subtraction_grok-final

May 10, 2024

1 Grokking Notebook

2 Setup

[1]:
```
# standard library imports
import copy
import dataclasses
from datetime import datetime
import itertools
import os
import random
import warnings
from functools import partial
from pathlib import Path
from typing import List, Optional, Union

warnings.filterwarnings("ignore", category=DeprecationWarning)
warnings.filterwarnings("ignore", category=UserWarning)
warnings.filterwarnings("ignore", category=FutureWarning)

# third party imports
# !pip install -r requirements.txt
# !pip install datasets
# !pip install einops
# !pip install torch==2.1.2
# !pip install -U kaleido
# !pip install transformer_lens
# !pip install git+https://github.com/neelnanda-io/neel-plotly.git
import datasets
import einops
from neel_plotly.plot import line, imshow, melt
import numpy as np
from pathlib import Path
import plotly.express as px
import plotly.io as pio
import torch
import torch.nn as nn
```

```
import torch.nn.functional as F
import torch.optim as optim
import tqdm.auto as tqdm
import transformer_lens
import transformer_lens.utils as utils
from fancy_einsum import einsum
from IPython import get_ipython
from IPython.display import HTML
from plotly.offline import init_notebook_mode, iplot
from torch.utils.data import DataLoader
from transformers import AutoConfig, AutoModelForCausalLM, AutoTokenizer
from transformer_lens.hook_points import HookPoint, HookedRootModule
from transformer_lens import ActivationCache, FactoredMatrix,␣
 ↪HookedTransformer, HookedTransformerConfig
```

[2]:
```
# Define constants
if torch.cuda.is_available():
    DEVICE = torch.device("cuda")
elif torch.backends.mps.is_available():
    DEVICE = torch.device("mps")
else:
    DEVICE = torch.device("cpu")
print(f"Device: {DEVICE}")

p = 113                     # prime
TRAIN_FRAC = 0.3            # fraction of data used for training
LEARNING_RATE = 1e-3        # learning rate for the optimizer
WEIGHT_DECAY = 1.0          # weight decay for regularization
BETAS = (0.9, 0.98)         # betas for optimizer
N_EPOCHS = 25000            # number of training epochs
CHECKPOINT_EVERY = 100      # interval at which to save model checkpoints
SEED = 598                  # seed for data shuffling for reproducibility

pio.renderers.default = "notebook_connected"

try:
    import google.colab
    IN_COLAB = True
    pio.renderers.default = "colab"
    print("Running in a Google Colab environment")

    # required packages for Colab
    packages = ["transformer-lens", "circuitsvis",
                "git+https://github.com/neelnanda-io/PySvelte.git"]
    for package in packages:
        get_ipython().run_line_magic('pip', f'install {package}')
```

```
    # Node.js for PySvelte
    get_ipython().system_raw("curl -fsSL https://deb.nodesource.com/setup_16.x␣
 ↪| sudo -E bash -")
    get_ipython().system_raw("sudo apt-get install -y nodejs")
except ImportError:
    IN_COLAB = False
    print("Running in a Jupyter notebook")

    ipython = get_ipython()
    ipython.magic("load_ext autoreload")
    ipython.magic("autoreload 2")

init_notebook_mode(connected=True)

# Update Plotly layout defaults
pio.templates['plotly'].layout.xaxis.title.font.size = 20
pio.templates['plotly'].layout.yaxis.title.font.size = 20
pio.templates['plotly'].layout.title.font.size = 30
```

```
Device: cuda
Running in a Jupyter notebook
```

2.1 Data Preparation

Input format: |a|b|=|

[3]:
```
# Create a matrix with repeated rows [0, 1, ..., p-1]
a_vector = einops.repeat(torch.arange(p), "i -> (i j)", j=p)

# Create a matrix with repeated columns [0, 1, ..., p-1]
b_vector = einops.repeat(torch.arange(p), "j -> (i j)", i=p)

# Create a matrix filled with the constant value 113
equals_vector = einops.repeat(torch.tensor(113), " -> (i j)", i=p, j=p)

# Stack the tensors to form a combined dataset
dataset = torch.stack([a_vector, b_vector, equals_vector], dim=1).to(DEVICE)

print("First 5 entries of the dataset:")
print(dataset[:5])
```

```
First 5 entries of the dataset:
tensor([[  0,   0, 113],
        [  0,   1, 113],
        [  0,   2, 113],
        [  0,   3, 113],
        [  0,   4, 113]], device='cuda:0')
```

[4]:
```
# Compute labels using modular subtraction
labels = (dataset[:, 0] - dataset[:, 1]) % p

print("First 5 labels:", labels[:5].tolist())
print("Last 5 labels:", labels[-5:].tolist())
```

```
First 5 labels: [0, 112, 111, 110, 109]
Last 5 labels: [4, 3, 2, 1, 0]
```

Convert this to a train + test set - 30% in the training set

[5]:
```
# Set a fixed seed to ensure reproducible results
torch.manual_seed(SEED)

# Generate a random permutation of indices for splitting the data
total_size = p * p
indices = torch.randperm(total_size)

# Calculate the cutoff index for training and testing data split
cutoff = int(total_size * TRAIN_FRAC)

# Split indices into training and testing sets
train_indices = indices[:cutoff]
test_indices = indices[cutoff:]

# Index into the original dataset to create training and testing subsets
train_data = dataset[train_indices]
train_labels = labels[train_indices]
test_data = dataset[test_indices]
test_labels = labels[test_indices]

# Print the first few entries of the training data and labels to verify
print("First 5 training data samples:\n", train_data[:5])
print("\nFirst 5 training labels:\n", train_labels[:5].tolist())
print("\nTraining data shape:\n", train_data.shape)

# Print the first few entries of the testing data and labels to verify
print("\nFirst 5 testing data samples:\n", test_data[:5])
print("\nFirst 5 testing labels:\n", test_labels[:5].tolist())
print("\nTesting data shape:\n", test_data.shape)
```

```
First 5 training data samples:
 tensor([[ 21,  31, 113],
        [ 30,  98, 113],
        [ 47,  10, 113],
        [ 86,  21, 113],
        [ 99,  83, 113]], device='cuda:0')

First 5 training labels:
```

```
 [103, 45, 37, 65, 16]

Training data shape:
 torch.Size([3830, 3])

First 5 testing data samples:
 tensor([[ 43,  40, 113],
        [ 31,  42, 113],
        [ 39,  63, 113],
        [ 35,  61, 113],
        [112, 102, 113]], device='cuda:0')

First 5 testing labels:
 [3, 102, 89, 87, 10]

Testing data shape:
 torch.Size([8939, 3])
```

2.2 Model Configuration

[6]:
```python
cfg = HookedTransformerConfig(
    n_layers=1,                # Number of transformer layers
    n_heads=4,                 # Number of attention heads in each transformer␣
 ↪layer
    d_model=128,               # Dimension of the model
    d_head=32,                 # Dimension of each attention head
    d_mlp=512,                 # Dimension of the feedforward network model in␣
 ↪transformers
    act_fn="relu",             # Activation function
    normalization_type=None,   # Type of normalization layer used in the␣
 ↪transformer, if any
    d_vocab=p + 1,             # Vocabulary size (input)
    d_vocab_out=p,             # Output vocabulary size
    n_ctx=3,                   # Context size or the length of the sequence to␣
 ↪be processed
    init_weights=True,         # Whether to initialize weights
    device=DEVICE,             # Specify the computation device
    seed=999                   # Random seed for reproducibility
)

model = HookedTransformer(cfg)

# disable biases in the model for simplification and clearer interpretation
for name, param in model.named_parameters():
    if "b_" in name:  # check if a bias term
        param.requires_grad = False  # disable gradient computation for biases
```

2.3 Optimizer and Loss Configuration

[7]:
```
# Initialize the optimizer with the specified parameters and settings
optimizer = torch.optim.AdamW(model.parameters(), lr=LEARNING_RATE,␣
↪weight_decay=WEIGHT_DECAY, betas=BETAS)

def loss_fn(logits, labels):
    """
    Calculate the cross-entropy loss between logits from a model and provided␣
↪labels.
    """

    # Reduce logits if 3D (batch_size, sequence_length, vocab_size) to 2D␣
↪(batch_size, vocab_size)
    if len(logits.shape) == 3:
        logits = logits[:, -1]

    logits = logits.to(torch.float64)  # float64 for numerical stability
    log_probs = logits.log_softmax(dim=-1)  # Apply log_softmax to logits
    correct_log_probs = log_probs.gather(dim=-1, index=labels.unsqueeze(-1))[:,␣
↪0]  # Gather correct log probabilities using labels
    return -correct_log_probs.mean()  # Return negative log likelihood mean as␣
↪the loss
```

Baseline losses (before training the model)

[8]:
```
# Calculate and print losses for both training and testing datasets
train_logits = model(train_data)
train_loss = loss_fn(train_logits, train_labels)
print(f"Training Loss: {train_loss.item()}")

test_logits = model(test_data)
test_loss = loss_fn(test_logits, test_labels)
print(f"Testing Loss: {test_loss.item()}")

# Print theoretical uniform loss for comparison
uniform_loss = np.log(p)  # Calculate uniform loss as the log of vocabulary size
print("Uniform loss:", uniform_loss)
```

```
Training Loss: 4.732243928736632
Testing Loss: 4.7345742216589235
Uniform loss: 4.727387818712341
```

2.4 Model Training

We train the model with full batch training rather than stochastic gradient descent to make training smoother and reduce the number of "slingshots". The **training loop** is in the next cell.

[9]:
```
train_losses = []
test_losses = []

model_checkpoints = []
checkpoint_epochs = []

for epoch in tqdm.tqdm(range(N_EPOCHS)):

    # compute logits and loss for training data
    train_logits = model(train_data)
    train_loss = loss_fn(train_logits, train_labels)
    train_loss.backward()  # perform backprop
    train_losses.append(train_loss.item())  # store train loss

    optimizer.step()  # update model parameters
    optimizer.zero_grad()  # reset gradients

    # eval model on test data w/o computing gradients
    with torch.inference_mode():
        test_logits = model(test_data)
        test_loss = loss_fn(test_logits, test_labels)
        test_losses.append(test_loss.item())  # store test loss

    # save checkpoint
    if (epoch + 1) % CHECKPOINT_EVERY == 0:
        checkpoint_epochs.append(epoch)
        model_checkpoints.append(copy.deepcopy(model.state_dict()))
        print(f"Epoch {epoch+1}: Train Loss {train_loss.item()}, Test Loss␣
↪{test_loss.item()}")
```

```
  0%|          | 0/25000 [00:00<?, ?it/s]
```

```
Epoch 100: Train Loss 3.0059929158656735, Test Loss 7.585852132661558
Epoch 200: Train Loss 0.0433928030809459, Test Loss 21.50859270250569
Epoch 300: Train Loss 0.011613317218309889, Test Loss 23.05310104694205
Epoch 400: Train Loss 0.003722114947235942, Test Loss 24.65551049972679
Epoch 500: Train Loss 0.0012342359170706364, Test Loss 26.386930483533444
Epoch 600: Train Loss 0.0004164863681278452, Test Loss 28.181343213291893
Epoch 700: Train Loss 0.000142424870530629, Test Loss 29.968578568585254
Epoch 800: Train Loss 4.943449470924235e-05, Test Loss 31.755781823422193
Epoch 900: Train Loss 1.7509446745346362e-05, Test Loss 33.533314778957084
Epoch 1000: Train Loss 6.413078273326838e-06, Test Loss 35.24915202922623
Epoch 1100: Train Loss 2.521738762056516e-06, Test Loss 36.82470144043266
Epoch 1200: Train Loss 1.1306003522936988e-06, Test Loss 38.14378332408898
Epoch 1300: Train Loss 6.208006910618917e-07, Test Loss 39.036160741043254
Epoch 1400: Train Loss 4.3270551948332257e-07, Test Loss 39.4453574632342
Epoch 1500: Train Loss 3.6928885288374146e-07, Test Loss 39.45362728786979
Epoch 1600: Train Loss 3.5295148751590436e-07, Test Loss 39.25414224757044
```

```
Epoch 1700: Train Loss 3.4909327559665586e-07, Test Loss 39.00591593388508
Epoch 1800: Train Loss 3.4834491988598763e-07, Test Loss 38.761168744323115
Epoch 1900: Train Loss 3.4781202707162017e-07, Test Loss 38.532412518597994
Epoch 2000: Train Loss 3.4686243948522545e-07, Test Loss 38.31023913491154
Epoch 2100: Train Loss 3.463073266325214e-07, Test Loss 38.101486488582815
Epoch 2200: Train Loss 3.451888487163518e-07, Test Loss 37.9020165872304
Epoch 2300: Train Loss 3.4425832964958593e-07, Test Loss 37.71580280965269
Epoch 2400: Train Loss 3.432064049541974e-07, Test Loss 37.537216319412565
Epoch 2500: Train Loss 3.419541931191985e-07, Test Loss 37.363936776193555
Epoch 2600: Train Loss 3.407407021829523e-07, Test Loss 37.19266815962763
Epoch 2700: Train Loss 3.3997814683869825e-07, Test Loss 37.02605739673977
Epoch 2800: Train Loss 3.3883417960322324e-07, Test Loss 36.86718930229241
Epoch 2900: Train Loss 3.379758439578644e-07, Test Loss 36.713057931427215
Epoch 3000: Train Loss 3.368419228295679e-07, Test Loss 36.56585822502598
Epoch 3100: Train Loss 3.3606918817184984e-07, Test Loss 36.423810188304095
Epoch 3200: Train Loss 3.3520038048268584e-07, Test Loss 36.28317709269258
Epoch 3300: Train Loss 3.3428142496317194e-07, Test Loss 36.14519447756681
Epoch 3400: Train Loss 3.3401543894738914e-07, Test Loss 36.010927965450364
Epoch 3500: Train Loss 3.3249521159124396e-07, Test Loss 35.88224901747096
Epoch 3600: Train Loss 3.3187187809524875e-07, Test Loss 35.75254736853727
Epoch 3700: Train Loss 3.3107344281295496e-07, Test Loss 35.63078874179894
Epoch 3800: Train Loss 3.3042759951174893e-07, Test Loss 35.50455120241498
Epoch 3900: Train Loss 3.297559393267913e-07, Test Loss 35.38088067152194
Epoch 4000: Train Loss 3.2925163576034576e-07, Test Loss 35.259108431164385
Epoch 4100: Train Loss 3.283214003622441e-07, Test Loss 35.14045188987611
Epoch 4200: Train Loss 3.277648913385401e-07, Test Loss 35.02595596280389
Epoch 4300: Train Loss 3.2702747724253517e-07, Test Loss 34.912002411333326
Epoch 4400: Train Loss 3.2612114260993855e-07, Test Loss 34.80368336938954
Epoch 4500: Train Loss 3.2563321003514105e-07, Test Loss 34.69220246152472
Epoch 4600: Train Loss 3.2490571964548114e-07, Test Loss 34.57822551153887
Epoch 4700: Train Loss 3.245427976199969e-07, Test Loss 34.462741668407695
Epoch 4800: Train Loss 3.2378673697368283e-07, Test Loss 34.34910871110902
Epoch 4900: Train Loss 3.227331683071813e-07, Test Loss 34.23600208916695
Epoch 5000: Train Loss 3.22201036591331e-07, Test Loss 34.121259142596806
Epoch 5100: Train Loss 3.2158510181449346e-07, Test Loss 34.00963838332935
Epoch 5200: Train Loss 3.207921415653919e-07, Test Loss 33.895098138088265
Epoch 5300: Train Loss 3.204876337787331e-07, Test Loss 33.778591393736434
Epoch 5400: Train Loss 3.196748609568877e-07, Test Loss 33.664101681157504
Epoch 5500: Train Loss 3.193759520297992e-07, Test Loss 33.54816160405954
Epoch 5600: Train Loss 3.187613829351393e-07, Test Loss 33.435063935723086
Epoch 5700: Train Loss 3.180560553697391e-07, Test Loss 33.31694790223854
Epoch 5800: Train Loss 3.174671705760103e-07, Test Loss 33.19959797028475
Epoch 5900: Train Loss 3.1693541207847544e-07, Test Loss 33.08162534574018
Epoch 6000: Train Loss 3.1634469064103353e-07, Test Loss 32.963466662473024
Epoch 6100: Train Loss 3.1562006314809003e-07, Test Loss 32.84109536093257
Epoch 6200: Train Loss 3.1512026986246795e-07, Test Loss 32.722049533093276
Epoch 6300: Train Loss 3.1470491328932095e-07, Test Loss 32.60156416938915
Epoch 6400: Train Loss 3.139866950478538e-07, Test Loss 32.47953733638902
```

Epoch 6500: Train Loss 3.1331029751733835e-07, Test Loss 32.357687046932575
Epoch 6600: Train Loss 3.128944705477748e-07, Test Loss 32.231229756619975
Epoch 6700: Train Loss 3.124197856790219e-07, Test Loss 32.105282406978255
Epoch 6800: Train Loss 3.122804076148314e-07, Test Loss 31.977643548906755
Epoch 6900: Train Loss 3.1154857839919285e-07, Test Loss 31.852835281874874
Epoch 7000: Train Loss 3.1074692348020247e-07, Test Loss 31.728123132054925
Epoch 7100: Train Loss 3.105676947639515e-07, Test Loss 31.5963106269146
Epoch 7200: Train Loss 3.098280957809076e-07, Test Loss 31.462016614363748
Epoch 7300: Train Loss 3.0975778038821466e-07, Test Loss 31.329358495037656
Epoch 7400: Train Loss 3.093170586569655e-07, Test Loss 31.194996621253786
Epoch 7500: Train Loss 3.082622527535832e-07, Test Loss 31.05591862724676
Epoch 7600: Train Loss 3.075545841944255e-07, Test Loss 30.916692743967992
Epoch 7700: Train Loss 3.0760896579065845e-07, Test Loss 30.771634384694448
Epoch 7800: Train Loss 3.068877029613456e-07, Test Loss 30.626560682502816
Epoch 7900: Train Loss 3.063083303832511e-07, Test Loss 30.473166809508978
Epoch 8000: Train Loss 3.054817829511975e-07, Test Loss 30.307674071520164
Epoch 8100: Train Loss 3.050581215589664e-07, Test Loss 30.138866624618995
Epoch 8200: Train Loss 3.045160207927289e-07, Test Loss 29.965632769158717
Epoch 8300: Train Loss 3.037960487993405e-07, Test Loss 29.78504642429459
Epoch 8400: Train Loss 3.035028078980125e-07, Test Loss 29.59625224671302
Epoch 8500: Train Loss 3.0275662909268694e-07, Test Loss 29.412542148807745
Epoch 8600: Train Loss 3.023738171744052e-07, Test Loss 29.22408696860575
Epoch 8700: Train Loss 3.0146514213465147e-07, Test Loss 29.02320219603434
Epoch 8800: Train Loss 3.0094786476828026e-07, Test Loss 28.814179590430516
Epoch 8900: Train Loss 3.00168313258567e-07, Test Loss 28.595000883025666
Epoch 9000: Train Loss 2.993989572766197e-07, Test Loss 28.375304123529446
Epoch 9100: Train Loss 2.987866474538456e-07, Test Loss 28.149548001892875
Epoch 9200: Train Loss 2.978433640803644e-07, Test Loss 27.91576874024424
Epoch 9300: Train Loss 2.972292823263152e-07, Test Loss 27.6796789112964
Epoch 9400: Train Loss 2.9671666616018235e-07, Test Loss 27.442318881023787
Epoch 9500: Train Loss 2.9592070178487414e-07, Test Loss 27.2026331839992
Epoch 9600: Train Loss 2.9597081633373693e-07, Test Loss 26.96711818497249
Epoch 9700: Train Loss 2.949582168336477e-07, Test Loss 26.732879457518656
Epoch 9800: Train Loss 2.9403822968761805e-07, Test Loss 26.50986403080346
Epoch 9900: Train Loss 2.9343834275465927e-07, Test Loss 26.29172915995637
Epoch 10000: Train Loss 2.929995493094401 4e-07, Test Loss 26.07418669760884
Epoch 10100: Train Loss 2.920902804908223e-07, Test Loss 25.862680185864225
Epoch 10200: Train Loss 2.9156904646592994e-07, Test Loss 25.660307124631547
Epoch 10300: Train Loss 2.9074930897434746e-07, Test Loss 25.453413846838554
Epoch 10400: Train Loss 2.902844798395594e-07, Test Loss 25.260588866394
Epoch 10500: Train Loss 2.89446766338922e-07, Test Loss 25.073239545736506
Epoch 10600: Train Loss 2.8883719174760617e-07, Test Loss 24.890931295422895
Epoch 10700: Train Loss 2.882568761916446e-07, Test Loss 24.715377200487623
Epoch 10800: Train Loss 2.876119825408956e-07, Test Loss 24.547406952941916
Epoch 10900: Train Loss 2.8705809516738446e-07, Test Loss 24.38627516115264
Epoch 11000: Train Loss 2.866562471945527e-07, Test Loss 24.230487136294066
Epoch 11100: Train Loss 2.8626720489661995e-07, Test Loss 24.084488021023578
Epoch 11200: Train Loss 2.8595899604486406e-07, Test Loss 23.944945755251418

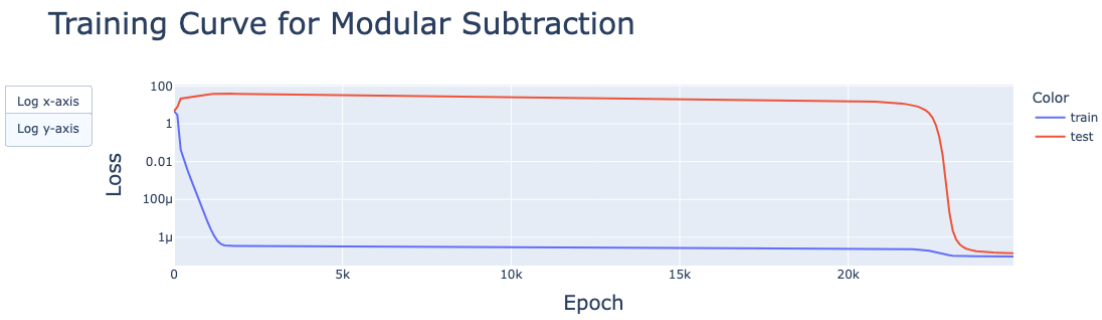
```
Epoch 11300: Train Loss 2.8534722947705023e-07, Test Loss 23.814250872648632
Epoch 11400: Train Loss 2.848904886042891e-07, Test Loss 23.685216176688243
Epoch 11500: Train Loss 2.840391302474721e-07, Test Loss 23.565191030547744
Epoch 11600: Train Loss 2.8396378732891045e-07, Test Loss 23.442964910061136
Epoch 11700: Train Loss 2.8318907201512907e-07, Test Loss 23.32618652162161
Epoch 11800: Train Loss 2.824645621239875e-07, Test Loss 23.215323722630927
Epoch 11900: Train Loss 2.8226523815488483e-07, Test Loss 23.10485065003443
Epoch 12000: Train Loss 2.815920328375191e-07, Test Loss 22.997718124414806
Epoch 12100: Train Loss 2.8151367076041007e-07, Test Loss 22.891693007907158
Epoch 12200: Train Loss 2.812607231196559e-07, Test Loss 22.793070613021293
Epoch 12300: Train Loss 2.804413468142806e-07, Test Loss 22.69837457541991
Epoch 12400: Train Loss 2.799909096426205e-07, Test Loss 22.605794648485393
Epoch 12500: Train Loss 2.793732583890288e-07, Test Loss 22.517957808024846
Epoch 12600: Train Loss 2.792300548412679e-07, Test Loss 22.430241141184244
Epoch 12700: Train Loss 2.786359442804761e-07, Test Loss 22.340345898311796
Epoch 12800: Train Loss 2.7804982840164644e-07, Test Loss 22.25438874734061
Epoch 12900: Train Loss 2.7773416277503557e-07, Test Loss 22.170485830125706
Epoch 13000: Train Loss 2.772257462480645e-07, Test Loss 22.083391305220868
Epoch 13100: Train Loss 2.768325103811197e-07, Test Loss 21.99878869201645
Epoch 13200: Train Loss 2.7623998920298234e-07, Test Loss 21.909905127954705
Epoch 13300: Train Loss 2.7586977495510347e-07, Test Loss 21.825840703697978
Epoch 13400: Train Loss 2.7633680839645537e-07, Test Loss 21.743652190631778
Epoch 13500: Train Loss 2.7561626801045614e-07, Test Loss 21.661830778537613
Epoch 13600: Train Loss 2.7473707428459853e-07, Test Loss 21.580497919609584
Epoch 13700: Train Loss 2.7436454549882824e-07, Test Loss 21.5001790336134
Epoch 13800: Train Loss 2.745384658393464e-07, Test Loss 21.421504377993564
Epoch 13900: Train Loss 2.7440941272193975e-07, Test Loss 21.348000412548075
Epoch 14000: Train Loss 2.7349232414350057e-07, Test Loss 21.273381319569665
Epoch 14100: Train Loss 2.737236596509249e-07, Test Loss 21.198730321340072
Epoch 14200: Train Loss 2.729597040247166e-07, Test Loss 21.123693632538014
Epoch 14300: Train Loss 2.724636566785039e-07, Test Loss 21.047488288297632
Epoch 14400: Train Loss 2.7220390118955706e-07, Test Loss 20.97535172233356
Epoch 14500: Train Loss 2.716668784428212e-07, Test Loss 20.903814809317584
Epoch 14600: Train Loss 2.7178386473679516e-07, Test Loss 20.83284987138072
Epoch 14700: Train Loss 2.7099662248429697e-07, Test Loss 20.76100065324923
Epoch 14800: Train Loss 2.7076277241934327e-07, Test Loss 20.688754886480677
Epoch 14900: Train Loss 2.7016780555630675e-07, Test Loss 20.617306931207324
Epoch 15000: Train Loss 2.7006143096844964e-07, Test Loss 20.545309409946267
Epoch 15100: Train Loss 2.6973271116969824e-07, Test Loss 20.477273936273278
Epoch 15200: Train Loss 2.69222765469408e-07, Test Loss 20.41181671386166
Epoch 15300: Train Loss 2.688772119380121e-07, Test Loss 20.34682265170542
Epoch 15400: Train Loss 2.690422889212149e-07, Test Loss 20.27801606762574
Epoch 15500: Train Loss 2.68835822177631e-07, Test Loss 20.212174160286803
Epoch 15600: Train Loss 2.681378902651492e-07, Test Loss 20.15126950434233
Epoch 15700: Train Loss 2.6799020047022144e-07, Test Loss 20.08806893459536
Epoch 15800: Train Loss 2.6742970909060967e-07, Test Loss 20.024661008419777
Epoch 15900: Train Loss 2.6719048462384877e-07, Test Loss 19.96337786007517
Epoch 16000: Train Loss 2.669501450617352e-07, Test Loss 19.90094820989677
```

```
Epoch 16100: Train Loss 2.666352652862406e-07, Test Loss 19.84031481805155
Epoch 16200: Train Loss 2.665148744763257e-07, Test Loss 19.779932529403222
Epoch 16300: Train Loss 2.661910011060449e-07, Test Loss 19.71413540420113
Epoch 16400: Train Loss 2.6591439005607694e-07, Test Loss 19.648836499688635
Epoch 16500: Train Loss 2.6560742570458227e-07, Test Loss 19.587963091606014
Epoch 16600: Train Loss 2.655027604312543e-07, Test Loss 19.52163073145481
Epoch 16700: Train Loss 2.648301943603607e-07, Test Loss 19.45713433193722
Epoch 16800: Train Loss 2.646745290527455e-07, Test Loss 19.392900187910183
Epoch 16900: Train Loss 2.6430889175008734e-07, Test Loss 19.325233677794174
Epoch 17000: Train Loss 2.6413883670602054e-07, Test Loss 19.261773518664867
Epoch 17100: Train Loss 2.63714201213968e-07, Test Loss 19.19251624415752
Epoch 17200: Train Loss 2.641679915970464e-07, Test Loss 19.12738363783729
Epoch 17300: Train Loss 2.633758944062895e-07, Test Loss 19.060861464980583
Epoch 17400: Train Loss 2.640181696358715e-07, Test Loss 18.990242062699476
Epoch 17500: Train Loss 2.626969643849367e-07, Test Loss 18.918208492096113
Epoch 17600: Train Loss 2.627431549620072e-07, Test Loss 18.84489681157512
Epoch 17700: Train Loss 2.623616656966995e-07, Test Loss 18.7703319883759
Epoch 17800: Train Loss 2.6187030999841114e-07, Test Loss 18.692893063957662
Epoch 17900: Train Loss 2.620434058393351e-07, Test Loss 18.615363047590325
Epoch 18000: Train Loss 2.613307993289291e-07, Test Loss 18.53905547546333
Epoch 18100: Train Loss 2.611146694429475e-07, Test Loss 18.457916028633246
Epoch 18200: Train Loss 2.6053564921046764e-07, Test Loss 18.375179595883623
Epoch 18300: Train Loss 2.6023724832004746e-07, Test Loss 18.291742787171504
Epoch 18400: Train Loss 2.607729424121593e-07, Test Loss 18.20414305058444
Epoch 18500: Train Loss 2.59786609214206e-07, Test Loss 18.116582748760937
Epoch 18600: Train Loss 2.599762285770104e-07, Test Loss 18.028376662449997
Epoch 18700: Train Loss 2.590095398406991e-07, Test Loss 17.9386782255589
Epoch 18800: Train Loss 2.5906110017362435e-07, Test Loss 17.84791985419625
Epoch 18900: Train Loss 2.5837485671988837e-07, Test Loss 17.7586806574108
Epoch 19000: Train Loss 2.59085285382657e-07, Test Loss 17.66320649324616
Epoch 19100: Train Loss 2.582808796167183e-07, Test Loss 17.5650094181237
Epoch 19200: Train Loss 2.572633637912052e-07, Test Loss 17.46494862767573
Epoch 19300: Train Loss 2.572825152738313e-07, Test Loss 17.35700168612239
Epoch 19400: Train Loss 2.566063413747103e-07, Test Loss 17.244088658375325
Epoch 19500: Train Loss 2.562339804326755e-07, Test Loss 17.125897403265416
Epoch 19600: Train Loss 2.5606038755315196e-07, Test Loss 17.00344264025629
Epoch 19700: Train Loss 2.5612673604755123e-07, Test Loss 16.874851562920682
Epoch 19800: Train Loss 2.550360051298966e-07, Test Loss 16.744159047843095
Epoch 19900: Train Loss 2.551803677193695e-07, Test Loss 16.603579005491042
Epoch 20000: Train Loss 2.5431337848709146e-07, Test Loss 16.457345422475324
Epoch 20100: Train Loss 2.539414004433693e-07, Test Loss 16.301432806113375
Epoch 20200: Train Loss 2.530323562007634e-07, Test Loss 16.127834962982707
Epoch 20300: Train Loss 2.52552268698652e-07, Test Loss 15.948448516947446
Epoch 20400: Train Loss 2.5206855454681e-07, Test Loss 15.756690104131888
Epoch 20500: Train Loss 2.516128996921646e-07, Test Loss 15.552315417213798
Epoch 20600: Train Loss 2.513208315968991e-07, Test Loss 15.332152995189544
Epoch 20700: Train Loss 2.500758323512368e-07, Test Loss 15.090404830562639
Epoch 20800: Train Loss 2.497428132097746e-07, Test Loss 14.832159040585516
```

```
Epoch 20900: Train Loss 2.484528471598014e-07, Test Loss 14.5490159909109
Epoch 21000: Train Loss 2.4737479898767124e-07, Test Loss 14.242816737463109
Epoch 21100: Train Loss 2.468684826005446e-07, Test Loss 13.910546364296174
Epoch 21200: Train Loss 2.4543321663183016e-07, Test Loss 13.547602377319254
Epoch 21300: Train Loss 2.444367784785341e-07, Test Loss 13.151763210687156
Epoch 21400: Train Loss 2.4293719856771905e-07, Test Loss 12.72023483627164
Epoch 21500: Train Loss 2.4141863536230376e-07, Test Loss 12.238193973186897
Epoch 21600: Train Loss 2.394019288723521e-07, Test Loss 11.69226578820518
Epoch 21700: Train Loss 2.3715824466991393e-07, Test Loss 11.084984955144336
Epoch 21800: Train Loss 2.345821614112085e-07, Test Loss 10.391696798511322
Epoch 21900: Train Loss 2.3142401220319705e-07, Test Loss 9.609330598039506
Epoch 22000: Train Loss 2.2768465111552399e-07, Test Loss 8.710229215016756
Epoch 22100: Train Loss 2.223579004215804e-07, Test Loss 7.679239665634533
Epoch 22200: Train Loss 2.1564641694624824e-07, Test Loss 6.488221928279197
Epoch 22300: Train Loss 2.0641988626585262e-07, Test Loss 5.114573732672333
Epoch 22400: Train Loss 1.941756934196694e-07, Test Loss 3.6277907994803447
Epoch 22500: Train Loss 1.7697237950608655e-07, Test Loss 2.120788453323954
Epoch 22600: Train Loss 1.597317534928197e-07, Test Loss 0.8613986249470285
Epoch 22700: Train Loss 1.4420968108529892e-07, Test Loss 0.2068959573022663
Epoch 22800: Train Loss 1.3078141814000665e-07, Test Loss 0.022912131156413006
Epoch 22900: Train Loss 1.1887624942445576e-07, Test Loss 0.0007046244538797306
Epoch 23000: Train Loss 1.1082432574281059e-07, Test Loss 2.1371932743634757e-05
Epoch 23100: Train Loss 1.0663132881610256e-07, Test Loss 2.2830343432812795e-06
Epoch 23200: Train Loss 1.0404317517844485e-07, Test Loss 7.767478227938102e-07
Epoch 23300: Train Loss 1.0220921200307207e-07, Test Loss 4.357209215580533e-07
Epoch 23400: Train Loss 1.0088398300015506e-07, Test Loss 3.091452030399501e-07
Epoch 23500: Train Loss 9.992837448721953e-08, Test Loss 2.4983269625982814e-07
Epoch 23600: Train Loss 9.92153034898023e-08, Test Loss 2.1741308666372775e-07
Epoch 23700: Train Loss 9.867784467986747e-08, Test Loss 1.977657696522551e-07
Epoch 23800: Train Loss 9.82429389321986e-08, Test Loss 1.8450607128812502e-07
Epoch 23900: Train Loss 9.788225508463568e-08, Test Loss 1.7496568897134502e-07
Epoch 24000: Train Loss 9.757359253983101e-08, Test Loss 1.6784643414253663e-07
Epoch 24100: Train Loss 9.731044435378818e-08, Test Loss 1.623772783707543e-07
Epoch 24200: Train Loss 9.707678633216759e-08, Test Loss 1.580274130190408e-07
Epoch 24300: Train Loss 9.687759341367583e-08, Test Loss 1.5435983032576414e-07
Epoch 24400: Train Loss 9.669566941368298e-08, Test Loss 1.513403722245134e-07
Epoch 24500: Train Loss 9.653359176490059e-08, Test Loss 1.4872395588332833e-07
Epoch 24600: Train Loss 9.638498711033151e-08, Test Loss 1.4648661913061672e-07
Epoch 24700: Train Loss 9.624721757500805e-08, Test Loss 1.4459082707753333e-07
Epoch 24800: Train Loss 9.612407710563697e-08, Test Loss 1.4287798170026138e-07
Epoch 24900: Train Loss 9.600568579230366e-08, Test Loss 1.413888794502117e-07
Epoch 25000: Train Loss 9.589689327141944e-08, Test Loss 1.4007408614072522e-07
```

3 Model Analysis

[10]:
```
line([train_losses[::100],
      test_losses[::100]],
     x=np.arange(0, len(train_losses), 100),
     xaxis="Epoch",
     yaxis="Loss",
     log_y=True,
     title="Training Curve for Modular Subtraction",
     line_labels=['train', 'test'],
     toggle_x=True,
     toggle_y=True)
```



[11]:
```
original_logits, cache = model.run_with_cache(dataset)

# Printing total number of elements in the original_logits tensor
print("Number of elements in logits:", original_logits.numel())

# Extracting embedding weights, excluding the last row
W_E = model.embed.W_E[:-1]
print("W_E shape:", W_E.shape)

# Computing a transformation through the network's first block
W_neur = W_E @ model.blocks[0].attn.W_V @ model.blocks[0].attn.W_O @ model.
 ↪blocks[0].mlp.W_in
print("W_neur shape:", W_neur.shape)

# Calculating the transformation from the final MLP output back to logits
W_logit = model.blocks[0].mlp.W_out @ model.unembed.W_U
print("W_logit shape:", W_logit.shape)
```

```
Number of elements in logits: 4328691
W_E shape: torch.Size([113, 128])
W_neur shape: torch.Size([4, 113, 512])
```

```
W_logit shape: torch.Size([512, 113])
```

[12]:
```python
# Calculating and printing the loss for the original logits
original_loss = loss_fn(original_logits, labels).item()
print("Original Loss:", original_loss)
```

```
Original Loss: 1.26823040798941e-07
```

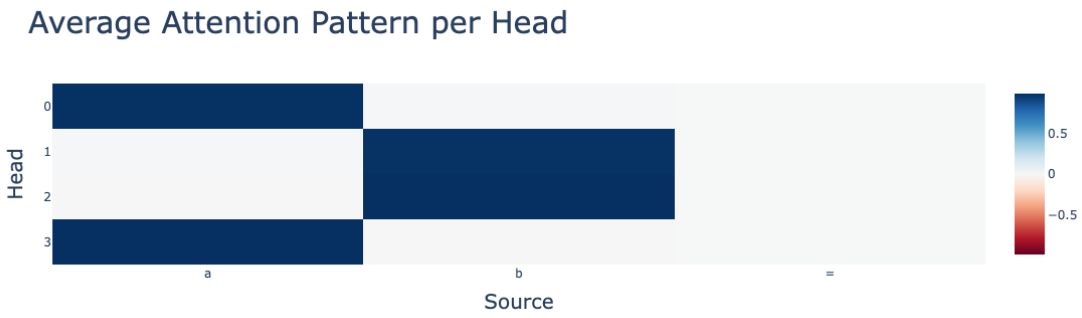
3.1 Attention Heads Analysis

[13]:
```python
# Extract attention patterns for the last head's first two columns
pattern_a = cache["pattern", 0, "attn"][:, :, -1, 0]  # Last head, first column
pattern_b = cache["pattern", 0, "attn"][:, :, -1, 1]  # Last head, second column
print("pattern_a shape:", pattern_a.shape)
print("pattern_b shape:", pattern_b.shape)

# Extract MLP layer's post-activations and pre-activations at the last position
neuron_acts = cache["post", 0, "mlp"][:, -1, :]  # Post-activations
neuron_pre_acts = cache["pre", 0, "mlp"][:, -1, :]  # Pre-activations
print("neuron_acts shape:", neuron_acts.shape)
print("neuron_pre_acts shape:", neuron_pre_acts.shape)

# Print the shapes of all cached items to understand what data is being stored
for param_name, param in cache.items():
    print(param_name, param.shape)
```

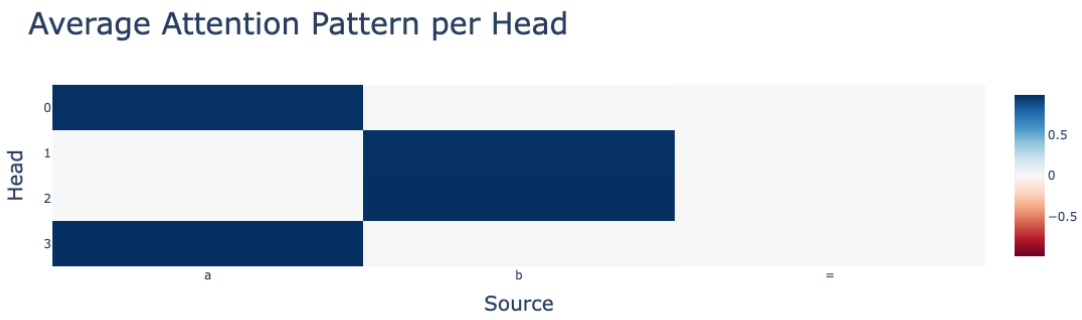
```
pattern_a shape: torch.Size([12769, 4])
pattern_b shape: torch.Size([12769, 4])
neuron_acts shape: torch.Size([12769, 512])
neuron_pre_acts shape: torch.Size([12769, 512])
hook_embed torch.Size([12769, 3, 128])
hook_pos_embed torch.Size([12769, 3, 128])
blocks.0.hook_resid_pre torch.Size([12769, 3, 128])
blocks.0.attn.hook_q torch.Size([12769, 3, 4, 32])
blocks.0.attn.hook_k torch.Size([12769, 3, 4, 32])
blocks.0.attn.hook_v torch.Size([12769, 3, 4, 32])
blocks.0.attn.hook_attn_scores torch.Size([12769, 4, 3, 3])
blocks.0.attn.hook_pattern torch.Size([12769, 4, 3, 3])
blocks.0.attn.hook_z torch.Size([12769, 3, 4, 32])
blocks.0.hook_attn_out torch.Size([12769, 3, 128])
blocks.0.hook_resid_mid torch.Size([12769, 3, 128])
blocks.0.mlp.hook_pre torch.Size([12769, 3, 512])
blocks.0.mlp.hook_post torch.Size([12769, 3, 512])
blocks.0.hook_mlp_out torch.Size([12769, 3, 128])
blocks.0.hook_resid_post torch.Size([12769, 3, 128])
```

[14]:
```python
imshow(cache["pattern", 0].mean(dim=0)[:, -1, :],
       title="Average Attention Pattern per Head",
```

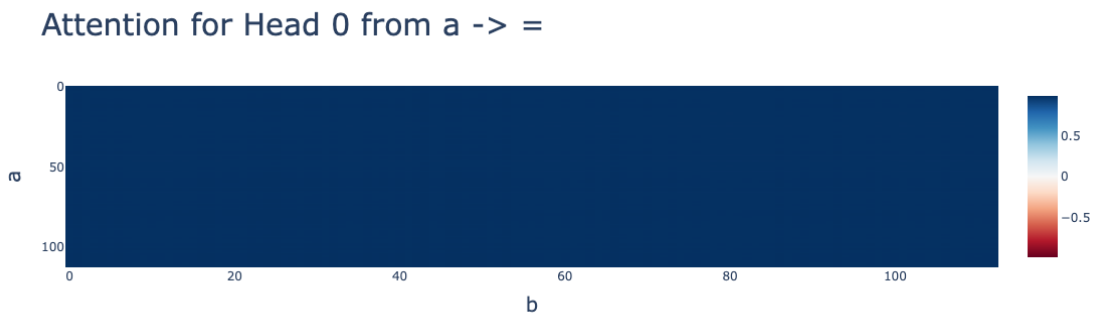
```
       xaxis="Source",
       yaxis="Head",
       x=['a', 'b', '='])
```



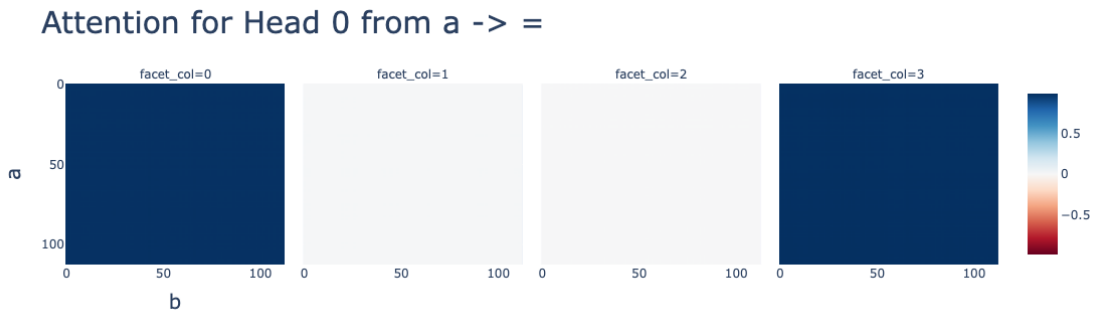
[15]:
```
imshow(cache["pattern", 0][5][:, -1, :],
       title="Average Attention Pattern per Head",
       xaxis="Source",
       yaxis="Head",
       x=['a', 'b', '='])
```



[16]:
```
imshow(cache["pattern", 0][:, 0, -1, 0].reshape(p, p),
       title="Attention for Head 0 from a -> =",
       xaxis="b",
       yaxis="a")
```



[17]:
```
imshow(einops.rearrange(cache["pattern", 0][:, :, -1, 0], "(a b) head -> head a␣
↪b", a=p, b=p),
       title="Attention for Head 0 from a -> =",
       xaxis="b",
       yaxis="a",
       facet_col=0)
```



3.2 Singular Value Decomposition

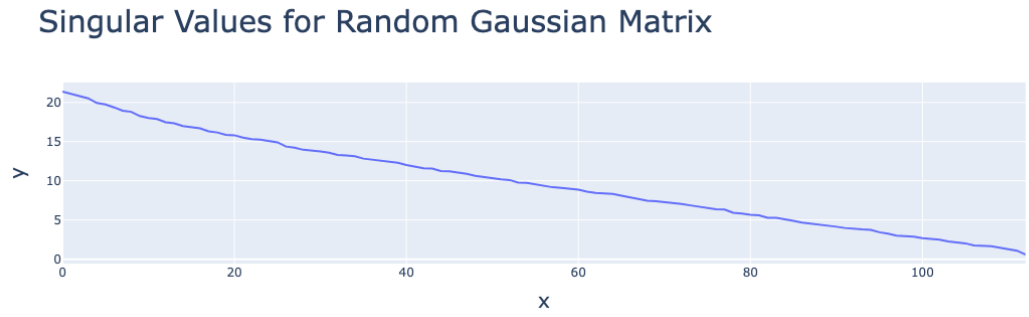
[18]:
```
W_E.shape
```

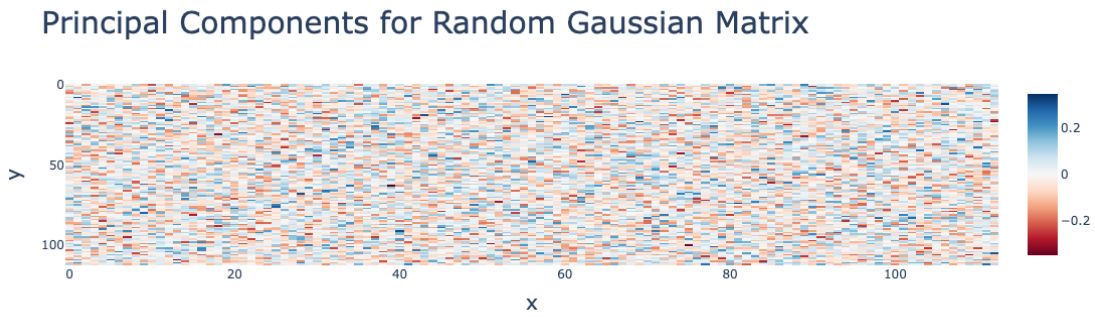
[18]:
```
torch.Size([113, 128])
```

First, we analyze the SVD of a RANDOM Gaussian matrix to serve as a baseline/control. Notice how the singular values are linearly decrease in importance. The singular values exhibit a wide range of values.

[19]:
```
# CONTROL - SVD on RANDOM Gaussian matrix
U, S, Vh = torch.svd(torch.randn_like(W_E))

line(S, title="Singular Values for Random Gaussian Matrix")
```

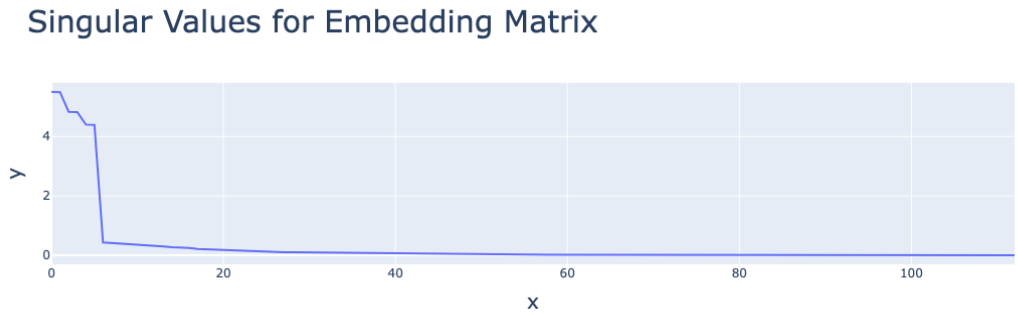
```
imshow(U, title="Principal Components for Random Gaussian Matrix")
```

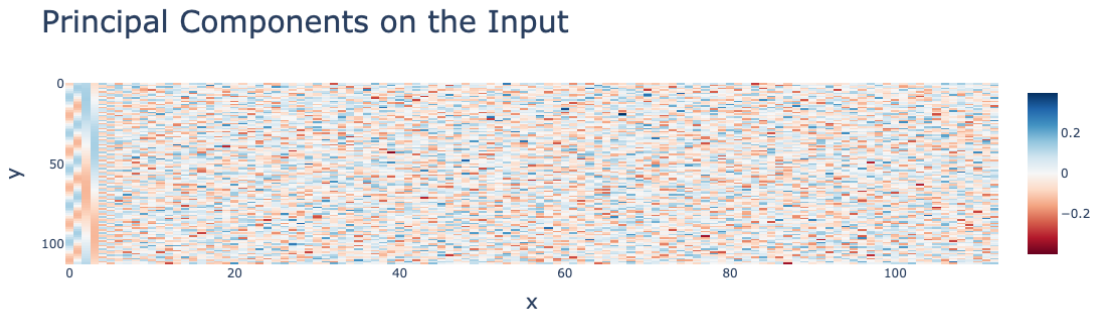




In contrast, SVD on our weight matrix yields only ~6 nontrivial singular values!

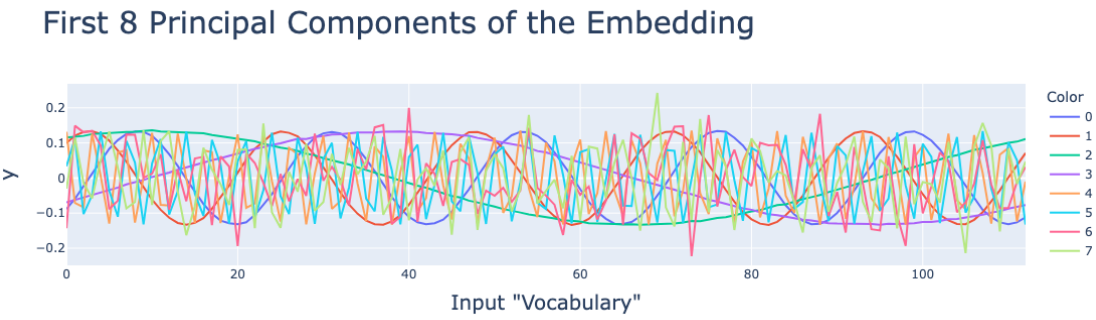
[20]:
```
# SVD on our weight matrix
U, S, Vh = torch.svd(W_E)

line(S, title="Singular Values for Embedding Matrix")
imshow(U, title="Principal Components on the Input")
```





We visualize the first few (8) principal components, extracted from U.

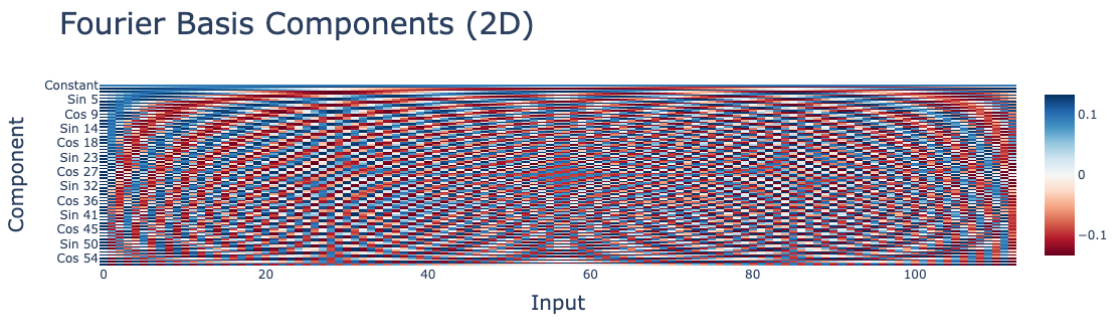
[21]:
```
# Extract the first 8 principal components from U
principal_components = U[:, :8].T  # transpose to make each row a PC
line(principal_components,
     title="First 8 Principal Components of the Embedding",
     xaxis='''Input "Vocabulary"''')
```



3.3 Fourier Basis Analysis

[22]:
```
fourier_basis = []
fourier_basis_names = []

# append a constant basis vector
fourier_basis.append(torch.ones(p))
fourier_basis_names.append("Constant")

# generate sine and cosine basis vectors
for freq in range(1, p//2+1):
```

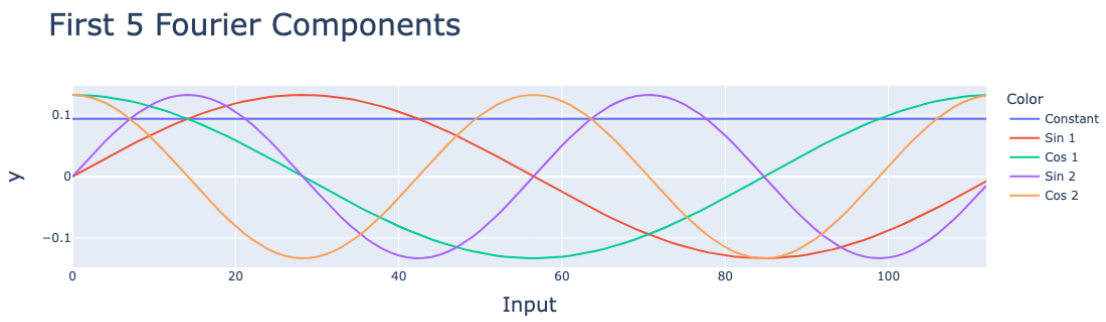
```
    # append sine components
    fourier_basis.append(torch.sin(torch.arange(p)*2 * torch.pi * freq / p))
    fourier_basis_names.append(f"Sin {freq}")

    # append cosine components
    fourier_basis.append(torch.cos(torch.arange(p)*2 * torch.pi * freq / p))
    fourier_basis_names.append(f"Cos {freq}")

fourier_basis = torch.stack(fourier_basis, dim=0).to(DEVICE)

# normalize each basis vector to have unit norm
fourier_basis = fourier_basis/fourier_basis.norm(dim=-1, keepdim=True)

imshow(fourier_basis,
       title="Fourier Basis Components (2D)",
       xaxis="Input",
       yaxis="Component",
       y=fourier_basis_names)
```

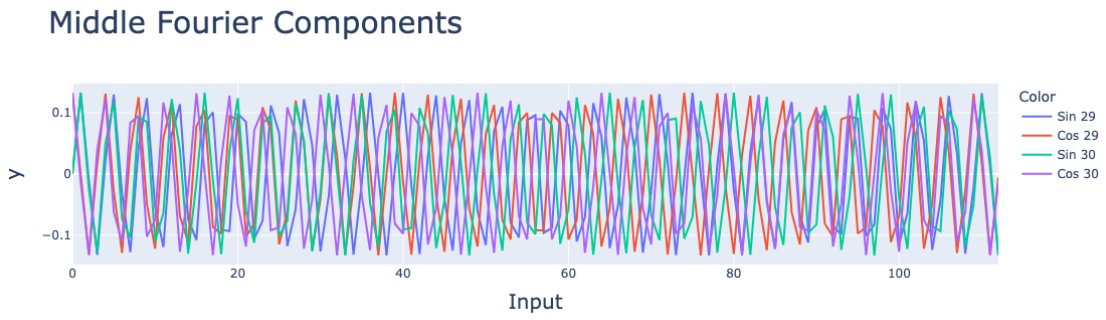


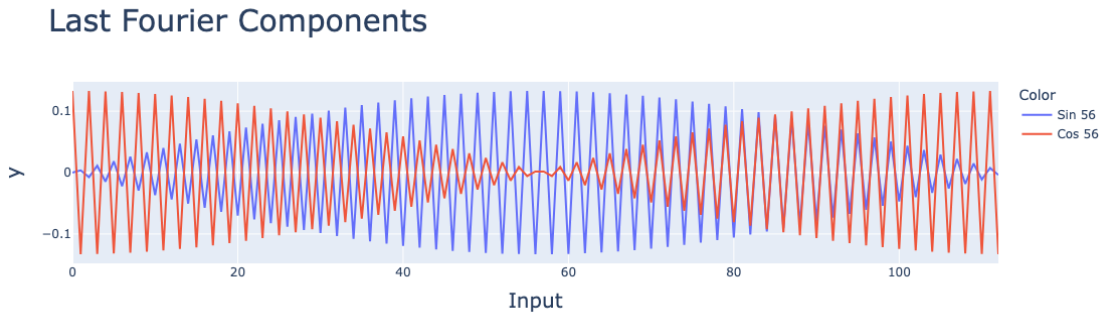
Below, we plot slices of the standard Fourier basis for p=113.

[23]:
```
# plot the first 5 Fourier components
line(fourier_basis[:5],
     xaxis="Input",
     line_labels=fourier_basis_names[:5],
     title="First 5 Fourier Components")

# plot middle range Fourier components
line(fourier_basis[57:61],
     xaxis="Input",
     line_labels=fourier_basis_names[57:61],
     title="Middle Fourier Components")

# plot last 2 Fourier components
```

```
line(fourier_basis[111:113],
     xaxis="Input",
     line_labels=fourier_basis_names[111:113],
     title="Last Fourier Components")
```

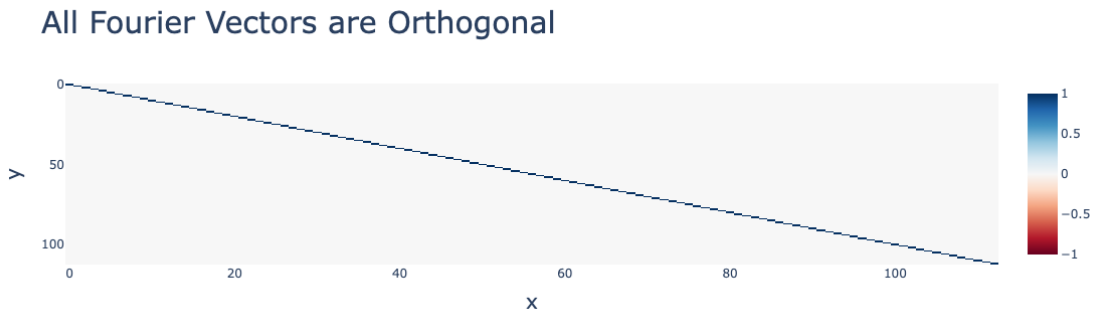






Below, we illustrate that all standard Fourier basis vectors are orthogonal.

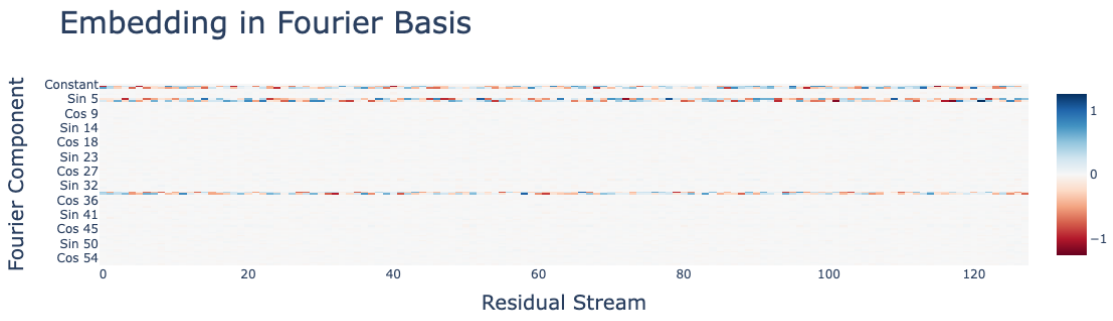
[24]:
```
imshow(fourier_basis @ fourier_basis.T,
       title="All Fourier Vectors are Orthogonal")
```



3.4 Projecting the Weight Matrix to the Fourier basis (to observe frequency)

After projecting our weight matrix to the Fourier basis, we observe that of the 113 Fourier components, only **6** of the basis vectors are used meaningfully! The embedding matrix and projection are low rank.

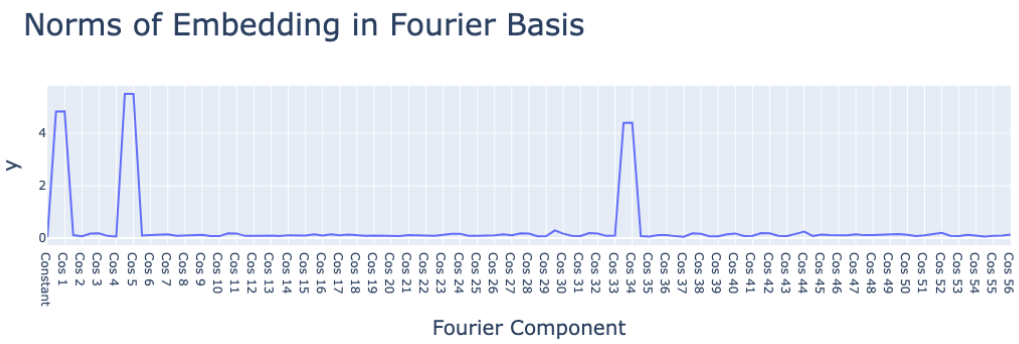
[25]:
```
# Compute the projection of the embedding weights onto the Fourier basis
projection = fourier_basis @ W_E

# heatmap where each cell represents the interaction strength between a
# Fourier component and a dimension of the embedding space
imshow(projection,
       yaxis="Fourier Component",
       xaxis="Residual Stream",
       y=fourier_basis_names,
       title="Embedding in Fourier Basis")
```



Next, we calculate the norm of each projection vector, which provides a measure of how much each

Fourier component contributes to the embedding space, effectively summarizing the projection strength. We observe that **sin(1), cos(1), sin(5), cos(5), sin(34), and cos(34)** are the most influential.

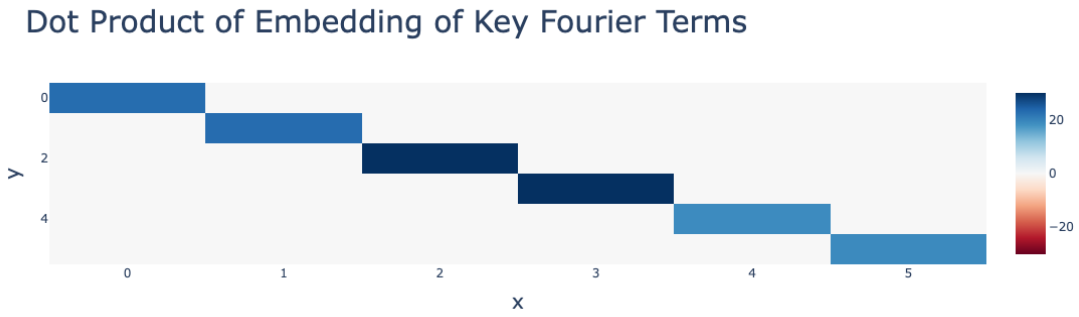
[26]:
```
# calculate the norm of each projection vector
projection_norms = projection.norm(dim=-1)

# Plot the norms of these projections to understand which Fourier components
# have the strongest influence in the embedding space.
line((fourier_basis @ W_E).norm(dim=-1),
     xaxis="Fourier Component",
     x=fourier_basis_names,
     title="Norms of Embedding in Fourier Basis")
```



Below, we explicitly identify the key frequencies which are crucial for further analysis. The other frequencies are essentially zero, so are discarded in further analysis.

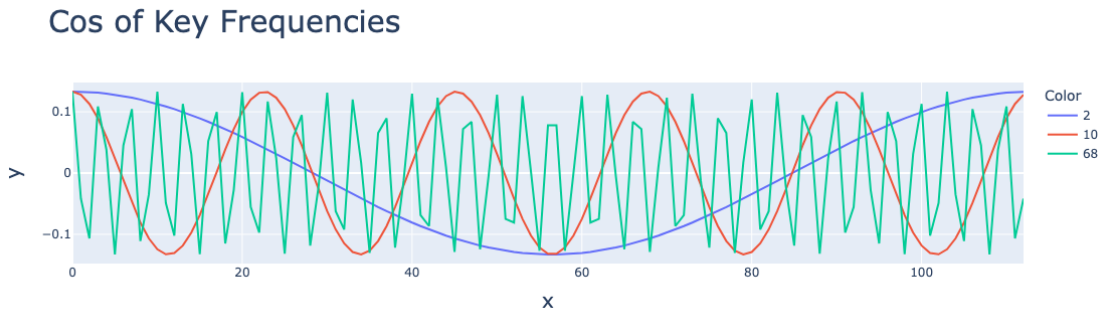
[27]:
```
# key frequencies
key_freqs = [1, 5, 34]

# indices for the Fourier basis, since each frequency has
# two corresponding indices in the Fourier basis for its sine and cosine components
key_freq_indices = [1, 2, 9, 10, 67, 68]  # Sine and Cosine indices for each frequency

# extract projections corresponding to the key frequencies
key_fourier_embed = projection[key_freq_indices]
print("key_fourier_embed", key_fourier_embed.shape)  # shape for verification

# dot product of the key Fourier embeddings with themselves
# represents the interactions between these key Fourier components
imshow(key_fourier_embed @ key_fourier_embed.T,
       title="Dot Product of Embedding of Key Fourier Terms")
```

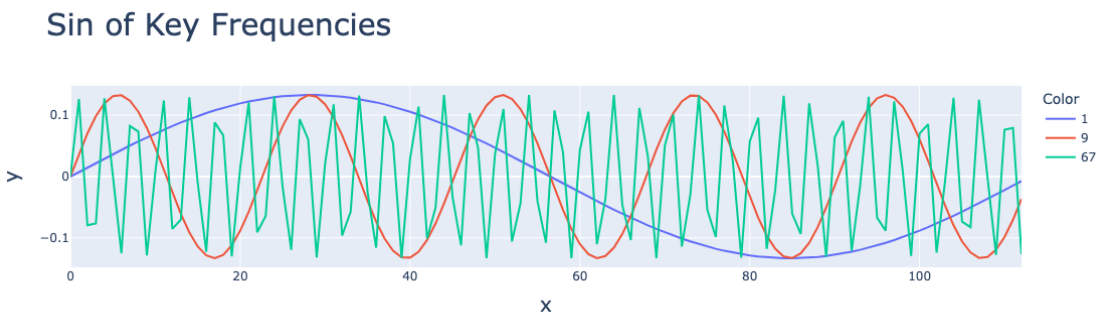
```
key_fourier_embed torch.Size([6, 128])
```



3.5 Key Frequencies Visualization

[28]:
```
line(fourier_basis[[2, 10, 68]],
     title="Cos of Key Frequencies",
     line_labels=[2, 10, 68])

line(fourier_basis[[1, 9, 67]],
     title="Sin of Key Frequencies",
     line_labels=[1, 9, 67])
```





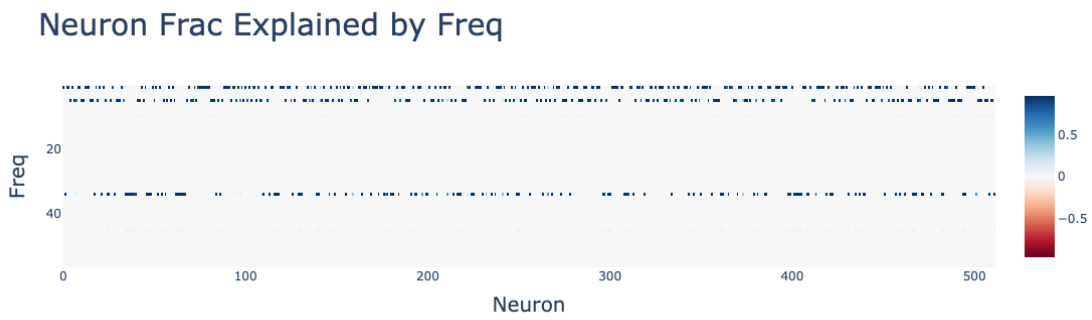
3.5.1 Neuron Clusters

[29]:
```
# project neuron activations to Fourier basis
fourier_neuron_acts = fourier_basis @ einops.rearrange(neuron_acts, "(a b)␣
 ↪neuron -> neuron a b", a=p, b=p) @ fourier_basis.T

# center these by removing the mean
fourier_neuron_acts[:, 0, 0] = 0.
print("fourier_neuron_acts", fourier_neuron_acts.shape)
```

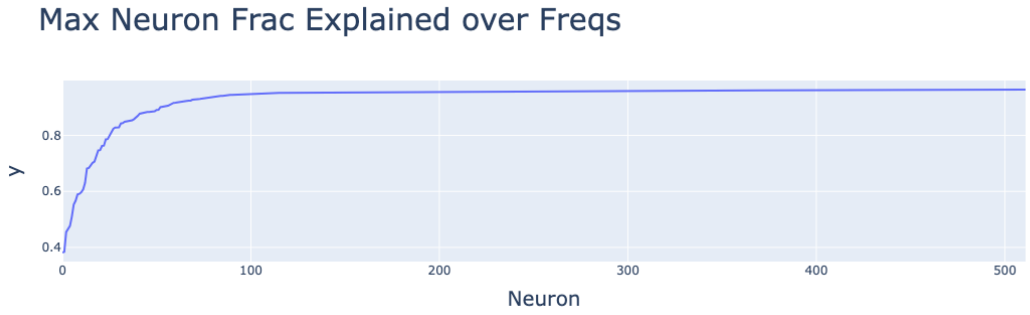
```
fourier_neuron_acts torch.Size([512, 113, 113])
```

Observe that each neuron's variance is substantially explained by activation along a few key frequencies.

[30]:
```
neuron_freq_norm = torch.zeros(p//2, model.cfg.d_mlp).to(DEVICE)
for freq in range(0, p//2):
    # consider both the sine and cosine components for each frequency
    for x in [0, 2*(freq+1) - 1, 2*(freq+1)]:
        for y in [0, 2*(freq+1) - 1, 2*(freq+1)]:
            neuron_freq_norm[freq] += fourier_neuron_acts[:, x, y]**2
neuron_freq_norm = neuron_freq_norm / fourier_neuron_acts.pow(2).sum(dim=[-1,␣
 ↪-2])[None, :]

# show what fraction of the neuron's variance is explained by each frequency.
imshow(neuron_freq_norm,
       xaxis="Neuron",
       yaxis="Freq",
       y=torch.arange(1, p//2+1),
       title="Neuron Frac Explained by Freq")
```



[31]:
```
line(neuron_freq_norm.max(dim=0).values.sort().values,
     xaxis="Neuron",
     title="Max Neuron Frac Explained over Freqs")
```



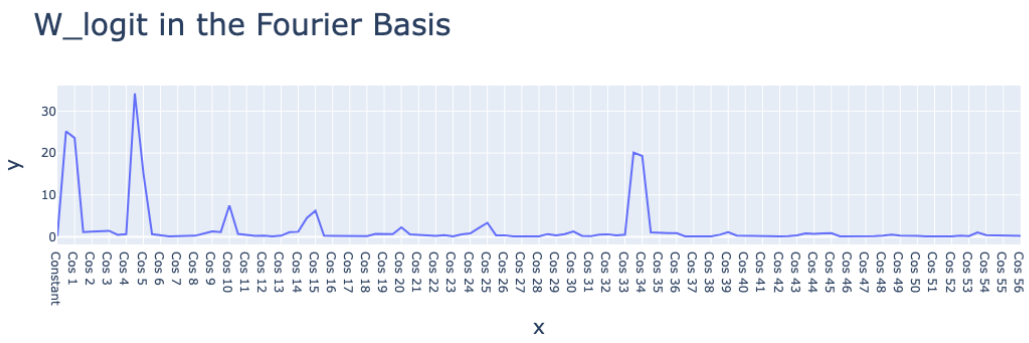
3.6 Neuron-Logit Weights Analysis

[32]:
```
# Multiply MLP output weights by the UN-embedding weights to project the␣
 ↪transformed outputs
# back to the input vocabulary space. This translates the learned␣
 ↪representations into
# interpretable outputs
W_logit = model.blocks[0].mlp.W_out @ model.unembed.W_U
print("W_logit", W_logit.shape)
```

```
W_logit torch.Size([512, 113])
```

Projecting the logit weight matrix (W_L) into the frequency domain (via Fourier transform) reveals that the outputs of this model consist of linear combinations of a sines and cosines of the key frequencies.

[33]:
```
line((W_logit @ fourier_basis.T).norm(dim=0),
     x=fourier_basis_names,
     title="W_logit in the Fourier Basis")
```



4 Black Box Methods & Progress Measures

4.1 Setup Code

[34]:
```
def test_logits(logits, bias_correction=False, original_logits=None,␣
↪mode="all"):
    """
    Adjusts and evaluates logits according to specified testing conditions.
    """
    # Ensure the logits are in the correct shape [p*p, p].
    if logits.shape[1] == p * p:
        logits = logits.T  # Transpose to shape [p*p, p+1]
    if logits.shape == torch.Size([p * p, p + 1]):
        logits = logits[:, :-1]

    # Reshape logits to ensure each row corresponds to a potential input
    logits = logits.reshape(p * p, p)

    # Apply bias correction if enabled
    if bias_correction:
        if original_logits is None:
            raise ValueError("Original logits must be provided for bias␣
↪correction.")

        # Calculate the mean difference between original logits and current␣
↪logits across all batches
        # Then adjust the current logits by adding this mean difference to each
        mean_difference = einops.reduce(original_logits - logits, "batch ... ->␣
↪...", "mean")
```

```
        logits += mean_difference

    # Compute the loss based on the specified mode.
    if mode == "train":
        return loss_fn(logits[train_indices], labels[train_indices])
    elif mode == "test":
        return loss_fn(logits[test_indices], labels[test_indices])
    elif mode == "all":
        return loss_fn(logits, labels)
```

[35]:
```
metric_cache = {}

def get_metrics(model, metric_cache, metric_fn, name, reset=False):
    """
    Evaluate and cache the metric results for a model at various checkpoints.
    """
    if reset or (name not in metric_cache) or (len(metric_cache[name]) == 0):
        metric_cache[name] = []

        for c, sd in enumerate(tqdm.tqdm(model_checkpoints)):
            model.reset_hooks()
            model.load_state_dict(sd)
            out = metric_fn(model)

            if isinstance(out, torch.Tensor):
                out = utils.to_numpy(out)

            metric_cache[name].append(out)

        model.load_state_dict(model_checkpoints[-1])

        try:
            metric_cache[name] = torch.tensor(metric_cache[name])
        except TypeError:  # Handle cases where the conversion fails
            metric_cache[name] = torch.tensor(np.array(metric_cache[name]))
```

4.2 Defining Progress Measures

4.2.1 Loss Curves

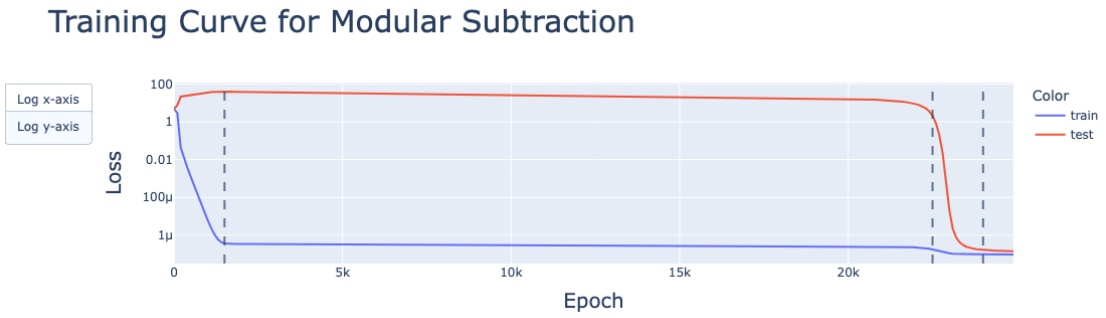
These epoch numbers are estimated from the plot and are useful for visualization.

[36]:
```
memorization_end_epoch = 1500
circuit_formation_end_epoch = 22500
cleanup_end_epoch = 24000
```

[37]:
```
def add_lines(figure):
    figure.add_vline(memorization_end_epoch, line_dash="dash", opacity=0.7)
```

```
    figure.add_vline(circuit_formation_end_epoch, line_dash="dash", opacity=0.7)
    figure.add_vline(cleanup_end_epoch, line_dash="dash", opacity=0.7)
    return figure
```

[38]:
```
fig = line([train_losses[::100],
            test_losses[::100]],
            x=np.arange(0, len(train_losses), 100),
            xaxis="Epoch",
            yaxis="Loss",
            log_y=True,
            title="Training Curve for Modular Subtraction",
            line_labels=['train', 'test'],
            toggle_x=True,
            toggle_y=True,
            return_fig=True)
add_lines(fig)
```



4.2.2 Logit Periodicity

[39]:
```
all_logits = original_logits[:, -1, :]
print(all_logits.shape)

all_logits = einops.rearrange(all_logits, "(a b) c -> a b c", a=p, b=p)
print(all_logits.shape)
```

```
torch.Size([12769, 113])
torch.Size([113, 113, 113])
```

Following our hypothesized formula, we begin constructing the logits that our formula would predict.

[40]:
```
coses = {}
for freq in key_freqs:
    print("Freq:", freq)
    a = torch.arange(p)[:, None, None]
```

```
    b = torch.arange(p)[None, :, None]
    c = torch.arange(p)[None, None, :]
    cube_predicted_logits = torch.cos(freq * 2 * torch.pi / p * (a - b - c)).
↪to(DEVICE)
    cube_predicted_logits /= cube_predicted_logits.norm()
    coses[freq] = cube_predicted_logits
```

```
Freq: 1
Freq: 5
Freq: 34
```

Observe that the cosine similarities are nontrivial: 0.3824, 0.6626, 0.2160, and for the residual 0.6067.

[41]:
```
approximated_logits = torch.zeros_like(all_logits)
for freq in key_freqs:
    print("Freq:", freq)

    # represents how much of `all_logits` is in the direction of the cosine␣
↪pattern.
    coeff = (all_logits * coses[freq]).sum()
    print("Coeff:", coeff)

    # normalized measure of how much the cosine pattern aligns with␣
↪`all_logits`.
    cosine_sim = coeff / all_logits.norm()
    print("Cosine Sim:", cosine_sim, "\n\n")

    # builds approx by adding contributions from each significant cosine␣
↪pattern.
    approximated_logits += coeff * coses[freq]

residual = all_logits - approximated_logits
print("Residual size:", residual.norm())
print("Residual fraction of norm:", residual.norm()/all_logits.norm())
```

```
Freq: 1
Coeff: tensor(19212.0059, device='cuda:0', grad_fn=<SumBackward0>)
Cosine Sim: tensor(0.3824, device='cuda:0', grad_fn=<DivBackward0>)


Freq: 5
Coeff: tensor(33291.1484, device='cuda:0', grad_fn=<SumBackward0>)
Cosine Sim: tensor(0.6626, device='cuda:0', grad_fn=<DivBackward0>)


Freq: 34
Coeff: tensor(10850.5361, device='cuda:0', grad_fn=<SumBackward0>)
```

```
Cosine Sim: tensor(0.2160, device='cuda:0', grad_fn=<DivBackward0>)


Residual size: tensor(30483.6914, device='cuda:0',
grad_fn=<LinalgVectorNormBackward0>)
Residual fraction of norm: tensor(0.6067, device='cuda:0',
grad_fn=<DivBackward0>)
```

For a random vector, the cosine similarity is very small! This corroborates the strength of our approximation.

[42]:
```
random_logit_cube = torch.randn_like(all_logits)
print((all_logits * random_logit_cube).sum()/random_logit_cube.norm()/
 ↪all_logits.norm())
```

```
tensor(-1.6255e-05, device='cuda:0', grad_fn=<DivBackward0>)
```

The loss using the approximated logits remains relatively stable (same magnitude) as the true logits of the model. This supports our hypothesized formula approximation.

[43]:
```
test_logits(all_logits)
```

[43]:
```
tensor(1.2682e-07, device='cuda:0', dtype=torch.float64,
       grad_fn=<NegBackward0>)
```

[44]:
```
test_logits(approximated_logits)
```

[44]:
```
tensor(4.2997e-07, device='cuda:0', dtype=torch.float64,
       grad_fn=<NegBackward0>)
```

We extend this to the training loop.

[45]:
```
cos_cube = []
for freq in range(1, p//2 + 1):
    a = torch.arange(p)[:, None, None]
    b = torch.arange(p)[None, :, None]
    c = torch.arange(p)[None, None, :]

    # calculate cosine values across the 3D grid.
    # cosine wave pattern is based on the formula: cos(freq * 2 * pi / p * (a -␣
 ↪b - c))
    cube_predicted_logits = torch.cos(freq * 2 * torch.pi / p * (a - b - c)).
 ↪to(DEVICE)

    # Normalize the cosine pattern to have unit norm.
    cube_predicted_logits /= cube_predicted_logits.norm()
    cos_cube.append(cube_predicted_logits)
cos_cube = torch.stack(cos_cube, dim=0)
print(cos_cube.shape)
```

```
torch.Size([56, 113, 113, 113])
```

[48]:
```python
def get_cos_coeffs(model):
    """
    Calculate the coefficients of cosine wave patterns for the model's output␣
 ↪logits.
    """
    logits = model(dataset)[:, -1]
    logits = einops.rearrange(logits, "(a b) c -> a b c", a=p, b=p)

    # projects the logits onto the space defined by each cosine pattern,
    # effectively measuring how much each pattern is represented in the logits.
    vals = (cos_cube * logits[None, :, :, :]).sum([-3, -2, -1])
    return vals

get_metrics(model, metric_cache, get_cos_coeffs, "cos_coeffs")
print("Cached cosine coefficients shape:", metric_cache["cos_coeffs"].shape)
```

```
Cached cosine coefficients shape: torch.Size([250, 56])
```

[49]:
```python
def get_cos_sim(model):
    """
    Calculate cosine similarity between the model's output logits and␣
 ↪predefined cosine patterns.
    """
    logits = model(dataset)[:, -1]
    logits = einops.rearrange(logits, "(a b) c -> a b c", a=p, b=p)

    # Calculate the dot product of logits and cosine patterns, then sum over␣
 ↪spatial dimensions.
    vals = (cos_cube * logits[None, :, :, :]).sum([-3, -2, -1])
    return vals / logits.norm()

get_metrics(model, metric_cache, get_cos_sim, "cos_sim")
print(metric_cache["cos_sim"].shape)
```

```
  0%|          | 0/250 [00:00<?, ?it/s]
torch.Size([250, 56])
```

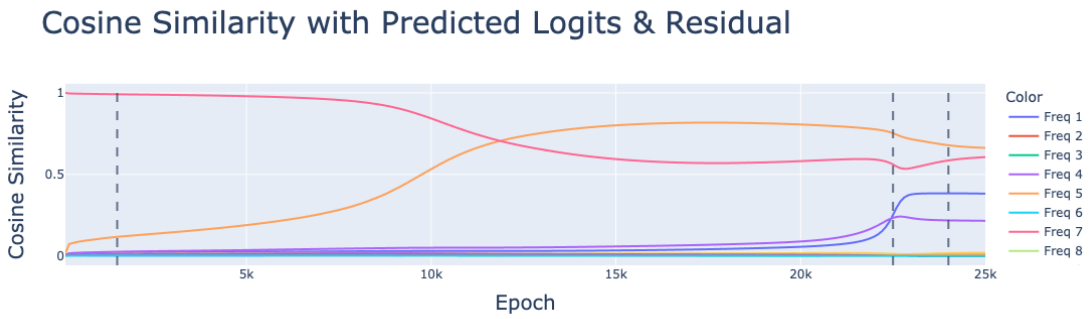
We confirm that output increasingly relies on the key frequencies as training progresses. The outputs **gradually** coalesce around these key frequencies—despite grokking feeling sudden!

Observe the importance of frequency 5 developing over time (along with the key frequencies 1 and 34). This provides a leading indicator of the eventual grok!

Further, we notice that in the beginning, the residual line shows that almost nothing can be explained by the cosine key frequencies (residuals have a value of 1), but near grokking, the residual value decreases significantly (as the key frequencies are able to explain the logits well).

[50]:
```
def get_residual_cos_sim(model):
    """
    Calculate the residual cosine similarity of model outputs relative to␣
 ↪logits predicted by the formula AND
    the residuals (the proportion of the logits that CANNOT be explained by the␣
 ↪cosine basis).
    """
    logits = model(dataset)[:, -1] # get last logits
    logits = einops.rearrange(logits, "(a b) c -> a b c", a=p, b=p)

    # Project logits onto each cosine pattern & sum results to get a scalar␣
 ↪value for each pattern
    vals = (cos_cube * logits[None, :, :, :]).sum([-3, -2, -1])

    residual = logits - (vals[:, None, None, None] * cos_cube).sum(dim=0)
    return residual.norm() / logits.norm()

get_metrics(model, metric_cache, get_residual_cos_sim, "residual_cos_sim")
print(metric_cache["residual_cos_sim"].shape)

fig = line([metric_cache["cos_sim"][:, i] for i in range(p//
 ↪2)]+[metric_cache["residual_cos_sim"]],
           line_labels=[f"Freq {i}" for i in range(1, p//2+1)]+["residual"],
           title="Cosine Similarity with Predicted Logits & Residual",
           xaxis="Epoch", x=checkpoint_epochs,
           yaxis="Cosine Similarity",
           return_fig=True)
add_lines(fig)
```

```
torch.Size([250])
```



4.3 Restricted Loss

We evaluate the restricted loss during the training loop. The restricted loss when we ablate every non-key frequency (i.e. we remove all frequencies except for 1, 5, and 34), and assess how the model performs. If our hypothesis is correct, then the model will still exhibit acceptable performance (i.e. low loss).

[51]:
```
def get_restricted_loss(model):
    logits, cache = model.run_with_cache(dataset)
    logits = logits[:, -1, :]
    neuron_acts = cache["post", 0, "mlp"][:, -1, :]

    approx_neuron_acts = torch.zeros_like(neuron_acts)

    a = torch.arange(p)[:, None]
    b = torch.arange(p)[None, :]
    for freq in key_freqs:
        # create & normalize a cosine wave matrix
        cos_apb_vec = torch.cos(freq * 2 * torch.pi / p * (a - b)).to(DEVICE)
        cos_apb_vec /= cos_apb_vec.norm()
        cos_apb_vec = einops.rearrange(cos_apb_vec, "a b -> (a b) 1")

        # project neuron activations onto cosine basis and add to approximations
        approx_neuron_acts += (neuron_acts * cos_apb_vec).sum(dim=0) *␣
↪cos_apb_vec

        # create & normalize a sine wave matrix
        sin_apb_vec = torch.sin(freq * 2 * torch.pi / p * (a - b)).to(DEVICE)
        sin_apb_vec /= sin_apb_vec.norm()
        sin_apb_vec = einops.rearrange(sin_apb_vec, "a b -> (a b) 1")

        # project neuron activations onto sine basis and add to approximations
        approx_neuron_acts += (neuron_acts * sin_apb_vec).sum(dim=0) *␣
↪sin_apb_vec

    restricted_logits = approx_neuron_acts @ model.blocks[0].mlp.W_out @ model.
↪unembed.W_U

    # Add bias term
    restricted_logits += logits.mean(dim=0, keepdim=True) - restricted_logits.
↪mean(dim=0, keepdim=True)
    return loss_fn(restricted_logits[test_indices], test_labels).item()

print("Restricted Loss:", get_restricted_loss(model))
```

```
Restricted Loss: 2.7462999527549743e-05
```

[52]:
```
# clear cache memory
torch.cuda.empty_cache()
```

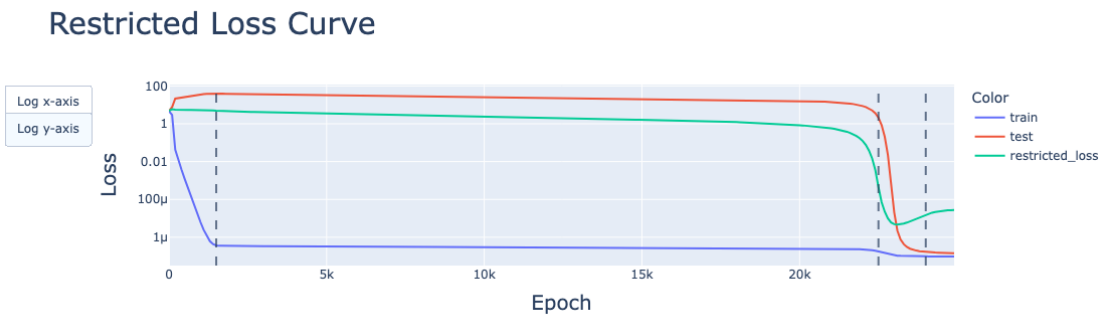
[53]:
```
get_metrics(model, metric_cache, get_restricted_loss, "restricted_loss",␣
 ↪reset=True)
print(metric_cache["restricted_loss"].shape)
```

```
  0%|          | 0/250 [00:00<?, ?it/s]
```

```
torch.Size([250])
```

As desired, our restricted loss nearly matches the train/test losses near 0 once the grok occurs. We expected it to remain relatively constant beforehand because the model is only able to learn using the key frequencies. This plot further validates our hypothesis.

[54]:
```
fig = line([train_losses[::100],
            test_losses[::100],
            metric_cache["restricted_loss"]],
           x=np.arange(0, len(train_losses), 100),
           xaxis="Epoch", yaxis="Loss",
           log_y=True,
           title="Restricted Loss Curve",
           line_labels=['train', 'test', "restricted_loss"],
           toggle_x=True,
           toggle_y=True,
           return_fig=True)
add_lines(fig)
```



We expect these two losses to be relatively stable (i.e. ratio of 1), until the grok, when the restricted loss becomes larger than the test loss because although the key frequencies can explain the model outputs well, they can't explain it as perfectly as when the model has access to **all** the frequencies.

[55]:
```
fig = line([metric_cache["restricted_loss"]/torch.tensor(test_losses[::100])],
           x=np.arange(0, len(train_losses), 100),
           xaxis="Epoch",
           yaxis="Loss",
```

```
            log_y=True,
            title="Restricted Loss to Test Loss Ratio",
            toggle_x=True,
            toggle_y=True,
            return_fig=True)
add_lines(fig)
```

4.4 Excluded Loss

Now, we determine the excluded loss by ablating ONLY the key frequencies (meaning frequencies 1, 5, and 34 are removed from the model). If our hpyothesis is correct, then our model performance will get *worse* after grokking.

[56]:
```
def get_excluded_loss(model):
    logits, cache = model.run_with_cache(dataset)
    logits = logits[:, -1, :]
    neuron_acts = cache["post", 0, "mlp"][:, -1, :]
    approx_neuron_acts = torch.zeros_like(neuron_acts)

    a = torch.arange(p)[:, None]
    b = torch.arange(p)[None, :]
    for freq in key_freqs:
        # create & normalize a cosine wave matrix
        cos_apb_vec = torch.cos(freq * 2 * torch.pi / p * (a - b)).to(DEVICE)
        cos_apb_vec /= cos_apb_vec.norm()
        cos_apb_vec = einops.rearrange(cos_apb_vec, "a b -> (a b) 1")

        # project neuron activations onto cosine basis and add to approximations
        approx_neuron_acts += (neuron_acts * cos_apb_vec).sum(dim=0) *␣
 ↪cos_apb_vec

        # create & normalize a sine wave matrix
        sin_apb_vec = torch.sin(freq * 2 * torch.pi / p * (a - b)).to(DEVICE)
        sin_apb_vec /= sin_apb_vec.norm()
```

```
        sin_apb_vec = einops.rearrange(sin_apb_vec, "a b -> (a b) 1")

        # project neuron activations onto sine basis and add to approximations
        approx_neuron_acts += (neuron_acts * sin_apb_vec).sum(dim=0) *␣
↪sin_apb_vec

    excluded_neuron_acts = neuron_acts - approx_neuron_acts
    residual_stream_final = excluded_neuron_acts @ model.blocks[0].mlp.W_out +␣
↪cache["resid_mid", 0][:, -1, :]
    excluded_logits = residual_stream_final @ model.unembed.W_U

    return loss_fn(excluded_logits[train_indices], train_labels)

print("Excluded Loss:", get_excluded_loss(model).item())
```

```
Excluded Loss: 22.358710780330977
```

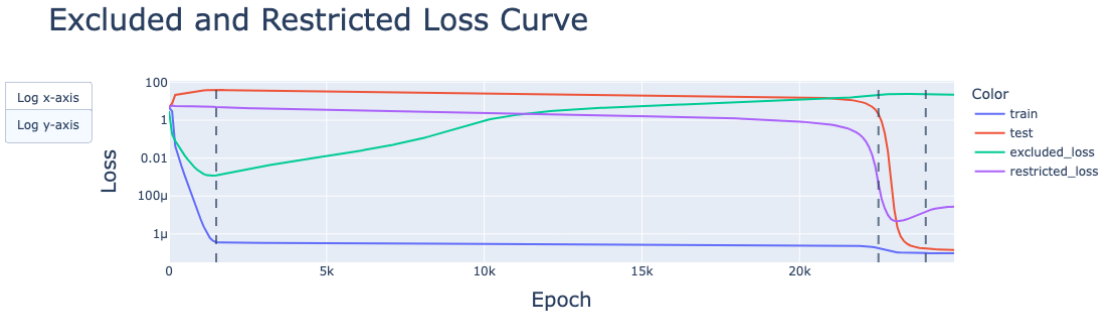
[57]:
```
get_metrics(model, metric_cache, get_excluded_loss, "excluded_loss", reset=True)
print(metric_cache["excluded_loss"].shape)
```

```
  0%|          | 0/250 [00:00<?, ?it/s]
```

```
torch.Size([250])
```

As the model begins generalizing, it increasingly relies on the key frequencies. So, in the excluded loss (where we have ablated them), we expect and, in fact, observe that the excluded loss gets worse after the grok, since the model's memorization has been "weight decayed away". So, the model is unable to rely on the key frequencies that support the general algorithm.

[58]:
```
fig = line([train_losses[::100],
            test_losses[::100],
            metric_cache["excluded_loss"],
            metric_cache["restricted_loss"]],
           x=np.arange(0, len(train_losses), 100),
           xaxis="Epoch",
           yaxis="Loss",
           log_y=True,
           title="Excluded and Restricted Loss Curve",
           line_labels=['train', 'test', "excluded_loss", "restricted_loss"],
           toggle_x=True,
           toggle_y=True,
           return_fig=True)

add_lines(fig)
```


Excluded and Restricted Loss Curve
Log x-axis
Log y-axis
Loss
100
1
0.01
100µ
1µ
0
5k
10k
15k
20k
Epoch
Color
train
test
excluded_loss
restricted_loss