Working Jupyter Notebook for HW2:

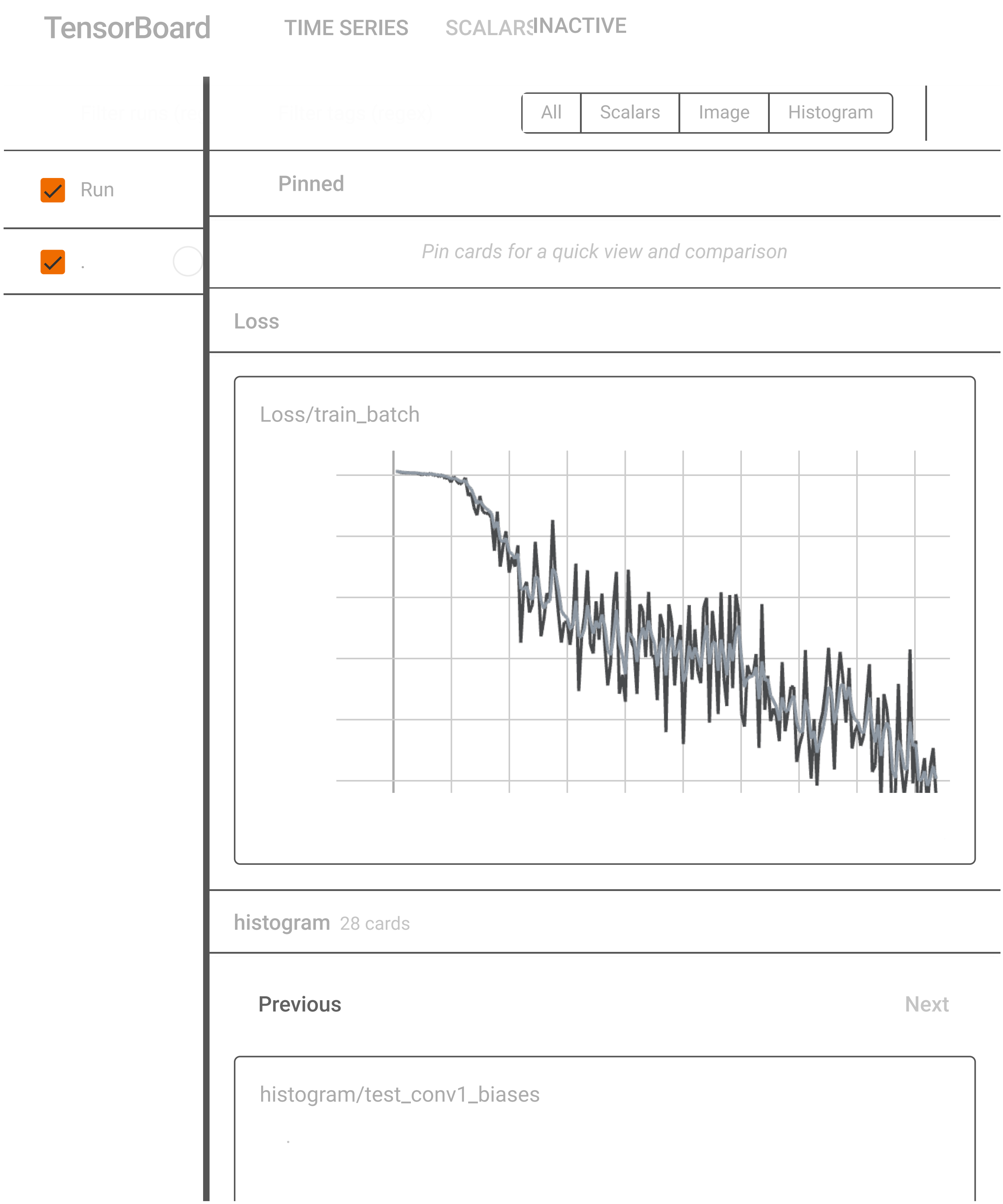
https://colab.research.google.com/drive/1CqsTUeYjTt-WCwewNb7eVhNnV5Jaou-I?usp=sharing

```
# %load_ext tensorboard
%reload_ext tensorboard
```

Problem 1

Tensorboard Setup

```
from datetime import datetime
from pathlib import Path

from torch.utils.tensorboard import SummaryWriter

writer = SummaryWriter("runs/LeNet5_experiment")

datetime_str = datetime.now().strftime('%b%d_%H-%M-%S')

base_folder = Path('runs/LeNet5_experiment')
base_folder.mkdir(parents=True, exist_ok=True)

logging_dir = base_folder / Path(datetime_str)
logging_dir.mkdir(exist_ok=True)

logging_dir = str(logging_dir.absolute())

writer = SummaryWriter(log_dir=logging_dir)

%tensorboard --logdir {logging_dir} --port 6000
```



Importing Libraries

```
import matplotlib.pyplot as plt
import numpy as np
import torch
import torch.nn as nn
import torch.optim as optim
import torchvision
import torchvision.transforms as transforms
```

Define Variables

```
# Initialize dictionary to save activations
# activations = {}

logging_interval = 10
batch_size = 128
epochs = 10

train_accuracies = []
test_accuracies = []
train_losses = []
```

Initializing Functions

```
def rgb_to_grayscale(img):
    gray = 0.299 * img[0] + 0.587 * img[1] + 0.114 * img[2]
    return gray.unsqueeze(0)  # Adds a channel at 0th dimension


activations = {}
def hook_fn(module, input, output):
    # layer_name = str(module)
    activations[str(module)] = output
```

Define LeNet5 Architecture

```
class LeNet5(nn.Module):
    def __init__(self):
        super(LeNet5, self).__init__()
        self.conv1 = nn.Conv2d(1, 32, 5)
        self.conv2 = nn.Conv2d(32, 64, 5)
        self.fc1 = nn.Linear(5 * 5 * 64, 1024)
        self.fc2 = nn.Linear(1024, 10)
        self.stats = {}

    def forward(self, x):
        x = nn.functional.relu(self.conv1(x))
        self.stats['conv1_weights'] = self.conv1.weight.data.cpu().numpy()
        self.stats['conv1_biases'] = self.conv1.bias.data.cpu().numpy()
        x = nn.functional.max_pool2d(x, 2)
        x = nn.functional.relu(self.conv2(x))
        self.stats['conv2_weights'] = self.conv2.weight.data.cpu().numpy()
        self.stats['conv2_biases'] = self.conv2.bias.data.cpu().numpy()
        x = nn.functional.max_pool2d(x, 2)
        x = x.view(-1, 5 * 5 * 64)
        x = nn.functional.relu(self.fc1(x))
        self.stats['fc1_weights'] = self.fc1.weight.data.cpu().numpy()
        self.stats['fc1_biases'] = self.fc1.bias.data.cpu().numpy()
        x = self.fc2(x)
        self.stats['fc2_weights'] = self.fc2.weight.data.cpu().numpy()
        self.stats['fc2_biases'] = self.fc2.bias.data.cpu().numpy()
        return nn.functional.softmax(x, dim=1)

model = LeNet5()
criterion = nn.CrossEntropyLoss()
optimizer = optim.SGD(model.parameters(), lr=0.01, momentum=0.9)
```

Data Loading

```
# Define the transformation for training and test datasets
transform = transforms.Compose([
    transforms.ToTensor(),
    transforms.Lambda(rgb_to_grayscale),
    transforms.Normalize((0.5,), (0.5,))
])
```

```
# Load the CIFAR10 dataset
trainset = torchvision.datasets.CIFAR10(root='./data', train=True, download=True, t
testset = torchvision.datasets.CIFAR10(root='./data', train=False, download=True, t
```

```
Files already downloaded and verified
Files already downloaded and verified
```

```
# DataLoader
train_loader = torch.utils.data.DataLoader(trainset, batch_size=batch_size, shuffle
test_loader = torch.utils.data.DataLoader(testset, batch_size=batch_size, shuffle=F
```

Training Loop

```
train_losses = []
test_accuracies = []
train_accuracies = []

for epoch in range(epochs):
    model.train()
    correct = 0
    total = 0
    running_loss = 0.0

    for i, data in enumerate(train_loader, 0):
        inputs, labels = data
        optimizer.zero_grad()

        # Forward pass
        outputs = model(inputs)
        loss = criterion(outputs, labels)

        # Backward pass and optimization
        loss.backward()
        optimizer.step()

        # Calculate accuracy
        _, predicted = torch.max(outputs.data, 1)
        total += labels.size(0)
        correct += (predicted == labels).sum().item()

        # Calculate running loss
        running_loss += loss.item()
        n_iter = epoch * len(train_loader) + i

        if (i + 1) % logging_interval == 0:
```

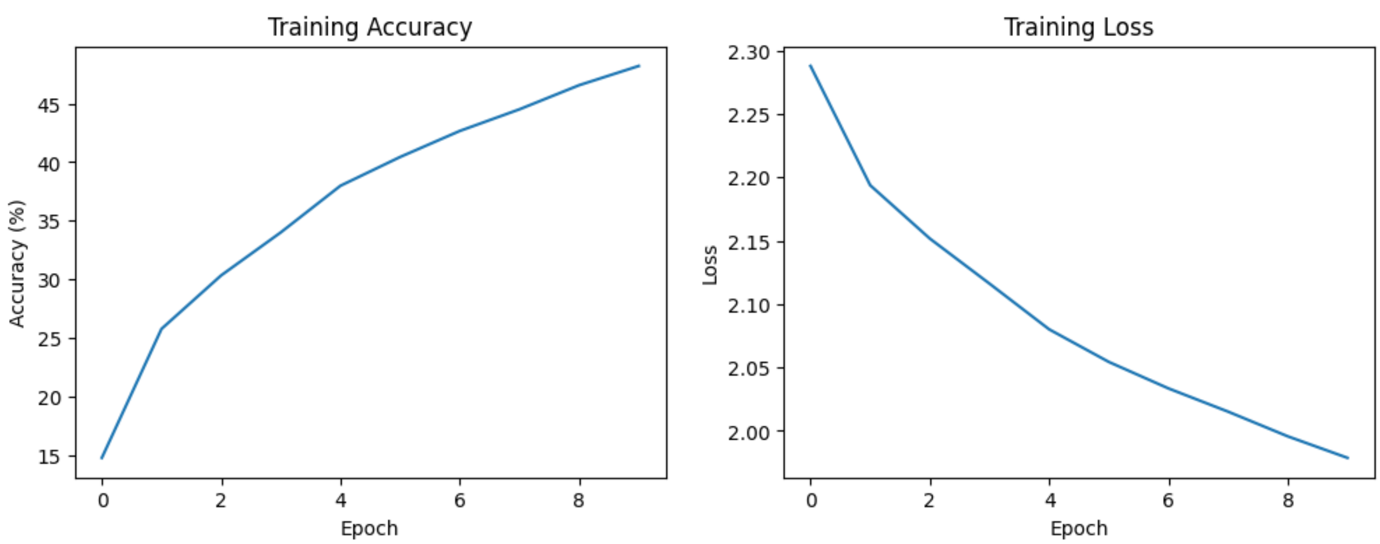
```
    if (i + 1) % logging_interval == 0:
        writer.add_scalar('Loss/train_batch', loss.item(), n_iter)
        writer.add_scalar('Accuracy/Train', 100 * correct / total, epoch)

        for key, value in model.stats.items():
            value_flat = value.flatten()
            writer.add_scalar(f'statistics/train_{key}_mean', np.mean(value_fla
            writer.add_scalar(f'statistics/train_{key}_std', np.std(value_flat)
            writer.add_scalar(f'statistics/train_{key}_min', np.min(value_flat)
            writer.add_scalar(f'statistics/train_{key}_max', np.max(value_flat)
            writer.add_histogram(f'histogram/train_{key}', value_flat, n_iter)

        for name, param in model.named_parameters():
            if 'weight' in name:
                writer.add_scalar(f'statistics/train_{name}_min', param.min().i
                writer.add_scalar(f'statistics/train_{name}_max', param.max().i
                writer.add_scalar(f'statistics/train_{name}_mean', param.mean()
                writer.add_scalar(f'statistics/train_{name}_std', param.std().i
                writer.add_histogram(f'histogram/test_{name}', param, n_iter)
            elif 'bias' in name:
                writer.add_scalar(f'statistics/train_{name}_min', param.min().i
                writer.add_scalar(f'statistics/train_{name}_max', param.max().i
                writer.add_scalar(f'statistics/train_{name}_mean', param.mean()
                writer.add_scalar(f'statistics/train_{name}_std', param.std().i
                writer.add_histogram(f'histogram/train_{name}', param, n_iter)

train_accuracies.append(100 * correct / total)
train_losses.append(running_loss / len(train_loader))

print(f"Epoch {epoch+1}, Loss: {running_loss / len(train_loader)}, Accuracy: {1

# Test Loop
model.eval()
correct = 0
total = 0

with torch.no_grad():
    for j, data in enumerate(test_loader):

        images, labels = data
        outputs = model(images)
        _, predicted = torch.max(outputs.data, 1)
        total += labels.size(0)
        correct += (predicted == labels).sum().item()
        m_iter = epoch * len(test_loader) + j
        if (j + 1) % logging_interval == 0:
              for key, value in model.stats.items():
                  value_flat = value.flatten()
                  writer.add_scalar(f'statistics/test_{key}_mean', np.mean(valu
```

```
                    writer.add_scalar(f'statistics/test_{key}_std', np.std(value_
                    writer.add_scalar(f'statistics/test_{key}_min', np.min(value_
                    writer.add_scalar(f'statistics/test_{key}_max', np.max(value_
                    writer.add_histogram(f'histogram/test_{key}', value_flat, m_i

                for name, param in model.named_parameters():
                    if 'weight' in name:
                        writer.add_scalar(f'statistics/test_{name}_min', param.mi
                        writer.add_scalar(f'statistics/test_{name}_max', param.ma
                        writer.add_scalar(f'statistics/test_{name}_mean', param.m
                        writer.add_scalar(f'statistics/test_{name}_std', param.st
                        writer.add_histogram(f'histogram/test_{name}', param, m_i
                    elif 'bias' in name:
                        writer.add_scalar(f'statistics/test_{name}_min', param.mi
                        writer.add_scalar(f'statistics/test_{name}_max', param.ma
                        writer.add_scalar(f'statistics/test_{name}_mean', param.m
                        writer.add_scalar(f'statistics/test_{name}_std', param.st
                        writer.add_histogram(f'histogram/test_{name}', param, m_i
    test_accuracies.append(100 * correct / total)
    print(f"Test Accuracy: {100 * correct / total}%")
    writer.add_scalar('Accuracy/test', 100 * correct / total, epoch)
```

```
Epoch 1, Loss: 2.2880112874843275, Accuracy: 14.786%
Test Accuracy: 21.01%
Epoch 2, Loss: 2.1938467855038852, Accuracy: 25.78%
Test Accuracy: 28.96%
Epoch 3, Loss: 2.15173975829883, Accuracy: 30.342%
Test Accuracy: 32.68%
Epoch 4, Loss: 2.116175899725131, Accuracy: 34.01%
Test Accuracy: 36.88%
Epoch 5, Loss: 2.0800950630851416, Accuracy: 37.986%
Test Accuracy: 39.23%
Epoch 6, Loss: 2.0543632992088336, Accuracy: 40.438%
Test Accuracy: 39.71%
Epoch 7, Loss: 2.033445654317851, Accuracy: 42.658%
Test Accuracy: 42.61%
Epoch 8, Loss: 2.0150980592688637, Accuracy: 44.502%
Test Accuracy: 44.79%
Epoch 9, Loss: 1.9956189235457984, Accuracy: 46.558%
Test Accuracy: 43.75%
Epoch 10, Loss: 1.9786441292604218, Accuracy: 48.198%
Test Accuracy: 46.88%
```

Plotting

```
plt.figure(figsize=(12, 4))
plt.subplot(1, 2, 1)
plt.title("Training Accuracy")
plt.plot(train_accuracies)
plt.xlabel("Epoch")
plt.ylabel("Accuracy (%)")

plt.subplot(1, 2, 2)
plt.title("Training Loss")
plt.plot(train_losses)
plt.xlabel("Epoch")
plt.ylabel("Loss")

plt.show()
```

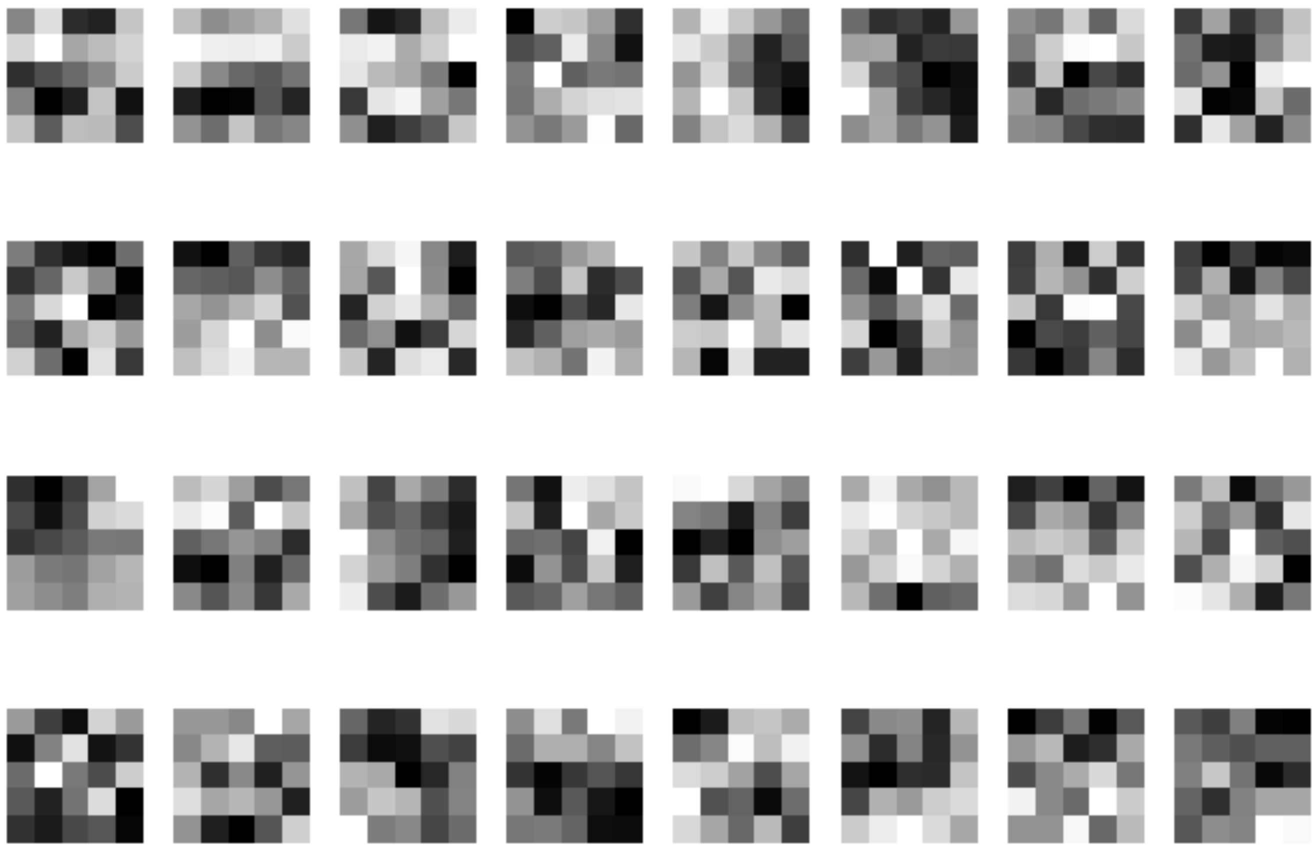


Extract Weights of the First Convolutional Layer (FCL)

```
weights = model.conv1.weight.data.numpy()
```

Visualize FCL's Weights (assuming 32 filter maps)

```
for i in range(32):
    plt.subplot(4, 8, i+1)
    plt.imshow(weights[i][0], cmap='gray')
    plt.axis('off')

plt.tight_layout()
plt.show()
```



Problem 2

The paper "Visualizing and Understanding Convolutional Networks" by Matthew Zeiler and Rob Fergus is a seminal work in deep learning. Published in 2013, it has significantly demystified the often "black-box" nature of Convolutional Neural Networks (CNNs), offering academics and industry practitioners critical insights into the architecture and function of CNNs.

Key Ideas:

1. **Deconvolutional Network**: A pivotal contribution of this paper is introducing a "deconvolutional network," a specialized architecture designed to map feature activations back to the input pixel space. This deconvolutional network effectively reverses the operations of the CNN's convolutional layers. The deconvolutional network is an invaluable tool for visualizing and understanding what each layer in the CNN has learned, thereby elucidating which features are most salient for the network's performance.
2. **Layer-wise Visualization**: The deconvolutional network enables the authors to showcase the hierarchical nature of the features learned by a CNN. Lower layers tend to capture primary attributes such as edges and textures. In contrast, higher layers capture increasingly abstract and complex combinations of the features learned in the preceding layers, which could range from object parts to entire objects.
3. **Feature Abstractions**: The paper demonstrates that feature abstractions become progressively complex as one moves deeper into the network. For example, while the first layer may specialize in edge detection, subsequent layers might focus on shapes by aggregating these edges, and even deeper layers may identify intricate structures or objects.
4. **Sensitivity Analysis**: This technique is introduced to pinpoint the input stimuli that activate specific feature maps, providing an alternative avenue for interpreting what the network is learning. Sensitivity analysis is crucial not just for understanding but also for diagnosing the model's behavior, highlighting how minute changes in the input can have a significant impact on the output.
5. **Improving Performance**: The visualization techniques are leveraged to identify and rectify architectural and training regimen issues, resulting in an optimized CNN model. This enhanced model sets new performance benchmarks across several standard metrics and challenges, including the ImageNet competition.
6. **Fine-Tuning and Transfer Learning**: The authors empirically demonstrate that the features learned in one domain or task can be effectively transferred to different but related fields or jobs. This domain transfer underscores the robust generalizability of CNNs.
7. **Validation**: The rigorous validation process credibly establishes the efficacy of the introduced visualization techniques. Performance comparisons before and after implementing architectural insights derived from visualizations lend additional credibility to these techniques.

The paper provides a comprehensive set of tools and methodologies for understanding, interpreting, and even diagnosing CNNs. These contributions have led to performance improvements and built greater trust in deploying CNNs across various applications. Moreover, the techniques introduced have set a precedent for future work in CNN interpretability and have been widely adopted across academic and industrial settings. Since its publication, the paper has been a cornerstone in deep learning, inspiring subsequent research on making neural networks more interpretable and transparent.

Applying one technique discussed in paper (Single Layer Visualization)

```
def plot_kernels(tensor, num_cols=8):
    # make 4D tensor if it is 3D
    if len(tensor.shape) == 3:
        tensor = tensor.unsqueeze(1)
    num_kernels = tensor.shape[0]
    num_rows = 1 + num_kernels // num_cols
    fig = plt.figure(figsize=(num_cols, num_rows))
    for i in range(num_kernels):
        ax1 = fig.add_subplot(num_rows, num_cols, i + 1)
        ax1.imshow(tensor[i][0, :, :], cmap="gray")
        ax1.axis('off')
        ax1.set_xticklabels([])
        ax1.set_yticklabels([])

    plt.subplots_adjust(wspace=0.1, hspace=0.1)
    plt.show()
```

```
def visualize_layer(model, layer_name, input_image):
    activation = {}
    def hook_fn(module, input, output):
        activation[layer_name] = output.detach()

    # Register hook
    layer = dict([*model.named_modules()])[layer_name]
    hook = layer.register_forward_hook(hook_fn)

    # Forward pass (assuming the input image is already a 4D tensor)
    model(input_image.unsqueeze(0))

    # Remove hook
    hook.remove()

    act = activation[layer_name].squeeze()
    # Make the activation 3D if it is 2D
    if len(act.shape) == 2:
        act = act.unsqueeze(0)
    plot_kernels(act)
```

```
# Assuming `images` and `labels` are one batch from your test_loader
with torch.no_grad():
    for i, data in enumerate(test_loader, 0):
        images, labels = data
        # Just take the first image from the first batch
        img = images[0]

        # Visualize the activations of the first convolution layer
        visualize_layer(model, 'conv1', img)

        # Exit after first batch
        break
```

```
# Register hooks for Conv layers
for name, layer in model.named_modules():
    if isinstance(layer, nn.Conv2d):
        layer.register_forward_hook(hook_fn)

# In your test loop, after you get the model output
with torch.no_grad():
    for i, data in enumerate(test_loader, 0):
        images, labels = data
        outputs = model(images)  # This will also populate 'activations' due to hoc

        # Log activations from convolutional layers
        grid = torchvision.utils.make_grid(images)
        writer.add_image('images', grid, 0)
        writer.add_graph(model, images)
        for name, activation in activations.items():
            writer.add_histogram(f"{name}.activation", activation.flatten(), i)
```

```
<ipython-input-7-52bdba1e040d>:12: TracerWarning: Converting a tensor to a Nur
  self.stats['conv1_weights'] = self.conv1.weight.data.cpu().numpy()
<ipython-input-7-52bdba1e040d>:13: TracerWarning: Converting a tensor to a Nur
  self.stats['conv1_biases'] = self.conv1.bias.data.cpu().numpy()
<ipython-input-7-52bdba1e040d>:16: TracerWarning: Converting a tensor to a Nur
  self.stats['conv2_weights'] = self.conv2.weight.data.cpu().numpy()
<ipython-input-7-52bdba1e040d>:17: TracerWarning: Converting a tensor to a Nur
  self.stats['conv2_biases'] = self.conv2.bias.data.cpu().numpy()
<ipython-input-7-52bdba1e040d>:21: TracerWarning: Converting a tensor to a Nur
  self.stats['fc1_weights'] = self.fc1.weight.data.cpu().numpy()
<ipython-input-7-52bdba1e040d>:22: TracerWarning: Converting a tensor to a Nur
  self.stats['fc1_biases'] = self.fc1.bias.data.cpu().numpy()
<ipython-input-7-52bdba1e040d>:24: TracerWarning: Converting a tensor to a Nur
  self.stats['fc2_weights'] = self.fc2.weight.data.cpu().numpy()
<ipython-input-7-52bdba1e040d>:25: TracerWarning: Converting a tensor to a Nur
  self.stats['fc2_biases'] = self.fc2.bias.data.cpu().numpy()
```

Problem 3

Importing Libraries

```
from datetime import datetime
from pathlib import Path

import numpy as np
import torch
import torch.nn as nn
import torch.nn.functional as F
import torch.optim as optim

from torchvision import datasets, transforms
from torch.utils.data import DataLoader
from torch.utils.tensorboard import SummaryWriter
```

Defining Models

```
class RNNBaseModel(nn.Module):
    def __init__(self, rnn_type, input_dim, hidden_dim, output_dim):
        super(RNNBaseModel, self).__init__()
        self.rnn = rnn_type(input_dim, hidden_dim, batch_first=True)
        self.fc = nn.Linear(hidden_dim, output_dim)
        self.stats = {}

    def forward(self, x):
        output, hn = self.rnn(x)
        self.stats['rnn_weights'] = self.rnn.weight_hh_l0.data.cpu().numpy()
        self.stats['rnn_biases'] = self.rnn.bias_hh_l0.data.cpu().numpy()

        output = self.fc(output[-1, :, :])
        self.stats['fc_weights'] = self.fc.weight.data.cpu().numpy()
        self.stats['fc_biases'] = self.fc.bias.data.cpu().numpy()

        output = F.log_softmax(output, dim=1)
        return output
```

```
class AdaptiveCNNModel(nn.Module):
    def __init__(self, num_filters1, num_filters2, fc_size):
        super(AdaptiveCNNModel, self).__init__()
        self.conv1 = nn.Conv2d(1, num_filters1, 5)
        self.conv2 = nn.Conv2d(num_filters1, num_filters2, 5)
        self.fc1_size = num_filters2 * 4 * 4  # This needs to be calculated based c
        self.fc1 = nn.Linear(self.fc1_size, fc_size)
        self.fc2 = nn.Linear(fc_size, 10)
        self.stats = {}

    def forward(self, x):
        # First Conv Layer
        x = self.conv1(x)
        self.stats['conv1_weights'] = self.conv1.weight.data.cpu().numpy()
        self.stats['conv1_biases'] = self.conv1.bias.data.cpu().numpy()
        x = F.relu(x)
        x = F.max_pool2d(x, 2)

        # Second Conv Layer
        x = self.conv2(x)
        self.stats['conv2_weights'] = self.conv2.weight.data.cpu().numpy()
        self.stats['conv2_biases'] = self.conv2.bias.data.cpu().numpy()
        x = F.relu(x)
        x = F.max_pool2d(x, 2)

        # Fully Connected Layers
        x = x.view(x.size(0), -1)  # Flattening
        x = self.fc1(x)
        self.stats['fc1_weights'] = self.fc1.weight.data.cpu().numpy()
        self.stats['fc1_biases'] = self.fc1.bias.data.cpu().numpy()
        x = F.relu(x)

        x = self.fc2(x)
        self.stats['fc2_weights'] = self.fc2.weight.data.cpu().numpy()
        self.stats['fc2_biases'] = self.fc2.bias.data.cpu().numpy()

        x = F.log_softmax(x, dim=1)

        return x
```

Declaring Parameters

General Hyperparams

```
batch_size = 28
logging_interval = 10
learning_rates = [0.001, 0.01]
optimizers = [optim.Adam, optim.SGD]
```

RNN Hyperparams

```
hidden_dims = [64, 128]
```

CNN Hyperparams

```
fc_size_list = [128, 256]
num_filters1_list = [32, 64]
num_filters2_list = [64, 128]
```

Data Loading

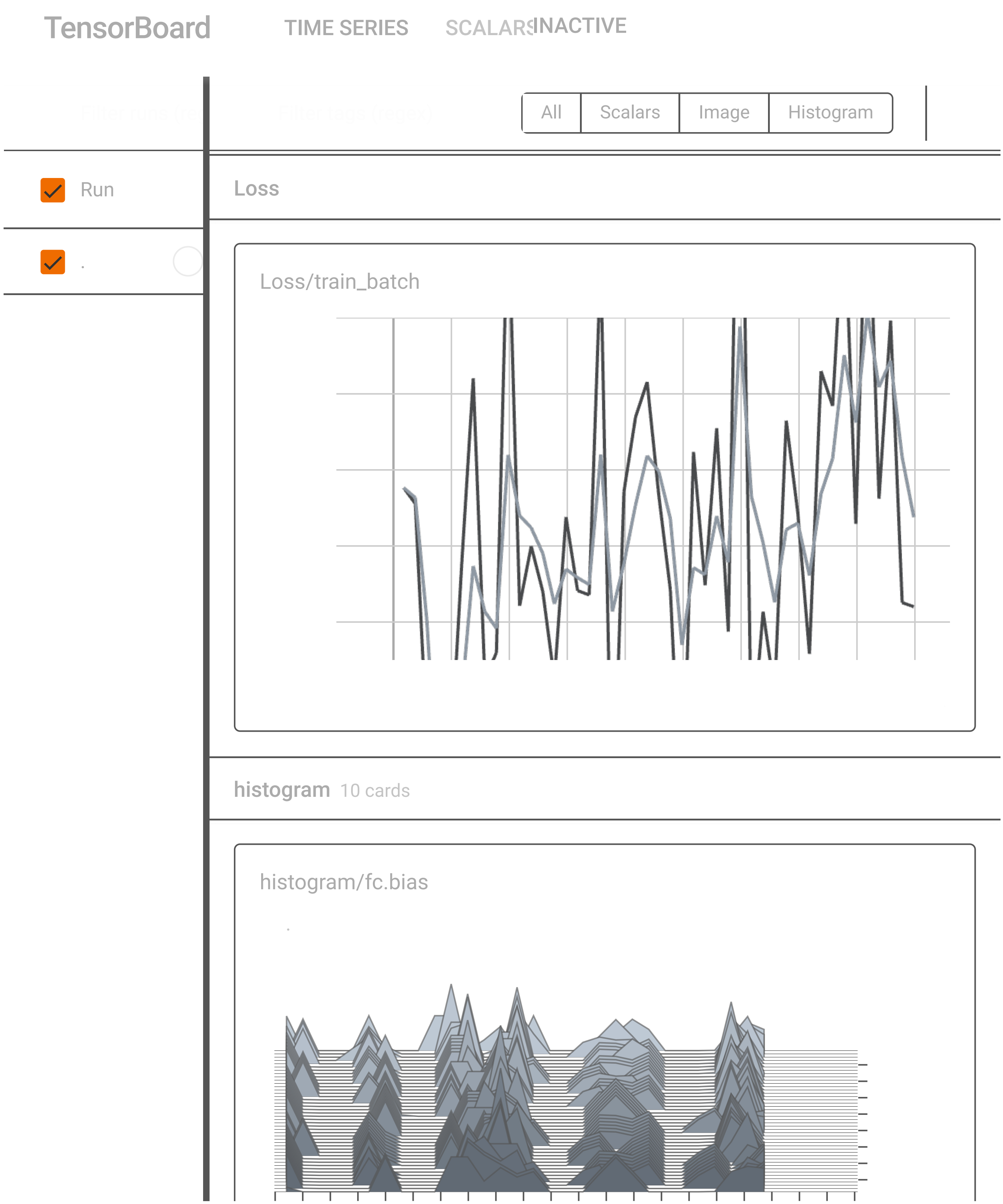
```
transform = transforms.Compose([transforms.ToTensor(), transforms.Normalize((0.5,),


trainset = datasets.MNIST(root='./data', train=True, download=True, transform=trans

testset = datasets.MNIST(root='./data', train=False, download=True, transform=trans


train_loader = DataLoader(trainset, batch_size=batch_size, shuffle=True, drop_last=

test_loader = DataLoader(testset, batch_size=batch_size, shuffle=False, drop_last=T
```

Hyperparameter Tuning & Visuals

RNN

```
datetime_str = datetime.now().strftime('%b%d_%H_%M_%S')
```

```
datetime_str = datetime.now().strftime('%b%d_%H-%M-%S')

base_folder = Path('runs/rnn_experiment')
base_folder.mkdir(parents=True, exist_ok=True)

logging_dir = base_folder / Path(datetime_str)
logging_dir.mkdir(exist_ok=True)

logging_dir = str(logging_dir.absolute())

writer = SummaryWriter(log_dir=logging_dir)

%tensorboard --logdir {logging_dir} --port 7001
```




```
train_losses = []
test_accuracies = []
train_accuracies = []
```

```
for i, hidden_dim in enumerate(hidden_dims):
  for j, lr in enumerate(learning_rates):
      for k, opt in enumerate(optimizers):
          model = RNNBaseModel(nn.RNN, batch_size, hidden_dim, 10)
          criterion = nn.CrossEntropyLoss()
          optimizer = opt(model.parameters(), lr=lr)
          print(f'Hyperparameters: [#{i+1} hidden_dim={hidden_dim}, #{j+1} lr={lr},
          for epoch in range(2):  # Limiting to 2 epochs for demonstration
              running_loss = 0.0
              correct = 0
              model.train()
              for batch_idx, (images, labels) in enumerate(train_loader):
                  n_iter = epoch * len(train_loader) + batch_idx
                  images = images.view(-1, batch_size, batch_size)
                  outputs = model(images)
                  loss = criterion(outputs, labels)

                  optimizer.zero_grad()
                  loss.backward()
                  optimizer.step()

                  total += labels.size(0)
                  running_loss += loss.item()
                  _, predicted = torch.max(outputs.data, 1)
                  correct += (predicted == labels).sum().item()

                  if (batch_idx + 1) % logging_interval == 0:
                      writer.add_scalar('Loss/train_batch', loss.item(), n_iter)
                      writer.add_scalar('Accuracy/train', 100. * correct /total, ep


                      for key, value in model.stats.items():
                          value_flat = value.flatten()
                          writer.add_scalar(f'statistics/{key}_mean', np.mean(value
                          writer.add_scalar(f'statistics/{key}_std', np.std(value_f
                          writer.add_scalar(f'statistics/{key}_min', np.min(value_f
                          writer.add_scalar(f'statistics/{key}_max', np.max(value_f
                          writer.add_histogram(f'histogram/{key}', value_flat, n_it

                      for name, param in model.named_parameters():
                          if 'weight' in name:
                              writer.add_scalar(f'statistics/{name}_min', param.mir
                              writer.add_scalar(f'statistics/{name}_max', param.max
                              writer.add_scalar(f'statistics/{name}_mean', param.me
                              writer.add_scalar(f'statistics/{name}_std', param.stc
                              writer.add_histogram(f'histogram/{name}', param, n_it
                          elif 'bias' in name:
```

```
                writer.add_scalar(f'statistics/{name}_min', param.mir
                writer.add_scalar(f'statistics/{name}_max', param.ma>
                writer.add_scalar(f'statistics/{name}_mean', param.me
                writer.add_scalar(f'statistics/{name}_std', param.stc
                writer.add_histogram(f'histogram/{name}', param, n_it

        print(f'Epoch [{epoch+1}/2], Step [{batch_idx+1}/{len(train_l
train_accuracies.append(100. * correct /total)
train_losses.append(running_loss / total)

# Test loop
model.eval()
correct = 0
total = 0
with torch.no_grad():
    for j, data in enumerate(test_loader):
        images, labels = data
        outputs = model(images)
        _, predicted = torch.max(outputs.data, 1)
        total += labels.size(0)
        correct += (predicted == labels).sum().item()
        m_iter = epoch * len(test_loader) + j
        if (j + 1) % logging_interval == 0:
              for key, value in model.stats.items():
                  value_flat = value.flatten()
                  writer.add_scalar(f'statistics/test_{key}_mean', np
                  writer.add_scalar(f'statistics/test_{key}_std', np.
                  writer.add_scalar(f'statistics/test_{key}_min', np.
                  writer.add_scalar(f'statistics/test_{key}_max', np.
                  writer.add_histogram(f'histogram/test_{key}', value

              for name, param in model.named_parameters():
                  if 'weight' in name:
                      writer.add_scalar(f'statistics/test_{name}_min'
                      writer.add_scalar(f'statistics/test_{name}_max'
                      writer.add_scalar(f'statistics/test_{name}_mear
                      writer.add_scalar(f'statistics/test_{name}_std'
                      writer.add_histogram(f'histogram/test_{name}',
                  elif 'bias' in name:
                      writer.add_scalar(f'statistics/test_{name}_min'
                      writer.add_scalar(f'statistics/test_{name}_max'
                      writer.add_scalar(f'statistics/test_{name}_mear
                      writer.add_scalar(f'statistics/test_{name}_std'
                      writer.add_histogram(f'histogram/test_{name}',
    writer.add_scalar('Accuracy/test', 100. * correct / total, epoch)
    print(f"Test Accuracy: {100 * correct / total}%")
    test_accuracies.append(100. * correct / total)
```

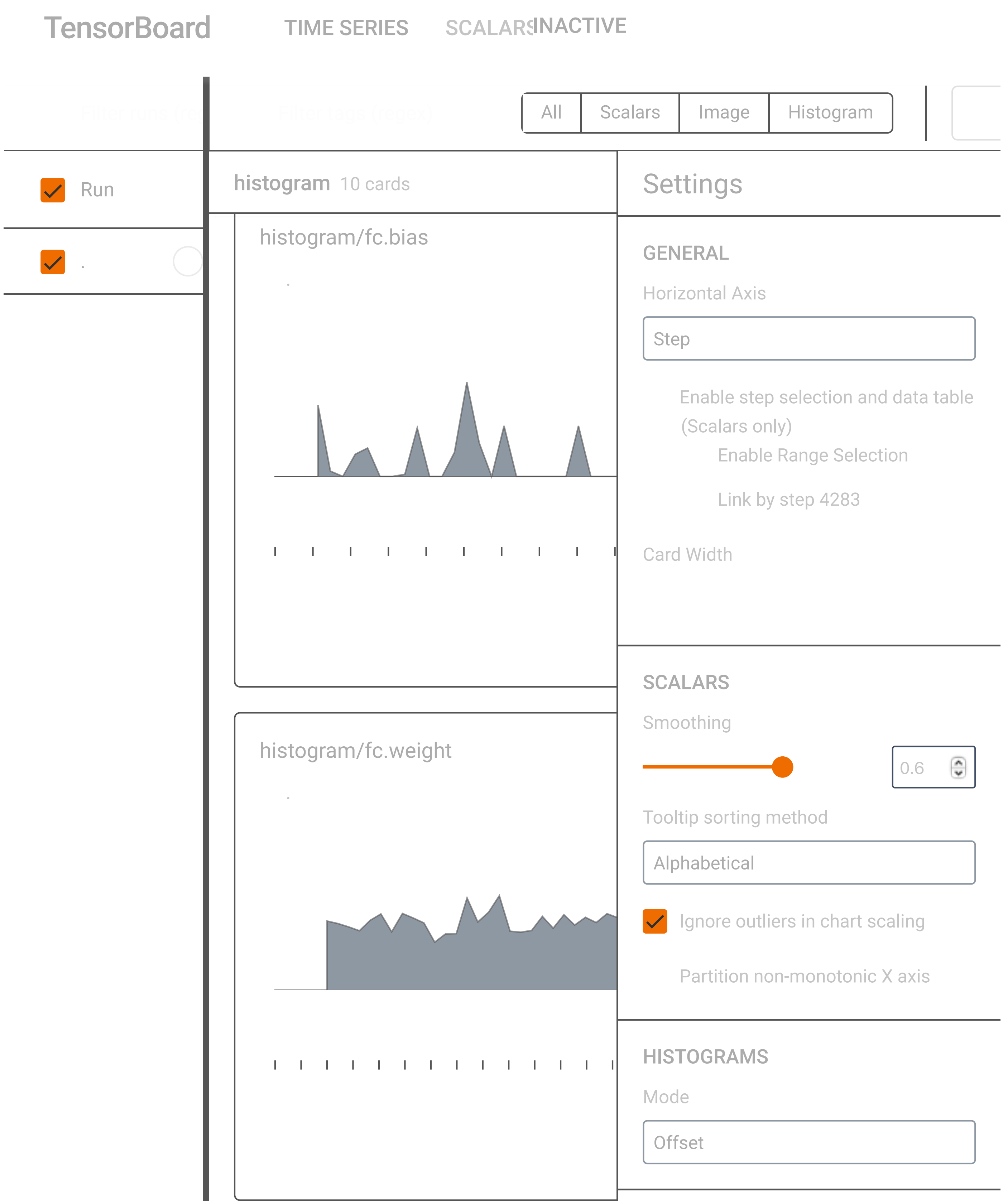
```
writer.close()
```

```
Epoch [2/2], Step [1360/2142], Loss: 2.2967
Epoch [2/2], Step [1370/2142], Loss: 2.3038
Epoch [2/2], Step [1380/2142], Loss: 2.3018
Epoch [2/2], Step [1390/2142], Loss: 2.3152
Epoch [2/2], Step [1400/2142], Loss: 2.2866
Epoch [2/2], Step [1410/2142], Loss: 2.3069
Epoch [2/2], Step [1420/2142], Loss: 2.3122
Epoch [2/2], Step [1430/2142], Loss: 2.2868
Epoch [2/2], Step [1440/2142], Loss: 2.3032
Epoch [2/2], Step [1450/2142], Loss: 2.3119
Epoch [2/2], Step [1460/2142], Loss: 2.3070
Epoch [2/2], Step [1470/2142], Loss: 2.3005
Epoch [2/2], Step [1480/2142], Loss: 2.2944
Epoch [2/2], Step [1490/2142], Loss: 2.2893
Epoch [2/2], Step [1500/2142], Loss: 2.2937
Epoch [2/2], Step [1510/2142], Loss: 2.3032
Epoch [2/2], Step [1520/2142], Loss: 2.3190
Epoch [2/2], Step [1530/2142], Loss: 2.3030
Epoch [2/2], Step [1540/2142], Loss: 2.3203
Epoch [2/2], Step [1550/2142], Loss: 2.2840
Epoch [2/2], Step [1560/2142], Loss: 2.2923
Epoch [2/2], Step [1570/2142], Loss: 2.3305
Epoch [2/2], Step [1580/2142], Loss: 2.3138
Epoch [2/2], Step [1590/2142], Loss: 2.3061
Epoch [2/2], Step [1600/2142], Loss: 2.2852
Epoch [2/2], Step [1610/2142], Loss: 2.2641
Epoch [2/2], Step [1620/2142], Loss: 2.3142
Epoch [2/2], Step [1630/2142], Loss: 2.2976
Epoch [2/2], Step [1640/2142], Loss: 2.3201
Epoch [2/2], Step [1650/2142], Loss: 2.2943
Epoch [2/2], Step [1660/2142], Loss: 2.3003
Epoch [2/2], Step [1670/2142], Loss: 2.3002
Epoch [2/2], Step [1680/2142], Loss: 2.3235
Epoch [2/2], Step [1690/2142], Loss: 2.3081
Epoch [2/2], Step [1700/2142], Loss: 2.3208
Epoch [2/2], Step [1710/2142], Loss: 2.3099
Epoch [2/2], Step [1720/2142], Loss: 2.2969
Epoch [2/2], Step [1730/2142], Loss: 2.3031
Epoch [2/2], Step [1740/2142], Loss: 2.2741
Epoch [2/2], Step [1750/2142], Loss: 2.3224
Epoch [2/2], Step [1760/2142], Loss: 2.2882
Epoch [2/2], Step [1770/2142], Loss: 2.2917
Epoch [2/2], Step [1780/2142], Loss: 2.2969
Epoch [2/2], Step [1790/2142], Loss: 2.2984
Epoch [2/2], Step [1800/2142], Loss: 2.3042
Epoch [2/2], Step [1810/2142], Loss: 2.3175
Epoch [2/2], Step [1820/2142], Loss: 2.2907
Epoch [2/2], Step [1830/2142], Loss: 2.2841
Epoch [2/2], Step [1840/2142], Loss: 2.3157
Epoch [2/2], Step [1850/2142], Loss: 2.3179
Epoch [2/2], Step [1860/2142], Loss: 2.2995
```

```
Epoch [2/2], Step [1870/2142], Loss: 2.3148
Epoch [2/2], Step [1880/2142], Loss: 2.2994
Epoch [2/2], Step [1890/2142], Loss: 2.2845
Epoch [2/2], Step [1900/2142], Loss: 2.2932
Epoch [2/2], Step [1910/2142], Loss: 2.3004
Epoch [2/2], Step [1920/2142], Loss: 2.3109
Epoch [2/2], Step [1930/2142], Loss: 2.3233
Epoch [2/2], Step [1940/2142], Loss: 2.3029
```

LSTM

```
datetime_str = datetime.now().strftime('%b%d_%H-%M-%S')

base_folder = Path('runs/lstm_experiment')
base_folder.mkdir(parents=True, exist_ok=True)

logging_dir = base_folder / Path(datetime_str)
logging_dir.mkdir(exist_ok=True)

logging_dir = str(logging_dir.absolute())

writer = SummaryWriter(log_dir=logging_dir)

%tensorboard --logdir {logging_dir} --port 7003
```




```
train_losses = []
test_accuracies = []
train_accuracies = []
```

```
for i, hidden_dim in enumerate(hidden_dims):
    for j, lr in enumerate(learning_rates):
        for k, opt in enumerate(optimizers):
            model = RNNBaseModel(nn.LSTM, batch_size, hidden_dim, 10)
            criterion = nn.CrossEntropyLoss()
            optimizer = opt(model.parameters(), lr=lr)
            print(f'Hyperparameters: [#{i+1} hidden_dim={hidden_dim}, #{j+1} lr={lr
            for epoch in range(2):  # Limiting to 2 epochs for demonstration
              running_loss = 0.0
              correct = 0
              model.train()
              for batch_idx, (images, labels) in enumerate(train_loader):
                  n_iter = epoch * len(train_loader) + batch_idx
                  images = images.view(-1, batch_size, batch_size)
                  outputs = model(images)
                  loss = criterion(outputs, labels)

                  optimizer.zero_grad()
                  loss.backward()
                  optimizer.step()

                  total += labels.size(0)
                  running_loss += loss.item()
                  _, predicted = torch.max(outputs.data, 1)
                  correct += (predicted == labels).sum().item()

                  if (batch_idx + 1) % logging_interval == 0:
                      writer.add_scalar('Loss/train_batch', loss.item(), n_iter)
                      writer.add_scalar('Accuracy/train', 100. * correct / total, e

                      for key, value in model.stats.items():
                          value_flat = value.flatten()
                          writer.add_scalar(f'statistics/train_{key}_mean', np.mean
                          writer.add_scalar(f'statistics/train_{key}_std', np.std(v
                          writer.add_scalar(f'statistics/train_{key}_min', np.min(v
                          writer.add_scalar(f'statistics/train_{key}_max', np.max(v
                          writer.add_histogram(f'histogram/train_{key}', value_flat

                      for name, param in model.named_parameters():
                          if 'weight' in name:
                              writer.add_scalar(f'statistics/train_{name}_min', par
                              writer.add_scalar(f'statistics/train_{name}_max', par
                              writer.add_scalar(f'statistics/train_{name}_mean', pa
                              writer.add_scalar(f'statistics/train_{name}_std', par
                              writer.add_histogram(f'histogram/train_{name}', param
                          elif 'bias' in name:
                              writer.add_scalar(f'statistics/train_{name}_min', par
```

```
                writer.add_scalar(f'statistics/train_{name}_max', par
                writer.add_scalar(f'statistics/train_{name}_mean', pa
                writer.add_scalar(f'statistics/train_{name}_std', par
                writer.add_histogram(f'histogram/train_{name}', param

        print(f'Epoch [{epoch+1}/2], Step [{batch_idx+1}/{len(train_l
train_accuracies.append(100. * correct / total)
train_losses.append(running_loss / total)

# Test Loop
model.eval()
correct = 0
total = 0

with torch.no_grad():
    for j, data in enumerate(test_loader):

        images, labels = data
        outputs = model(images)
        _, predicted = torch.max(outputs.data, 1)
        total += labels.size(0)
        correct += (predicted == labels).sum().item()
        m_iter = epoch * len(test_loader) + j
        if (j + 1) % logging_interval == 0:
              for key, value in model.stats.items():
                  value_flat = value.flatten()
                  writer.add_scalar(f'statistics/test_{key}_mean', np
                  writer.add_scalar(f'statistics/test_{key}_std', np.
                  writer.add_scalar(f'statistics/test_{key}_min', np.
                  writer.add_scalar(f'statistics/test_{key}_max', np.
                  writer.add_histogram(f'histogram/test_{key}', value

              for name, param in model.named_parameters():
                  if 'weight' in name:
                      writer.add_scalar(f'statistics/test_{name}_min'
                      writer.add_scalar(f'statistics/test_{name}_max'
                      writer.add_scalar(f'statistics/test_{name}_mean
                      writer.add_scalar(f'statistics/test_{name}_std'
                      writer.add_histogram(f'histogram/test_{name}',
                  elif 'bias' in name:
                      writer.add_scalar(f'statistics/test_{name}_min'
                      writer.add_scalar(f'statistics/test_{name}_max'
                      writer.add_scalar(f'statistics/test_{name}_mean
                      writer.add_scalar(f'statistics/test_{name}_std'
                      writer.add_histogram(f'histogram/test_{name}',
    writer.add_scalar('Accuracy/test', 100 * correct / total, epoch)
    print(f"Test Accuracy: {100 * correct / total}%")
    test_accuracies.append(100 * correct / total)
```

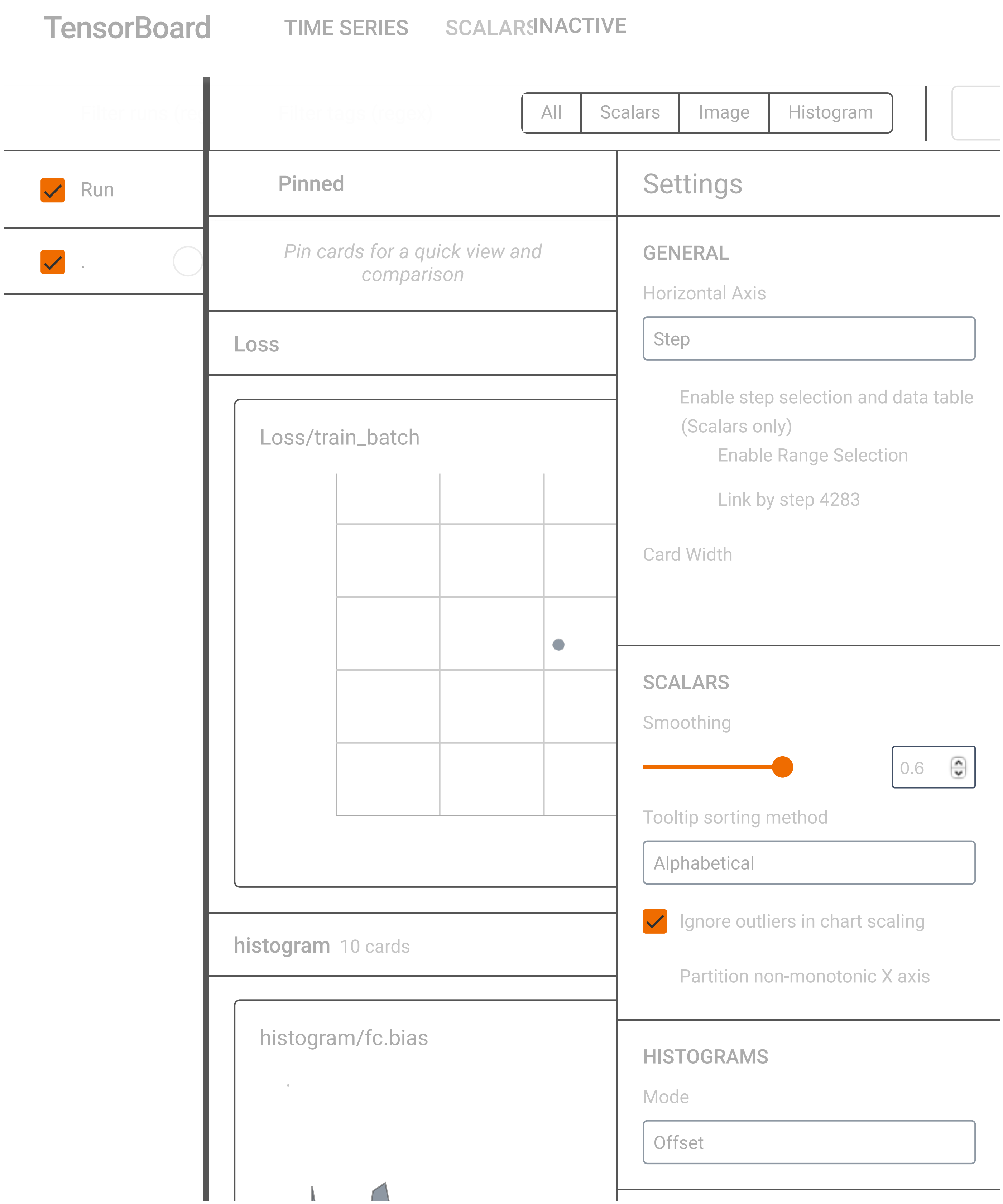
```
writer.close()
```

```
Epoch [2/2], Step [1000/2142], Loss: 2.3005
Epoch [2/2], Step [1010/2142], Loss: 2.3205
Epoch [2/2], Step [1020/2142], Loss: 2.2924
Epoch [2/2], Step [1030/2142], Loss: 2.2869
Epoch [2/2], Step [1040/2142], Loss: 2.2891
Epoch [2/2], Step [1050/2142], Loss: 2.3053
Epoch [2/2], Step [1060/2142], Loss: 2.2917
Epoch [2/2], Step [1070/2142], Loss: 2.2815
Epoch [2/2], Step [1080/2142], Loss: 2.2734
Epoch [2/2], Step [1090/2142], Loss: 2.2834
Epoch [2/2], Step [1100/2142], Loss: 2.3043
Epoch [2/2], Step [1110/2142], Loss: 2.3071
Epoch [2/2], Step [1120/2142], Loss: 2.3112
Epoch [2/2], Step [1130/2142], Loss: 2.3069
Epoch [2/2], Step [1140/2142], Loss: 2.3059
Epoch [2/2], Step [1150/2142], Loss: 2.2956
Epoch [2/2], Step [1160/2142], Loss: 2.2901
Epoch [2/2], Step [1170/2142], Loss: 2.3133
Epoch [2/2], Step [1180/2142], Loss: 2.3049
Epoch [2/2], Step [1190/2142], Loss: 2.3001
Epoch [2/2], Step [1200/2142], Loss: 2.3021
Epoch [2/2], Step [1210/2142], Loss: 2.2963
Epoch [2/2], Step [1220/2142], Loss: 2.2869
Epoch [2/2], Step [1230/2142], Loss: 2.3057
Epoch [2/2], Step [1240/2142], Loss: 2.3058
Epoch [2/2], Step [1250/2142], Loss: 2.3005
Epoch [2/2], Step [1260/2142], Loss: 2.2998
Epoch [2/2], Step [1270/2142], Loss: 2.3089
Epoch [2/2], Step [1280/2142], Loss: 2.2985
Epoch [2/2], Step [1290/2142], Loss: 2.2975
Epoch [2/2], Step [1300/2142], Loss: 2.2971
Epoch [2/2], Step [1310/2142], Loss: 2.3132
Epoch [2/2], Step [1320/2142], Loss: 2.2831
Epoch [2/2], Step [1330/2142], Loss: 2.3156
Epoch [2/2], Step [1340/2142], Loss: 2.3099
Epoch [2/2], Step [1350/2142], Loss: 2.2891
Epoch [2/2], Step [1360/2142], Loss: 2.3203
Epoch [2/2], Step [1370/2142], Loss: 2.2906
Epoch [2/2], Step [1380/2142], Loss: 2.3050
Epoch [2/2], Step [1390/2142], Loss: 2.3028
Epoch [2/2], Step [1400/2142], Loss: 2.2939
Epoch [2/2], Step [1410/2142], Loss: 2.3006
Epoch [2/2], Step [1420/2142], Loss: 2.2987
Epoch [2/2], Step [1430/2142], Loss: 2.3054
Epoch [2/2], Step [1440/2142], Loss: 2.3161
Epoch [2/2], Step [1450/2142], Loss: 2.3000
Epoch [2/2], Step [1460/2142], Loss: 2.3095
Epoch [2/2], Step [1470/2142], Loss: 2.2944
Epoch [2/2], Step [1480/2142], Loss: 2.3143
Epoch [2/2], Step [1490/2142], Loss: 2.2916
```

```
Epoch [2/2], Step [1500/2142], Loss: 2.2925
Epoch [2/2], Step [1510/2142], Loss: 2.3234
Epoch [2/2], Step [1520/2142], Loss: 2.2966
Epoch [2/2], Step [1530/2142], Loss: 2.2837
Epoch [2/2], Step [1540/2142], Loss: 2.2856
Epoch [2/2], Step [1550/2142], Loss: 2.2823
Epoch [2/2], Step [1560/2142], Loss: 2.3100
Epoch [2/2], Step [1570/2142], Loss: 2.2963
Epoch [2/2], Step [1580/2142], Loss: 2.2991
```

GRU

```
datetime_str = datetime.now().strftime('%b%d_%H-%M-%S')

base_folder = Path('runs/gru_experiment')
base_folder.mkdir(parents=True, exist_ok=True)

logging_dir = base_folder / Path(datetime_str)
logging_dir.mkdir(exist_ok=True)

logging_dir = str(logging_dir.absolute())

writer = SummaryWriter(log_dir=logging_dir)

%tensorboard --logdir {logging_dir} --port 7005
```




```
train_losses = []
test_accuracies = []
train_accuracies = []
```

```
for i, hidden_dim in enumerate(hidden_dims):
    for j, lr in enumerate(learning_rates):
        for k, opt in enumerate(optimizers):
            model = RNNBaseModel(nn.GRU, batch_size, hidden_dim, 10)
            criterion = nn.CrossEntropyLoss()
            optimizer = opt(model.parameters(), lr=lr)
            print(f'Hyperparameters: [#{i+1} hidden_dim={hidden_dim}, #{j+1} lr={lr
            for epoch in range(2):  # Limiting to 2 epochs for demonstration
                running_loss = 0.0
                correct = 0
                model.train()
                for batch_idx, (images, labels) in enumerate(train_loader):
                    n_iter = epoch * len(train_loader) + batch_idx
                    images = images.view(-1, batch_size, batch_size)
                    outputs = model(images)
                    loss = criterion(outputs, labels)

                    optimizer.zero_grad()
                    loss.backward()
                    optimizer.step()

                    total += labels.size(0)
                    running_loss += loss.item()
                    _, predicted = torch.max(outputs.data, 1)
                    correct += (predicted == labels).sum().item()

                    if (batch_idx + 1) % logging_interval == 0:
                        writer.add_scalar('Loss/train_batch', loss.item(), n_iter)
                        writer.add_scalar('Accuracy/train', 100 * correct / total,

                        for key, value in model.stats.items():
                            value_flat = value.flatten()
                            writer.add_scalar(f'statistics/train_{key}_mean', np.me
                            writer.add_scalar(f'statistics/train_{key}_std', np.std
                            writer.add_scalar(f'statistics/train_{key}_min', np.min
                            writer.add_scalar(f'statistics/train_{key}_max', np.max
                            writer.add_histogram(f'histogram/train_{key}', value_fl

                        for name, param in model.named_parameters():
                            if 'weight' in name:
                                writer.add_scalar(f'statistics/train_{name}_min', p
                                writer.add_scalar(f'statistics/train_{name}_max', p
                                writer.add_scalar(f'statistics/train_{name}_mean',
                                writer.add_scalar(f'statistics/train_{name}_std', p
                                writer.add_histogram(f'histogram/train_{name}', par
                            elif 'bias' in name:
                                writer.add_scalar(f'statistics/train_{name}_min', p
```

```
                writer.add_scalar(f'statistics/train_{name}_max', p
                writer.add_scalar(f'statistics/train_{name}_mean',
                writer.add_scalar(f'statistics/train_{name}_std', p
                writer.add_histogram(f'histogram/train_{name}', par

        print(f'Epoch [{epoch+1}/2], Step [{batch_idx+1}/{len(trair
train_accuracies.append(100. * correct / total)
train_losses.append(running_loss / total)

# Test Loop
model.eval()
correct = 0
total = 0

with torch.no_grad():
    for j, data in enumerate(test_loader):

        images, labels = data
        outputs = model(images)
        _, predicted = torch.max(outputs.data, 1)
        total += labels.size(0)
        correct += (predicted == labels).sum().item()
        m_iter = epoch * len(test_loader) + j
        if (j + 1) % logging_interval == 0:
              for key, value in model.stats.items():
                  value_flat = value.flatten()
                  writer.add_scalar(f'statistics/test_{key}_mean',
                  writer.add_scalar(f'statistics/test_{key}_std', r
                  writer.add_scalar(f'statistics/test_{key}_min', r
                  writer.add_scalar(f'statistics/test_{key}_max', r
                  writer.add_histogram(f'histogram/test_{key}', val

              for name, param in model.named_parameters():
                  if 'weight' in name:
                      writer.add_scalar(f'statistics/test_{name}_mi
                      writer.add_scalar(f'statistics/test_{name}_ma
                      writer.add_scalar(f'statistics/test_{name}_me
                      writer.add_scalar(f'statistics/test_{name}_st
                      writer.add_histogram(f'histogram/test_{name}'
                  elif 'bias' in name:
                      writer.add_scalar(f'statistics/test_{name}_mi
                      writer.add_scalar(f'statistics/test_{name}_ma
                      writer.add_scalar(f'statistics/test_{name}_me
                      writer.add_scalar(f'statistics/test_{name}_st
                      writer.add_histogram(f'histogram/test_{name}'
    writer.add_scalar('Accuracy/test', 100 * correct / total, epoch
    print(f"Test Accuracy: {100 * correct / total}%")
    test_accuracies.append(100 * correct / total)
```

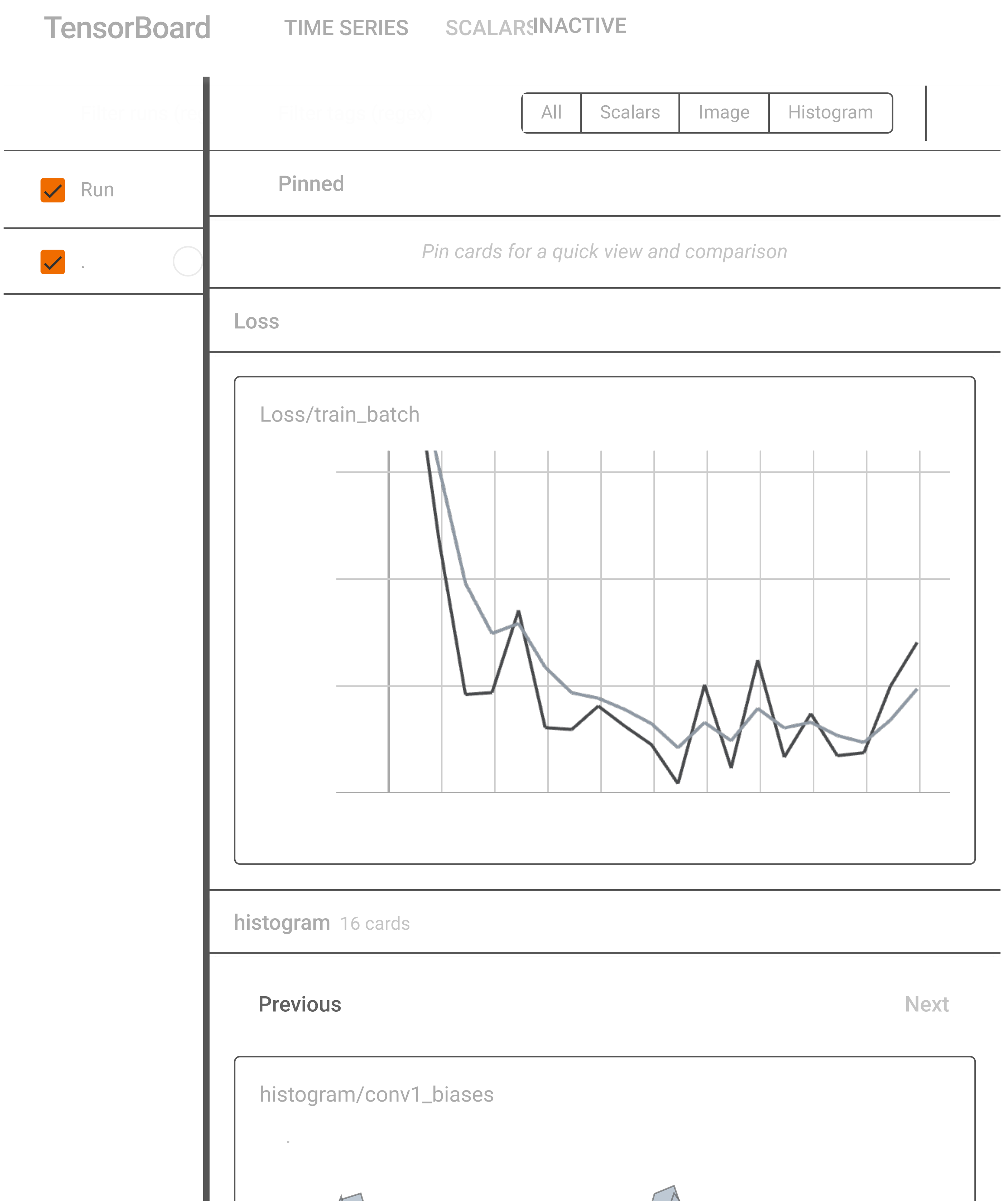
```
writer.close()
```

```
Epoch [2/2], Step [1270/2142], Loss: 2.2935
Epoch [2/2], Step [1280/2142], Loss: 2.3112
Epoch [2/2], Step [1290/2142], Loss: 2.2980
Epoch [2/2], Step [1300/2142], Loss: 2.2978
Epoch [2/2], Step [1310/2142], Loss: 2.2916
Epoch [2/2], Step [1320/2142], Loss: 2.3099
Epoch [2/2], Step [1330/2142], Loss: 2.3106
Epoch [2/2], Step [1340/2142], Loss: 2.3031
Epoch [2/2], Step [1350/2142], Loss: 2.2972
Epoch [2/2], Step [1360/2142], Loss: 2.3043
Epoch [2/2], Step [1370/2142], Loss: 2.3043
Epoch [2/2], Step [1380/2142], Loss: 2.2928
Epoch [2/2], Step [1390/2142], Loss: 2.3021
Epoch [2/2], Step [1400/2142], Loss: 2.3011
Epoch [2/2], Step [1410/2142], Loss: 2.2874
Epoch [2/2], Step [1420/2142], Loss: 2.2883
Epoch [2/2], Step [1430/2142], Loss: 2.2764
Epoch [2/2], Step [1440/2142], Loss: 2.3087
Epoch [2/2], Step [1450/2142], Loss: 2.3034
Epoch [2/2], Step [1460/2142], Loss: 2.3235
Epoch [2/2], Step [1470/2142], Loss: 2.2640
Epoch [2/2], Step [1480/2142], Loss: 2.2672
Epoch [2/2], Step [1490/2142], Loss: 2.3012
Epoch [2/2], Step [1500/2142], Loss: 2.2908
Epoch [2/2], Step [1510/2142], Loss: 2.3004
Epoch [2/2], Step [1520/2142], Loss: 2.3027
Epoch [2/2], Step [1530/2142], Loss: 2.3018
Epoch [2/2], Step [1540/2142], Loss: 2.3055
Epoch [2/2], Step [1550/2142], Loss: 2.3197
Epoch [2/2], Step [1560/2142], Loss: 2.3000
Epoch [2/2], Step [1570/2142], Loss: 2.3136
Epoch [2/2], Step [1580/2142], Loss: 2.3181
Epoch [2/2], Step [1590/2142], Loss: 2.3005
Epoch [2/2], Step [1600/2142], Loss: 2.3016
Epoch [2/2], Step [1610/2142], Loss: 2.2847
Epoch [2/2], Step [1620/2142], Loss: 2.2909
Epoch [2/2], Step [1630/2142], Loss: 2.3201
Epoch [2/2], Step [1640/2142], Loss: 2.2936
Epoch [2/2], Step [1650/2142], Loss: 2.3291
Epoch [2/2], Step [1660/2142], Loss: 2.2985
Epoch [2/2], Step [1670/2142], Loss: 2.2786
Epoch [2/2], Step [1680/2142], Loss: 2.2781
Epoch [2/2], Step [1690/2142], Loss: 2.2950
Epoch [2/2], Step [1700/2142], Loss: 2.2900
Epoch [2/2], Step [1710/2142], Loss: 2.3083
Epoch [2/2], Step [1720/2142], Loss: 2.2891
Epoch [2/2], Step [1730/2142], Loss: 2.2922
Epoch [2/2], Step [1740/2142], Loss: 2.2854
Epoch [2/2], Step [1750/2142], Loss: 2.3284
Epoch [2/2], Step [1760/2142], Loss: 2.3117
```

```
Epoch [2/2], Step [1770/2142], Loss: 2.3148
Epoch [2/2], Step [1780/2142], Loss: 2.2871
Epoch [2/2], Step [1790/2142], Loss: 2.2938
Epoch [2/2], Step [1800/2142], Loss: 2.3186
Epoch [2/2], Step [1810/2142], Loss: 2.2807
Epoch [2/2], Step [1820/2142], Loss: 2.2953
Epoch [2/2], Step [1830/2142], Loss: 2.3033
Epoch [2/2], Step [1840/2142], Loss: 2.3407
Epoch [2/2], Step [1850/2142], Loss: 2.3090
```

CNN Comparison

```
datetime_str = datetime.now().strftime('%b%d_%H-%M-%S')

base_folder = Path('runs/cnn_experiment')
base_folder.mkdir(parents=True, exist_ok=True)

logging_dir = base_folder / Path(datetime_str)
logging_dir.mkdir(exist_ok=True)

logging_dir = str(logging_dir.absolute())

writer = SummaryWriter(log_dir=logging_dir)

%tensorboard --logdir {logging_dir} --port 7007
```




```
for i, num_filters1 in enumerate(num_filters1_list):
    for j, num_filters2 in enumerate(num_filters2_list):
        for k, fc_size in enumerate(fc_size_list):
            for l, lr in enumerate(learning_rates):
```

```
for m, opt in enumerate(optimizers):
    model = AdaptiveCNNModel(num_filters1, num_filters2, fc_size)
    cnn_optimizer = opt(model.parameters(), lr=lr)
    cnn_criterion = nn.CrossEntropyLoss()
    print(f'Hyperparameters: [#{i+1} num_filters1={num_filters1}, #
    for epoch in range(2):  # Limiting to 2 epochs for demonstratio
        model.train()
        correct = 0
        running_loss = 0
        for batch_idx, (data, target) in enumerate(train_loader):
            cnn_optimizer.zero_grad()
            output = model(data)
            loss = F.nll_loss(output, target)
            loss.backward()
            cnn_optimizer.step()
            running_loss += loss.item()

            pred = output.argmax(dim=1, keepdim=True)
            correct += pred.eq(target.view_as(pred)).sum().item()

            n_iter = epoch * len(train_loader) + batch_idx
            if (batch_idx + 1) % logging_interval == 0:
                writer.add_scalar('Loss/train_batch', loss.item(),
                writer.add_scalar('Accuracy/train', 100 * correct /

                for key, value in model.stats.items():
                    value_flat = value.flatten()
                    writer.add_scalar(f'statistics/train_{key}_mean
                    writer.add_scalar(f'statistics/train_{key}_std'
                    writer.add_scalar(f'statistics/train_{key}_min'
                    writer.add_scalar(f'statistics/train_{key}_max'
                    writer.add_histogram(f'histogram/train_{key}',

                for name, param in model.named_parameters():
                    if 'weight' in name:
                        writer.add_scalar(f'statistics/train_{name}
                        writer.add_scalar(f'statistics/train_{name}
                        writer.add_scalar(f'statistics/train_{name}
                        writer.add_scalar(f'statistics/train_{name}
                        writer.add_histogram(f'histogram/train_{nam
                    elif 'bias' in name:
                        writer.add_scalar(f'statistics/rain_{name}_
                        writer.add_scalar(f'statisticsrain_{name}_m
                        writer.add_scalar(f'statistics/rain_{name}_
                        writer.add_scalar(f'statistics/rain_{name}_
                        writer.add_histogram(f'histogram/rain_{name

                print(f'Epoch [{epoch+1}/2], Step [{batch_idx+1}/{l
```

```
            train_accuracies.append(100 * correct / total)
            train_losses.append(running_loss / total)

            # Test Loop
            model.eval()
            correct = 0
            total = 0

            with torch.no_grad():
                for j, data in enumerate(test_loader):

                    images, labels = data
                    outputs = model(images)
                    _, predicted = torch.max(outputs.data, 1)
                    total += labels.size(0)
                    correct += (predicted == labels).sum().item()
                    m_iter = epoch * len(test_loader) + j
                    if (j + 1) % logging_interval == 0:
                          for key, value in model.stats.items():
                              value_flat = value.flatten()
                              writer.add_scalar(f'statistics/test_{key}
                              writer.add_scalar(f'statistics/test_{key}
                              writer.add_scalar(f'statistics/test_{key}
                              writer.add_scalar(f'statistics/test_{key}
                              writer.add_histogram(f'histogram/test_{ke

                          for name, param in model.named_parameters():
                              if 'weight' in name:
                                  writer.add_scalar(f'statistics/test_{
                                  writer.add_scalar(f'statistics/test_{
                                  writer.add_scalar(f'statistics/test_{
                                  writer.add_scalar(f'statistics/test_{
                                  writer.add_histogram(f'histogram/test_
                              elif 'bias' in name:
                                  writer.add_scalar(f'statistics/test_{
                                  writer.add_scalar(f'statistics/test_{
                                  writer.add_scalar(f'statistics/test_{
                                  writer.add_scalar(f'statistics/test_{
                                  writer.add_histogram(f'histogram/test_
            writer.add_scalar('Accuracy/test', 100 * correct / total, e
            print(f"Test Accuracy: {100 * correct / total}%")
            test_accuracies.append(100 * correct / total)
writer.close()
```

```
    Epoch [2/2], Step [1230/2142], Loss: 0.1224
    Epoch [2/2], Step [1240/2142], Loss: 0.0110
    Epoch [2/2], Step [1250/2142], Loss: 0.0514
    Epoch [2/2], Step [1260/2142], Loss: 0.0958
    Epoch [2/2], Step [1270/2142], Loss: 0.0364
```

```
Epoch [2/2], Step [1280/2142], Loss: 0.0514
Epoch [2/2], Step [1290/2142], Loss: 0.0323
Epoch [2/2], Step [1300/2142], Loss: 0.3162
Epoch [2/2], Step [1310/2142], Loss: 0.0842
Epoch [2/2], Step [1320/2142], Loss: 0.0295
Epoch [2/2], Step [1330/2142], Loss: 0.3445
Epoch [2/2], Step [1340/2142], Loss: 0.0933
Epoch [2/2], Step [1350/2142], Loss: 0.0151
Epoch [2/2], Step [1360/2142], Loss: 0.0173
Epoch [2/2], Step [1370/2142], Loss: 0.2274
Epoch [2/2], Step [1380/2142], Loss: 0.3112
Epoch [2/2], Step [1390/2142], Loss: 0.0226
Epoch [2/2], Step [1400/2142], Loss: 0.0058
Epoch [2/2], Step [1410/2142], Loss: 0.0115
Epoch [2/2], Step [1420/2142], Loss: 0.0970
Epoch [2/2], Step [1430/2142], Loss: 0.1925
Epoch [2/2], Step [1440/2142], Loss: 0.0069
Epoch [2/2], Step [1450/2142], Loss: 0.0170
Epoch [2/2], Step [1460/2142], Loss: 0.0397
Epoch [2/2], Step [1470/2142], Loss: 0.0341
Epoch [2/2], Step [1480/2142], Loss: 0.1366
Epoch [2/2], Step [1490/2142], Loss: 0.0322
Epoch [2/2], Step [1500/2142], Loss: 0.2234
Epoch [2/2], Step [1510/2142], Loss: 0.0547
Epoch [2/2], Step [1520/2142], Loss: 0.0091
Epoch [2/2], Step [1530/2142], Loss: 0.0106
Epoch [2/2], Step [1540/2142], Loss: 0.0209
Epoch [2/2], Step [1550/2142], Loss: 0.2992
Epoch [2/2], Step [1560/2142], Loss: 0.3741
Epoch [2/2], Step [1570/2142], Loss: 0.0175
Epoch [2/2], Step [1580/2142], Loss: 0.0764
Epoch [2/2], Step [1590/2142], Loss: 0.0872
Epoch [2/2], Step [1600/2142], Loss: 0.0700
Epoch [2/2], Step [1610/2142], Loss: 0.2254
Epoch [2/2], Step [1620/2142], Loss: 0.0114
Epoch [2/2], Step [1630/2142], Loss: 0.2529
Epoch [2/2], Step [1640/2142], Loss: 0.0503
Epoch [2/2], Step [1650/2142], Loss: 0.0189
Epoch [2/2], Step [1660/2142], Loss: 0.0263
Epoch [2/2], Step [1670/2142], Loss: 0.0956
Epoch [2/2], Step [1680/2142], Loss: 0.0076
Epoch [2/2], Step [1690/2142], Loss: 0.0230
Epoch [2/2], Step [1700/2142], Loss: 0.0909
Epoch [2/2], Step [1710/2142], Loss: 0.0690
Epoch [2/2], Step [1720/2142], Loss: 0.0423
Epoch [2/2], Step [1730/2142], Loss: 0.0151
Epoch [2/2], Step [1740/2142], Loss: 0.0748
Epoch [2/2], Step [1750/2142], Loss: 0.1872
Epoch [2/2], Step [1760/2142], Loss: 0.0917
Epoch [2/2], Step [1770/2142], Loss: 0.1895
Epoch [2/2], Step [1780/2142], Loss: 0.0517
Epoch [2/2], Step [1790/2142], Loss: 0.0204
Epoch [2/2], Step [1800/2142], Loss: 0.0279
```

Epoch [2/2], Step [1810/2142], Loss: 0.0513