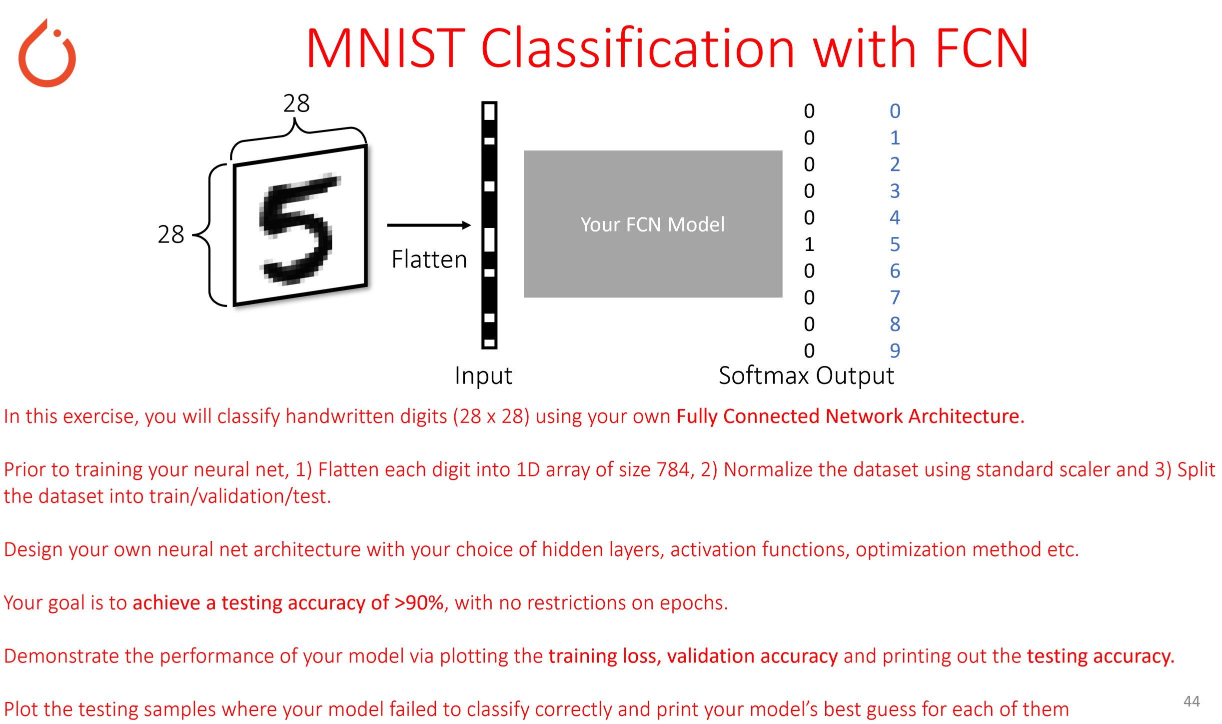
Lab 3 Report:

MNIST Classification with FCN

Name:

```
# Import necessary packages

%matplotlib inline

import matplotlib.pyplot as plt

import torch
import torchvision
import numpy as np


from IPython.display import Image # For displaying images in colab jupyter cell


Image('lab3_exercise.PNG', width = 1000)
```



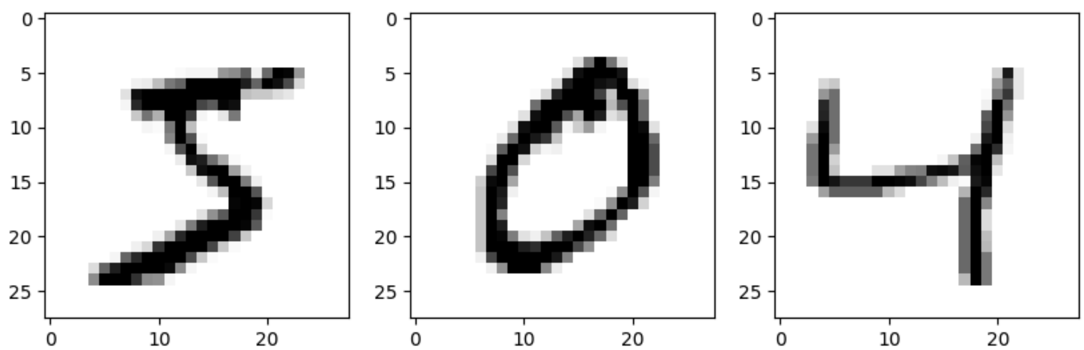
Prepare Data

```
# Load MNIST Dataset in Numpy

# 1000 training samples where each sample feature is a greyscale image with shape (28, 28)
# 1000 training targets where each target is an integer indicating the true digit
mnist_train_features = np.load('mnist_train_features.npy')
mnist_train_targets = np.load('mnist_train_targets.npy')

# 100 testing samples + targets
mnist_test_features = np.load('mnist_test_features.npy')
mnist_test_targets = np.load('mnist_test_targets.npy')

# Print the dimensions of training sample features/targets
print(mnist_train_features.shape, mnist_train_targets.shape)
# Print the dimensions of testing sample features/targets
print(mnist_test_features.shape, mnist_test_targets.shape)
```

```
(1000, 28, 28) (1000,)
(100, 28, 28) (100,)
```

```
# Let's visualize some training samples

plt.figure(figsize = (10, 10))

plt.subplot(1,3,1)
plt.imshow(mnist_train_features[0], cmap = 'Greys')

plt.subplot(1,3,2)
plt.imshow(mnist_train_features[1], cmap = 'Greys')

plt.subplot(1,3,3)
plt.imshow(mnist_train_features[2], cmap = 'Greys')
```

```
<matplotlib.image.AxesImage at 0x32769d3a0>
```




```
# Reshape features via flattening the images
# This refers to reshape each sample from a 2d array to a 1d array.
# hint: np.reshape() function could be useful here\
mnist_test_features = mnist_test_features.reshape(100, 784)
mnist_train_features = mnist_train_features.reshape(1000, 784)
# mnist_train_features = np.reshape(1000, -1)
# mnist_test_features = np.reshape(100, -1)

print(mnist_train_features.shape, mnist_test_features.shape)
```

```
(1000, 784) (100, 784)
```

```
from sklearn.preprocessing import StandardScaler
# Scale the dataset according to standard scaling
scaler = StandardScaler()
mnist_train_features = scaler.fit_transform(mnist_train_features)
mnist_test_features = scaler.fit_transform(mnist_test_features)
```

```
from sklearn.model_selection import train_test_split
# Split training dataset into Train (90%), Validation (10%)
mnist_train_features, mnist_validation_features, mnist_train_targets, mnist_validation_targets = train_test_split(mnist_train_fe
```

Define Model

```
class mnistClassification(torch.nn.Module):

    def __init__(self, input_dim, output_dim, hidden_dim, dropout):

        super(mnistClassification, self).__init__()
        # use 3 linear layers and one dropout layer to mitigate overfitting
        self.linear1 = torch.nn.Linear(input_dim, hidden_dim)
        self.linear2 = torch.nn.Linear(hidden_dim, hidden_dim)
        self.linear3 = torch.nn.Linear(hidden_dim, output_dim)
        self.drop1 = torch.nn.Dropout(p = dropout)

    def forward(self, x):
        # use relu and sigmoid activation functions, for this problem provides no noticeable difference
        out = torch.nn.functional.relu(self.linear1(x))
        out = self.drop1(out)
        out = torch.nn.functional.sigmoid(self.linear2(out))
        out = self.drop1(out)
        out = self.linear3(out)
        # not using softmax here because CrossEntropyLoss expects raw values

        return out
```

Define Hyperparameters

```
# Initialize our neural network model with input and output dimensions
model = mnistClassification(input_dim=784, output_dim=10, hidden_dim=256, dropout=0.4)

# Define the learning rate and epoch
learning_rate = 0.0003
epochs = 300
batchsize = 32

# Define loss function and optimizer
loss_func = torch.nn.CrossEntropyLoss()
optimizer = torch.optim.Adam(model.parameters(), lr = learning_rate, weight_decay=1e-4)

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

# Run this line if PyTorch GPU is available
if torch.cuda.is_available():
    model.cuda()

model
```

```
mnistClassification(
  (linear1): Linear(in_features=784, out_features=256, bias=True)
  (linear2): Linear(in_features=256, out_features=256, bias=True)
  (linear3): Linear(in_features=256, out_features=10, bias=True)
  (drop1): Dropout(p=0.4, inplace=False)
)
```

Identify Tracked Values

```
# Placeholders for training loss and validation accuracy during training
# Training loss should be tracked for each iteration (1 iteration -> single forward pass to the network)
# Validation accuracy should be evaluated every 'Epoch' (1 epoch -> full training dataset)
# If using batch gradient, 1 iteration = 1 epoch

train_loss_list = []
validation_accuracy_list = []
```

Train Model

```
import tqdm

# Convert the training, validation, testing dataset (NumPy arrays) into torch tensors
mnist_test_features = torch.tensor(mnist_test_features, dtype=torch.float32)
mnist_test_targets = torch.tensor(mnist_test_targets, dtype=torch.long)
mnist_train_features = torch.tensor(mnist_train_features, dtype=torch.float32)
mnist_train_targets = torch.tensor(mnist_train_targets, dtype=torch.long)
```

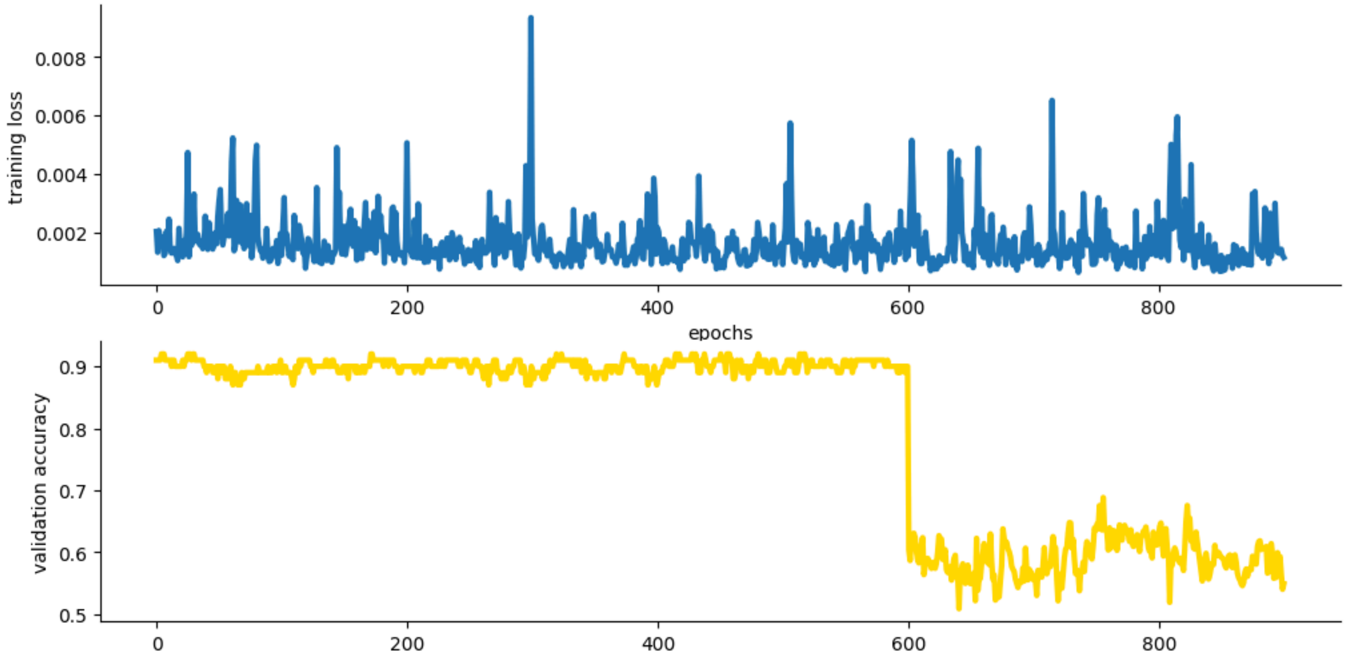
```
mnist_validation_features = torch.tensor(mnist_validation_features, dtype=torch.float32)
mnist_validation_targets = torch.tensor(mnist_validation_targets, dtype=torch.long)

# Create TensorDatasets (feature, target)
mnist_train = torch.utils.data.TensorDataset(mnist_train_features, mnist_train_targets)
mnist_validation = torch.utils.data.TensorDataset(mnist_validation_features, mnist_validation_targets)
mnist_test = torch.utils.data.TensorDataset(mnist_test_features, mnist_test_targets)

# Create data loaders, using batches for more efficient training
train_loader = torch.utils.data.DataLoader(mnist_train, batch_size=batchsize, shuffle=True)
val_loader = torch.utils.data.DataLoader(mnist_validation, batch_size=batchsize)
test_loader = torch.utils.data.DataLoader(mnist_test, batch_size=batchsize)


# Training Loop ---------------------------------------------------------------------------------------

for epoch in tqdm.trange(epochs):
    model.train()
    train_loss = 0
    correct = 0
    total = 0

    # each batch runs the inputs through the model and minimizes loss function
    for inputs, targets in train_loader:
        inputs, targets = inputs.to(device), targets.to(device)

        optimizer.zero_grad()
        outputs = model(inputs)

        loss = loss_func(outputs, targets)
        loss.backward()
        optimizer.step()

        train_loss += loss.item() * inputs.size(0)
        predicted = torch.argmax(outputs, 1)
        correct += (predicted == targets).sum().item()
        total += targets.size(0)

    avg_train_loss = train_loss / total
    accuracy = correct / total
    train_loss_list.append(avg_train_loss)

    # Validation phase
    model.eval()
    val_loss = 0
    val_correct = 0
    val_total = 0

    # validate the training using no_grad to test model (doesn't include dropout)
    with torch.no_grad():
        for val_inputs, val_targets in val_loader:
            val_inputs, val_targets = val_inputs.to(device), val_targets.to(device)

            val_outputs = model(val_inputs)
            loss = loss_func(val_outputs, val_targets)

            val_loss += loss.item() * val_inputs.size(0)
            val_predicted = torch.argmax(val_outputs, dim=1)
            val_correct += (val_predicted == val_targets).sum().item()
            val_total += val_targets.size(0)

    avg_val_loss = val_loss / val_total
    val_accuracy = val_correct / val_total
    validation_accuracy_list.append(avg_val_loss)

    print(f"Epoch {epoch+1}/{epochs} - "
          f"Train Loss: {avg_train_loss:.4f}, Accuracy: {accuracy:.4f} - "
          f"Val Loss: {avg_val_loss:.4f}, Val Accuracy: {val_accuracy:.4f}")
```

```
 18%|██         | 53/300 [00:01<00:08, 29.78it/s]Epoch 47/300 - Train Loss: 0.0011, Accuracy: 1.0000 - Val Loss: 0.5657, Va
```

```
Epoch 48/300 - Train Loss: 0.0009, Accuracy: 1.0000 - Val Loss: 0.5794, Val Accuracy: 0.9000
Epoch 49/300 - Train Loss: 0.0011, Accuracy: 1.0000 - Val Loss: 0.5594, Val Accuracy: 0.9000
Epoch 50/300 - Train Loss: 0.0013, Accuracy: 1.0000 - Val Loss: 0.5496, Val Accuracy: 0.9000
Epoch 51/300 - Train Loss: 0.0011, Accuracy: 1.0000 - Val Loss: 0.5598, Val Accuracy: 0.9000
Epoch 52/300 - Train Loss: 0.0008, Accuracy: 1.0000 - Val Loss: 0.5610, Val Accuracy: 0.9000
Epoch 53/300 - Train Loss: 0.0021, Accuracy: 1.0000 - Val Loss: 0.5671, Val Accuracy: 0.9000
 20%|██        | 59/300 [00:02<00:08, 28.41it/s]Epoch 54/300 - Train Loss: 0.0016, Accuracy: 1.0000 - Val Loss: 0.5220, Va
Epoch 55/300 - Train Loss: 0.0027, Accuracy: 0.9989 - Val Loss: 0.6228, Val Accuracy: 0.8900
Epoch 56/300 - Train Loss: 0.0049, Accuracy: 0.9989 - Val Loss: 0.5387, Val Accuracy: 0.9100
Epoch 57/300 - Train Loss: 0.0016, Accuracy: 1.0000 - Val Loss: 0.5572, Val Accuracy: 0.9100
Epoch 58/300 - Train Loss: 0.0013, Accuracy: 1.0000 - Val Loss: 0.5578, Val Accuracy: 0.9100
Epoch 59/300 - Train Loss: 0.0028, Accuracy: 1.0000 - Val Loss: 0.5820, Val Accuracy: 0.9000
 22%|██        | 65/300 [00:02<00:08, 28.61it/s]Epoch 60/300 - Train Loss: 0.0012, Accuracy: 1.0000 - Val Loss: 0.5988, Va
Epoch 61/300 - Train Loss: 0.0011, Accuracy: 1.0000 - Val Loss: 0.6142, Val Accuracy: 0.9000
Epoch 62/300 - Train Loss: 0.0019, Accuracy: 1.0000 - Val Loss: 0.5907, Val Accuracy: 0.8900
Epoch 63/300 - Train Loss: 0.0012, Accuracy: 1.0000 - Val Loss: 0.5845, Val Accuracy: 0.9100
Epoch 64/300 - Train Loss: 0.0012, Accuracy: 1.0000 - Val Loss: 0.5903, Val Accuracy: 0.9000
Epoch 65/300 - Train Loss: 0.0009, Accuracy: 1.0000 - Val Loss: 0.5805, Val Accuracy: 0.9100
 24%|██        | 71/300 [00:02<00:08, 28.03it/s]Epoch 66/300 - Train Loss: 0.0025, Accuracy: 1.0000 - Val Loss: 0.6299, Va
Epoch 67/300 - Train Loss: 0.0026, Accuracy: 1.0000 - Val Loss: 0.5862, Val Accuracy: 0.9000
Epoch 68/300 - Train Loss: 0.0013, Accuracy: 1.0000 - Val Loss: 0.5568, Val Accuracy: 0.9000
Epoch 69/300 - Train Loss: 0.0011, Accuracy: 1.0000 - Val Loss: 0.5637, Val Accuracy: 0.9000
Epoch 70/300 - Train Loss: 0.0010, Accuracy: 1.0000 - Val Loss: 0.5230, Val Accuracy: 0.9000
Epoch 71/300 - Train Loss: 0.0012, Accuracy: 1.0000 - Val Loss: 0.5340, Val Accuracy: 0.8900
 26%|██        | 77/300 [00:02<00:07, 28.58it/s]Epoch 72/300 - Train Loss: 0.0009, Accuracy: 1.0000 - Val Loss: 0.5270, Va
Epoch 73/300 - Train Loss: 0.0016, Accuracy: 1.0000 - Val Loss: 0.5283, Val Accuracy: 0.9000
Epoch 74/300 - Train Loss: 0.0019, Accuracy: 1.0000 - Val Loss: 0.5541, Val Accuracy: 0.9000
Epoch 75/300 - Train Loss: 0.0013, Accuracy: 1.0000 - Val Loss: 0.5787, Val Accuracy: 0.8900
Epoch 76/300 - Train Loss: 0.0022, Accuracy: 1.0000 - Val Loss: 0.6379, Val Accuracy: 0.8700
Epoch 77/300 - Train Loss: 0.0017, Accuracy: 1.0000 - Val Loss: 0.6225, Val Accuracy: 0.8700
 28%|██▊       | 84/300 [00:02<00:07, 29.59it/s]Epoch 78/300 - Train Loss: 0.0009, Accuracy: 1.0000 - Val Loss: 0.6166, Va
Epoch 79/300 - Train Loss: 0.0011, Accuracy: 1.0000 - Val Loss: 0.6160, Val Accuracy: 0.8800
Epoch 80/300 - Train Loss: 0.0017, Accuracy: 1.0000 - Val Loss: 0.6058, Val Accuracy: 0.9100
Epoch 81/300 - Train Loss: 0.0012, Accuracy: 1.0000 - Val Loss: 0.6012, Val Accuracy: 0.9100
Epoch 82/300 - Train Loss: 0.0012, Accuracy: 1.0000 - Val Loss: 0.5932, Val Accuracy: 0.9100
Epoch 83/300 - Train Loss: 0.0012, Accuracy: 1.0000 - Val Loss: 0.5774, Val Accuracy: 0.9100
Epoch 84/300 - Train Loss: 0.0009, Accuracy: 1.0000 - Val Loss: 0.5726, Val Accuracy: 0.9000
 29%|██▉       | 88/300 [00:03<00:07, 30.22it/s]Epoch 85/300 - Train Loss: 0.0019, Accuracy: 0.9989 - Val Loss: 0.5692, Va
Epoch 86/300 - Train Loss: 0.0010, Accuracy: 1.0000 - Val Loss: 0.5567, Val Accuracy: 0.9100
Epoch 87/300 - Train Loss: 0.0007, Accuracy: 1.0000 - Val Loss: 0.5497, Val Accuracy: 0.9200
Epoch 88/300 - Train Loss: 0.0011, Accuracy: 1.0000 - Val Loss: 0.5432, Val Accuracy: 0.9100
Epoch 89/300 - Train Loss: 0.0012, Accuracy: 1.0000 - Val Loss: 0.5532, Val Accuracy: 0.9100
Epoch 90/300 - Train Loss: 0.0012, Accuracy: 1.0000 - Val Loss: 0.5510, Val Accuracy: 0.9200
 32%|███▏      | 95/300 [00:03<00:07, 27.11it/s]Epoch 91/300 - Train Loss: 0.0010, Accuracy: 1.0000 - Val Loss: 0.5522, Va
Epoch 92/300 - Train Loss: 0.0018, Accuracy: 0.9989 - Val Loss: 0.5749, Val Accuracy: 0.9000
Epoch 93/300 - Train Loss: 0.0020, Accuracy: 1.0000 - Val Loss: 0.5538, Val Accuracy: 0.9000
Epoch 94/300 - Train Loss: 0.0016, Accuracy: 1.0000 - Val Loss: 0.6068, Val Accuracy: 0.8900
Epoch 95/300 - Train Loss: 0.0018, Accuracy: 1.0000 - Val Loss: 0.5694, Val Accuracy: 0.8900
 34%|███▎      | 101/300 [00:03<00:07, 28.13it/s]Epoch 96/300 - Train Loss: 0.0011, Accuracy: 1.0000 - Val Loss: 0.5556, V
Epoch 97/300 - Train Loss: 0.0029, Accuracy: 0.9989 - Val Loss: 0.5727, Val Accuracy: 0.9000
```

Visualize and Evaluate Model

```python
# Import seaborn for prettier plots

import seaborn as sns


# Visualize training loss

plt.figure(figsize = (12, 6))

# Visualize training loss with respect to iterations (1 iteration -> single batch)
plt.subplot(2, 1, 1)
plt.plot(train_loss_list, linewidth = 3)
plt.ylabel("training loss")
plt.xlabel("epochs")
sns.despine()

# Visualize validation accuracy with respect to epochs
plt.subplot(2, 1, 2)
plt.plot(validation_accuracy_list, linewidth = 3, color = 'gold')
plt.ylabel("validation accuracy")
sns.despine()
```




```
# Compute the testing accuracy

# Set model to evaluation mode
model.eval()

# Track test accuracy
test_correct = 0
test_total = 0
test_loss = 0.0

with torch.no_grad():
    for inputs, targets in test_loader:
        inputs, targets = inputs.to(device), targets.to(device)

        outputs = model(inputs)
        loss = loss_func(outputs, targets)

        test_loss += loss.item() * inputs.size(0)  # sum loss over batch
        _, predicted = torch.max(outputs, 1)
        test_correct += (predicted == targets).sum().item()
        test_total += targets.size(0)

# Final metrics
avg_test_loss = test_loss / test_total
test_accuracy = test_correct / test_total

print(f" Test Loss: {avg_test_loss:.4f}")
print(f" Test Accuracy: {test_accuracy:.4f}")
```

```
 Test Loss: 0.3062
 Test Accuracy: 0.9300
```

```
# Plot 5 incorrectly classified testing samples and print the model predictions for each of them
# You can use np.reshape() to convert flattened 1D array back to 2D array

# Make sure model is in evaluation mode
model.eval()

misclassified = []

with torch.no_grad():
    for inputs, targets in test_loader:
        inputs, targets = inputs.to(device), targets.to(device)
        outputs = model(inputs)
        _, preds = torch.max(outputs, 1)

        # Get indices where prediction != ground truth
```

```
        mismatch = (preds != targets).nonzero(as_tuple=True)[0]

        for idx in mismatch:
            img = inputs[idx].cpu().numpy().reshape(28, 28)
            true_label = targets[idx].item()
            pred_label = preds[idx].item()
            misclassified.append((img, true_label, pred_label))

        if len(misclassified) >= 5:
            break  # Stop after 5 misclassified samples

# Plotting
plt.figure(figsize=(10, 4))
for i, (img, true, pred) in enumerate(misclassified[:5]):
    plt.subplot(1, 5, i + 1)
    plt.imshow(img, cmap="gray")
    plt.title(f"Pred: {pred}\nTrue: {true}")
    plt.axis("off")
plt.tight_layout()
plt.show()
```