PART-1: Dense Network

```
%load_ext tensorboard
import tensorflow as tf
import datetime

# Clear any logs from previous runs
!rm -rf ./logs/
```

```
%reload_ext tensorboard
```

```
import os
import torch
import torch.nn as nn
import torch.optim as optim
from torch.utils.data import DataLoader, TensorDataset
from torchvision import datasets, transforms
import numpy as np
from google.colab import drive
from torch.utils.tensorboard import SummaryWriter  # Corrected import statement
```

```
# Load the data from the file
data = np.load('emnist_letters.npz')

# Access the arrays containing images and labels
train_images = data['train_images']
train_labels = data['train_labels']
val_images = data['validate_images']
val_labels = data['validate_labels']
test_images = data['test_images']
test_labels = data['test_labels']


# Create datasets
train_dataset = TensorDataset(torch.tensor(train_images), torch.tensor(train_labels)
val_dataset = TensorDataset(torch.tensor(val_images), torch.tensor(val_labels))
test_dataset = TensorDataset(torch.tensor(test_images), torch.tensor(test_labels))

# Create data loaders
train_loader = DataLoader(train_dataset, batch_size=64, shuffle=True)
val_loader = DataLoader(val_dataset, batch_size=64)
```

```
test_loader = DataLoader(test_dataset, batch_size=64)
```

```
device = (
    "cuda"
    if torch.cuda.is_available()
    else "mps"
    if torch.backends.mps.is_available()
    else "cpu"
)
print(f"Using {device} device")
```

```
Using cuda device
```

```
# Model Definition
class NeuralNetwork(nn.Module):
    def __init__(self):
        super(NeuralNetwork, self).__init__()
        # self.flatten = nn.Flatten()
        self.fc1 = nn.Linear(28 * 28, 300, dtype=torch.float32)  # One neuron per fe
        self.fc2 = nn.Linear(300,150, dtype=torch.float32)  # arbitrarily selecting
        self.fc3 = nn.Linear(150, 50, dtype=torch.float32)
        self.fc4 = nn.Linear(50, 27, dtype=torch.float32)  # 27 Classes present in d

    def forward(self, x):
        # x = self.flatten(x)
        # Cast input data to torch.float32
        x = x.to(torch.float32)
        x = torch.relu(self.fc1(x))
        x = torch.relu(self.fc2(x))
        x = torch.relu(self.fc3(x))
        x = self.fc4(x)
        return x


model = NeuralNetwork()
print(model)

# Inspect a sample of the data
for data, target in train_loader:
    print("Target shape:", target.shape)
    print("Target dataset:", target)
    break
```

```
NeuralNetwork(
  (fc1): Linear(in_features=784, out_features=300, bias=True)
  (fc2): Linear(in_features=300, out_features=150, bias=True)
```

```
    (fc3): Linear(in_features=150, out_features=50, bias=True)
    (fc4): Linear(in_features=50, out_features=27, bias=True)
  )
  Target shape: torch.Size([64, 27])
  Target dataset: tensor([[0., 0., 0.,  ..., 0., 0., 0.],
          [0., 0., 0.,  ..., 0., 0., 0.],
          [0., 0., 0.,  ..., 0., 1., 0.],
          ...,
          [0., 0., 0.,  ..., 0., 0., 0.],
          [0., 1., 0.,  ..., 0., 0., 0.],
          [0., 0., 0.,  ..., 1., 0., 0.]])
```

```
# Training
optimizer = optim.Adam(model.parameters(), lr=0.001)
criterion = nn.CrossEntropyLoss()

# Create a directory for TensorBoard logs
log_dir = "logs/fit/" + datetime.datetime.now().strftime("%Y%m%d-%H%M%S")
os.makedirs(log_dir, exist_ok=True)

# Create a SummaryWriter for TensorBoard
writer = SummaryWriter(log_dir=log_dir)

def train(model, train_loader, optimizer, criterion, epochs):
    for epoch in range(epochs):
        model.train()
        epoch_loss = 0.0  # Track epoch loss
        for batch_idx, (data, target) in enumerate(train_loader):
            optimizer.zero_grad()
            output = model(data)

            # Convert one-hot encoded target to class labels (1D tensor)
            target_labels = torch.nonzero(target, as_tuple=True)[1]

            # Calculate loss using class labels
            loss = criterion(output, target_labels)

            loss.backward()
            optimizer.step()

            epoch_loss += loss.item()  # Accumulate batch loss

        # Print epoch results
        print('Epoch {} - Loss: {:.6f}'.format(epoch, epoch_loss / len(train_loader)
        # Log the training loss to TensorBoard
        writer.add_scalar('Loss/train', epoch_loss / len(train_loader), epoch)
```

```
def test(model, test_loader):
    model.eval()
    test_loss = 0
    correct = 0
    with torch.no_grad():
        for data, target in test_loader:
            output = model(data)
            # Convert one-hot encoded target to class labels (1D tensor)
            target_labels = torch.nonzero(target, as_tuple=True)[1]
            test_loss += criterion(output, target_labels).item()
            pred = output.argmax(dim=1, keepdim=True)
            correct += pred.eq(target_labels.view_as(pred)).sum().item()

    test_loss /= len(test_loader.dataset)
    print('\nTest set: Average loss: {:.4f}, Accuracy: {}/{} ({:.0f}%)\n'.format(
        test_loss, correct, len(test_loader.dataset),
        100. * correct / len(test_loader.dataset)))
```

```
train(model, train_loader, optimizer, criterion, epochs=10)
# Close the SummaryWriter
writer.close()
test(model, test_loader)
```

```
Epoch 0 - Loss: 0.835394
Epoch 1 - Loss: 0.388811
Epoch 2 - Loss: 0.306091
Epoch 3 - Loss: 0.261132
Epoch 4 - Loss: 0.230223
Epoch 5 - Loss: 0.204188
Epoch 6 - Loss: 0.187152
Epoch 7 - Loss: 0.171736
Epoch 8 - Loss: 0.158153
Epoch 9 - Loss: 0.146823

Test set: Average loss: 0.0050, Accuracy: 18932/20800 (91%)
```

Graphing the data from the Dense network

First, lets generate confusion matrix of the dense network's classifications

We'll start by calculating the number of true and false positives

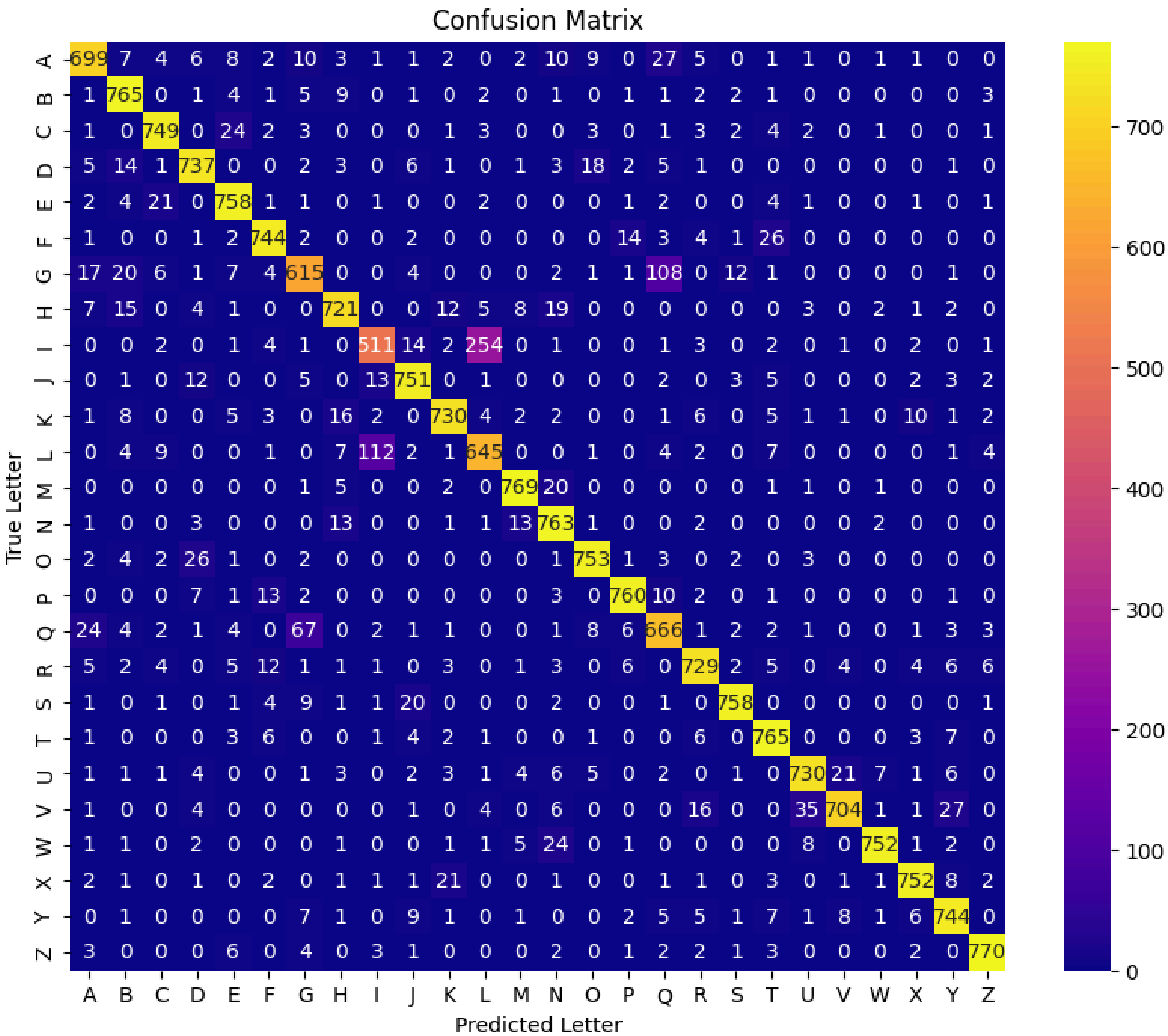
```
import numpy as np

# Generate predictions for the test set
def generate_predictions(model, test_loader):
    model.eval()
```

```
    all_predictions = []
    all_targets = []
    with torch.no_grad():
        for data, target in test_loader:
            output = model(data)
            all_predictions.extend(output.argmax(dim=1).cpu().numpy())
            all_targets.extend(target.cpu().numpy())
    return np.array(all_predictions), np.array(all_targets)
def compute_tp_fp(predictions, targets, class_label):
    # Convert one-hot encoded targets to class labels
    target_labels = np.argmax(targets, axis=1)

    # Compute True Positives (TP) and False Positives (FP) for the specified class l
    tp = np.sum((predictions == class_label) & (target_labels == class_label))
    fp = np.sum((predictions == class_label) & (target_labels != class_label))

    return tp, fp

# Generate predictions for the test set
test_predictions, test_targets = generate_predictions(model, test_loader)

# Compute TP and FP for each class
for class_label in range(27):
    tp, fp = compute_tp_fp(test_predictions, test_targets, class_label)
    print(f"Class {class_label}: TP={tp}, FP={fp}")
```

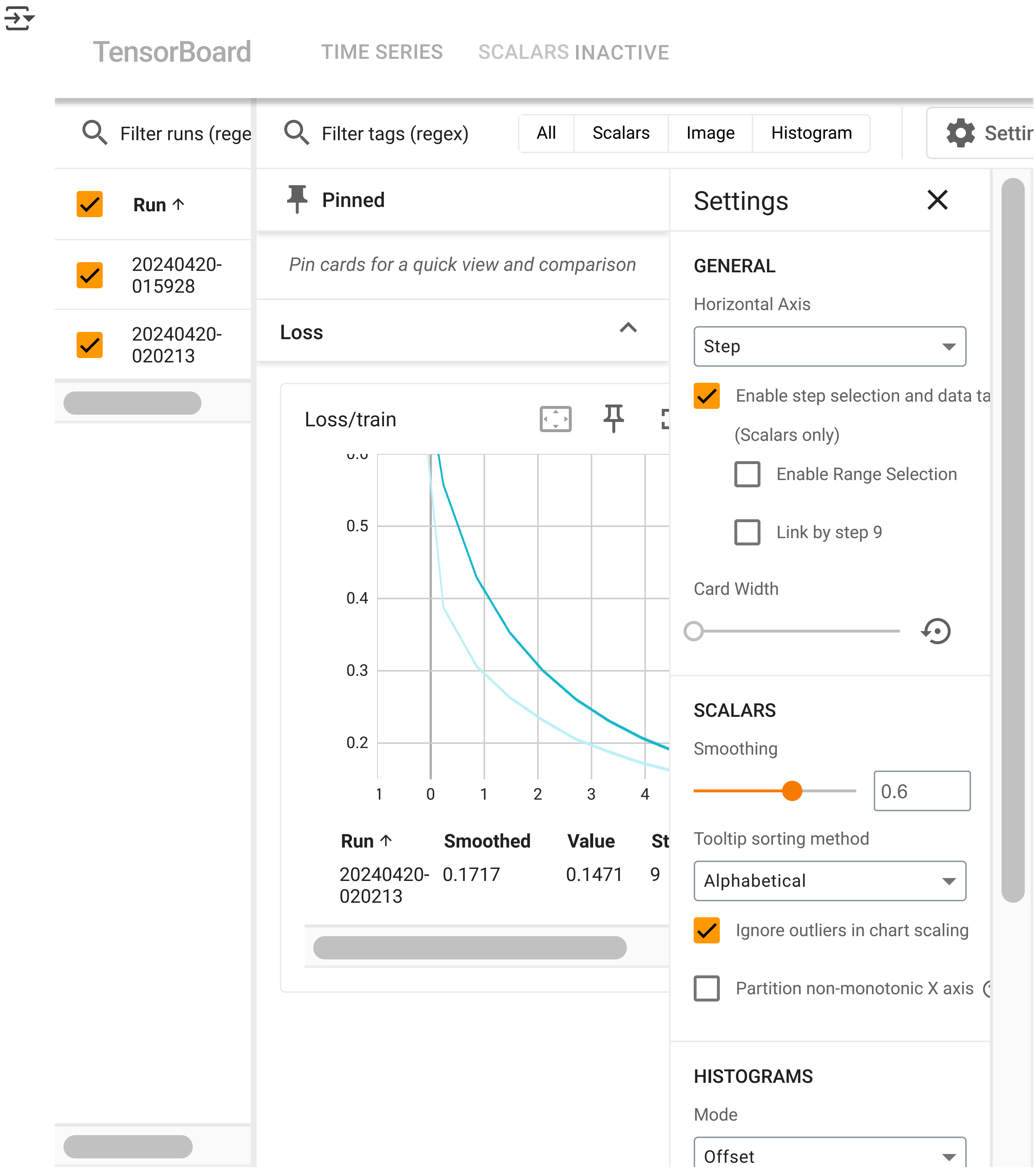
```
Class 0: TP=0, FP=0
Class 1: TP=699, FP=77
Class 2: TP=765, FP=87
Class 3: TP=749, FP=53
Class 4: TP=737, FP=73
Class 5: TP=758, FP=73
Class 6: TP=744, FP=55
Class 7: TP=615, FP=123
Class 8: TP=721, FP=64
Class 9: TP=511, FP=138
Class 10: TP=751, FP=69
Class 11: TP=730, FP=54
Class 12: TP=645, FP=279
Class 13: TP=769, FP=37
Class 14: TP=763, FP=107
Class 15: TP=753, FP=47
Class 16: TP=760, FP=36
Class 17: TP=666, FP=179
Class 18: TP=729, FP=61
Class 19: TP=758, FP=29
Class 20: TP=765, FP=78
Class 21: TP=730, FP=57
Class 22: TP=704, FP=36
Class 23: TP=752, FP=17
Class 24: TP=752, FP=36
Class 25: TP=744, FP=69
Class 26: TP=770, FP=26
```

Confusion Matrix

```
import matplotlib.pyplot as plt
import seaborn as sns

from sklearn.metrics import confusion_matrix

# Generate predictions for the test set
test_predictions, test_targets = generate_predictions(model, test_loader)

# Convert one-hot encoded targets to class labels
test_targets_single = np.argmax(test_targets, axis=1)

# Compute confusion matrix
cm = confusion_matrix(test_targets_single, test_predictions)

# Generate labels for the letters using Unicode
letter_labels = [chr(ord('A') + i) for i in range(26)]

# Plot confusion matrix as heatmap with x-axis on top and "plasma" colormap
plt.figure(figsize=(10, 8))
sns.heatmap(cm, annot=True, fmt="d", cmap="plasma", xticklabels=letter_labels, ytick
plt.xlabel("Predicted Letter")
plt.ylabel("True Letter")
plt.title("Confusion Matrix")
plt.show()
```




```
%tensorboard --logdir logs/fit
```



Performance Comparison between Dense Network and OPIUM based Classifier

The Dense Network used 3 hidden layers but the OPIUM based classifier used 10,000 hidden layers. Even with the huge increase in the hidden layers the accuracy of the OPIUM based classifier on the letters dataset remained at (85.15% ± 0.12%), while the dense network had a much better accuracy of 91%. In comparison with the OPIUM based classifier the dense network is more compact and runs more efficiently.

PART-2: Convolutional Network

```
import tensorflow as tf
from tensorflow.keras.callbacks import EarlyStopping, ModelCheckpoint, TensorBoard
import datetime
import numpy as np
import matplotlib.pyplot as plt
import tensorflow as tf
from tensorflow.keras.models import Sequential
from tensorflow.keras.layers import Conv2D, Activation, BatchNormalization, MaxPooli
from tensorflow.keras.models import load_model
```

```
# Clear any logs from previous runs
!rm -rf ./logs/
# Load the data from the file
data = np.load('emnist_letters.npz')

# Access the arrays containing images and labels
train_images = data['train_images']
train_labels = data['train_labels']
validate_images = data['validate_images']
validate_labels = data['validate_labels']
test_images = data['test_images']
test_labels = data['test_labels']
```

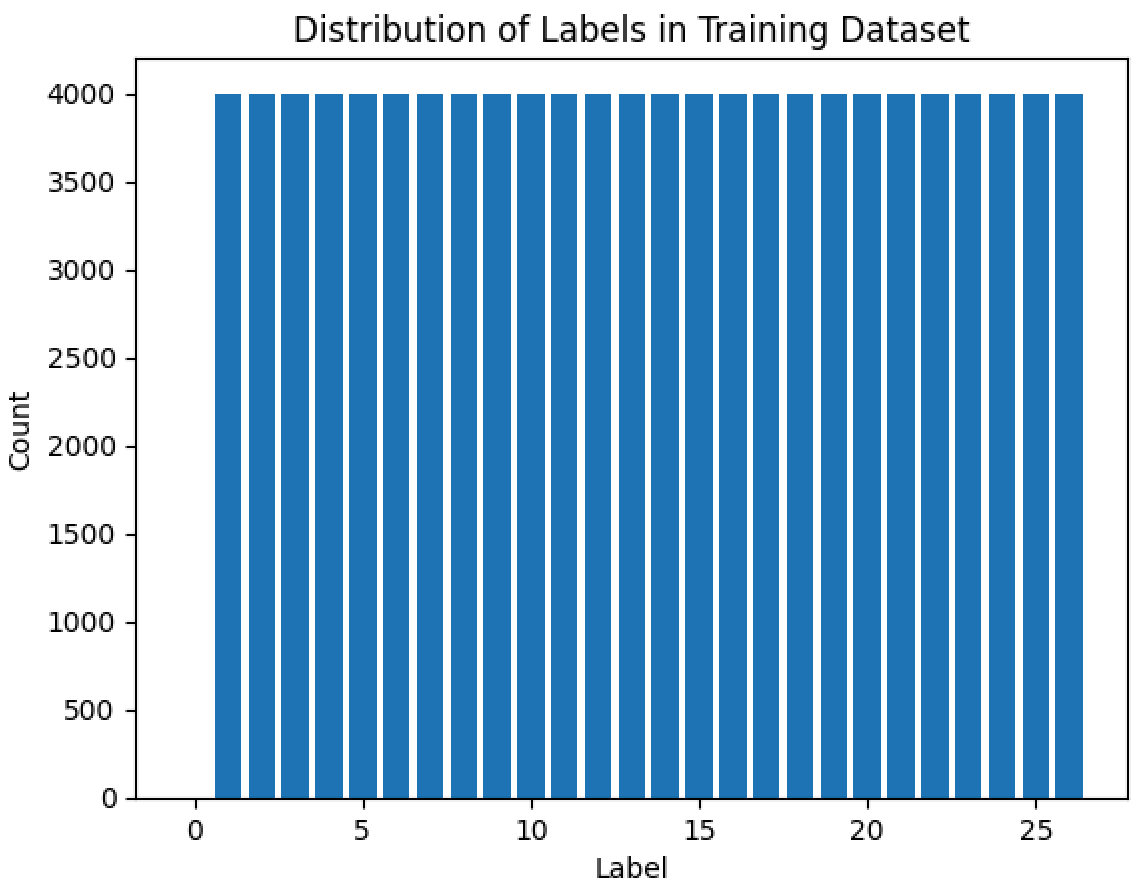
```
# Define the log directory for TensorBoard
log_dir = "logs/fit/" + datetime.datetime.now().strftime("%Y%m%d-%H%M%S")

# Load the TensorBoard notebook extension
%load_ext tensorboard

# Define TensorBoard callback
tensorboard_callback = TensorBoard(log_dir=log_dir, histogram_freq=1)
```

```
The tensorboard extension is already loaded. To reload it, use:
  %reload_ext tensorboard
```

```
num_train_images = train_images.shape[0]
print("Number of images in the training dataset:", num_train_images)
```

```
Number of images in the training dataset: 104000
```

```
# Count occurrences of each label
label_counts = np.sum(train_labels, axis=0)

# Plot the distribution of labels
plt.bar(range(len(label_counts)), label_counts)
plt.xlabel('Label')
plt.ylabel('Count')
plt.title('Distribution of Labels in Training Dataset')
plt.show()
```




```
print(train_images.shape)
print(validate_images.shape)
```

```
(104000, 784)
(20800, 784)
```

```
!pip install tensorflow
```

```
Requirement already satisfied: tensorflow in /usr/local/lib/python3.10/dist-pack
Requirement already satisfied: absl-py>=1.0.0 in /usr/local/lib/python3.10/dist-
```

```
Requirement already satisfied: astunparse>=1.6.0 in /usr/local/lib/python3.10/di
Requirement already satisfied: flatbuffers>=23.5.26 in /usr/local/lib/python3.10
Requirement already satisfied: gast!=0.5.0,!=0.5.1,!=0.5.2,>=0.2.1 in /usr/local
Requirement already satisfied: google-pasta>=0.1.1 in /usr/local/lib/python3.10/
Requirement already satisfied: h5py>=2.9.0 in /usr/local/lib/python3.10/dist-pac
Requirement already satisfied: libclang>=13.0.0 in /usr/local/lib/python3.10/dis
Requirement already satisfied: ml-dtypes~=0.2.0 in /usr/local/lib/python3.10/dis
Requirement already satisfied: numpy<2.0.0,>=1.23.5 in /usr/local/lib/python3.10
Requirement already satisfied: opt-einsum>=2.3.2 in /usr/local/lib/python3.10/di
Requirement already satisfied: packaging in /usr/local/lib/python3.10/dist-packa
Requirement already satisfied: protobuf!=4.21.0,!=4.21.1,!=4.21.2,!=4.21.3,!=4.2
Requirement already satisfied: setuptools in /usr/local/lib/python3.10/dist-pack
Requirement already satisfied: six>=1.12.0 in /usr/local/lib/python3.10/dist-pac
Requirement already satisfied: termcolor>=1.1.0 in /usr/local/lib/python3.10/dis
Requirement already satisfied: typing-extensions>=3.6.6 in /usr/local/lib/python
Requirement already satisfied: wrapt<1.15,>=1.11.0 in /usr/local/lib/python3.10/
Requirement already satisfied: tensorflow-io-gcs-filesystem>=0.23.1 in /usr/loca
Requirement already satisfied: grpcio<2.0,>=1.24.3 in /usr/local/lib/python3.10/
Requirement already satisfied: tensorboard<2.16,>=2.15 in /usr/local/lib/python3
Requirement already satisfied: tensorflow-estimator<2.16,>=2.15.0 in /usr/local/
Requirement already satisfied: keras<2.16,>=2.15.0 in /usr/local/lib/python3.10/
Requirement already satisfied: wheel<1.0,>=0.23.0 in /usr/local/lib/python3.10/d
Requirement already satisfied: google-auth<3,>=1.6.3 in /usr/local/lib/python3.1
Requirement already satisfied: google-auth-oauthlib<2,>=0.5 in /usr/local/lib/py
Requirement already satisfied: markdown>=2.6.8 in /usr/local/lib/python3.10/dist
Requirement already satisfied: requests<3,>=2.21.0 in /usr/local/lib/python3.10/
Requirement already satisfied: tensorboard-data-server<0.8.0,>=0.7.0 in /usr/loc
Requirement already satisfied: werkzeug>=1.0.1 in /usr/local/lib/python3.10/dist
Requirement already satisfied: cachetools<6.0,>=2.0.0 in /usr/local/lib/python3.
Requirement already satisfied: pyasn1-modules>=0.2.1 in /usr/local/lib/python3.1
Requirement already satisfied: rsa<5,>=3.1.4 in /usr/local/lib/python3.10/dist-p
Requirement already satisfied: requests-oauthlib>=0.7.0 in /usr/local/lib/python
Requirement already satisfied: charset-normalizer<4,>=2 in /usr/local/lib/python
Requirement already satisfied: idna<4,>=2.5 in /usr/local/lib/python3.10/dist-pa
Requirement already satisfied: urllib3<3,>=1.21.1 in /usr/local/lib/python3.10/d
Requirement already satisfied: certifi>=2017.4.17 in /usr/local/lib/python3.10/d
Requirement already satisfied: MarkupSafe>=2.1.1 in /usr/local/lib/python3.10/di
Requirement already satisfied: pyasn1<0.7.0,>=0.4.6 in /usr/local/lib/python3.10
Requirement already satisfied: oauthlib>=3.0.0 in /usr/local/lib/python3.10/dist
```

```
# Define the strategy
strategy = tf.distribute.MirroredStrategy()
with strategy.scope():
    model = Sequential()

    # Reshape the input images to their original 28x28 shape (assuming original shap
    model.add(tf.keras.layers.Reshape((28, 28, 1), input_shape=(784,)))

    # Feature Learning Layers
    model.add(Conv2D(32,                  # Number of filters/Kernels
                     (3,3),               # Size of kernels (3x3 matrix)
                     strides = 1,         # Step size for sliding the kernel across
```

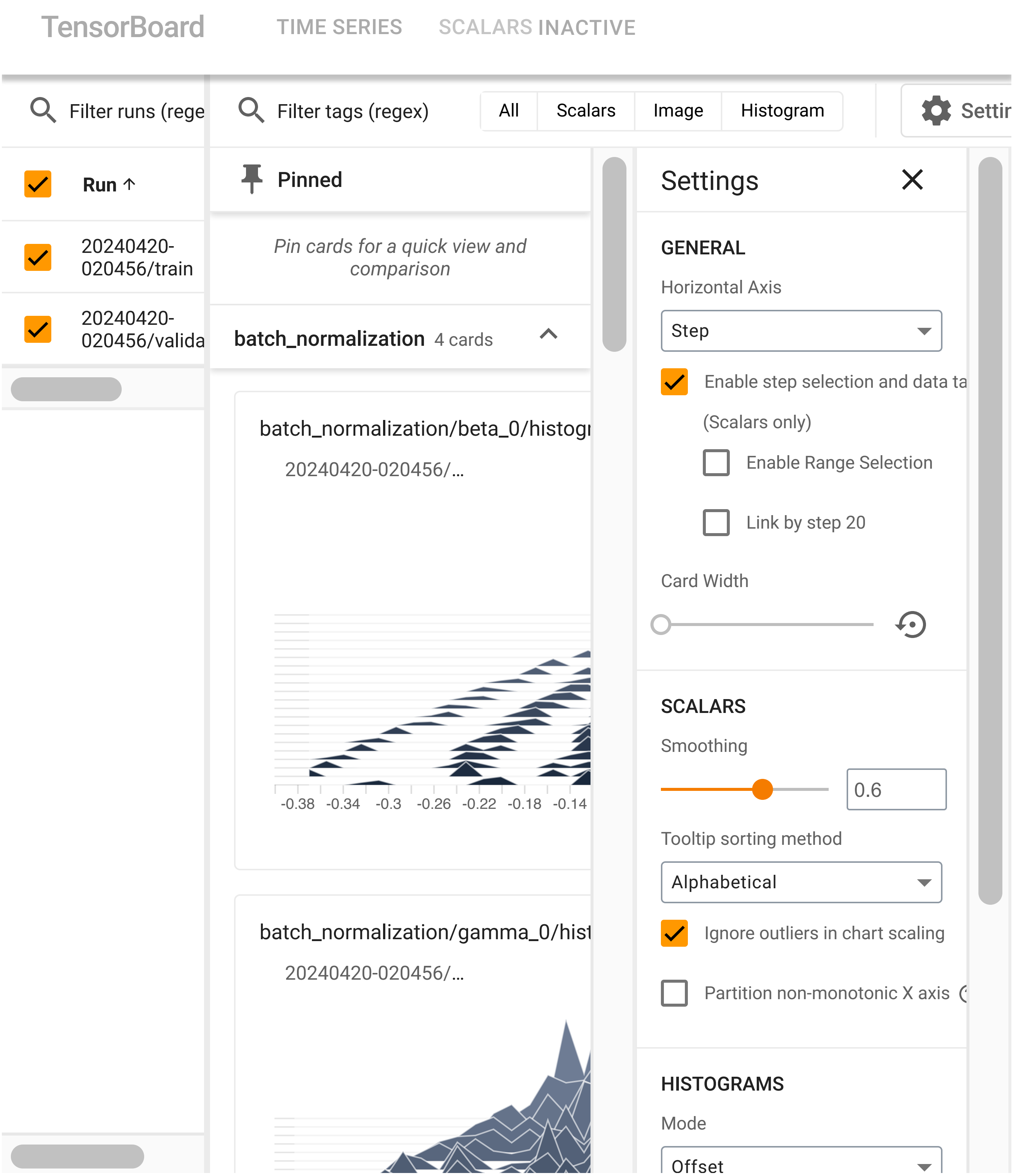
```
                    padding = 'same'    # 'Same' ensures that the output feature ma
                   ))
    model.add(Activation('relu'))# Activation function
    model.add(BatchNormalization())
    model.add(MaxPooling2D(pool_size = (2,2), padding = 'same'))
    model.add(Dropout(0.2))

    model.add(Conv2D(64, (5,5), padding = 'same'))
    model.add(Activation('relu'))
    model.add(BatchNormalization())
    model.add(MaxPooling2D(pool_size = (2,2), padding = 'same'))
    model.add(Dropout(0.2))

    model.add(Conv2D(128, (3,3), padding = 'same'))
    model.add(Activation('relu'))
    model.add(BatchNormalization())
    model.add(MaxPooling2D(pool_size = (2,2), padding = 'same'))
    model.add(Dropout(0.3))

    # Flattening tensors
    model.add(Flatten())

    # Fully-Connected Layers
    model.add(Dense(2048))
    model.add(Activation('relu'))
    model.add(Dropout(0.5))

    # Output Layer
    model.add(Dense(27, activation = 'softmax')) # Classification layer


model.compile(optimizer = tf.keras.optimizers.RMSprop(0.0001), # 1e-4
              loss = 'categorical_crossentropy', # Ideal for multiclass tasks
              metrics = ['accuracy']) # Evaluation metric

# Defining an Early Stopping and Model Checkpoints
early_stopping = EarlyStopping(monitor = 'val_accuracy',
                              patience = 5, mode = 'max',
                              restore_best_weights = True)

checkpoint = ModelCheckpoint('best_model.h5',
                            monitor = 'val_accuracy',
                            save_best_only = True)


# Define the number of epochs
num_epochs = 50

# Fit the model to the training data
history = model.fit(train_images, train_labels,
                    epochs=num_epochs,
                    validation_data=(validate_images, validate_labels),
```

```
                    callbacks=[early_stopping, checkpoint, tensorboard_callback])
```

```
Epoch 1/50
3248/3250 [============================>.] - ETA: 0s - loss: 0.7824 - accuracy:
  saving_api.save_model(
3250/3250 [==============================] - 37s 8ms/step - loss: 0.7821 - accur
Epoch 2/50
3250/3250 [==============================] - 27s 8ms/step - loss: 0.3694 - accur
Epoch 3/50
3250/3250 [==============================] - 25s 8ms/step - loss: 0.3162 - accur
Epoch 4/50
3250/3250 [==============================] - 28s 9ms/step - loss: 0.2871 - accur
Epoch 5/50
3250/3250 [==============================] - 26s 8ms/step - loss: 0.2725 - accur
Epoch 6/50
3250/3250 [==============================] - 25s 8ms/step - loss: 0.2670 - accur
Epoch 7/50
3250/3250 [==============================] - 26s 8ms/step - loss: 0.2585 - accur
Epoch 8/50
3250/3250 [==============================] - 25s 8ms/step - loss: 0.2532 - accur
Epoch 9/50
3250/3250 [==============================] - 27s 8ms/step - loss: 0.2506 - accur
Epoch 10/50
3250/3250 [==============================] - 26s 8ms/step - loss: 0.2447 - accur
Epoch 11/50
3250/3250 [==============================] - 25s 8ms/step - loss: 0.2402 - accur
Epoch 12/50
3250/3250 [==============================] - 25s 8ms/step - loss: 0.2404 - accur
Epoch 13/50
3250/3250 [==============================] - 25s 8ms/step - loss: 0.2337 - accur
Epoch 14/50
3250/3250 [==============================] - 25s 8ms/step - loss: 0.2322 - accur
Epoch 15/50
3250/3250 [==============================] - 27s 8ms/step - loss: 0.2253 - accur
Epoch 16/50
3250/3250 [==============================] - 25s 8ms/step - loss: 0.2241 - accur
Epoch 17/50
3250/3250 [==============================] - 25s 8ms/step - loss: 0.2201 - accur
Epoch 18/50
3250/3250 [==============================] - 25s 8ms/step - loss: 0.2183 - accur
Epoch 19/50
3250/3250 [==============================] - 25s 8ms/step - loss: 0.2146 - accur
Epoch 20/50
3250/3250 [==============================] - 26s 8ms/step - loss: 0.2164 - accur
Epoch 21/50
3250/3250 [==============================] - 25s 8ms/step - loss: 0.2114 - accur
```

```
%tensorboard --logdir logs/fit
```

```
Reusing TensorBoard on port 6006 (pid 3879), started 0:10:45 ago. (Use '!kill 3879' to kill it.)
```




```
# Load the best model
best_model = load_model('best_model.h5')

# Evaluate the best model on test data
test_loss, test_accuracy = best_model.evaluate(test_images, test_labels)
```

```
print('Test Loss:', test_loss)
print('Test Accuracy:', test_accuracy)
```

```
650/650 [==============================] - 2s 3ms/step - loss: 0.1834 - accuracy
Test Loss: 0.18337306380271912
Test Accuracy: 0.9402884840965271
```

Generate True and False Positives for Convolutional Network

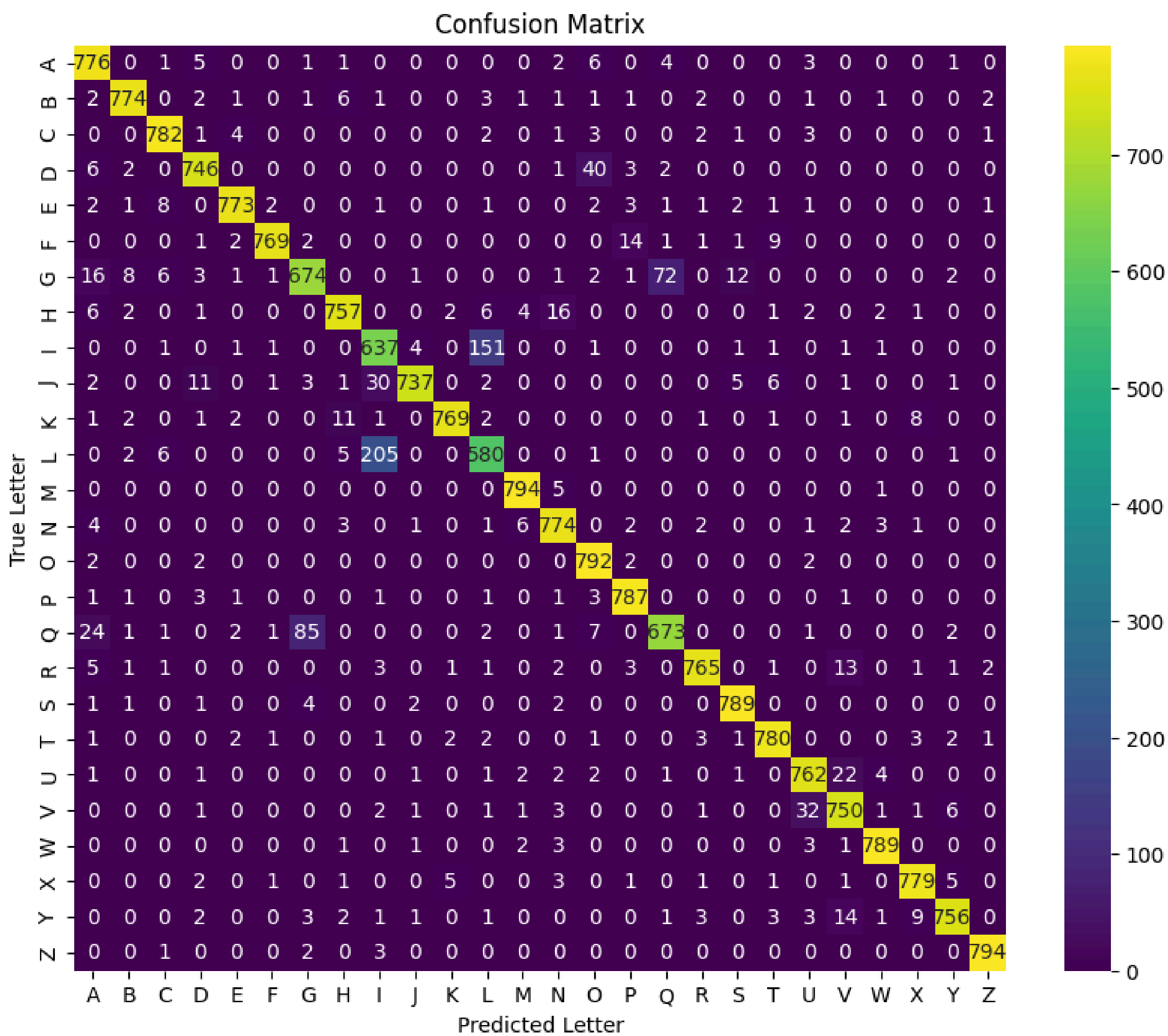
```
import numpy as np
from tensorflow.keras.models import load_model

# Load the best model
best_model = load_model('best_model.h5')

# Generate predictions for the test set
def generate_predictions(model, test_images):
    predictions = model.predict(test_images)
    return np.argmax(predictions, axis=1)

# Load the test data
test_images = data['test_images']
test_labels = data['test_labels']

# Flatten the test labels
test_labels_flat = np.argmax(test_labels, axis=1)

# Compute TP and FP for each class
def compute_tp_fp(predictions, targets, class_label):
    # Compute True Positives (TP) and False Positives (FP) for the specified class l
    tp = np.sum((predictions == class_label) & (targets == class_label))
    fp = np.sum((predictions == class_label) & (targets != class_label))
    return tp, fp

# Generate predictions for the test set
test_predictions = generate_predictions(best_model, test_images)

# Compute TP and FP for each class
for class_label in range(27):
    tp, fp = compute_tp_fp(test_predictions, test_labels_flat, class_label)
    print(f"Class {class_label}: TP={tp}, FP={fp}")
```

```
650/650 [==============================] - 1s 2ms/step
Class 0: TP=0, FP=0
Class 1: TP=776, FP=74
Class 2: TP=774, FP=21
Class 3: TP=782, FP=25
Class 4: TP=746, FP=37
```

```
Class 5: TP=773, FP=16
Class 6: TP=769, FP=8
Class 7: TP=674, FP=101
Class 8: TP=757, FP=31
Class 9: TP=637, FP=249
Class 10: TP=737, FP=12
Class 11: TP=769, FP=10
Class 12: TP=580, FP=177
Class 13: TP=794, FP=16
Class 14: TP=774, FP=44
Class 15: TP=792, FP=69
Class 16: TP=787, FP=30
Class 17: TP=673, FP=82
Class 18: TP=765, FP=17
Class 19: TP=789, FP=24
Class 20: TP=780, FP=24
Class 21: TP=762, FP=52
Class 22: TP=750, FP=57
Class 23: TP=789, FP=14
Class 24: TP=779, FP=24
Class 25: TP=756, FP=21
Class 26: TP=794, FP=7
```

Confusion Matrix

```
import matplotlib.pyplot as plt
import seaborn as sns

# Define class labels (assuming class labels are represented as integers from 0 to 2
class_labels = range(26)
letter_labels = [chr(ord('A') + i) for i in class_labels]

# Compute confusion matrix
cm = confusion_matrix(test_labels_flat, test_predictions)

# Plot confusion matrix as heatmap with "viridis" colormap
plt.figure(figsize=(10, 8))
sns.heatmap(cm, annot=True, fmt="d", cmap="viridis", xticklabels=letter_labels, ytic
plt.xlabel("Predicted Letter")
plt.ylabel("True Letter")
plt.title("Confusion Matrix")
plt.show()
```



Performance Comparison between Dense Network and CNN

The dense network performed at 91% accuracy while the convolution network performed a bit better at 94%. The models both struggled in the same areas and both had most of their misidentifications in the same place. Similar letters were frequently misidentified as each other- for example both of the networks had the most issues misidentifying the letter I as the letter L, and vice versa. The second highest misidentifications were Q and G. Overall though, the Convolution network was more consistent in identifying letters, with far more pairs of letters at 0 total misidentifications.

Part-3: GAN

```
import numpy as np
from torch.utils.data import Dataset, DataLoader
import argparse
import os
import random
import torch
import torch.nn as nn
import torch.nn.parallel
import torch.optim as optim
import torch.utils.data
import torchvision.datasets as dset
import torchvision.transforms as transforms
import torchvision.utils as vutils
import numpy as np
import matplotlib.pyplot as plt
import matplotlib.animation as animation
from IPython.display import HTML
from torch.utils.tensorboard import SummaryWriter
writer = SummaryWriter()

# Load the data from the file

data = np.load('emnist_letters.npz')

# Access the arrays containing images and labels
train_images = data['train_images']
train_labels = data['train_labels']
validate_images = data['validate_images']
validate_labels = data['validate_labels']
test_images = data['test_images']
test_labels = data['test_labels']


# Concatenate images and labels arrays
all_images = np.concatenate([train_images, validate_images, test_images], axis=0)
all_labels = np.concatenate([train_labels, validate_labels, test_labels], axis=0)



# Number of workers for dataloader
workers = 2

# Batch size during training
```

```
batch_size = 128

# Spatial size of training images. All images will be resized to this
#   size using a transformer.
image_size = 64

# Number of channels in the training images. For color images this is 3
nc = 1

# Size of z latent vector (i.e. size of generator input)
nz = 100

# Size of feature maps in generator
ngf = 28

# Size of feature maps in discriminator
ndf = 28

# Number of training epochs
num_epochs = 50

# Learning rate for optimizers
lr = 0.0002

# Beta1 hyperparameter for Adam optimizers
beta1 = 0.5

# Number of GPUs available. Use 0 for CPU mode.
ngpu = 1




class CustomDataset(Dataset):
    def __init__(self, images, labels, transform=None):
        self.images = images
        self.labels = labels
        self.transform = transform

    def __len__(self):
        return len(self.images)

    def __getitem__(self, idx):
        image = self.images[idx].reshape(28, 28)  # Reshape flattened image to 2D
        label = self.labels[idx]

        if self.transform:
            image = self.transform(image)

        return image, label

# Transform for image preprocessing
```

```
transform = transforms.Compose([
    transforms.ToPILImage(),
    transforms.Resize(image_size),
    transforms.CenterCrop(image_size),
    transforms.ToTensor()
])

# Create custom dataset instances
train_dataset = CustomDataset(all_images, all_labels, transform=transform)


# Create dataloaders
train_dataloader = DataLoader(train_dataset, batch_size=batch_size, shuffle=True, nu


# Decide which device we want to run on
device = torch.device("cuda:0" if (torch.cuda.is_available() and ngpu > 0) else "cpu
real_batch = next(iter(train_dataloader))
plt.figure(figsize=(8,8))
plt.axis("off")
plt.title("Training Images")
plt.imshow(np.transpose(vutils.make_grid(real_batch[0].to(device)[:64], padding=2, n
plt.show()
```

```
/usr/lib/python3.10/multiprocessing/popen_fork.py:66: RuntimeWarning: os.fork()
  self.pid = os.fork()
```

Training Images

```
# custom weights initialization called on ``netG`` and ``netD``
def weights_init(m):
    classname = m.__class__.__name__
    if classname.find('Conv') != -1:
        nn.init.normal_(m.weight.data, 0.0, 0.02)
    elif classname.find('BatchNorm') != -1:
        nn.init.normal_(m.weight.data, 1.0, 0.02)
        nn.init.constant_(m.bias.data, 0)


# Generator Code
```

```
class Generator(nn.Module):
    def __init__(self, ngpu):
        super(Generator, self).__init__()
        self.ngpu = ngpu
        self.main = nn.Sequential(
            # input is Z, going into a convolution
            nn.ConvTranspose2d( nz, ngf * 8, 4, 1, 0, bias=False),
            nn.BatchNorm2d(ngf * 8),
            nn.ReLU(True),
            # state size. ``(ngf*8) x 4 x 4``
            nn.ConvTranspose2d(ngf * 8, ngf * 4, 4, 2, 1, bias=False),
            nn.BatchNorm2d(ngf * 4),
            nn.ReLU(True),
            # state size. ``(ngf*4) x 8 x 8``
            nn.ConvTranspose2d( ngf * 4, ngf * 2, 4, 2, 1, bias=False),
            nn.BatchNorm2d(ngf * 2),
            nn.ReLU(True),
            # state size. ``(ngf*2) x 16 x 16``
            nn.ConvTranspose2d( ngf * 2, ngf, 4, 2, 1, bias=False),
            nn.BatchNorm2d(ngf),
            nn.ReLU(True),
            # state size. ``(ngf) x 32 x 32``
            nn.ConvTranspose2d( ngf, nc, 4, 2, 1, bias=False),
            nn.Tanh()
            # state size. ``(nc) x 64 x 64``
        )

    def forward(self, input):
        return self.main(input)


# Create the generator
netG = Generator(ngpu).to(device)

# Handle multi-GPU if desired
if (device.type == 'cuda') and (ngpu > 1):
    netG = nn.DataParallel(netG, list(range(ngpu)))

# Apply the ``weights_init`` function to randomly initialize all weights
#  to ``mean=0``, ``stdev=0.02``.
netG.apply(weights_init)

# Print the model
print(netG)
```

```
Generator(
  (main): Sequential(
    (0): ConvTranspose2d(100, 224, kernel_size=(4, 4), stride=(1, 1), bias=False
    (1): BatchNorm2d(224, eps=1e-05, momentum=0.1, affine=True, track_running_st
    (2): ReLU(inplace=True)
    (3): ConvTranspose2d(224, 112, kernel_size=(4, 4), stride=(2, 2), padding=(1
    (4): BatchNorm2d(112, eps=1e-05, momentum=0.1, affine=True, track_running_st
```

```
    (5): ReLU(inplace=True)
    (6): ConvTranspose2d(112, 56, kernel_size=(4, 4), stride=(2, 2), padding=(1,
    (7): BatchNorm2d(56, eps=1e-05, momentum=0.1, affine=True, track_running_sta
    (8): ReLU(inplace=True)
    (9): ConvTranspose2d(56, 28, kernel_size=(4, 4), stride=(2, 2), padding=(1,
    (10): BatchNorm2d(28, eps=1e-05, momentum=0.1, affine=True, track_running_st
    (11): ReLU(inplace=True)
    (12): ConvTranspose2d(28, 1, kernel_size=(4, 4), stride=(2, 2), padding=(1,
    (13): Tanh()
  )
)
```

```
class Discriminator(nn.Module):
    def __init__(self, ngpu):
        super(Discriminator, self).__init__()
        self.ngpu = ngpu
        self.main = nn.Sequential(
            # input is ``(nc) x 64 x 64``
            nn.Conv2d(nc, ndf, 4, 2, 1, bias=False),
            nn.LeakyReLU(0.2, inplace=True),
            # state size. ``(ndf) x 32 x 32``
            nn.Conv2d(ndf, ndf * 2, 4, 2, 1, bias=False),
            nn.BatchNorm2d(ndf * 2),
            nn.LeakyReLU(0.2, inplace=True),
            # state size. ``(ndf*2) x 16 x 16``
            nn.Conv2d(ndf * 2, ndf * 4, 4, 2, 1, bias=False),
            nn.BatchNorm2d(ndf * 4),
            nn.LeakyReLU(0.2, inplace=True),
            # state size. ``(ndf*4) x 8 x 8``
            nn.Conv2d(ndf * 4, ndf * 8, 4, 2, 1, bias=False),
            nn.BatchNorm2d(ndf * 8),
            nn.LeakyReLU(0.2, inplace=True),
            # state size. ``(ndf*8) x 4 x 4``
            nn.Conv2d(ndf * 8, 1, 4, 1, 0, bias=False),
            nn.Sigmoid()
        )

    def forward(self, input):
        return self.main(input)


# Create the Discriminator
netD = Discriminator(ngpu).to(device)

# Handle multi-GPU if desired
if (device.type == 'cuda') and (ngpu > 1):
    netD = nn.DataParallel(netD, list(range(ngpu)))

# Apply the ``weights_init`` function to randomly initialize all weights
# like this: ``to mean=0, stdev=0.2``.
```

```
netD.apply(weights_init)

# Print the model
print(netD)
```

```
Discriminator(
  (main): Sequential(
    (0): Conv2d(1, 28, kernel_size=(4, 4), stride=(2, 2), padding=(1, 1), bias=F
    (1): LeakyReLU(negative_slope=0.2, inplace=True)
    (2): Conv2d(28, 56, kernel_size=(4, 4), stride=(2, 2), padding=(1, 1), bias=
    (3): BatchNorm2d(56, eps=1e-05, momentum=0.1, affine=True, track_running_sta
    (4): LeakyReLU(negative_slope=0.2, inplace=True)
    (5): Conv2d(56, 112, kernel_size=(4, 4), stride=(2, 2), padding=(1, 1), bias
    (6): BatchNorm2d(112, eps=1e-05, momentum=0.1, affine=True, track_running_st
    (7): LeakyReLU(negative_slope=0.2, inplace=True)
    (8): Conv2d(112, 224, kernel_size=(4, 4), stride=(2, 2), padding=(1, 1), bia
    (9): BatchNorm2d(224, eps=1e-05, momentum=0.1, affine=True, track_running_st
    (10): LeakyReLU(negative_slope=0.2, inplace=True)
    (11): Conv2d(224, 1, kernel_size=(4, 4), stride=(1, 1), bias=False)
    (12): Sigmoid()
  )
)
```

```
# Initialize the ``BCELoss`` function
criterion = nn.BCELoss()

# Create batch of latent vectors that we will use to visualize
#  the progression of the generator
fixed_noise = torch.randn(64, nz, 1, 1, device=device)

# Establish convention for real and fake labels during training
real_label = 1.
fake_label = 0.

# Setup Adam optimizers for both G and D
optimizerD = optim.Adam(netD.parameters(), lr=lr, betas=(beta1, 0.999))
optimizerG = optim.Adam(netG.parameters(), lr=lr, betas=(beta1, 0.999))


# Training Loop

# Lists to keep track of progress
img_list = []
G_losses = []
D_losses = []
iters = 0

%load_ext tensorboard


print("Starting Training Loop...")
```

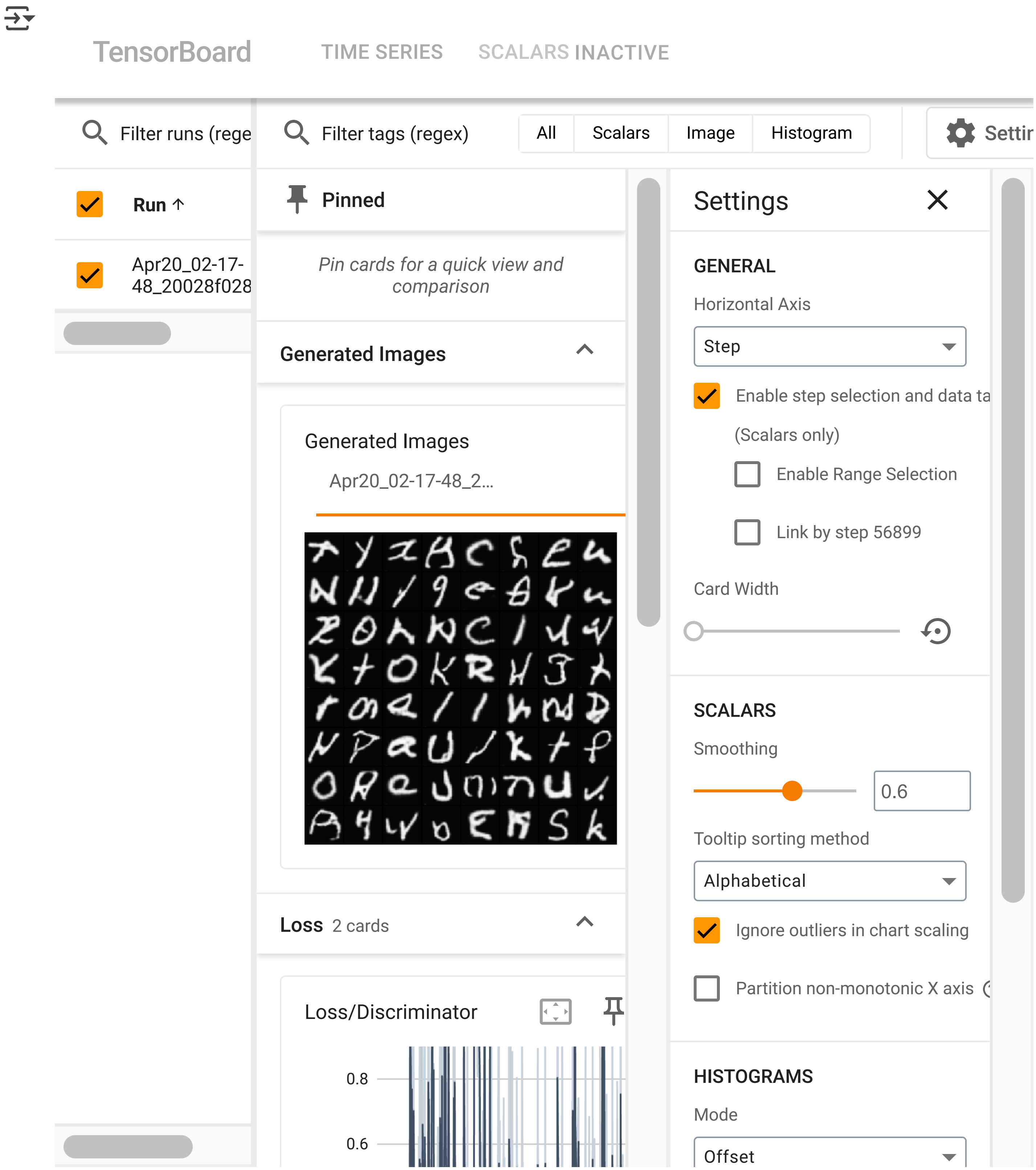
```
    # For each epoch
    for epoch in range(num_epochs):
        # For each batch in the dataloader
        for i, data in enumerate(train_dataloader, 0):

            ############################
            # (1) Update D network: maximize log(D(x)) + log(1 - D(G(z)))
            ###########################
            ## Train with all-real batch
            netD.zero_grad()
            # Format batch
            real_cpu = data[0].to(device)
            b_size = real_cpu.size(0)
            label = torch.full((b_size,), real_label, dtype=torch.float, device=device)
            # Forward pass real batch through D
            output = netD(real_cpu).view(-1)
            # Calculate loss on all-real batch
            errD_real = criterion(output, label)
            # Calculate gradients for D in backward pass
            errD_real.backward()
            D_x = output.mean().item()

            ## Train with all-fake batch
            # Generate batch of latent vectors
            noise = torch.randn(b_size, nz, 1, 1, device=device)
            # Generate fake image batch with G
            fake = netG(noise)
            label.fill_(fake_label)
            # Classify all fake batch with D
            output = netD(fake.detach()).view(-1)
            # Calculate D's loss on the all-fake batch
            errD_fake = criterion(output, label)
            # Calculate the gradients for this batch, accumulated (summed) with previous
            errD_fake.backward()
            D_G_z1 = output.mean().item()
            # Compute error of D as sum over the fake and the real batches
            errD = errD_real + errD_fake
            # Update D
            optimizerD.step()

            ############################
            # (2) Update G network: maximize log(D(G(z)))
            ###########################
            netG.zero_grad()
            label.fill_(real_label)  # fake labels are real for generator cost
            # Since we just updated D, perform another forward pass of all-fake batch th
            output = netD(fake).view(-1)
            # Calculate G's loss based on this output
            errG = criterion(output, label)
            # Calculate gradients for G
            errG.backward()
```

```
        D_G_z2 = output.mean().item()
        # Update G
        optimizerG.step()

        # Output training stats
        if i % 50 == 0:
            print('[%d/%d][%d/%d]\tLoss_D: %.4f\tLoss_G: %.4f\tD(x): %.4f\tD(G(z)):
                  % (epoch, num_epochs, i, len(train_dataloader),
                     errD.item(), errG.item(), D_x, D_G_z1, D_G_z2))

        # Save Losses for plotting later
        G_losses.append(errG.item())
        D_losses.append(errD.item())

        # Check how the generator is doing by saving G's output on fixed_noise
        if (iters % 500 == 0) or ((epoch == num_epochs-1) and (i == len(train_datalo
            with torch.no_grad():
                fake = netG(fixed_noise).detach().cpu()
            img_list.append(vutils.make_grid(fake, padding=2, normalize=True))



        # Log scalar values
        writer.add_scalar('Loss/Discriminator', errD.item(), global_step=iters)
        writer.add_scalar('Loss/Generator', errG.item(), global_step=iters)
        writer.add_scalar('Performance/D(x)', D_x, global_step=iters)
        writer.add_scalar('Performance/D(G(z1))', D_G_z1, global_step=iters)
        writer.add_scalar('Performance/D(G(z2))', D_G_z2, global_step=iters)

        # Log images generated by the GAN
        if iters % 500 == 0 or ((epoch == num_epochs-1) and (i == len(train_dataload
            with torch.no_grad():
                fake = netG(fixed_noise).detach().cpu()
            img_grid = vutils.make_grid(fake, padding=2, normalize=True)
            writer.add_image('Generated Images', img_grid, global_step=iters)

        iters += 1
```

```
The tensorboard extension is already loaded. To reload it, use:
  %reload_ext tensorboard
Starting Training Loop...
[0/50][0/1138]  Loss_D: 1.8856  Loss_G: 1.3766  D(x): 0.2853    D(G(z)): 0.37
[0/50][50/1138] Loss_D: 0.0734  Loss_G: 6.1430  D(x): 0.9705    D(G(z)): 0.04
[0/50][100/1138]        Loss_D: 0.0200  Loss_G: 7.0660  D(x): 0.9933    D(G(z
[0/50][150/1138]        Loss_D: 0.2827  Loss_G: 8.9739  D(x): 0.8583    D(G(z
[0/50][200/1138]        Loss_D: 0.0866  Loss_G: 5.5112  D(x): 0.9695    D(G(z
[0/50][250/1138]        Loss_D: 0.1766  Loss_G: 4.3192  D(x): 0.8790    D(G(z
[0/50][300/1138]        Loss_D: 0.2166  Loss_G: 3.4513  D(x): 0.8553    D(G(z
[0/50][350/1138]        Loss_D: 0.1073  Loss_G: 3.7556  D(x): 0.9503    D(G(z
[0/50][400/1138]        Loss_D: 0.1605  Loss_G: 3.8256  D(x): 0.9367    D(G(z
[0/50][450/1138]        Loss_D: 0.5088  Loss_G: 1.0462  D(x): 0.6536    D(G(z
[0/50][500/1138]        Loss_D: 0.1870  Loss_G: 3.3786  D(x): 0.9149    D(G(z
[0/50][550/1138]        Loss_D: 0.0776  Loss_G: 3.6571  D(x): 0.9545    D(G(z
[0/50][600/1138]        Loss_D: 0.3016  Loss_G: 2.5315  D(x): 0.8422    D(G(z
```

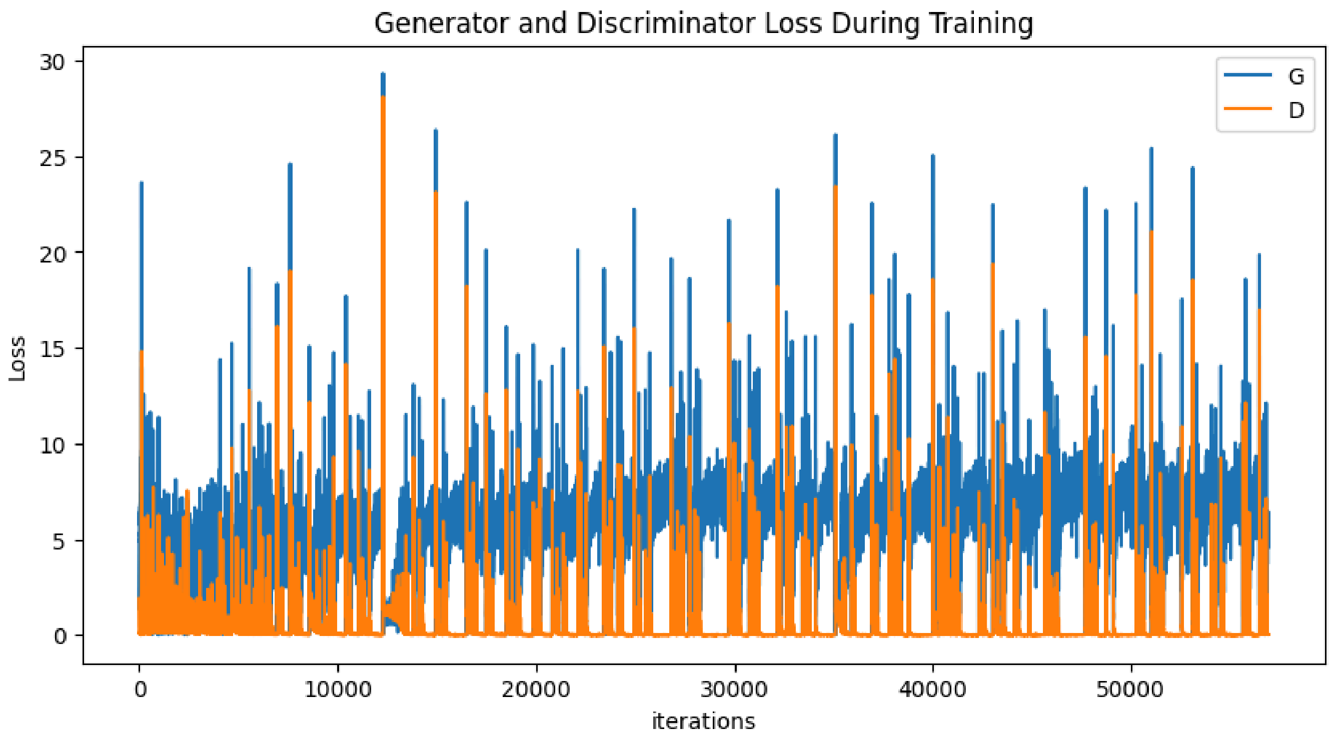
```
[0/50][650/1138]        Loss_D: 0.1148  Loss_G: 3.2126  D(x): 0.9519    D(G(z
[0/50][700/1138]        Loss_D: 0.2113  Loss_G: 3.0272  D(x): 0.9257    D(G(z
[0/50][750/1138]        Loss_D: 0.2317  Loss_G: 2.3149  D(x): 0.8929    D(G(z
[0/50][800/1138]        Loss_D: 0.4373  Loss_G: 3.2826  D(x): 0.9385    D(G(z
[0/50][850/1138]        Loss_D: 0.1860  Loss_G: 2.9189  D(x): 0.9311    D(G(z
[0/50][900/1138]        Loss_D: 0.1644  Loss_G: 3.1573  D(x): 0.9354    D(G(z
[0/50][950/1138]        Loss_D: 0.2125  Loss_G: 2.7380  D(x): 0.8871    D(G(z
[0/50][1000/1138]       Loss_D: 1.2272  Loss_G: 0.0016  D(x): 0.3684    D(G(z
[0/50][1050/1138]       Loss_D: 0.2499  Loss_G: 2.9425  D(x): 0.8987    D(G(z
[0/50][1100/1138]       Loss_D: 0.3494  Loss_G: 1.9796  D(x): 0.7954    D(G(z
[1/50][0/1138]  Loss_D: 0.1809  Loss_G: 3.2387  D(x): 0.9552    D(G(z)): 0.1
[1/50][50/1138] Loss_D: 0.4720  Loss_G: 1.7447  D(x): 0.7897    D(G(z)): 0.1
[1/50][100/1138]        Loss_D: 0.1476  Loss_G: 3.2569  D(x): 0.9431    D(G(z
[1/50][150/1138]        Loss_D: 0.6997  Loss_G: 7.5305  D(x): 0.9899    D(G(z
[1/50][200/1138]        Loss_D: 0.3494  Loss_G: 3.0406  D(x): 0.9276    D(G(z
[1/50][250/1138]        Loss_D: 0.5386  Loss_G: 2.3750  D(x): 0.7172    D(G(z
[1/50][300/1138]        Loss_D: 0.2177  Loss_G: 2.4332  D(x): 0.9372    D(G(z
[1/50][350/1138]        Loss_D: 1.4094  Loss_G: 0.0092  D(x): 0.3663    D(G(z
[1/50][400/1138]        Loss_D: 0.3489  Loss_G: 2.1725  D(x): 0.8232    D(G(z
[1/50][450/1138]        Loss_D: 0.2670  Loss_G: 1.8421  D(x): 0.8479    D(G(z
[1/50][500/1138]        Loss_D: 0.1895  Loss_G: 2.4232  D(x): 0.9109    D(G(z
[1/50][550/1138]        Loss_D: 0.1631  Loss_G: 3.4621  D(x): 0.8747    D(G(z
[1/50][600/1138]        Loss_D: 0.5265  Loss_G: 2.8027  D(x): 0.9022    D(G(z
[1/50][650/1138]        Loss_D: 0.3135  Loss_G: 2.4694  D(x): 0.8936    D(G(z
[1/50][700/1138]        Loss_D: 0.1190  Loss_G: 2.9570  D(x): 0.9550    D(G(z
[1/50][750/1138]        Loss_D: 1.1398  Loss_G: 1.0827  D(x): 0.5845    D(G(z
[1/50][800/1138]        Loss_D: 0.1905  Loss_G: 3.2324  D(x): 0.8867    D(G(z
[1/50][850/1138]        Loss_D: 0.1749  Loss_G: 4.0558  D(x): 0.9545    D(G(z
[1/50][900/1138]        Loss_D: 0.6403  Loss_G: 1.1777  D(x): 0.5743    D(G(z
[1/50][950/1138]        Loss_D: 1.0962  Loss_G: 1.2221  D(x): 0.7071    D(G(z
[1/50][1000/1138]       Loss_D: 0.4899  Loss_G: 3.9073  D(x): 0.9624    D(G(z
[1/50][1050/1138]       Loss_D: 1.1566  Loss_G: 0.7766  D(x): 0.4426    D(G(z
[1/50][1100/1138]       Loss_D: 0.2599  Loss_G: 2.0171  D(x): 0.8386    D(G(z
[2/50][0/1138]  Loss_D: 0.1813  Loss_G: 3.4713  D(x): 0.8491    D(G(z)): 0.0
[2/50][50/1138] Loss_D: 0.4838  Loss_G: 2.4311  D(x): 0.8045    D(G(z)): 0.1
[2/50][100/1138]        Loss_D: 0.2966  Loss_G: 2.0177  D(x): 0.8254    D(G(z
[2/50][150/1138]        Loss_D: 0.1374  Loss_G: 2.8968  D(x): 0.8971    D(G(z
[2/50][200/1138]        Loss_D: 0.2701  Loss_G: 3.1398  D(x): 0.8833    D(G(z
[2/50][250/1138]        Loss_D: 0.1641  Loss_G: 3.8185  D(x): 0.9850    D(G(z
[2/50][300/1138]        Loss_D: 1.0090  Loss_G: 0.6511  D(x): 0.4305    D(G(z
```

```
%tensorboard --logdir runs
```



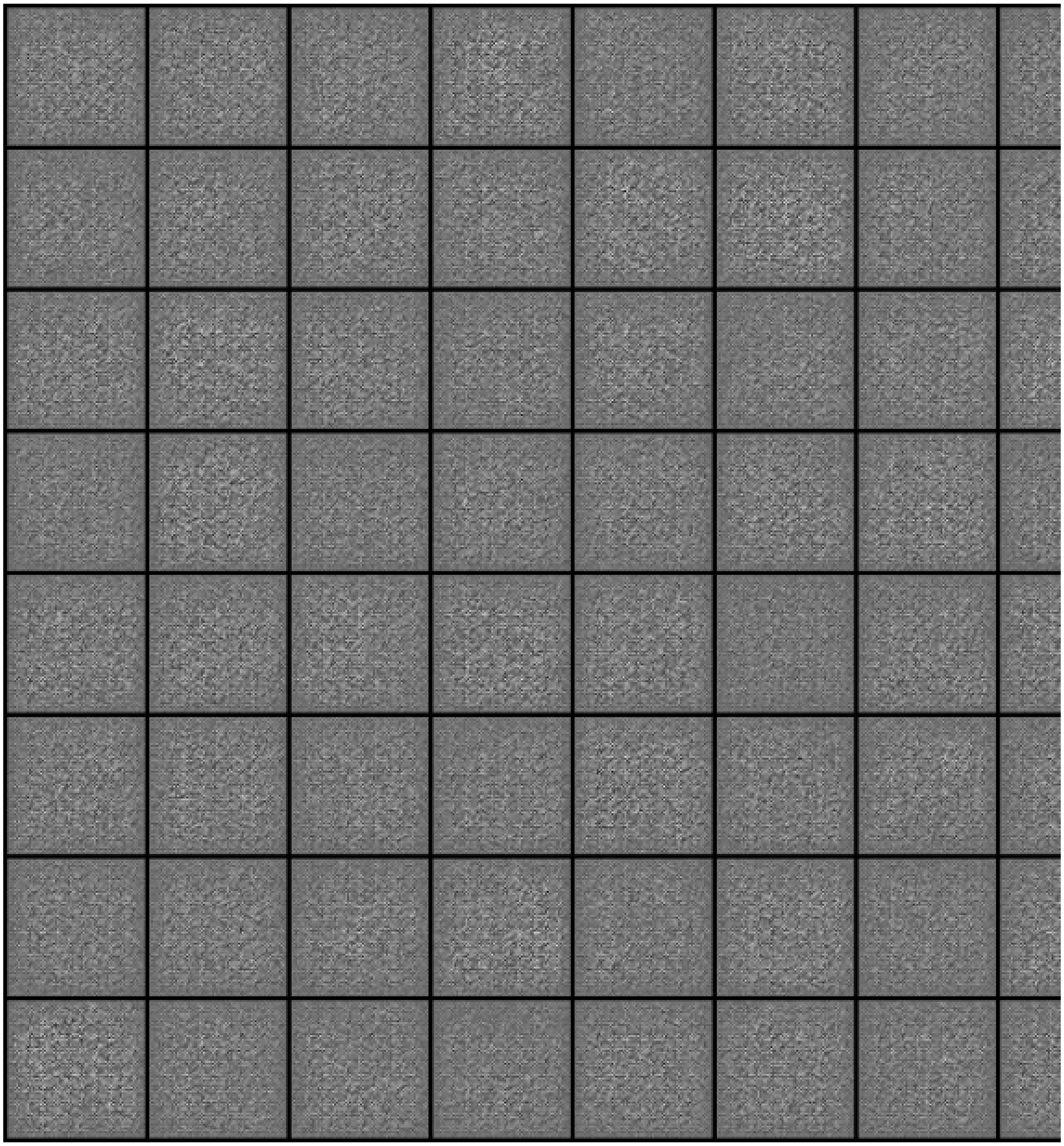

```
plt.figure(figsize=(10,5))
plt.title("Generator and Discriminator Loss During Training")
plt.plot(G_losses,label="G")
plt.plot(D_losses,label="D")
plt.xlabel("iterations")
plt.ylabel("Loss")
```

```
plt.legend()
plt.show()
```




```
fig = plt.figure(figsize=(8,8))
plt.axis("off")
ims = [[plt.imshow(np.transpose(i,(1,2,0)), animated=True)] for i in img_list]
ani = animation.ArtistAnimation(fig, ims, interval=1000, repeat_delay=1000, blit=Tru

HTML(ani.to_jshtml())
```

```
WARNING:matplotlib.animation:Animation size has reached 21053899 bytes, exceedin
```



Once Loop Reflect