Lab 3 Report:

MNIST Classification with FCN

Name: Forest Tschirhart

In [172…

```
# Import necessary packages

%matplotlib inline

import matplotlib.pyplot as plt
import tqdm
import torch
import torchvision
import numpy as np
from sklearn.preprocessing import StandardScaler, MinMaxScaler
```

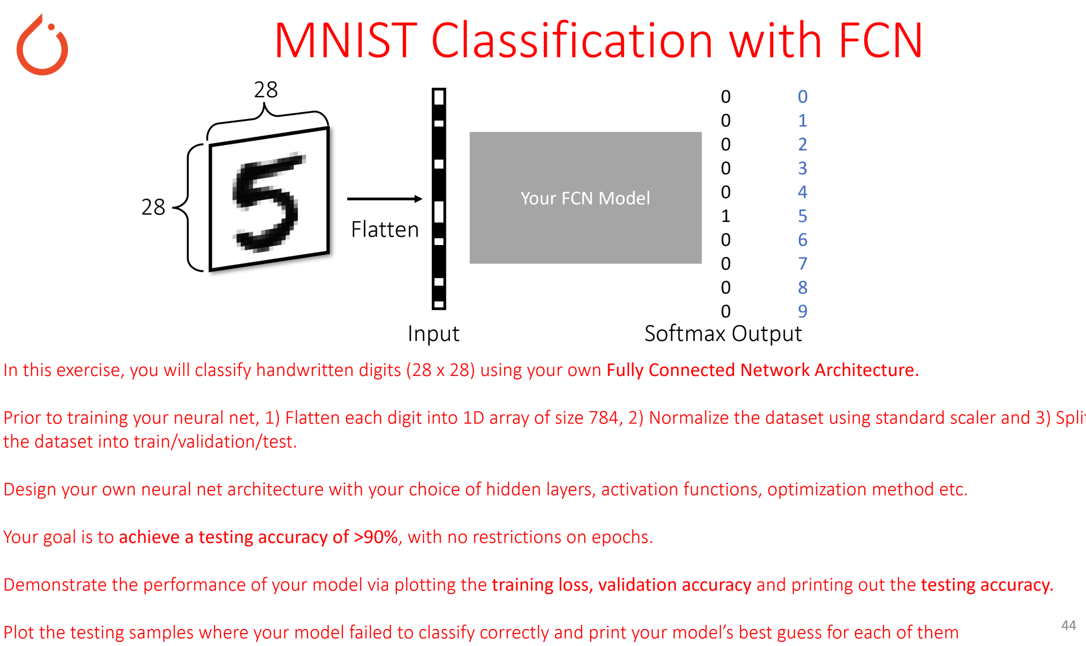
In [173…

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

In [174…

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

Out[174…



Prepare Data

In [175…

```
# Load MNIST Dataset in Numpy

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

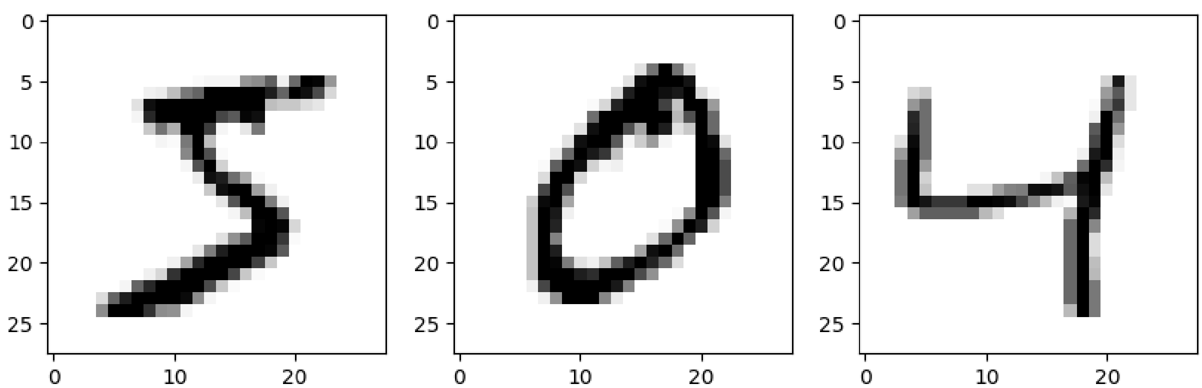
```
mnist_train_targets = np.load('mnist_train_targets.npy')

# 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)

print(mnist_train_targets[:10]) # print a sample
```

```
(1000, 28, 28) (1000,)
(100, 28, 28) (100,)
[5 0 4 1 9 2 1 3 1 4]
```

In [176…

```
# 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')
```

Out[176… <matplotlib.image.AxesImage at 0x724f0f77a540>



In [177…

```
# 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_train_features = np.reshape(mnist_train_features, (1000, 784), copy=Fa
mnist_test_features = np.reshape(mnist_test_features, (100, 784), copy=False

print(mnist_train_features.shape, mnist_test_features.shape) # check the dim
```

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

In [178…

```
# Scale the dataset according to standard scaling
def scaling(train, test, type = 'standard'): # scale using different methods
    if type == 'standard':
        scaler = StandardScaler()
        sc_train = scaler.fit_transform(train)
        sc_test = scaler.transform(test)
    elif type == 'minmax':
        scaler = MinMaxScaler()
        sc_train = scaler.fit_transform(train)
        sc_test = scaler.transform(test)
    else:
        raise ValueError("Invalid scaling type. Choose 'standard' or 'minmax
    return sc_train, sc_test

mnist_train_features, mnist_test_features = scaling(mnist_train_features, mn
                                                    type = 'standard')
```

In [179…

```
# Split training dataset into Train (90%), Validation (10%)

valnum = int(0.1 * mnist_train_features.shape[0]) # 10% of training samples

mnist_validation_features = mnist_train_features[:valnum]
mnist_validation_targets = mnist_train_targets[:valnum]

mnist_train_features = mnist_train_features[valnum:]
mnist_train_targets = mnist_train_targets[valnum:]
```

Define Model

In [180…

```
class mnistClassification(torch.nn.Module):
    # added some arguments to make the model more flexible so that I could
    #  quickly experiment with different architectures
    def __init__(self, input_dim, output_dim, layers=[128],
                 dropout_prob = 0.3):

        super(mnistClassification, self).__init__()
        self.num_layers = len(layers)
        # need different layer sizes for different number of layers so
        # that they are compatible
        if self.num_layers == 1:
            self.fc_in = torch.nn.Linear(input_dim, layers[0])
            self.fc_out = torch.nn.Linear(layers[0], output_dim)
            self.bn1 = torch.nn.BatchNorm1d(layers[0])
            self.drop1 = torch.nn.Dropout(dropout_prob)
        elif self.num_layers == 2:
            self.fc_in = torch.nn.Linear(input_dim, layers[0])
            self.fc1 = torch.nn.Linear(layers[0], layers[1])
            self.fc_out = torch.nn.Linear(layers[1], output_dim)
            self.bn1 = torch.nn.BatchNorm1d(layers[0])
            self.bn2 = torch.nn.BatchNorm1d(layers[1])
            self.drop1 = torch.nn.Dropout(dropout_prob)
            self.drop2 = torch.nn.Dropout(dropout_prob)
        elif self.num_layers == 3:
            self.fc_in = torch.nn.Linear(input_dim, layers[0])
```

```
            self.fc1 = torch.nn.Linear(layers[0], layers[1])
            self.fc2 = torch.nn.Linear(layers[1], layers[2])
            self.fc_out = torch.nn.Linear(layers[2], output_dim)
            self.bn1 = torch.nn.BatchNorm1d(layers[0])
            self.bn2 = torch.nn.BatchNorm1d(layers[1])
            self.bn3 = torch.nn.BatchNorm1d(layers[2])
            self.drop1 = torch.nn.Dropout(dropout_prob)
            self.drop2 = torch.nn.Dropout(dropout_prob)
            self.drop3 = torch.nn.Dropout(dropout_prob)
        else:
            raise ValueError('Invalid depth. Max depth is 3')
        # can choose activation function here
        self.activation = torch.nn.ReLU()

    # Foward pass defined as usual, put all in one line for smaller code
    # cell but the structure is as follows:
    # 1. input layer
    # 2. batch normalization
    # 3. dropout
    # 4. activation function
    # repeat for al layers
    # 5. output layer doesnt get batch normalization, dropout or
    # activation function applied

    # Note: We do not apply softmax to the output layer because
    # pytorch's CrossEntropyLoss function already applies softmax
    # internally and expects raw logits as input.
    # If we applied softmax here it would get applied twice,
    # resulting in instabilities and slower convergence.
    def forward(self, x):
        if self.num_layers == 1:
            out = self.fc_out(self.activation(self.drop1(self.bn1(
                self.fc_in(x)))))
        elif self.num_layers == 2:
            out = self.fc_out(self.activation(self.drop2(self.bn2(
                self.fc1(self.activation(self.drop1(self.bn1(
                    self.fc_in(x)))))))))
        elif self.num_layers == 3:
            out = self.fc_out(self.activation(self.drop3(self.bn3(
                self.fc2(self.activation(self.drop2(self.bn2(
                    self.fc1(self.activation(self.drop1(self.bn1(self.fc_in(
        else:
            raise ValueError('Invalid depth. Max depth is 3')
        return out
```

In [181…

```
# trying out initialization to improve performance
def initialize_weights_relu(m, activation='relu'):
    if isinstance(m, torch.nn.Linear):  # Apply only to Linear layers
        # He initialization is best for ReLU
        if activation == 'relu':
            torch.nn.init.kaiming_normal_(m.weight, nonlinearity='relu')
        elif activation == 'tanh':
            torch.nn.init.kaiming_normal_(m.weight, nonlinearity='tanh')
        # Xavier initialization is best for sigmoid
        elif activation == 'sigmoid':
            torch.nn.init.xavier_normal_(m.weight, gain=torch.nn.init.calcul
```

```
        if m.bias is not None:
            torch.nn.init.zeros_(m.bias)  # Just biases initialized to zero
```

Define Hyperparameters

In [182…

```
# Initialize our neural network model with input and output dimensions
# using funnel architecture here
model = mnistClassification(input_dim=784, output_dim=10,
                            layers=[512, 128, 64], dropout_prob=0.3)

model.apply(lambda m: initialize_weights_relu(m, activation='relu')) # initi

# Define the learning rate and epoch
learning_rate = 0.003
epochs = 150
print(f'Learning rate: {learning_rate}')
print(f'Epochs: {epochs}')

# Define the L2 regularization lambda parameter
reg_lambda = 0.06
print(f'Regularization lambda: {reg_lambda}')

# Define the mini batch size
batch_pwr = 5
batchsize = 2**batch_pwr
print(f'Batch size: {batchsize}')

# Define loss function and optimizer
# Optimizer is SGD with momentum 0.7, and learning rate and weight decay as
# defined above. Wanted to try without using Adam for developing better
# understanding of manual tuning
loss_func = torch.nn.CrossEntropyLoss()
optimizer = torch.optim.SGD(model.parameters(), lr=learning_rate,
                            weight_decay=reg_lambda, momentum=0.7)

# Run this line if you have PyTorch GPU version
if torch.cuda.is_available():
    model.cuda()

model
```

```
Learning rate: 0.003
Epochs: 150
Regularization lambda: 0.06
Batch size: 32
```

Out[182…

```
mnistClassification(
  (fc_in): Linear(in_features=784, out_features=512, bias=True)
  (fc1): Linear(in_features=512, out_features=128, bias=True)
  (fc2): Linear(in_features=128, out_features=64, bias=True)
  (fc_out): Linear(in_features=64, out_features=10, bias=True)
  (bn1): BatchNorm1d(512, eps=1e-05, momentum=0.1, affine=True, track_runni
ng_stats=True)
  (bn2): BatchNorm1d(128, eps=1e-05, momentum=0.1, affine=True, track_runni
ng_stats=True)
  (bn3): BatchNorm1d(64, eps=1e-05, momentum=0.1, affine=True, track_runnin
g_stats=True)
  (drop1): Dropout(p=0.3, inplace=False)
  (drop2): Dropout(p=0.3, inplace=False)
  (drop3): Dropout(p=0.3, inplace=False)
  (activation): ReLU()
)
```

Identify Tracked Values

In [183…

```python
# 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 = []
# also want to look at validation loss to check for overfitting
validation_loss_list = []
```

Train Model

In [184…

```python
# Convert the training, validation, testing dataset (NumPy arrays) into torc

train_inputs = torch.from_numpy(mnist_train_features).float()
train_targets = torch.from_numpy(mnist_train_targets).long()
#idk why we have to use long here but it works

validation_inputs = torch.from_numpy(mnist_validation_features).float()
validation_targets = torch.from_numpy(mnist_validation_targets).long()

testing_inputs = torch.from_numpy(mnist_test_features).float()
testing_targets = torch.from_numpy(mnist_test_targets).long()

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

for epoch in tqdm.trange(epochs):
    # get random indices to shuffle the training data
    indices = torch.randperm(train_inputs.size(0))

    # Shuffle both tensors while preserving the correspondence between
    # inputs and targets
```

```
    temp_train_inputs = train_inputs[indices]
    temp_train_targets = train_targets[indices]

    # Iterate over mini-batches of training data
    for i in range(0, len(train_inputs), batchsize):
         # indexing doesnt error here cause exclusivity, we dont waste any
         # remainder of data here cause of the way python handles slicing
        mb_inputs = temp_train_inputs[i:i+batchsize]
        mb_targets = temp_train_targets[i:i+batchsize]

        optimizer.zero_grad()

        train_outputs = model(mb_inputs)

        loss = loss_func(train_outputs, mb_targets)

        train_loss_list.append(loss.item())

        loss.backward()

        optimizer.step()


    # Compute Validation Accuracy -----------------------------------------
    model.eval() # turn off dropout and batch normalization
    with torch.no_grad(): # turn off gradient tracking

        validation_outputs = model(validation_inputs)

        val_loss = loss_func(validation_outputs, validation_targets)
        # append validation loss for each epoch, not minibatch
        validation_loss_list.append(val_loss.item())

        predicted = torch.argmax(validation_outputs, dim=1)
        # dont even need softmax here cause we aren't feeding it to
        # cross entropy. Raw logit indices correspond to labels 1-10

        # Compute the accuracy
        correct = (predicted == validation_targets).sum().item()
        # boolean logic returns true false array
        # since True=1 False=0 we can just sum
        accuracy = correct / validation_targets.size(0)
        # divide by number of samples to get accuracy

        validation_accuracy_list.append(accuracy) # save accuracy for plots
    model.train() # turn on dropout and batch normalization for next iterati
```

```
100%|██████████| 150/150 [00:07<00:00, 20.25it/s]
```

In [185…

```
# Figuring out array sizes so that I can plot the loss per minibatch and
# the validation loss per epoch superimposed on same graph
datasize = len(train_inputs)
print(f'Training data size: {datasize} \nBatchsize: {batchsize} \nEpochs: {e
#print(len(train_inputs)/batchsize)

# +1 accounts for remainder slice iteration in each minibatch
```

```
data_divs = (datasize // batchsize) + 1
print(f'Datasize/batchsize, plus remainder slice --> # of training data divi
step_range = np.arange(0, data_divs*epochs)
# define "xaxis" for minibatch training loss
epoch_spacing_range = np.linspace(0, epochs*data_divs, epochs+1)[1:]
# define "xaxis" for validation accuracy

#print(step_range)
print(epoch_spacing_range[-1])
# Epoch loss should be plotted at the end of each epoch, not end of each
# minibatch. Should end at 5699 but this is good enough for diagnostic
# purposes

print(f'Training loss list length:          {len(train_loss_list)}')
print(f'step_range length:                  {len(step_range)}')

print(f'Validation accuracy list length:    {len(validation_accuracy_list)}'
print(f'Validation loss list length:        {len(validation_loss_list)}')
print(f'epoch_spacing_range length:         {len(epoch_spacing_range)}')
```

```
Training data size: 900
Batchsize: 32
Epochs: 150

Datasize/batchsize, plus remainder slice --> # of training data divisions by
batch in each epoch: 29

4350.0
Training loss list length:          4350
step_range length:                  4350
Validation accuracy list length:    150
Validation loss list length:        150
epoch_spacing_range length:         150
```

Visualize and Evaluate Model

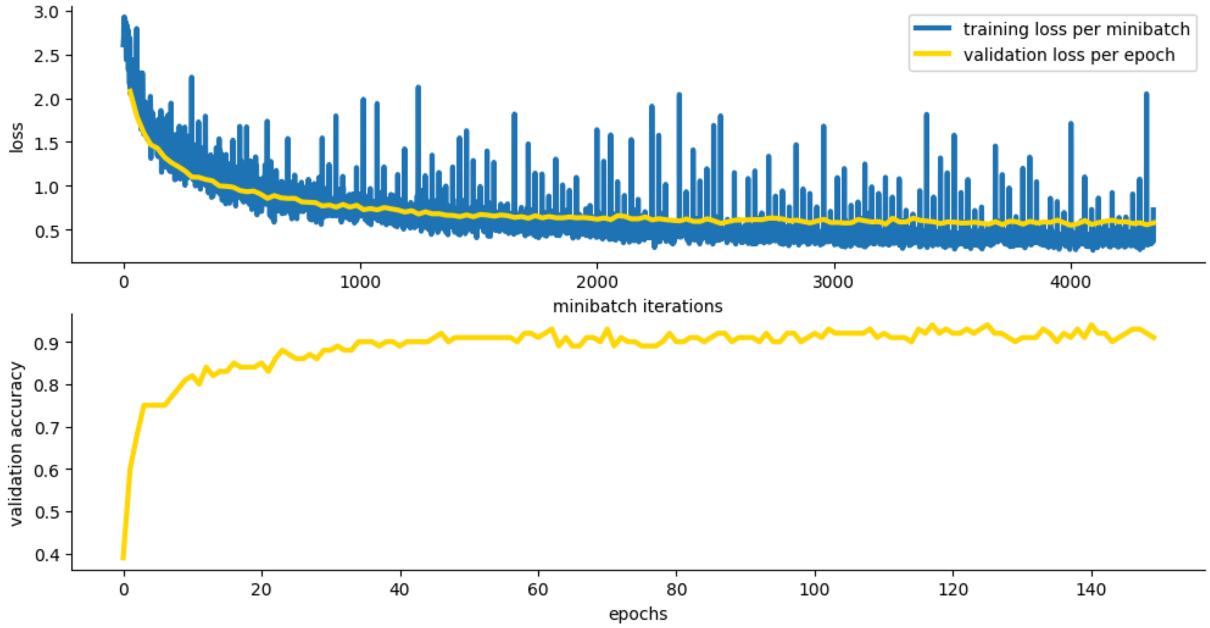
In [186…

```
# Import seaborn for prettier plots

import seaborn as sns
```

In [187…

```
# Visualize training loss

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

# Visualize training loss with respect to iterations
# (1 iteration -> single mini batch)
plt.subplot(2, 1, 1)
# this allows for visualizing minibatch training loss noise but noiseless
# validation loss
# use x axes computed in prev cell
plt.plot(step_range, train_loss_list, linewidth = 3)
plt.plot(epoch_spacing_range, validation_loss_list, linewidth = 3, color = '
plt.ylabel("loss")
plt.xlabel("minibatch iterations")
plt.legend(['training loss per minibatch', 'validation loss per epoch'])
```

```python
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")
plt.xlabel("epochs")
sns.despine()
```



In [188…

```python
# Compute the testing accuracy
model.eval()
with torch.no_grad():

    test_outputs = model(testing_inputs) # raw logits
    test_predicted = torch.argmax(test_outputs, dim=1)
    # convert to predicted labels as before

    # Compute the correct/ incorrect predictions and keep in array
    # with corresponding indices
    test_sucesses = test_predicted == testing_targets

    test_correct_num = (test_sucesses).sum().item()
    test_accuracy = test_correct_num / testing_targets.size(0)

    print(f"Test Accuracy: {test_accuracy*100}%")
model.train()
```

```
Test Accuracy: 94.0%
```

Out[188…

```
mnistClassification(
  (fc_in): Linear(in_features=784, out_features=512, bias=True)
  (fc1): Linear(in_features=512, out_features=128, bias=True)
  (fc2): Linear(in_features=128, out_features=64, bias=True)
  (fc_out): Linear(in_features=64, out_features=10, bias=True)
  (bn1): BatchNorm1d(512, eps=1e-05, momentum=0.1, affine=True, track_runni
ng_stats=True)
  (bn2): BatchNorm1d(128, eps=1e-05, momentum=0.1, affine=True, track_runni
ng_stats=True)
  (bn3): BatchNorm1d(64, eps=1e-05, momentum=0.1, affine=True, track_runnin
g_stats=True)
  (drop1): Dropout(p=0.3, inplace=False)
  (drop2): Dropout(p=0.3, inplace=False)
  (drop3): Dropout(p=0.3, inplace=False)
  (activation): ReLU()
)
```

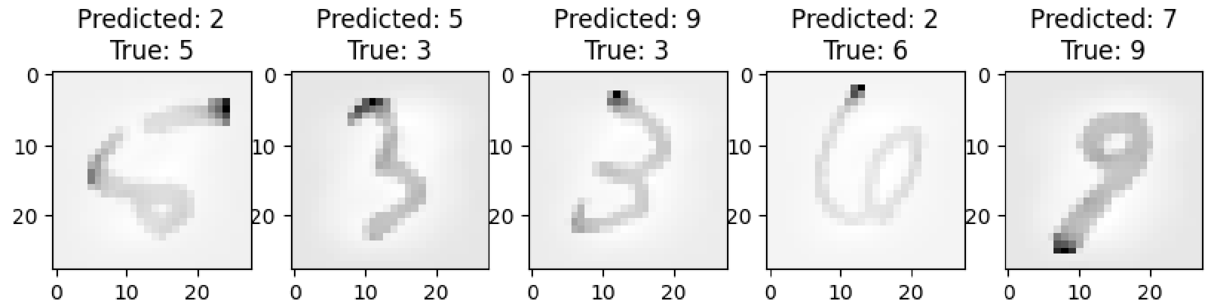
In [189…

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

failed_i = np.where(test_sucesses == False)[0]
# use boolean array from prev cell to find failed indices
#print(failed_i[:10])
```

In [190…

```
# convert back to 2D
mnist_test_features_resized = np.reshape(mnist_test_features,
                                          (100, 28, 28), copy=False)

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

plt.subplot(3,5,1)
# show the first failed sample
plt.imshow(mnist_test_features_resized[failed_i[0]], cmap = 'Greys')
# show the predicted and true labels for the failed sample
plt.title("Predicted: " + str(test_predicted[failed_i[0]].item()) +
           "\nTrue: " + str(testing_targets[failed_i[0]].item()))

plt.subplot(3,5,2)
plt.imshow(mnist_test_features_resized[failed_i[1]], cmap = 'Greys')
plt.title("Predicted: " + str(test_predicted[failed_i[1]].item()) +
           "\nTrue: " + str(testing_targets[failed_i[1]].item()))

plt.subplot(3,5,3)
plt.imshow(mnist_test_features_resized[failed_i[2]], cmap = 'Greys')
plt.title("Predicted: " + str(test_predicted[failed_i[2]].item()) + "\nTrue:

plt.subplot(3,5,4)
plt.imshow(mnist_test_features_resized[failed_i[3]], cmap = 'Greys')
plt.title("Predicted: " + str(test_predicted[failed_i[3]].item()) + "\nTrue:


plt.subplot(3,5,5)
plt.imshow(mnist_test_features_resized[failed_i[4]], cmap = 'Greys')
plt.title("Predicted: " + str(test_predicted[failed_i[4]].item()) + "\nTrue:
```

Out[190… Text(0.5, 1.0, 'Predicted: 7\nTrue: 9')



In [ ]:

In [ ]:

In [ ]: