[1.2] TransformerLens & induction circuits

Introduction

These pages are designed to get you introduc :o Neel's **TransformerLens** library.

Most of the sections are constructed in the following way:

1. A particular feature of TransformerLens is introduced.
2. You are given an exercise, in which you have to apply the feature.

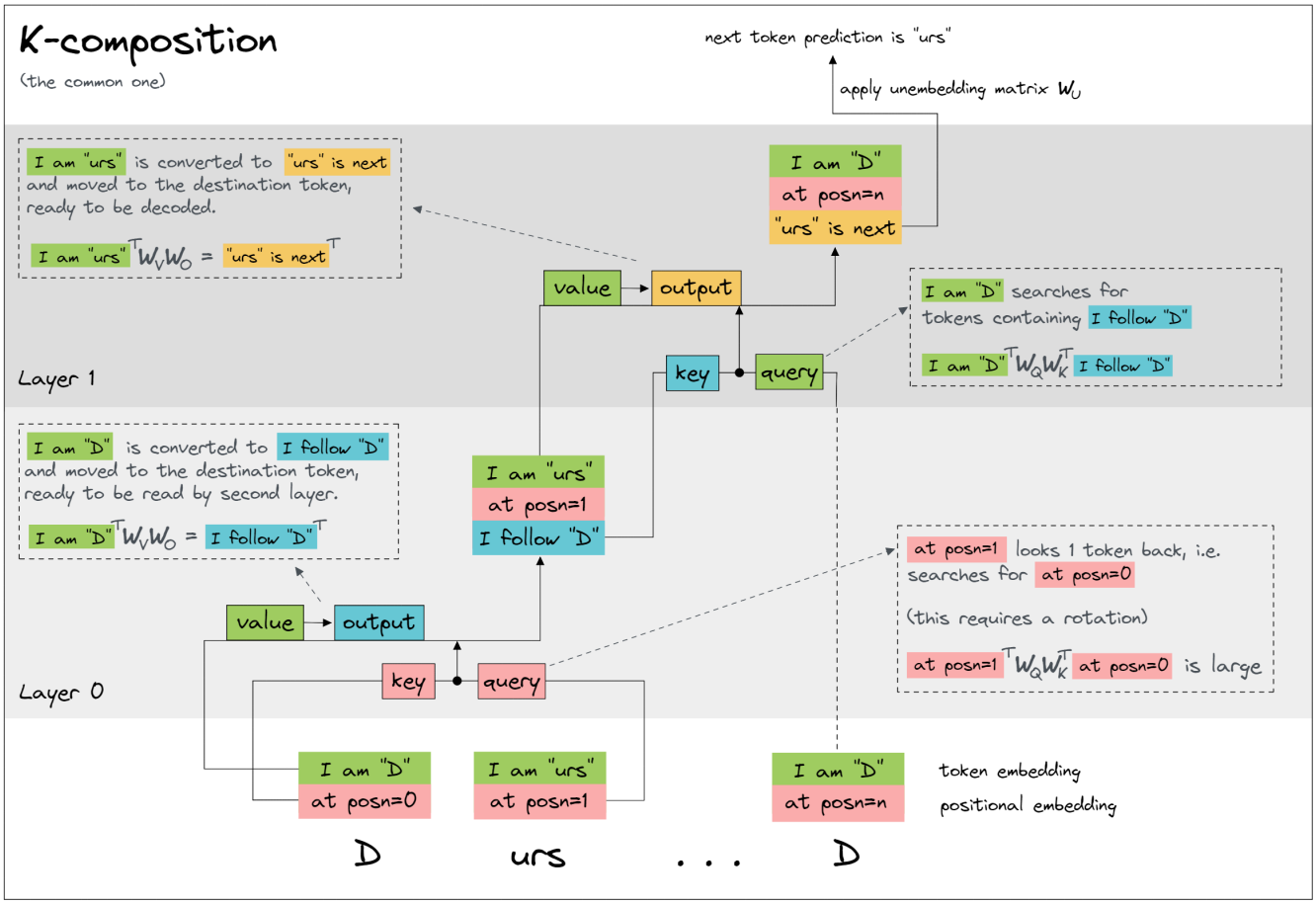
The running theme of the exercises is **induction circuits**. Induction circuits are a particular type of circuit in a transformer, which can perform basic in-context learning. You should read the corresponding section of Neel's glossary, before continuing. This LessWrong post might also help; it contains some diagrams (like the one below) which walk through the induction mechanism step by step.

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!

How a 2-layer transformer learns the world "Dursley" (tokenized as ["D", "urs", "ley"]) in-context.

Key for different subspaces in the residual stream, and how the model interprets them:

token encoding subspace (i.e. "this token is X")	= rows of W_E
positional encoding subspace (i.e. "this token is at position X")	= rows of W_{pos}
decoding subspace (i.e. "the next token will be X")	= cols of W_U
prev token subspace (i.e. "the previous token was X")	= "intermediate information"



Content & Learning Objectives

1 TransformerLens: Introduction

This section is designed to get you up to speed with the TransformerLens library. You'll learn how to load and run models, and learn about the shared architecture template for all of these models (the latter of which should be familiar to you if you've already done the exercises that come before these, since many of the same design principles are followed).

> Learning objectives
>
> - Load and run a `HookedTransformer` model
> - Understand the basic architecture of these models
> - Use the model's tokenizer to convert text to tokens, and vice versa
> - Know how to cache activations, and to access activations from the cache

- Use `circuitsvis` to visualise attention heads

2️⃣ Finding induction heads

Here, you'll learn about induction heads, how they work and why they are important. You'll also learn how to identify them from the characteristic induction head stripe in their attention patterns when the model input is a repeating sequence.

> Learning objectives
>
> - Understand what induction heads are, and the algorithm they are implementing
> - Inspect activation patterns to identify basic attention head patterns, and write your own functions to detect attention heads for you
> - Identify induction heads by looking at the attention patterns produced from a repeating random sequence

3️⃣ TransformerLens: Hooks

Next, you'll learn about hooks, which are a great feature of TransformerLens allowing you to access and intervene on activations within the model. We will mainly focus on the basics of hooks and using them to access activations (we'll mainly save the causal interventions for the later IOI exercises). You will also build some tools to perform logit attribution within your model, so you can identify which components are responsible for your model's performance on certain tasks.

> Learning objectives
>
> - Understand what hooks are, and how they are used in TransformerLens
> - Use hooks to access activations, process the results, and write them to an external tensor
> - Build tools to perform attribution, i.e. detecting which components of your model are responsible for performance on a given task
> - Understand how hooks can be used to perform basic interventions like **ablation**

4️⃣ Reverse-engineering induction circuits

Lastly, these exercises show you how you can reverse-engineer a circuit by looking directly at a transformer's weights (which can be considered a "gold standard" of interpretability; something not possible in every situation). You'll examine QK and OV circuits by multiplying through matrices (and learn how the FactoredMatrix class makes matrices like these much easier to analyse). You'll also look for evidence of composition between two induction heads, and once you've found it then you'll investigate the functionality of the full circuit formed from this composition.

> Learning objectives
>
> - Understand the difference between investigating a circuit by looking at activtion patterns, and reverse-engineering a circuit by looking directly at the weights
> - Use the factored matrix class to inspect the QK and OV circuits within an induction circuit
> - Perform further exploration of induction circuits: composition scores, and targeted ablations

Setup (don't read, just run!)

```
try:
    import google.colab # type: ignore
    IN_COLAB = True
except:
    IN_COLAB = False

import os, sys

if IN_COLAB:
    # Install packages
    %pip install einops
    %pip install jaxtyping
    %pip install transformer_lens
    %pip install pytorch_lightning
    %pip install git+https://github.com/callummcdougall/CircuitsVis.git#subdirec

    # Code to download the necessary files (e.g. solutions, test funcs)
    import os, sys
    if not os.path.exists("chapter1_transformers"):
        # !wget https://github.com/callummcdougall/ARENA_2.0/archive/refs/heads/
        # !unzip /content/main.zip 'ARENA_2.0-main/chapter1_transformers/exercis
        sys.path.append("/content/chapter1_transformers/exercises")
        os.remove("/content/main.zip")
        # os.rename("ARENA_2.0-main/chapter1_transformers", "chapter1_transforme
        # os.rmdir("ARENA_2.0-main")
        os.chdir("chapter1_transformers/exercises")
else:
    from IPython import get_ipython
    ipython = get_ipython()
    ipython.run_line_magic("load_ext", "autoreload")
    ipython.run_line_magic("autoreload", "2")

    CHAPTER = r"chapter1_transformers"
    CHAPTER_DIR = r"./" if CHAPTER in os.listdir() else os.getcwd().split(CHAPTE
    EXERCISES_DIR = CHAPTER_DIR + f"{CHAPTER}/exercises"
    sys.path.append(EXERCISES_DIR)

# Things that need to be done manually
```

```
39
40 # Comment out the line `assert attn_scores.shape` in `def mask_scores`
41 # Comment out the top_1_acc lines, because CUDA errors
42
43 # END
```

```
Looking in indexes: https://pypi.org/simple, https://us-python.pkg.dev/colab-w
Requirement already satisfied: einops in /usr/local/lib/python3.10/dist-packag
Looking in indexes: https://pypi.org/simple, https://us-python.pkg.dev/colab-w
Requirement already satisfied: jaxtyping in /usr/local/lib/python3.10/dist-pac
Requirement already satisfied: numpy>=1.20.0 in /usr/local/lib/python3.10/dist
Requirement already satisfied: typeguard>=2.13.3 in /usr/local/lib/python3.10/
Requirement already satisfied: typing-extensions>=3.7.4.1 in /usr/local/lib/py
Looking in indexes: https://pypi.org/simple, https://us-python.pkg.dev/colab-w
Requirement already satisfied: transformer_lens in /usr/local/lib/python3.10/d
Requirement already satisfied: datasets>=2.7.1 in /usr/local/lib/python3.10/di
Requirement already satisfied: einops>=0.6.0 in /usr/local/lib/python3.10/dist
Requirement already satisfied: fancy-einsum>=0.0.3 in /usr/local/lib/python3.1
Requirement already satisfied: jaxtyping>=0.2.11 in /usr/local/lib/python3.10/
Requirement already satisfied: numpy>=1.23 in /usr/local/lib/python3.10/dist-p
Requirement already satisfied: pandas>=1.1.5 in /usr/local/lib/python3.10/dist
Requirement already satisfied: rich>=12.6.0 in /usr/local/lib/python3.10/dist-
Requirement already satisfied: torch>=1.10 in /usr/local/lib/python3.10/dist-p
Requirement already satisfied: tqdm>=4.64.1 in /usr/local/lib/python3.10/dist-
Requirement already satisfied: transformers>=4.25.1 in /usr/local/lib/python3.
Requirement already satisfied: wandb>=0.13.5 in /usr/local/lib/python3.10/dist
Requirement already satisfied: pyarrow>=8.0.0 in /usr/local/lib/python3.10/dis
Requirement already satisfied: dill<0.3.7,>=0.3.0 in /usr/local/lib/python3.10
Requirement already satisfied: requests>=2.19.0 in /usr/local/lib/python3.10/d
Requirement already satisfied: xxhash in /usr/local/lib/python3.10/dist-packag
Requirement already satisfied: multiprocess in /usr/local/lib/python3.10/dist-
Requirement already satisfied: fsspec[http]>=2021.11.1 in /usr/local/lib/pytho
Requirement already satisfied: aiohttp in /usr/local/lib/python3.10/dist-packa
Requirement already satisfied: huggingface-hub<1.0.0,>=0.11.0 in /usr/local/li
Requirement already satisfied: packaging in /usr/local/lib/python3.10/dist-pac
Requirement already satisfied: pyyaml>=5.1 in /usr/local/lib/python3.10/dist-p
Requirement already satisfied: typeguard>=2.13.3 in /usr/local/lib/python3.10/
Requirement already satisfied: typing-extensions>=3.7.4.1 in /usr/local/lib/py
Requirement already satisfied: python-dateutil>=2.8.1 in /usr/local/lib/python
Requirement already satisfied: pytz>=2020.1 in /usr/local/lib/python3.10/dist-
Requirement already satisfied: markdown-it-py<3.0.0,>=2.2.0 in /usr/local/lib/
Requirement already satisfied: pygments<3.0.0,>=2.13.0 in /usr/local/lib/pytho
Requirement already satisfied: filelock in /usr/local/lib/python3.10/dist-pack
Requirement already satisfied: sympy in /usr/local/lib/python3.10/dist-package
Requirement already satisfied: networkx in /usr/local/lib/python3.10/dist-pack
Requirement already satisfied: jinja2 in /usr/local/lib/python3.10/dist-packag
Requirement already satisfied: triton==2.0.0 in /usr/local/lib/python3.10/dist
Requirement already satisfied: cmake in /usr/local/lib/python3.10/dist-package
Requirement already satisfied: lit in /usr/local/lib/python3.10/dist-packages
Requirement already satisfied: regex!=2019.12.17 in /usr/local/lib/python3.10/
Requirement already satisfied: tokenizers!=0.11.3,<0.14,>=0.11.1 in /usr/local
Requirement already satisfied: safetensors>=0.3.1 in /usr/local/lib/python3.10
Requirement already satisfied: Click!=8.0.0,>=7.0 in /usr/local/lib/python3.10
Requirement already satisfied: GitPython!=3.1.29,>=1.0.0 in /usr/local/lib/pyt
Requirement already satisfied: psutil>=5.0.0 in /usr/local/lib/python3.10/dist
Requirement already satisfied: sentry-sdk>=1.0.0 in /usr/local/lib/python3.10/
Requirement already satisfied: docker-pycreds>=0.4.0 in /usr/local/lib/python3
Requirement already satisfied: pathtools in /usr/local/lib/python3.10/dist-pac
Requirement already satisfied: setproctitle in /usr/local/lib/python3.10/dist-
```

```
Requirement already satisfied: setuptools in /usr/local/lib/python3.10/dist-pa
Requirement already satisfied: appdirs>=1.4.3 in /usr/local/lib/python3.10/dis
Requirement already satisfied: protobuf!=4.21.0,<5,>=3.19.0 in /usr/local/lib/
Requirement already satisfied: six>=1.4.0 in /usr/local/lib/python3.10/dist-pa
```

```
# !pip install plotly
# !pip install transformer_lens
# !pip install git+https://github.com/callummcdougall/CircuitsVis.git#subdirecto
# !pip install jaxtyping
# !pip install einops
# %pip install protobuf==3.20.*
# %pip install pytorch_lightning
```

```
import os
import sys
import plotly.express as px
import torch as t
from torch import Tensor
import torch.nn as nn
import torch.nn.functional as F
from pathlib import Path
import numpy as np
import einops
from jaxtyping import Int, Float
from typing import List, Optional, Tuple
import functools
from tqdm import tqdm
from IPython.display import display
import webbrowser
import gdown
from transformer_lens.hook_points import HookPoint
from transformer_lens import utils, HookedTransformer, HookedTransformerConfig,
import circuitsvis as cv

# Make sure exercises are in the path
chapter = r"chapter1_transformers"
exercises_dir = Path(f"{os.getcwd().split(chapter)[0]}/{chapter}/exercises").res
section_dir = (exercises_dir / "part2_intro_to_mech_interp").resolve()
if str(exercises_dir) not in sys.path: sys.path.append(str(exercises_dir))

from plotly_utils import imshow, hist, plot_comp_scores, plot_logit_attribution,
from part1_transformer_from_scratch.solutions import get_log_probs
import part2_intro_to_mech_interp.tests as tests

# Saves computation time, since we don't need it for the contents of this notebo
t.set_grad_enabled(False)

device = t.device("cuda" if t.cuda.is_available() else "cpu")

MAIN = __name__ == "__main__"
```

1️⃣ TransformerLens: Introduction

> Learning objectives
>
> - Load and run a `HookedTransformer` model
> - Understand the basic architecture of these models
> - Use the model's tokenizer to convert text to tokens, and vice versa
> - Know how to cache activations, and to access activations from the cache
> - Use `circuitsvis` to visualise attention heads

Introduction

Note - most of this is written from the POV of Neel Nanda.

This is a demo notebook for TransformerLens, **a library I (Neel Nanda) wrote for doing mechanistic interpretability of GPT-2 Style language models.** The goal of mechanistic interpretability is to take a trained model and reverse engineer the algorithms the model learned during training from its weights. It is a fact about the world today that we have computer programs that can essentially speak English at a human level (GPT-3, PaLM, etc), yet we have no idea how they work nor how to write one ourselves. This offends me greatly, and I would like to solve this! Mechanistic interpretability is a very young and small field, and there are a *lot* of open problems - if you would like to help, please try working on one! **Check out my list of concrete open problems to figure out where to start.**

I wrote this library because after I left the Anthropic interpretability team and started doing independent research, I got extremely frustrated by the state of open source tooling. There's a lot of excellent infrastructure like HuggingFace and DeepSpeed to *use* or *train* models, but very little to dig into their internals and reverse engineer how they work. **This library tries to solve that**, and to make it easy to get into the field even if you don't work at an industry org with real infrastructure! The core features were heavily inspired by Anthropic's excellent Garcon tool. Credit to Nelson Elhage and Chris Olah for building Garcon and showing me the value of good infrastructure for accelerating exploratory research!

The core design principle I've followed is to enable exploratory analysis - one of the most fun parts of mechanistic interpretability compared to normal ML is the extremely short feedback loops! The point of this library is to keep the gap between having an experiment idea and seeing the results as small as possible, to make it easy for **research to feel like play** and to enter a flow state. This notebook demonstrates how the library works and how to use it, but if you want to see how well it works for exploratory research, check out my notebook analysing Indirect Objection Identification or my recording of myself doing research!

Loading and Running Models

TransformerLens comes loaded with >40 open source GPT-style models. You can load any of them in with `HookedTransformer.from_pretrained(MODEL_NAME)`. For this demo notebook we'll look at GPT-2 Small, an 80M parameter model, see the Available Models section for info on the rest.

```
1 gpt2_small: HookedTransformer = HookedTransformer.from_pretrained("gpt2-small")
```

```
Downloading (…)lve/main/config.json: 100%                    665/665 [00:00
Downloading model.safetensors: 100%                          548M/548M [00:08<
Downloading (…)neration_config.json: 100%                    124/124 [00:00
Downloading (…)olve/main/vocab.json: 100%                    1.04M/1.04M
Downloading (…)olve/main/merges.txt: 100%                    456k/456k [00
Downloading (…)/main/tokenizer.json: 100%                    1.36M/1.36M [0
Using pad_token, but it is not set yet.
Loaded pretrained model gpt2-small into HookedTransformer
```

HookedTransformerConfig

Alternatively, you can define a config object, then call `HookedTransformer.from_config(cfg)` to define your model. This is particularly useful when you want to have finer control over the architecture of your model. We'll see an example of this in the next section, when we define an attention-only model to study induction heads.

Even if you don't define your model in this way, you can still access the config object through the `cfg` attribute of the model.

Exercise - inspect your model

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

Use `gpt2_small.cfg` to find the following, for your GPT-2 Small model:

- Number of layers
- Number of heads per layer

- Maximum context window

You might have to check out the documentation page for some of these. If you're in VSCode then you can reach it by right-clicking on `HookedTransformerConfig` in the sidebar, and choosing "Go to definition". If you're in Colab, then you can read the GitHub page.

▶ Solution

Running your model

Models can be run on a single string or a tensor of tokens (shape: `[batch, position]`, all integers). The possible return types are:

- `"logits"` (shape `[batch, position, d_vocab]`, floats),
- `"loss"` (the cross-entropy loss when predicting the next token),
- `"both"` (a tuple of `(logits, loss)`)
- `None` (run the model, but don't calculate the logits - this is faster when we only want to use intermediate activations)

```
model_description_text = '''## Loading Models

HookedTransformer comes loaded with >40 open source GPT-style models. You can l

For this demo notebook we'll look at GPT-2 Small, an 80M parameter model. To tr

loss = gpt2_small(model_description_text, return_type="loss")
print("Model loss:", loss)
```

```
Model loss: tensor(4.3443, device='cuda:0')
```

Transformer architecture

HookedTransformer is a somewhat adapted GPT-2 architecture, but is computationally identical. The most significant changes are to the internal structure of the attention heads:

- The weights `W_K`, `W_Q`, `W_V` mapping the residual stream to queries, keys and values are 3 separate matrices, rather than big concatenated one.
- The weight matrices `W_K`, `W_Q`, `W_V`, `W_O` and activations have separate `head_index` and `d_head` axes, rather than flattening them into one big axis.
  - The activations all have shape `[batch, position, head_index, d_head]`.
  - `W_K`, `W_Q`, `W_V` have shape `[head_index, d_model, d_head]` and `W_O` has shape `[head_index, d_head, d_model]`
- **Important - we generally follow the convention that weight matrices multiply on the right rather than the left.** In other words, they have shape `[input, output]`, and we have `new_activation = old_activation @ weights + bias`.

- Click the dropdown below for examples of this, if it seems unintuitive.

▼ Examples of matrix multiplication in our model

- **Query matrices**
  - Each query matrix `W_Q` for a particular layer and head has shape `[d_model, d_head]`.
  - So if a vector `x` in the residual stream has length `d_model`, then the corresponding query vector is `x @ W_Q`, which has length `d_head`.
- **Embedding matrix**
  - The embedding matrix `W_E` has shape `[d_vocab, d_model]`.
  - So if `A` is a one-hot-encoded vector of length `d_vocab` corresponding to a particular token, then the embedding vector for this token is `A @ W_E`, which has length `d_model`.

The actual code is a bit of a mess, as there's a variety of Boolean flags to make it consistent with the various different model families in TransformerLens - to understand it and the internal structure, I instead recommend reading the code in [CleanTransformerDemo].

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.
  - They can be accessed direction fromm the model, e.g. `model.W_E` for the embedding matrix.
- **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).

The link below shows a diagram of a single layer (called a `TransformerBlock`) for an attention-only model with no biases. Each box corresponds to an **activation** (and also tells you the name of the corresponding hook point, which we will eventually use to access those activations). The red text below each box tells you the shape of the activation (ignoring the batch dimension). Each arrow corresponds to an operation on an activation; where there are **parameters** involved these are labelled on the arrows.

Link to diagram

The next link is to a diagram of a `TransformerBlock` with full features (including biases, layernorms, and MLPs). Don't worry if not all of this makes sense at first - we'll return to some of the details later. As we work with these transformers, we'll get more comfortable with their architecture.

Link to diagram

A few shortctus to make your lives easier when using these models:

- You can index weights like `W_Q` directly from the model via e.g. `model.blocks[0].attn.W_Q` (which gives you the `[nheads, d_model, d_head]` query weights for all heads in layer 0).
    - But an easier way is just to index with `model.W_Q`, which gives you the `[nlayers, nheads, d_model, d_head]` tensor containing **every** query weight in the model.
- Similarly, there exist shortcuts `model.W_E`, `model.W_U` and `model.W_pos` for the embeddings, unembeddings and positional embeddings respectively.
- With models containing MLP layers, you also have `model.W_in` and `model.W_out` for the linear layers.
- The same is true for all biases (e.g. `model.b_Q` for all query biases).

Tokenization

The tokenizer is stored inside the model, and you can access it using `model.tokenizer`. There are also a few helper methods that call the tokenizer under the hood, for instance:

- `model.to_str_tokens(text)` converts a string into a tensor of tokens-as-strings.
- `model.to_tokens(text)` converts a string into a tensor of tokens.
- `model.to_string(tokens)` converts a tensor of tokens into a string.

Examples of use:

```
print(gpt2_small.to_str_tokens("gpt2"))
print(gpt2_small.to_tokens("gpt2"))
print(gpt2_small.to_string([50256, 70, 457, 17]))
```

```
['<|endoftext|>', 'g', 'pt', '2']
tensor([[50256,    70,   457,    17]], device='cuda:0')
<|endoftext|>gpt2
```

▶ Aside - `<|endoftext|>`

Exercise - how many words does your model guess correctly?

Difficulty: 🟠🟠⚪⚪⚪
Importance: 🟠🟠🟠⚪⚪

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

Consider the `model_description_text` you fed into your model above. How many words did your model guess correctly? Which words were correct?

```
logits: Tensor = gpt2_small(model_description_text, return_type="logits")
prediction = logits.argmax(dim=-1).squeeze()[:-1]

# YOUR CODE HERE - get the model's prediction on the text
true_tokens = gpt2_small.to_tokens(model_description_text).squeeze()[1:]
is_correct = (prediction == true_tokens)

print(f"Model accuracy: {is_correct.sum()}/{len(true_tokens)}")
print(f"Correct words: {gpt2_small.to_str_tokens(prediction[is_correct])}")
```

```
Model accuracy: 32/112
Correct words: ['\n', '\n', 'former', ' with', ' models', '.', ' can', ' of',
```

▶ Hint

▶ Solution

Induction heads are a special kind of attention head which we'll examine a lot more in coming exercises. They allow a model to perform in-context learning of a specific form: generalising from one observation that token `B` follows token `A`, to predict that token `B` will follow `A` in future occurrences of `A`, even if these two tokens had never appeared together in the model's training data.

Can you see evidence of any induction heads at work, on this text?

▶ Evidence of induction heads

Caching all Activations

The first basic operation when doing mechanistic interpretability is to break open the black box of the model and look at all of the internal activations of a model. This can be done with `logits, cache = model.run_with_cache(tokens)`. Let's try this out, on the first sentence from the GPT-2 paper.

▶ Aside - a note on `remove_batch_dim`

```
gpt2_text = "Natural language processing tasks, such as question answering, mach
gpt2_tokens = gpt2_small.to_tokens(gpt2_text)
gpt2_logits, gpt2_cache = gpt2_small.run_with_cache(gpt2_tokens, remove_batch_di
```

If you inspect the `gpt2_cache` object, you should see that it contains a very large number of keys, each one corresponding to a different activation in the model. You can access the keys by indexing the cache directly, or by a more convenient indexing shorthand. For instance, the code:

```
attn_patterns_layer_0 = gpt2_cache["pattern", 0]
```

returns the same thing as:

```
attn_patterns_layer_0_copy = gpt2_cache["blocks.0.attn.hook_pattern"]

t.testing.assert_close(attn_patterns_layer_0, attn_patterns_layer_0_copy)
```

▶ Aside: `utils.get_act_name`

Exercise - verify activations

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

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

If you're already comfortable implementing things like attention calculations (e.g. having
```

Verify that `hook_q`, `hook_k` and `hook_pattern` are related to each other in the way implied by the diagram. Do this by computing `layer0_pattern_from_cache` (the attention pattern taken directly from the cache, for layer 0) and `layer0_pattern_from_q_and_k` (the attention pattern calculated from `hook_q` and `hook_k`, for layer 0). Remember that attention pattern is the probabilities, so you'll need to scale and softmax appropriately.

```
layer0_pattern_from_cache = gpt2_cache["pattern", 0]


# YOUR CODE HERE - define `layer0_pattern_from_q_and_k` manually, by manually pe
q, k = gpt2_cache["q", 0], gpt2_cache["k", 0]
seq, nhead, headsize = q.shape
layer0_attn_scores = einops.einsum(q, k, "seqQ n h, seqK n h -> n seqQ seqK")
mask = t.triu(t.ones((seq, seq), dtype=bool), diagonal=1).to(device)
layer0_attn_scores.masked_fill_(mask, -1e9)
layer0_pattern_from_q_and_k = (layer0_attn_scores / headsize**0.5).softmax(-1)


t.testing.assert_close(layer0_pattern_from_cache, layer0_pattern_from_q_and_k)
print("Tests passed!")
```

```
Tests passed!
```

▶ Hint
▶ Solution

Visualising Attention Heads

A key insight from the Mathematical Frameworks paper is that we should focus on interpreting the parts of the model that are intrinsically interpretable - the input tokens, the output logits and the attention patterns. Everything else (the residual stream, keys, queries, values, etc) are compressed intermediate states when calculating meaningful things. So a natural place to start is classifying heads by their attention patterns on various texts.

When doing interpretability, it's always good to begin by visualising your data, rather than taking summary statistics. Summary statistics can be super misleading! But now that we have visualised the attention patterns, we can create some basic summary statistics and use our visualisations to validate them! (Accordingly, being good at web dev/data visualisation is a surprisingly useful skillset! Neural networks are very high-dimensional object.)

Let's visualize the attention pattern of all the heads in layer 0, using Alan Cooney's CircuitsVis library (based on Anthropic's PySvelte library). If you did the previous set of exercises, you'll have seen this library before.

We will use the function `cv.attention.attention_patterns`, which takes two important arguments:

- `attention`: Attention head activations.
    - This should be a tensor of shape `[nhead, seq_dest, seq_src]`, i.e. the `[i, :, :]` th element is the grid of attention patterns (probabilities) for the `i` th attention head.
    - We get this by indexing our `gpt2_cache` object.
- `tokens`: List of tokens (e.g. `["A", "person"]`).
    - Sequence length must match that inferred from `attention`.
    - This is used to label the grid.
    - We get this by using the `gpt2_small.to_str_tokens` method.

(Another optional argument is `attention_head_names`, which is useful if you don't just want the heads to show up as "Head 0", "Head 1", ... in the visualisation. Also, note that we can use `cv.attention.attention_patterns` too, which has the same syntax but presents information differently, and can be more helpful in some cases.)

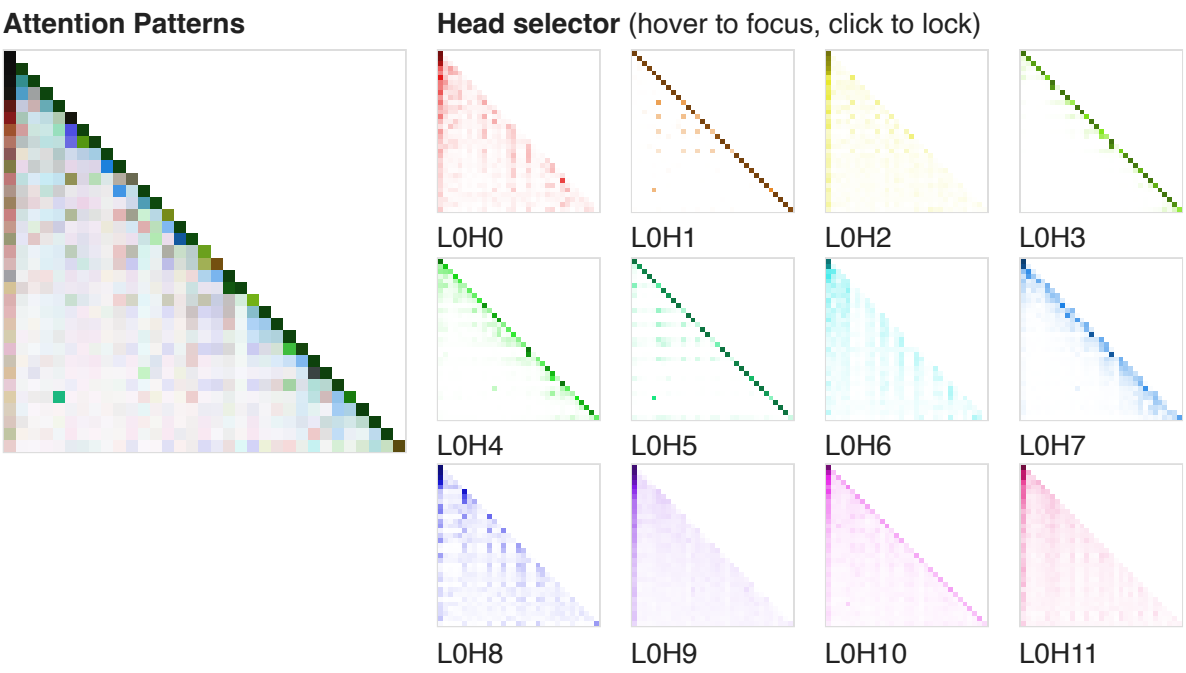
▶ Help - my `attention_heads` plots are behaving weirdly (e.g. they continually shrink after I plot them).

This visualization is interactive! Try hovering over a token or head, and click to lock. The grid on the top left and for each head is the attention pattern as a destination position by source position

grid. It's lower triangular because GPT-2 has **causal attention**, attention can only look backwards, so information can only move forwards in the network.

```
print(type(gpt2_cache))
attention_pattern = gpt2_cache["pattern", 0, "attn"]
print(attention_pattern.shape)
gpt2_str_tokens = gpt2_small.to_str_tokens(gpt2_text)

print("Layer 0 Head Attention Patterns:")
display(cv.attention.attention_patterns(
    tokens=gpt2_str_tokens,
    attention=attention_pattern,
    attention_head_names=[f"L0H{i}" for i in range(12)],
))
```

```
<class 'transformer_lens.ActivationCache.ActivationCache'>
torch.Size([12, 33, 33])
Layer 0 Head Attention Patterns:
```



Hover over heads to see the attention patterns; click on a head to lock it. Hover over each token to see which other tokens it attends to (or which other tokens attend to it - you can see this by changing the dropdown to `Destination <- Source`).

▶ Other circuitsvis functions - neuron activations

2️⃣ Finding induction heads

> Learning objectives
>
> - Understand what induction heads are, and the algorithm they are implementing
> - Inspect activation patterns to identify basic attention head patterns, and write your own functions to detect attention heads for you
> - Identify induction heads by looking at the attention patterns produced from a repeating random sequence

Introducing Our Toy Attention-Only Model

Here we introduce a toy 2L attention-only transformer trained specifically for today. Some changes to make them easier to interpret:

- It has only attention blocks.
- The positional embeddings are only added to each key and query vector in the attention layers as opposed to the token embeddings (meaning that the residual stream can't directly encode positional information).
    - This turns out to make it *way* easier for induction heads to form, it happens 2-3x times earlier - see the comparison of two training runs here. (The bump in each curve is the formation of induction heads.)
    - The argument that does this below is `positional_embedding_type="shortformer"`.
- It has no MLP layers, no LayerNorms, and no biases.
- There are separate embed and unembed matrices (i.e. the weights are not tied).

We now define our model with a `HookedTransformerConfig` object. This is similar to the `Config` object we used in the previous set of exercises, although it has a lot more features. You can look at the documentation page (Right-click, "Go to Definition" in VSCode) to seee what the different arguments do.

```
cfg = HookedTransformerConfig(
    d_model=768,
    d_head=64,
    n_heads=12,
    n_layers=2,
    n_ctx=2048,
    d_vocab=50278,
    attention_dir="causal",
    attn_only=True, # defaults to False
```

```
    tokenizer_name="EleutherAI/gpt-neox-20b",
    seed=398,
    use_attn_result=True,
    normalization_type=None, # defaults to "LN", i.e. layernorm with weights & b
    positional_embedding_type="shortformer"
)
```

▶ An aside about tokenizers

Below, you'll load in your weights, with some useful boilerplate code to download files from Google Drive and store them in a specified filepath:

```
weights_dir = (section_dir / "attn_only_2L_half.pth").resolve()

if not weights_dir.exists():
    url = "https://drive.google.com/uc?id=1vcZLJnJoYKQs-2KOjkd6LvHZrkSdoxhu"
    output = str(weights_dir)
    gdown.download(url, output)
```

```
Downloading...
From: https://drive.google.com/uc?id=1vcZLJnJoYKQs-2KOjkd6LvHZrkSdoxhu
To: /content/chapter1_transformers/exercises/part2_intro_to_mech_interp/attn_o
100%|██████████| 184M/184M [00:01<00:00, 141MB/s]
```

Finally, we'll create our model and load in the weights:

```
model = HookedTransformer(cfg)
pretrained_weights = t.load(weights_dir, map_location=device)
model.load_state_dict(pretrained_weights)
```

```
Downloading                                  156/156 [00:00<00:00,
(…)okenizer_config.json: 100%                9.71kB/s]
Downloading                                  1.08M/1.08M [00:00<00:00,
(…)olve/main/vocab.json: 100%                59.4MB/s]
Downloading                                  457k/457k [00:00<00:00,
(…)olve/main/merges.txt: 100%                10.1MB/s]
Downloading                                  2.11M/2.11M [00:00<00:00,
(…)/main/tokenizer.json: 100%                21.0MB/s]
Downloading                                  90.0/90.0 [00:00<00:00,
(…)cial_tokens_map.json: 100%                5.88kB/s]
Using pad_token, but it is not set yet.
<All keys matched successfully>
```

Use the diagram at this link to remind yourself of the relevant hook names.

Exercise - visualise attention patterns

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

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

It's important to be comfortable using circuitsvis, and the cache object.
```

This exercise should be very quick - you can reuse code from the previous section. You should look at the solution if you're still stuck after 5-10 minutes.

Visualise the attention patterns for both layers of your model, on the following prompt:

```
text = "We think that powerful, significantly superhuman machine intelligence is

logits, cache = model.run_with_cache(text, remove_batch_dim=True)
```

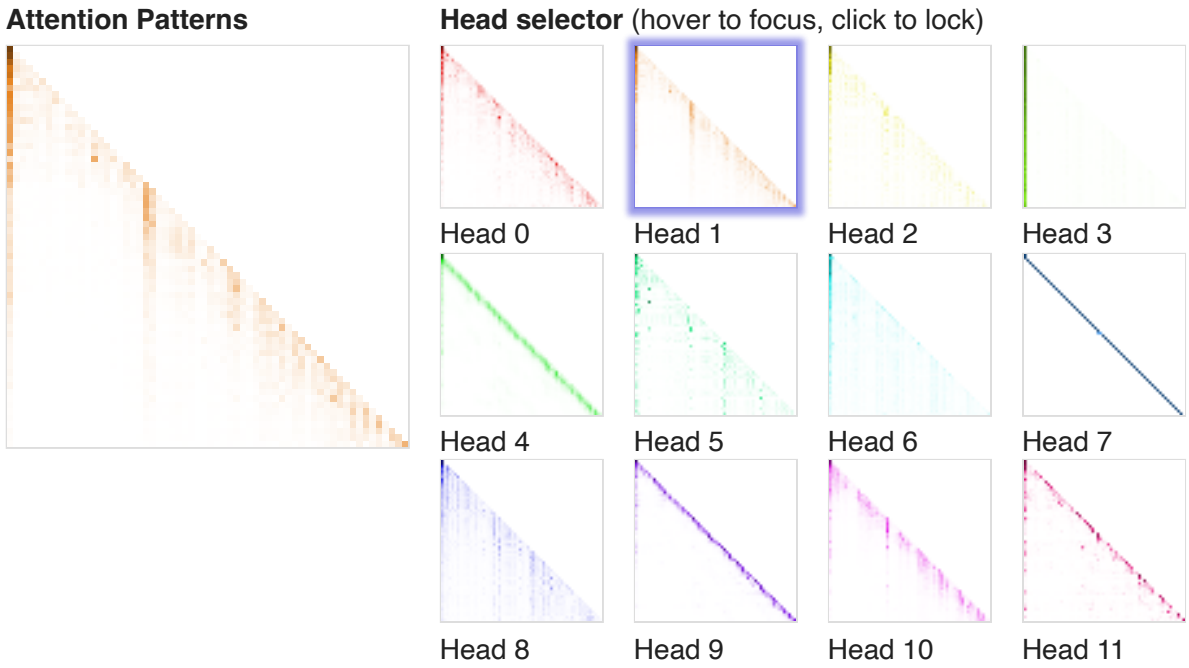
(Note that we've run the model on the string `text`, rather than on tokens like we did previously when creating a cache - this is something that `HookedTransformer` allows.)

Inspect the attention patterns. What do you notice about the attention heads?

You should spot three relatively distinctive basic patterns, which occur in multiple heads. What are these patterns, and can you guess why they might be present?

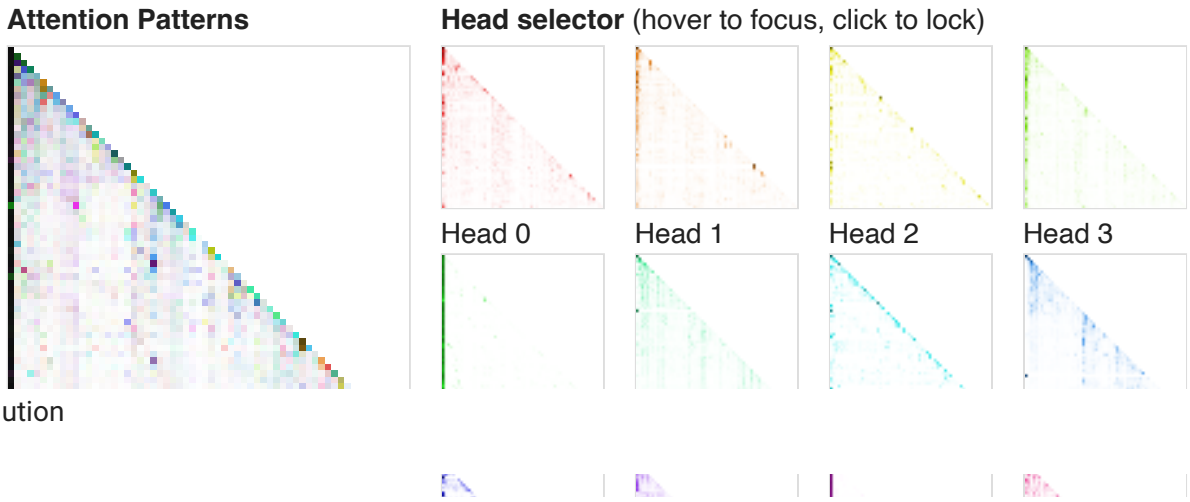
▶ Aside - what to do if your plots won't show up

```
# YOUR CODE HERE - visualize attention
str_tokens = model.to_str_tokens(text)
for layer in range(model.cfg.n_layers):
    attention_pattern = cache["pattern", layer]
    display(cv.attention.attention_patterns(tokens=str_tokens, attention=attent
```



Tokens (click to focus) Source ← Destination

<|endoftext|>We think that powerful, significantly superhuman machine intelligence is more likely than not to be created this century. If current machine learning techniques were scaled up to this level, we think they would by default produce systems that are deceptive or manipulative, and that no solid plans are known for how to avoid this.



▶ Solution

Now that we've observed our three basic attention patterns, it's time to make detectors for those patterns!

Head 8 Head 9 Head 10 Head 11

Exercise - write your own detectors

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

You shouldn't spend more than 15-20 minutes on these exercises.
```

```
These exercises shouldn't be too challenging, if you understand attention patterns. Use th
```

You should fill in the functions below, which act as detectors for particular types of heads. Validate your detectors by comparing these results to the visual attention patterns above - summary statistics on their own can be dodgy, but are much more reliable if you can validate it by directly playing with the data.

Tasks like this are useful, because we need to be able to take our observations / intuitions about what a model is doing, and translate these into quantitative measures. As the exercises proceed, we'll be creating some much more interesting tools and detectors!

Note - there's no objectively correct answer for which heads are doing which tasks, and which detectors can spot them. You should just try and come up with something plausible-seeming, which identifies the kind of behaviour you're looking for.

```
def current_attn_detector(cache: ActivationCache) -> List[str]:
    '''
    Returns a list e.g. ["0.2", "1.4", "1.9"] of "layer.head" which you judge to
    '''
    # SOLUTION
    attn_heads = []
    for layer in range(model.cfg.n_layers):
        for head in range(model.cfg.n_heads):
            attention_pattern = cache["pattern", layer][head]
            # take avg of diagonal elements
            score = attention_pattern.diagonal().mean()
            if score > 0.4:
                attn_heads.append(f"{layer}.{head}")
    return attn_heads

def prev_attn_detector(cache: ActivationCache) -> List[str]:
    '''
    Returns a list e.g. ["0.2", "1.4", "1.9"] of "layer.head" which you judge to
    '''
    # SOLUTION
    attn_heads = []
    for layer in range(model.cfg.n_layers):
        for head in range(model.cfg.n_heads):
            attention_pattern = cache["pattern", layer][head]
            # take avg of sub-diagonal elements
            score = attention_pattern.diagonal(-1).mean()
            if score > 0.4:
                attn_heads.append(f"{layer}.{head}")
    return attn_heads

def first_attn_detector(cache: ActivationCache) -> List[str]:
    '''
    Returns a list e.g. ["0.2", "1.4", "1.9"] of "layer.head" which you judge to
    '''
    # SOLUTION
```

```
    attn_heads = []
    for layer in range(model.cfg.n_layers):
        for head in range(model.cfg.n_heads):
            attention_pattern = cache["pattern", layer][head]
            # take avg of 0th elements
            score = attention_pattern[:, 0].mean()
            if score > 0.4:
                attn_heads.append(f"{layer}.{head}")
    return attn_heads


print("Heads attending to current token  = ", ", ".join(current_attn_detector(ca
print("Heads attending to previous token = ", ", ".join(prev_attn_detector(cache
print("Heads attending to first token    = ", ", ".join(first_attn_detector(cach
```

```
Heads attending to current token  =  0.9
Heads attending to previous token =  0.7
Heads attending to first token    =  0.3, 1.4, 1.10
```

▶ Hint

▶ Solution

Compare the printouts to your attention visualisations above. Do they seem to make sense?

Bonus - try different text

Try inputting different text, and see how stable your results are. Do you always get the same classifications for heads?

Now, it's time to turn our attention to induction heads.

What are induction heads?

(Note: I use induction **head** to refer to the head in the second layer which attends to the 'token immediately after the copy of the current token', and induction **circuit** to refer to the circuit consisting of the composition of a **previous token head** in layer 0 and an **induction head** in layer 1)

Induction heads are the first sophisticated circuit we see in transformers! And are sufficiently interesting that we wrote another paper just about them.

▶ An aside on why induction heads are a big deal

Again, you are strongly recommended to read the corresponding section of the glossary, before continuing (or this LessWrong post). In brief, however, the induction circuit consists of a previous token head in layer 0 and an induction head in layer 1, where the induction head learns to attend to the token immediately *after* copies of the current token via K-Composition with the previous token head.

Question - why couldn't an induction head form in a 1L model?

▶ Answer

Checking for the induction capability

A striking thing about models with induction heads is that, given a repeated sequence of random tokens, they can predict the repeated half of the sequence. This is nothing like it's training data, so this is kind of wild! The ability to predict this kind of out of distribution generalisation is a strong point of evidence that you've really understood a circuit.

To check that this model has induction heads, we're going to run it on exactly that, and compare performance on the two halves - you should see a striking difference in the per token losses.

Note - we're using small sequences (and just one sequence), since the results are very obvious and this makes it easier to visualise. In practice we'd obviously use larger ones on more subtle tasks. But it's often easiest to iterate and debug on small tasks.

Exercise - plot per-token loss on repeated sequence

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

You shouldn't spend more than 10-15 minutes on these exercises.
```

You should fill in the functions below. We've given you the first line of the first function, which defines a prefix (remember we need the BOS token for GPT-2, since it was trained to have one).

```
def generate_repeated_tokens(
    model: HookedTransformer, seq_len: int, batch: int = 1
) -> Int[Tensor, "batch full_seq_len"]:
    '''
    Generates a sequence of repeated random tokens

    Outputs are:
        rep_tokens: [batch, 1+2*seq_len]
    '''
    # SOLUTION
    prefix = (t.ones(batch, 1) * model.tokenizer.bos_token_id).long()
    rep_tokens_half = t.randint(0, model.cfg.d_vocab, (batch, seq_len), dtype=t.
    rep_tokens = t.cat([prefix, rep_tokens_half, rep_tokens_half], dim=-1).to(de
    return rep_tokens



def run_and_cache_model_repeated_tokens(model: HookedTransformer, seq_len: int,
    '''
    Generates a sequence of repeated random tokens, and runs the model on it, re

```

```
    Should use the `generate_repeated_tokens` function above

    Outputs are:
        rep_tokens: [batch, 1+2*seq_len]
        rep_logits: [batch, 1+2*seq_len, d_vocab]
        rep_cache: The cache of the model run on rep_tokens
    '''
    # SOLUTION
    rep_tokens = generate_repeated_tokens(model, seq_len, batch)
    rep_logits, rep_cache = model.run_with_cache(rep_tokens)
    return rep_tokens, rep_logits, rep_cache


seq_len = 50
batch = 1
(rep_tokens, rep_logits, rep_cache) = run_and_cache_model_repeated_tokens(model,
rep_cache.remove_batch_dim()
rep_str = model.to_str_tokens(rep_tokens)
model.reset_hooks()
log_probs = get_log_probs(rep_logits, rep_tokens).squeeze()

print(f"Performance on the first half: {log_probs[:seq_len].mean():.3f}")
print(f"Performance on the second half: {log_probs[seq_len:].mean():.3f}")

plot_loss_difference(log_probs, rep_str, seq_len)
```

```
Performance on the first half: -14.240
```

▶ Hint

▶ Solution

Per token log-prob on correct token

Looking for Induction Attention Patterns

The next natural thing to check for is the induction attention pattern.

First, go back to the attention patterns visualisation code from earlier (i.e. `cv.attention.attention_heads` or `attention_patterns`) and manually check for likely heads in the second layer. Which ones do you think might be serving as induction heads?

Note - above, we defined the `rep_str` object for you, so you can use it in your `circuitsvis` functions.

```
1 # YOUR CODE HERE - display the attention patterns stored in `rep_cache`, for eac
```

▶ What you should see (only click after you've made your own observations):

Exercise - make an induction-head detector

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

You shouldn't spend more than 5-10 minutes on this exercise.

This exercise should be very similar to the earlier detector exercises (with a slightly mc
```

Now, you should make an induction pattern score function, which looks for the average attention paid to the offset diagonal. Do this in the same style as our earlier head scorers.

```
1 def induction_attn_detector(cache: ActivationCache) -> List[str]:
2     '''
3     Returns a list e.g. ["0.2", "1.4", "1.9"] of "layer.head" which you judge to
4 
5     Remember - the tokens used to generate rep_cache are (bos_token, *rand_token
6     '''
7     # SOLUTION
8     attn_heads = []
9     for layer in range(model.cfg.n_layers):
10         for head in range(model.cfg.n_heads):
11             attention_pattern = cache["pattern", layer][head]
12             # take avg of (-seq_len+1)-offset elements
13             seq_len = (attention_pattern.shape[-1] - 1) // 2
14             score = attention_pattern.diagonal(-seq_len+1).mean()
```

```
            if score > 0.4:
                attn_heads.append(f"{layer}.{head}")
    return attn_heads


print("Induction heads = ", ", ".join(induction_attn_detector(rep_cache)))
```

```
Induction heads =  1.4, 1.10
```

If this function works as expected, then you should see output that matches your observations from `circuitsvis` (i.e. the heads which you observed to be induction heads are being classified as induction heads by your function here).

▶ Help - I'm not sure what offset to use.

▶ Solution

3️⃣ TransformerLens: Hooks

> Learning objectives
>
> - Understand what hooks are, and how they are used in TransformerLens
> - Use hooks to access activations, process the results, and write them to an external tensor
> - Build tools to perform attribution, i.e. detecting which components of your model are responsible for performance on a given task
> - Understand how hooks can be used to perform basic interventions like **ablation**

What are hooks?

One of the great things about interpreting neural networks is that we have *full control* over our system. From a computational perspective, we know exactly what operations are going on inside (even if we don't know what they mean!). And we can make precise, surgical edits and see how the model's behaviour and other internals change. This is an extremely powerful tool, because it can let us e.g. set up careful counterfactuals and causal intervention to easily understand model behaviour.

Accordingly, being able to do this is a pretty core operation, and this is one of the main things TransformerLens supports! The key feature here is **hook points**. Every activation inside the transformer is surrounded by a hook point, which allows us to edit or intervene on it.

We do this by adding a **hook function** to that activation, and then calling `model.run_with_hooks`.

(Terminology note - because basically all the activations in our model have an associated hook point, we'll sometimes use the terms "hook" and "activation" interchangeably.)

Hook functions

Hook functions take two arguments: `activation_value` and `hook_point`. The `activation_value` is a tensor representing some activation in the model, just like the values in our `ActivationCache`. The `hook_point` is an object which gives us methods like `hook.layer()` or attributes like `hook.name` that are sometimes useful to call within the function.

If we're using hooks to edit activations, then the hook function should return a tensor of the same shape as the activation value. But we can also just have our hook function access the activation, do some processing, and write the results to some external variable (in which case our hook function should just not return anything).

An example hook function for changing the attention patterns at a particular layer might look like:

```
def hook_function(
    attn_pattern: Float[Tensor, "batch heads seq_len seq_len"],
    hook: HookPoint
) -> TT["batch", "heads", "seq_len", "seq_len"]:

    # modify attn_pattern (can be inplace)
    return attn_pattern
```

Running with hooks

Once you've defined a hook function (or functions), you should call `model.run_with_hooks`. A typical call to this function might look like:

```
loss = model.run_with_hooks(
    tokens,
    return_type="loss",
    fwd_hooks=[
        ('blocks.1.attn.hook_pattern', hook_function)
    ]
)
```

Let's break this code down.

- `tokens` represents our model's input.
- `return_type="loss"` is used here because we're modifying our activations and seeing how this affects the loss.
    - We could also return the logits, or just use `return_type=None` if we only want to access the intermediate activations and we don't care about the output.
- `fwd_hooks` is a list of 2-tuples of (hook name, hook function).
    - The hook name is a string that specifies which activation we want to hook.
    - The hook function gets run with the corresponding activation as its first argument.

A bit more about hooks

Here are a few extra notes for how to squeeze even more functionality out of hooks. If you'd prefer, you can jump ahead to see an actual example of hooks being used, and come back to this section later.

▶ Resetting hooks

▶ Adding multiple hooks at once

▶ `utils.get_act_name`

▶ Using `functools.partial` to create variations on hooks

And here are some points of interest, which aren't vital to understand:

▶ Relationship to PyTorch hooks

▶ How are TransformerLens hooks actually implemented?

Hooks: Accessing Activations

In later sections, we'll write some code to intervene on hooks, which is really the core feature that makes them so useful for interpretability. But for now, let's just look at how to access them without changing their value. This can be achieved by having the hook function write to a global variable, and return nothing (rather than modifying the activation in place).

Why might we want to do this? It turns out to be useful for things like:

- Extracting activations for a specific task
- Doing some long-running calculation across many inputs, e.g. finding the text that most activates a specific neuron

Note that, in theory, this could all be done using the `run_with_cache` function we used in the previous section, combined with post-processing of the cache result. But using hooks can be more intuitive and memory efficient.

Exercise - calculate induction scores with hooks

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

You shouldn't spend more than 15-20 minutes on this exercise.

This is our first exercise with hooks, which are an absolutely vital TransformerLens tool.
```

To start with, we'll look at how hooks can be used to get the same results as from the previous section (where we ran our induction head detector functions on the values in the cache).

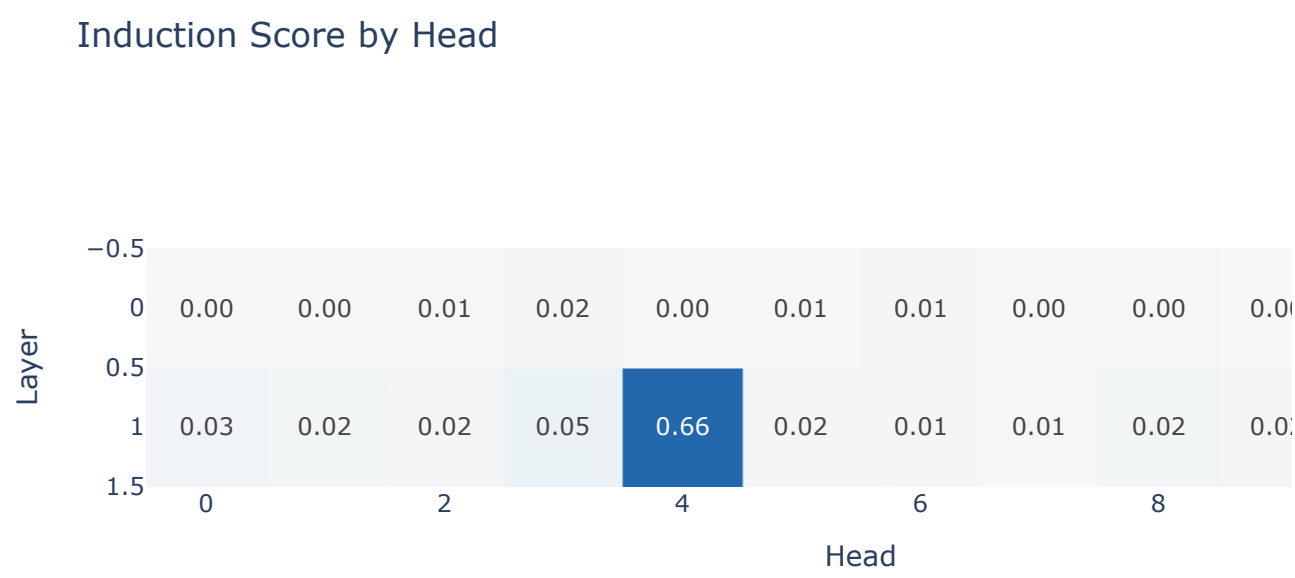
Most of the code has already been provided for you below; the only thing you need to do is **implement the `induction_score_hook` function**. As mentioned, this function takes two arguments: the activation value (which in this case will be our attention pattern) and the hook object (which gives us some useful methods and attributes that we can access in the function, e.g. `hook.layer()` to return the layer, or `hook.name` to return the name, which is the same as the name in the cache).

Your function should do the following:

- Calculate the induction score for the attention pattern `pattern`, using the same methodology as you used in the previous section when you wrote your induction head detectors.
    - Note that this time, the batch dimension is greater than 1, so you should compute the average attention score over the batch dimension.
    - Also note that you are computing the induction score for all heads at once, rather than one at a time. You might find the arguments `dim1` and `dim2` of the `torch.diagonal` function useful.
- Write this score to the tensor `induction_score_store`, which is a global variable that we've provided for you. The `[i, j]` th element of this tensor should be the induction score for the `j` th head in the `i` th layer.

```
seq_len = 50
batch = 10
rep_tokens_10 = generate_repeated_tokens(model, seq_len, batch)

# We make a tensor to store the induction score for each head.
# We put it on the model's device to avoid needing to move things between the GP
induction_score_store = t.zeros((model.cfg.n_layers, model.cfg.n_heads), device=

def induction_score_hook(
    pattern: Float[Tensor, "batch head_index dest_pos source_pos"],
    hook: HookPoint,
):
    '''
    Calculates the induction score, and stores it in the [layer, head] position
    '''
```

```
    # SOLUTION
    # Take the diagonal of attn paid from each dest posn to src posns (seq_len-1
    # (This only has entries for tokens with index>=seq_len)
    induction_stripe = pattern.diagonal(dim1=-2, dim2=-1, offset=1-seq_len)
    # Get an average score per head
    induction_score = einops.reduce(induction_stripe, "batch head_index position
    # Store the result.
    induction_score_store[hook.layer(), :] = induction_score

# We make a boolean filter on activation names, that's true only on attention pa
pattern_hook_names_filter = lambda name: name.endswith("pattern")

# Run with hooks (this is where we write to the `induction_score_store` tensor`)
model.run_with_hooks(
    rep_tokens_10,
    return_type=None, # For efficiency, we don't need to calculate the logits
    fwd_hooks=[(
        pattern_hook_names_filter,
        induction_score_hook
    )]
)

# Plot the induction scores for each head in each layer
imshow(
    induction_score_store,
    labels={"x": "Head", "y": "Layer"},
    title="Induction Score by Head",
    text_auto=".2f",
    width=900, height=400
)
```



If this function has been implemented correctly, you should see a result matching your observations from the previous section: a high induction score (close to 1) for all the heads which you identified as induction heads, and a low score (close to 0) for all others.

▶ Help - I'm not sure how to implement this function.

▶ Solution

Exercise - find induction heads in GPT2-small

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

You shouldn't spend more than 10-20 minutes on this exercise.

Here, you mostly just need to use previously defined functions and interpret the results,
```

This is your first opportunity to investigate a larger and more extensively trained model, rather than the simple 2-layer model we've been using so far. None of the code required is new (you can copy most of it from previous sections), so these exercises shouldn't take very long.

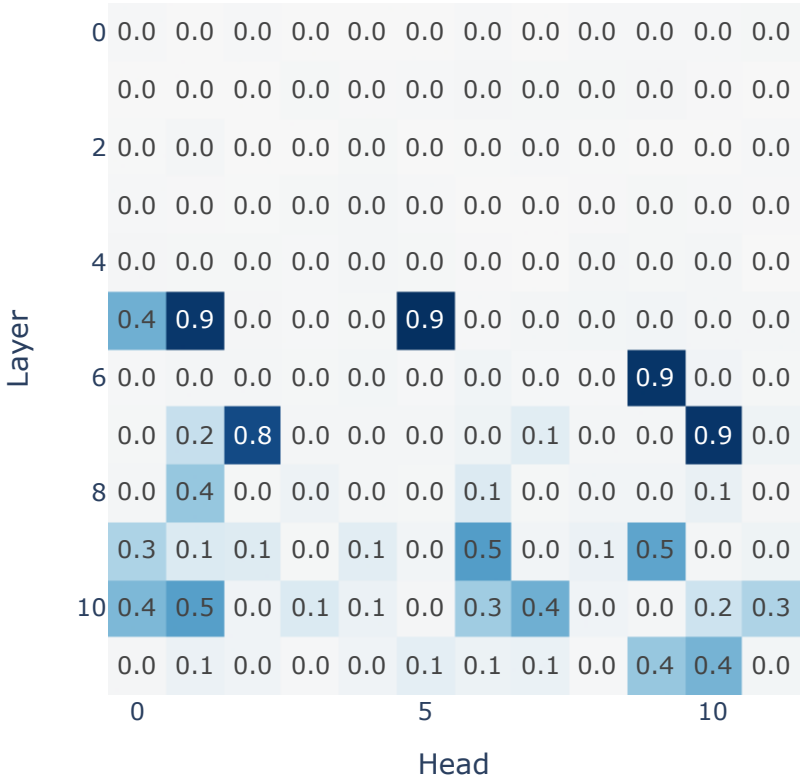
Perform the same analysis on your `gpt2_small`. You should observe that some heads, particularly in a couple of the middle layers, have high induction scores. Use CircuitsVis to plot the attention patterns for these heads when run on the repeated token sequences, and verify that they look like induction heads.

Note - you can make CircuitsVis plots (and other visualisations) using hooks rather than plotting directly from the cache. For example, we've given you a hook function which will display the attention patterns at a given hook when you include it in a call to `model.run_with_hooks`.

```
def visualize_pattern_hook(
    pattern: Float[Tensor, "batch head_index dest_pos source_pos"],
    hook: HookPoint,
):
    print("Layer: ", hook.layer())
    display(
        cv.attention.attention_patterns(
            tokens=gpt2_small.to_str_tokens(rep_tokens[0]),
            attention=pattern.mean(0)
        )
    )


# YOUR CODE HERE - find induction heads in gpt2_small
seq_len = 50
batch = 10
rep_tokens_10 = generate_repeated_tokens(gpt2_small, seq_len, batch)

```

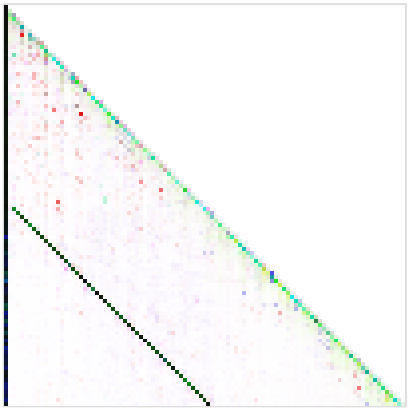
```
induction_score_store = t.zeros((gpt2_small.cfg.n_layers, gpt2_small.cfg.n_heads

gpt2_small.run_with_hooks(
    rep_tokens_10,
    return_type=None, # For efficiency, we don't need to calculate the logits
    fwd_hooks=[(
        pattern_hook_names_filter,
        induction_score_hook
    )]
)

imshow(
    induction_score_store,
    labels={"x": "Head", "y": "Layer"},
    title="Induction Score by Head",
    text_auto=".1f",
    width=800
)

# Observation: heads 5.1, 5.5, 6.9, 7.2, 7.10 are all strongly induction-y.
# Confirm observation by visualizing attn patterns for layers 5 through 7:

for induction_head_layer in [5, 6, 7]:
    gpt2_small.run_with_hooks(
        rep_tokens,
        return_type=None, # For efficiency, we don't need to calculate the logit
        fwd_hooks=[
            (utils.get_act_name("pattern", induction_head_layer), visualize_patt
        ]
    )
```

Induction Score by Head

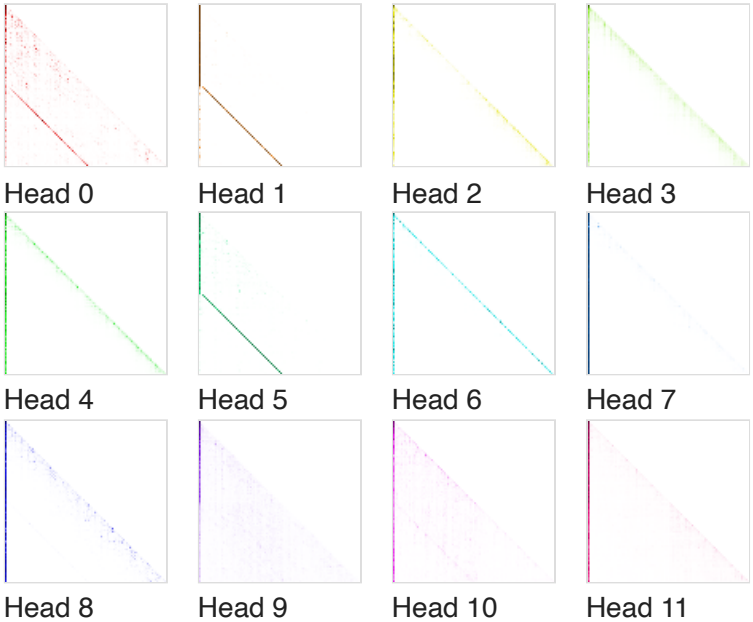



```
Layer:  5
```

Attention Patterns



Head selector (hover to focus, click to lock)



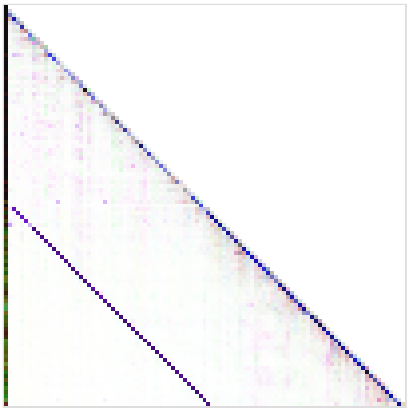
Tokens (click to focus) Source ← Destination

!Maker poetynamic ValueQueen concealedarn worldwide omin Kendrickrepaker timesMaria BrmA eleph ought Latvia plate 137 vividly're athlete mdOct nurture NFC doubts bacter dimensionGeneric noted TCP Override cyber ard freeing Service localization Monetary Hak offic crept Palest biting bonedienthandler introduMaker poetynamic ValueQueen concealedarn worldwide omin Kendrickrepaker timesMaria BrmA

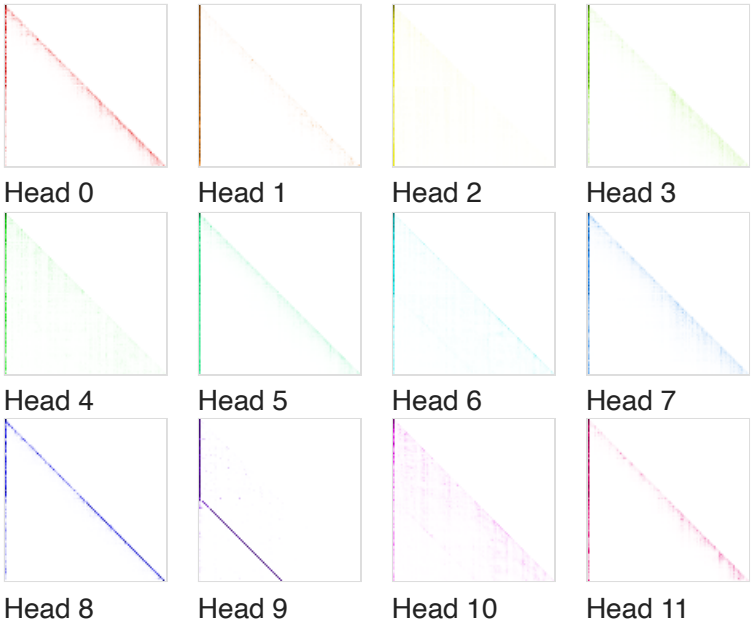
eleph ought Latvia plate 137 vividly're athlete mdOct nurture NFC doubts bacter dimensionGeneric noted TCPOverride cyber ard freeing Service localization Monetary Hak offic crept Palest biting bonedienthandler introdu

```
Layer:  6
```

Attention Patterns



Head selector (hover to focus, click to lock)



Tokens (click to focus) Source ← Destination

!Maker poetynamic ValueQueen concealedarn worldwide omin Kendrickrepaker timesMaria BrmA eleph ought Latvia plate 137 vividly're athlete mdOct nurture NFC doubts bacter dimensionGeneric noted TCP Override cyber ard freeing Service localization Monetary Hak offic crept Palest biting bonedienthandler introduMaker poetynamic ValueQueen concealedarn worldwide omin Kendrickrepaker timesMaria BrmA eleph ought Latvia plate 137 vividly're athlete mdOct nurture NFC doubts bacter dimensionGeneric noted TCPOverride cyber ard freeing Service localization Monetary Hak offic crept Palest biting bonedienthandler introdu

```
Layer:  7
```

Attention Patterns

Head selector (hover to focus, click to lock)

▶ Solution

Building interpretability tools

In order to develop a mechanistic understanding for how transformers perform certain tasks, we need to be able to answer questions like:

> *How much of the model's performance on some particular task is attributable to each component of the model?*

where "component" here might mean, for example, a specific head in a layer.

There are many ways to approach a question like this. For example, we might look at how a head interacts with other heads in different layers, or we might perform a causal intervention by seeing how well the model performs if we remove the effect of this head. However, we'll keep things simple for now, and ask the question: **what are the direct contributions of this head to the output logits?**

Direct Logit attribution

A consequence of the residual stream is that the output logits are the sum of the contributions of each layer, and thus the sum of the results of each head. This means we can decompose the output logits into a term coming from each head and directly do attribution like this!

▶ A concrete example

Your mission here is to write a function to look at how much each component contributes to the correct logit. Your components are:

- The direct path (i.e. the residual connections from the embedding to unembedding),
- Each layer 0 head (via the residual connection and skipping layer 1)
- Each layer 1 head

To emphasise, these are not paths from the start to the end of the model, these are paths from the output of some component directly to the logits - we make no assumptions about how each path was calculated!

A few important notes for this exercise:

- Here we are just looking at the DIRECT effect on the logits, i.e. the thing that this component writes / embeds into the residual stream - if heads compose with other heads and affect logits like that, or inhibit logits for other tokens to boost the correct one we will not pick up on this!
- By looking at just the logits corresponding to the correct token, our data is much lower dimensional because we can ignore all other tokens other than the correct next one (Dealing with a 50K vocab size is a pain!). But this comes at the cost of missing out on more subtle effects, like a head suppressing other plausible logits, to increase the log prob of the correct one.
    - There are other situations where our job might be easier. For instance, in the IOI task (which we'll discuss shortly) we're just comparing the logits of the indirect object to the logits of the direct object, meaning we can use the **difference between these logits**, and ignore all the other logits.
- When calculating correct output logits, we will get tensors with a dimension `(position - 1,)`, not `(position,)` - we remove the final element of the output (logits), and the first element of labels (tokens). This is because we're predicting the *next* token, and we don't know the token after the final token, so we ignore it.

▶ Aside - centering `W_U`

▶ Question - why don't we do this to the log probs instead?

Exercise - build logit attribution tool

Difficulty: 🟠🟠🟠⚪⚪
Importance: 🟠🟠🟠🟠⚪

You shouldn't spend more than 10-15 minutes on this exercise.

This exercise is important, but has quite a few messy einsums, so you might get more value

You should implement the `logit_attribution` function below. This should return the contribution of each component in the "correct direction". We've already given you the unembedding vectors for the correct direction, `W_U_correct_tokens` (note that we take the `[1:]` slice of tokens, for reasons discussed above).

The code below this function will check your logit attribution function is working correctly, by taking the sum of logit attributions and comparing it to the actual values in the residual stream at the end of your model.

```
def logit_attribution(
    embed: Float[Tensor, "seq d_model"],
    l1_results: Float[Tensor, "seq nheads d_model"],
    l2_results: Float[Tensor, "seq nheads d_model"],
    W_U: Float[Tensor, "d_model d_vocab"],
    tokens: Float[Tensor, "seq"]
) -> Float[Tensor, "seq-1 n_components"]:
    '''
    Inputs:
        embed: the embeddings of the tokens (i.e. token + position embeddings)
        l1_results: the outputs of the attention heads at layer 1 (with head as
        l2_results: the outputs of the attention heads at layer 2 (with head as
        W_U: the unembedding matrix
        tokens: the token ids of the sequence

    Returns:
        Tensor of shape (seq_len-1, n_components)
        represents the concatenation (along dim=-1) of logit attributions from:
            the direct path (seq-1,1)
            layer 0 logits (seq-1, n_heads)
            layer 1 logits (seq-1, n_heads)
        so n_components = 1 + 2*n_heads
    '''
    W_U_correct_tokens = W_U[:, tokens[1:]]
    # SOLUTION
    direct_attributions = einops.einsum(W_U_correct_tokens, embed[:-1], "emb seq
    l1_attributions = einops.einsum(W_U_correct_tokens, l1_results[:-1], "emb se
    l2_attributions = einops.einsum(W_U_correct_tokens, l2_results[:-1], "emb se
```

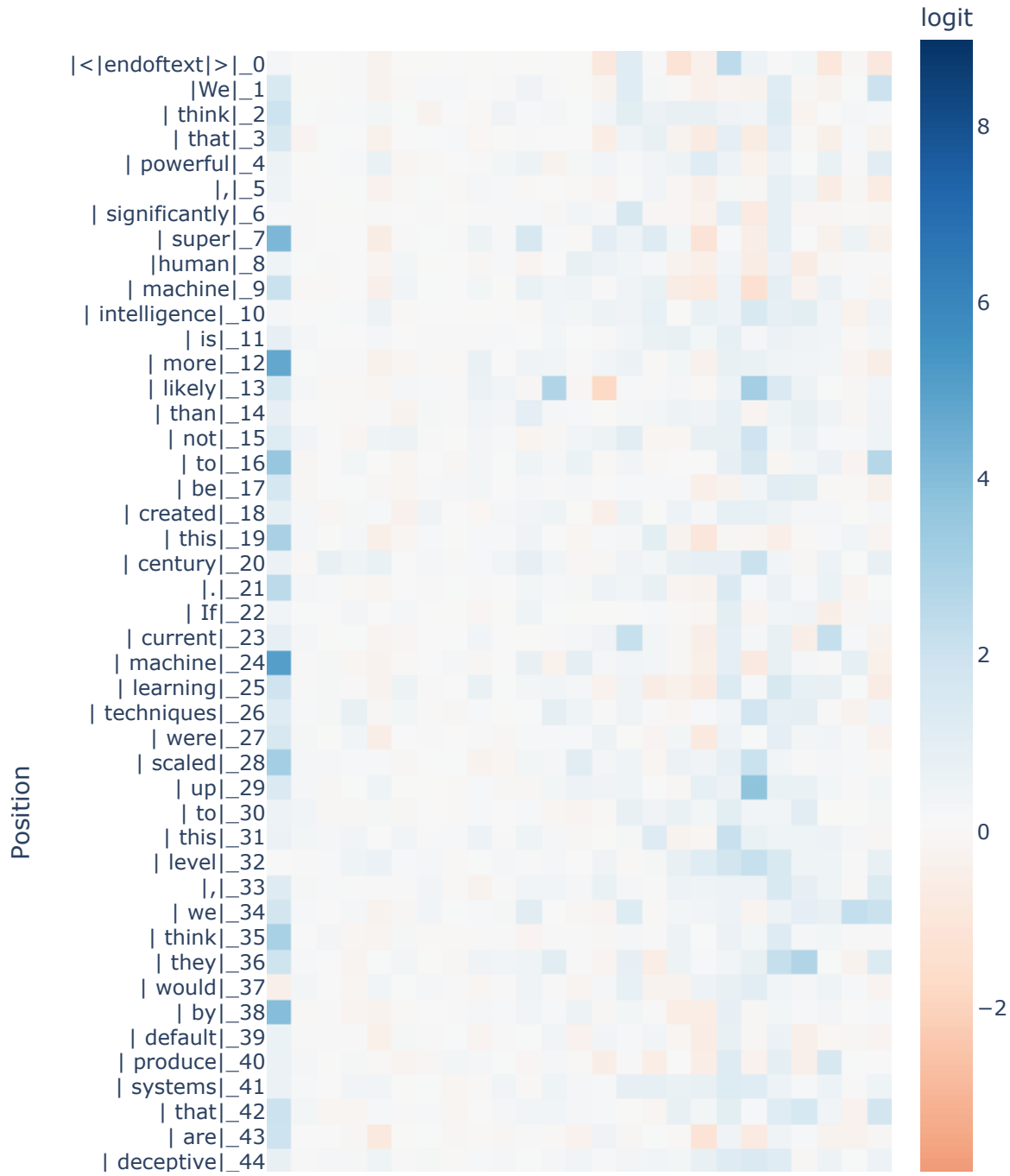
```
    return t.concat([direct_attributions.unsqueeze(-1), l1_attributions, l2_attr


text = "We think that powerful, significantly superhuman machine intelligence is
logits, cache = model.run_with_cache(text, remove_batch_dim=True)
str_tokens = model.to_str_tokens(text)
tokens = model.to_tokens(text)

with t.inference_mode():
    embed = cache["embed"]
    l1_results = cache["result", 0]
    l2_results = cache["result", 1]
    logit_attr = logit_attribution(embed, l1_results, l2_results, model.W_U, tok
    # Uses fancy indexing to get a len(tokens[0])-1 length tensor, where the kth
    correct_token_logits = logits[0, t.arange(len(tokens[0]) - 1), tokens[0, 1:]
    t.testing.assert_close(logit_attr.sum(1), correct_token_logits, atol=1e-3, r
    print("Tests passed!")
```

```
Tests passed!
```

▶ Solution

Once you've got the tests working, you can visualise the logit attributions for each path through the model. We've provided you with the helper function `plot_logit_attribution`, which presents the results in a nice way.

```
embed = cache["embed"]
l1_results = cache["result", 0]
l2_results = cache["result", 1]
logit_attr = logit_attribution(embed, l1_results, l2_results, model.W_U, tokens[

plot_logit_attribution(model, logit_attr, tokens)
```



Question - what is the interpretation of this plot?

You should find that the most variation in the logit attribution comes from the direct path. In particular, some of the tokens in the direct path have a very high logit attribution (e.g. tokens 12, 24 and 46). Can you guess what gives them in particular such a high logit attribution?

▶ Answer

Another feature of the plot - the heads in the second layer seem to have much higher contributions than the heads in the first layer. Why do you think this might be?

▶ Hint

▶ Answer

Exercise - logit attribution for the induction heads

Difficulty: 🟠🟠🟠⚪⚪
Importance: 🟠🟠⚪⚪⚪

You shouldn't spend more than ~10 minutes on this exercise.

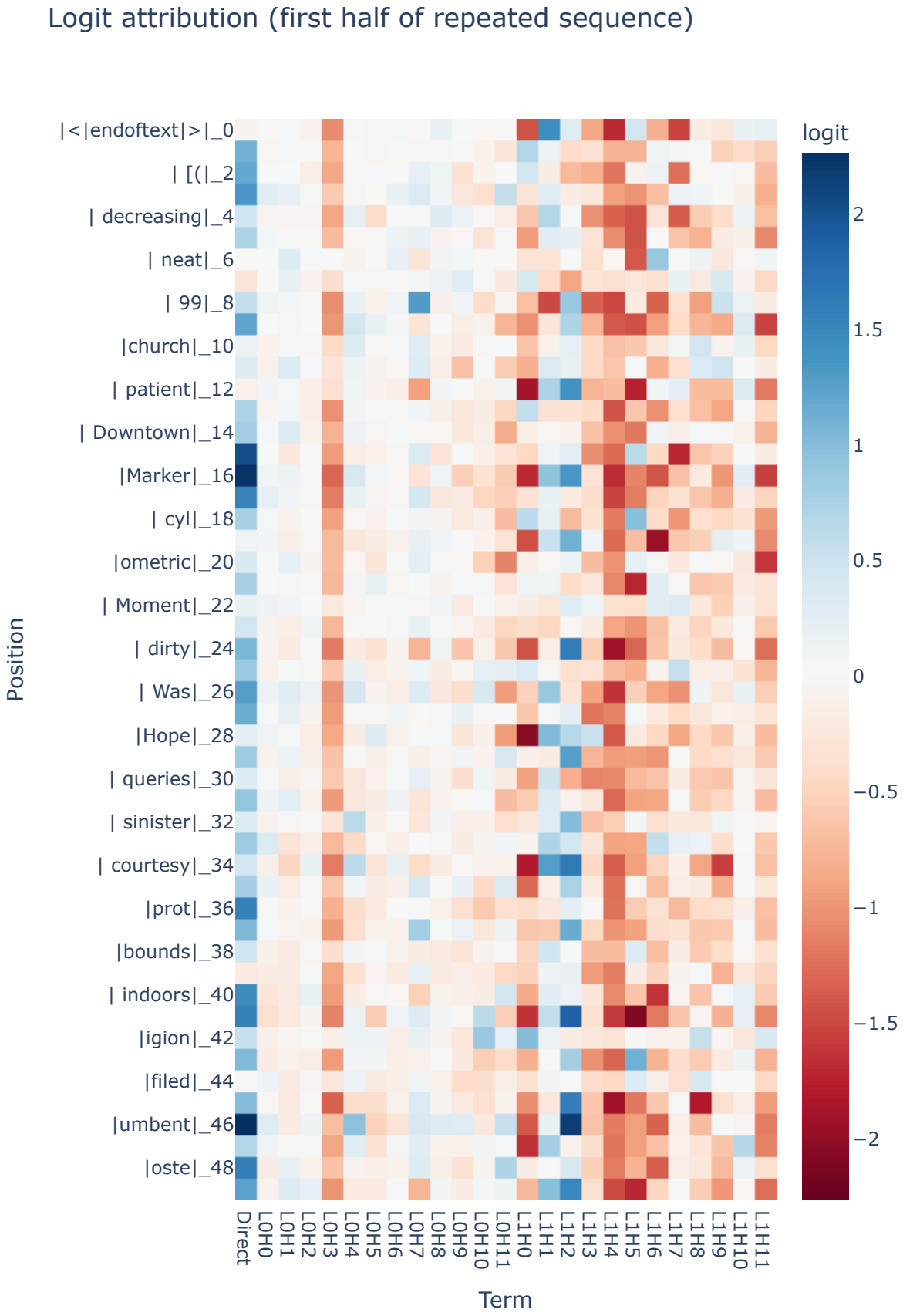
This exercise just involves calling logit_attribution with appropriate arguments. Don't sp

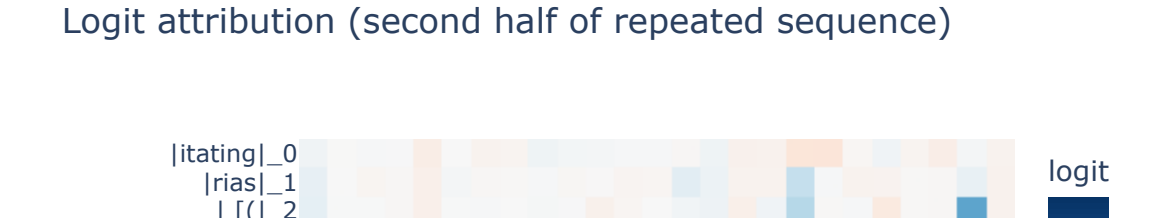
Perform logit attribution for your attention-only model `model`, on the `rep_cache`. What do you expect to see?

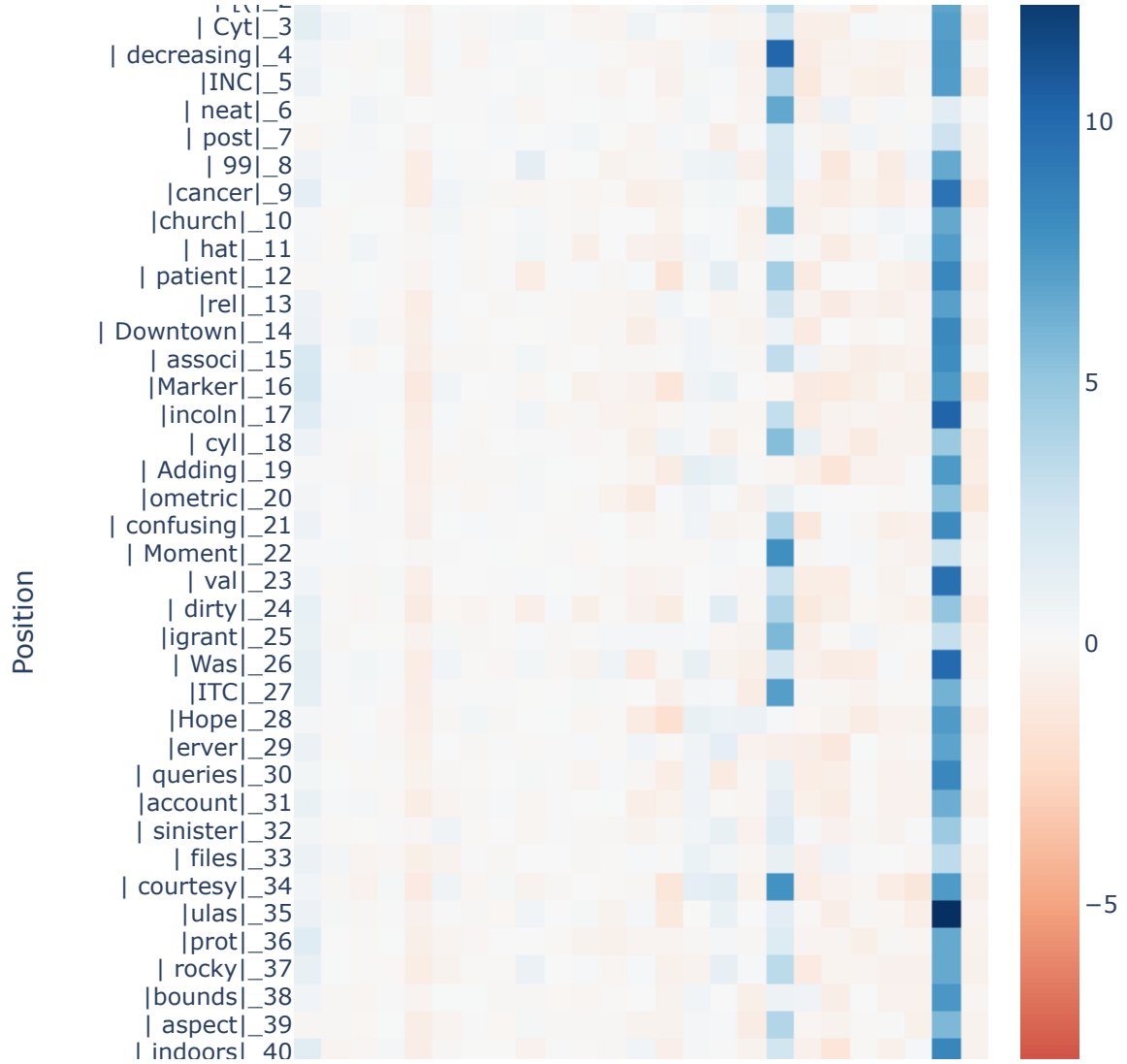
Remember, you'll need to split the sequence in two, with one overlapping token (since predicting the next token involves removing the final token with no label) - your `logit_attr` should both have shape `[seq_len, 2*n_heads + 1]` (ie `[50, 25]` here).

▶ Note - the first plot will be pretty meaningless. Can you see why?

```
seq_len = 50

embed = rep_cache["embed"]
l1_results = rep_cache["result", 0]
l2_results = rep_cache["result", 1]
first_half_tokens = rep_tokens[0, : 1 + seq_len]
second_half_tokens = rep_tokens[0, seq_len:]


# YOUR CODE HERE - define `first_half_logit_attr` and `second_half_logit_attr`
# (each with a single call to the `logit_attribution` function)
first_half_logit_attr = logit_attribution(embed[:seq_len+1], l1_results[:seq_len
second_half_logit_attr = logit_attribution(embed[seq_len:], l1_results[seq_len:]


assert first_half_logit_attr.shape == (seq_len, 2*model.cfg.n_heads + 1)
assert second_half_logit_attr.shape == (seq_len, 2*model.cfg.n_heads + 1)

plot_logit_attribution(model, first_half_logit_attr, first_half_tokens, "Logit a
plot_logit_attribution(model, second_half_logit_attr, second_half_tokens, "Logit
```


Logit attribution (first half of repeated sequence)
Position
|<|endoftext|>|_0
| [(|_2
| decreasing|_4
| neat|_6
| 99|_8
|church|_10
| patient|_12
| Downtown|_14
|Marker|_16
| cyl|_18
|ometric|_20
| Moment|_22
| dirty|_24
| Was|_26
|Hope|_28
| queries|_30
| sinister|_32
| courtesy|_34
|prot|_36
|bounds|_38
| indoors|_40
|igion|_42
|filed|_44
|umbent|_46
|oste|_48
Direct
L0H0
L0H1
L0H2
L0H3
L0H4
L0H5
L0H6
L0H7
L0H8
L0H9
L0H10
L0H11
L1H0
L1H1
L1H2
L1H3
L1H4
L1H5
L1H6
L1H7
L1H8
L1H9
L1H10
L1H11
Term
logit
2
1.5
1
0.5
0
−0.5
−1
−1.5
−2

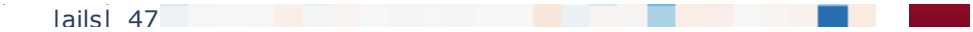

Logit attribution (second half of repeated sequence)
|itating|_0
|rias|_1
logit



▶ Solution

What is the interpretation of this plot, in the context of our induction head circuit?

▶ Answer



Hooks: Intervening on Activations

Now that we've built some tools to decompose our model's output, it's time to start making causal interventions.

Ablations

Let's start with a simple example: **ablation**. An ablation is a simple causal intervention on a model - we pick some part of it and set it to zero. This is a crude proxy for how much that part matters. Further, if we have some story about how a specific circuit in the model enables some capability, showing that ablating *other* parts does nothing can be strong evidence of this.

As mentioned in [the glossary], there are many ways to do ablation. We'll focus on the simplest: zero-ablation (even though it's somewhat unprincipled).

Exercise - induction head ablation

Difficulty: 🟠🟠⚪⚪⚪
Importance: 🟠🟠🟠🟠⚪

You shouldn't spend more than ~10 minutes on this exercise.

This exercise is conceptually important, but very short.

The code below provides a template for performing zero-ablation on the value vectors at a particular head (i.e. the vectors we get when applying the weight matrices `W_V` to the residual stream). If you're confused about what different activations mean, you can refer back to the diagram.

The only thing left for you to do is fill in the function `head_ablation_hook` so that it performs zero-ablation on the head given by `head_index_to_ablate`. In other words, your function should return a modified version of `value` with this head ablated. (Technically you don't have to return any tensor if you're modifying `value` in-place; this is just convention.)

A few notes to help explain the code below:

- In the `get_ablation_scores` function, we run our hook function in a for loop: once for each layer and each head. Each time, we write a single value to the tensor `ablation_scores` that stores the results of ablating that particular head.
- We use `cross_entropy_loss` as a metric for model performance, rather than logit difference like in the previous section.
    - If the head didn't have any effect on the output, then the ablation score would be zero (since the loss doesn't increase when we ablate).
    - If the head was very important for making correct predictions, then we should see a very large ablation score.
- We use `functools.partial` to create a temporary hook function with the head number fixed. This is a nice way to avoid having to write a separate hook function for each head.
- We use `utils.get_act_name` to get the name of our activation. This is a nice shorthand way of getting the full name.

```
def head_ablation_hook(
    attn_result: Float[Tensor, "batch seq n_heads d_model"],
    hook: HookPoint,
    head_index_to_ablate: int
) -> Float[Tensor, "batch seq n_heads d_model"]:
    # SOLUTION
    attn_result[:, :, head_index_to_ablate, :] = 0.0
    return attn_result


def cross_entropy_loss(logits, tokens):
```

```
    '''
    Computes the mean cross entropy between logits (the model's prediction) and
    '''
    log_probs = F.log_softmax(logits, dim=-1)
    pred_log_probs = t.gather(log_probs[:, :-1], -1, tokens[:, 1:, None])[..., 0
    return -pred_log_probs.mean()



def get_ablation_scores(
    model: HookedTransformer,
    tokens: Int[Tensor, "batch seq"]
) -> Float[Tensor, "n_layers n_heads"]:
    '''
    Returns a tensor of shape (n_layers, n_heads) containing the increase in cro
    '''
    # Initialize an object to store the ablation scores
    ablation_scores = t.zeros((model.cfg.n_layers, model.cfg.n_heads), device=mo

    # Calculating loss without any ablation, to act as a baseline
    model.reset_hooks()
    logits = model(tokens, return_type="logits")
    loss_no_ablation = cross_entropy_loss(logits, tokens)

    for layer in tqdm(range(model.cfg.n_layers)):
        for head in range(model.cfg.n_heads):
            # Use functools.partial to create a temporary hook function with the
            temp_hook_fn = functools.partial(head_ablation_hook, head_index_to_a
            # Run the model with the ablation hook
            ablated_logits = model.run_with_hooks(tokens, fwd_hooks=[
                (utils.get_act_name("result", layer), temp_hook_fn)
            ])
            # Calculate the logit difference
            loss = cross_entropy_loss(ablated_logits, tokens)
            # Store the result, subtracting the clean loss so that a value of ze
            ablation_scores[layer, head] = loss - loss_no_ablation

    return ablation_scores


ablation_scores = get_ablation_scores(model, rep_tokens)
tests.test_get_ablation_scores(ablation_scores, model, rep_tokens)
```

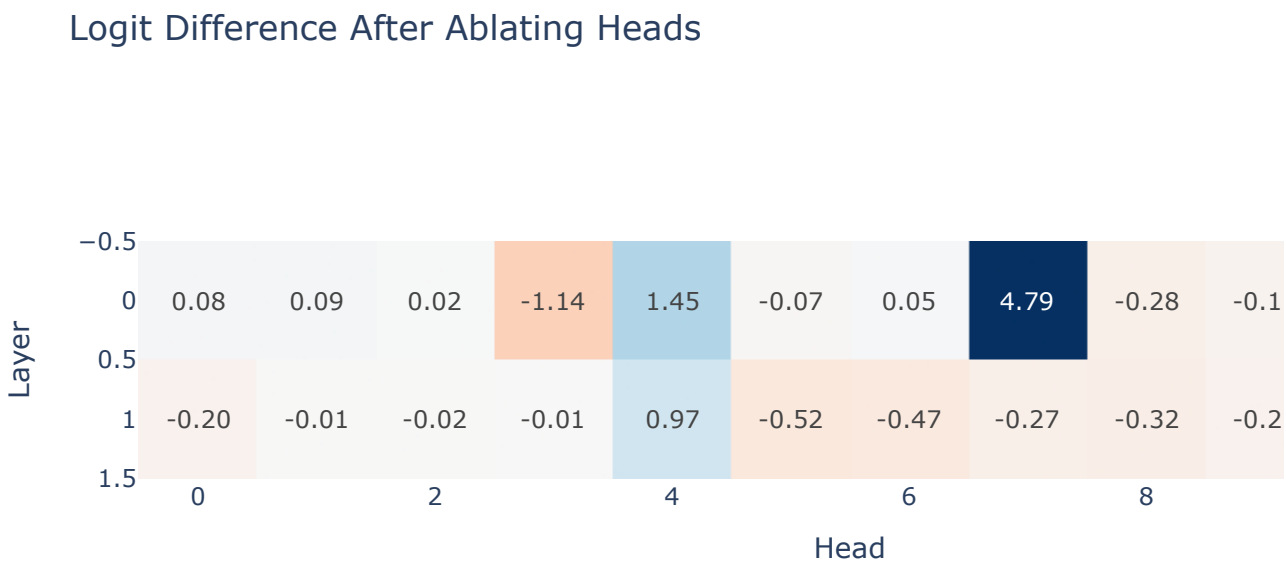
```
100%|██████████| 2/2 [00:00<00:00,  6.52it/s]
100%|██████████| 2/2 [00:00<00:00,  7.01it/s]All tests in `test_get_ablation_s
```

▶ Solution

Once you've passed the tests, you can plot the results:

```
imshow(
    ablation_scores,
```

```
    labels={"x": "Head", "y": "Layer", "color": "Logit diff"},
    title="Logit Difference After Ablating Heads",
    text_auto=".2f",
    width=900, height=400
)
```



What is your interpretation of these results?

▶ Interpretation

Bonus - different ablations

Try ablating *every* head apart from the previous token head and the two induction heads. What does this do to performance?

What if you mean ablate it, rather than zero ablating it?

*(In later sections e.g. Indirect Object Identification, we'll use hooks to implement some more advanced causal interventions, such as **activation patching** and **causal tracing**.)*

4️⃣ Reverse-engineering induction circuits

> Learning objectives
>
> - Understand the difference between investigating a circuit by looking at activtion patterns, and reverse-engineering a circuit by looking directly at the weights

> - Use the factored matrix class to inspect the QK and OV circuits within an induction circuit
> - Perform further exploration of induction circuits: composition scores, and targeted ablations

In previous exercises, we looked at the attention patterns and attributions of attention heads to try and identify which ones were important in the induction circuit. This might be a good way to get a feel for the circuit, but it's not a very rigorous way to understand it. It would be better described as **feature analysis**, where we observe *that* a particular head seems to be performing some task on a certain class of inputs, without identifying *why* it does so.

Now we're going to do some more rigorous mechanistic analysis - digging into the weights and using them to reverse engineer the induction head algorithm and verify that it is really doing what we think it is.

Refresher - the induction circuit

Before we get into the meat of this section, let's refresh the results we've gotten so far from investigating induction heads. We've found:

- When fed repeated sequences of tokens, heads `1.4` and `1.10` have the characteristic induction head attention pattern of a diagonal stripe with offset `seq_len - 1`.
    - We saw this both from the CircuitsVis results, and from the fact that these heads had high induction scores by our chosen metric (with all other heads having much lower scores).
- We also saw that head `0.7` strongly attends to the previous token in the sequence (even on non-repeated sequences).
- We performed **logit attribution** on the model, and found that the values written to the residual stream by heads `1.4` and `1.10` were both important for getting us correct predictions in the second half of the sequence.
- We performed **zero-ablation** on the model, and found that heads `0.7`, `1.4` and `1.10` all resulted in a large accuracy degradation on the repeated sequence task when they were ablated.

Based on all these observations, try and summarise the induction circuit and how it works, in your own words. You should try and link your explanation to the QK and OV circuits for particular heads, and describe what type (or types) of attention head composition are taking place.

You can use the dropdown below to check your understanding.

▶ My summary of the algorithm

Refresher - QK and OV circuits

Before we start, a brief terminology note. I'll refer to weight matrices for a particular layer and head using superscript notation, e.g. $W_Q^{1.4}$ is the query matrix for the 4th head in layer 1, and it has shape `[d_model, d_head]` (remember that we multiply with weight matrices on the right). Similarly, attention patterns will be denoted $A^{1.4}$ (remember that these are **activations**, not parameters, since they're given by the formula $A^h = xW_{QK}^h x^T$, where x is the residual stream (with shape `[seq_len, d_model]`).

As a shorthand, I'll often have A denote the one-hot encoding of token `A` (i.e. the vector with zeros everywhere except a one at the index of `A`), so $A^T W_E$ is the embedding vector for `A`.

Lastly, I'll refer to special matrix products as follows:

- $W_{OV}^h := W_V^h W_O^h$ is the **OV circuit** for head h, and $W_E W_{OV}^h W_U$ is the **full OV circuit**.
- $W_{QK}^h := W_Q^h (W_K^h)^T$ is the **QK circuit** for head h, and $W_E W_{QK}^h W_E^T$ is the **full QK circuit**.

Note that the order of these matrices are slightly different from the **Mathematical Frameworks** paper - this is a consequence of the way TransformerLens stores its weight matrices.

Question - what is the interpretation of each of the following matrices?

There are quite a lot of questions here, but they are conceptually important. If you're confused, you might want to read the answers to the first few questions and then try the later ones.

In your answers, you should describe the type of input it takes, and what the outputs represent.

W_{OV}^h

▶ Answer

$W_E W_{OV}^h W_U$

▶ Hint

▶ Answer

W_{QK}^h

▶ Answer

$W_E W_{QK}^h W_E^T$

▶ Answer

$W_{pos} W_{QK}^h W_{pos}^T$

▶ Answer

$W_E W_{OV}^{h_1} W_{QK}^{h_2} W_E^T$

where h_1 is in an earlier layer than h_2.

▶ Hint

▶ Answer

Before we start, there's a problem that we might run into when calculating all these matrices. Some of them are massive, and might not fit on our GPU. For instance, both full circuit matrices have shape $(d_{\text{vocab}}, d_{\text{vocab}})$, which in our case means $50278 \times 50278 \approx 2.5 \times 10^9$ elements. Even if your GPU can handle this, it still seems inefficient. Is there any way we can meaningfully analyse these matrices, without actually having to calculate them?

Factored Matrix class

In transformer interpretability, we often need to analyse low rank factorized matrices - a matrix $M = AB$, where M is `[large, large]`, but A is `[large, small]` and B is `[small, large]`. This is a common structure in transformers.

For instance, we can factorise the OV circuit above as $W_{OV}^h = W_V^h W_O^h$, where W_V^h has shape `[768, 64]` and W_O^h has shape `[64, 768]`. For an even more extreme example, the full OV circuit can be written as $(W_E W_V^h)(W_O^h W_U)$, where these two matrices have shape `[50278, 64]` and `[64, 50278]` respectively. Similarly, we can write the full QK circuit as $(W_E W_Q^h)(W_E W_K^h)^T$.

The `FactoredMatrix` class is a convenient way to work with these. It implements efficient algorithms for various operations on these, such as computing the trace, eigenvalues, Frobenius norm, singular value decomposition, and products with other matrices. It can (approximately) act as a drop-in replacement for the original matrix.

This is all possible because knowing the factorisation of a matrix gives us a much easier way of computing its important properties. Intuitively, since $M = AB$ is a very large matrix that operates on very small subspaces, we shouldn't expect knowing the actual values M_{ij} to be the most efficient way of storing it!

Exercise - deriving properties of a factored matrix

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

You shouldn't spend more than 10-25 minutes on this exercise.

If you're less interested in the maths, you can skip these exercises.
```

To give you an idea of what kinds of properties you can easily compute if you have a factored matrix, let's try and derive some ourselves.

Suppose we have $M = AB$, where A has shape (m, n), B has shape (n, m), and $m > n$. So M is a size-(m, m) matrix with rank at most n.

Question - how can you easily compute the trace of M?

▶ Answer

Question - how can you easily compute the eigenvalues of M?

(As you'll see in later exercises, eigenvalues are very important for evaluating matrices, for instance we can assess the copying scores of an OV circuit by looking at the eigenvalues of W_{OV}.)

▶ Hint

▶ Answer

Question (hard) - how can you easily compute the SVD of M?

▶ Hint

▶ Answer

If you're curious, you can go to the `FactoredMatrix` documentation to see the implementation of the SVD calculation, as well as other properties and operations.

Now that we've discussed some of the motivations behind having a `FactoredMatrix` class, let's see it in action.

Basic Examples

We can use the basic class directly - let's make a factored matrix directly and look at the basic operations:

```
A = t.randn(5, 2)
B = t.randn(2, 5)
AB = A @ B
AB_factor = FactoredMatrix(A, B)
print("Norms:")
print(AB.norm())
print(AB_factor.norm())

print(f"Right dimension: {AB_factor.rdim}, Left dimension: {AB_factor.ldim}, Hid
```

```
Norms:
tensor(4.2634)
tensor(4.2634)
Right dimension: 5, Left dimension: 5, Hidden dimension: 2
```

We can also look at the eigenvalues and singular values of the matrix. Note that, because the matrix is rank 2 but 5 by 5, the final 3 eigenvalues and singular values are zero - the factored class omits the zeros.

```
print("Eigenvalues:")
print(t.linalg.eig(AB).eigenvalues)
print(AB_factor.eigenvalues)
print()
print("Singular Values:")
```

```
print(t.linalg.svd(AB).S)
print(AB_factor.S)
print("Full SVD:")
print(AB_factor.svd())
```

```
Eigenvalues:
tensor([-2.3842e-07+0.j, -1.0227e-01+0.j, -2.6258e+00+0.j,  1.4203e-07+0.j,
        -2.9085e-08+0.j])
tensor([-2.6258+0.j, -0.1023+0.j])

Singular Values:
tensor([3.5544e+00, 2.3544e+00, 1.8405e-07, 6.4530e-08, 7.5758e-10])
tensor([3.5544, 2.3544])
Full SVD:
(tensor([[ 0.6489,  0.1496],
        [ 0.0352, -0.8238],
        [-0.1083,  0.3193],
        [-0.2428, -0.4036],
        [-0.7121,  0.1847]]), tensor([3.5544, 2.3544]), tensor([[ 0.3051,  0.5
        [-0.0195,  0.7605],
        [ 0.5901,  0.1734],
        [ 0.1802, -0.0910],
        [ 0.7252, -0.3208]]))
```

▶ Aside - the sizes of objects returned by the SVD method.

We can multiply a factored matrix with an unfactored matrix to get another factored matrix (as in example below). We can also multiply two factored matrices together to get another factored matrix.

```
C = t.randn(5, 300)
ABC = AB @ C
ABC_factor = AB_factor @ C
print("Unfactored:", ABC.shape, ABC.norm())
print("Factored:", ABC_factor.shape, ABC_factor.norm())
print(f"Right dimension: {ABC_factor.rdim}, Left dimension: {ABC_factor.ldim}, H
```

```
Unfactored: torch.Size([5, 300]) tensor(73.5401)
Factored: torch.Size([5, 300]) tensor(73.5401)
Right dimension: 300, Left dimension: 5, Hidden dimension: 2
```

If we want to collapse this back to an unfactored matrix, we can use the `AB` property to get the product:

```
AB_unfactored = AB_factor.AB
t.testing.assert_close(AB_unfactored, AB)
```

Reverse-engineering circuits

Within our induction circuit, we have four individual circuits: the OV and QK circuits in our previous token head, and the OV and QK circuits in our attention head. In the following sections

of the exercise, we'll reverse-engineer each of these circuits in turn.

- In the section **OV copying circuit**, we'll look at the layer-1 OV circuit.
- In the section **QK prev-token circuit**, we'll look at the layer-0 QK circuit.
- The third section (**K-composition**) is a bit trickier, because it involves looking at the composition of the layer-0 OV circuit **and** layer-1 QK circuit. We will have to do two things:
    1. Show that these two circuits are composing (i.e. that the output of the layer-0 OV circuit is the main determinant of the key vectors in the layer-1 QK circuit).
    2. Show that the joint operation of these two circuits is "make the second instance of a token attend to the token *following* an earlier instance.

The dropdown below contains a diagram explaining how the three sections relate to the different components of the induction circuit. You might have to open it in a new tab to see it clearly.

▶ Diagram

After this, we'll have a look at composition scores, which are a more mathematically justified way of showing that two attention heads are composing (without having to look at their behaviour on any particular class of inputs, since it is a property of the actual model weights).

[1] OV copying circuit

Let's start with an easy parts of the circuit - the copying OV circuit of `1.4` and `1.10`. Let's start with head 4. The only interpretable (read: **privileged basis**) things here are the input tokens and output logits, so we want to study the matrix:

$$W_E W_{OV}^{1.4} W_U$$

(and same for `1.10`). This is the $(d_{\text{vocab}}, d_{\text{vocab}})$-shape matrix that combines with the attention pattern to get us from input to output.

We want to calculate this matrix, and inspect it. We should find that its diagonal values are very high, and its non-diagonal values are much lower.

Question - why should we expect this observation? (you may find it helpful to refer back to the previous section, where you described what the interpretation of different matrices was.)

▶ Hint

Answer

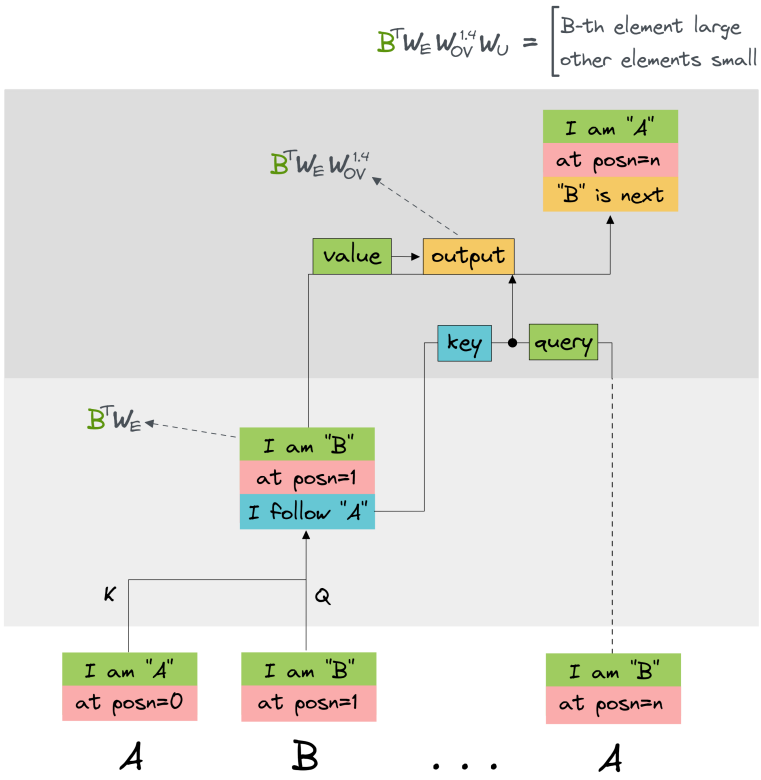
If our repeating sequence is `A B ... A B`, then:

$$B^T W_E W_{OV}^{1.4} W_U$$

is the **vector of logits which gets moved from the first `B` token to the second `A` token, to be used as the prediction for the token following the second `A` token**. It should result in a high prediction for `B`, and a low prediction for everything else. In other words, the `(B, X)`-th element of this matrix should be highest for `X=B`, which is exactly what we claimed.

If this still seems confusing, let's break it down bit by bit. We have:

- $B^T W_E$ is the token-embedding of `B`.
- $B^T W_E W_{OV}^{1.4}$ is the vector which gets moved from the first `B` token to the second `A` token, by our attention head (because the second `A` token attends strongly to the first `B`).
- $B^T W_E W_{OV}^{1.4} W_U$ is the vector of logits representing how this attention head affects the prediction of the token following the second `A`. There should be a higher logit for `B` than for any other token, because the attention head is trying to copy `B` to the second `A`.



Exercise - compute OV circuit for `1.4`

This is the first of several similar exercises where you calculate a circuit by multiplying matrices. This exercise is pretty important (in particular, you should make sure you understand what this matrix represents and why we're interested in it), but the actual calculation shouldn't take very long.

You should compute it as a `FactoredMatrix` object.

Remember, you can access the model's weights directly e.g. using `model.W_E` or `model.W_Q` (the latter gives you all the `W_Q` matrices, indexed by layer and head).

```
head_index = 4
layer = 1

W_O = model.W_O[layer, head_index]
W_V = model.W_V[layer, head_index]
W_E = model.W_E
W_U = model.W_U
```

```
 8
 9 OV_circuit = FactoredMatrix(W_V, W_O)
10 full_OV_circuit = W_E @ OV_circuit @ W_U
11
12 tests.test_full_OV_circuit(full_OV_circuit, model, layer, head_index)
```

```
    All tests in `test_full_OV_circuit` passed!
```

▶ Help - I'm not sure how to use this class to compute a product of more than 2 matrices.

▶ Solution

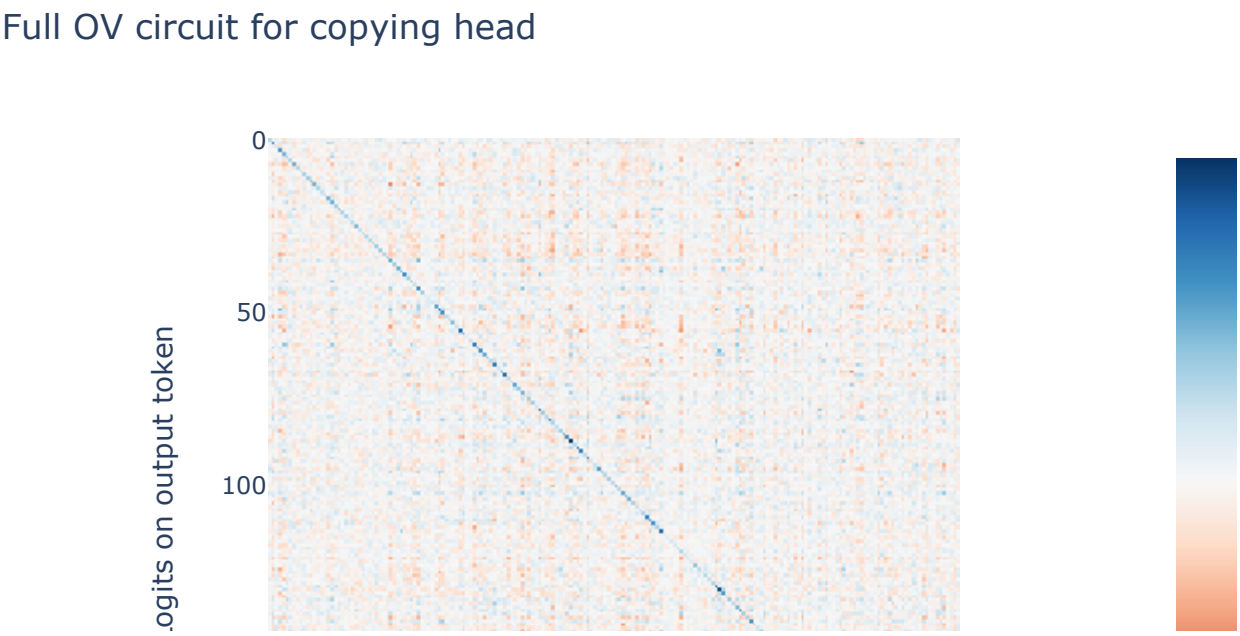
Exercise - verify this is the identity

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

You shouldn't spend more than 5-10 minutes on this exercise.

This exercise should be very short; it only requires 2 lines of code. Understanding it cor
```

Now we want to check that this matrix is the identity. Since it's in factored matrix form, this is a bit tricky, but there are still things we can do.

First, to validate that it looks diagonal-ish, let's pick 200 random rows and columns and visualise that - it should at least look identity-ish here!

```
 1 # YOUR CODE HERE - get a random sample from the full OV circuit, so it can be pl
 2 indices = t.randint(0, model.cfg.d_vocab, (200,))
 3 full_OV_circuit_sample = full_OV_circuit[indices, indices].AB
 4
 5
 6 imshow(
 7     full_OV_circuit_sample,
 8     labels={"x": "Input token", "y": "Logits on output token"},
 9     title="Full OV circuit for copying head",
10     width=700,
11 )
```



▶ Aside - indexing factored matrices

▶ Solution

You should observe a pretty distinct diagonal pattern here, which is a good sign. However, the matrix is pretty noisy so it probably won't be exactly the identity. Instead, we should come up with a summary statistic to capture a rough sense of "closeness to the identity".

Accuracy is a good summary statistic - what fraction of the time is the largest logit in a row on the diagonal? Even if there's lots of noise, you'd probably still expect the largest logit to be on the diagonal a good deal of the time.

If you're on a Colab or have a powerful GPU, you should be able to compute the full matrix and perform this test. If not, the most efficient method is to iterate through the rows (or batches of rows). Remember - if `M = AB` is a factored matrix, then the ith row of `M` is `A[i, :] @ B`.

Bonus exercise: Top-5 accuracy is also a good metric (use `t.topk`, take the indices output).

```
def top_1_acc(full_OV_circuit: FactoredMatrix) -> float:
    '''
    This should take the argmax of each column (ie over dim=0) and return the fr
    '''
    # SOLUTION
    AB = full_OV_circuit.AB

    return (t.argmax(AB, dim=1) == t.arange(AB.shape[0]).to(device)).float().mea


print(f"Fraction of the time that the best logit is on the diagonal: {top_1_acc(
```

```
Fraction of the time that the best logit is on the diagonal: 0.3079
```

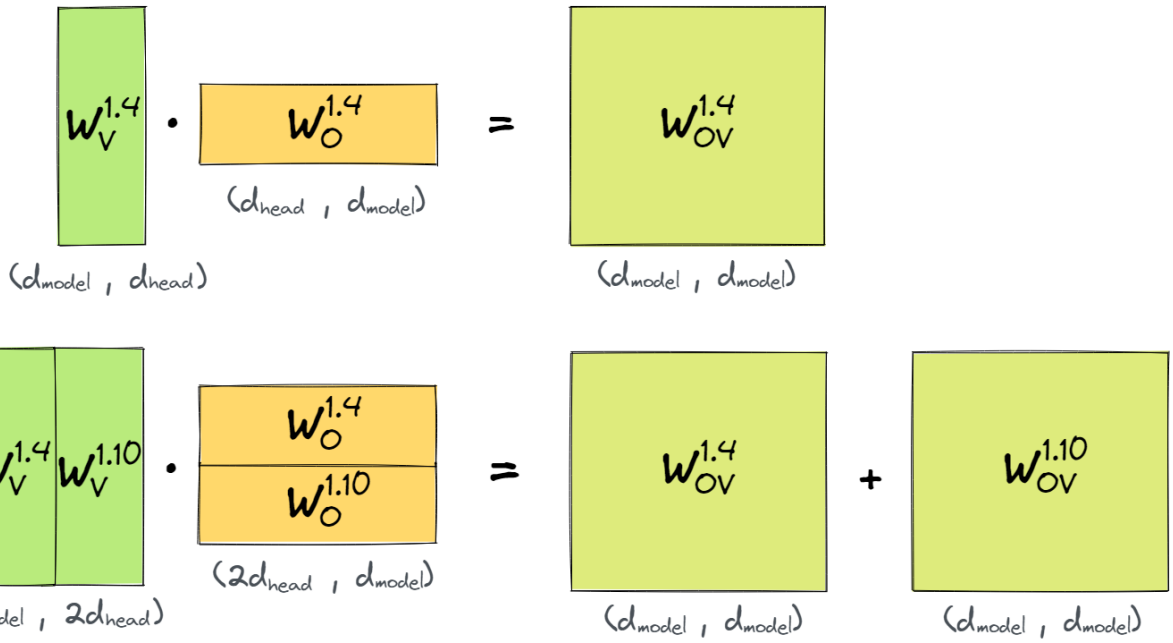
▶ Solution

This should return about 30.79% - pretty underwhelming. It goes up to 47.73% for top-5. What's up with that?

Exercise - compute effective circuit

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

You shouldn't spend more than 5-10 minutes on this exercise.

This exercise should be very short; it only requires 2 lines of code. Understanding it cor
```

Now we return to why we have *two* induction heads. If both have the same attention pattern, the effective OV circuit is actually $W_E(W_V^{1.4}W_O^{1.4} + W_V^{1.10}W_O^{1.10})W_U$, and this is what matters. So let's re-run our analysis on this!



▶ Question - why might the model want to split the circuit across two heads?

```
1 # YOUR CODE HERE - compute the effective OV circuit, and run `top_1_acc` on it
```

▶ Solution (and expected output)

[2] QK prev-token circuit

The other easy circuit is the QK-circuit of L0H7 - how does it know to be a previous token circuit?

We can multiply out the full QK circuit via the positional embeddings:

$$W_{\text{pos}} W_Q^{0.7} (W_K^{0.7})^T W_{\text{pos}}^T$$

to get a matrix `pos_by_pos` of shape `[max_ctx, max_ctx]` (max ctx = max context length, i.e. maximum length of a sequence we're allowing, which is set by our choice of dimensions in W_{pos}).

Note that in this case, our max context window is 2048 (we can check this via `model.cfg.n_ctx`). This is much smaller than the 50k-size matrices we were working with in the previous section, so we shouldn't need to use the factored matrix class here.

Once we calculate it, we can then mask it and apply a softmax, and should get a clear stripe on the lower diagonal (Tip: Click and drag to zoom in, hover over cells to see their values and indices!)

▶ Question - why should we expect this matrix to have a lower-diagonal stripe?

Why is it justified to ignore token encodings? In this case, it turns out that the positional encodings have a much larger effect on the attention scores than the token encodings. If you want, you can verify this for yourself - after going through the next section (reverse-engineering K-composition), you'll have a better sense of how to perform attribution on the inputs to attention heads, and assess their importance).

Exercise - compute full QK-circuit for `0.7`

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

You shouldn't spend more than 10-15 minutes on this exercise.

The fiddly masking and scaling steps make this exercise a bit harder than the last one. Lc
```

This is another relatively simple matrix multiplication, although it's a bit fiddly on account of the masking and scaling steps. If you understand what you're being asked to do but still not passing the tests, you should probably look at the solution.

Now, you should compute and plot the matrix.

Remember, you're calculating the attention pattern (i.e. probabilites) not the scores. You'll need to mask the scores (you can use the `mask_scores` function we've provided you with), and scale them.

```
1 def mask_scores(attn_scores: Float[Tensor, "query_nctx key_nctx"]):
2     '''Mask the attention scores so that tokens don't attend to previous tokens.
3     assert attn_scores.shape == (model.cfg.n_ctx, model.cfg.n_ctx)
```

```
    mask = t.tril(t.ones_like(attn_scores)).bool()
    neg_inf = t.tensor(-1.0e6).to(attn_scores.device)
    masked_attn_scores = t.where(mask, attn_scores, neg_inf)
    return masked_attn_scores


# YOUR CODE HERE - calculate the matrix `pos_by_pos_pattern` as described above
layer = 0
head_index = 7

W_pos = model.W_pos
W_QK = model.W_Q[layer, head_index] @ model.W_K[layer, head_index].T
pos_by_pos_scores = W_pos @ W_QK @ W_pos.T
masked_scaled = mask_scores(pos_by_pos_scores / model.cfg.d_head ** 0.5)
pos_by_pos_pattern = t.softmax(masked_scaled, dim=-1)


tests.test_pos_by_pos_pattern(pos_by_pos_pattern, model, layer, head_index)
```

```
All tests in `test_full_OV_circuit` passed!
```

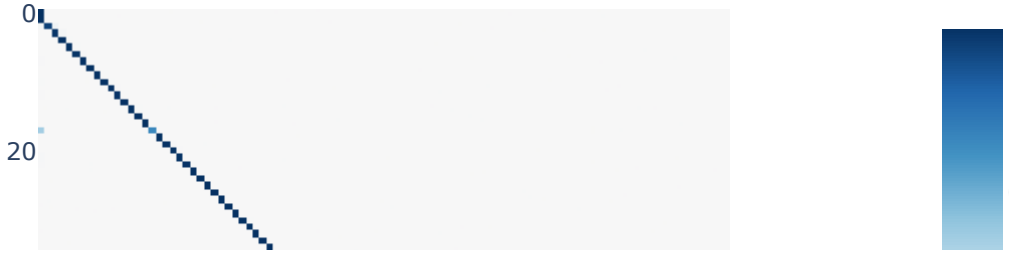
▶ Solution

Once the tests pass, you can plot a corner of your matrix:

```
print(f"Avg lower-diagonal value: {pos_by_pos_pattern.diag(-1).mean():.4f}")

imshow(
    utils.to_numpy(pos_by_pos_pattern[:100, :100]),
    labels={"x": "Key", "y": "Query"},
    title="Attention patterns for prev-token QK circuit, first 100 indices",
    width=700
)
```

```
Avg lower-diagonal value: 0.9978
```

Attention patterns for prev-token QK circuit, first 100 indices



[3] K-composition circuit

We now dig into the hard part of the circuit - demonstrating the K-Composition between the previous token head and the induction head.

Splitting activations

We can repeat the trick from the logit attribution scores. The QK-input for layer 1 is the sum of 14 terms (2+n_heads) - the token embedding, the positional embedding, and the results of each layer 0 head. So for each head H in layer 1, the query tensor (ditto key) corresponding to sequence position i is:

$$xW_Q^{1.\mathrm{H}} = (e + pe + \sum_{h=0}^{11} x^{0.\mathrm{h}})W_Q^{1.\mathrm{H}}$$

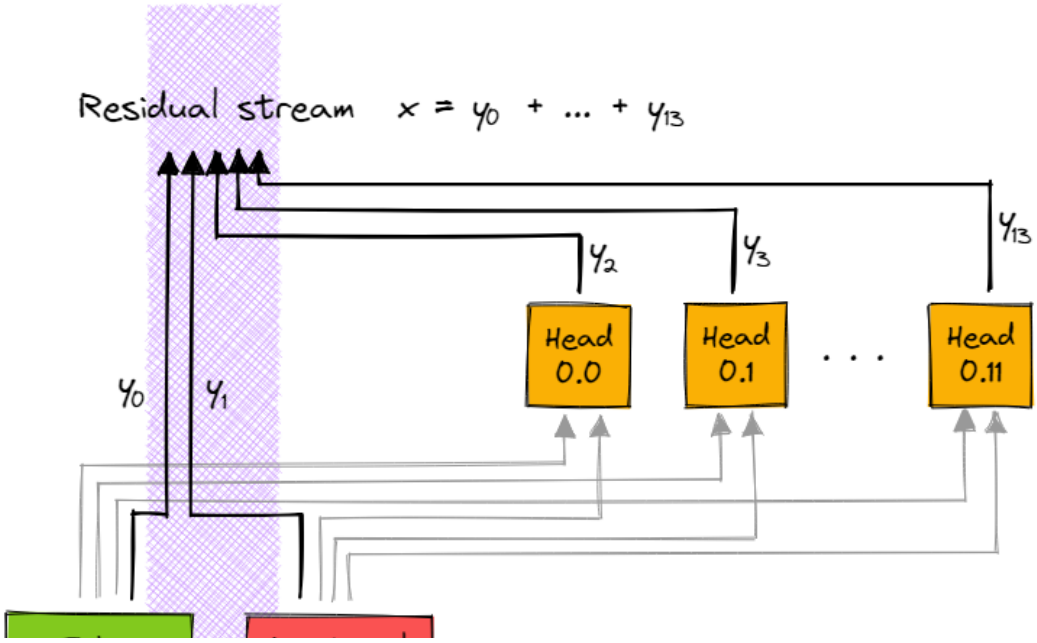
$$= eW_Q^{1.\mathrm{H}} + peW_Q^{1.\mathrm{H}} + \sum_{h=0}^{11} x^{0.\mathrm{h}}W_Q^{1.\mathrm{H}}$$

where e stands for the token embedding, pe for the positional embedding, and $x^{0.\mathrm{h}}$ for the output of head h in layer 0 (and the sum of these tensors equals the residual stream x). All these tensors have shape `[seq, d_model]`. So we can treat the expression above as a sum of matrix multiplications `[seq, d_model] @ [d_model, d_head] -> [seq, d_head]`.

For ease of notation, I'll refer to the 14 inputs as $(y_0, y_1, \ldots, y_{13})$ rather than $(e, pe, x^{0.\mathrm{h}}, \ldots, x^{11.h})$. So we have:

$$xW_Q^h = \sum_{i=0}^{13} y_i W_Q^h$$

with each y_i having shape `[seq, d_model]`, and the sum of y_is being the full residual stream x. Here is a diagram to illustrate:



Exercise - analyse the relative importance

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

You shouldn't spend more than 15-25 minutes on these exercises.

Most of these functions just involve indexing and einsums, but conceptual understanding /
```

We can now analyse the relative importance of these 14 terms! A very crude measure is to take the norm of each term (by component and position).

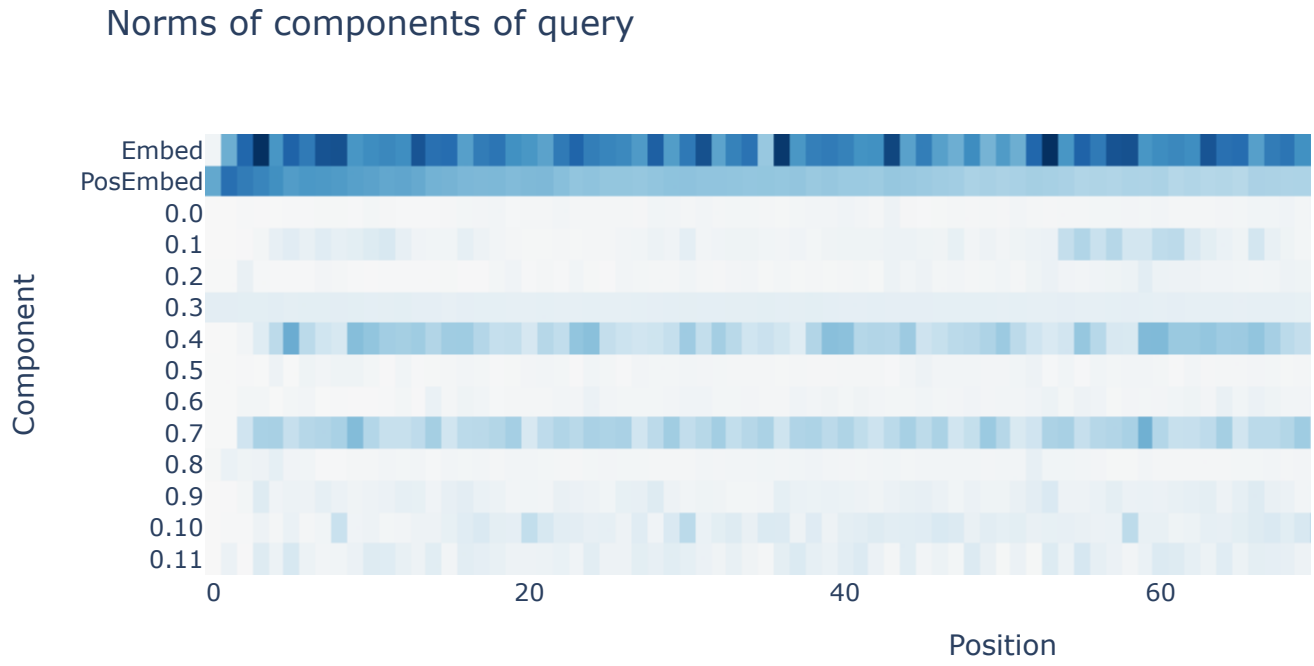
Note that this is a pretty dodgy metric - q and k are not inherently interpretable! But it can be a good and easy-to-compute proxy.

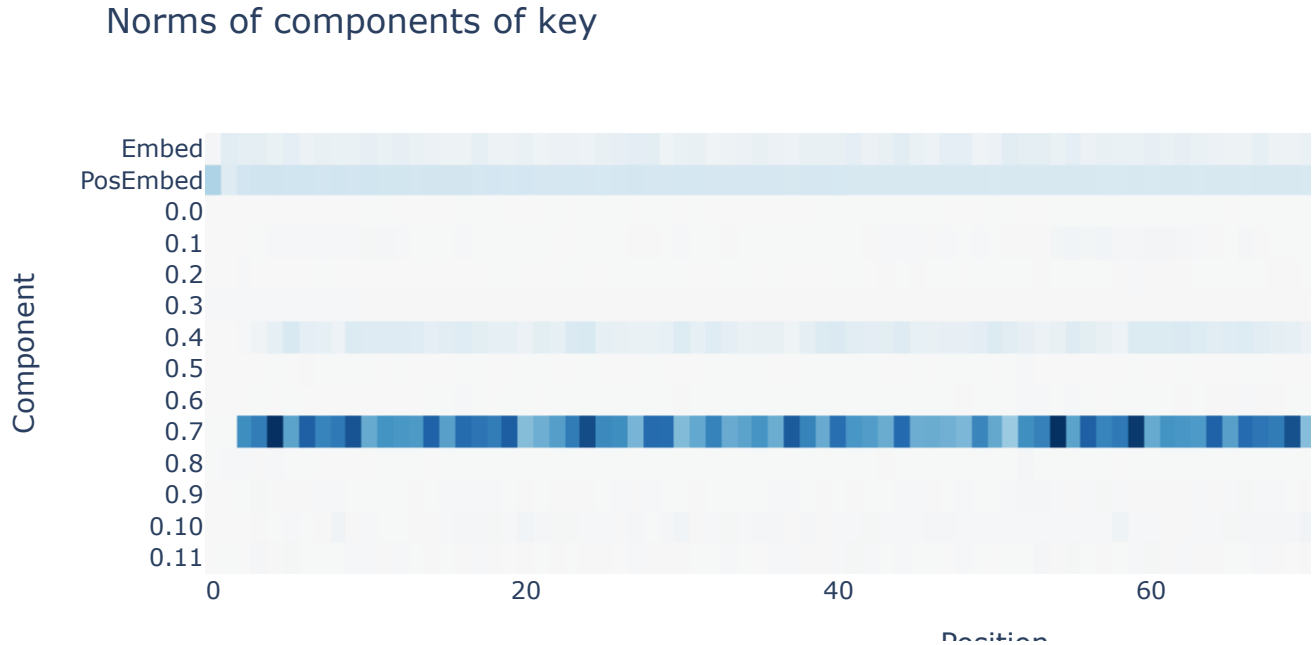
▶ Question - why are Q and K not inherently interpretable? Why might the norm be a good metric in spite of this?

Fill in the functions below:

```python
def decompose_qk_input(cache: ActivationCache) -> t.Tensor:
    '''
    Output is decomposed_qk_input, with shape [2+num_heads, seq, d_model]

    The [i, 0, 0]th element is y_i (from notation above)
    '''
    # SOLUTION
    y0 = cache["embed"].unsqueeze(0) # shape (1, seq, d_model)
    y1 = cache["pos_embed"].unsqueeze(0) # shape (1, seq, d_model)
    y_rest = cache["result", 0].transpose(0, 1) # shape (12, seq, d_model)

    return t.concat([y0, y1, y_rest], dim=0)
```

```
13
14
15 def decompose_q(decomposed_qk_input: t.Tensor, ind_head_index: int) -> t.Tensor:
16     '''
17     Output is decomposed_q with shape [2+num_heads, position, d_head]
18
19     The [i, 0, 0]th element is y_i @ W_Q (so the sum along axis 0 is just the q-
20     '''
21     # SOLUTION
22     W_Q = model.W_Q[1, ind_head_index]
23
24     return einops.einsum(
25         decomposed_qk_input, W_Q,
26         "n seq d_head, d_head d_model -> n seq d_model"
27     )
28
29
30 def decompose_k(decomposed_qk_input: t.Tensor, ind_head_index: int) -> t.Tensor:
31     '''
32     Output is decomposed_k with shape [2+num_heads, position, d_head]
33
34     The [i, 0, 0]th element is y_i @ W_K(so the sum along axis 0 is just the k-v
35     '''
36     # SOLUTION
37     W_K = model.W_K[1, ind_head_index]
38
39     return einops.einsum(
40         decomposed_qk_input, W_K,
41         "n seq d_head, d_head d_model -> n seq d_model"
42     )
43
44
45 ind_head_index = 4
46 # First we get decomposed q and k input, and check they're what we expect
47 decomposed_qk_input = decompose_qk_input(rep_cache)
48 decomposed_q = decompose_q(decomposed_qk_input, ind_head_index)
49 decomposed_k = decompose_k(decomposed_qk_input, ind_head_index)
50 t.testing.assert_close(decomposed_qk_input.sum(0), rep_cache["resid_pre", 1] + r
51 t.testing.assert_close(decomposed_q.sum(0), rep_cache["q", 1][:, ind_head_index]
52 t.testing.assert_close(decomposed_k.sum(0), rep_cache["k", 1][:, ind_head_index]
53 # Second, we plot our results
54 component_labels = ["Embed", "PosEmbed"] + [f"0.{h}" for h in range(model.cfg.n_
55 for decomposed_input, name in [(decomposed_q, "query"), (decomposed_k, "key")]:
56     imshow(
57         utils.to_numpy(decomposed_input.pow(2).sum([-1])),
58         labels={"x": "Position", "y": "Component"},
59         title=f"Norms of components of {name}",
60         y=component_labels,
61         width=1000, height=400
62     )
```





▶ Solution

▶ A technical note on the positional embeddings - optional, feel free to skip this.

This tells us which heads are probably important, but we can do better than that. Rather than looking at the query and key components separately, we can see how they combine together - i.e. take the decomposed attention scores.

This is a bilinear function of q and k, and so we will end up with a `decomposed_scores` tensor with shape `[query_component, key_component, query_pos, key_pos]`, where summing along BOTH of the first axes will give us the original attention scores (pre-mask).

Exercise - decompose attention scores

Difficulty: 🟠🟠⚪⚪⚪
Importance: 🟠🟠🟠🟠⚪

You shouldn't spend more than 5-10 minutes on this exercise.

Having already done the previous exercises, this one should be easier.

Implement the function giving the decomposed scores (remember to scale by `sqrt(d_head)` !) For now, don't mask it.

▶ Question - why do I focus on the attention scores, not the attention pattern? (i.e. pre softmax not post softmax)

▶ Help - I'm confused about what we're doing / why we're doing it.

```
def decompose_attn_scores(decomposed_q: t.Tensor, decomposed_k: t.Tensor) -> t.T
    '''
    Output is decomposed_scores with shape [query_component, key_component, quer

    The [i, j, 0, 0]th element is y_i @ W_QK @ y_j^T (so the sum along both firs
    '''
    # SOLUTION
    return einops.einsum(
        decomposed_q, decomposed_k,
        "q_comp q_pos d_model, k_comp k_pos d_model -> q_comp k_comp q_pos k_pos
    )

tests.test_decompose_attn_scores(decompose_attn_scores, decomposed_q, decomposed
```

```
All tests in `test_decompose_attn_scores` passed!
```

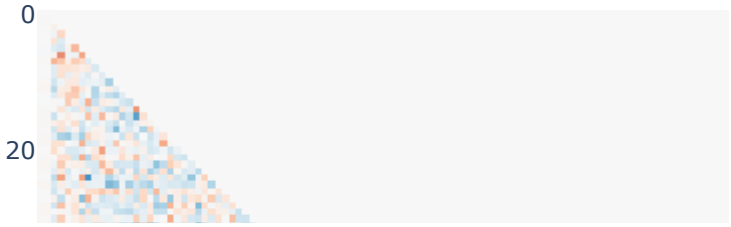
▶ Solution

Once these tests have passed, you can plot the results:

```
decomposed_scores = decompose_attn_scores(decomposed_q, decomposed_k)
decomposed_stds = einops.reduce(
    decomposed_scores,
    "query_decomp key_decomp query_pos key_pos -> query_decomp key_decomp",
    t.std
)

# First plot: attention score contribution from (query_component, key_component)
imshow(
    utils.to_numpy(t.tril(decomposed_scores[0, 9])),
    title="Attention score contributions from (query, key) = (embed, output of L
    width=800
)

# Second plot: std dev over query and key positions, shown by component
```

```
imshow(
    utils.to_numpy(decomposed_stds),
    labels={"x": "Key Component", "y": "Query Component"},
    title="Standard deviations of attention score contributions (by key and quer
    x=component_labels,
    y=component_labels,
    width=800
)
```

Attention score contributions from (query, key) = (embed, output of L0



▶ Question - what is the interpretation of these plots?

Interpreting the full circuit

Now we know that head `1.4` is composing with head `0.7` via K composition, we can multiply through to create a full circuit:

$$W_E\, W_{QK}^{1.4}\, (W_{OV}^{0.7})^T\, W_E^T$$

and verify that it's the identity. (Note, when we say identity here, we're again thinking about it as a distribution over logits, so this should be taken to mean "high diagonal values", and we'll be using our previous metric of `top_1_acc`.)

Question - why should this be the identity?

▼ Answer

This matrix is a bilinear form. Its diagonal elements (A, A) are:

$$A^T\, W_E\, W_{QK}^{1.4}\, W_{OV}^{0.7}\, W_E^T\, A = \underbrace{(A^T W_E W_Q^{1.4})}_{\text{query}}\underbrace{(A^T W_E W_{OV}^{0.7} W_K^{1.4})^T}_{\text{key}}$$

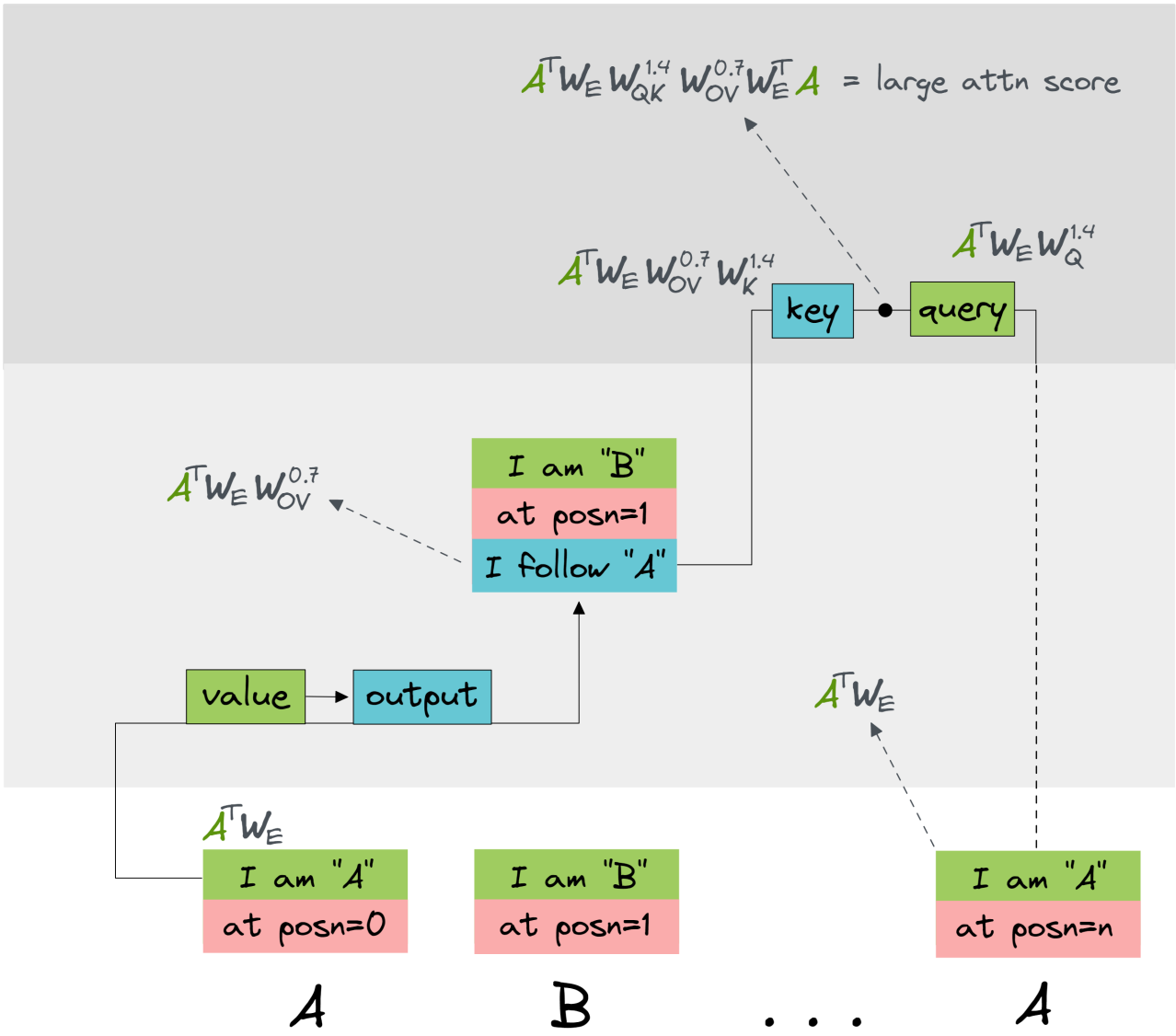
Intuitively, the query is saying "**I'm looking for a token which followed** A", and the key is saying "**I *am* a token which folllowed** A" (recall that $A^T W_E W_{OV}^{0.7}$ is the vector which gets moved one position forward by our prev token head `0.7`).

Now, consider the off-diagonal elements (A, X) (for $X \neq A$). We expect these to be small, because the key doesn't match the query:

$$A^T\, W_E\, W_{QK}^{1.4}\, W_{OV}^{0.7}\, W_E^T\, X = \underbrace{(\text{I'm looking for a token which followed A})}_{\text{query}} \cdot \underbrace{(\text{I am a token}}$$

Hence, we expect this to be the identity.

An illustration:



Exercise - compute the K-comp circuit

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

You shouldn't spend more than 10-20 minutes on this exercise.
```

Calculate the matrix above, as a `FactoredMatrix` object.

▶ Aside about multiplying FactoredMatrix objects together.

```
def find_K_comp_full_circuit(
    model: HookedTransformer,
    prev_token_head_index: int,
    ind_head_index: int
) -> FactoredMatrix:
    '''
    Returns a (vocab, vocab)-size FactoredMatrix, with the first dimension being
    '''
```

```
    # SOLUTION
    W_E = model.W_E
    W_Q = model.W_Q[1, ind_head_index]
    W_K = model.W_K[1, ind_head_index]
    W_O = model.W_O[0, prev_token_head_index]
    W_V = model.W_V[0, prev_token_head_index]

    Q = W_E @ W_Q
    K = W_E @ W_V @ W_O @ W_K
    return FactoredMatrix(Q, K.T)

prev_token_head_index = 7
ind_head_index = 4
K_comp_circuit = find_K_comp_full_circuit(model, prev_token_head_index, ind_head

tests.test_find_K_comp_full_circuit(find_K_comp_full_circuit, model)

print(f"Fraction of tokens where the highest activating key is the same token: {
```

```
All tests in `test_find_K_comp_full_circuit` passed!
Fraction of tokens where the highest activating key is the same token: 0.5201
```

▶ Solution

You can also try this out for `ind_head_index = 10`. Do you get a better result?

Note - unlike last time, it doesn't make sense to consider the "effective circuit" formed by adding together the weight matrices for heads `1.4` and `1.10`. Why not?

▶ Answer

Further Exploration of Induction Circuits

I now consider us to have fully reverse engineered an induction circuit - by both interpreting the features and by reverse engineering the circuit from the weights. But there's a bunch more ideas that we can apply for finding circuits in networks that are fun to practice on induction heads, so here's some bonus content - feel free to skip to the later bonus ideas though.

Composition scores

A particularly cool idea in the paper is the idea of virtual weights, or compositional scores. (Though I came up with it, so I'm deeply biased!). This is used to identify induction heads.

The key idea of compositional scores is that the residual stream is a large space, and each head is reading and writing from small subspaces. By default, any two heads will have little overlap between their subspaces (in the same way that any two random vectors have almost zero dot product in a large vector space). But if two heads are deliberately composing, then they will likely want to ensure they write and read from similar subspaces, so that minimal information is lost. As a result, we can just directly look at "how much overlap there is" between the output space of the earlier head and the K, Q, or V input space of the later head.

We represent the **output space** with $W_{OV} = W_V W_O$. Call matrices like this W_A.

We represent the **input space** with $W_{QK} = W_Q W_K^T$ (for Q-composition), $W_{QK}^T = W_K W_Q^T$ (for K-Composition) or $W_{OV} = W_V W_O$ (for V-Composition, of the later head). Call matrices like these W_B (we've used this notation so that W_B refers to a later head, and W_A to an earlier head).

▶ Help - I don't understand what motivates these definitions.

How do we formalise overlap? This is basically an open question, but a surprisingly good metric is $\frac{\|W_A W_B\|_F}{\|W_B\|_F \|W_A\|_F}$ where $\|W\|_F = \sum_{i,j} W_{i,j}^2$ is the Frobenius norm, the sum of squared elements. (If you're dying of curiosity as to what makes this a good metric, you can jump to the section immediately after the exercises below.)

Exercise - calculate composition scores

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

You shouldn't spend more than 15-25 minutes on these exercises.

Writing a composition score function should be easy. The slightly harder part is getting t
```

Let's calculate this metric for all pairs of heads in layer 0 and layer 1 for each of K, Q and V composition and plot it.

We'll start by implementing this using plain old tensors (later on we'll see how this can be sped up using the `FactoredMatrix` class). We also won't worry about batching our calculations yet; we'll just do one matrix at a time.

We've given you tensors `q_comp_scores` etc. to hold the composition scores for each of Q, K and V composition (i.e. the `[i, j]` th element of `q_comp_scores` is the Q-composition score between the output from the `i` th head in layer 0 and the input to the `j` th head in layer 1). You should complete the function `get_comp_score`, and then fill in each of these tensors.

```
def get_comp_score(
    W_A: Float[Tensor, "in_A out_A"],
    W_B: Float[Tensor, "out_A out_B"]
) -> float:
    '''
    Return the composition score between W_A and W_B.
    '''
    # SOLUTION
    W_A_norm = W_A.pow(2).sum().sqrt()
    W_B_norm = W_B.pow(2).sum().sqrt()
    W_AB_norm = (W_A @ W_B).pow(2).sum().sqrt()

```

```
    return (W_AB_norm / (W_A_norm * W_B_norm)).item()


tests.test_get_comp_score(get_comp_score)
```

Once you've passed the tests, you can fill in all the composition scores. Here you should just use a for loop, iterating over all possible pairs of `W_A` in layer 0 and `W_B` in layer 1, for each type of composition. Later on, we'll look at ways to batch this computation.

```
# Get all QK and OV matrices
W_QK = model.W_Q @ model.W_K.transpose(-1, -2)
W_OV = model.W_V @ model.W_O

# Define tensors to hold the composition scores
composition_scores = {
    "Q": t.zeros(model.cfg.n_heads, model.cfg.n_heads).to(device),
    "K": t.zeros(model.cfg.n_heads, model.cfg.n_heads).to(device),
    "V": t.zeros(model.cfg.n_heads, model.cfg.n_heads).to(device),
}


# YOUR CODE HERE - fill in each tensor in the dictionary, by looping over W_A an
for i in tqdm(range(model.cfg.n_heads)):
    for j in range(model.cfg.n_heads):
        composition_scores["Q"][i, j] = get_comp_score(W_OV[0, i], W_QK[1, j])
        composition_scores["K"][i, j] = get_comp_score(W_OV[0, i], W_QK[1, j].T)
        composition_scores["V"][i, j] = get_comp_score(W_OV[0, i], W_OV[1, j])


# Plot the composition scores
for comp_type in "QKV":
    plot_comp_scores(model, composition_scores[comp_type], f"{comp_type} Composi
```

```
100%|██████████| 12/12 [00:00<00:00, 54.08it/s]
```

▶ Solution

Exercise - Setting a Baseline

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

You shouldn't spend more than ~10 minutes on this exercise.
```

To interpret the above graphs we need a baseline! A good one is what the scores look like at initialisation. Make a function that randomly generates a composition score 200 times and tries

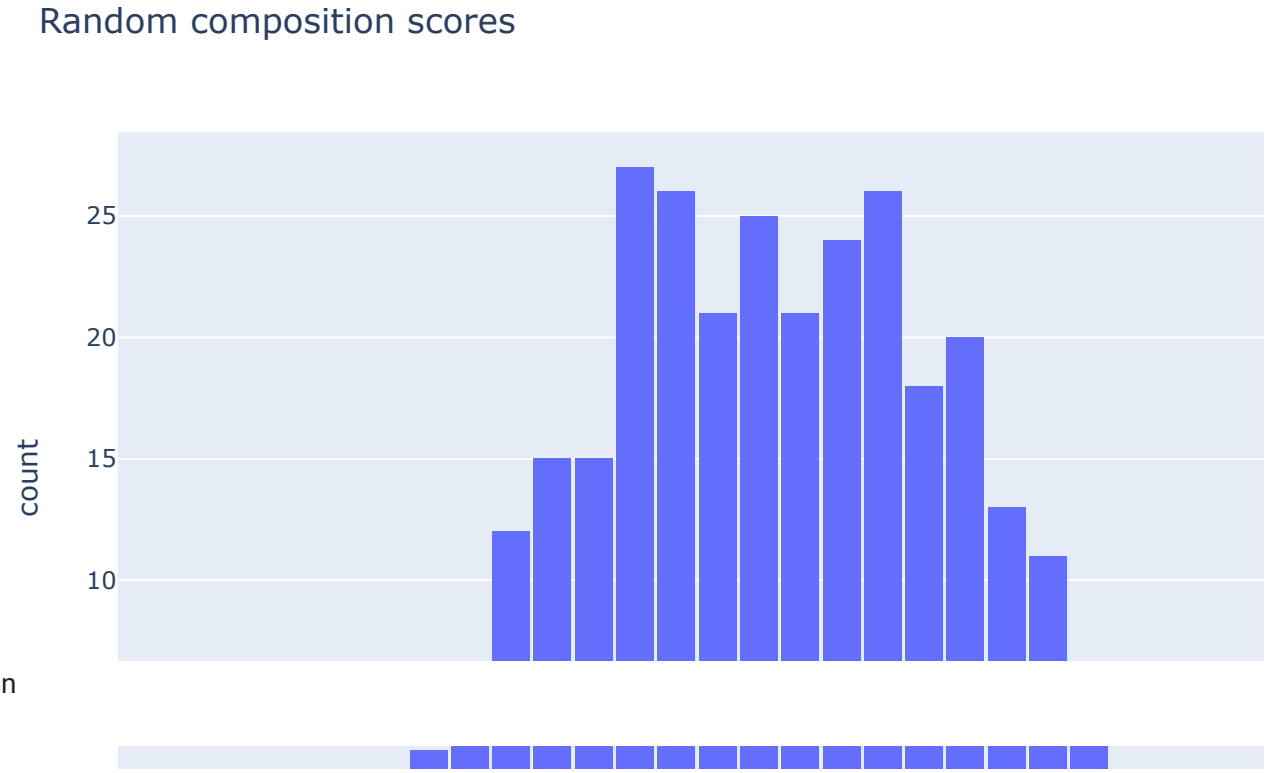
this. Remember to generate 4 `[d_head, d_model]` matrices, not 2 `[d_model, d_model]` matrices! This model was initialised with **Kaiming Uniform Initialisation**:

```
W = t.empty(shape)
nn.init.kaiming_uniform_(W, a=np.sqrt(5))
```

(Ideally we'd do a more efficient generation involving batching, and more samples, but we won't worry about that yet.)

```
def generate_single_random_comp_score() -> float:
    '''
    Write a function which generates a single composition score for random matri
    '''
    # SOLUTION
    W_A_left = t.empty(model.cfg.d_model, model.cfg.d_head)
    W_B_left = t.empty(model.cfg.d_model, model.cfg.d_head)
    W_A_right = t.empty(model.cfg.d_model, model.cfg.d_head)
    W_B_right = t.empty(model.cfg.d_model, model.cfg.d_head)

    for W in [W_A_left, W_B_left, W_A_right, W_B_right]:
        nn.init.kaiming_uniform_(W, a=np.sqrt(5))

    W_A = W_A_left @ W_A_right.T
    W_B = W_B_left @ W_B_right.T

    return get_comp_score(W_A, W_B)


n_samples = 300
comp_scores_baseline = np.zeros(n_samples)
for i in tqdm(range(n_samples)):
    comp_scores_baseline[i] = generate_single_random_comp_score()
print("\nMean:", comp_scores_baseline.mean())
print("Std:", comp_scores_baseline.std())
hist(
    comp_scores_baseline,
    nbins=50,
    width=800,
    labels={"x": "Composition score"},
    title="Random composition scores"
)
```

```
100%|██████████| 300/300 [00:04<00:00, 61.36it/s]
Mean: 0.036093534318109355
Std: 0.00044222332379797975
```



▶ Solution

We can re-plot our above graphs with this baseline set to white. Look for interesting things in this graph!

```
baseline = comp_scores_baseline.mean()
for comp_type, comp_scores in composition_scores.items():
    plot_comp_scores(model, comp_scores, f"{comp_type} Composition Scores", base
```

▶ Some interesting things to observe:

Theory + Efficient Implementation

So, what's up with that metric? The key is a cute linear algebra result that the squared Frobenius norm is equal to the sum of the squared singular values.

▶ Proof

So if $W_A = U_A S_A V_A^T$, $W_B = U_B S_B V_B^T$, then $\|W_A\|_F = \|S_A\|_F$, $\|W_B\|_F = \|S_B\|_F$ and $\|W_A W_B\|_F = \|S_A V_A^T U_B S_B\|_F$. In some sense, $V_A^T U_B$ represents how aligned the subspaces written to and read from are, and the S_A and S_B terms weights by the importance of those subspaces.

▶ Click here, if this explanation still seems confusing.

Exercise - batching, and using the `FactoredMatrix` class

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

This exercise is optional, and not a vitally important conceptual part  of this section. I
```

We can also use this insight to write a more efficient way to calculate composition scores - this is extremely useful if you want to do this analysis at scale! The key is that we know that our matrices have a low rank factorisation, and it's much cheaper to calculate the SVD of a narrow matrix than one that's large in both dimensions. See the [algorithm described at the end of the paper] (search for SVD).

So we can work with the `FactoredMatrix` class. This also provides the method `.norm()` which returns the Frobenium norm. This is also a good opportunity to bring back baching - this will sometimes be useful in our analysis. In the function below, `W_As` and `W_Bs` are both >2D factored matrices (e.g. they might represent the OV circuits for all heads in a particular layer, or across multiple layers), and the function's output should be a tensor of composition scores for each pair of matrices `(W_A, W_B)` in the >2D tensors `(W_As, W_Bs)`.

```
def get_batched_comp_scores(
    W_As: FactoredMatrix,
    W_Bs: FactoredMatrix
) -> t.Tensor:
    '''Computes the compositional scores from indexed factored matrices W_As and

    Each of W_As and W_Bs is a FactoredMatrix object which is indexed by all but
        W_As.shape == (*A_idx, A_in, A_out)
        W_Bs.shape == (*B_idx, B_in, B_out)
        A_out == B_in

    Return: tensor of shape (*A_idx, *B_idx) where the [*a_idx, *b_idx]th elemen
    '''
    # SOLUTION
    W_As = FactoredMatrix(
        W_As.A.reshape(-1, 1, *W_As.A.shape[-2:]),
        W_As.B.reshape(-1, 1, *W_As.B.shape[-2:]),
    )
    W_Bs = FactoredMatrix(
        W_Bs.A.reshape(1, -1, *W_Bs.A.shape[-2:]),
        W_Bs.B.reshape(1, -1, *W_Bs.B.shape[-2:]),
    )

    # Compute the product
    W_ABs = W_As @ W_Bs

    # Compute the norms, and return the metric
```

```
    return W_ABs.norm() / (W_As.norm() * W_Bs.norm())

W_QK = FactoredMatrix(model.W_Q, model.W_K.transpose(-1, -2))
W_OV = FactoredMatrix(model.W_V, model.W_O)

composition_scores_batched = dict()
composition_scores_batched["Q"] = get_batched_comp_scores(W_OV[0], W_QK[1])
composition_scores_batched["K"] = get_batched_comp_scores(W_OV[0], W_QK[1].T) #
composition_scores_batched["V"] = get_batched_comp_scores(W_OV[0], W_OV[1])

t.testing.assert_close(composition_scores_batched["Q"], composition_scores["Q"])
t.testing.assert_close(composition_scores_batched["K"], composition_scores["K"])
t.testing.assert_close(composition_scores_batched["V"], composition_scores["V"])
print("Tests passed - your `get_batched_comp_scores` function is working!")
```

```
Tests passed - your `get_batched_comp_scores` function is working!
```

▶ Hint

▶ Solution

Targeted Ablations

We can refine the ablation technique to detect composition by looking at the effect of the ablation on the attention pattern of an induction head, rather than the loss. Let's implement this!

Gotcha - by default, `run_with_hooks` removes any existing hooks when it runs. If you want to use caching, set the `reset_hooks_start` flag to False.

```
def ablation_induction_score(prev_head_index: Optional[int], ind_head_index: int
    '''
    Takes as input the index of the L0 head and the index of the L1 head, and th
    '''

    def ablation_hook(v, hook):
        if prev_head_index is not None:
            v[:, :, prev_head_index] = 0.0
        return v

    def induction_pattern_hook(attn, hook):
        hook.ctx[prev_head_index] = attn[0, ind_head_index].diag(-(seq_len - 1))

    model.run_with_hooks(
        rep_tokens,
        fwd_hooks=[
            (utils.get_act_name("v", 0), ablation_hook),
            (utils.get_act_name("pattern", 1), induction_pattern_hook)
        ],
    )
    return model.blocks[1].attn.hook_pattern.ctx[prev_head_index].item()


baseline_induction_score = ablation_induction_score(None, 4)
```