to the last token in the prompt (i.e. the prediction for the *next* token),
3. Sampling from this distribution to get a new token, using `self.sample_next_token(input_ids, logits, **kwargs)` (here, `kwargs` contains all the sampling-specific arguments, e.g. temperature, top-k, etc.),
4. Appending this new token to the input tokens, and repeating the process until we meet one of two termination critera:
    - We generate `max_tokens_generated` new tokens, or
    - We generate the end-of-sequence token (which we can access via `self.tokenizer.eos_token_id`).

Finally, we use `self.tokenizer.decode` to convert the generated token ids back into a string, and return this string.

We also have a `verbose` argument - when this is true you can print your output while it's being sampled.

Below is some code which tests your sampling function by performing greedy sampling (which means always choosing the most likely next token at each step).

▶ Why does `temperature=0.0` correspond to greedy sampling?

A few hints:

- Don't forget about tensor shapes! Your model's input should always have a batch dimension, i.e. it should be shape `(1, seq_len)`.
- Also remember to have your tensors be on the same device (we have a global `device` variable).
- Remember to put your model in evaluation mode, using `model.eval()`.

```
sampler = TransformerSampler(model, tokenizer)

prompt = "Jingle bells, jingle bells, jingle all the way"
print(f"Greedy decoding with prompt: {prompt!r}\n")

output = sampler.sample(prompt, max_tokens_generated=8, temperature=0.0)
print(f"Your model said: {output!r}\n")

expected = "Jingle bells, jingle bells, jingle all the way up to the top of the mountain."
assert output == expected

print("Tests passed!")
```

```
Greedy decoding with prompt: 'Jingle bells, jingle bells, jingle all the way'

Your model said: 'Jingle bells, jingle bells, jingle all the way up to the top of the mountain.'

Tests passed!
```

▶ Solution

Sampling with Categorical

Now, we'll move into implementing specific sampling methods.

PyTorch provides a [`distributions` package] with a number of convenient methods for sampling from various distributions.

For now, we just need [`t.distributions.categorical.Categorical`]. Use this to implement `sample_basic`, which just samples from the provided logits (which may have already been modified by the temperature and frequency penalties).

Note that this will be slow since we aren't batching the samples, but don't worry about speed for now.

Exercise - Basic Sampling

```
Difficulty: 🟠🟠⚪⚪⚪
Importance: 🟠🟠⚪⚪⚪

You should spend up to 5-10 minutes on this exercise.
```

Implement basic sampling in the `TransformerSampler` class above, then run the code below to verify your solution works.

```
prompt = "John and Mary went to the"
input_ids = tokenizer.encode(prompt, return_tensors="pt").to(device)
logits = model(input_ids)[0, -1]

expected_top_5 = {
    " church": 0.0648,
    " house": 0.0367,
```

```
    " temple": 0.0145,
    " same": 0.0104,
    " Church": 0.0097
}
frequency_of_top_5 = defaultdict(int)

N = 10_000
for _ in tqdm(range(N)):
    token = TransformerSampler.sample_next_token(input_ids.squeeze(), logits)
    frequency_of_top_5[tokenizer.decode(token)] += 1

for word in expected_top_5:
    expected_freq = expected_top_5[word]
    observed_freq = frequency_of_top_5[word] / N
    print(f"Word: {word!r:<9}. Expected freq {expected_freq:.4f}, observed freq {observed_freq:.4f}")
    assert abs(observed_freq - expected_freq) < 0.01, "Try increasing N if this fails by a small amount."

print("Tests passed!")
```

```
100%                                                10000/10000 [00:09<00:00, 1081.97it/s]
Word: ' church'. Expected freq 0.0648, observed freq 0.0660
Word: ' house' . Expected freq 0.0367, observed freq 0.0376
Word: ' temple'. Expected freq 0.0145, observed freq 0.0118
Word: ' same'  . Expected freq 0.0104, observed freq 0.0102
Word: ' Church'. Expected freq 0.0097, observed freq 0.0093
Tests passed!
```

▶ Solution

Exercise - Temperature

Difficulty: 🟠⚪⚪⚪⚪
Importance: 🟠🟠⚪⚪⚪

You should spend up to 5-10 minutes on this exercise.

Temperature sounds fancy, but it's literally just dividing the logits by the temperature. You should implement this in your `TransformerSampler` class now.

```
logits = t.tensor([1, 2]).log()

cold_logits = TransformerSampler.apply_temperature(logits, temperature=0.001)
print('A low temperature "sharpens" or "peaks" the distribution: ', cold_logits)
t.testing.assert_close(cold_logits, 1000.0 * logits)

hot_logits = TransformerSampler.apply_temperature(logits, temperature=1000.0)
print("A high temperature flattens the distribution: ", hot_logits)
t.testing.assert_close(hot_logits, 0.001 * logits)

print("Tests passed!")
```

```
A low temperature "sharpens" or "peaks" the distribution:  tensor([  0.0000, 693.1472])
A high temperature flattens the distribution:  tensor([0.0000, 0.0007])
Tests passed!
```

▶ Solution

▶ Question - what is the limit of applying 'sample_basic' after adjusting with temperature, when temperature goes to zero? How about when temperature goes to infinity?

Exercise - Frequency Penalty

Difficulty: 🟠🟠⚪⚪⚪
Importance: 🟠⚪⚪⚪⚪

You should spend up to 10-15 minutes on this exercise.

The frequency penalty is simple as well: count the number of occurrences of each token, then subtract `freq_penalty` for each occurrence. Hint: use `t.bincount` (documentation here) to do this in a vectorized way.

You should implement the `apply_frequency_penalty` method in your `TransformerSampler` class now, then run the cell below to check your solution.

▶ Help - I'm getting a `RuntimeError`; my tensor sizes don't match.

```
bieber_prompt = "And I was like Baby, baby, baby, oh Like, Baby, baby, baby, no Like, Baby, baby, baby, oh I thought you'd
input_ids = tokenizer.encode(bieber_prompt, return_tensors="pt")
logits = t.ones(tokenizer.vocab_size)
penalized_logits = TransformerSampler.apply_frequency_penalty(input_ids.squeeze(), logits, 2.0)

assert penalized_logits[5156].item() == -11, "Expected 6 occurrences of ' baby' with leading space, 1-2*6=-11"
assert penalized_logits[14801].item() == -5, "Expected 3 occurrences of ' Baby' with leading space, 1-2*3=-5"

print("Tests passed!")
```

```
Tests passed!
```

▶ Solution

Sampling - Manual Testing

Run the below cell to get a sense for the `temperature` and `freq_penalty` arguments. Play with your own prompt and try other values.

Note: your model can generate newlines or non-printing characters, so calling `print` on generated text sometimes looks awkward on screen. You can call `repr` on the string before printing to have the string escaped nicely.

```
sampler = TransformerSampler(model, tokenizer)

N_RUNS = 1
your_prompt = "Jingle bells, jingle bells, jingle all the way"
cases = [
    ("High freq penalty", dict(frequency_penalty=100.0)),
    ("Negative freq penalty", dict(frequency_penalty=-3.0)),
    ("Too hot!", dict(temperature=2.0)),
    ("Pleasantly cool", dict(temperature=0.7)),
    ("Pleasantly warm", dict(temperature=0.9)),
    ("Too cold!", dict(temperature=0.01)),
]

table = Table("Name", "Kwargs", "Output", title="Sampling - Manual Testing")

for (name, kwargs) in cases:
    for i in range(N_RUNS):
        output = sampler.sample(your_prompt, max_tokens_generated=24, **kwargs)
        table.add_row(name, repr(kwargs), repr(output) + "\n")

rprint(table)
```

Sampling - Manual Testing

Name	Kwargs	Output
High freq penalty	{'frequency_penalty': 100.0}	'Jingle bells, jingle bells, jingle all the way to our hearts. But I would not strive for a life without tangible things - as throughout history e has found when your'
Negative freq penalty	{'frequency_penalty': -3.0}	'Jingle bells, jingle bells, jingle all the way, bells, bells, bells, bells, bells, bells, bells, bells, bells, bells, bells'
Too hot!	{'temperature': 2.0}	'Jingle bells, jingle bells, jingle all the way through :-) Lucas Elena US Writ Hist Evan Diaz8588 CAT Winner Mar MAX unsurprisingly involved c77 ord archive sanction discovery antiqu'
Pleasantly cool	{'temperature': 0.7}	"Jingle bells, jingle bells, jingle all the way up there. You know, the heart is not for us.\n\nAnd then the world isn't for us."
Pleasantly warm	{'temperature': 0.9}	"Jingle bells, jingle bells, jingle all the way to the Chinese castle. Why are you asking me to explore so far? I'm getting too excited. Say, what"
Too cold!	{'temperature': 0.01}	'Jingle bells, jingle bells, jingle all the way up to the top of the mountain.\n\nThe first time I saw the mountain, I was in the middle of'

Top-K Sampling

Conceptually, the steps in top-k sampling are:

- Find the `top_k` largest probabilities (you can use `torch.topk`)

- Set all other probabilities to zero
- Normalize and sample

Exercise - implement `sample_top_k`

```
Difficulty: 🟠🟠⚪⚪⚪
Importance: 🟠⚪⚪⚪⚪

You should spend up to 5-10 minutes on this exercise.
```

Implement the method `sample_top_k` now. Your implementation should stay in log-space throughout (don't exponentiate to obtain probabilities). This means you don't actually need to worry about normalizing, because `Categorical` accepts unnormalised logits.

```
prompt = "John and Mary went to the"
input_ids = tokenizer.encode(prompt, return_tensors="pt").to(device)
logits = model(input_ids)[0, -1]

expected_top_5 = {
    " church": 0.0648,
    " house": 0.0367,
    " temple": 0.0145,
    " same": 0.0104,
    " Church": 0.0097
}
topk_5_sum = sum(expected_top_5.values())

observed_freqs = defaultdict(int)

N = 10000
for _ in tqdm(range(N)):
    token = TransformerSampler.sample_next_token(input_ids.squeeze(), logits, top_k=5)
    observed_freqs[tokenizer.decode(token)] += 1

for word in expected_top_5:
    expected_freq = expected_top_5[word] / topk_5_sum
    observed_freq = observed_freqs[word] / N
    print(f"Word: {word!r:<9}. Expected freq = {expected_freq:.4f}, observed freq = {observed_freq:.4f}")
    assert abs(observed_freq - expected_freq) < 0.015, "Try increasing N if this fails by a small amount."
```

```
100%                                        10000/10000 [00:11<00:00, 893.97it/s]
Word: ' church'. Expected freq = 0.4761, observed freq = 0.4776
Word: ' house' . Expected freq = 0.2697, observed freq = 0.2742
Word: ' temple'. Expected freq = 0.1065, observed freq = 0.1019
Word: ' same'  . Expected freq = 0.0764, observed freq = 0.0759
Word: ' Church'. Expected freq = 0.0713, observed freq = 0.0704
```

▶ Solution

Top-K Sampling - Example

The GPT-2 paper famously included an example prompt about unicorns. Now it's your turn to see just how cherry picked this example was.

The paper claims they used `top_k=40` and best of 10 samples.

```
sampler = TransformerSampler(model, tokenizer)

your_prompt = "In a shocking finding, scientist discovered a herd of unicorns living in a remote, previously unexplored val
output = sampler.sample(your_prompt, temperature=0.7, top_k=40, max_tokens_generated=64)
rprint(f"Your model said:\n\n[bold dark_orange]{output}")
```

```
Your model said:

In a shocking finding, scientist discovered a herd of unicorns living in a remote, previously unexplored valley, in
the Andes Mountains. Even more surprising to the researchers was the fact that the unicorns spoke perfect English.

"We had a lot of information about the linguistic abilities of these creatures and how they could communicate, but
we didn't have the information to actually understand them," says Dr. John Smith, a graduate student in the
Department of Animal Sciences at the University of California, Berkeley.

The researchers used a technique
```

Top-p aka Nucleus Sampling

The basic idea is that we choose the most likely words, up until the total probability of words we've chosen crosses some threshold. Then we sample from those chosen words based on their logits.

The steps are:

- Sort the probabilities from largest to smallest
- Find the cutoff point where the cumulative probability first equals or exceeds `top_p`. We do the cutoff inclusively, keeping the first probability above the threshold.
- If the number of kept probabilities is less than `min_tokens_to_keep`, keep that many tokens instead.
- Set all other probabilities to zero
- Normalize and sample

Optionally, refer to the paper [The Curious Case of Neural Text Degeneration] for some comparison of different methods.

Exercise - implement `sample_top_p`

```
Difficulty: 🟠🟠🟠⚪⚪
Importance: 🟠⚪⚪⚪⚪

You should spend up to 15-20 minutes on this exercise.
```

▶ Example of top-p sampling (if you're confused)

▶ Help - I'm stuck on how to implement this function.

```
prompt = "John and Mary went to the"
input_ids = tokenizer.encode(prompt, return_tensors="pt").to(device)
logits = model(input_ids)[0, -1]

expected_top_10pct = {
    " church": 0.0648,
    " house": 0.0367, # These are the two most likely tokens, and add up to >10%
}
top_10pct_sum = sum(expected_top_10pct.values())

observed_freqs = defaultdict(int)

N = 10000
for _ in tqdm(range(N)):
    token = TransformerSampler.sample_next_token(input_ids.squeeze(), logits, top_p=0.1)
    observed_freqs[tokenizer.decode(token)] += 1

for word in expected_top_10pct:
    expected_freq = expected_top_10pct[word] / top_10pct_sum
    observed_freq = observed_freqs[word] / N
    print(f"Word: {word!r:<9}. Expected freq {expected_freq:.4f}, observed freq {observed_freq:.4f}")
    assert abs(observed_freq - expected_freq) < 0.01, "Try increasing N if this fails by a small amount."
```

```
100%                                                10000/10000 [00:13<00:00, 732.01it/s]
Word: ' church'. Expected freq 0.6384, observed freq 0.6305
Word: ' house' . Expected freq 0.3616, observed freq 0.3695
```

▶ Solution

Top-p Sampling - Example

```
sampler = TransformerSampler(model, tokenizer)

your_prompt = "Eliezer Shlomo Yudkowsky (born September 11, 1979) is an American decision and artificial intelligence (AI)
output = sampler.sample(your_prompt, temperature=0.7, top_p=0.95, max_tokens_generated=64)
rprint(f"Your model said:\n\n[bold dark_orange]{output}")
```

```
Your model said:

Eliezer Shlomo Yudkowsky (born September 11, 1979) is an American decision and artificial intelligence (AI)
theorist and writer, best known for his articles on artificial intelligence and human intelligence. He is also the
creator of the recent book "Machine Intelligence: A Study of Human Brain Development."

While speaking at the 2012 ACM meeting, Shlomo mentioned that he has been working on the same topic for the past
two years, and that he was surprised
```

Beam search

Finally, we'll implement a more advanced way of searching over output: **beam search**. You should read the HuggingFace page on beam search before moving on.

In beam search, we maintain a list of size `num_beams` completions which are the most likely completions so far as measured by the product of their probabilities. Since this product can become very small, we use the sum of log probabilities instead. Note - log probabilities are *not* the same as your model's output. We get log probabilities by first taking softmax of our output and then taking log. You can do this with the `log_softmax` function / tensor method.

▶ Log probabilities are equal to the logit output after being translated by some amount X (where X is a function of the original logit output). Can you prove this?

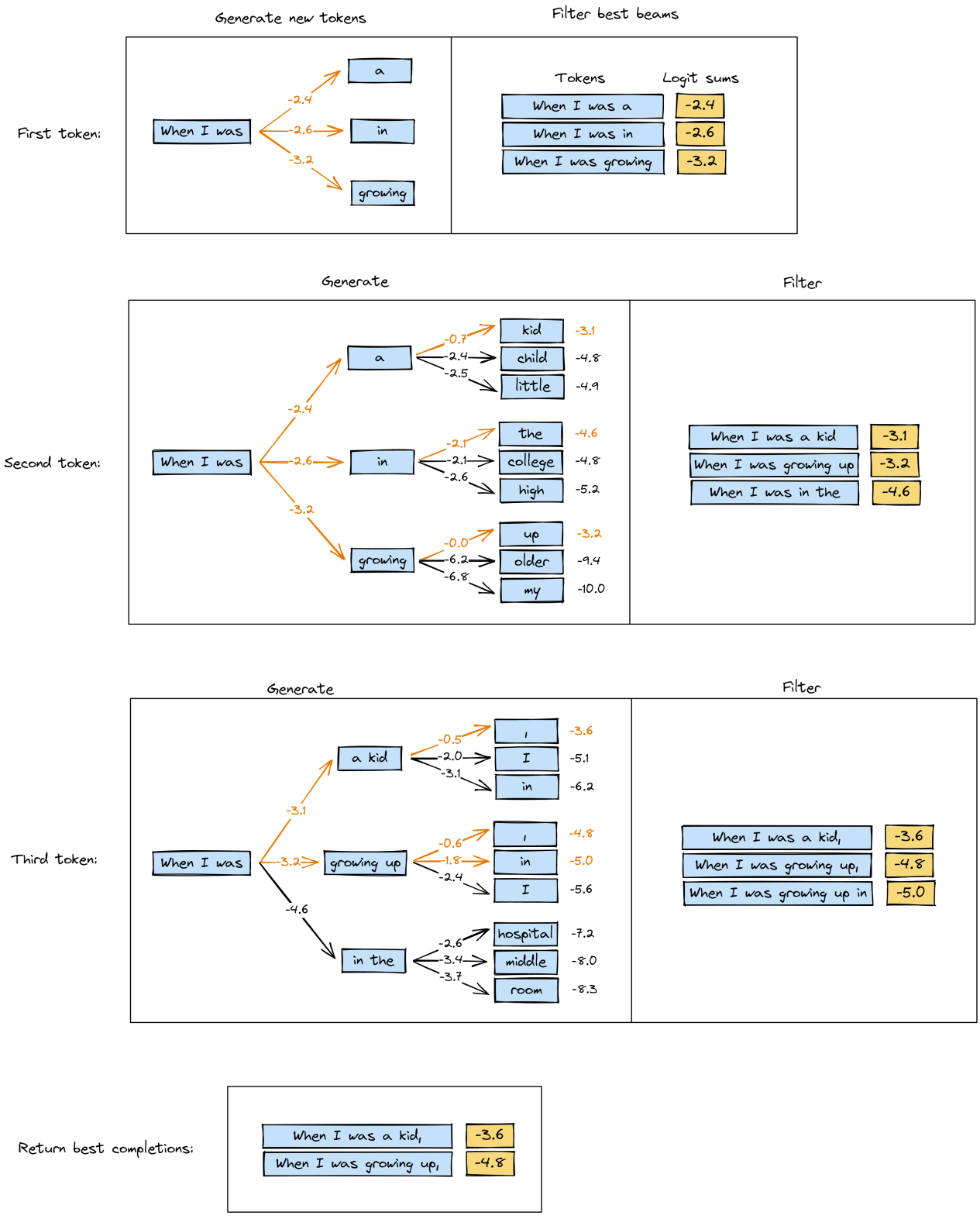
▶ Why do you think we use log softmax rather than logit output?

At each iteration, we run the batch of completions through the model and take the log-softmax to obtain `d_vocab` log-probs for each completion, or `num_beams * d_vocab` possible next completions in total.

If we kept all of these, then we would have `num_beams * d_vocab * d_vocab` completions after the next iteration which is way too many, so instead we sort them by their score and loop through from best (highest) log probability to worst (lowest).

The illustration below might help (based on real results from this method). Here, we have the following hyperparameters:

```
num_beams = 3
max_new_tokens = 3
num_return_sequences = 2
```



Note how after each "generate" stage, we have `num_beams ** 2` possible completions, which we then filter down to `num_beams`. Can you see why we need to generate this many (and what might happen if we generated fewer)?

How do we deal with sequences that terminate early (i.e. by generating an EOS token)? Answer - we append them to the list of completions which we'll return at the end, and remove them from the generation tree. Our algorithm terminates when either all our sequences have length `max_new_tokens` larger than the initial prompt length, or we've generated `num_returns_sequences` terminating sequences.

Exercise - implement `beam_search`

Difficulty: 🟠🟠🟠🟠⚪
Importance: 🟠⚪⚪⚪⚪

```
You should spend up to 30-40 minutes on this exercise.
```

You should now complete the `beam_search` method in your `TransformerSampler` class.

We've provided one possible template for you to use: the class `Beams`, with important methods `generate` and `filter` for you to fit in (which correspond to the two stages in the diagram above). There are also a few of helper functions in this class:

- `new_beams`, which creates a new `Beams` object from an old one.
- `__getitem__`, which allows you to index into a `Beams` object to get a specific batch of beams.
- `logprobs_and_completions`, which turns a `Beams` object into a list of (logprob sum, string completion) tuples (useful for getting your final output).
- `print`, which prints out the current state of the beams (useful for debugging, if you run `beam_search` with `verbose=True`).

You can then fill in the `beam_search` function, using this class and its methods.

We've provided unit tests for the `generate` and `filter` functions, so you can verify that these are correct before moving on to the full `beam_search` function.

Note that using the `Beams` class is not strictly necessary, you could fill in the `beam_search` function directly if you prefer. The `Beams` class is just meant to provide one example way you might implement this function. Often, modular code like this is easier to write and debug, and easier to extend to cover new use cases (e.g. when we use caching in the bonus exercises).

Why all the n-gram repetition?

You should observe that, while the output of beam search is sometimes more fluent than some of the other sampling methods you implement, it also has an unfortunate tendency to repeat sentences or sequences. This makes sense - if the model produces a sentence with a relatively high logit sum, then it will want to produce the same sentence again even if it doesn't make a lot of sense in context.

A common solution is to ban repetition of n-grams. We've provided the argument `no_repeat_ngram_size` in the `generate` method for this purpose. Using this argument should prevent the model from repeating any n-grams of that size. Good values of this parameter to try are 2 or 3.

However, first you should focus on getting a working version of beam search *without* using this argument.

▶ Hint (for `no_repeat_ngram_size`)

```python
@dataclass
class Beams:
    '''Class to store beams during beam search.'''
    model: DemoTransformer
    tokenizer: GPT2TokenizerFast
    logprob_sums: Float[Tensor, "batch"]
    tokens: Int[Tensor, "batch seq"]

    def new_beams(self, logprob_sums, tokens) -> "Beams":
        '''Creates a new Beams object with the same model and tokenizer.'''
        return Beams(self.model, self.tokenizer, logprob_sums, tokens)

    def __getitem__(self, idx) -> "Beams":
        '''Allows you to take a slice of the beams object along the batch dimension.'''
        return self.new_beams(self.logprob_sums[idx], self.tokens[idx])

    @property
    def logprobs_and_completions(self) -> List[Tuple[float, str]]:
        '''Returns self as a list of logprob sums and completions (useful for getting final output).'''
        return [
            (logprob_sum.item(), self.tokenizer.decode(tokens))
            for (logprob_sum, tokens) in zip(self.logprob_sums, self.tokens)
        ]


    def generate(self, toks_per_beam: int, no_repeat_ngram_size: Optional[int] = None) -> "Beams":
        '''
        Starting from the current set of beams (which has length `num_beams`), returns a new
        set of `num_beams * toks_per_beam`, containing the best `toks_per_beam` continuations for each
        of the original beams.

        Optional argument `no_repeat_ngram_size` means your model won't generate any sequences with
        a repeating n-gram of this length.
        '''
        # SOLUTION
        # Get the output logprobs for the next token (for every sequence in current beams)
        logprobs: Tensor = self.model(self.tokens)[:, -1, :].log_softmax(-1)

        # Get the top `toks_per_beam` tokens for each sequence
        # topk_logprobs, topk_tokenIDs = logprobs.topk(k=toks_per_beam)
        topk_logprobs, topk_tokenIDs = self.get_topk_non_repeating(logprobs, no_repeat_ngram_size, k=toks_per_beam)

        # Get all of the new possible beams, via einops operations
```

```
        #   Here, we're effectively flattening out the batch dimension and k dimension, to give us tensors
        #   with every possible combination of (original sequence, new token) pairs.)
        new_logprob_sums = sum([
            einops.repeat(self.logprob_sums, "batch -> (batch k)", k=toks_per_beam),
            einops.rearrange(topk_logprobs, "batch k -> (batch k)")
        ])
        new_tokens = t.concat([
            einops.repeat(self.tokens, "batch seq -> (batch k) seq", k=toks_per_beam),
            einops.rearrange(topk_tokenIDs, "batch k -> (batch k) 1")
        ], dim=-1)
        return self.new_beams(new_logprob_sums, new_tokens)


    def filter(self, num_beams: int) -> Tuple["Beams", "Beams"]:
        '''
        Returns:
            best_beams: Beams
                filtered version of self, containing all best `num_beams` which are also not terminated.

            early_terminations: Beams
                filtered version of self, containing all best `num_beams` which are also terminated.
                i.e. the sum of lengths of these two should equal `num_beams`.
        '''
        # SOLUTION
        # Get the indices of top `num_beams` beams
        top_beam_indices = self.logprob_sums.topk(k=num_beams, dim=0).indices.tolist()
        # Get the indices of terminated sequences
        new_tokens = self.tokens[:, -1]
        terminated_indices = t.nonzero(new_tokens == self.tokenizer.eos_token_id)

        # Get the indices of the `num_beams` best sequences (some terminated, some not terminated)
        best_continuing = [i for i in top_beam_indices if i not in terminated_indices]
        best_terminated = [i for i in top_beam_indices if i in terminated_indices]

        # Return the beam objects from these indices
        return self[best_continuing], self[best_terminated]


    def print(self, title="Best completions", max_print_chars=80) -> None:
        '''
        Prints out a set of sequences with their corresponding logitsums.
        '''
        if len(self.tokens) == 0:
            return
        table = Table("logitsum", "completion", title=title)
        for logprob_sum, tokens in zip(self.logprob_sums, self.tokens):
            text = self.tokenizer.decode(tokens)
            if len(repr(text)) > max_print_chars:
                text = text[:int(0.3 * max_print_chars)] + " ... " + text[-int(0.7 * max_print_chars):]
            table.add_row(f"{logprob_sum:>8.3f}", repr(text))
        rprint(table)


    def get_topk_non_repeating(
        self,
        logprobs: Float[Tensor, "batch d_vocab"],
        no_repeat_ngram_size: Optional[int],
        k: int,
    ) -> Tuple[Float[Tensor, "k"], Int[Tensor, "k"]]:
        '''
        logprobs:
            tensor of the log-probs for the next token
        no_repeat_ngram_size:
            size of ngram to avoid repeating
        k:
            number of top logits to return, for each beam in our collection

        Returns:
            equivalent to the output of `logprobs.topk(dim=-1)`, but makes sure
            that no returned tokens would produce an ngram of size  `no_repeat_ngram_size`
            which has already appeared in `self.tokens`.
        '''
        batch, seq_len = self.tokens.shape
        neg_inf = t.tensor(-1.0e4).to(device)

        # If completion isn't long enough for a repetition, or we have no restructions, just return topk
        if (no_repeat_ngram_size is not None) and (seq_len > no_repeat_ngram_size-1):
            # Otherwise, we need to check for ngram repetitions
            # First, get the most recent `no_repeat_ngram_size-1` tokens
            last_ngram_prefix = self.tokens[:, seq_len - (no_repeat_ngram_size-1):]
            # Next, find all the tokens we're not allowed to generate (by going iterating through past ngrams and seeing if
            for i in range(seq_len - (no_repeat_ngram_size-1)):
```

```
                ngrams = self.tokens[:, i:i+no_repeat_ngram_size] # (batch, ngram)
                ngrams_are_repeated = (ngrams[:, :-1] == last_ngram_prefix).all(-1) # (batch,)
                ngram_end_tokens = ngrams[:, [-1]] # (batch, 1)
                # Fill logprobs with neginf wherever the ngrams are repeated
                logprobs[range(batch), ngram_end_tokens] = t.where(
                    ngrams_are_repeated,
                    neg_inf,
                    logprobs[range(batch), ngram_end_tokens],
                )

        # Finally, get our actual tokens
        return logprobs.topk(k=k, dim=-1)


@t.inference_mode()
def beam_search(
    self: TransformerSampler,
    prompt: str,
    num_return_sequences: int,
    num_beams: int,
    max_new_tokens: int,
    no_repeat_ngram_size: Optional[int] = None,
    verbose=False
) -> List[Tuple[float, Tensor]]:
    '''
    Implements a beam search, by repeatedly performing the `generate` and `filter` steps (starting
    from the initial prompt) until either of the two stopping criteria are met:

        (1) we've generated `max_new_tokens` tokens, or
        (2) we've generated `num_returns_sequences` terminating sequences.

    To modularize this function, most of the actual complexity is in the Beams class,
    in the `generate` and `filter` methods.
    '''

    assert num_return_sequences <= num_beams
    self.model.eval()

    # SOLUTION
    tokens = self.tokenizer.encode(prompt, return_tensors="pt").to(device)

    # List for final beams to return (and early terminations)
    final_logprobs_and_completions: List[Tuple[float, str]] = []
    # Keep track of all best beams after each step
    best_beams = Beams(self.model, self.tokenizer, t.tensor([0.0]).to(device), tokens)

    for n in tqdm(range(max_new_tokens)):

        # Generation step
        best_beams = best_beams.generate(toks_per_beam=num_beams, no_repeat_ngram_size=no_repeat_ngram_size)

        # Filtering step
        best_beams, best_beams_terminated = best_beams.filter(num_beams=num_beams)
        final_logprobs_and_completions.extend(best_beams_terminated.logprobs_and_completions)

        # Print output
        if verbose:
            best_beams.print()

        # Check stopping condition
        if len(final_logprobs_and_completions) >= num_return_sequences:
            return final_logprobs_and_completions[:num_return_sequences]

    final_logprobs_and_completions.extend(best_beams.logprobs_and_completions)
    final_logprobs_and_completions = final_logprobs_and_completions[:num_return_sequences]
    return final_logprobs_and_completions
```

Example usage of the `Beams` class, and the `print` method (not the logitsums aren't necessarily accurate, this example is just an illustration):

```
beams = Beams(
    model,
    tokenizer,
    logprob_sums = t.tensor([-10.0, -15.0, -20.0]).to(device),
    tokens = t.tensor([
        [5661, 318, 262, 2368],
        [5661, 318, 262, 1218],
        [5661, 318, 262, 717],
    ]).to(device)
)

beams.print()
```

Best completions

logitsum	completion
-10.000	'this is the third'
-15.000	'this is the second'
-20.000	'this is the first'

And here are some unit tests for your `generate` and `filter` methods:

```
print("Testing generate, without no_repeat_ngram_size argument:")
new_beams = beams.generate(toks_per_beam=2)
new_beams.print()
assert new_beams.logprobs_and_completions[0][1] == "this is the third time"
```

```
Testing generate, without no_repeat_ngram_size argument:
```

Best completions

logitsum	completion
-11.829	'this is the third time'
-13.488	'this is the third year'
-16.610	'this is the second time'
-18.534	'this is the second of'
-21.050	'this is the first time'
-22.922	'this is the first of'

```
print("Testing generate, with no_repeat_ngram_size argument:")

bigram_beams = Beams(
    model,
    tokenizer,
    logprob_sums = t.tensor([-0.0]).to(device),
    tokens = t.tensor([[530, 734, 530, 734]]).to(device)
    # tokens are " one two one two"
)

# With no_repeat_ngram_size=1, should not generate the token " one" or " two"
new_bigram_beams = bigram_beams.generate(toks_per_beam=3, no_repeat_ngram_size=1)
new_bigram_beams.print()
assert all([not (completion[1].endswith(" one") or completion[1].endswith(" two")) for completion in new_bigram_beams.logpr

# With no_repeat_ngram_size=2, it can generate " two" (which it should), but not " one"
new_bigram_beams = bigram_beams.generate(toks_per_beam=3, no_repeat_ngram_size=2)
new_bigram_beams.print()
assert all([not completion[1].endswith(" one") for completion in new_bigram_beams.logprobs_and_completions])
assert any([not completion[1].endswith(" two") for completion in new_bigram_beams.logprobs_and_completions])

print("All tests for `generate` passed!")
```

```
Testing generate, with no_repeat_ngram_size argument:
```

Best completions

logitsum	completion
-1.660	' one two one two three'
-2.761	' one two one two.'
-2.883	' one two one two,'

Best completions

logitsum	completion
-1.660	' one two one two three'
-1.824	' one two one two two'
-2.761	' one two one two.'

```
All tests for `generate` passed!
```

```
logprob_sums = t.tensor([-1.0, -2.0]).to(device)
tokens = t.tensor([
    [19485, 13],
    [19485, tokenizer.eos_token_id]
]).to(device)

beams_with_eos = Beams(model, tokenizer, logprob_sums, tokens)
best_beams, early_terminations = beams_with_eos.filter(2)

t.testing.assert_close(best_beams.logprob_sums, logprob_sums[[0]])
t.testing.assert_close(best_beams.tokens, tokens[[0]])

assert early_terminations.logprobs_and_completions == [(-2.0, "Stop" + tokenizer.eos_token)]

print("All tests for `filter` passed!")
```

```
All tests for `filter` passed!
```

▶ Solutions (for `generate` and `filter`)

Once you've passed both these unit tests, you can try implementing the full beam search function. It should create a `Beams` object from the initial prompt, and then repeatedly call `generate` and `filter` until the stopping criteria are met.

```
TransformerSampler.beam_search = beam_search

sampler = TransformerSampler(model, tokenizer)

prompt = "The ships hung in the sky in much the same way that"
orig_len = len(tokenizer.encode(prompt))

final_logitsums_and_completions = sampler.beam_search(
    prompt=prompt,
    num_return_sequences=3,
    num_beams=40,
    max_new_tokens=60,
    no_repeat_ngram_size=2,
    verbose=False
)

# Print all the best output
for logprob_sum, text in final_logitsums_and_completions:
    avg_logprob_as_prob = t.tensor(logprob_sum / (len(tokenizer.encode(text)) - orig_len)).exp().item()
    print("=" * 25 + f" Avg logprob (as probability) = {avg_logprob_as_prob:.3f} " + "=" * 25)
    rprint("Best output:\n\n[bold dark_orange]" + text)
```

```
100%                                                60/60 [00:04<00:00, 8.92it/s]
========================= Avg logprob (as probability) = 0.255 =========================
Best output:

The ships hung in the sky in much the same way that they did at the beginning of the Second World War.

For the first time in history, the U.S. Navy was able to carry out a full-scale amphibious assault on a large
number of targets in a short period of time. In doing so, it allowed the Navy to
========================= Avg logprob (as probability) = 0.254 =========================
Best output:

The ships hung in the sky in much the same way that they did at the beginning of the Second World War.

For the first time in history, the U.S. Navy was able to carry out a full-scale amphibious assault on a large
number of targets in a short period of time. It was a major victory for the United States
========================= Avg logprob (as probability) = 0.254 =========================
Best output:

The ships hung in the sky in much the same way that they did at the beginning of the Second World War.

For the first time in history, the U.S. Navy was able to carry out a full-scale amphibious assault on a large
number of targets in a short period of time. In fact, it was only a matter of
```

▶ Solution (full)

Caching

This section is also designed to be challenging, and take quite some time. There are many different ways to solve it, and you're expected to try and find your own way (you should think about this for a while before looking at the suggestions in the dropdowns). Additionally, you might not find it as interesting as some of the other sections. In this case, and if you have a lot of extra time, you might want to start on the "building BERT" exercises from this chapter.

How can caching help us?

The text generation we've done so far is needlessly re-computing certain values, which is very noticeable when you try to generate longer sequences.

Suppose you're generating text, and you've already run GPT on the sentence "My life motto:". Now you want to run the model on the sentence "My life motto: Always". Which computations from the first sentence can you reuse?