Indirect Object Identification

Introduction

This notebook / document is built around the Interpretability in the Wild paper, in which the authors aim to understand the **indirect object identification circuit** in GPT-2 small. This circuit is resposible for the model's ability to complete sentences like `"John and Mary went to the shops, John gave a bag to"` with the correct token `" Mary"`.

It is loosely divided into different sections, each one with their own flavour. Sections 1, 2 & 3 are derived from Neel Nanda's notebook Exploratory_Analysis_Demo. The flavour of these exercises is experimental and loose, with a focus on demonstrating what exploratory analysis looks like in practice with the transformerlens library. They skimp on rigour, and instead try to speedrun the process of finding suggestive evidence for this circuit. The code and exercises are simple and generic, but accompanied with a lot of detail about what each stage is doing, and why (plus several optional details and tangents). Section 4 introduces you to the idea of **path patching**, which is a more rigorous and structured way of analysing the model's behaviour. Here, you'll be replicating some of the results of the paper, which will serve to rigorously validate the insights gained from earlier sections. It's the most technically dense of all five sections. Lastly, sections 5 & 6 are much less structured, and have a stronger focus on open-ended exercises & letting you go off and explore for yourself.

Which exercises you want to do will depend on what you're hoping to get out of these exercises. For example:

- You want to understand activation patching - **1, 2, 3**
- You want to get a sense of how to do exploratory analysis on a model - **1, 2, 3**
- You want to understand activation and path patching - **1, 2, 3, 4**
- You want to understand the IOI circuit fully, and replicate the paper's key results - **1, 2, 3, 4, 5**
- You want to understand the IOI circuit fully, and replicate the paper's key results (but you already understand activation patching) - **1, 2, 4, 5**
- You want to understand IOI, and then dive deeper e.g. by looking for more circuits in models or investigating anomalies - **1, 2, 3, 4, 5, 6**

Note - if you find yourself getting frequent CUDA memory errors, you can periodically call `torch.cuda.empty_cache()` to free up some memory.

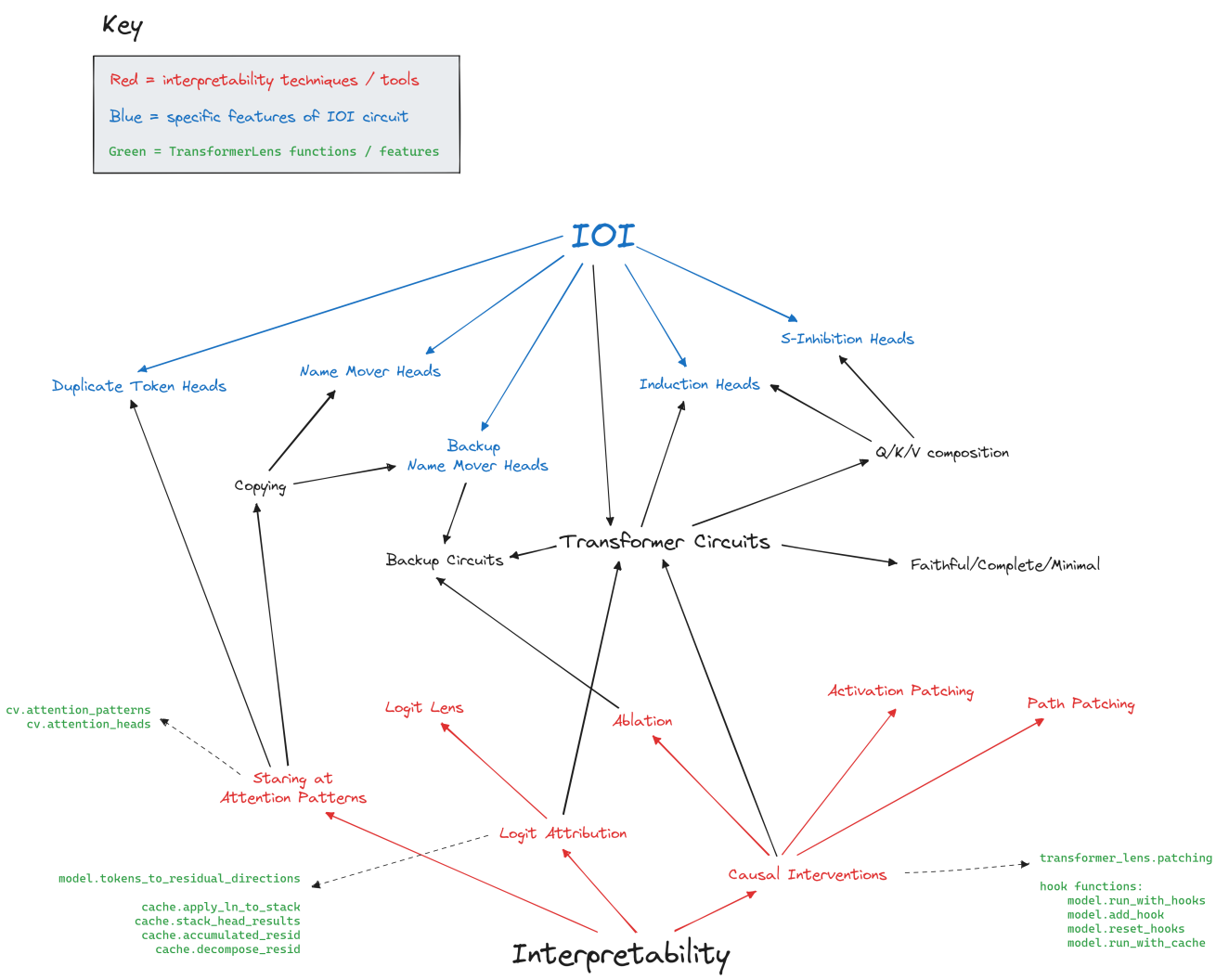
Each exercise will have a difficulty and importance rating out of 5, as well as an estimated maximum time you should spend on these exercises and sometimes a short annotation. You should interpret the ratings & time estimates relatively (e.g. if you find yourself spending about 50% longer on the exercises 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!

Have some sense of proportion

At a surface level, these exercises are designed to take you through the indirect object identification circuit. But it's also designed to make you a better interpretability researcher! As a result, most exercises will be doing a combination of:

1. Showing you some new feature/component of the circuit, and
2. Teaching you how to use tools and interpret results in a broader mech interp context.

Here is a rough conceptual graph showing all the different things you should be thinking about when going through these exercises, and how they relate to both of these goals, as well all the `transformerlens` tools which will help you.



A key idea to have in mind during these exercises is the spectrum from simpler, more exploratory tools to more rigoruous, complex tools. On the far left, you have something like inspecting attention patterns, which can give a decent (but sometimes misleading) picture of what an attention head is doing. These should be some of the first tools you reach for, and you should be using them a lot even before you have concrete hypotheses about a circuit. On the far right, you have something like path patching, which is a pretty rigorous and effortful tool that is best used when you already have reasonably concrete hypotheses about a circuit. As we go through the exercises, we'll transition from left to right along this spectrum.

The IOI task

The first step when trying to reverse engineer a circuit in a model is to identify *what* capability we want to reverse engineer. Indirect Object Identification is a task studied in Redwood Research's excellent Interpretability in the Wild paper (see Neel Nanda's interview with the authors or Kevin Wang's Twitter thread for an overview). The task is to complete sentences like "When Mary and John went to the store, John gave a drink to" with " Mary" rather than " John".

In the paper they rigorously reverse engineer a 26 head circuit, with 7 separate categories of heads used to perform this capability. The circuit they found roughly breaks down into three parts:

1. Identify what names are in the sentence
2. Identify which names are duplicated
3. Predict the name that is *not* duplicated

Why was this task chosen? The authors give a very good explanation for their choice in their video walkthrough of their paper, which you are encouraged to watch. To be brief, some of the reasons were:

- This is a fairly common grammatical structure, so we should expect the model to build some circuitry for solving it quite early on (after it's finished with all the more basic stuff, like n-grams, punctuation, induction, and simpler grammatical structures than this one).
- It's easy to measure: the model always puts a much higher probability on the IO and S tokens (i.e. `" Mary"` and `" John"`) than any others, and this is especially true once the model starts being stripped down to the core part of the circuit we're studying. So we can just take the logit difference between these two tokens, and use this as a metric for how well the model can solve the task.

- It is a crisp and well-defined task, so less likely to be solved in terms of memorisation of a large bag of heuristics (unlike e.g. tasks like "predict that the number `n+1` will follow `n`, which as Neel mentions in the video walkthrough is actually much more annoying and subtle than it first seems!).

A terminology note: `IO` will refer to the indirect object (in the example, `" Mary"`), `S1` and `S2` will refer to the two instances of the subject token (i.e. `" John"`), and `end` will refer to the end token `" to"` (because this is the position we take our prediction from, and we don't care about any tokens after this point). We will also sometimes use `S` to refer to the identity of the subject token (rather than referring to the first or second instance in particular).

Keeping track of your guesses & predictions

There's a lot to keep track of in these exercises as we work through them. You'll be exposed to new functions and modules from transformerlens, new ways to causally intervene in models, all the while building up your understanding of how the IOI task is performed. The notebook starts off exploratory in nature (lots of plotting and investigation), and gradually moves into more technical details, refined analysis, and replication of the paper's results, as we improve our understanding of the IOI circuit. You are recommended to keep a document or page of notes nearby as you go through these exercises, so you can keep track of the main takeaways from each section, as well as your hypotheses for how the model performs the task, and your ideas for how you might go off and test these hypotheses on your own if the notebook were to suddenly end.

If you are feeling extremely confused at any point, you can come back to the dropdown below, which contains diagrams explaining how the circuit works. There is also an accompanying intuitive explanation which you might find more helpful. However, I'd recommend you try and go through the notebook unassisted before looking at these.

▶ Intuitive explanation of IOI circuit
▶ Diagram 1 (simple)
▶ Diagram 2 (complex)

Content & Learning Objectives

1️⃣ Model & Task Setup

> Learning objectives
>
> - Understand the IOI task, and why the authors chose to study it
> - Build functions to demonstrate the model's performance on this task

2️⃣ Logit Attribution

> Learning objectives
>
> - Perform direct logit attribution to figure out which heads are writing to the residual stream in a significant way
> - Learn how to use different transformerlens helper functions, which decompose the residual stream in different ways

3️⃣ Activation Patching

> Learning objectives
>
> - Understand the idea of activation patching, and how it can be used
>   - Implement some of the activation patching helper functinos in transformerlens from scratch (i.e. using hooks)
> - Use activation patching to track the layers & sequence positions in the residual stream where important information is stored and processed
> - By the end of this section, you should be able to draw a rough sketch of the IOI circuit

4️⃣ Path Patching

> Learning objectives
>
> - Understand the idea of path patching, and how it differs from activation patching
> - Implement path patching from scratch (i.e. using hooks)
> - Replicate several of the results in the IOI paper

5️⃣ Paper Replication

> Learning objectives
>
> - Replicate most of the other results from the IOI paper

- Practice more open-ended, less guided coding

6️⃣ Bonus / exploring anomalies

Learning objectives

- Explore other parts of the model (e.g. negative name mover heads, and induction heads)
- Understand the subtleties present in model circuits, and the fact that there are often more parts to a circuit than seem obvious after initial investigation
- Understand the importance of the three quantitative criteria used by the paper: **faithfulness**, **completeness** and **minimality**

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 git+https://github.com/callummcdougall/CircuitsVis.git#subdirectory=python

    # 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/main.zip
        !unzip /content/main.zip 'ARENA_2.0-main/chapter1_transformers/exercises/*'
        sys.path.append("/content/ARENA_2.0-main/chapter1_transformers/exercises")
        os.remove("/content/main.zip")
        os.rename("ARENA_2.0-main/chapter1_transformers", "chapter1_transformers")
        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")

# Things that need to be done manually

# Need to unindent the following functions which take default args. A regex:

# def (logits_to_ave_logit_diff|residual_stack_to_logit_diff|get_path_patch_head_to_final_resid_post|get_path_patch_

# END

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

```
Looking in indexes: https://pypi.org/simple, https://us-python.pkg.dev/colab-wheels/public/simple/
Collecting einops
  Downloading einops-0.6.1-py3-none-any.whl (42 kB)
     ━━━━━━━━━━━━━━━━━━━━━━━━━━━━━━━━━━━━━━━━ 42.2/42.2 kB 2.9 MB/s eta 0:00:00
Installing collected packages: einops
Successfully installed einops-0.6.1
Looking in indexes: https://pypi.org/simple, https://us-python.pkg.dev/colab-wheels/public/simple/
Collecting jaxtyping
  Downloading jaxtyping-0.2.20-py3-none-any.whl (24 kB)
Requirement already satisfied: numpy>=1.20.0 in /usr/local/lib/python3.10/dist-packages (from jaxtyping) (1.22.4)
Collecting typeguard>=2.13.3 (from jaxtyping)
  Downloading typeguard-4.0.0-py3-none-any.whl (33 kB)
Requirement already satisfied: typing-extensions>=3.7.4.1 in /usr/local/lib/python3.10/dist-packages (from jaxtyp
Installing collected packages: typeguard, jaxtyping
Successfully installed jaxtyping-0.2.20 typeguard-4.0.0
Looking in indexes: https://pypi.org/simple, https://us-python.pkg.dev/colab-wheels/public/simple/
Collecting transformer_lens
  Downloading transformer_lens-1.2.2-py3-none-any.whl (88 kB)
     ━━━━━━━━━━━━━━━━━━━━━━━━━━━━━━━━━━━━━━━━ 88.9/88.9 kB 4.7 MB/s eta 0:00:00
Collecting datasets>=2.7.1 (from transformer_lens)
  Downloading datasets-2.13.1-py3-none-any.whl (486 kB)
     ━━━━━━━━━━━━━━━━━━━━━━━━━━━━━━━━━━━━━━━━ 486.2/486.2 kB 22.1 MB/s eta 0:00:00
Requirement already satisfied: einops>=0.6.0 in /usr/local/lib/python3.10/dist-packages (from transformer_lens) ((
Collecting fancy-einsum>=0.0.3 (from transformer_lens)
  Downloading fancy_einsum-0.0.3-py3-none-any.whl (6.2 kB)
Requirement already satisfied: jaxtyping>=0.2.11 in /usr/local/lib/python3.10/dist-packages (from transformer_lens
Collecting numpy>=1.23 (from transformer_lens)
  Downloading numpy-1.25.0-cp310-cp310-manylinux_2_17_x86_64.manylinux2014_x86_64.whl (17.6 MB)
     ━━━━━━━━━━━━━━━━━━━━━━━━━━━━━━━━━━━━━━━━ 17.6/17.6 MB 43.8 MB/s eta 0:00:00
Requirement already satisfied: pandas>=1.1.5 in /usr/local/lib/python3.10/dist-packages (from transformer_lens) (1
Requirement already satisfied: rich>=12.6.0 in /usr/local/lib/python3.10/dist-packages (from transformer_lens) (13
Requirement already satisfied: torch>=1.10 in /usr/local/lib/python3.10/dist-packages (from transformer_lens) (2.0
Requirement already satisfied: tqdm>=4.64.1 in /usr/local/lib/python3.10/dist-packages (from transformer_lens) (4
Collecting transformers>=4.25.1 (from transformer_lens)
  Downloading transformers-4.30.2-py3-none-any.whl (7.2 MB)
     ━━━━━━━━━━━━━━━━━━━━━━━━━━━━━━━━━━━━━━━━ 7.2/7.2 MB 85.7 MB/s eta 0:00:00
Collecting wandb>=0.13.5 (from transformer_lens)
  Downloading wandb-0.15.4-py3-none-any.whl (2.1 MB)
     ━━━━━━━━━━━━━━━━━━━━━━━━━━━━━━━━━━━━━━━━ 2.1/2.1 MB 90.2 MB/s eta 0:00:00
Requirement already satisfied: pyarrow>=8.0.0 in /usr/local/lib/python3.10/dist-packages (from datasets>=2.7.1->t
Collecting dill<0.3.7,>=0.3.0 (from datasets>=2.7.1->transformer_lens)
  Downloading dill-0.3.6-py3-none-any.whl (110 kB)
     ━━━━━━━━━━━━━━━━━━━━━━━━━━━━━━━━━━━━━━━━ 110.5/110.5 kB 16.4 MB/s eta 0:00:00
Requirement already satisfied: requests>=2.19.0 in /usr/local/lib/python3.10/dist-packages (from datasets>=2.7.1-
Collecting xxhash (from datasets>=2.7.1->transformer_lens)
  Downloading xxhash-3.2.0-cp310-cp310-manylinux_2_17_x86_64.manylinux2014_x86_64.whl (212 kB)
     ━━━━━━━━━━━━━━━━━━━━━━━━━━━━━━━━━━━━━━━━ 212.5/212.5 kB 14.8 MB/s eta 0:00:00
Collecting multiprocess (from datasets>=2.7.1->transformer_lens)
  Downloading multiprocess-0.70.14-py310-none-any.whl (134 kB)
     ━━━━━━━━━━━━━━━━━━━━━━━━━━━━━━━━━━━━━━━━ 134.3/134.3 kB 14.1 MB/s eta 0:00:00
Requirement already satisfied: fsspec[http]>=2021.11.1 in /usr/local/lib/python3.10/dist-packages (from datasets>=
Collecting aiohttp (from datasets>=2.7.1->transformer_lens)
  Downloading aiohttp-3.8.4-cp310-cp310-manylinux_2_17_x86_64.manylinux2014_x86_64.whl (1.0 MB)
     ━━━━━━━━━━━━━━━━━━━━━━━━━━━━━━━━━━━━━━━━ 1.0/1.0 MB 68.9 MB/s eta 0:00:00
Collecting huggingface-hub<1.0.0,>=0.11.0 (from datasets>=2.7.1->transformer_lens)
  Downloading huggingface_hub-0.15.1-py3-none-any.whl (236 kB)
     ━━━━━━━━━━━━━━━━━━━━━━━━━━━━━━━━━━━━━━━━ 236.8/236.8 kB 31.3 MB/s eta 0:00:00
Requirement already satisfied: packaging in /usr/local/lib/python3.10/dist-packages (from datasets>=2.7.1->transfo
Requirement already satisfied: pyyaml>=5.1 in /usr/local/lib/python3.10/dist-packages (from datasets>=2.7.1->trans
Requirement already satisfied: typeguard>=2.13.3 in /usr/local/lib/python3.10/dist-packages (from jaxtyping>=0.2.1
Requirement already satisfied: typing-extensions>=3.7.4.1 in /usr/local/lib/python3.10/dist-packages (from jaxtyp
Requirement already satisfied: python-dateutil>=2.8.1 in /usr/local/lib/python3.10/dist-packages (from pandas>=1.1
Requirement already satisfied: pytz>=2020.1 in /usr/local/lib/python3.10/dist-packages (from pandas>=1.1.5->trans
Requirement already satisfied: markdown-it-py>=2.2.0 in /usr/local/lib/python3.10/dist-packages (from rich>=12.6.0
Requirement already satisfied: pygments<3.0.0,>=2.13.0 in /usr/local/lib/python3.10/dist-packages (from rich>=12.6
Requirement already satisfied: filelock in /usr/local/lib/python3.10/dist-packages (from torch>=1.10->transformer_
Requirement already satisfied: sympy in /usr/local/lib/python3.10/dist-packages (from torch>=1.10->transformer_len
Requirement already satisfied: networkx in /usr/local/lib/python3.10/dist-packages (from torch>=1.10->transformer_
Requirement already satisfied: jinja2 in /usr/local/lib/python3.10/dist-packages (from torch>=1.10->transformer_le
Requirement already satisfied: triton==2.0.0 in /usr/local/lib/python3.10/dist-packages (from torch>=1.10->transfo
Requirement already satisfied: cmake in /usr/local/lib/python3.10/dist-packages (from triton==2.0.0->torch>=1.10->
Requirement already satisfied: lit in /usr/local/lib/python3.10/dist-packages (from triton==2.0.0->torch>=1.10->tr
Requirement already satisfied: regex!=2019.12.17 in /usr/local/lib/python3.10/dist-packages (from transformers>=4
Collecting tokenizers!=0.11.3,<0.14,>=0.11.1 (from transformers>=4.25.1->transformer_lens)
  Downloading tokenizers-0.13.3-cp310-cp310-manylinux_2_17_x86_64.manylinux2014_x86_64.whl (7.8 MB)
     ━━━━━━━━━━━━━━━━━━━━━━━━━━━━━━━━━━━━━━━━ 7.8/7.8 MB 62.6 MB/s eta 0:00:00
Collecting safetensors>=0.3.1 (from transformers>=4.25.1->transformer_lens)
  Downloading safetensors-0.3.1-cp310-cp310-manylinux_2_17_x86_64.manylinux2014_x86_64.whl (1.3 MB)
     ━━━━━━━━━━━━━━━━━━━━━━━━━━━━━━━━━━━━━━━━ 1.3/1.3 MB 82.8 MB/s eta 0:00:00
Requirement already satisfied: Click!=8.0.0,>=7.0 in /usr/local/lib/python3.10/dist-packages (from wandb>=0.13.5-
Collecting GitPython!=3.1.29,>=1.0.0 (from wandb>=0.13.5->transformer_lens)
  Downloading GitPython-3.1.31-py3-none-any.whl (184 kB)
     ━━━━━━━━━━━━━━━━━━━━━━━━━━━━━━━━━━━━━━━━ 184.3/184.3 kB 26.6 MB/s eta 0:00:00
Requirement already satisfied: psutil>=5.0.0 in /usr/local/lib/python3.10/dist-packages (from wandb>=0.13.5->trans
Collecting sentry-sdk>=1.0.0 (from wandb>=0.13.5->transformer_lens)
  Downloading sentry_sdk-1.26.0-py2.py3-none-any.whl (209 kB)
```

```
Downloading sentry_sdk-1.26.0-py2.py3-none-any.whl (209 kB)
     ━━━━━━━━━━━━━━━━━━━━━━━━━━━━━━━━━━━━━━━━ 209.4/209.4 kB 28.9 MB/s eta 0:00:00
Collecting docker-pycreds>=0.4.0 (from wandb>=0.13.5->transformer_lens)
  Downloading docker_pycreds-0.4.0-py2.py3-none-any.whl (9.0 kB)
Collecting pathtools (from wandb>=0.13.5->transformer_lens)
  Downloading pathtools-0.1.2.tar.gz (11 kB)
  Preparing metadata (setup.py) ... done
Collecting setproctitle (from wandb>=0.13.5->transformer_lens)
  Downloading setproctitle-1.3.2-cp310-cp310-manylinux_2_5_x86_64.manylinux1_x86_64.manylinux_2_17_x86_64.manylinu
Requirement already satisfied: setuptools in /usr/local/lib/python3.10/dist-packages (from wandb>=0.13.5->transfo
Requirement already satisfied: appdirs>=1.4.3 in /usr/local/lib/python3.10/dist-packages (from wandb>=0.13.5->tra
Requirement already satisfied: protobuf!=4.21.0,<5,>=3.19.0 in /usr/local/lib/python3.10/dist-packages (from wandb
Requirement already satisfied: six>=1.4.0 in /usr/local/lib/python3.10/dist-packages (from docker-pycreds>=0.4.0->
Requirement already satisfied: attrs>=17.3.0 in /usr/local/lib/python3.10/dist-packages (from aiohttp->datasets>=2
Requirement already satisfied: charset-normalizer<4.0,>=2.0 in /usr/local/lib/python3.10/dist-packages (from aioht
Collecting multidict<7.0,>=4.5 (from aiohttp->datasets>=2.7.1->transformer_lens)
  Downloading multidict-6.0.4-cp310-cp310-manylinux_2_17_x86_64.manylinux2014_x86_64.whl (114 kB)
     ━━━━━━━━━━━━━━━━━━━━━━━━━━━━━━━━━━━━━━━━ 114.5/114.5 kB 14.3 MB/s eta 0:00:00
Collecting async-timeout<5.0,>=4.0.0a3 (from aiohttp->datasets>=2.7.1->transformer_lens)
  Downloading async_timeout-4.0.2-py3-none-any.whl (5.8 kB)
Collecting yarl<2.0,>=1.0 (from aiohttp->datasets>=2.7.1->transformer_lens)
  Downloading yarl-1.9.2-cp310-cp310-manylinux_2_17_x86_64.manylinux2014_x86_64.whl (268 kB)
     ━━━━━━━━━━━━━━━━━━━━━━━━━━━━━━━━━━━━━━━━ 268.8/268.8 kB 38.3 MB/s eta 0:00:00
Collecting frozenlist>=1.1.1 (from aiohttp->datasets>=2.7.1->transformer_lens)
  Downloading frozenlist-1.3.3-cp310-cp310-manylinux_2_5_x86_64.manylinux1_x86_64.manylinux_2_17_x86_64.manylinux2
     ━━━━━━━━━━━━━━━━━━━━━━━━━━━━━━━━━━━━━━━━ 149.6/149.6 kB 21.4 MB/s eta 0:00:00
Collecting aiosignal>=1.1.2 (from aiohttp->datasets>=2.7.1->transformer_lens)
  Downloading aiosignal-1.3.1-py3-none-any.whl (7.6 kB)
Collecting gitdb<5,>=4.0.1 (from GitPython!=3.1.29,>=1.0.0->wandb>=0.13.5->transformer_lens)
  Downloading gitdb-4.0.10-py3-none-any.whl (62 kB)
     ━━━━━━━━━━━━━━━━━━━━━━━━━━━━━━━━━━━━━━━━ 62.7/62.7 kB 10.1 MB/s eta 0:00:00
Requirement already satisfied: mdurl~=0.1 in /usr/local/lib/python3.10/dist-packages (from markdown-it-py>=2.2.0->
Requirement already satisfied: urllib3<1.27,>=1.21.1 in /usr/local/lib/python3.10/dist-packages (from requests>=2
Requirement already satisfied: certifi>=2017.4.17 in /usr/local/lib/python3.10/dist-packages (from requests>=2.19
Requirement already satisfied: idna<4,>=2.5 in /usr/local/lib/python3.10/dist-packages (from requests>=2.19.0->dat
Requirement already satisfied: MarkupSafe>=2.0 in /usr/local/lib/python3.10/dist-packages (from jinja2->torch>=1.1
Requirement already satisfied: mpmath>=0.19 in /usr/local/lib/python3.10/dist-packages (from sympy->torch>=1.10->t
Collecting smmap<6,>=3.0.1 (from gitdb<5,>=4.0.1->GitPython!=3.1.29,>=1.0.0->wandb>=0.13.5->transformer_lens)
  Downloading smmap-5.0.0-py3-none-any.whl (24 kB)
Building wheels for collected packages: pathtools
  Building wheel for pathtools (setup.py) ... done
  Created wheel for pathtools: filename=pathtools-0.1.2-py3-none-any.whl size=8791 sha256=ce11490bff07c20c4753a5d7
  Stored in directory: /root/.cache/pip/wheels/e7/f3/22/152153d6eb222ee7a56ff8617d80ee5207207a8c00a7aab794
Successfully built pathtools
Installing collected packages: tokenizers, safetensors, pathtools, xxhash, smmap, setproctitle, sentry-sdk, numpy,
  Attempting uninstall: numpy
    Found existing installation: numpy 1.22.4
    Uninstalling numpy-1.22.4:
      Successfully uninstalled numpy-1.22.4
ERROR: pip's dependency resolver does not currently take into account all the packages that are installed. This be
numba 0.56.4 requires numpy<1.24,>=1.18, but you have numpy 1.25.0 which is incompatible.
tensorflow 2.12.0 requires numpy<1.24,>=1.22, but you have numpy 1.25.0 which is incompatible.
Successfully installed GitPython-3.1.31 aiohttp-3.8.4 aiosignal-1.3.1 async-timeout-4.0.2 datasets-2.13.1 dill-0.3
WARNING: The following packages were previously imported in this runtime:
  [numpy]
You must restart the runtime in order to use newly installed versions.
```

RESTART RUNTIME

```
Looking in indexes: https://pypi.org/simple, https://us-python.pkg.dev/colab-wheels/public/simple/
Collecting git+https://github.com/callummcdougall/CircuitsVis.git#subdirectory=python
  Cloning https://github.com/callummcdougall/CircuitsVis.git to /tmp/pip-req-build-vh_4efmb
  Running command git clone --filter=blob:none --quiet https://github.com/callummcdougall/CircuitsVis.git /tmp/pi
  Resolved https://github.com/callummcdougall/CircuitsVis.git to commit 5469cc5177cb11139470b18b3dc8ff174c3acf8d
  Installing build dependencies ... done
  Getting requirements to build wheel ... done
  Preparing metadata (pyproject.toml) ... done
Collecting importlib-metadata<6.0.0,>=5.1.0 (from circuitsvis==0.0.0)
  Downloading importlib_metadata-5.2.0-py3-none-any.whl (21 kB)
Requirement already satisfied: numpy<2.0,>=1.23 in /usr/local/lib/python3.10/dist-packages (from circuitsvis==0.0
Requirement already satisfied: torch<3.0,>=2.0 in /usr/local/lib/python3.10/dist-packages (from circuitsvis==0.0.0
Requirement already satisfied: zipp>=0.5 in /usr/local/lib/python3.10/dist-packages (from importlib-metadata<6.0.0
Requirement already satisfied: filelock in /usr/local/lib/python3.10/dist-packages (from torch<3.0,>=2.0->circuits
Requirement already satisfied: typing-extensions in /usr/local/lib/python3.10/dist-packages (from torch<3.0,>=2.0-
Requirement already satisfied: sympy in /usr/local/lib/python3.10/dist-packages (from torch<3.0,>=2.0->circuitsvis
Requirement already satisfied: networkx in /usr/local/lib/python3.10/dist-packages (from torch<3.0,>=2.0->circuits
Requirement already satisfied: jinja2 in /usr/local/lib/python3.10/dist-packages (from torch<3.0,>=2.0->circuitsvi
Requirement already satisfied: triton==2.0.0 in /usr/local/lib/python3.10/dist-packages (from torch<3.0,>=2.0->cir
Requirement already satisfied: cmake in /usr/local/lib/python3.10/dist-packages (from triton==2.0.0->torch<3.0,>=2
Requirement already satisfied: lit in /usr/local/lib/python3.10/dist-packages (from triton==2.0.0->torch<3.0,>=2.0
Requirement already satisfied: MarkupSafe>=2.0 in /usr/local/lib/python3.10/dist-packages (from jinja2->torch<3.0,
Requirement already satisfied: mpmath>=0.19 in /usr/local/lib/python3.10/dist-packages (from sympy->torch<3.0,>=2
Building wheels for collected packages: circuitsvis
  Building wheel for circuitsvis (pyproject.toml) ... done
  Created wheel for circuitsvis: filename=circuitsvis-0.0.0-py3-none-any.whl size=1784960 sha256=06b96870e7fd6dbf1
  Stored in directory: /tmp/pip-ephem-wheel-cache-uv53vz4k/wheels/86/be/ad/78078aba9344d200aad61b63d35cdaecdec1602
Successfully built circuitsvis
Installing collected packages: importlib-metadata, circuitsvis
```

```
Installing collected packages: importlib-metadata, circuitsvis
Successfully installed circuitsvis-0.0.0 importlib-metadata-5.2.0
--2023-06-26 00:35:25--  https://github.com/callummcdougall/ARENA_2.0/archive/refs/heads/main.zip
Resolving github.com (github.com)... 140.82.121.3
Connecting to github.com (github.com)|140.82.121.3|:443... connected.
HTTP request sent, awaiting response... 302 Found
Location: https://codeload.github.com/callummcdougall/ARENA_2.0/zip/refs/heads/main [following]
--2023-06-26 00:35:25--  https://codeload.github.com/callummcdougall/ARENA_2.0/zip/refs/heads/main
Resolving codeload.github.com (codeload.github.com)... 140.82.121.10
Connecting to codeload.github.com (codeload.github.com)|140.82.121.10|:443... connected.
HTTP request sent, awaiting response... 200 OK
Length: unspecified [application/zip]
Saving to: 'main.zip'

main.zip                [       <=>          ]  28.55M  11.5MB/s    in 2.5s

2023-06-26 00:35:28 (11.5 MB/s) - 'main.zip' saved [29941075]

Archive:  /content/main.zip
7c4a75381316b96d86eda8afd072afa2f2a50834
   creating: ARENA_2.0-main/chapter1_transformers/exercises/
   creating: ARENA_2.0-main/chapter1_transformers/exercises/part1_transformer_from_scratch/
  inflating: ARENA_2.0-main/chapter1_transformers/exercises/part1_transformer_from_scratch/solutions.py
   creating: ARENA_2.0-main/chapter1_transformers/exercises/part2_intro_to_mech_interp/
  inflating: ARENA_2.0-main/chapter1_transformers/exercises/part2_intro_to_mech_interp/solutions.py
  inflating: ARENA_2.0-main/chapter1_transformers/exercises/part2_intro_to_mech_interp/tests.py
   creating: ARENA_2.0-main/chapter1_transformers/exercises/part3_indirect_object_identification/
  inflating: ARENA_2.0-main/chapter1_transformers/exercises/part3_indirect_object_identification/ioi_circuit_extra
  inflating: ARENA_2.0-main/chapter1_transformers/exercises/part3_indirect_object_identification/ioi_dataset.py
  inflating: ARENA_2.0-main/chapter1_transformers/exercises/part3_indirect_object_identification/solutions.py
  inflating: ARENA_2.0-main/chapter1_transformers/exercises/part3_indirect_object_identification/tests.py
   creating: ARENA_2.0-main/chapter1_transformers/exercises/part4_interp_on_algorithmic_model/
  inflating: ARENA_2.0-main/chapter1_transformers/exercises/part4_interp_on_algorithmic_model/brackets_data.json
  inflating: ARENA_2.0-main/chapter1_transformers/exercises/part4_interp_on_algorithmic_model/brackets_datasets.py
  inflating: ARENA_2.0-main/chapter1_transformers/exercises/part4_interp_on_algorithmic_model/brackets_model_state
  inflating: ARENA_2.0-main/chapter1_transformers/exercises/part4_interp_on_algorithmic_model/solutions.py
  inflating: ARENA_2.0-main/chapter1_transformers/exercises/part4_interp_on_algorithmic_model/tests.py
   creating: ARENA_2.0-main/chapter1_transformers/exercises/part5_grokking_and_modular_arithmetic/
  inflating: ARENA_2.0-main/chapter1_transformers/exercises/part5_grokking_and_modular_arithmetic/my_utils.py
  inflating: ARENA_2.0-main/chapter1_transformers/exercises/part5_grokking_and_modular_arithmetic/solutions.py
  inflating: ARENA_2.0-main/chapter1_transformers/exercises/part5_grokking_and_modular_arithmetic/tests.py
   creating: ARENA_2.0-main/chapter1_transformers/exercises/part6_othellogpt/
```

```python
import os; os.environ["ACCELERATE_DISABLE_RICH"] = "1"
import sys
from pathlib import Path
import torch as t
from torch import Tensor
import numpy as np
import einops
from tqdm.notebook import tqdm
import plotly.express as px
import webbrowser
import re
import itertools
from jaxtyping import Float, Int, Bool
from typing import List, Optional, Callable, Tuple, Dict, Literal, Set
from functools import partial
from IPython.display import display, HTML
from rich.table import Table, Column
from rich import print as rprint
import circuitsvis as cv
from pathlib import Path
from transformer_lens.hook_points import HookPoint
from transformer_lens import utils, HookedTransformer, ActivationCache
from transformer_lens.components import Embed, Unembed, LayerNorm, MLP

t.set_grad_enabled(False)

# Make sure exercises are in the path
chapter = r"chapter1_transformers"
exercises_dir = Path(f"{os.getcwd().split(chapter)[0]}/{chapter}/exercises").resolve()
section_dir = (exercises_dir / "part3_indirect_object_identification").resolve()
if str(exercises_dir) not in sys.path: sys.path.append(str(exercises_dir))

from plotly_utils import imshow, line, scatter, bar
import part3_indirect_object_identification.tests as tests

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

MAIN = __name__ == "__main__"
```

1️⃣ Model & Task Setup

> Learning objectives
>
> - Understand the IOI task, and why the authors chose to study it
> - Build functions to demonstrate the model's performance on this task

Loading our model

The first step is to load in our model, GPT-2 Small, a 12 layer and 80M parameter transformer with `HookedTransformer.from_pretrained`. The various flags are simplifications that preserve the model's output but simplify its internals.

```
model = HookedTransformer.from_pretrained(
    "gpt2-small",
    center_unembed=True,
    center_writing_weights=True,
    fold_ln=True,
    refactor_factored_attn_matrices=True,
)
```

```
Downloading (…)lve/main/config.json: 100%  665/665 [00:00<00:00, 34.5kB/s]
Downloading model.safetensors: 100%  548M/548M [00:01<00:00, 285MB/s]
Downloading (…)neration_config.json: 100%  124/124 [00:00<00:00, 3.14kB/s]
Downloading (…)olve/main/vocab.json:  1.04M/? [00:00<00:00, 14.1MB/s]
Downloading (…)olve/main/merges.txt:  456k/? [00:00<00:00, 1.62MB/s]
Downloading (…)/main/tokenizer.json:  1.36M/? [00:00<00:00, 4.59MB/s]
Using pad_token, but it is not set yet.
```

▼ Note on `refactor_factored_attn_matrices` (optional)

This argument means we redefine the matrices W_Q, W_K, W_V and W_O in the model (without changing the model's actual behaviour).

For example, we know that instead of working with W_Q and W_K individually, the only matrix we actually need to use in the model is the low-rank matrix $W_Q W_K^T$ (note that I'm using the convention of matrix multiplication on the right, which matches the code in transformerlens and previous exercises in this series, but doesn't match Anthropic's Mathematical Frameworks paper). So if we perform singular value decomposition $W_Q W_K^T = USV^T$, then we see that we can just as easily define $W_Q = U\sqrt{S}$ and $W_K = V\sqrt{S}$ and use these instead. This means that W_Q and W_K both have orthogonal columns with matching norms. You can investigate this yourself (e.g. using the code below). This is arguably a more interpretable setup, because now there's no obvious asymmetry between the keys and queries.

There's also some fiddlyness with how biases are handled in this factorisation, which is why the comments above don't hold absolutely (see the documnentation for more info).

```
# Show column norms are the same (except first few, for fiddly bias reasons)
line([model.W_Q[0, 0].pow(2).sum(0), model.W_K[0, 0].pow(2).sum(0)])
# Show columns are orthogonal (except first few, again)
W_Q_dot_products = einops.einsum(
    model.W_Q[0, 0], model.W_Q[0, 0], "d_model d_head_1, d_model d_head_2 -> d_head_1 d_head_2"
)
imshow(W_Q_dot_products)
```

In a similar way, since $W_{OV} = W_V W_O = USV^T$, we can define $W_V = US$ and $W_O = V^T$. This is arguably a more interpretable setup, because now W_O is just a rotation, and doesn't change the norm, so z has the same norm as the result of the head.

▼ Note on `fold_ln`, `center_unembed` and `center_writing_weights` (optional)

See link here for comments.

The next step is to verify that the model can *actually* do the task! Here we use `utils.test_prompt`, and see that the model is significantly better at predicting Mary than John!

▼ Asides

Note: If we were being careful, we'd want to run the model on a range of prompts and find the average performance. We'll do more stuff like this in the fourth section (when we try to replicate some of the paper's results, and take a more rigorous approach).

`prepend_bos` is a flag to add a BOS (beginning of sequence) to the start of the prompt. GPT-2 was not trained with this, but I find that it often makes model behaviour more stable, as the first token is treated weirdly.

```
# Here is where we test on a single prompt
# Result: 70% probability on Mary, as we expect

example_prompt = "After John and Mary went to the store, John gave a bottle of milk to"
example_answer = " Mary"
utils.test_prompt(example_prompt, example_answer, model, prepend_bos=True)
```

We now want to find a reference prompt to run the model on. Even though our ultimate goal is to reverse engineer how this behaviour is done in general, often the best way to start out in mechanistic interpretability is by zooming in on a concrete example and understanding it in detail, and only *then* zooming out and verifying that our analysis generalises. In section 3, we'll work with a dataset similar to the one used by the paper authors, but this probably wouldn't be the first thing we reached for if we were just doing initial investigations.

We'll run the model on 4 instances of this task, each prompt given twice - one with the first name as the indirect object, one with the second name. To make our lives easier, we'll carefully choose prompts with single token names and the corresponding names in the same token positions.

▼ Aside on tokenization

We want models that can take in arbitrary text, but models need to have a fixed vocabulary. So the solution is to define a vocabulary of **tokens** and to deterministically break up arbitrary text into tokens. Tokens are, essentially, subwords, and are determined by finding the most frequent substrings - this means that tokens vary a lot in length and frequency!

Tokens are a *massive* headache and are one of the most annoying things about reverse engineering language models... Different names will be different numbers of tokens, different prompts will have the relevant tokens at different positions, different prompts will have different total numbers of tokens, etc. Language models often devote significant amounts of parameters in early layers to convert inputs from tokens to a more sensible internal format (and do the reverse in later layers). You really, really want to avoid needing to think about tokenization wherever possible when doing exploratory analysis (though, of course, it's relevant later when trying to flesh out your analysis and make it rigorous!). HookedTransformer comes with several helper methods to deal with tokens: `to_tokens, to_string, to_str_tokens, to_single_token, get_token_position`

Exercise: I recommend using `model.to_str_tokens` to explore how the model tokenizes different strings. In particular, try adding or removing spaces at the start, or changing capitalization - these change tokenization!

```
prompt_format = [
    "When John and Mary went to the shops,{} gave the bag to",
    "When Tom and James went to the park,{} gave the ball to",
    "When Dan and Sid went to the shops,{} gave an apple to",
    "After Martin and Amy went to the park,{} gave a drink to",
]
name_pairs = [
    (" Mary", " John"),
    (" Tom", " James"),
    (" Dan", " Sid"),
    (" Martin", " Amy"),
]

# Define 8 prompts, in 4 groups of 2 (with adjacent prompts having answers swapped)
prompts = [
```

```
    prompt.format(name)
    for (prompt, names) in zip(prompt_format, name_pairs) for name in names[::-1]
]
# Define the answers for each prompt, in the form (correct, incorrect)
answers = [names[::i] for names in name_pairs for i in (1, -1)]
# Define the answer tokens (same shape as the answers)
answer_tokens = t.concat([
    model.to_tokens(names, prepend_bos=False).T for names in answers
])

rprint(prompts)
rprint(answers)
rprint(answer_tokens)
```

```
table = Table("Prompt", "Correct", "Incorrect", title="Prompts & Answers:")

for prompt, answer in zip(prompts, answers):
    table.add_row(prompt, repr(answer[0]), repr(answer[1]))

rprint(table)
```

▼ Aside - the `rich` library

The outputs above were created by `rich`, a fun library which prints things in nice formats. It has functions like `rich.table.Table`, which are very easy to use but can produce visually clear outputs which are sometimes useful.

You can also color the columns of a table, by using the `rich.table.Column` argument with the `style` parameter:

```
cols = [
    "Prompt",
    Column("Correct", style="rgb(0,200,0) bold"),
    Column("Incorrect", style="rgb(255,0,0) bold"),
]
table = Table(*cols, title="Prompts & Answers:")

for prompt, answer in zip(prompts, answers):
    table.add_row(prompt, repr(answer[0]), repr(answer[1]))
```

```
rprint(table)
```

We now run the model on these prompts and use `run_with_cache` to get both the logits and a cache of all internal activations for later analysis.

```
tokens = model.to_tokens(prompts, prepend_bos=True)
# Move the tokens to the GPU
tokens = tokens.to(device)
# Run the model and cache all activations
original_logits, cache = model.run_with_cache(tokens)
```

We'll later be evaluating how model performance differs upon performing various interventions, so it's useful to have a metric to measure model performance. Our metric here will be the **logit difference**, the difference in logit between the indirect object's name and the subject's name (eg, `logit(Mary) - logit(John)`).

Exercise - implement the performance evaluation function

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

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

It's important to understand exactly what this function is computing, and why it matters.
```

This function should take in your model's logit output (shape `(batch, seq, d_vocab)`), and the array of answer tokens (shape `(batch, 2)`, containing the token ids of correct and incorrect answers respectively for each sequence), and return the logit difference as described above. If `per_prompt` is False, then it should take the mean over the batch dimension, if not then it should return an array of length `batch`.

```
def logits_to_ave_logit_diff(
    logits: Float[Tensor, "batch seq d_vocab"],
    answer_tokens: Float[Tensor, "batch 2"] = answer_tokens,
    per_prompt: bool = False
):
    '''
    Returns logit difference between the correct and incorrect answer.

    If per_prompt=True, return the array of differences rather than the average.
    '''
    # SOLUTION
    # Only the final logits are relevant for the answer
    final_logits: Float[Tensor, "batch d_vocab"] = logits[:, -1, :]
    # Get the logits corresponding to the indirect object / subject tokens respectively
    answer_logits: Float[Tensor, "batch 2"] = final_logits.gather(dim=-1, index=answer_tokens)
    # Find logit difference
    correct_logits, incorrect_logits = answer_logits.unbind(dim=-1)
    answer_logit_diff = correct_logits - incorrect_logits
    return answer_logit_diff if per_prompt else answer_logit_diff.mean()


tests.test_logits_to_ave_logit_diff(logits_to_ave_logit_diff)

original_per_prompt_diff = logits_to_ave_logit_diff(original_logits, answer_tokens, per_prompt=True)
print("Per prompt logit difference:", original_per_prompt_diff)
original_average_logit_diff = logits_to_ave_logit_diff(original_logits, answer_tokens)
print("Average logit difference:", original_average_logit_diff)

cols = [
    "Prompt",
    Column("Correct", style="rgb(0,200,0) bold"),
    Column("Incorrect", style="rgb(255,0,0) bold"),
    Column("Logit Difference", style="bold")
]
table = Table(*cols, title="Logit differences")

for prompt, answer, logit_diff in zip(prompts, answers, original_per_prompt_diff):
    table.add_row(prompt, repr(answer[0]), repr(answer[1]), f"{logit_diff.item():.3f}")

rprint(table)
```

▶ Solution

Brainstorm What's Actually Going On

Before diving into running experiments, it's often useful to spend some time actually reasoning about how the behaviour in question could be implemented in the transformer. **This is optional, and you'll likely get the most out of engaging with this section if you have a decent understanding already of what a transformer is and how it works!**

You don't have to do this and forming hypotheses after exploration is also reasonable, but I think it's often easier to explore and interpret results with some grounding in what you might find. In this particular case, I'm cheating somewhat, since I know the answer, but I'm trying to simulate the process of reasoning about it!

Note that often your hypothesis will be wrong in some ways and often be completely off. We're doing science here, and the goal is to understand how the model *actually* works, and to form true beliefs! There are two separate traps here at two extremes that it's worth tracking:

- Confusion: Having no hypotheses at all, getting a lot of data and not knowing what to do with it, and just floundering around
- Dogmatism: Being overconfident in an incorrect hypothesis and being unwilling to let go of it when reality contradicts you, or flinching away from running the experiments that might disconfirm it.

Exercise: Spend some time thinking through how you might imagine this behaviour being implemented in a transformer. Try to think through this for yourself before reading through my thoughts!

▶ (*) **My reasoning**

2️⃣ Logit Attribution

> Learning objectives
>
> - Perform direct logit attribution to figure out which heads are writing to the residual stream in a significant way
> - Learn how to use different transformerlens helper functions, which decompose the residual stream in different ways

Direct Logit Attribution

The easiest part of the model to understand is the output - this is what the model is trained to optimize, and so it can always be directly interpreted! Often the right approach to reverse engineering a circuit is to start at the end, understand how the model produces the right answer, and to then work backwards (you will have seen this if you went through the balanced bracket classifier task, and in fact if you did then this section will probably be quite familiar to you and you should feel free to just skim through it). The main technique used to do this is called **direct logit attribution**

Background: The central object of a transformer is the **residual stream**. This is the sum of the outputs of each layer and of the original token and positional embedding. Importantly, this means that any linear function of the residual stream can be perfectly decomposed into the contribution of each layer of the transformer. Further, each attention layer's output can be broken down into the sum of the output of each head (See A Mathematical Framework for Transformer Circuits for details), and each MLP layer's output can be broken down into the sum of the output of each neuron (and a bias term for each layer).

The logits of a model are `logits=Unembed(LayerNorm(final_residual_stream))`. The Unembed is a linear map, and LayerNorm is approximately a linear map, so we can decompose the logits into the sum of the contributions of each component, and look at which components contribute the most to the logit of the correct token! This is called **direct logit attribution**. Here we look at the direct attribution to the logit difference!

Background and motivation of the logit difference

Logit difference is actually a *really* nice and elegant metric and is a particularly nice aspect of the setup of Indirect Object Identification. In general, there are two natural ways to interpret the model's outputs: the output logits, or the output log probabilities (or probabilities).

The logits are much nicer and easier to understand, as noted above. However, the model is trained to optimize the cross-entropy loss (the average of log probability of the correct token). This means it does not directly optimize the logits, and indeed if the model adds an arbitrary constant to every logit, the log probabilities are unchanged.

But we have:

```
log_probs == logits.log_softmax(dim=-1) == logits - logsumexp(logits)
```

and so:

```
log_probs(" Mary") - log_probs(" John") = logits(" Mary") - logits(" John")
```

- the ability to add an arbitrary constant cancels out!

▶ Technical details (if this equivalence doesn't seem obvious to you)

Further, the metric helps us isolate the precise capability we care about - figuring out *which* name is the Indirect Object. There are many other components of the task - deciding whether to return an article (the) or pronoun (her) or name, realising that the sentence wants a person next at all, etc. By taking the logit difference we control for all of that.

Our metric is further refined, because each prompt is repeated twice, for each possible indirect object. This controls for irrelevant behaviour such as the model learning that John is a more frequent token than Mary (this actually happens! The final layernorm bias increases the

John logit by 1 relative to the Mary logit). Another way to handle this would be to use a large enough dataset (with names randomly chosen) that this effect is averaged out, which is what we'll do in section 3.

▶ Ignoring LayerNorm

Logit diff directions

Getting an output logit is equivalent to projecting onto a direction in the residual stream, and the same is true for getting the logit diff.

▶ If it's not clear what is meant by this statement, read this dropdown.

We use `model.tokens_to_residual_directions` to map the answer tokens to that direction, and then convert this to a logit difference direction for each batch

```
answer_residual_directions: Float[Tensor, "batch 2 d_model"] = model.tokens_to_residual_directions(answer_tokens)
print("Answer residual directions shape:", answer_residual_directions.shape)

correct_residual_directions, incorrect_residual_directions = answer_residual_directions.unbind(dim=1)
logit_diff_directions: Float[Tensor, "batch d_model"] = correct_residual_directions - incorrect_residual_directions
print(f"Logit difference directions shape:", logit_diff_directions.shape)
```

▶ Aside - type annotations

To verify that this works, we can apply this to the final residual stream for our cached prompts (after applying LayerNorm scaling) and verify that we get the same answer.

▶ Technical details

The code below does the following:

- Gets the final residual stream values from the `cache` object (which you should already have defined above).
- Apply layernorm scaling to these values.
    - This is done by `cache.apply_to_ln_stack`, a helpful function which takes a stack of residual stream values (e.g. a batch, or the residual stream decomposed into components), treats them as the input to a specific layer, and applies the layer norm scaling of that layer to them.
    - The keyword arguments here indicate that our input is the residual stream values for the last sequence position, and we want to apply the final layernorm in the model.
- Project them along the unembedding directions (you've already defined these above, as `logit_diff_directions`).

```
# cache syntax - resid_post is the residual stream at the end of the layer, -1 gets the final layer. The general syn
final_residual_stream: Float[Tensor, "batch seq d_model"] = cache["resid_post", -1]
print(f"Final residual stream shape: {final_residual_stream.shape}")
final_token_residual_stream: Float[Tensor, "batch d_model"] = final_residual_stream[:, -1, :]

# Apply LayerNorm scaling (to just the final sequence position)
# pos_slice is the subset of the positions we take - here the final token of each prompt
scaled_final_token_residual_stream = cache.apply_ln_to_stack(final_token_residual_stream, layer=-1, pos_slice=-1)

average_logit_diff = einops.einsum(
    scaled_final_token_residual_stream, logit_diff_directions,
    "batch d_model, batch d_model ->"
) / len(prompts)

print(f"Calculated average logit diff: {average_logit_diff:.10f}")
print(f"Original logit difference:     {original_average_logit_diff:.10f}")

t.testing.assert_close(average_logit_diff, original_average_logit_diff)
```

Logit Lens

We can now decompose the residual stream! First we apply a technique called the **logit lens** - this looks at the residual stream after each layer and calculates the logit difference from that. This simulates what happens if we delete all subsequence layers.

Exercise - implement `residual_stack_to_logit_diff`

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

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

Again, make sure you understand what the output of this function represents.
```

This function should look a lot like your code immediately above. `residual_stack` is a tensor of shape `(..., batch, d_model)` containing the residual stream values for the final sequence position. You should apply the final layernorm to these values, then project them in the logit difference directions.

```
def residual_stack_to_logit_diff(
    residual_stack: Float[Tensor, "... batch d_model"],
    cache: ActivationCache,
    logit_diff_directions: Float[Tensor, "batch d_model"] = logit_diff_directions,
) -> Float[Tensor, "..."]:
    '''
    Gets the avg logit difference between the correct and incorrect answer for a given
    stack of components in the residual stream.
    '''
    # SOLUTION
    batch_size = residual_stack.size(-2)
    scaled_residual_stack = cache.apply_ln_to_stack(residual_stack, layer=-1, pos_slice=-1)
    return einops.einsum(
        scaled_residual_stack, logit_diff_directions,
        "... batch d_model, batch d_model -> ..."
    ) / batch_size


# Test function by checking that it gives the same result as the original logit difference
t.testing.assert_close(
    residual_stack_to_logit_diff(final_token_residual_stream, cache),
    original_average_logit_diff
)
```

▶ Solution

Once you have the solution, you can plot your results.

▶ Details on `accumulated_resid`

```
accumulated_residual, labels = cache.accumulated_resid(layer=-1, incl_mid=True, pos_slice=-1, return_labels=True)
# accumulated_residual has shape (component, batch, d_model)

logit_lens_logit_diffs: Float[Tensor, "component"] = residual_stack_to_logit_diff(accumulated_residual, cache)

line(
    logit_lens_logit_diffs,
    hovermode="x unified",
    title="Logit Difference From Accumulated Residual Stream",
    labels={"x": "Layer", "y": "Logit Diff"},
    xaxis_tickvals=labels,
    width=800
)
```

▶ Question - what is the interpretation of this plot? What does this tell you about how the model solves this task?

Layer Attribution

We can repeat the above analysis but for each layer (this is equivalent to the differences between adjacent residual streams)

Note: Annoying terminology overload - layer k of a transformer means the kth **transformer block**, but each block consists of an **attention layer** (to move information around) *and* an **MLP layer** (to process information).

```
per_layer_residual, labels = cache.decompose_resid(layer=-1, pos_slice=-1, return_labels=True)
per_layer_logit_diffs = residual_stack_to_logit_diff(per_layer_residual, cache)

line(
    per_layer_logit_diffs,
    hovermode="x unified",
    title="Logit Difference From Each Layer",
    labels={"x": "Layer", "y": "Logit Diff"},
    xaxis_tickvals=labels,
    width=800
)
```

▶ Question - what is the interpretation of this plot? What does this tell you about how the model solves this task?

Head Attribution

We can further break down the output of each attention layer into the sum of the outputs of each attention head. Each attention layer consists of 12 heads, which each act independently and additively.

▶ Decomposing attention output into sums of heads

```
per_head_residual, labels = cache.stack_head_results(layer=-1, pos_slice=-1, return_labels=True)
per_head_residual = einops.rearrange(
    per_head_residual,
    "(layer head) ... -> layer head ...",
    layer=model.cfg.n_layers
)
per_head_logit_diffs = residual_stack_to_logit_diff(per_head_residual, cache)

imshow(
    per_head_logit_diffs,
    labels={"x":"Head", "y":"Layer"},
    title="Logit Difference From Each Head",
    width=600
)
```

We see that only a few heads really matter - heads 9.6 and 9.9 contribute a lot positively (explaining why attention layer 9 is so important), while heads 10.7 and 11.10 contribute a lot negatively (explaining why attention layer 10 and layer 11 are actively harmful). These correspond to (some of) the name movers and negative name movers discussed in the paper. There are also several heads that matter positively or negatively but less strongly (other name movers and backu name movers)

There are a few meta observations worth making here - our model has 144 heads, yet we could localise this behaviour to a handful of specific heads, using straightforward, general techniques. This supports the claim in A Mathematical Framework that attention heads are the right level of abstraction to understand attention. It also really surprising that there are *negative* heads - eg 10.7 makes the incorrect logit 7x *more* likely. I'm not sure what's going on there, though the paper discusses some possibilities.

Recap of useful functions from this section

Here, we take stock of all the functions from transformerlens which you might not have seen previously.

- `cache.apply_ln_to_stack`
  - Apply layernorm scaling to a stack of residual stream values.
  - We used this to help us go from "final value in residual stream" to "projection of logits in logit difference directions", without getting the code too messy!
- `cache.accumulated_resid(layer=None)`

- Returns the accumulated residual stream up to layer `layer` (or up to the final value of residual stream if layer is None), i.e. a stack of previous residual streams up to that layer's input.
- Useful when studying the **logit lens**.
- First dimension of output is `(0_pre, 0_mid, 1_pre, 1_mid, ..., final_post)`

- `cache.decompose_resid(layer)`.
  - Decomposes the residual stream input to layer `layer` into a stack of the output of previous layers. The sum of these is the input to layer `layer`.
  - First dimension of output is `(embed, pos_embed, 0_attn_out, 0_mlp_out, ...)`.
- `cache.stack_head_results(layer)`
  - Returns a stack of all head results (i.e. residual stream contribution) up to layer `layer`
  - (i.e. like `decompose_resid` except it splits each attention layer by head rather than splitting each layer by attention/MLP)
  - First dimension of output is `layer * head` (we needed to rearrange to `(layer, head)` to plot it).

Attention Analysis

Attention heads are particularly fruitful to study because we can look directly at their attention patterns and study from what positions they move information from and to. This is particularly useful here as we're looking at the direct effect on the logits so we need only look at the attention patterns from the final token.

We use the `circuitsvis` library (developed from Anthropic's PySvelte library) to visualize the attention patterns! We visualize the top 3 positive and negative heads by direct logit attribution, and show these for the first prompt (as an illustration).

▶ Interpreting Attention Patterns

```
def topk_of_Nd_tensor(tensor: Float[Tensor, "rows cols"], k: int):
    '''
    Helper function: does same as tensor.topk(k).indices, but works over 2D tensors.
    Returns a list of indices, i.e. shape [k, tensor.ndim].

    Example: if tensor is 2D array of values for each head in each layer, this will
    return a list of heads.
    '''
    i = t.topk(tensor.flatten(), k).indices
    return np.array(np.unravel_index(utils.to_numpy(i), tensor.shape)).T.tolist()


k = 3

for head_type in ["Positive", "Negative"]:

    # Get the heads with largest (or smallest) contribution to the logit difference
    top_heads = topk_of_Nd_tensor(per_head_logit_diffs * (1 if head_type=="Positive" else -1), k)

    # Get all their attention patterns
    attn_patterns_for_important_heads: Float[Tensor, "head q k"] = t.stack([
        cache["pattern", layer][:, head][0]
         for layer, head in top_heads
    ])

    # Display results
    display(HTML(f"<h2>Top {k} {head_type} Logit Attribution Heads</h2>"))
    display(cv.attention.attention_patterns(
        attention = attn_patterns_for_important_heads,
        tokens = model.to_str_tokens(tokens[0]),
        attention_head_names = [f"{layer}.{head}" for layer, head in top_heads],
    ))
```

Reminder - you can use `attention_patterns` or `attention_heads` for these visuals. The former lets you see the actual values, the latter lets you hover over tokens in a printed sentence (and it provides other useful features like locking on tokens, or a superposition of all heads in the display). Both can be useful in different contexts (although I'd recommend usually using `attention_patterns`, it's more useful in most cases for quickly getting a sense of attention patterns).

Try replacing `attention_patterns` above with `attention_heads`, and compare the output.

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

From these plots, you might want to start thinking about the algorithm which is being implemented. In particular, for the attention heads with high positive attribution scores, where is `" to"` attending to? How might this head be affecting the logit diff score?

We'll save a full hypothesis for how the model works until the end of the next section.

3️⃣ Activation Patching

> Learning objectives
>
> - Understand the idea of activation patching, and how it can be used
>   - Implement some of the activation patching helper functinos in transformerlens from scratch (i.e. using hooks)
> - Use activation patching to track the layers & sequence positions in the residual stream where important information is stored and processed
> - By the end of this section, you should be able to draw a rough sketch of the IOI circuit

Introduction

The obvious limitation to the techniques used above is that they only look at the very end of the circuit - the parts that directly affect the logits. Clearly this is not sufficient to understand the circuit! We want to understand how things compose together to produce this final output, and ideally to produce an end-to-end circuit fully explaining this behaviour.

The technique we'll use to investigate this is called **activation patching**. This was first introduced in David Bau and Kevin Meng's excellent ROME paper, there called causal tracing.

The setup of activation patching is to take two runs of the model on two different inputs, the clean run and the corrupted run. The clean run outputs the correct answer and the corrupted run does not. The key idea is that we give the model the corrupted input, but then **intervene** on a specific activation and **patch** in the corresponding activation from the clean run (ie replace the corrupted activation with the clean activation), and then continue the run. And we then measure how much the output has updated towards the correct answer.

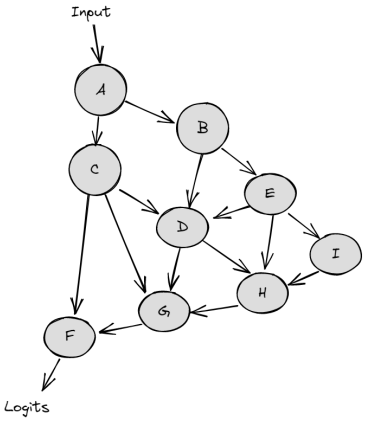
We can then iterate over many possible activations and look at how much they affect the corrupted run. If patching in an activation significantly increases the probability of the correct answer, this allows us to *localise* which activations matter.

In other words, this is a **noising** algorithm (unlike last section which was mostly **denoising**).

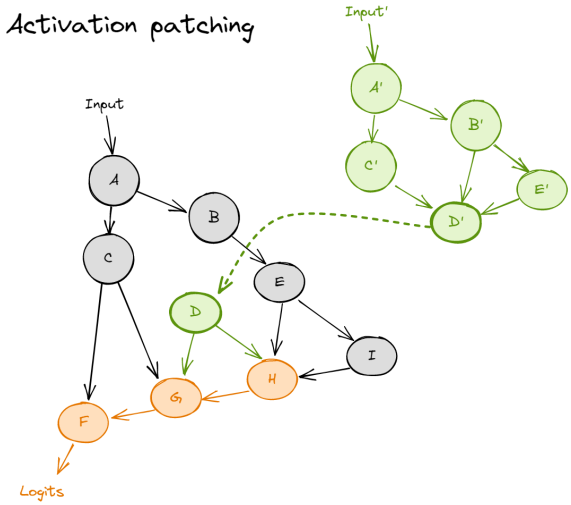
The ability to localise is a key move in mechanistic interpretability - if the computation is diffuse and spread across the entire model, it is likely much harder to form a clean mechanistic story for what's going on. But if we can identify precisely which parts of the model matter, we can then zoom in and determine what they represent and how they connect up with each other, and ultimately reverse engineer the underlying circuit that they represent.

The diagrams below demonstrate activation patching on an abstract neural network (the nodes represent activations, and the arrows between them are weight connections).

A regular forward pass on the clean input looks like:

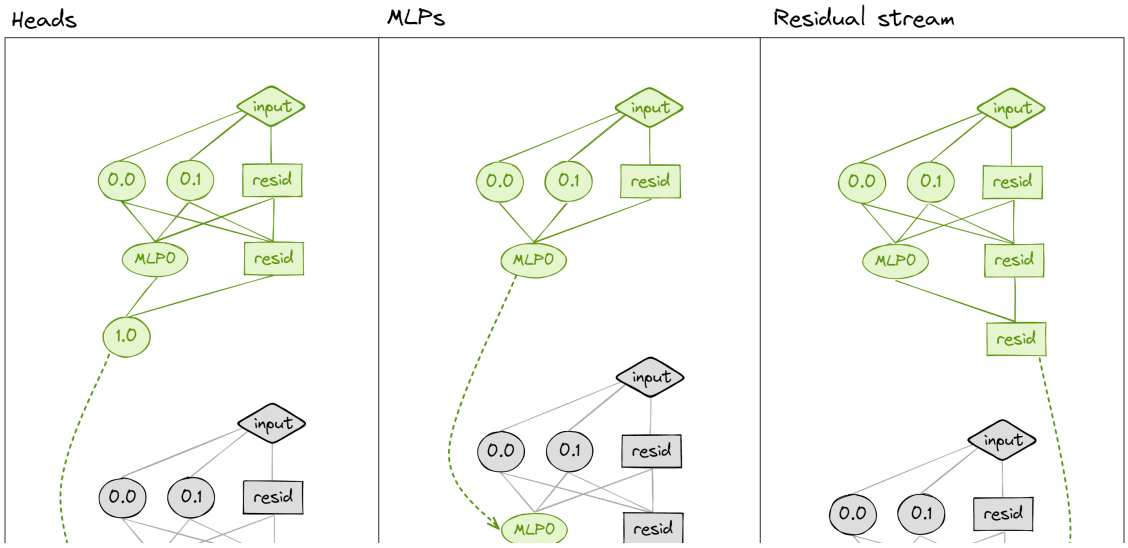


And activation patching from a corrupted input (green) into a forward pass for the clean input (black) looks like:



where the dotted line represents patching in a value (i.e. during the forward pass on the clean input, we replace node D with the value it takes on the corrupted input). Nodes H, G and F are colored orange, to represent that they now follow a distribution which is not the same as clean or corrupted.

We can patch into a transformer in many different ways (e.g. values of the residual stream, the MLP, or attention heads' output - see below). We can also get even more granular by patching at particular sequence positions (not shown in diagram).



The above was all fairly abstract, so let's zoom in and lay out a concrete example to understand Indirect Object Identification.

Here our clean input will be the original sentences (e.g. `"When Mary and John went to the store, John gave a drink to"`) and our corrupted input will have the subject token flipped (e.g. `"When Mary and John went to the store, Mary gave a drink to"`). Patching from the corrupted input to the clean input is a causal intervention which will allow us to understand precisely which parts of the network are identifying the indirect object. If a component is important, then patching into that component will reverse the signal that this component produces, hence making performance much worse.

Question - we could instead have our corrupted sentence be `"When John and Mary went to the store, Mary gave a drink to"` (i.e. flip all 3 occurrences of names in the sentence). Why do you think we don't do this?

▶ Hint

▶ Answer

One natural thing to patch in is the residual stream at a specific layer and specific position. For example, the model is likely intitially doing some processing on the `S2` token to realise that it's a duplicate, but then uses attention to move that information to the `end` token. So patching in the residual stream at the `end` token will likely matter a lot in later layers but not at all in early layers.

We can zoom in much further and patch in specific activations from specific layers. For example, we think that the output of head 9.9 on the final token is significant for directly connecting to the logits, so we predict that just patching the output of this head will significantly affect performance.

Note that this technique does *not* tell us how the components of the circuit connect up, just what they are.

TransformerLens has helpful built-in functions to perform activation patching, but in order to understand the process better, you're now going to implement some of these functions from first principles (i.e. just using hooks). You'll be able to test your functions by comparing their output to the built-in functions.

If you need a refresher on hooks, you can return to the exercises on induction heads (which take you through how to use hooks, as well as how to cache activations).

```
from transformer_lens import patching
```

Creating a metric

Before we patch, we need to create a metric for evaluating a set of logits. Since we'll be running our **corrupted prompts** (with `S2` replaced with the wrong name) and patching in our **clean prompts**, it makes sense to choose a metric such that:

- A value of zero means no change (from the performance on the corrupted prompt)
- A value of one means clean performance has been completely recovered

For example, if we patched in the entire clean prompt, we'd get a value of one. If our patching actually makes the model even better at solving the task than its regular behaviour on the clean prompt then we'd get a value greater than 1, but generally we expect values between 0 and 1.

It also makes sense to have the metric be a linear function of the logit difference. This is enough to uniquely specify a metric.

```
clean_tokens = tokens
# Swap each adjacent pair to get corrupted tokens
indices = [i+1 if i % 2 == 0 else i-1 for i in range(len(tokens))]
corrupted_tokens = clean_tokens[indices]

print(
    "Clean string 0:    ", model.to_string(clean_tokens[0]), "\n"
    "Corrupted string 0:", model.to_string(corrupted_tokens[0])
)

clean_logits, clean_cache = model.run_with_cache(clean_tokens)
corrupted_logits, corrupted_cache = model.run_with_cache(corrupted_tokens)

clean_logit_diff = logits_to_ave_logit_diff(clean_logits, answer_tokens)
print(f"Clean logit diff: {clean_logit_diff:.4f}")

corrupted_logit_diff = logits_to_ave_logit_diff(corrupted_logits, answer_tokens)
print(f"Corrupted logit diff: {corrupted_logit_diff:.4f}")
```

Exercise - create a metric

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

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

Fill in the function `ioi_metric` below, to create the required metric. Note that we can afford to use default arguments in this function, because we'll be using the same dataset for this whole section.

Important note - this function needs to return a scalar tensor, rather than a float. If not, then some of the patching functions later on won't work. The type signature of this is `Float[Tensor, ""]`.

Second important note - we've defined this to be 0 when performance is the same as on corrupted input, and 1 when it's the same as on clean input. Why have we done it this way around? The answer is that, in this section, we'll be applying **denoising** rather than **noising** methods. **Denoising** means we start with the corrupted input (i.e. no signal, or negative signal) and patch in with the clean input (i.e. positive signal). This is an important conceptual distinction. When we perform denoising, we're looking for parts of the model which are **sufficient** for the task (e.g. which parts of the model have enough information to recover the correct answer from the corrupted input). Our "null hypothesis" is that a part of the model isn't important, and so changing its value won't get us from corrupted to clean values. On the other hand, **noising** means taking a clean run and patching in with corrupted values, and it tests whether a component is **necessary**. Our "null hypothesis" is that a component isn't important, and so replacing its values with those on the corrupted input won't make the model worse. In this section, we'll be doing denoising (i.e. starting with corrupted values and patching in with clean values). In later sections, we'll be doing noising, and we'll define a new metric function for this.

▶ More on noising vs. denoising

```
def ioi_metric(
    logits: Float[Tensor, "batch seq d_vocab"],
    answer_tokens: Float[Tensor, "batch 2"] = answer_tokens,
    corrupted_logit_diff: float = corrupted_logit_diff,
    clean_logit_diff: float = clean_logit_diff,
) -> Float[Tensor, ""]:
    '''
    Linear function of logit diff, calibrated so that it equals 0 when performance is
    same as on corrupted input, and 1 when performance is same as on clean input.
    '''
    # SOLUTION
    patched_logit_diff = logits_to_ave_logit_diff(logits, answer_tokens)
    return (patched_logit_diff - corrupted_logit_diff) / (clean_logit_diff  - corrupted_logit_diff)


t.testing.assert_close(ioi_metric(clean_logits).item(), 1.0)
t.testing.assert_close(ioi_metric(corrupted_logits).item(), 0.0)
t.testing.assert_close(ioi_metric((clean_logits + corrupted_logits) / 2).item(), 0.5)
```

▶ Solution

Residual Stream Patching

Lets begin with a simple example: we patch in the residual stream at the start of each layer and for each token position. Before you write your own function to do this, let's see what this looks like with TransformerLens' `patching` module. Run the code below.

```
act_patch_resid_pre = patching.get_act_patch_resid_pre(
    model = model,
    corrupted_tokens = corrupted_tokens,
    clean_cache = clean_cache,
    patching_metric = ioi_metric
)

labels = [f"{tok} {i}" for i, tok in enumerate(model.to_str_tokens(clean_tokens[0]))]

imshow(
    act_patch_resid_pre,
    labels={"x": "Position", "y": "Layer"},
    x=labels,
    title="resid_pre Activation Patching",
    width=600
)
```

Question - what is the interpretation of this graph? What significant things does it tell you about the nature of how the model solves this task?

▶ Hint

▶ Answer

Exercise - implement head-to-residual patching

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

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

It's very important to understand how patching works. Many subsequent exercises will build on this one.
```

Now, you should implement the `get_act_patch_resid_pre` function below, which should behave just like the one above. A quick refresher on how to use hooks in this way:

- Hook functions take arguments `tensor: t.Tensor` and `hook: HookPoint`. It's often easier to define a hook function taking more arguments than these, and then use `functools.partial` when it actually comes time to add your hook.
- The function `model.run_with_hooks` takes arguments:
    - The tokens to run (as first argument)
    - `fwd_hooks` - a list of `(hook_name, hook_fn)` tuples. Remember that you can use `utils.get_act_name` to get hook names.
- Tip - it's good practice to have `model.reset_hooks()` at the start of functions which add and run hooks. This is because sometimes hooks fail to be removed (if they cause an error while running). There's nothing more frustrating than fixing a hook error only to get the same error message, not realising that you've failed to clear the broken hook!

```
def patch_residual_component(
    corrupted_residual_component: Float[Tensor, "batch pos d_model"],
    hook: HookPoint,
    pos: int,
    clean_cache: ActivationCache
) -> Float[Tensor, "batch pos d_model"]:
    '''
    Patches a given sequence position in the residual stream, using the value
    from the clean cache.
    '''
    # SOLUTION
    corrupted_residual_component[:, pos, :] = clean_cache[hook.name][:, pos, :]
    return corrupted_residual_component


def get_act_patch_resid_pre(
    model: HookedTransformer,
    corrupted_tokens: Float[Tensor, "batch pos"],
    clean_cache: ActivationCache,
    patching_metric: Callable[[Float[Tensor, "batch pos d_vocab"]], float]
) -> Float[Tensor, "layer pos"]:
    '''
    Returns an array of results of patching each position at each layer in the residual
    stream, using the value from the clean cache.

    The results are calculated using the patching_metric function, which should be
    called on the model's logit output.
    '''
    # SOLUTION
    model.reset_hooks()
    seq_len = corrupted_tokens.size(1)
    results = t.zeros(model.cfg.n_layers, seq_len, device="cuda", dtype=t.float32)

    for layer in tqdm(range(model.cfg.n_layers)):
        for position in range(seq_len):
            hook_fn = partial(patch_residual_component, pos=position, clean_cache=clean_cache)
            patched_logits = model.run_with_hooks(
                corrupted_tokens,
                fwd_hooks = [(utils.get_act_name("resid_pre", layer), hook_fn)],
            )
            results[layer, position] = patching_metric(patched_logits)

    return results



act_patch_resid_pre_own = get_act_patch_resid_pre(model, corrupted_tokens, clean_cache, ioi_metric)

t.testing.assert_close(act_patch_resid_pre, act_patch_resid_pre_own)
```

▶ Solution

Once you've passed the tests, you can plot your results.

```
imshow(
    act_patch_resid_pre_own,
    x=labels,
    title="Logit Difference From Patched Residual Stream",
    labels={"x":"Sequence Position", "y":"Layer"},
    width=600 # If you remove this argument, the plot will usually fill the available space
)
```

')

Patching in residual stream by block

Rather than just patching to the residual stream in each layer, we can also patch just after the attention layer or just after the MLP. This gives is a slightly more refined view of which tokens matter and when.

The function `patching.get_act_patch_block_every` works just like `get_act_patch_resid_pre`, but rather than just patching to the residual stream, it patches to `resid_pre`, `attn_out` and `mlp_out`, and returns a tensor of shape `(3, n_layers, seq_len)`.

One important thing to note - we're cycling through the `resid_pre`, `attn_out` and `mlp_out` and only patching one of them at a time, rather than patching all three at once.

```
act_patch_block_every = patching.get_act_patch_block_every(model, corrupted_tokens, clean_cache, ioi_metric)

imshow(
    act_patch_block_every,
    x=labels,
    facet_col=0, # This argument tells plotly which dimension to split into separate plots
    facet_labels=["Residual Stream", "Attn Output", "MLP Output"], # Subtitles of separate plots
    title="Logit Difference From Patched Attn Head Output",
    labels={"x": "Sequence Position", "y": "Layer"},
    width=1000,
)
```

▶ What is the interpretation of the second two plots?

Exercise (optional) - implement head-to-block patching

Difficulty: 🟠🟠⚪⚪⚪
Importance: 🟠🟠⚪⚪⚪

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

Most code can be copied from the last exercise.

If you want, you can implement the `get_act_patch_resid_pre` function for fun, although it's similar enough to the previous exercise that doing this isn't compulsory.

```
def get_act_patch_block_every(
    model: HookedTransformer,
    corrupted_tokens: Float[Tensor, "batch pos"],
    clean_cache: ActivationCache,
    patching_metric: Callable[[Float[Tensor, "batch pos d_vocab"]], float]
) -> Float[Tensor, "layer pos"]:
    '''
    Returns an array of results of patching each position at each layer in the residual
    stream, using the value from the clean cache.

    The results are calculated using the patching_metric function, which should be
    called on the model's logit output.
    '''
    # SOLUTION
    model.reset_hooks()
    results = t.zeros(3, model.cfg.n_layers, tokens.size(1), device="cuda", dtype=t.float32)

    for component_idx, component in enumerate(["resid_pre", "attn_out", "mlp_out"]):
        for layer in tqdm(range(model.cfg.n_layers)):
            for position in range(corrupted_tokens.shape[1]):
                hook_fn = partial(patch_residual_component, pos=position, clean_cache=clean_cache)
                patched_logits = model.run_with_hooks(
                    corrupted_tokens,
                    fwd_hooks = [(utils.get_act_name(component, layer), hook_fn)],
                )
                results[component_idx, layer, position] = patching_metric(patched_logits)

    return results
```

```
act_patch_block_every_own = get_act_patch_block_every(model, corrupted_tokens, clean_cache, ioi_metric)

t.testing.assert_close(act_patch_block_every, act_patch_block_every_own)

imshow(
    act_patch_block_every_own,
    x=labels,
    facet_col=0,
    facet_labels=["Residual Stream", "Attn Output", "MLP Output"],
    title="Logit Difference From Patched Attn Head Output",
    labels={"x": "Sequence Position", "y": "Layer"},
    width=1000
)
```

▶ Solution

Head Patching

We can refine the above analysis by patching in individual heads! This is somewhat more annoying, because there are now three dimensions `(head_index, position and layer)`.

The code below patches a head's output over all sequence positions, and returns the results (for each head in the model).

```
act_patch_attn_head_out_all_pos = patching.get_act_patch_attn_head_out_all_pos(
    model,
    corrupted_tokens,
    clean_cache,
    ioi_metric
)

imshow(
    act_patch_attn_head_out_all_pos,
    labels={"y": "Layer", "x": "Head"},
    title="attn_head_out Activation Patching (All Pos)",
    width=600
)
```

▶ What are the interpretations of this graph? Which heads do you think are important?

Exercise - implement head-to-head patching

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

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

Again, it should be similar to the first patching exercise (you can copy code).
```

You should implement your own version of this patching function below.

You'll need to define a new hook function, but most of the code from the previous exercise should be reusable.

▶ Help - I'm not sure what hook name to use for my patching.

```python
def patch_head_vector(
    corrupted_head_vector: Float[Tensor, "batch pos head_index d_head"],
    hook: HookPoint,
    head_index: int,
    clean_cache: ActivationCache
) -> Float[Tensor, "batch pos head_index d_head"]:
    '''
    Patches the output of a given head (before it's added to the residual stream) at
    every sequence position, using the value from the clean cache.
    '''
    # SOLUTION
    corrupted_head_vector[:, :, head_index] = clean_cache[hook.name][:, :, head_index]
    return corrupted_head_vector


def get_act_patch_attn_head_out_all_pos(
    model: HookedTransformer,
    corrupted_tokens: Float[Tensor, "batch pos"],
    clean_cache: ActivationCache,
    patching_metric: Callable
) -> Float[Tensor, "layer head"]:
    '''
    Returns an array of results of patching at all positions for each head in each
    layer, using the value from the clean cache.

    The results are calculated using the patching_metric function, which should be
    called on the model's logit output.
    '''
    # SOLUTION
    model.reset_hooks()
    results = t.zeros(model.cfg.n_layers, model.cfg.n_heads, device="cuda", dtype=t.float32)

    for layer in tqdm(range(model.cfg.n_layers)):
        for head in range(model.cfg.n_heads):
            hook_fn = partial(patch_head_vector, head_index=head, clean_cache=clean_cache)
            patched_logits = model.run_with_hooks(
                corrupted_tokens,
                fwd_hooks = [(utils.get_act_name("z", layer), hook_fn)],
                return_type="logits"
            )
            results[layer, head] = patching_metric(patched_logits)

    return results


act_patch_attn_head_out_all_pos_own = get_act_patch_attn_head_out_all_pos(model, corrupted_tokens, clean_cache, ioi_

t.testing.assert_close(act_patch_attn_head_out_all_pos, act_patch_attn_head_out_all_pos_own)
```

```
49
50 imshow(
51     act_patch_attn_head_out_all_pos_own,
52     title="Logit Difference From Patched Attn Head Output",
53     labels={"x":"Head", "y":"Layer"},
54     width=600
55 )
```

▶ Solution

Decomposing Heads

Finally, we'll look at one more example of activation patching.

Decomposing attention layers into patching in individual heads has already helped us localise the behaviour a lot. But we can understand it further by decomposing heads. An attention head consists of two semi-independent operations - calculating *where* to move information from and to (represented by the attention pattern and implemented via the QK-circuit) and calculating *what* information to move (represented by the value vectors and implemented by the OV circuit). We can disentangle which of these is important by patching in just the attention pattern *or* the value vectors. See A Mathematical Framework or Neel's walkthrough video for more on this decomposition.

A useful function for doing this is `get_act_patch_attn_head_all_pos_every`. Rather than just patching on head output (like the previous one), it patches on:

- Output (this is equivalent to patching the value the head writes to the residual stream)
- Querys (i.e. the patching the query vectors, without changing the key or value vectors)
- Keys
- Values
- Patterns (i.e. the attention patterns).

Again, note that this function isn't patching multiple things at once. It's looping through each of these five, and getting the results from patching them one at a time.

```
act_patch_attn_head_all_pos_every = patching.get_act_patch_attn_head_all_pos_every(
    model,
    corrupted_tokens,
    clean_cache,
    ioi_metric
)

imshow(
```

```
    act_patch_attn_head_all_pos_every,
    facet_col=0,
    facet_labels=["Output", "Query", "Key", "Value", "Pattern"],
    title="Activation Patching Per Head (All Pos)",
    labels={"x": "Head", "y": "Layer"},
)
```

Exercise (optional) - implement head-to-head-input patching

Difficulty: 🟠🟠⚪⚪⚪
Importance: 🟠🟠⚪⚪⚪

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

Most code can be copied from the last exercise.

Again, if you want to implement this yourself then you can do so below, but it isn't a compulsory exercise because it isn't conceptually different from the previous exercises. If you don't implement it, then you should still look at the solution to make sure you understand what's going on.

```
def patch_attn_patterns(
    corrupted_head_vector: Float[Tensor, "batch head_index pos_q pos_k"],
    hook: HookPoint,
    head_index: int,
    clean_cache: ActivationCache
) -> Float[Tensor, "batch pos head_index d_head"]:
    '''
    Patches the attn patterns of a given head at every sequence position, using
    the value from the clean cache.
    '''
    # SOLUTION
    corrupted_head_vector[:, head_index] = clean_cache[hook.name][:, head_index]
    return corrupted_head_vector


def get_act_patch_attn_head_all_pos_every(
```

```
    model: HookedTransformer,
    corrupted_tokens: Float[Tensor, "batch pos"],
    clean_cache: ActivationCache,
    patching_metric: Callable
) -> Float[Tensor, "layer head"]:
    '''
    Returns an array of results of patching at all positions for each head in each
    layer (using the value from the clean cache) for output, queries, keys, values
    and attn pattern in turn.

    The results are calculated using the patching_metric function, which should be
    called on the model's logit output.
    '''
    # SOLUTION
    results = t.zeros(5, model.cfg.n_layers, model.cfg.n_heads, device="cuda", dtype=t.float32)
    # Loop over each component in turn
    for component_idx, component in enumerate(["z", "q", "k", "v", "pattern"]):
        for layer in tqdm(range(model.cfg.n_layers)):
            for head in range(model.cfg.n_heads):
                # Get different hook function if we're doing attention probs
                hook_fn_general = patch_attn_patterns if component == "pattern" else patch_head_vector
                hook_fn = partial(hook_fn_general, head_index=head, clean_cache=clean_cache)
                # Get patched logits
                patched_logits = model.run_with_hooks(
                    corrupted_tokens,
                    fwd_hooks = [(utils.get_act_name(component, layer), hook_fn)],
                    return_type="logits"
                )
                results[component_idx, layer, head] = patching_metric(patched_logits)

    return results


act_patch_attn_head_all_pos_every_own = get_act_patch_attn_head_all_pos_every(
    model,
    corrupted_tokens,
    clean_cache,
    ioi_metric
)

t.testing.assert_close(act_patch_attn_head_all_pos_every, act_patch_attn_head_all_pos_every_own)

imshow(
    act_patch_attn_head_all_pos_every_own,
    facet_col=0,
    facet_labels=["Output", "Query", "Key", "Value", "Pattern"],
    title="Activation Patching Per Head (All Pos)",
    labels={"x": "Head", "y": "Layer"},
    width=1200
)
```

▶ Solution

Note - we can do this in an even more fine-grained way; the function `patching.get_act_patch_attn_head_by_pos_every` (i.e. same as above but replacing `all_pos` with `by_pos`) will give you the same decomposition, but by sequence position *as well as* by layer, head and component. The same holds for the `patching.get_act_patch_attn_head_out_all_pos` function earlier (replace `all_pos` with `by_pos`). These functions are unsurprisingly pretty slow though!

This plot has some striking features. For instance, this shows us that we have at least three different groups of heads:

- Earlier heads (`3.0, 5.5, 6.9`) which matter because of their attention patterns (specifically their query vectors).
- Middle heads in layers 7 & 8 (`7.3, 7.9, 8.6, 8.10`) seem to matter more because of their value vectors.
- Later heads which improve the logit difference (`9.9, 10.0`), which matter because of their query vectors.

Question - what is the significance of the results for the middle heads (i.e. the important ones in layers 7 & 8)? In particular, how should we interpret the fact that value patching has a much bigger effect than the other two forms of patching?

Hint - if you're confused, try plotting the attention patterns of heads `7.3, 7.9, 8.6, 8.10`. You can mostly reuse the code from above when we displayed the output of attention heads.

▶ Code to plot attention heads

▶ Answer

Consolidating Understanding

OK, let's zoom out and reconsolidate. Here's a recap of the most important observations we have so far:

- Heads `9.9, 9.6,` and `10.0` are the most important heads in terms of directly writing to the residual stream. In all these heads, the `END` attends strongly to the `IO`.
  - We discovered this by taking the values written by each head in each layer to the residual stream, and projecting them along the logit diff direction by using `residual_stack_to_logit_diff`. We also looked at attention patterns using `circuitsvis`.
  - **This suggests that these heads are copying `IO` to `end`, to use it as the predicted next token.**
  - The question then becomes *"how do these heads know to attend to this token, and not attend to `S`?"*

- All the action is on `S2` until layer 7 and then transitions to `END`. And that attention layers matter a lot, MLP layers not so much (apart from MLP0, likely as an extended embedding).
  - We discovered this by doing **activation patching** on `resid_pre, attn_out,` and `mlp_out`.
  - **This suggests that there is a cluster of heads in layers 7 & 8, which move information from `S2` to `END`. We deduce that this information is how heads `9.9, 9.6` and `10.0` know to attend to `IO`.**
  - The question then becomes *"what is this information, how does it end up in the `S2` token, and how does `END` know to attend to it?"*

- The significant heads in layers 7 & 8 are `7.3, 7.9, 8.6, 8.10`. These heads have high activation patching values for their value vectors, less so for their queries and keys.
  - We discovered this by doing **activation patching** on the value inputs for these heads.
  - **This supports the previous observation, and it tells us that the interesting computation goes into *what gets moved* from `S2` to `END`, rather than the fact that `END` attends to `S2`.**.
  - We still don't know: *"what is this information, and how does it end up in the `S2` token?"*

- As well as the 2 clusters of heads given above, there's a third cluster of important heads: early heads (e.g. `3.0, 5.5, 6.9`) whose query vectors are particularly important for getting good performance.

- We discovered this by doing **activation patching** on the query inputs for these heads.

With all this in mind, can you come up with a theory for what these three heads are doing, and come up with a simple model of the whole circuit?

Hint - if you're still stuck, try plotting the attention pattern of head `3.0`. The patterns of `5.5` and `6.9` might seem a bit confusing at first (they add complications to the "simplest possible picture" of how the circuit works); we'll discuss them later so they don't get in the way of understanding the core of the circuit.

▶ Answer (and simple diagram of circuit)

Now, let's flesh out this picture a bit more by comparing our results to the paper results. Below is a more complicated version of the diagram in the dropdown above, which also labels the important heads. The diagram is based on the paper's original diagram. Don't worry if you don't understand everything in this diagram; the boundaries of the circuit are fuzzy and the "role" of every head is in this circuit is a leaky abstraction. Rather, this diagram is meant to point your intuitions in the right direction for better understanding this circuit.

▶ Diagram of large circuit

Here are the main ways it differs from the one above:

Induction heads

Rather than just having duplicate token heads in the first cluster of heads, we have two other types of heads as well: previous token heads and induction heads. The induction heads do the same thing as the duplicate token heads, via an induction mechanism. They cause token `S2` to attend to `S1+1` (mediated by the previous token heads), and their output is used as both a pointer to `S1` and as a signal that `S1` is duplicated (more on the distinction between these two in the paragraph "Position vs token information being moved" below).

(Note - the original paper's diagram implies the induction heads and duplicate token heads compose with each other. This is misleading, and is not the case.)

Why are induction heads used in this circuit? We'll dig into this more in the bonus section, but one likely possibility is that induction heads are just a thing that forms very early on in training by default, and so it makes sense for the model to repurpose this already-existing machinery for this job. See this paper for more on induction heads, and how / why they form.

Negative & Backup name mover heads

Earlier, we saw that some heads in later layers were actually harming performance. These heads turn out to be doing something pretty similar to name mover heads, but in reverse (i.e. they inhibit the correct answer). It's not obvious why the model does this; the paper speculates that these heads might help the model "hedge" so as to avoid high cross-entropy loss when making mistakes.

Backup name mover heads are possibly even weirder. It turns out that when we **ablate** the name mover heads, these ones pick up the slack and do the task anyway (even though they don't seem to do it when the NMHs aren't ablated). This is an example of **built-in redundancy** in the model. One possible explanation is that this resulted from the model being trained with dropout, although this explanation isn't fully satisfying (models trained without dropout still seem to have BNMHs, although they aren't as strong as they are in this model). Like with induction heads, we'll dig into this more in the final section.

Positional vs token information

There are 2 kinds of S-inhibition heads shown in the diagram - ones that inhibit based on positional information (pink), and ones that inhibit based on token information (purple). It's not clear which heads are doing which (and in fact some heads might be doing both!).

The paper has an ingenious way of teasing apart which type of information is being used by which of the S-inhibition heads, which we'll discuss in the final section.

K-composition in S-inhibition heads

When we did activation patching on the keys and values of S-inhibition heads, we found that the values were important and the keys weren't. We concluded that K-composition isn't really happening in these heads, and `END` must be paying attention to `S2` for reasons other than the duplicate token information (e.g. it might just be paying attention to the closest name, or to any names which aren't separated from it by a comma). Although this is mostly true, it turns out that there is a bit of K-composition happening in these heads. We can think of this as the duplicate token heads writing the "duplicated" flag to the residual stream (without containing any information about the identity and position of this token), and this flag is being used by the keys of the S-inhibition heads (i.e. they make `END` pay attention to `S2`). In the diagram, this is represented by the dark grey boxes (rather than just the light grey boxes we had in the simplified version). We haven't seen any evidence for this happening yet, but we will in the next section (when we look at path patching).

Note - whether the early heads are writing positional information or "duplicate flag" information to the residual stream is not necessarily related to whether the head is an induction head or a duplicate token head. In principle, either type of head could write either type of information.

4️⃣ Path Patching

> Learning objectives
>
> - Understand the idea of path patching, and how it differs from activation patching
> - Implement path patching from scratch (i.e. using hooks)
> - Replicate several of the results in the IOI paper

This section will be a lot less conceptual and exploratory than the last two sections, and a lot more technical and rigorous. You'll learn what path patching is and how it works, and you'll use it to replicate many of the paper's results (as well as some other paper results not related to path patching).

Setup

Here, we'll be more closely following the setup that the paper's authors used, rather than the rough-and-ready exploration we used in the first few sections. To be clear, a lot of the rigour that we'll be using in the setup here isn't necessary if you're just starting to investigate a model's circuit. This rigour is necessary if you're publishing a paper, but it can take a lot of time and effort!

```
from part3_indirect_object_identification.ioi_dataset import NAMES, IOIDataset
```

The dataset we'll be using is an instance of `IOIDataset`, which is generated by randomly choosing names from the `NAMES` list (as well as sentence templates and objects from different lists). You can look at the `ioi_dataset.py` file to see details of how this is done.

(Note - you can reduce `N` if you're getting memory errors from running this code. If you're still getting memory errors from `N = 10` then you're recommended to switch to Colab, or to use a virtual machine e.g. via Lambda Labs.)

```
N = 25
ioi_dataset = IOIDataset(
    prompt_type="mixed",
    N=N,
    tokenizer=model.tokenizer,
    prepend_bos=False,
    seed=1,
    device=str(device)
)
```

This dataset has a few useful attributes & methods. Here are the main ones you should be aware of for these exercises:

- `toks` is a tensor of shape `(batch_size, max_seq_len)` containing the token IDs (i.e. this is what you pass to your model)
- `s_tokenIDs` and `io_tokenIDs` are lists containing the token IDs for the subjects and objects
- `sentences` is a list containing the sentences (as strings)
- `word_idx` is a dictionary mapping word types (e.g. `"S1"`, `"S2"`, `"IO"` or `"end"`) to tensors containing the positions of those words for each sequence in the dataset.
    - This is particularly handy for indexing, since the positions of the subject, indirect object, and end tokens are no longer the same in every sentence like they were in previous sections.

Firstly, what dataset should we use for patching? In the previous section we just flipped the subject and indirect object tokens around, which meant the direction of the signal was flipped around. However, what we'll be doing here is a bit more principled - rather than flipping the IOI signal, we'll be erasing it. We do this by constructing a new dataset from `ioi_dataset` which replaces every name with a different random name. This way, the sentence structure stays the same, but all information related to the actual indirect object identification task (i.e. the identities and positions of repeated names) has been erased.

For instance, given the sentence `"When John and Mary went to the shops, John gave the bag to Mary"`, the corresponding sentence in the ABC dataset might be `"When Edward and Laura went to the shops, Adam gave the bag to Mary"`. We would

expect the residual stream for the latter prompt to carry no token or positional information which could help it solve the IOI task (i.e. favouring `Mary` over `John`, or favouring the 2nd token over the 4th token).

We define this dataset below. Note the syntax of the `gen_flipped_prompts` method - the letters tell us how to replace the names in the sequence. For instance, `ABB->XYZ` tells us to take sentences of the form `"When Mary and John went to the store, John gave a drink to Mary"` with `"When [X] and [Y] went to the store, [Z] gave a drink to Mary"` for 3 independent randomly chosen names `[X]`, `[Y]` and `[Z]`. We'll use this function more in the bonus section, when we're trying to disentangle positional and token signals (since we can also do fun things like `ABB->BAB` to swap the first two names, etc).

```
abc_dataset = ioi_dataset.gen_flipped_prompts("ABB->XYZ, BAB->XYZ")
```

Let's take a look at this dataset. We'll define a helper function `make_table`, which prints out tables after being fed columns rather than rows (don't worry about the syntax, it's not important).

```
def format_prompt(sentence: str) -> str:
    '''Format a prompt by underlining names (for rich print)'''
    return re.sub("(" + "|".join(NAMES) + ")", lambda x: f"[u bold dark_orange]{x.group(0)}[/]", sentence) + "\n"


def make_table(cols, colnames, title="", n_rows=5, decimals=4):
    '''Makes and displays a table, from cols rather than rows (using rich print)'''
    table = Table(*colnames, title=title)
    rows = list(zip(*cols))
    f = lambda x: x if isinstance(x, str) else f"{x:.{decimals}f}"
    for row in rows[:n_rows]:
        table.add_row(*list(map(f, row)))
    rprint(table)
```

```
make_table(
    colnames = ["IOI prompt", "IOI subj", "IOI indirect obj", "ABC prompt"],
    cols = [
        map(format_prompt, ioi_dataset.sentences),
        model.to_string(ioi_dataset.s_tokenIDs).split(),
        model.to_string(ioi_dataset.io_tokenIDs).split(),
        map(format_prompt, abc_dataset.sentences),
    ],
    title = "Sentences from IOI vs ABC distribution",
)
```

Next, we'll define functions similar to the ones from previous sections. We've just given you these, rather than making you repeat the exercise of writing them (although you should compare these functions to the ones you wrote earlier, and make sure you understand how they work).

We'll call these functions something slightly different, so as not to pollute namespace.

```
def logits_to_ave_logit_diff_2(logits: Float[Tensor, "batch seq d_vocab"], ioi_dataset: IOIDataset = ioi_dataset, pe
    '''
    Returns logit difference between the correct and incorrect answer.
```

```
    If per_prompt=True, return the array of differences rather than the average.
    '''

    # Only the final logits are relevant for the answer
    # Get the logits corresponding to the indirect object / subject tokens respectively
    io_logits: Float[Tensor, "batch"] = logits[range(logits.size(0)), ioi_dataset.word_idx["end"], ioi_dataset.io_to
    s_logits: Float[Tensor, "batch"] = logits[range(logits.size(0)), ioi_dataset.word_idx["end"], ioi_dataset.s_toke
    # Find logit difference
    answer_logit_diff = io_logits - s_logits
    return answer_logit_diff if per_prompt else answer_logit_diff.mean()


model.reset_hooks(including_permanent=True)

ioi_logits_original, ioi_cache = model.run_with_cache(ioi_dataset.toks)
abc_logits_original, abc_cache = model.run_with_cache(abc_dataset.toks)

ioi_per_prompt_diff = logits_to_ave_logit_diff_2(ioi_logits_original, per_prompt=True)
abc_per_prompt_diff = logits_to_ave_logit_diff_2(abc_logits_original, per_prompt=True)

ioi_average_logit_diff = logits_to_ave_logit_diff_2(ioi_logits_original).item()
abc_average_logit_diff = logits_to_ave_logit_diff_2(abc_logits_original).item()
```

```
print(f"Average logit diff (IOI dataset): {ioi_average_logit_diff:.4f}")
print(f"Average logit diff (ABC dataset): {abc_average_logit_diff:.4f}")

make_table(
    colnames = ["IOI prompt", "IOI logit diff", "ABC prompt", "ABC logit diff"],
    cols = [
        map(format_prompt, ioi_dataset.sentences),
        ioi_per_prompt_diff,
        map(format_prompt, abc_dataset.sentences),
        abc_per_prompt_diff,
    ],
    title = "Sentences from IOI vs ABC distribution",
)
```

Note that we're always measuring performance ***with respect to the correct answers for the IOI dataset, not the ABC dataset***, because we want our ABC dataset to carry no information that helps with the IOI task (hence patching it in gives us signals which are totally uncorrelated with the correct answer). For instance, the model will obviously not complete sentences like `"When Max and Victoria got a snack at the store, Clark decided to give it to"` with the name `"Tyler"`.

Finally, let's define a new `ioi_metric` function which works for our new data.

In order to match the paper's results, we'll use a different convention here. 0 means performance is the same as on the IOI dataset (i.e. hasn't been harmed in any way), and -1 means performance is the same as on the ABC dataset (i.e. the model has completely lost the ability to distinguish between the subject and indirect object).

Again, we'll call this function something slightly different.

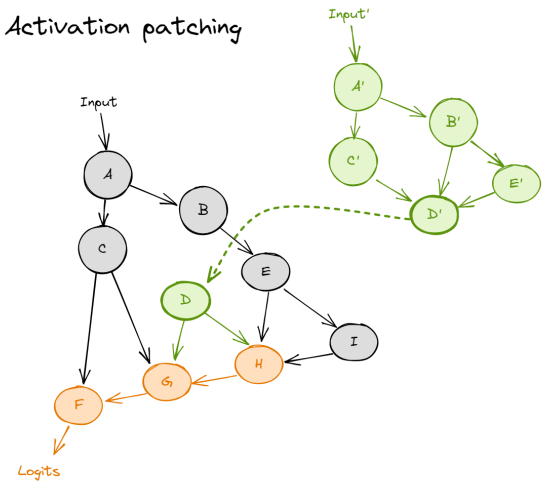
```
def ioi_metric_2(
    logits: Float[Tensor, "batch seq d_vocab"],
    clean_logit_diff: float = ioi_average_logit_diff,
    corrupted_logit_diff: float = abc_average_logit_diff,
    ioi_dataset: IOIDataset = ioi_dataset,
) -> float:
    '''
    We calibrate this so that the value is 0 when performance isn't harmed (i.e. same as IOI dataset),
    and -1 when performance has been destroyed (i.e. is same as ABC dataset).
    '''
    patched_logit_diff = logits_to_ave_logit_diff_2(logits, ioi_dataset)
    return (patched_logit_diff - clean_logit_diff) / (clean_logit_diff - corrupted_logit_diff)


print(f"IOI metric (IOI dataset): {ioi_metric_2(ioi_logits_original):.4f}")
print(f"IOI metric (ABC dataset): {ioi_metric_2(abc_logits_original):.4f}")
```

What is path patching?

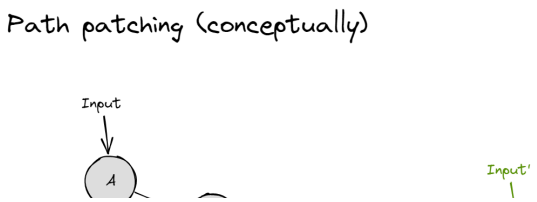
In the previous section, we looked at activation patching, which answers questions like *what would happen if you took an attention head, and swapped the value it writes to the residual stream with the value it would have written under a different distribution, while keeping everything else the same?*. This proved to be a good way to examine the role of individual components like attention heads, and it allowed us to perform some more subtle analysis like patching keys / queries / values in turn to figure out which of them were more important for which heads.

However, when we're studying a circuit, rather than just swapping out an entire attention head, we might want to ask more nuanced questions like *what would happen if the direct input from attention head A to head B (where B comes after A) was swapped out with the value it would have been under a different distribution, while keeping everything else the same?*. Rather than answering the general question of how important attention heads are, this answers the more specific question of how important the circuit formed by connecting up these two attention heads is. Path patching is designed to answer questions like these.

The following diagrams might help explain the difference between activation and path patching in transformers. Recall that activation patching looked like:

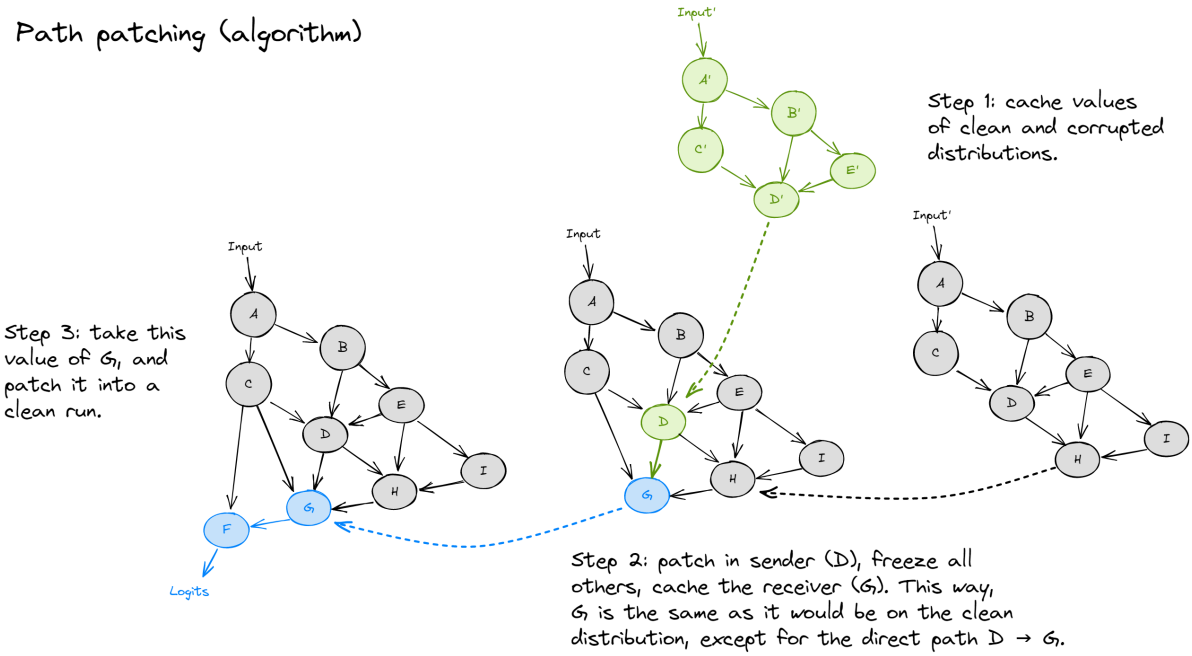


where the black and green distributions are our clean and corrupted datasets respectively (so this would be `ioi_dataset` and `abc_dataset`). In contrast, path patching involves replacing **edges** rather than **nodes**. In the diagram below, we're replacing the edge $D \to G$ with what it would be on the corrupted distribution. So in our patched run, G is calculated just like it would be on the clean distribution, but as if the **direct** input from D had come from the corrupted distribution instead.



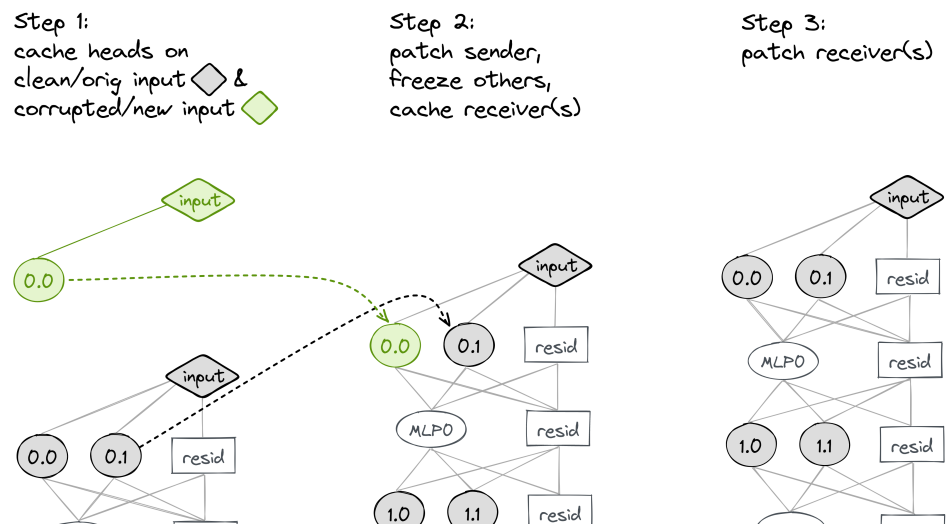
Unfortunately, for a transformer, this is easier to describe than to actually implement. This is because the "nodes" are attention heads, and the "edges" are all tangled together in the residual stream (that is to say, it's not clear how one could change the value of one edge without without affecting every path that includes that edge). The solution is to use the 3-step algorithm shown in the diagram below (which reads from right to left).

Terminology note - we call head D the **sender node**, and head G the **receiver node**. Also, by "freezing" nodes, we mean "patch with the value that is the same as the input". For instance, if we didn't freeze head H in step 2 below, it would have a different value because it would be affected by the corrupted value of head D.

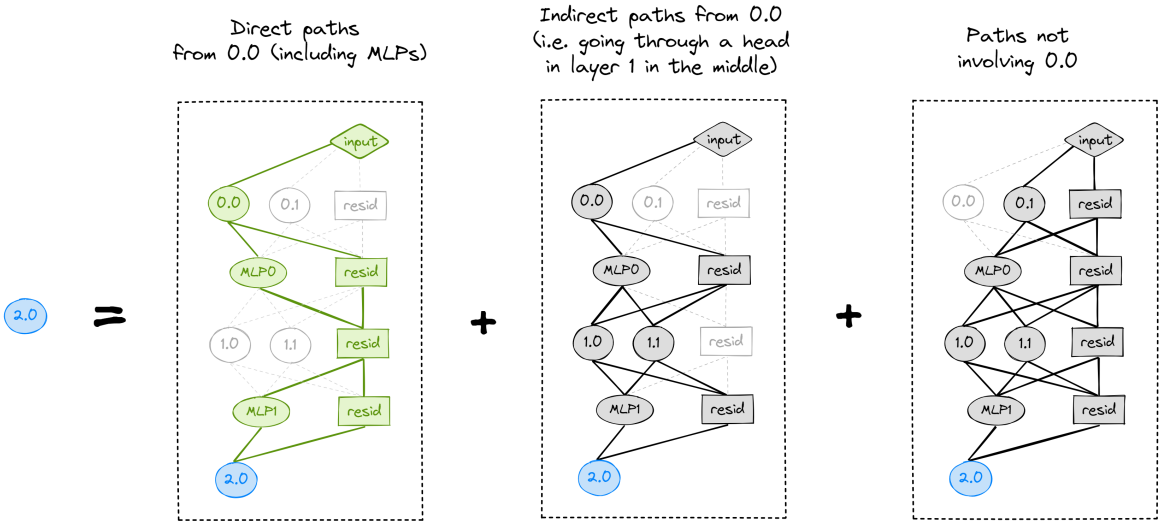


Let's make this concrete, and take a simple 3-layer transformer with 2 heads per layer. Let's perform path patching on the edge from head `0.0` to `2.0` (terminology note: `0.0` is the **sender**, and `2.0` is the **receiver**). Note that here, we're considering "direct paths" as anything that doesn't go through another attention head (so it can go through any combination of MLPs). Intuitively, the nodes (attention heads) are the only things that can move information around in the model, and this is the thing we want to study. In contrast, MLPs just perform information processing, and they're not as interesting for this task.

Our 3-step process looks like the diagram below (remember green is corrupted, grey is clean).



Why does this work? If you stare at the middle picture above for long enough, you'll realise that the contribution from every non-direct path from `0.0` → `2.0` is the same as it would be on the clean distribution, while all the direct paths' contributions are the same as they would be on the corrupted distribution.



Why MLPs?

You might be wondering why we're including MLPs as part of our direct path. The short answer is that this is what the IOI paper does, and we're trying to replicate it! The slightly longer answer is that both this method and a method which doesn't count MLPs as the direct path are justifiable.

To take one example, suppose the output of head `0.0` is being used directly by head `2.0`, but one of the MLPs is acting as a mediator. To oversimplify, we might imagine that `0.0` writes the vector v into the residual stream, some neuron detects v and writes w to the residual stream, and `2.0` detects w. If we didn't count MLPs as a direct path then we wouldn't catch this causal relationship. The drawback is that things get a bit messier, because now we're essentially passing a "fake input" into our MLPs, and it's dangerous to assume that any operation as clean as the one previously described (with vectors v, w) would still happen under these new circumstances.

Also, having MLPs as part of the direct path doesn't help us understand what role the MLPs play in the circuit, all it does is tell us that some of them are important! Luckily, in the IOI circuit, MLPs aren't important (except for MLP0), and so doing both these forms of path patching get pretty similar results. As an optional exercise, you can reproduce the results from the following few sections using this different form of path patching. It's actually algorithmically easier to implement, because we only need one forward pass rather than two. Can you see why?

▶ Answer

Path Patching: Name Mover Heads

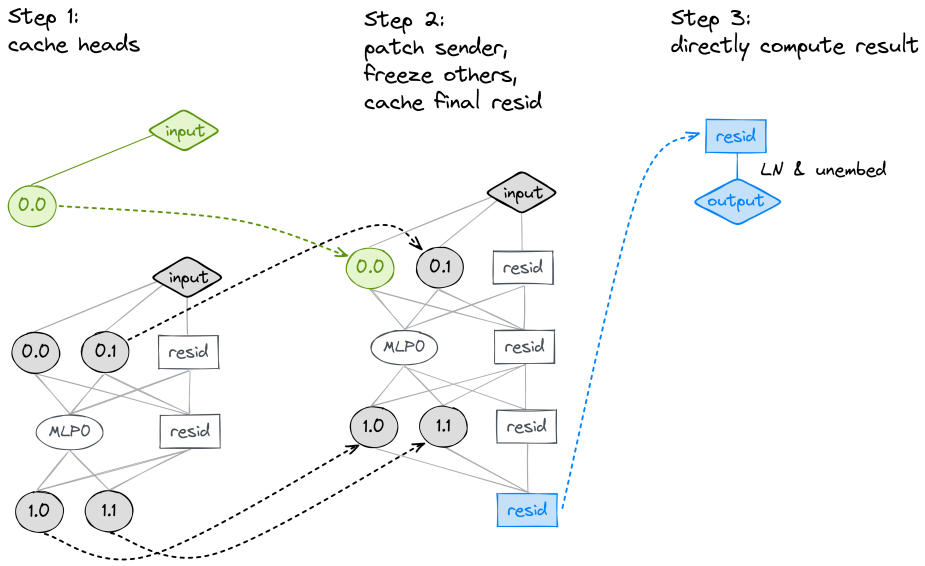
We'll start with a simple type of path patching - with just one receiver node, which is the final value of the residual stream. We've only discussed receiver nodes being other attention heads so far, but the same priciples hold for any choice of receiver nodes.

▶ Question - can you explain the difference between path patching from an attention head to the residual stream, and activation patching on that attention head?

This patching is described at the start of section 3.1 in the paper (page 5). The 3-step process will look like:

1. Run the model on clean and corrupted input. Cache the head outputs.
2. Run the model on clean input, with the sender head **patched** from the corrupted input, and every other head **frozen** to their values on the clean input. Cache the final value of the residual stream (i.e. `resid_post` in the final layer).
3. Normally we would re-run the model on the clean input and patch in the cached value of the final residual stream, but in this case we don't need to because we can just unembed the final value of the residual stream directly without having to run another forward pass.

Here is an illustration for a 2-layer transformer:



Exercise - implement path patching to the final residual stream value

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

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

```
Path patching is a very challenging algorithm with many different steps.
```

You should implement path patching from heads to the residual stream, as described above (and in the paper).

This exercise is expected to be challenging, with several moving parts. We've purposefully left it very open-ended, without even giving you a docstring for the function you'll be writing.

Here are a few hints / tips for how to proceed:

- Split your function up into 3 parts (one for each of the steps above), and write each section one at a time.
- You'll need a new hook function: one which performs freezing / patching for step 2 of the algorithm.
- You can reuse a lot of code from your activation patching function.
- When calling `model.run_with_cache`, you can use the keyword argument `names_filter`, which is a function from name to boolean. If you use this argument, your model will only cache activtions with a name which passes this filter (e.g. you can use it like `names_filter = lambda name: name.endswith("q")` to only cache query vectors).

You can also look at the dropdowns to get more hints and guidance (e.g. if you want to start from a function docstring).

You'll know you've succeeded if you can plot the results, and replicate Figure 3(b) from the paper (at the top of page 6).

Note - if you use `model.add_hook` then `model.run_with_cache`, you might have to pass the argument `level=1` to the `add_hook` method. I don't know why the function sometimes fails unless you do this (this bug only started appearing after the exercises were written). I've not had time to track this down, but extra credit to anyone who can (-:

▶ Click here to get a docstring for the main function.

▶ Click here to get a docstring for the main function, plus some annotations and function structure.

```
def patch_or_freeze_head_vectors(
    orig_head_vector: Float[Tensor, "batch pos head_index d_head"],
```

```
    hook: HookPoint,
    new_cache: ActivationCache,
    orig_cache: ActivationCache,
    head_to_patch: Tuple[int, int],
) -> Float[Tensor, "batch pos head_index d_head"]:
    '''
    This helps implement step 2 of path patching. We freeze all head outputs (i.e. set them
    to their values in orig_cache), except for head_to_patch (if it's in this layer) which
    we patch with the value from new_cache.

    head_to_patch: tuple of (layer, head)
        we can use hook.layer() to check if the head to patch is in this layer
    '''
    # Setting using ..., otherwise changing orig_head_vector will edit cache value too
    orig_head_vector[...] = orig_cache[hook.name][...]
    if head_to_patch[0] == hook.layer():
        orig_head_vector[:, :, head_to_patch[1]] = new_cache[hook.name][:, :, head_to_patch[1]]
    return orig_head_vector



def get_path_patch_head_to_final_resid_post(
    model: HookedTransformer,
    patching_metric: Callable,
    new_dataset: IOIDataset = abc_dataset,
    orig_dataset: IOIDataset = ioi_dataset,
    new_cache: Optional[ActivationCache] = abc_cache,
    orig_cache: Optional[ActivationCache] = ioi_cache,
) -> Float[Tensor, "layer head"]:
    # SOLUTION
    '''
    Performs path patching (see algorithm in appendix B of IOI paper), with:

        sender head = (each head, looped through, one at a time)
        receiver node = final value of residual stream

    Returns:
        tensor of metric values for every possible sender head
    '''
    model.reset_hooks()
    results = t.zeros(model.cfg.n_layers, model.cfg.n_heads, device="cuda", dtype=t.float32)

    resid_post_hook_name = utils.get_act_name("resid_post", model.cfg.n_layers - 1)
    resid_post_name_filter = lambda name: name == resid_post_hook_name


    # ========== Step 1 ==========
    # Gather activations on x_orig and x_new

    # Note the use of names_filter for the run_with_cache function. Using it means we
    # only cache the things we need (in this case, just attn head outputs).
    z_name_filter = lambda name: name.endswith("z")
    if new_cache is None:
        _, new_cache = model.run_with_cache(
            new_dataset.toks,
            names_filter=z_name_filter,
            return_type=None
        )
    if orig_cache is None:
        _, orig_cache = model.run_with_cache(
            orig_dataset.toks,
            names_filter=z_name_filter,
            return_type=None
        )


    # Looping over every possible sender head (the receiver is always the final resid_post)
    # Note use of itertools (gives us a smoother progress bar)
    for (sender_layer, sender_head) in tqdm(list(itertools.product(range(model.cfg.n_layers), range(model.cfg.n_head

        # ========== Step 2 ==========
        # Run on x_orig, with sender head patched from x_new, every other head frozen

        hook_fn = partial(
            patch_or_freeze_head_vectors,
            new_cache=new_cache,
            orig_cache=orig_cache,
            head_to_patch=(sender_layer, sender_head),
        )
```

```
        model.add_hook(z_name_filter, hook_fn)

        _, patched_cache = model.run_with_cache(
            orig_dataset.toks,
            names_filter=resid_post_name_filter,
            return_type=None
        )
        # if (sender_layer, sender_head) == (9, 9):
        #     return patched_cache
        assert set(patched_cache.keys()) == {resid_post_hook_name}

        # ========== Step 3 ==========
        # Unembed the final residual stream value, to get our patched logits

        patched_logits = model.unembed(model.ln_final(patched_cache[resid_post_hook_name]))

        # Save the results
        results[sender_layer, sender_head] = patching_metric(patched_logits)

    return results


path_patch_head_to_final_resid_post = get_path_patch_head_to_final_resid_post(model, ioi_metric_2)

imshow(
    100 * path_patch_head_to_final_resid_post,
    title="Direct effect on logit difference",
    labels={"x":"Head", "y":"Layer", "color": "Logit diff. variation"},
    coloraxis=dict(colorbar_ticksuffix = "%"),
    width=600,
)
```

▶ Solution

What is the interpretation of this plot? How does it compare to the equivalent plot we got from activation patching? (Remember that our metric is defined in a different way, so we should expect a sign difference between the two results.)

▶ Some thoughts

Path Patching: S-Inhibition Heads

In the first section on path patching, we performed a simple kind of patching - from the output of an attention head to the final value of the residual stream. Here we'll do something a bit more interesting, and patch from the output of one head to the input of a later head. The

purpose of this is to examine exactly how two heads are composing, and what effect the composed heads have on the model's output.

We got a hint of this in the previous section, where we patched the values of the S-inhibition heads and found that they were important. But this didn't tell us which inputs to these value vectors were important; we had to make educated guesses about this based on our analysis earlier parts of the model. In path patching, we can perform a more precise test to find which heads are important.

The paper's results from path patching are shown in figure 5(b), on page 7.

Exercise - implement path patching from head to head

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

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

You'll need a new hook function, but copying code from the previous exercise should make this one easier.
```

You should fill in the function `get_path_patch_head_to_head` below. It takes as arguments a list of receiver nodes (as well as the type of input - keys, queries, or values), and returns a tensor of shape* `(layer, head)` where each element is the result of running the patching metric on the output of the model, after applying the 3-step path patching algorithm from one of the model's heads to all the receiver heads. You should be able to replicate the paper's results (figure 5(b)).

**Actually, you don't need to return all layers, because the causal effect from any sender head which is on the same or a later layer than the last of your receiver heads will necessarily be zero.*

If you want a bit more guidance, you can use the dropdown below to see the ways in which this function should be different from your first path patching function (in most ways these functions will be similar, so you can start by copying that function).

▶ Differences from first path patching function

```python
def patch_head_input(
    orig_activation: Float[Tensor, "batch pos head_idx d_head"],
    hook: HookPoint,
    patched_cache: ActivationCache,
    head_list: List[Tuple[int, int]],
) -> Float[Tensor, "batch pos head_idx d_head"]:
    '''
    Function which can patch any combination of heads in layers,
    according to the heads in head_list.
    '''
    heads_to_patch = [head for layer, head in head_list if layer == hook.layer()]
    orig_activation[:, :, heads_to_patch] = patched_cache[hook.name][:, :, heads_to_patch]
    return orig_activation



def get_path_patch_head_to_heads(
    receiver_heads: List[Tuple[int, int]],
    receiver_input: str,
    model: HookedTransformer,
    patching_metric: Callable,
    new_dataset: IOIDataset = abc_dataset,
    orig_dataset: IOIDataset = ioi_dataset,
    new_cache: Optional[ActivationCache] = None,
    orig_cache: Optional[ActivationCache] = None,
) -> Float[Tensor, "layer head"]:
    '''
    Performs path patching (see algorithm in appendix B of IOI paper), with:

        sender head = (each head, looped through, one at a time)
        receiver node = input to a later head (or set of heads)

    The receiver node is specified by receiver_heads and receiver_input.
    Example (for S-inhibition path patching the queries):
        receiver_heads = [(8, 6), (8, 10), (7, 9), (7, 3)],
        receiver_input = "v"

    Returns:
        tensor of metric values for every possible sender head
    '''
    # SOLUTION
    model.reset_hooks()

    assert receiver_input in ("k", "q", "v")
```

```
    assert receiver_input in ("k", "q", "v")
    receiver_layers = set(next(zip(*receiver_heads)))
    receiver_hook_names = [utils.get_act_name(receiver_input, layer) for layer in receiver_layers]
    receiver_hook_names_filter = lambda name: name in receiver_hook_names

    results = t.zeros(max(receiver_layers), model.cfg.n_heads, device="cuda", dtype=t.float32)

    # ========== Step 1 ==========
    # Gather activations on x_orig and x_new

    # Note the use of names_filter for the run_with_cache function. Using it means we
    # only cache the things we need (in this case, just attn head outputs).
    z_name_filter = lambda name: name.endswith("z")
    if new_cache is None:
        _, new_cache = model.run_with_cache(
            new_dataset.toks,
            names_filter=z_name_filter,
            return_type=None
        )
    if orig_cache is None:
        _, orig_cache = model.run_with_cache(
            orig_dataset.toks,
            names_filter=z_name_filter,
            return_type=None
        )

    # Note, the sender layer will always be before the final receiver layer, otherwise there will
    # be no causal effect from sender -> receiver. So we only need to loop this far.
    for (sender_layer, sender_head) in tqdm(list(itertools.product(
        range(max(receiver_layers)),
        range(model.cfg.n_heads)
    ))):

        # ========== Step 2 ==========
        # Run on x_orig, with sender head patched from x_new, every other head frozen

        hook_fn = partial(
            patch_or_freeze_head_vectors,
            new_cache=new_cache,
            orig_cache=orig_cache,
            head_to_patch=(sender_layer, sender_head),
        )
        model.add_hook(z_name_filter, hook_fn, level=1)

        _, patched_cache = model.run_with_cache(
            orig_dataset.toks,
            names_filter=receiver_hook_names_filter,
            return_type=None
        )
        # model.reset_hooks(including_permanent=True)
        assert set(patched_cache.keys()) == set(receiver_hook_names)

        # ========== Step 3 ==========
        # Run on x_orig, patching in the receiver node(s) from the previously cached value

        hook_fn = partial(
            patch_head_input,
            patched_cache=patched_cache,
            head_list=receiver_heads,
        )
        patched_logits = model.run_with_hooks(
            orig_dataset.toks,
            fwd_hooks = [(receiver_hook_names_filter, hook_fn)],
            return_type="logits"
        )

        # Save the results
        results[sender_layer, sender_head] = patching_metric(patched_logits)

    return results
```

```
model.reset_hooks()

s_inhibition_value_path_patching_results = get_path_patch_head_to_heads(
    receiver_heads = [(8, 6), (8, 10), (7, 9), (7, 3)],
    receiver_input = "v",
    model = model,
    patching_metric = ioi_metric_2
)
```

```
)

imshow(
    100 * s_inhibition_value_path_patching_results,
    title="Direct effect on S-Inhibition Heads' values",
    labels={"x": "Head", "y": "Layer", "color": "Logit diff.<br>variation"},
    width=600,
    coloraxis=dict(colorbar_ticksuffix = "%"),
)
```

▶ Question - what is the interpretation of this plot?

▶ Solution

5️⃣ Paper Replication

> Learning objectives
>
> - Replicate most of the other results from the IOI paper
> - Practice more open-ended, less guided coding

This section will be a lot more open-ended and challenging. You'll be given less guidance in the exercises.

Copying & writing direction results

We'll start this section by replicating the paper's analysis of the **name mover heads** and **negative name mover heads**. Our previous analysis should have pretty much convinced us that these heads are copying / negatively copying our indirect object token, but the results here show this with a bit more rigour.

Exercise - replicate writing direction results

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

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

These exercises are much more challenging than they are conceptually important.
```

Let's look at figure 3(c) from the paper. This plots the output of the strongest name mover and negative name mover heads against the attention probabilities for `END` attending to `IO` or `S` (color-coded).

Some clarifications:

- "Projection" here is being used synonymously with "dot product".
- We're projecting onto the name embedding, i.e. the embedding vector for the token which is being paid attention to. This is not the same as the logit diff (which we got by projecting the heads' output onto the difference between the unembedding vectors for `IO` and `S`).
    - We're doing this because the question we're trying to answer is *"does the attention head copy (or anti-copy) the names which it pays attention to?"*

You should write code to replicate the paper's results in the cells below. Given four 1D tensors storing the results for a particular head (i.e. the projections and attention probabilities, for the `IO` and `S` tokens respectively), we've given you code to generate a plot which looks like the one in the paper. Again, you'll know that your code has worked if you can get results that resemble those in the paper.

```
def scatter_embedding_vs_attn(
    attn_from_end_to_io: Float[Tensor, "batch"],
    attn_from_end_to_s: Float[Tensor, "batch"],
    projection_in_io_dir: Float[Tensor, "batch"],
    projection_in_s_dir: Float[Tensor, "batch"],
    layer: int,
    head: int
):
    scatter(
        x=t.concat([attn_from_end_to_io, attn_from_end_to_s], dim=0),
        y=t.concat([projection_in_io_dir, projection_in_s_dir], dim=0),
        color=["IO"] * N + ["S"] * N,
        title=f"Projection of the output of {layer}.{head} along the name<br>embedding vs attention probability on r
        title_x=0.5,
        labels={"x": "Attn prob on name", "y": "Dot w Name Embed", "color": "Name type"},
        color_discrete_sequence=["#72FF64", "#C9A5F7"],
        width=650
    )
```

```
def calculate_and_show_scatter_embedding_vs_attn(
    layer: int,
    head: int,
    cache: ActivationCache = ioi_cache,
    dataset: IOIDataset = ioi_dataset,
) -> None:
    '''
    Creates and plots a figure equivalent to 3(c) in the paper.

    This should involve computing the four 1D tensors:
        attn_from_end_to_io
        attn_from_end_to_s
        projection_in_io_dir
        projection_in_s_dir
    and then calling the scatter_embedding_vs_attn function.
    '''
    # SOLUTION
    # Get the value written to the residual stream at the end token by this head
    z: Float[Tensor, "batch seq d_head"] = cache[utils.get_act_name("z", layer)][:, :, head]
    N = z.size(0)
    output: Float[Tensor, "batch seq d_model"] = z @ model.W_O[layer, head]
    output_on_end_token: Float[Tensor, "batch d_model"] = output[t.arange(N), dataset.word_idx["end"]]

    # Get the directions we'll be projecting onto
    io_unembedding: Float[Tensor, "batch d_model"] = model.W_U.T[dataset.io_tokenIDs]
    s_unembedding: Float[Tensor, "batch d_model"] = model.W_U.T[dataset.s_tokenIDs]

    # Get the value of projections, by multiplying and summing over the d_model dimension
    projection_in_io_dir: Float[Tensor, "batch"] = (output_on_end_token * io_unembedding).sum(-1)
    projection_in_s_dir: Float[Tensor, "batch"] = (output_on_end_token * s_unembedding).sum(-1)

    # Get attention probs, and index to get the probabilities from END -> IO / S
    attn_probs: Float[Tensor, "batch q k"] = cache["pattern", layer][:, head]
    attn_from_end_to_io = attn_probs[t.arange(N), dataset.word_idx["end"], dataset.word_idx["IO"]]
    attn_from_end_to_s = attn_probs[t.arange(N), dataset.word_idx["end"], dataset.word_idx["S1"]]

    # Show scatter plot
    scatter_embedding_vs_attn(
```

```
        attn_from_end_to_io,
        attn_from_end_to_s,
        projection_in_io_dir,
        projection_in_s_dir,
        layer,
        head
    )


nmh = (9, 9)
calculate_and_show_scatter_embedding_vs_attn(*nmh)

nnmh = (11, 10)
calculate_and_show_scatter_embedding_vs_attn(*nnmh)
```

▶ Solution

▶ Question - what is the interpretation of this plot in the paper?

Exercise - replicate copying score results

Difficulty: 🟠🟠🟠🟠🟠

Importance: 🟠🟠⚪⚪⚪

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

These exercises are much more challenging than they are conceptually important.
```

Now let's do a different kind of test of the name mover heads' copying, by looking directly at the OV circuits.

From page 6 of the paper:

> To check that the Name Mover Heads copy names generally, we studied what values are written via the heads' OV matrix. Specifically, we first obtained the state of the residual stream at the position of each name token after the first MLP layer. Then, we multiplied this by the OV matrix of a Name Mover Head (simulating what would happen if the head attended perfectly to that token), multiplied by the unembedding matrix, and applied the final layer norm to obtain logit probabilities. We compute the proportion of samples that contain the input name token in the top 5 logits (N = 1000) and call this the copy score. All three Name Mover Heads have a copy score above 95%, compared to less than 20% for an average head.
>
> Negative Name Mover Heads ... have a large negative copy score–the copy score calculated with the negative of the OV matrix (98% compared to 12% for an average head).

Note the similarity between their method and how we studied copying in induction heads, during an earlier set of exercises. However, there are differences (e.g. we're only looking at whether the head copies names, rather than whether it copies tokens in general).

You should replicate these results by completing the `get_copying_scores` function below.

You could do this by indexing from the `ioi_cache`, but a much more principled alternative would be to embed all the names in the `NAMES` list and apply operations like MLPs, layernorms and OV matrices manually. This is what the solutions do.

A few notes:

- You can use `model.to_tokens` to convert the names to tokens. Remember to use `prepend_bos=False`, since you just want the tokens of names so you can embed them. Note that this function will treat the list of names as a batch of single-token inputs, which works fine for our purposes.
- You can apply MLPs and layernorms as functions, by just indexing the model's blocks (e.g. use `model.blocks[i].mlp` or `model.blocks[j].ln1` as a function). Remember that `ln1` is the layernorm that comes before attention, and `ln2` comes before the MLP.
- Remember that you need to apply MLP0 before you apply the OV matrix (which is why we omit the 0th layer in our scores). The reason for this is that ablating MLP0 has a strangely large effect in gpt2-small relative to ablating other MLPs, possibly because it's acting as an extended embedding (see here for an explanation).

Also, you shouldn't expect to get exactly the same results as the paper (because some parts of this experiment have been set up very slightly different), although you probably shouldn't be off by more than 10%.

```python
def get_copying_scores(
    model: HookedTransformer,
    k: int = 5,
    names: list = NAMES
) -> Float[Tensor, "2 layer-1 head"]:
    '''
    Gets copying scores (both positive and negative) as described in page 6 of the IOI paper, for every (layer, head

    Returns these in a 3D tensor (the first dimension is for positive vs negative).

    Omits the 0th layer, because this is before MLP0 (which we're claiming acts as an extended embedding).
    '''
    # SOLUTION
    results = t.zeros((2, model.cfg.n_layers, model.cfg.n_heads), device="cuda")

    # Define components from our model (for typechecking, and cleaner code)
    embed: Embed = model.embed
    mlp0: MLP = model.blocks[0].mlp
    ln0: LayerNorm = model.blocks[0].ln2
    unembed: Unembed = model.unembed
    ln_final: LayerNorm = model.ln_final

    # Get embeddings for the names in our list
    name_tokens: Int[Tensor, "batch 1"] = model.to_tokens(names, prepend_bos=False)
    name_embeddings: Int[Tensor, "batch 1 d_model"] = embed(name_tokens)

    # Get residual stream after applying MLP
    resid_after_mlp1 = name_embeddings + mlp0(ln0(name_embeddings))

    # Loop over all (layer, head) pairs
```

```
    for layer in range(1, model.cfg.n_layers):
        for head in range(model.cfg.n_heads):

            # Get W_OV matrix
            W_OV = model.W_V[layer, head] @ model.W_O[layer, head]

            # Get residual stream after applying W_OV or -W_OV respectively
            # (note, because of bias b_U, it matters that we do sign flip here, not later)
            resid_after_OV_pos = resid_after_mlp1 @ W_OV
            resid_after_OV_neg = resid_after_mlp1 @ -W_OV

            # Get logits from value of residual stream
            logits_pos: Float[Tensor, "batch d_vocab"] = unembed(ln_final(resid_after_OV_pos)).squeeze()
            logits_neg: Float[Tensor, "batch d_vocab"] = unembed(ln_final(resid_after_OV_neg)).squeeze()

            # Check how many are in top k
            topk_logits: Int[Tensor, "batch k"] = t.topk(logits_pos, dim=-1, k=k).indices
            in_topk = (topk_logits == name_tokens).any(-1)
            # Check how many are in bottom k
            bottomk_logits: Int[Tensor, "batch k"] = t.topk(logits_neg, dim=-1, k=k).indices
            in_bottomk = (bottomk_logits == name_tokens).any(-1)

            # Fill in results
            results[:, layer-1, head] = t.tensor([in_topk.float().mean(), in_bottomk.float().mean()])

    return results
```

```
copying_results = get_copying_scores(model)

imshow(
    copying_results,
    facet_col=0,
    facet_labels=["Positive copying scores", "Negative copying scores"],
    title="Copying scores of attention heads' OV circuits",
    width=800
)
```

```
heads = {"name mover": [(9, 9), (10, 0), (9, 6)], "negative name mover": [(10, 7), (11, 10)]}

for i, name in enumerate(["name mover", "negative name mover"]):
    make_table(
        title=f"Copying Scores ({name} heads)",
        colnames=["Head", "Score"],
        cols=[
            list(map(str, heads[name])) + ["[dark_orange bold]Average"],
            [f"{copying_results[i, layer-1, head]:.2%}" for (layer, head) in heads[name]] + [f"[dark_orange bold]{cc
        ]
    )
```

▶ Solution

Validation of early heads

There are three different kinds of heads which appear early in the circuit, which can be validated by looking at their attention patterns on simple random sequences of tokens. Can you figure out which three types these are, and how to validate them in this way?

▶ Answer

Exercise - perform head validation

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

You should spend up to 20-30 minutes on this exercise.

Understanding how to identify certain types of heads by their characteristic attention patterns is important.
```

Once you've read the answer in the dropdown above, you should perform this validation. The result should be a replication of Figure 18 in the paper (don't look at this figure until you've attempted the question above, because it will give away the answer!).

We've provided a template for this function. Note use of `typing.Literal`, which is how we indicate that the argument should be one of the following options.

```python
def generate_repeated_tokens(
    model: HookedTransformer,
    seq_len: int,
    batch: int = 1
) -> Float[Tensor, "batch 2*seq_len"]:
    '''
    Generates a sequence of repeated random tokens (no start token).
    '''
    rep_tokens_half = t.randint(0, model.cfg.d_vocab, (batch, seq_len), dtype=t.int64)
    rep_tokens = t.cat([rep_tokens_half, rep_tokens_half], dim=-1).to(device)
    return rep_tokens



def get_attn_scores(
    model: HookedTransformer,
    seq_len: int,
    batch: int,
    head_type: Literal["duplicate", "prev", "induction"]
):
    '''
    Returns attention scores for sequence of duplicated tokens, for every head.
    '''
    # SOLUTION
    rep_tokens = generate_repeated_tokens(model, seq_len, batch)

    _, cache = model.run_with_cache(
        rep tokens,
```

```
        return_type=None,
        names_filter=lambda name: name.endswith("pattern")
    )

    # Get the right indices for the attention scores

    if head_type == "duplicate":
        src_indices = range(seq_len)
        dest_indices = range(seq_len, 2 * seq_len)
    elif head_type == "prev":
        src_indices = range(seq_len)
        dest_indices = range(1, seq_len + 1)
    elif head_type == "induction":
        dest_indices = range(seq_len, 2 * seq_len)
        src_indices = range(1, seq_len + 1)
    else:
        raise ValueError(f"Unknown head type {head_type}")

    results = t.zeros(model.cfg.n_layers, model.cfg.n_heads, device="cuda", dtype=t.float32)
    for layer in range(model.cfg.n_layers):
        for head in range(model.cfg.n_heads):
            attn_scores: Float[Tensor, "batch head dest src"] = cache["pattern", layer]
            avg_attn_on_duplicates = attn_scores[:, head, dest_indices, src_indices].mean().item()
            results[layer, head] = avg_attn_on_duplicates

    return results


def plot_early_head_validation_results(seq_len: int = 50, batch: int = 50):
    '''
    Produces a plot that looks like Figure 18 in the paper.
    '''
    head_types = ["duplicate", "prev", "induction"]

    results = t.stack([
        get_attn_scores(model, seq_len, batch, head_type=head_type)
        for head_type in head_types
    ])

    imshow(
        results,
        facet_col=0,
        facet_labels=[
            f"{head_type.capitalize()} token attention prob.<br>on sequences of random tokens"
            for head_type in head_types
        ],
        labels={"x": "Head", "y": "Layer"},
        width=1300,
    )


model.reset_hooks()
plot_early_head_validation_results()
```

▶ Solution

Note - these figures suggest that it would be a useful bit of infrastructure to have a "wiki" for the heads of a model, giving their scores according to some metrics re head functions, like the ones we've seen here. HookedTransformer makes this pretty easy to make, as just changing the name input to `HookedTransformer.from_pretrained` gives a different model but in the same architecture, so the same code should work. If you want to make this, I'd love to see it!

As a proof of concept, I made a mosaic of all induction heads across the 40 models then in HookedTransformer.

Minimal Circuit

Background: faithfulness, completeness, and minimality

The authors developed three criteria for validating their circuit explanations: faithfulness, completeness and minimality. They are defined as follows:

- **Faithful** = the circuit can perform as well as the whole model
- **Complete** = the circuit contains all nodes used to perform the task
- **Minimal** = the circuit doesn't contain nodes irrelevant to the task

If all three criteria are met, then the circuit is considered a reliable explanation for model behaviour.

Exercise - can you understand why each of these criteria is important? For each pair of criteria, is it possible for a circuit to meet them both but fail the third (and if yes, can you give examples?).

▶ Answer

Now that we've analysed most components of the circuit and we have a rough idea of how they work, the next step is to ablate everything except those core components and verify that the model still performs well.

This ablation is pretty massive - we're ablating everything except for the output of each of our key attention heads (e.g. duplicate token heads or S-inhibition heads) at a single sequence position (e.g. for the DTHs this is the `S2` token, and for SIHs this is the `end` token). Given that our core circuit has 26 heads in total, and our sequences have length around 20 on average, this means we're ablating all but $(26/144)/20 \approx 1\%$ of our attention heads' output (and the number of possible paths through the model is reduced by ***much*** more than this).

How do we ablate? We could use zero-ablation, but this actually has some non-obvious problems. To explain why intuitively, heads might be "expecting" non-zero input, and setting the input to zero is essentially an arbitrary choice which takes it off distribution. You can think of this as adding a bias term to this head, which might mess up subsequent computation and lead to noisy results. We could also use mean-ablation (i.e. set a head's output to its average output over `ioi_dataset`), but the problem here is that taking the mean over this dataset might contain relevant information for solving the IOI task. For example the `is_duplicated` flag which gets written to `S2` will be present for all sequences, so averaging won't remove this information.

Can you think of a way to solve this problem? After you've considered this, you can use the dropdown below to see how the authors handled this.

▶ Solution

One other complication - the sentences have different templates, and the positions of tokens like `S` and `IO` are not consistent across these templates (we avoided this problem in previous exercises by choosing a very small set of sentences, where all the important tokens had the same indices). An example of two templates with different positions:

```
"Then, [B] and [A] had a long argument and after that [B] said to [A]"
"After the lunch [B] and [A] went to the [PLACE], and [B] gave a [OBJECT] to [A]"
```

Can you guess what the authors did to solve this problem?

▶ Answer

Exercise - constructing the minimal circuit

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

This exercise is expected to take a long time; at least an hour. It is probably the most challenging exercise in this notebook.
```

You now have enough information to perform ablation on your model, to get the minimal circuit. Below, you can try to implement this yourself.

This exercise is very technically challenging, so you're welcome to skip it if it doesn't seem interesting to you. However, I recommend you have a read of the solution, to understand the rough contours of how this ablation works.

If you want to attempt this task, then you can start with the code below. We define two dictionaries, one mapping head types to the heads in the model which are of that type, and the other mapping head types to the sequence positions which we *won't* be ablating for those types of head.

```
CIRCUIT = {
    "name mover": [(9, 9), (10, 0), (9, 6)],
    "backup name mover": [(10, 10), (10, 6), (10, 2), (10, 1), (11, 2), (9, 7), (9, 0), (11, 9)],
    "negative name mover": [(10, 7), (11, 10)],
    "s2 inhibition": [(7, 3), (7, 9), (8, 6), (8, 10)],
    "induction": [(5, 5), (5, 8), (5, 9), (6, 9)],
    "duplicate token": [(0, 1), (0, 10), (3, 0)],
    "previous token": [(2, 2), (4, 11)],
}

SEQ_POS_TO_KEEP = {
    "name mover": "end",
    "backup name mover": "end",
    "negative name mover": "end",
    "s2 inhibition": "end",
    "induction": "S2",
    "duplicate token": "S2",
    "previous token": "S1+1",
}
```

To be clear, the things that we'll be mean-ablating are:

- Every head not in the `CIRCUIT` dict
- Every sequence position for the heads in `CIRCUIT` dict, except for the sequence positions given by the `SEQ_POS_TO_KEEP` dict

And we'll be mean-ablating by replacing a head's output with the mean output for `abc_dataset`, over all sentences with the same template as the sentence in the batch. You can access the templates for a dataset using the `dataset.groups` attribute, which returns a list of tensors (each one containing the indices of sequences in the batch sharing the same template).

Now, you can try and complete the following function, which should add a ***permanent hook*** to perform this ablation whenever the model is run on the `ioi_dataset` (note that this hook will only make sense if the model is run on this dataset, so we should reset hooks if we run it on a different dataset).

Permanent hooks are a useful feature of transformerlens. They behave just like regular hooks, except they aren't removed when you run the model (e.g. using `model.run_with_cache` or `model.run_with_hooks`). The only way to remove them is with:

```
model.reset_hooks(including_permanent=True)
```

You can add permanent hooks as follows:

```
model.add_hook(hook_name, hook_fn, is_permanent=True)
```

where `hook_name` can be a string or a filter function mapping strings to booleans.

```
def get_heads_and_posns_to_keep(
    means_dataset: IOIDataset,
    model: HookedTransformer,
    circuit: Dict[str, List[Tuple[int, int]]],
    seq_pos_to_keep: Dict[str, str],
```

```
) -> Dict[int, Bool[Tensor, "batch seq head"]]:
    '''
    Returns a dictionary mapping layers to a boolean mask giving the indices of the
    z output which *shouldn't* be mean-ablated.

    The output of this function will be used for the hook function that does ablation.
    '''
    heads_and_posns_to_keep = {}
    batch, seq, n_heads = len(means_dataset), means_dataset.max_len, model.cfg.n_heads

    for layer in range(model.cfg.n_layers):

        mask = t.zeros(size=(batch, seq, n_heads))

        for (head_type, head_list) in circuit.items():
            seq_pos = seq_pos_to_keep[head_type]
            indices = means_dataset.word_idx[seq_pos]
            for (layer_idx, head_idx) in head_list:
                if layer_idx == layer:
                    mask[:, indices, head_idx] = 1

        heads_and_posns_to_keep[layer] = mask.bool()

    return heads_and_posns_to_keep



def hook_fn_mask_z(
    z: Float[Tensor, "batch seq head d_head"],
    hook: HookPoint,
    heads_and_posns_to_keep: Dict[int, Bool[Tensor, "batch seq head"]],
    means: Float[Tensor, "layer batch seq head d_head"],
) -> Float[Tensor, "batch seq head d_head"]:
    '''
    Hook function which masks the z output of a transformer head.

    heads_and_posns_to_keep
        Dict created with the get_heads_and_posns_to_keep function. This tells
        us where to mask.

    means
        Tensor of mean z values of the means_dataset over each group of prompts
        with the same template. This tells us what values to mask with.
    '''
    # Get the mask for this layer, and add d_head=1 dimension so it broadcasts correctly
    mask_for_this_layer = heads_and_posns_to_keep[hook.layer()].unsqueeze(-1).to(z.device)

    # Set z values to the mean
    z = t.where(mask_for_this_layer, z, means[hook.layer()])

    return z


def compute_means_by_template(
    means_dataset: IOIDataset,
    model: HookedTransformer
) -> Float[Tensor, "layer batch seq head_idx d_head"]:
    '''
    Returns the mean of each head's output over the means dataset. This mean is
    computed separately for each group of prompts with the same template (these
    are given by means_dataset.groups).
    '''
    # Cache the outputs of every head
    _, means_cache = model.run_with_cache(
        means_dataset.toks.long(),
        return_type=None,
        names_filter=lambda name: name.endswith("z"),
    )
    # Create tensor to store means
    n_layers, n_heads, d_head = model.cfg.n_layers, model.cfg.n_heads, model.cfg.d_head
    batch, seq_len = len(means_dataset), means_dataset.max_len
    means = t.zeros(size=(n_layers, batch, seq_len, n_heads, d_head), device=model.cfg.device)

    # Get set of different templates for this data
    for layer in range(model.cfg.n_layers):
        z_for_this_layer: Float[Tensor, "batch seq head d_head"] = means_cache[utils.get_act_name("z", layer)]
        for template_group in means_dataset.groups:
            z_for_this_template = z_for_this_layer[template_group]
```

```
            z_means_for_this_template = einops.reduce(z_for_this_template, "batch seq head d_head -> seq head d_head
            means[layer, template_group] = z_means_for_this_template

    return means


def add_mean_ablation_hook(
    model: HookedTransformer,
    means_dataset: IOIDataset,
    circuit: Dict[str, List[Tuple[int, int]]] = CIRCUIT,
    seq_pos_to_keep: Dict[str, str] = SEQ_POS_TO_KEEP,
    is_permanent: bool = True,
) -> HookedTransformer:
    '''
    Adds a permanent hook to the model, which ablates according to the circuit and
    seq_pos_to_keep dictionaries.

    In other words, when the model is run on ioi_dataset, every head's output will
    be replaced with the mean over means_dataset for sequences with the same template,
    except for a subset of heads and sequence positions as specified by the circuit
    and seq_pos_to_keep dicts.
    '''

    model.reset_hooks(including_permanent=True)

    # Compute the mean of each head's output on the ABC dataset, grouped by template
    means = compute_means_by_template(means_dataset, model)

    # Convert this into a boolean map
    heads_and_posns_to_keep = get_heads_and_posns_to_keep(means_dataset, model, circuit, seq_pos_to_keep)

    # Get a hook function which will patch in the mean z values for each head, at
    # all positions which aren't important for the circuit
    hook_fn = partial(
        hook_fn_mask_z,
        heads_and_posns_to_keep=heads_and_posns_to_keep,
        means=means
    )

    # Apply hook
    model.add_hook(lambda name: name.endswith("z"), hook_fn, is_permanent=is_permanent)

    return model
```

To test whether your function works, you can use the function provided to you, and see if the logit difference from your implementation of the circuit matches this one:

```
import part3_indirect_object_identification.ioi_circuit_extraction as ioi_circuit_extraction

model = ioi_circuit_extraction.add_mean_ablation_hook(model, means_dataset=abc_dataset, circuit=CIRCUIT, seq_pos_to_

ioi_logits_minimal = model(ioi_dataset.toks)

print(f"Average logit difference (IOI dataset, using entire model): {logits_to_ave_logit_diff_2(ioi_logits_original)
print(f"Average logit difference (IOI dataset, only using circuit): {logits_to_ave_logit_diff_2(ioi_logits_minimal):
```

```
model = add_mean_ablation_hook(model, means_dataset=abc_dataset, circuit=CIRCUIT, seq_pos_to_keep=SEQ_POS_TO_KEEP)

ioi_logits_minimal = model(ioi_dataset.toks)

print(f"Average logit difference (IOI dataset, using entire model): {logits_to_ave_logit_diff_2(ioi_logits_original)
print(f"Average logit difference (IOI dataset, only using circuit): {logits_to_ave_logit_diff_2(ioi_logits_minimal):
```

You should find that the logit difference only drops by a small amount, and is still high enough to represent a high likelihood ratio favouring the IO token over S.

▶ Hint (docstrings of some functions which will be useful for your main function)

▶ Solution

Exercise - calculate minimality scores

Difficulty: 🟠🟠🟠🟠🟠
Importance: 🟠🟠⚪⚪⚪

This exercise is expected to take a long time; at least an hour. It is probably the second most challenging exercise in this notebook.

We'll conclude this section by replicating figure 7 of the paper, which shows the minimality scores for the model.

Again, this exercise is very challenging and is designed to be done with minimal guidance. You will need to read the relevant sections of the paper which explain this plot: section 4 (experimental validation), from the start up to the end of section 4.2. Note that you won't need to perform the sampling algorithm described on page 10, because we're giving you the set K for each component, in the form of the dictionary below (this is based on the information given in figure 20 of the paper, the "minimality sets" table).

```
K_FOR_EACH_COMPONENT = {
    (9, 9): set(),
    (10, 0): {(9, 9)},
    (9, 6): {(9, 9), (10, 0)},
    (10, 7): {(11, 10)},
    (11, 10): {(10, 7)},
    (8, 10): {(7, 9), (8, 6), (7, 3)},
    (7, 9): {(8, 10), (8, 6), (7, 3)},
    (8, 6): {(7, 9), (8, 10), (7, 3)},
    (7, 3): {(7, 9), (8, 10), (8, 6)},
    (5, 5): {(5, 9), (6, 9), (5, 8)},
    (5, 9): {(11, 10), (10, 7)},
    (6, 9): {(5, 9), (5, 5), (5, 8)},
    (5, 8): {(11, 10), (10, 7)},
    (0, 1): {(0, 10), (3, 0)},
    (0, 10): {(0, 1), (3, 0)},
    (3, 0): {(0, 1), (0, 10)},
    (4, 11): {(2, 2)},
    (2, 2): {(4, 11)},
    (11, 2): {(9, 9), (10, 0), (9, 6)},
    (10, 6): {(9, 9), (10, 0), (9, 6), (11, 2)},
    (10, 10): {(9, 9), (10, 0), (9, 6), (11, 2), (10, 6)},
    (10, 2): {(9, 9), (10, 0), (9, 6), (11, 2), (10, 6), (10, 10)},
    (9, 7): {(9, 9), (10, 0), (9, 6), (11, 2), (10, 6), (10, 10), (10, 2)},
    (10, 1): {(9, 9), (10, 0), (9, 6), (11, 2), (10, 6), (10, 10), (10, 2), (9, 7)},
    (11, 9): {(9, 9), (10, 0), (9, 6), (9, 0)},
    (9, 0): {(9, 9), (10, 0), (9, 6), (11, 9)},
}
```

Also, given a dictionary `minimality_scores` (which maps heads to their scores), the following code will produce a plot that looks like the one from the paper:

```
def plot_minimal_set_results(minimality_scores: Dict[Tuple[int, int], float]):
    '''
    Plots the minimality results, in a way resembling figure 7 in the paper.

    minimality_scores:
        Dict with elements like (9, 9): minimality score for head 9.9 (as described
        in section 4.2 of the paper)
    '''

    CIRCUIT_reversed = {head: k for k, v in CIRCUIT.items() for head in v}
    colors = [CIRCUIT_reversed[head].capitalize() + " head" for head in minimality_scores.keys()]
    color_sequence = [px.colors.qualitative.Dark2[i] for i in [0, 1, 2, 5, 3, 6]] + ["#BAEA84"]

    bar(
        list(minimality_scores.values()),
        x=list(map(str, minimality_scores.keys())),
        labels={"x": "Attention head", "y": "Change in logit diff", "color": "Head type"},
        color=colors,
        template="ggplot2",
        color_discrete_sequence=color_sequence,
        bargap=0.02,
        yaxis_tickformat=".0%",
        legend_title_text="",
        title="Plot of minimality scores (as percentages of full model logit diff)",
```

```
        width=800,
        hovermode="x unified"
    )
```

```
# YOUR CODE HERE - define the `minimality_scores` dictionary, to be used in the plot function given above
def get_score(
    model: HookedTransformer,
    ioi_dataset: IOIDataset,
    abc_dataset: IOIDataset,
    K: Set[Tuple[int, int]],
    C: Dict[str, List[Tuple[int, int]]],
) -> float:
    '''
    Returns the value F(C \ K), where F is the logit diff, C is the
    core circuit, and K is the set of circuit components to remove.
    '''
    C_excl_K = {k: [head for head in v if head not in K] for k, v in C.items()}
    model = add_mean_ablation_hook(model, abc_dataset, C_excl_K, SEQ_POS_TO_KEEP)
    logits = model(ioi_dataset.toks)
    score = logits_to_ave_logit_diff_2(logits, ioi_dataset).item()

    return score


def get_minimality_score(
    model: HookedTransformer,
    ioi_dataset: IOIDataset,
    abc_dataset: IOIDataset,
    v: Tuple[int, int],
    K: Set[Tuple[int, int]],
    C: Dict[str, List[Tuple[int, int]]] = CIRCUIT,
) -> float:
    '''
    Returns the value | F(C \ K_union_v) - F(C | K) |, where F is
    the logit diff, C is the core circuit, K is the set of circuit
    components to remove, and v is a head (not in K).
    '''
    assert v not in K
    K_union_v = K | {v}
    C_excl_K_score = get_score(model, ioi_dataset, abc_dataset, K, C)
    C_excl_Kv_score = get_score(model, ioi_dataset, abc_dataset, K_union_v, C)

    return abs(C_excl_K_score - C_excl_Kv_score)


def get_all_minimality_scores(
    model: HookedTransformer,
    ioi_dataset: IOIDataset = ioi_dataset,
    abc_dataset: IOIDataset = abc_dataset,
    k_for_each_component: Dict = K_FOR_EACH_COMPONENT
) -> Dict[Tuple[int, int], float]:
    '''
    Returns dict of minimality scores for every head in the model (as
    a fraction of F(M), the logit diff of the full model).

    Warning - this resets all hooks at the end (including permanent).
    '''
    # Get full circuit score F(M), to divide minimality scores by
    model.reset_hooks(including_permanent=True)
    logits = model(ioi_dataset.toks)
    full_circuit_score = logits_to_ave_logit_diff_2(logits, ioi_dataset).item()

    # Get all minimality scores, using the `get_minimality_score` function
    minimality_scores = {}
    for v, K in tqdm(k_for_each_component.items()):
        score = get_minimality_score(model, ioi_dataset, abc_dataset, v, K)
        minimality_scores[v] = score / full_circuit_score

    model.reset_hooks(including_permanent=True)

    return minimality_scores


minimality_scores = get_all_minimality_scores(model)
```

```
plot_minimal_set_results(minimality_scores)
```

▶ Hint (docstrings of some functions which will be useful for your main function)
▶ Solution

Note - your results won't be exactly the same as the paper's, because of random error (e.g. the order of importanc of heads within each category might not be the same, especially heads with a small effect on the model like the backup name mover heads). But they should be reasonably similar in their important features.

6️⃣ Bonus / exploring anomalies

> Learning objectives
>
> - Explore other parts of the model (e.g. negative name mover heads, and induction heads)
> - Understand the subtleties present in model circuits, and the fact that there are often more parts to a circuit than seem obvious after initial investigation
> - Understand the importance of the three quantitative criteria used by the paper: **faithfulness**, **completeness** and **minimality**

Here, you'll explore some weirder parts of the circuit which we haven't looked at in detail yet. Specifically, there are three parts to explore:

- Early induction heads
- Backup name mover heads
- Positional vs token information being moved

These three sections are all optional, and you can do as many of them as you want (in whichever order you prefer). There will also be some further directions of investigation at the end of this section, which have been suggested either by the authors or by Neel.

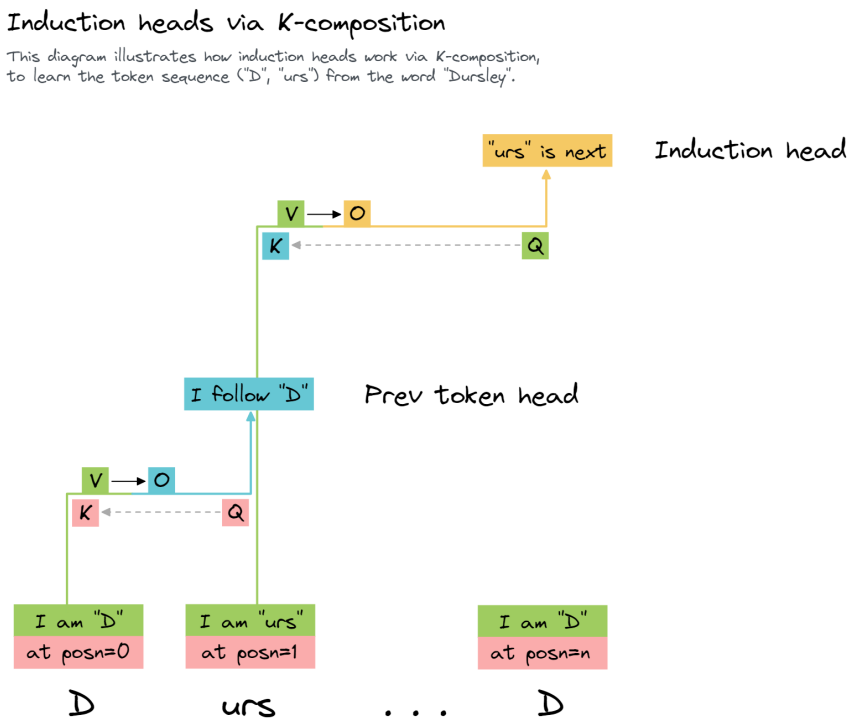
Early induction heads

As we've discussed, a really weird observation is that some of the early heads detecting duplicated tokens are induction heads, not just direct duplicate token heads. This is very weird! What's up with that?

First off, let's just recap what induction heads are. An induction head is an important type of attention head that can detect and continue repeated sequences. It is the second head in a two head induction circuit, which looks for previous copies of the current token and attends

to the token *after* it, and then copies that to the current position and predicts that it will come next. They're enough of a big deal that Anthropic wrote a whole paper on them.

The diagram below shows a diagram for how they work to predict that the token `"urs"` follows `" D"`, the second time the word `"Dursley"` appears (note that we assume the model has not been trained on Harry Potter, so this is an example of in-context learning).



Why is it surprising that induction heads up here? It's surprising because it feels like overkill. The model doesn't care about *what* token comes after the first copy of the subject, just that it's duplicated. And it already has simpler duplicate token heads. My best guess is that it just already had induction heads around and that, in addition to their main function, they *also* only activate on duplicated tokens. So it was useful to repurpose this existing machinery.

This suggests that as we look for circuits in larger models life may get more and more complicated, as components in simpler circuits get repurposed and built upon.

First, in the cell below, you should visualise the attention patterns of the induction heads (`5.5` and `6.9`) on sequences containg repeated tokens, and verify that they are attending to the token *after* the previous instance of that same token. You might want to repeat code you wrote in the "Validation of duplicate token heads" section.

```
model.reset_hooks(including_permanent=True)

attn_heads = [(5, 5), (6, 9)]

# Get repeating sequences (note we could also take mean over larger batch)
batch = 1
seq_len = 15
rep_tokens = generate_repeated_tokens(model, seq_len, batch)

# Run cache (we only need attention patterns for layers 5 and 6)
_, cache = model.run_with_cache(
    rep_tokens,
    return_type = None,
    names_filter = lambda name: name.endswith("pattern") and any(f".{layer}." in name for layer, head in attn_heads)
)

# Display results
attn = t.stack([
    cache["pattern", layer][0, head]
    for (layer, head) in attn_heads
])
cv.attention.attention_patterns(
    tokens = model.to_str_tokens(rep_tokens[0]),
    attention = attn,
    attention_head_names = [f"{layer}.{head}" for (layer, head) in attn_heads]
)
```

One implication of this is that it's useful to categories heads according to whether they occur in simpler circuits, so that as we look for more complex circuits we can easily look for them. This is Hooked to do here! An interesting fact about induction heads is that they work on a sequence of repeated random tokens - notable for being wildly off distribution from the natural language GPT-2 was trained on. Being able to predict a model's behaviour off distribution is a good mark of success for mechanistic interpretability! This is a good sanity check for whether a head is an induction head or not.

We can characterise an induction head by just giving a sequence of random tokens repeated once, and measuring the average attention paid from the second copy of a token to the token after the first copy. At the same time, we can also measure the average attention paid from the second copy of a token to the first copy of the token, which is the attention that the induction head would pay if it were a duplicate token head, and the average attention paid to the previous token to find previous token heads.

Note that this is a superficial study of whether something is an induction head - we totally ignore the question of whether it actually does boost the correct token or whether it composes with a single previous head and how. In particular, we sometimes get anti-induction heads which suppress the induction-y token (no clue why!), and this technique will find those too.

Exercise - validate prev token heads via patching

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

This just involves performing a specific kind of patching, with functions you've already written.
```

The paper mentions that heads `2.2` and `4.11` are previous token heads. Hopefully you already validated this in the previous section by plotting the previous token scores (in your replication of Figure 18). But this time, you'll perform a particular kind of path patching to prove that these heads are functioning as previous token heads, in the way implied by our circuit diagram.

▶ Question - what kind of path patching should you perform?

```
model.reset_hooks(including_permanent=True)


# YOUR CODE HERE - create `induction_head_key_path_patching_results`
induction_head_key_path_patching_results = get_path_patch_head_to_heads(
    receiver_heads = [(5, 5), (6, 9)],
    receiver_input = "k",
    model = model,
    patching_metric = ioi_metric_2
)
```

```
imshow(
    100 * induction_head_key_path_patching_results,
    title="Direct effect on Induction Heads' keys",
    labels={"x": "Head", "y": "Layer", "color": "Logit diff.<br>variation"},
    coloraxis=dict(colorbar_ticksuffix = "%"),
    width=600,
)
```

▶ Solution

You might notice that there are two other heads in the induction heads box in the paper's diagram, both in brackets (heads `5.8` and `5.9`). Can you try and figure out what these heads are doing, and why they're in brackets?

▶ Hint
▶ Answer
▶ Aside - a lesson in nonlinearity

Backup name mover heads

Another fascinating anomaly is that of the **backup name mover heads**. A standard technique to apply when interpreting model internals is ablations, or knock-out. If we run the model but intervene to set a specific head to zero, what happens? If the model is robust to this intervention, then naively we can be confident that the head is not doing anything important, and conversely if the model is much worse at the task this suggests that head was important. There are several conceptual flaws with this approach, making the evidence only suggestive, e.g. that the average output of the head may be far from zero and so the knockout may send it far from expected activations, breaking internals on *any* task. But it's still an Hooked technique to apply to give some data.

But a wild finding in the paper is that models have **built in redundancy**. If we knock out one of the name movers, then there are some backup name movers in later layers that *change their behaviour* and do (some of) the job of the original name mover head. This means that naive knock-out will significantly underestimate the importance of the name movers.

Let's test this! Let's ablate the most important name mover (which is `9.9`, as the code below verifies for us) on just the `end` token, and compare performance.

```
model.reset_hooks(including_permanent=True)

ioi_logits, ioi_cache = model.run_with_cache(ioi_dataset.toks)
original_average_logit_diff = logits_to_ave_logit_diff_2(ioi_logits)
```

```
s_unembeddings = model.W_U.T[ioi_dataset.s_tokenIDs]
io_unembeddings = model.W_U.T[ioi_dataset.io_tokenIDs]
logit_diff_directions: Float[Tensor, "batch d_model"] =  io_unembeddings - s_unembeddings

per_head_residual, labels = ioi_cache.stack_head_results(layer=-1, return_labels=True)
per_head_residual = einops.rearrange(
    per_head_residual[:, t.arange(len(ioi_dataset)).to(device), ioi_dataset.word_idx["end"].to(device)],
    "(layer head) batch d_model -> layer head batch d_model",
    layer=model.cfg.n_layers
)
```

```
11
12 per_head_logit_diffs = residual_stack_to_logit_diff(per_head_residual, ioi_cache, logit_diff_directions)
13
14 top_layer, top_head = topk_of_Nd_tensor(per_head_logit_diffs, k=1)[0]
15 print(f"Top Name Mover to ablate: {top_layer}.{top_head}")
```

```
 1 # Getting means we can use to ablate
 2 abc_means = ioi_circuit_extraction.compute_means_by_template(abc_dataset, model)[top_layer]
 3
 4 # Define hook function and add to model
 5 def ablate_top_head_hook(z: Float[Tensor, "batch pos head_index d_head"], hook):
 6     '''
 7     Ablates hook by patching in results
 8     '''
 9     z[range(len(ioi_dataset)), ioi_dataset.word_idx["end"], top_head] = abc_means[range(len(ioi_dataset)), ioi_datas
10     return z
11
12 model.add_hook(utils.get_act_name("z", top_layer), ablate_top_head_hook)
13
14 # Runs the model, temporarily adds caching hooks and then removes *all* hooks after running, including the ablation
15 ablated_logits, ablated_cache = model.run_with_cache(ioi_dataset.toks)
16 rprint("\n".join([
17     f"{original_average_logit_diff:.4f} = Original logit diff",
18     f"{per_head_logit_diffs[top_layer, top_head]:.4f} = Direct Logit Attribution of top name mover head",
19     f"{original_average_logit_diff - per_head_logit_diffs[top_layer, top_head]:.4f} = Naive prediction of post ablat
20     f"{logits_to_ave_logit_diff_2(ablated_logits):.4f} = Logit diff after ablating L{top_layer}H{top_head}",
21 ]))
```

What's going on here? We calculate the logit diff for our full model, and how much of that is coming directly from head `9.9`. Given this, we come up with an estimate for what the logit diff will fall to when we ablate this head. In fact, performance is **much** better than this naive prediction.

Why is this happening? As before, we can look at the direct logit attribution of each head to get a sense of what's going on.

```
 1 per_head_ablated_residual, labels = ablated_cache.stack_head_results(layer=-1, return_labels=True)
 2 per_head_ablated_residual = einops.rearrange(
 3     per_head_ablated_residual[:, t.arange(len(ioi_dataset)).to(device), ioi_dataset.word_idx["end"].to(device)],
 4     "(layer head) batch d_model -> layer head batch d_model",
 5     layer=model.cfg.n_layers
 6 )
 7 per_head_ablated_logit_diffs = residual_stack_to_logit_diff(per_head_ablated_residual, ablated_cache, logit_diff_dir
 8 per_head_ablated_logit_diffs = per_head_ablated_logit_diffs.reshape(model.cfg.n_layers, model.cfg.n_heads)
 9
10 imshow(
11     t.stack([
12         per_head_logit_diffs,
13         per_head_ablated_logit_diffs,
14         per_head_ablated_logit_diffs - per_head_logit_diffs
15     ]),
16     title="Direct logit contribution by head, pre / post ablation",
17     labels={"x":"Head", "y":"Layer"},
18     facet_col=0,
19     facet_labels=["No ablation", "9.9 is ablated", "Change in head contribution post-ablation"],
20 )
21
22 scatter(
23     y=per_head_logit_diffs.flatten(),
24     x=per_head_ablated_logit_diffs.flatten(),
25     hover_name=labels,
26     range_x=(-1, 1),
27     range_y=(-2, 2),
28     labels={"x": "Ablated", "y": "Original"},
29     title="Original vs Post-Ablation Direct Logit Attribution of Heads",
30     width=600,
31     add_line="y=x"
32 )
```

The first plots show us that, after we ablate head `9.9`, while its direct contribution to the logit diff falls (obviously), a lot of contributions from other heads (particularly in layer 10) actually increase. The second plot shows this in a different way (the distinctive heads in the right hand heatmap are the same as the heads lying well below the y=x line in the scatter plot).

One natural hypothesis is that this is because the final LayerNorm scaling has changed, which can scale up or down the final residual stream. This is slightly true, and we can see that the typical head is a bit off from the x=y line. But the average LN scaling ratio is 1.04, and this should uniformly change *all* heads by the same factor, so this can't be sufficient.

```
ln_scaling_no_ablation = ioi_cache["ln_final.hook_scale"][t.arange(len(ioi_dataset)), ioi_dataset.word_idx["end"]].s
ln_scaling_ablated = ablated_cache["ln_final.hook_scale"][t.arange(len(ioi_dataset)), ioi_dataset.word_idx["end"]].s
```

```
scatter(
    y=ln_scaling_ablated,
    x=ln_scaling_no_ablation,
    labels={"x": "No ablation", "y": "Ablation"},
    title=f"Final LN scaling factors compared (ablation vs no ablation)<br>Average ratio = {(ln_scaling_no_ablation
    width=700,
    add_line="y=x"
)
```

Exercise to the reader: Can you finish off this analysis? What's going on here? Why are the backup name movers changing their behaviour? Why is one negative name mover becoming significantly less important?

Positional vs token information being moved

In section A of the appendix (titled **Disentangling token and positional signal in the output of S-Inhibition Heads**), the authors attempt to figure out whether the S-Inhibition heads are using token or positional information to supress the attention paid to `S1`. This is illustrated in my IOI diagram, by purple vs pink boxes.

The way the authors find out which one is which is ingenious. They construct datasets from the original IOI dataset, with some of the signals erased or flipped. For instance, if they want to examine the effect of inverting the positional information but preserving the token information written by the S-inhibition heads, they can replace sentences like:

```
When Mary and John went to the store, John gave a drink to Mary
```

with:

```
When John and Mary went to the store, John gave a drink to Mary
```

Let's be exactly clear on why this works. The information written to the `end` token position by the S-inhibition heads will be some combination of "don't pay attention to the duplicated token" and "don't pay attention to the token that's in the same position as the duplicated token". If we run our model on the first sentence above but then patch in the second sentence, then:

- The "**don't pay attention to the duplicated token**" signal will be unchanged (because this signal still refers to John)
- The "**don't pay attention to the token that's in the same position as the duplicated token**" signal will flip (because this information points to the position of `Mary` in the second sentence, hence to the position of `John` in the first sentence).

That was just one example (flipping positional information, keeping token information the same), but we can do any of six different types of flip:

Token signal	Positional signal	Sentence	ABB -> ?
Same	Same	`When Mary and John went to the store, John gave a drink to Mary`	ABB
Random	Same	`When Emma and Paul went to the store, Paul gave ...`	CDD
Inverted	Same	`When John and Mary went to the store, Mary gave ...`	BAA
Same	Inverted	`When John and Mary went to the store, John gave ...`	BAB
Random	Inverted	`When Paul and Emma went to the store, Emma gave ...`	DCD
Inverted	Inverted	`When Mary and John went to the store, Mary gave ...`	ABA

We use the `gen_flipped_prompts` method to generate each of these datasets:

```
datasets: List[Tuple[Tuple, str, IOIDataset]] = [
    ((0, 0), "original", ioi_dataset),
    ((1, 0), "random token", ioi_dataset.gen_flipped_prompts("ABB->CDD, BAB->DCD")),
    ((2, 0), "inverted token", ioi_dataset.gen_flipped_prompts("ABB->BAA, BAB->ABA")),
    ((0, 1), "inverted position", ioi_dataset.gen_flipped_prompts("ABB->BAB, BAB->ABB")),
    ((1, 1), "inverted position, random token", ioi_dataset.gen_flipped_prompts("ABB->DCD, BAB->CDD")),
    ((2, 1), "inverted position, inverted token", ioi_dataset.gen_flipped_prompts("ABB->ABA, BAB->BAA")),
]
```

Note - the purpose of the type annotation for `datasets` is so that, when we iterate through datasets, the type checker can identify the third item in each iterate as an `IOIDataset`, and autocomplete methods for us.

```
results = t.zeros(3, 2).to(device)

s2_inhibition_heads = CIRCUIT["s2 inhibition"]
layers = set(layer for layer, head in s2_inhibition_heads)

names_filter=lambda name: name in [utils.get_act_name("z", layer) for layer in layers]

def patching_hook_fn(z: Float[Tensor, "batch seq head d_head"], hook: HookPoint, cache: ActivationCache):
    heads_to_patch = [head for layer, head in s2_inhibition_heads if layer == hook.layer()]
    z[:, :, heads_to_patch] = cache[hook.name][:, :, heads_to_patch]
    return z

for ((row, col), desc, dataset) in datasets:

    # Get cache of values from the modified dataset
    _, cache_for_patching = model.run_with_cache(
        dataset.toks,
        names_filter=names_filter,
        return_type=None
    )

    # Run model on IOI dataset, but patch S-inhibition heads with signals from modified dataset
    patched_logits = model.run_with_hooks(
        ioi_dataset.toks,
        fwd_hooks=[(names_filter, partial(patching_hook_fn, cache=cache_for_patching))]
    )

    # Get logit diff for patched results
    # Note, we still use IOI dataset for our "correct answers" reference point
    results[row, col] = logits_to_ave_logit_diff_2(patched_logits, ioi_dataset)
```

```
imshow(
    results,
    labels={"x": "Positional signal", "y": "Token signal"},
    x=["Original", "Inverted"],
    y=["Original", "Random", "Inverted"],
    title="Logit diff after changing all S2 inhibition heads' output signals via patching",
    text_auto=".2f"
)
```

What are your interpretations of this plot?

▶ Some thoughts

Let's dig a little deeper. Rather than just looking at the S-inhibition heads collectively, we can look at each of them individually.

Exercise - decompose S-Inhibition heads

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

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

This involves a lot of duplicating code from above.
```

Make the same plot as above, but after intervening on each of the S-inhibition heads individually.

You can do this by creating a `results` tensor of shape `(M, 3, 2)` where `M` is the number of S-inhibition heads, and each slice contains the results of intervening on that particular head, then using the `facet` arguments of the `imshow` function:

(You might prefer to plot the results in the form of "change in logit diff from clean value, as a fraction of clean value", to make the results look clearer.)

```
results = t.zeros(len(CIRCUIT["s2 inhibition"]), 3, 2).to(device)
# Your code here - fill in results!

def patching_hook_fn(
    z: Float[Tensor, "batch seq head d_head"],
    hook: HookPoint,
    cache: ActivationCache,
    head: int
):
    z[:, :, head] = cache[hook.name][:, :, head]
    return z

for i, (layer, head) in enumerate(CIRCUIT["s2 inhibition"]):

    model.reset_hooks(including_permanent=True)

    hook_name = utils.get_act_name("z", layer)

    for ((row, col), desc, dataset) in datasets:

        # Get cache of values from the modified dataset
        _, cache_for_patching = model.run_with_cache(
            dataset.toks,
            names_filter=lambda name: name == hook_name,
            return_type=None
        )

        # Run model on IOI dataset, but patch S-inhibition heads with signals from modified dataset
        patched_logits = model.run_with_hooks(
            ioi_dataset.toks,
            fwd_hooks=[(hook_name, partial(patching_hook_fn, cache=cache_for_patching, head=head))]
        )

        # Get logit diff for patched results
        # Note, we still use IOI dataset for our "correct answers" reference point
        results[i, row, col] = logits_to_ave_logit_diff_2(patched_logits, ioi_dataset)
```

```
imshow(
    (results - results[0, 0, 0]) / results[0, 0, 0],
    labels={"x": "Positional signal", "y": "Token signal"},
    x=["Original", "Inverted"],
    y=["Original", "Random", "Inverted"],
    title="Logit diff after patching individual S2 inhibition heads (as proportion of clean logit diff)",
    facet_col=0,
```