```python
import os
DEVELOPMENT_MODE = False
# Detect if we're running in Google Colab
try:
    import google.colab
    IN_COLAB = True
    print("Running as a Colab notebook")
except:
    IN_COLAB = False

# Install if in Colab
#if IN_COLAB:
#   %pip install transformer_lens
#    %pip install circuitsvis
#    %pip install nnsight
    # Install a faster Node version
#    !curl -fsSL https://deb.nodesource.com/setup_16.x | sudo -E bash -; sudo apt-get install

# Hot reload in development mode & not running on the CD
if not IN_COLAB:
    from IPython import get_ipython
    ip = get_ipython()
    if not ip.extension_manager.loaded:
        ip.extension_manager.load('autoreload')
        %autoreload 2

IN_GITHUB = os.getenv("GITHUB_ACTIONS") == "true"
```

```
Running as a Colab notebook
```

```python
! pip install jaxtyping
! pip install git+https://github.com/TransformerLensOrg/TransformerLens
! pip install nnsight
! pip install datasets
! pip install BitsAndBytes
```

```python
import torch
import torch.nn as nn
from typing import Dict, List, Optional, Union, NamedTuple
from transformers import PreTrainedModel
import einops
from dataclasses import dataclass
from typing import Literal
import logging

# Named tuple for storing both logits and loss, matching TransformerLens's Output class
class Output(NamedTuple):
    logits: torch.Tensor  # [batch, pos, d_vocab]
    loss: Optional[torch.Tensor] = None  # Either scalar or [batch, pos-1]

@dataclass
class ActivationCache:
    """A wrapper around a dictionary of activations with helper functions for analysis.

    This provides similar functionality to TransformerLens's ActivationCache but works with
    Hugging Face models. Key operations include:
    - Storing activations from model forward passes
    - Decomposing residual streams
    - Computing head/neuron contributions
    - Applying LayerNorm scaling
    """
    cache_dict: Dict[str, torch.Tensor]
    model: PreTrainedModel
    has_batch_dim: bool = True

    def __getitem__(self, key: str) -> torch.Tensor:
        """Get activation by key name."""
        return self.cache_dict[key]

    def keys(self):
        """Get all activation keys."""
        return self.cache_dict.keys()

    def decompose_resid(self, layer: Optional[int] = None, mlp_input: bool = False,
                        apply_ln: bool = False, pos_slice: Optional[Union[slice, int]] = None,
                        mode: Literal['all', 'mlp', 'attn'] = 'all') -> torch.Tensor:
        """Decompose residual stream into components from each layer.

        Args:
            layer: Which layer to decompose up to (None means all layers)
            mlp_input: Whether to include attention output for current layer
            apply_ln: Whether to apply LayerNorm scaling
            pos_slice: Which positions to keep
            mode: Which components to include ('all', 'mlp', or 'attn')

        Returns:
            Tensor of shape [components, batch, pos, d_model] containing the decomposed
            residual stream
        """
        components = []

        # Add embeddings if needed
        if mode in ['all', 'attn']:
            components.append(self['embed'])
            if 'pos_embed' in self.cache_dict:
```

Resources

You are subscribed to Colab Pro. Learn more
Available: 48 compute units
Usage rate: approximately 1.66 per hour
You have 1 active session.

Manage sessions

Python 3 Google Compute Engine backend (GPU)
Showing resources from 15:06 to 15:56

System RAM
4.8 / 51.0 GB

GPU RAM
5.7 / 15.0 GB

Disk
39.3 / 235.7 GB

```
                components.append(self['pos_embed'])

        # Process each layer
        n_layers = layer if layer is not None else self.model.config.num_hidden_layers
        for l in range(n_layers):
            # Add attention components
            if mode in ['all', 'attn']:
                if f'blocks.{l}.attn.hook_result' in self.cache_dict:
                    components.append(self[f'blocks.{l}.attn.hook_result'])

            # Add MLP components
            if mode in ['all', 'mlp']:
                if f'blocks.{l}.mlp.hook_post' in self.cache_dict:
                    components.append(self[f'blocks.{l}.mlp.hook_post'])

        # Stack components
        components = torch.stack(components)

        # Apply position slicing if needed
        if pos_slice is not None:
            components = components[..., pos_slice, :]

        # Apply layer norm if requested
        if apply_ln:
            components = self.apply_ln_to_stack(components, layer=layer)

        return components

    def apply_ln_to_stack(self, residual_stack: torch.Tensor, layer: Optional[int] = None,
                          mlp_input: bool = False) -> torch.Tensor:
        """Apply appropriate LayerNorm scaling to a stack of residual stream components."""
        # Get LayerNorm weights/bias for target layer
        if layer is None:
            layer = self.model.config.num_hidden_layers

        ln_weight = self.model.transformer.layers[layer].ln1.weight
        ln_bias = self.model.transformer.layers[layer].ln1.bias

        # Calculate LayerNorm statistics
        mean = residual_stack.mean(dim=-1, keepdim=True)
        var = ((residual_stack - mean) ** 2).mean(dim=-1, keepdim=True)
        normed = (residual_stack - mean) / (var + 1e-5).sqrt()

        # Apply weights and bias
        return ln_weight * normed + ln_bias

    def stack_activation(self, activation_name: str, layer: int = -1) -> torch.Tensor:
        """Stack activations with a given name from all layers up to layer."""
        components = []
        n_layers = layer if layer >= 0 else self.model.config.num_hidden_layers

        for l in range(n_layers):
            key = f"blocks.{l}.{activation_name}"
            if key in self.cache_dict:
                components.append(self.cache_dict[key])

        return torch.stack(components)

def create_caching_hooks(model, hook_points):
    """Create forward hooks to cache activations at specified points in a Qwen model."""
    cache = {}
    hooks = []

    def get_activation(name):
        def hook(module, input, output):
            cache[name] = output
        return hook

    # Parse hook points and attach hooks
    for hook_point in hook_points:
        # Split into components (e.g. "model.layers.0.input_layernorm")
        components = hook_point.split('.')

        # Navigate model architecture to find module
        current_module = model
        for comp in components:
            if comp == "model":
                current_module = current_module.model  # Access the inner model
            elif comp.isdigit():
                current_module = current_module[int(comp)]
            else:
                current_module = getattr(current_module, comp)

        # Register hook
        hook = current_module.register_forward_hook(get_activation(hook_point))
        hooks.append(hook)

    return hooks, ActivationCache(cache, model)

def run_with_cache(model, inputs, names_filter=None):
    """Run model with activation caching, properly handling quantized models."""
    # Create list of hook points if not specified
    if names_filter is None:
        names_filter = [
            "model.embed_tokens",
            f"model.layers.{model.config.num_hidden_layers-1}.input_layernorm"
        ]

    # Create and attach hooks
```

```
    hooks, cache = create_caching_hooks(model, names_filter)

    try:
        # Ensure inputs are properly formatted
        if not isinstance(inputs, dict):
            inputs = {
                "input_ids": inputs,
                "attention_mask": torch.ones_like(inputs),
                "return_dict": True
            }

        # Run model without autocast
        with torch.no_grad():
            outputs = model(**inputs)

        # Create Output object
        if hasattr(outputs, 'loss'):
            return Output(outputs.logits, outputs.loss), cache
        else:
            return Output(outputs.logits), cache

    finally:
        # Always remove hooks
        for hook in hooks:
            hook.remove()
```

```
from nnsight import LanguageModel
import torch
from transformers import BitsAndBytesConfig

# First, let's set up the 4-bit quantization configuration
# This matches the settings used by Unsloth for their quantized models
quantization_config = BitsAndBytesConfig(
    load_in_4bit=True,
    bnb_4bit_quant_type="nf4",  # Normal Float 4 format
    bnb_4bit_compute_dtype=torch.float16,
    bnb_4bit_use_double_quant=True  # Nested quantization for additional memory savings
)

# Disable gradients since we're doing inference only
torch.set_grad_enabled(False)

# Load the base instruction model (4-bit quantized version)
base_model = LanguageModel(
    "unsloth/Qwen2.5-14B-Instruct-bnb-4bit",
    device_map="cuda:0",
    trust_remote_code=True,  # Required for Qwen models
    quantization_config=quantization_config
)

# Load the COT model (4-bit quantized version)
cot_model = LanguageModel(
    "unsloth/DeepSeek-R1-Distill-Qwen-14B-unsloth-bnb-4bit",

    device_map="cuda:1",
    trust_remote_code=True,
    quantization_config=quantization_config
)
```

```
# %%
import os
from IPython import get_ipython

ipython = get_ipython()
# Code to automatically update the HookedTransformer code as its edited without restarting the
if ipython is not None:
    ipython.magic("load_ext autoreload")
    ipython.magic("autoreload 2")

import plotly.io as pio
pio.renderers.default = "jupyterlab"

# Import stuff
import einops
import json
import argparse

from datasets import load_dataset
from pathlib import Path
import plotly.express as px
from torch.distributions.categorical import Categorical
from tqdm import tqdm
import torch
import numpy as np
#from transformer_lens import HookedTransformer
from jaxtyping import Float
#from transformer_lens.hook_points import HookPoint

from functools import partial

from IPython.display import HTML

from transformer_lens.utils import to_numpy
import pandas as pd

from html import escape
import colorsys
```

```
39
40
41 import wandb
42
43 import plotly.graph_objects as go
44
45 update_layout_set = {
46     "xaxis_range", "yaxis_range", "hovermode", "xaxis_title", "yaxis_title", "colorbar", "colo
47      "title_x", "bargap", "bargroupgap", "xaxis_tickformat", "yaxis_tickformat", "title_y", "
48      "xaxis_gridwidth", "xaxis_gridcolor", "yaxis_showgrid", "yaxis_gridwidth"
49 }
50
51 def imshow(tensor, renderer=None, xaxis="", yaxis="", **kwargs):
52     if isinstance(tensor, list):
53         tensor = torch.stack(tensor)
54     kwargs_post = {k: v for k, v in kwargs.items() if k in update_layout_set}
55     kwargs_pre = {k: v for k, v in kwargs.items() if k not in update_layout_set}
56     if "facet_labels" in kwargs_pre:
57         facet_labels = kwargs_pre.pop("facet_labels")
58     else:
59         facet_labels = None
60     if "color_continuous_scale" not in kwargs_pre:
61         kwargs_pre["color_continuous_scale"] = "RdBu"
62     fig = px.imshow(to_numpy(tensor), color_continuous_midpoint=0.0,labels={"x":xaxis, "y":ya
63     if facet_labels:
64         for i, label in enumerate(facet_labels):
65             fig.layout.annotations[i]['text'] = label
66
67     fig.show(renderer)
68
69 def line(tensor, renderer=None, xaxis="", yaxis="", **kwargs):
70     px.line(y=to_numpy(tensor), labels={"x":xaxis, "y":yaxis}, **kwargs).show(renderer)
71
72 def scatter(x, y, xaxis="", yaxis="", caxis="", renderer=None, return_fig=False, **kwargs):
73     x = to_numpy(x)
74     y = to_numpy(y)
75     fig = px.scatter(y=y, x=x, labels={"x":xaxis, "y":yaxis, "color":caxis}, **kwargs)
76     if return_fig:
77         return fig
78     fig.show(renderer)
79
80 def lines(lines_list, x=None, mode='lines', labels=None, xaxis='', yaxis='', title = '', log_
81     # Helper function to plot multiple lines
82     if type(lines_list)==torch.Tensor:
83         lines_list = [lines_list[i] for i in range(lines_list.shape[0])]
84     if x is None:
85         x=np.arange(len(lines_list[0]))
86     fig = go.Figure(layout={'title':title})
87     fig.update_xaxes(title=xaxis)
88     fig.update_yaxes(title=yaxis)
89     for c, line in enumerate(lines_list):
90         if type(line)==torch.Tensor:
91             line = to_numpy(line)
92         if labels is not None:
93             label = labels[c]
94         else:
95             label = c
96         fig.add_trace(go.Scatter(x=x, y=line, mode=mode, name=label, hovertext=hover, **kwargs
97     if log_y:
98         fig.update_layout(yaxis_type="log")
99     fig.show()
100
101 def bar(tensor, renderer=None, xaxis="", yaxis="", **kwargs):
102     px.bar(
103         y=to_numpy(tensor),
104         labels={"x": xaxis, "y": yaxis},
105         template="simple_white",
106         **kwargs).show(renderer)
107
108 def create_html(strings, values, saturation=0.5, allow_different_length=False):
109     # escape strings to deal with tabs, newlines, etc.
110     escaped_strings = [escape(s, quote=True) for s in strings]
111     processed_strings = [
112         s.replace("\n", "<br/>").replace("\t", " ").replace(" ", " ")
113         for s in escaped_strings
114     ]
115
116     if isinstance(values, torch.Tensor) and len(values.shape)>1:
117         values = values.flatten().tolist()
118
119     if not allow_different_length:
120         assert len(processed_strings) == len(values)
121
122     # scale values
123     max_value = max(max(values), -min(values))+1e-3
124     scaled_values = [v / max_value * saturation for v in values]
125
126     # create html
127     html = ""
128     for i, s in enumerate(processed_strings):
129         if i<len(scaled_values):
130             v = scaled_values[i]
131         else:
132             v = 0
133         if v < 0:
134             hue = 0  # hue for red in HSV
135         else:
136             hue = 0.66  # hue for blue in HSV
137         rgb_color = colorsys.hsv_to_rgb(
```

```
            hue, v, 1
        )  # hsv color with hue 0.66 (blue), saturation as v, value 1
        hex_color = "#%02x%02x%02x" % (
            int(rgb_color[0] * 255),
            int(rgb_color[1] * 255),
            int(rgb_color[2] * 255),
        )
        html += f'<span style="background-color: {hex_color}; border: 1px solid lightgray; for

    display(HTML(html))

# crosscoder stuff

def arg_parse_update_cfg(default_cfg):
    """
    Helper function to take in a dictionary of arguments, convert these to command line argume

    If in Ipython, just returns with no changes
    """
    if get_ipython() is not None:
        # Is in IPython
        print("In IPython - skipped argparse")
        return default_cfg
    cfg = dict(default_cfg)
    parser = argparse.ArgumentParser()
    for key, value in default_cfg.items():
        if type(value) == bool:
            # argparse for Booleans is broken rip. Now you put in a flag to change the default
            if value:
                parser.add_argument(f"--{key}", action="store_false")
            else:
                parser.add_argument(f"--{key}", action="store_true")

        else:
            parser.add_argument(f"--{key}", type=type(value), default=value)
    args = parser.parse_args()
    parsed_args = vars(args)
    cfg.update(parsed_args)
    print("Updated config")
    print(json.dumps(cfg, indent=2))
    return cfg

def load_pile_lmsys_mixed_tokens():
    try:
        print("Loading data from disk")
        all_tokens = torch.load("/workspace/data/pile-lmsys-mix-1m-tokenized-gemma-2.pt")
    except:
        print("Data is not cached. Loading data from HF")
        data = load_dataset(
            "ckkissane/pile-lmsys-mix-1m-tokenized-gemma-2",
            split="train",
            cache_dir="/workspace/cache/"
        )
        data.save_to_disk("/workspace/data/pile-lmsys-mix-1m-tokenized-gemma-2.hf")
        data.set_format(type="torch", columns=["input_ids"])
        all_tokens = data["input_ids"]
        torch.save(all_tokens, "/workspace/data/pile-lmsys-mix-1m-tokenized-gemma-2.pt")
        print(f"Saved tokens to disk")
    return all_tokens
```

```
class Buffer:
    """
    Buffer implementation that properly handles activation collection from nnsight models.
    """
    def __init__(self, cfg, model_A, model_B, all_tokens):
        self.cfg = cfg
        self.model_A = model_A
        self.model_B = model_B
        self.all_tokens = all_tokens

        # Buffer setup
        self.buffer_size = cfg["batch_size"] * cfg["buffer_mult"]
        self.buffer_batches = self.buffer_size // (cfg["seq_len"] - 1)
        self.buffer_size = self.buffer_batches * (cfg["seq_len"] - 1)

        # Initialize buffer with appropriate dtype
        self.buffer = torch.zeros(
            (self.buffer_size, 2, model_A.config.hidden_size),
            dtype=torch.bfloat16,
            device=cfg["device"],
            requires_grad=False
        )

        self.token_pointer = 0
        self.first = True
        self.normalize = True

        # Pre-compute normalization factors
        print("Calculating normalization factors...")
        norm_A = self._estimate_norm_scaling_factor(model_A, "Model A")
        norm_B = self._estimate_norm_scaling_factor(model_B, "Model B")

        self.normalisation_factor = torch.tensor(
            [norm_A, norm_B],
            device=cfg["device"],
            dtype=torch.float32
        )

```

```
        self.refresh()

    def _collect_activations(self, model, tokens):
        """Collect activations from a model using nnsight's context manager."""
        with model.trace() as trace:
            # Get the target layer
            layer = model
            for comp in self.cfg["hook_point"].split('.'):
                if comp == "model":
                    continue
                elif comp == "layers":
                    continue
                elif comp.isdigit():
                    layer = layer.layers[int(comp)]
                else:
                    layer = getattr(layer, comp)

            # Run forward pass
            model(input_ids=tokens, attention_mask=torch.ones_like(tokens))

            # Get activations - now they're materialized
            return layer.output[0].to(dtype=torch.bfloat16)

    @torch.no_grad()
    def _estimate_norm_scaling_factor(self, model, desc, n_batches=100):
        """Calculate norm scaling factor with proper tensor handling."""
        print(f"\nEstimating norm scaling factor for {desc}")
        norms = []
        batch_size = min(8, self.cfg["model_batch_size"])

        try:
            model.eval()

            for i in range(n_batches):
                # Get a batch of tokens
                tokens = self.all_tokens[i * batch_size : (i + 1) * batch_size].to(model.devi

                try:
                    # Collect activations using nnsight's context manager
                    acts = self._collect_activations(model, tokens)

                    # Calculate norm - activations are now materialized
                    norm = acts.norm(dim=-1).mean().item()
                    norms.append(norm)

                except Exception as e:
                    print(f"Batch {i} failed: {str(e)}")
                    continue

                if i % 10 == 0:
                    torch.cuda.empty_cache()

        except Exception as e:
            print(f"Error during norm estimation: {str(e)}")

        if not norms:
            print(f"Warning: Using default scaling factor for {desc}")
            return np.sqrt(model.config.hidden_size)

        mean_norm = np.mean(norms)
        return np.sqrt(model.config.hidden_size) / mean_norm

    @torch.no_grad()
    def refresh(self):
        """Refresh buffer with new activations."""
        self.pointer = 0
        print("Refreshing the buffer!")

        num_batches = self.buffer_batches if self.first else self.buffer_batches // 2
        self.first = False

        batch_size = min(8, self.cfg["model_batch_size"])

        for i in range(0, num_batches, batch_size):
            tokens = self.all_tokens[
                self.token_pointer : min(
                    self.token_pointer + batch_size,
                    num_batches
                )
            ].to(self.model_A.device)

            try:
                # Collect activations from both models
                acts_A = self._collect_activations(self.model_A, tokens)
                acts_B = self._collect_activations(self.model_B, tokens)

                # Stack and process activations
                acts = torch.stack([acts_A, acts_B], dim=0)
                acts = acts[:, :, 1:, :]  # Drop BOS token

                # Reshape for buffer storage
                acts = einops.rearrange(
                    acts,
                    "n_layers batch seq_len d_model -> (batch seq_len) n_layers d_model"
                )

                self.buffer[self.pointer : self.pointer + acts.shape[0]] = acts
                self.pointer += acts.shape[0]
                self.token_pointer += batch_size
```

```
            except Exception as e:
                print(f"Error in batch {i}: {str(e)}")
                continue

            if i % 10 == 0:
                torch.cuda.empty_cache()

        # Shuffle buffer
        self.pointer = 0
        self.buffer = self.buffer[torch.randperm(self.buffer.shape[0]).to(self.cfg["device"])]

    @torch.no_grad()
    def next(self):
        """Get next batch of activations."""
        out = self.buffer[self.pointer : self.pointer + self.cfg["batch_size"]]
        self.pointer += self.cfg["batch_size"]

        if self.pointer > self.buffer.shape[0] // 2 - self.cfg["batch_size"]:
            self.refresh()

        if self.normalize:
            out = out * self.normalisation_factor[None, :, None]

        return out
```

```
import wandb
from google.colab import userdata
import os
os.environ["WANDB_API_KEY"] = userdata.get('WANDB_API_KEY')
wandb.login()
from huggingface_hub.hf_api import HfFolder

HfFolder.save_token(userdata.get('HF_TOKEN'))

```

```
wandb: Currently logged in as: jacktpayne51 (jacktpayne51-macquarie-university)
```

```
from tqdm.auto import tqdm  # Properly import tqdm
import torch
from datasets import load_dataset
from pathlib import Path
import os

def load_or_create_qwen_tokens():
    """
    Creates and caches a dataset of tokens specifically for Qwen models using the
    uncopyrighted version of the Pile and LMSys chat data.
    """
    # First, ensure the cache directory exists
    cache_dir = Path("/workspace/data")
    cache_dir.mkdir(parents=True, exist_ok=True)
    cache_path = cache_dir / "pile-lmsys-mix-1m-tokenized-qwen.pt"

    try:
        print("Looking for previously cached Qwen-tokenized data...")
        # Add weights_only=True for security
        all_tokens = torch.load(cache_path, weights_only=True)
        print(f"Found cached data! Loading {len(all_tokens)} tokenized sequences...")

    except (FileNotFoundError, RuntimeError):
        print("No cached data found. Creating new dataset...")

        print("Loading uncopyrighted Pile data...")
        pile = load_dataset(
            "monology/pile-uncopyrighted",
            split="train",
            streaming=True
        )

        print("Loading LMSys chat data...")
        lmsys = load_dataset(
            "lmsys/lmsys-chat-1m",
            split="train",
            streaming=True
        )

        texts = []
        target_samples = 10_000  # 500k from each source for balance

        print(f"Collecting balanced dataset of {target_samples * 2} samples...")
        pile_count = 0
        lmsys_count = 0

        # Collect Pile samples with progress tracking
        pile_iter = iter(pile)
        print("Collecting Pile samples...")
        while pile_count < target_samples:
            try:
                sample = next(pile_iter)
                texts.append(sample['text'])
                pile_count += 1
                if pile_count % 1000 == 0:
                    print(f"Collected {pile_count}/{target_samples} Pile samples")
            except StopIteration:
                break

```

```
        # Collect LMSys samples with progress tracking
        lmsys_iter = iter(lmsys)
        print("\nCollecting LMSys samples...")
        while lmsys_count < target_samples:
            try:
                sample = next(lmsys_iter)
                chat_text = format_chat_for_qwen(sample["conversation"])
                texts.append(chat_text)
                lmsys_count += 1
                if lmsys_count % 1000 == 0:
                    print(f"Collected {lmsys_count}/{target_samples} LMSys samples")
            except StopIteration:
                break

        print(f"\nTokenizing {len(texts)} sequences with Qwen tokenizer...")
        # Process in smaller batches to manage memory
        batch_size = 1000
        all_tokens = []
        total_batches = len(texts) // batch_size + (1 if len(texts) % batch_size else 0)

        for i in range(0, len(texts), batch_size):
            batch = texts[i:i + batch_size]
            print(f"Processing batch {i//batch_size + 1}/{total_batches}")
            tokenized = base_model.tokenizer(
                batch,
                padding=True,
                truncation=True,
                max_length=1024,
                return_tensors="pt"
            )
            all_tokens.append(tokenized.input_ids)

        # Concatenate all batches
        all_tokens = torch.cat(all_tokens, dim=0)

        print("Caching tokenized data...")
        torch.save(all_tokens, cache_path)

    print(f"Successfully prepared {len(all_tokens)} sequences")
    return all_tokens

def format_chat_for_qwen(conversation):
    """
    Formats LMSys chat data to match Qwen's expected chat format.
    This ensures the model will process chat data similarly to how it was trained.

    Args:
        conversation: A list of message dictionaries with 'role' and 'content' keys

    Returns:
        str: A formatted chat string in Qwen's expected format
    """
    formatted_chat = ""
    for message in conversation:
        role = message["role"]
        content = message["content"]

        # Format based on Qwen's chat template
        if role == "user":
            formatted_chat += f"<|im_start|>user\n{content}<|im_end|>\n"
        elif role == "assistant":
            formatted_chat += f"<|im_start|>assistant\n{content}<|im_end|>\n"

    return formatted_chat
```

```
cfg = {
        # Core training parameters
        "batch_size": 1024,  # Reduced from 4096
        "buffer_mult": 32,   # Reduced from 128
        "model_batch_size": 8,  # Significantly reduced
        "lr": 5e-5,
        "l1_coeff": 2,
        "beta1": 0.9,
        "beta2": 0.999,

        # Model architecture settings
        "dict_size": 16384,
        "hook_point": "model.layers.24.input_layernorm",
        "d_in": base_model.config.hidden_size,

        # Hardware settings
        "device": "cuda:0",
        "seq_len": 1024,
        "enc_dtype": "fp16",

        # Training schedule
        "num_tokens": 400_000_000,
        "save_every": 30000,
        "log_every": 100,

        # Additional settings
        "seed": 42,
        "wandb_project": "crosscoder-cot-analysis",
        "wandb_entity": "jacktpayne51",
        "dec_init_norm": 0.08
    }
def setup_training(base_model, cot_model):
    """
```

```
    Prepares the crosscoder training environment with Qwen-specific settings.
    """
    print("Beginning training setup...")

    # Load our Qwen-specific tokenized data
    all_tokens = load_or_create_qwen_tokens()

    cfg = {
        # Core training parameters
        "batch_size": 1024,  # Reduced from 4096
        "buffer_mult": 32,   # Reduced from 128
        "model_batch_size": 8,  # Significantly reduced
        "lr": 5e-5,
        "l1_coeff": 2,
        "beta1": 0.9,
        "beta2": 0.999,

        # Model architecture settings
        "dict_size": 16384,
        "hook_point": "model.layers.24.input_layernorm",
        "d_in": base_model.config.hidden_size,

        # Hardware settings
        "device": "cuda:0",
        "seq_len": 1024,
        "enc_dtype": "fp16",

        # Training schedule
        "num_tokens": 400_000_000,
        "save_every": 30000,
        "log_every": 100,

        # Additional settings
        "seed": 42,
        "wandb_project": "crosscoder-cot-analysis",
        "wandb_entity": "jacktpayne51",
        "dec_init_norm": 0.08
    }

    print("Creating activation buffer...")
    buffer = Buffer(
        cfg=cfg,
        model_A=base_model,
        model_B=cot_model,
        all_tokens=all_tokens
    )

    print("Training setup complete!")
    return cfg, buffer
```

```

from torch import nn
import pprint
import torch.nn.functional as F
from typing import Optional, Union
from huggingface_hub import hf_hub_download

from typing import NamedTuple

DTYPES = {"fp32": torch.float32, "fp16": torch.float16, "bf16": torch.bfloat16}
SAVE_DIR = Path("/workspace/crosscoder-model-diff-replication/checkpoints")

class LossOutput(NamedTuple):
    # loss: torch.Tensor
    l2_loss: torch.Tensor
    l1_loss: torch.Tensor
    l0_loss: torch.Tensor
    explained_variance: torch.Tensor
    explained_variance_A: torch.Tensor
    explained_variance_B: torch.Tensor

class CrossCoder(nn.Module):
    def __init__(self, cfg):
        super().__init__()
        self.cfg = cfg
        d_hidden = self.cfg["dict_size"]
        d_in = self.cfg["d_in"]
        self.dtype = DTYPES[self.cfg["enc_dtype"]]
        torch.manual_seed(self.cfg["seed"])
        # hardcoding n_models to 2
        self.W_enc = nn.Parameter(
            torch.empty(2, d_in, d_hidden, dtype=self.dtype)
        )
        self.W_dec = nn.Parameter(
            torch.nn.init.normal_(
                torch.empty(
                    d_hidden, 2, d_in, dtype=self.dtype
                )
            )
        )
        self.W_dec = nn.Parameter(
            torch.nn.init.normal_(
                torch.empty(
                    d_hidden, 2, d_in, dtype=self.dtype
                )
            )
        )
        # Make norm of W_dec 0.1 for each column, separate per layer
```

```
        self.W_dec.data = (
            self.W_dec.data / self.W_dec.data.norm(dim=-1, keepdim=True) * self.cfg["dec_init_
        )
        # Initialise W_enc to be the transpose of W_dec
        self.W_enc.data = einops.rearrange(
            self.W_dec.data.clone(),
            "d_hidden n_models d_model -> n_models d_model d_hidden",
        )
        self.b_enc = nn.Parameter(torch.zeros(d_hidden, dtype=self.dtype))
        self.b_dec = nn.Parameter(
            torch.zeros((2, d_in), dtype=self.dtype)
        )
        self.d_hidden = d_hidden

        self.to(self.cfg["device"])
        self.save_dir = None
        self.save_version = 0

    def encode(self, x, apply_relu=True):
        # x: [batch, n_models, d_model]
        x_enc = einops.einsum(
            x,
            self.W_enc,
            "batch n_models d_model, n_models d_model d_hidden -> batch d_hidden",
        )
        if apply_relu:
            acts = F.relu(x_enc + self.b_enc)
        else:
            acts = x_enc + self.b_enc
        return acts

    def decode(self, acts):
        # acts: [batch, d_hidden]
        acts_dec = einops.einsum(
            acts,
            self.W_dec,
            "batch d_hidden, d_hidden n_models d_model -> batch n_models d_model",
        )
        return acts_dec + self.b_dec

    def forward(self, x):
        # x: [batch, n_models, d_model]
        acts = self.encode(x)
        return self.decode(acts)

    def get_losses(self, x):
        # x: [batch, n_models, d_model]
        x = x.to(self.dtype)
        acts = self.encode(x)
        # acts: [batch, d_hidden]
        x_reconstruct = self.decode(acts)
        diff = x_reconstruct.float() - x.float()
        squared_diff = diff.pow(2)
        l2_per_batch = einops.reduce(squared_diff, 'batch n_models d_model -> batch', 'sum')
        l2_loss = l2_per_batch.mean()

        total_variance = einops.reduce((x - x.mean(0)).pow(2), 'batch n_models d_model -> bat
        explained_variance = 1 - l2_per_batch / total_variance

        per_token_l2_loss_A = (x_reconstruct[:, 0, :] - x[:, 0, :]).pow(2).sum(dim=-1).squeeze
        total_variance_A = (x[:, 0, :] - x[:, 0, :].mean(0)).pow(2).sum(-1).squeeze()
        explained_variance_A = 1 - per_token_l2_loss_A / total_variance_A

        per_token_l2_loss_B = (x_reconstruct[:, 1, :] - x[:, 1, :]).pow(2).sum(dim=-1).squeeze
        total_variance_B = (x[:, 1, :] - x[:, 1, :].mean(0)).pow(2).sum(-1).squeeze()
        explained_variance_B = 1 - per_token_l2_loss_B / total_variance_B

        decoder_norms = self.W_dec.norm(dim=-1)
        # decoder_norms: [d_hidden, n_models]
        total_decoder_norm = einops.reduce(decoder_norms, 'd_hidden n_models -> d_hidden', 'su
        l1_loss = (acts * total_decoder_norm[None, :]).sum(-1).mean(0)

        l0_loss = (acts>0).float().sum(-1).mean()

        return LossOutput(l2_loss=l2_loss, l1_loss=l1_loss, l0_loss=l0_loss, explained_varian

    def create_save_dir(self):
        base_dir = Path("/workspace/crosscoder-model-diff-replication/checkpoints")
        version_list = [
            int(file.name.split("_")[1])
            for file in list(SAVE_DIR.iterdir())
            if "version" in str(file)
        ]
        if len(version_list):
            version = 1 + max(version_list)
        else:
            version = 0
        self.save_dir = base_dir / f"version_{version}"
        self.save_dir.mkdir(parents=True)

    def save(self):
        if self.save_dir is None:
            self.create_save_dir()
        weight_path = self.save_dir / f"{self.save_version}.pt"
        cfg_path = self.save_dir / f"{self.save_version}_cfg.json"

        torch.save(self.state_dict(), weight_path)
        with open(cfg_path, "w") as f:
            json.dump(self.cfg, f)
```

```
        print(f"Saved as version {self.save_version} in {self.save_dir}")
        self.save_version += 1

    @classmethod
    def load_from_hf(
        cls,
        repo_id: str = "ckkissane/crosscoder-gemma-2-2b-model-diff",
        path: str = "blocks.14.hook_resid_pre",
        device: Optional[Union[str, torch.device]] = None
    ) -> "CrossCoder":
        """
        Load CrossCoder weights and config from HuggingFace.

        Args:
            repo_id: HuggingFace repository ID
            path: Path within the repo to the weights/config
            model: The transformer model instance needed for initialization
            device: Device to load the model to (defaults to cfg device if not specified)

        Returns:
            Initialized CrossCoder instance
        """

        # Download config and weights
        config_path = hf_hub_download(
            repo_id=repo_id,
            filename=f"{path}/cfg.json"
        )
        weights_path = hf_hub_download(
            repo_id=repo_id,
            filename=f"{path}/cc_weights.pt"
        )

        # Load config
        with open(config_path, 'r') as f:
            cfg = json.load(f)

        # Override device if specified
        if device is not None:
            cfg["device"] = str(device)

        # Initialize CrossCoder with config
        instance = cls(cfg)

        # Load weights
        state_dict = torch.load(weights_path, map_location=cfg["device"])
        instance.load_state_dict(state_dict)

        return instance

    @classmethod
    def load(cls, version_dir, checkpoint_version):
        save_dir = Path("/workspace/crosscoder-model-diff-replication/checkpoints") / str(vers
        cfg_path = save_dir / f"{str(checkpoint_version)}_cfg.json"
        weight_path = save_dir / f"{str(checkpoint_version)}.pt"

        cfg = json.load(open(cfg_path, "r"))
        pprint.pprint(cfg)
        self = cls(cfg=cfg)
        self.load_state_dict(torch.load(weight_path))
        return self
```

```

import tqdm

from torch.nn.utils import clip_grad_norm_
class Trainer:
    def __init__(self, cfg, model_A, model_B, all_tokens):
        self.cfg = cfg
        self.model_A = model_A
        self.model_B = model_B
        self.crosscoder = CrossCoder(cfg)
        self.buffer = Buffer(cfg, model_A, model_B, all_tokens)
        self.total_steps = cfg["num_tokens"] // cfg["batch_size"]

        self.optimizer = torch.optim.Adam(
            self.crosscoder.parameters(),
            lr=cfg["lr"],
            betas=(cfg["beta1"], cfg["beta2"]),
        )
        self.scheduler = torch.optim.lr_scheduler.LambdaLR(
            self.optimizer, self.lr_lambda
        )
        self.step_counter = 0

        wandb.init(project=cfg["wandb_project"], entity=cfg["wandb_entity"])

    def lr_lambda(self, step):
        if step < 0.8 * self.total_steps:
            return 1.0
        else:
            return 1.0 - (step - 0.8 * self.total_steps) / (0.2 * self.total_steps)

    def get_l1_coeff(self):
        # Linearly increases from 0 to cfg["l1_coeff"] over the first 0.05 * self.total_steps
        if self.step_counter < 0.05 * self.total_steps:
            return self.cfg["l1_coeff"] * self.step_counter / (0.05 * self.total_steps)
```

```
        else:
            return self.cfg["l1_coeff"]

    def step(self):
        acts = self.buffer.next()
        losses = self.crosscoder.get_losses(acts)
        loss = losses.l2_loss + self.get_l1_coeff() * losses.l1_loss
        loss.backward()
        clip_grad_norm_(self.crosscoder.parameters(), max_norm=1.0)
        self.optimizer.step()
        self.scheduler.step()
        self.optimizer.zero_grad()

        loss_dict = {
            "loss": loss.item(),
            "l2_loss": losses.l2_loss.item(),
            "l1_loss": losses.l1_loss.item(),
            "l0_loss": losses.l0_loss.item(),
            "l1_coeff": self.get_l1_coeff(),
            "lr": self.scheduler.get_last_lr()[0],
            "explained_variance": losses.explained_variance.mean().item(),
            "explained_variance_A": losses.explained_variance_A.mean().item(),
            "explained_variance_B": losses.explained_variance_B.mean().item(),
        }
        self.step_counter += 1
        return loss_dict

    def log(self, loss_dict):
        wandb.log(loss_dict, step=self.step_counter)
        print(loss_dict)

    def save(self):
        self.crosscoder.save()

    def train(self):
        self.step_counter = 0
        try:
            for i in tqdm.trange(self.total_steps):
                loss_dict = self.step()
                if i % self.cfg["log_every"] == 0:
                    self.log(loss_dict)
                if (i + 1) % self.cfg["save_every"] == 0:
                    self.save()
        finally:
            self.save()


def train_crosscoder(base_model, cot_model):
    # Set up our configuration and buffer
    cfg, buffer = setup_training(base_model, cot_model)

    # Initialize our crosscoder
    crosscoder = CrossCoder(cfg)

    # Create trainer instance
    trainer = Trainer(
        cfg=cfg,
        model_A=base_model,
        model_B=cot_model,
        all_tokens=buffer
    )

    # Start training
    print("Beginning crosscoder training...")
    trainer.train()

    return crosscoder

# Main execution
if __name__ == "__main__":
    # Train the crosscoder
    crosscoder = train_crosscoder(base_model, cot_model)

    # Save the final model
    crosscoder.save()
```

```
Beginning training setup...
Looking for previously cached Qwen-tokenized data...
Found cached data! Loading 20000 tokenized sequences...
Successfully prepared 20000 sequences
Creating activation buffer...
Calculating normalization factors...

Estimating norm scaling factor for Model A
Batch 0 failed: 'Qwen2ForCausalLM' object has no attribute 'layers'
Batch 1 failed: 'Qwen2ForCausalLM' object has no attribute 'layers'
Batch 2 failed: 'Qwen2ForCausalLM' object has no attribute 'layers'
Batch 3 failed: 'Qwen2ForCausalLM' object has no attribute 'layers'
Batch 4 failed: 'Qwen2ForCausalLM' object has no attribute 'layers'
Batch 5 failed: 'Qwen2ForCausalLM' object has no attribute 'layers'
Batch 6 failed: 'Qwen2ForCausalLM' object has no attribute 'layers'
Batch 7 failed: 'Qwen2ForCausalLM' object has no attribute 'layers'
Batch 8 failed: 'Qwen2ForCausalLM' object has no attribute 'layers'
Batch 9 failed: 'Qwen2ForCausalLM' object has no attribute 'layers'
Batch 10 failed: 'Qwen2ForCausalLM' object has no attribute 'layers'
Batch 11 failed: 'Qwen2ForCausalLM' object has no attribute 'layers'
Batch 12 failed: 'Qwen2ForCausalLM' object has no attribute 'layers'
Batch 13 failed: 'Qwen2ForCausalLM' object has no attribute 'layers'
Batch 14 failed: 'Qwen2ForCausalLM' object has no attribute 'layers'
Batch 15 failed: 'Qwen2ForCausalLM' object has no attribute 'layers'
Batch 16 failed: 'Qwen2ForCausalLM' object has no attribute 'layers'
Batch 17 failed: 'Qwen2ForCausalLM' object has no attribute 'layers'
Batch 18 failed: 'Qwen2ForCausalLM' object has no attribute 'layers'
Batch 19 failed: 'Qwen2ForCausalLM' object has no attribute 'layers'
Batch 20 failed: 'Qwen2ForCausalLM' object has no attribute 'layers'
Batch 21 failed: 'Qwen2ForCausalLM' object has no attribute 'layers'
Batch 22 failed: 'Qwen2ForCausalLM' object has no attribute 'layers'
Batch 23 failed: 'Qwen2ForCausalLM' object has no attribute 'layers'
Batch 24 failed: 'Qwen2ForCausalLM' object has no attribute 'layers'
Batch 25 failed: 'Qwen2ForCausalLM' object has no attribute 'layers'
Batch 26 failed: 'Qwen2ForCausalLM' object has no attribute 'layers'
Batch 27 failed: 'Qwen2ForCausalLM' object has no attribute 'layers'
Batch 28 failed: 'Qwen2ForCausalLM' object has no attribute 'layers'
Batch 29 failed: 'Qwen2ForCausalLM' object has no attribute 'layers'
Batch 30 failed: 'Qwen2ForCausalLM' object has no attribute 'layers'
Batch 31 failed: 'Qwen2ForCausalLM' object has no attribute 'layers'
Batch 32 failed: 'Qwen2ForCausalLM' object has no attribute 'layers'
Batch 33 failed: 'Qwen2ForCausalLM' object has no attribute 'layers'
Batch 34 failed: 'Qwen2ForCausalLM' object has no attribute 'layers'
Batch 35 failed: 'Qwen2ForCausalLM' object has no attribute 'layers'
Batch 36 failed: 'Qwen2ForCausalLM' object has no attribute 'layers'
Batch 37 failed: 'Qwen2ForCausalLM' object has no attribute 'layers'
Batch 38 failed: 'Qwen2ForCausalLM' object has no attribute 'layers'
Batch 39 failed: 'Qwen2ForCausalLM' object has no attribute 'layers'
Batch 40 failed: 'Qwen2ForCausalLM' object has no attribute 'layers'
Batch 41 failed: 'Qwen2ForCausalLM' object has no attribute 'layers'
Batch 42 failed: 'Qwen2ForCausalLM' object has no attribute 'layers'
Batch 43 failed: 'Qwen2ForCausalLM' object has no attribute 'layers'
Batch 44 failed: 'Qwen2ForCausalLM' object has no attribute 'layers'
Batch 45 failed: 'Qwen2ForCausalLM' object has no attribute 'layers'
Batch 46 failed: 'Qwen2ForCausalLM' object has no attribute 'layers'
Batch 47 failed: 'Qwen2ForCausalLM' object has no attribute 'layers'
Batch 48 failed: 'Qwen2ForCausalLM' object has no attribute 'layers'
Batch 49 failed: 'Qwen2ForCausalLM' object has no attribute 'layers'
Batch 50 failed: 'Qwen2ForCausalLM' object has no attribute 'layers'
Batch 51 failed: 'Qwen2ForCausalLM' object has no attribute 'layers'
Batch 52 failed: 'Qwen2ForCausalLM' object has no attribute 'layers'
Batch 53 failed: 'Qwen2ForCausalLM' object has no attribute 'layers'
Batch 54 failed: 'Qwen2ForCausalLM' object has no attribute 'layers'
Batch 55 failed: 'Qwen2ForCausalLM' object has no attribute 'layers'
Batch 56 failed: 'Qwen2ForCausalLM' object has no attribute 'layers'
Batch 57 failed: 'Qwen2ForCausalLM' object has no attribute 'layers'
Batch 58 failed: 'Qwen2ForCausalLM' object has no attribute 'layers'
Batch 59 failed: 'Qwen2ForCausalLM' object has no attribute 'layers'
Batch 60 failed: 'Qwen2ForCausalLM' object has no attribute 'layers'
Batch 61 failed: 'Qwen2ForCausalLM' object has no attribute 'layers'
Batch 62 failed: 'Qwen2ForCausalLM' object has no attribute 'layers'
Batch 63 failed: 'Qwen2ForCausalLM' object has no attribute 'layers'
Batch 64 failed: 'Qwen2ForCausalLM' object has no attribute 'layers'
Batch 65 failed: 'Qwen2ForCausalLM' object has no attribute 'layers'
Batch 66 failed: 'Qwen2ForCausalLM' object has no attribute 'layers'
Batch 67 failed: 'Qwen2ForCausalLM' object has no attribute 'layers'
Batch 68 failed: 'Qwen2ForCausalLM' object has no attribute 'layers'
Batch 69 failed: 'Qwen2ForCausalLM' object has no attribute 'layers'
Batch 70 failed: 'Qwen2ForCausalLM' object has no attribute 'layers'
Batch 71 failed: 'Qwen2ForCausalLM' object has no attribute 'layers'
Batch 72 failed: 'Qwen2ForCausalLM' object has no attribute 'layers'
Batch 73 failed: 'Qwen2ForCausalLM' object has no attribute 'layers'
Batch 74 failed: 'Qwen2ForCausalLM' object has no attribute 'layers'
Batch 75 failed: 'Qwen2ForCausalLM' object has no attribute 'layers'
Batch 76 failed: 'Qwen2ForCausalLM' object has no attribute 'layers'
Batch 77 failed: 'Qwen2ForCausalLM' object has no attribute 'layers'
Batch 78 failed: 'Qwen2ForCausalLM' object has no attribute 'layers'
Batch 79 failed: 'Qwen2ForCausalLM' object has no attribute 'layers'
Batch 80 failed: 'Qwen2ForCausalLM' object has no attribute 'layers'
Batch 81 failed: 'Qwen2ForCausalLM' object has no attribute 'layers'
Batch 82 failed: 'Qwen2ForCausalLM' object has no attribute 'layers'
Batch 83 failed: 'Qwen2ForCausalLM' object has no attribute 'layers'
Batch 84 failed: 'Qwen2ForCausalLM' object has no attribute 'layers'
Batch 85 failed: 'Qwen2ForCausalLM' object has no attribute 'layers'
Batch 86 failed: 'Qwen2ForCausalLM' object has no attribute 'layers'
Batch 87 failed: 'Qwen2ForCausalLM' object has no attribute 'layers'
Batch 88 failed: 'Qwen2ForCausalLM' object has no attribute 'layers'
```