STOR 235 — Lab 8

Instructions

There are two parts to this lab:

1. Example code for training neural networks for image recognition;
2. An image recognition task for you to complete.

There are no problems in the first part. There is a single problem in the second part, under **Problem.**

There is a code block under this introductory material that imports various things you will use later. Please run it.

Important Information About Submitting Your Assignment

Please upload your assignment as a PDF file, and make sure to select to all relevant pages for each part. Make sure that your PDF file contains all of your work, and that nothing has been cut off at the end. **You will not receive credit for solutions that are not in the PDF.**

```
from tensorflow.keras import layers, models
from tensorflow.keras.utils import to_categorical
import matplotlib.pyplot as plt
```

Part 1: A Neural Network Example

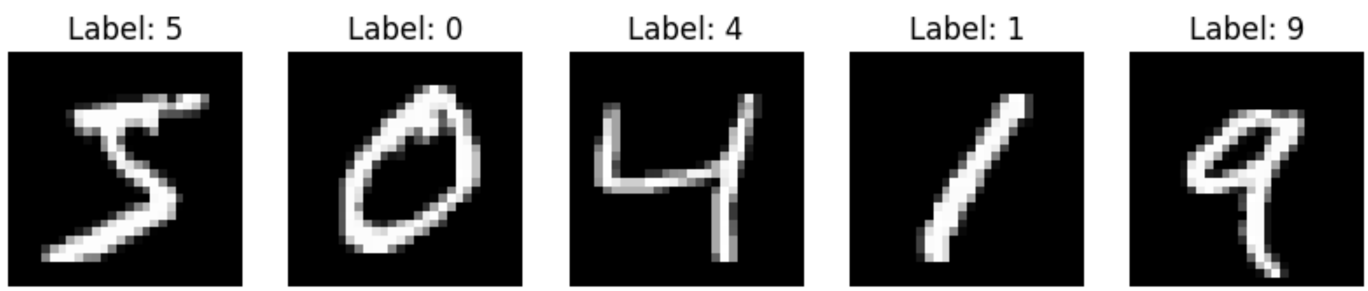
We now return to the MNIST handwriting dataset from the third lab. This time, we'll approach the classification problem using neural networks.

We begin by loading and visualizing the data.

```
# Load MNIST dataset
from tensorflow.keras.datasets import mnist
(x_train, y_train), (x_test, y_test) = mnist.load_data()


# Basic visualization of some MNIST elements
def visualize_mnist(x_data, y_data, num_samples):
    plt.figure(figsize=(10, 2))
    for i in range(num_samples):
        plt.subplot(1, num_samples, i + 1)
        plt.imshow(x_data[i], cmap='gray')
        plt.title(f"Label: {y_data[i]}")
        plt.axis('off')
    plt.show()

# Visualize some training samples
visualize_mnist(x_train, y_train, num_samples=5)
```



Next, we rescale the values of the pixels. Orignally on a scale from 0 to 255, we put them in the range [0,1]. This is not necessary, strictly speaking, but it is common to standardize data in this way (so parameters like the learning rate can easily be compared on a common scale between different models). We also relabel the pixels so that the images are vectors of length 784, instead of matrices of size 28 by 28.

```
# Normalize the input images to [0, 1] range
x_train = x_train.astype("float32") / 255.0
x_test = x_test.astype("float32") / 255.0

# Flatten the 28x28 images into vectors of size 784
x_train = x_train.reshape(-1, 28 * 28)
x_test = x_test.reshape(-1, 28 * 28)
```

We also conver the labels to vectors using one-hot encoding, as discussed on the slide.

```
# Convert labels to one-hot encoding
y_train_cat = to_categorical(y_train, 10)
y_test_cat = to_categorical(y_test, 10)

# Label vectors match the images above
print(y_train[0],y_train[1],y_train[2],y_train[3],y_train[4])
print(y_train_cat[0],y_train_cat[1],y_train_cat[2],y_train_cat[3],y_train_cat[4])
```

```
5 0 4 1 9
[0. 0. 0. 0. 0. 1. 0. 0. 0. 0.] [1. 0. 0. 0. 0. 0. 0. 0. 0. 0.] [0. 0. 0. 0. 
```

We now implement a softmax regression model for classifying the handwritten digits. Recall that this is a neural network with no hidden layer. The optimizer is `sgd` (stochastic gradient descent), and the loss function is the cross-entropy, both discussed on the slides. The `batch_size` parameter controls how the number of randomly subsampled points used to estimate the gradient at each step (see the slides for more).

The training progress is reported in "epochs." One epoch is a single pass through the entire training set (which consists of around 945 batches, and hence 945 gradient updates).

```
softmax_model = models.Sequential([
    layers.Input(shape=(784,)),
    layers.Dense(10, activation='softmax')
])

softmax_model.compile(optimizer='sgd',
                      loss='categorical_crossentropy',
                      metrics=['accuracy'])

print("Training Softmax Regression Model...")
softmax_model.fit(x_train, y_train_cat, epochs=10, batch_size=32)

# Evaluate on test data
test_loss, test_acc = softmax_model.evaluate(x_test, y_test_cat)
print(f"Softmax Regression Test Accuracy: {test_acc:.4f}")
```

```
Training Softmax Regression Model...
Epoch 1/10
1875/1875 ━━━━━━━━━━━━━━━━━━━━ 3s 2ms/step - accuracy: 0.7055 - loss: 1.1434
Epoch 2/10
1875/1875 ━━━━━━━━━━━━━━━━━━━━ 3s 1ms/step - accuracy: 0.8744 - loss: 0.4815
Epoch 3/10
1875/1875 ━━━━━━━━━━━━━━━━━━━━ 3s 2ms/step - accuracy: 0.8886 - loss: 0.4134
Epoch 4/10
1875/1875 ━━━━━━━━━━━━━━━━━━━━ 5s 1ms/step - accuracy: 0.8961 - loss: 0.3826
Epoch 5/10
1875/1875 ━━━━━━━━━━━━━━━━━━━━ 5s 1ms/step - accuracy: 0.9008 - loss: 0.3593
Epoch 6/10
1875/1875 ━━━━━━━━━━━━━━━━━━━━ 3s 2ms/step - accuracy: 0.9037 - loss: 0.3482
Epoch 7/10
1875/1875 ━━━━━━━━━━━━━━━━━━━━ 5s 1ms/step - accuracy: 0.9064 - loss: 0.3377
Epoch 8/10
1875/1875 ━━━━━━━━━━━━━━━━━━━━ 2s 1ms/step - accuracy: 0.9037 - loss: 0.3373
Epoch 9/10
1875/1875 ━━━━━━━━━━━━━━━━━━━━ 3s 1ms/step - accuracy: 0.9086 - loss: 0.3271
Epoch 10/10
1875/1875 ━━━━━━━━━━━━━━━━━━━━ 5s 1ms/step - accuracy: 0.9100 - loss: 0.3219
313/313 ━━━━━━━━━━━━━━━━━━━━ 1s 1ms/step - accuracy: 0.9036 - loss: 0.3502
Softmax Regression Test Accuracy: 0.9161
```

This should give you around 90% accuracy (or better) on the test set.

Next, we add a hidden layer, and try this larger network. This gives a bit better accuracy.

```
# Multilayer Perceptron (MLP) with One Hidden Layer
mlp_model = models.Sequential([
    layers.Input(shape=(784,)),
    layers.Dense(128, activation='relu'),
    layers.Dense(10, activation='softmax')
])

mlp_model.compile(optimizer='sgd',
                  loss='categorical_crossentropy',
                  metrics=['accuracy'])

print("\nTraining MLP Model...")
mlp_model.fit(x_train, y_train_cat, epochs=10, batch_size=32)

# Evaluate on test data
test_loss, test_acc = mlp_model.evaluate(x_test, y_test_cat)
print(f"MLP Test Accuracy: {test_acc:.4f}")
```

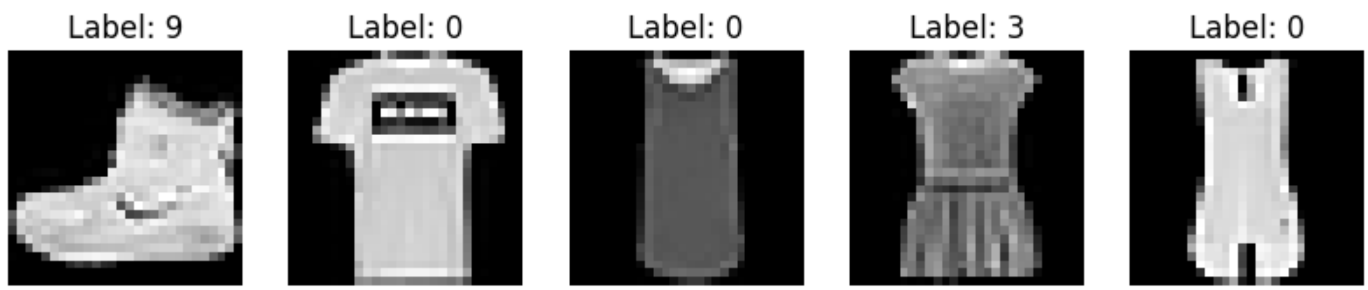
```
Training MLP Model...
Epoch 1/10
1875/1875 ━━━━━━━━━━━━━━━━━━━━ 4s 2ms/step - accuracy: 0.7269 - loss: 1.0342
Epoch 2/10
1875/1875 ━━━━━━━━━━━━━━━━━━━━ 3s 2ms/step - accuracy: 0.9011 - loss: 0.3552
Epoch 3/10
1875/1875 ━━━━━━━━━━━━━━━━━━━━ 4s 2ms/step - accuracy: 0.9174 - loss: 0.2967
Epoch 4/10
1875/1875 ━━━━━━━━━━━━━━━━━━━━ 3s 2ms/step - accuracy: 0.9255 - loss: 0.2655
Epoch 5/10
1875/1875 ━━━━━━━━━━━━━━━━━━━━ 3s 2ms/step - accuracy: 0.9327 - loss: 0.2417
Epoch 6/10
1875/1875 ━━━━━━━━━━━━━━━━━━━━ 6s 2ms/step - accuracy: 0.9388 - loss: 0.2182
Epoch 7/10
1875/1875 ━━━━━━━━━━━━━━━━━━━━ 4s 2ms/step - accuracy: 0.9436 - loss: 0.1994
Epoch 8/10
1875/1875 ━━━━━━━━━━━━━━━━━━━━ 6s 2ms/step - accuracy: 0.9469 - loss: 0.1898
Epoch 9/10
1875/1875 ━━━━━━━━━━━━━━━━━━━━ 3s 2ms/step - accuracy: 0.9504 - loss: 0.1779
Epoch 10/10
1875/1875 ━━━━━━━━━━━━━━━━━━━━ 5s 2ms/step - accuracy: 0.9537 - loss: 0.1650
313/313 ━━━━━━━━━━━━━━━━━━━━ 1s 2ms/step - accuracy: 0.9413 - loss: 0.1902
MLP Test Accuracy: 0.9516
```

Part 2: Fashion MNIST

We now load the Fashion MNIST dataset, visualize some of the training data, and normalize the pixel values to lie in [0,1] (just as before).

```
# Load the Fashion MNIST dataset
from tensorflow.keras.datasets import fashion_mnist
(x_train, y_train), (x_test, y_test) = fashion_mnist.load_data()
```

```
# Visualize some training data
visualize_mnist(x_train, y_train, num_samples=5)
```




```
# Normalize the pixel values to be between 0 and 1
x_train = x_train.astype('float32') / 255.0
x_test = x_test.astype('float32') / 255.0

# Flatten the 28x28 images into vectors of size 784
x_train_f = x_train.reshape(-1, 28 * 28)
x_test_f = x_test.reshape(-1, 28 * 28)

# One-hot encode the labels
y_train_f = to_categorical(y_train, 10)
y_test_f = to_categorical(y_test, 10)
```

Problem. Train a neural network that achieves 87.5% accuracy on the test set `y_test_f`.

Hint. Just copying the architecture from the previous part won't be quite good enough. Try making the hidden layer larger, adding another hidden layer (or more than one, perhaps with decreasing sizes), training for more epochs, and adjusting the learning rate of SGD. Code that you can use to change the learning rate is below. The default (use in the examples above) is 0.01. If you get really stuck, you can try copying the convolutional network shown in the slides, instead of just using a multi-layer perceptron.

```
(x_train, y_train), (x_test, y_test) = fashion_mnist.load_data()

x_train = x_train.astype("float32") / 255.0
x_test = x_test.astype("float32") / 255.0

x_train = x_train.reshape(-1, 28 * 28)
x_test = x_test.reshape(-1, 28 * 28)

y_train_f = to_categorical(y_train, 10)
y_test_f = to_categorical(y_test, 10)

model = models.Sequential([
    layers.Input(shape=(784,)),
    layers.Dense(256, activation='relu'),
    layers.Dense(128, activation='relu'),
    layers.Dense(10, activation='softmax')
])

optimizer = SGD(learning_rate=0.01)
model.compile(optimizer=optimizer,
              loss='categorical_crossentropy',
              metrics=['accuracy'])

print("Training Model...")
model.fit(x_train, y_train_f, epochs=20, batch_size=32)

test_loss, test_acc = model.evaluate(x_test, y_test_f)
print(f"\nFinal Test Accuracy: {test_acc:.4f}")
```

```
Training Model...
Epoch 1/20
1875/1875 ━━━━━━━━━━━━━━━━━━━━ 6s 3ms/step - accuracy: 0.6977 - loss: 0.9768
Epoch 2/20
1875/1875 ━━━━━━━━━━━━━━━━━━━━ 11s 3ms/step - accuracy: 0.8321 - loss: 0.4899
Epoch 3/20
1875/1875 ━━━━━━━━━━━━━━━━━━━━ 6s 3ms/step - accuracy: 0.8441 - loss: 0.4441
Epoch 4/20
1875/1875 ━━━━━━━━━━━━━━━━━━━━ 11s 3ms/step - accuracy: 0.8580 - loss: 0.4057
Epoch 5/20
1875/1875 ━━━━━━━━━━━━━━━━━━━━ 10s 3ms/step - accuracy: 0.8640 - loss: 0.3921
Epoch 6/20
1875/1875 ━━━━━━━━━━━━━━━━━━━━ 6s 3ms/step - accuracy: 0.8692 - loss: 0.3707
Epoch 7/20
1875/1875 ━━━━━━━━━━━━━━━━━━━━ 10s 3ms/step - accuracy: 0.8732 - loss: 0.3634
Epoch 8/20
1875/1875 ━━━━━━━━━━━━━━━━━━━━ 11s 3ms/step - accuracy: 0.8777 - loss: 0.3469
Epoch 9/20
1875/1875 ━━━━━━━━━━━━━━━━━━━━ 10s 3ms/step - accuracy: 0.8793 - loss: 0.3401
Epoch 10/20
1875/1875 ━━━━━━━━━━━━━━━━━━━━ 6s 3ms/step - accuracy: 0.8822 - loss: 0.3274
Epoch 11/20
1875/1875 ━━━━━━━━━━━━━━━━━━━━ 10s 3ms/step - accuracy: 0.8844 - loss: 0.3193
Epoch 12/20
1875/1875 ━━━━━━━━━━━━━━━━━━━━ 6s 3ms/step - accuracy: 0.8899 - loss: 0.3102
Epoch 13/20
1875/1875 ━━━━━━━━━━━━━━━━━━━━ 6s 3ms/step - accuracy: 0.8926 - loss: 0.3028
Epoch 14/20
1875/1875 ━━━━━━━━━━━━━━━━━━━━ 5s 3ms/step - accuracy: 0.8920 - loss: 0.2998
Epoch 15/20
1875/1875 ━━━━━━━━━━━━━━━━━━━━ 5s 3ms/step - accuracy: 0.8952 - loss: 0.2928
Epoch 16/20
1875/1875 ━━━━━━━━━━━━━━━━━━━━ 6s 3ms/step - accuracy: 0.8950 - loss: 0.2916
Epoch 17/20
1875/1875 ━━━━━━━━━━━━━━━━━━━━ 6s 3ms/step - accuracy: 0.8967 - loss: 0.2871
Epoch 18/20
1875/1875 ━━━━━━━━━━━━━━━━━━━━ 6s 3ms/step - accuracy: 0.9028 - loss: 0.2718
Epoch 19/20
1875/1875 ━━━━━━━━━━━━━━━━━━━━ 6s 3ms/step - accuracy: 0.9044 - loss: 0.2673
Epoch 20/20
1875/1875 ━━━━━━━━━━━━━━━━━━━━ 7s 4ms/step - accuracy: 0.9027 - loss: 0.2680
313/313 ━━━━━━━━━━━━━━━━━━━━ 1s 2ms/step - accuracy: 0.8812 - loss: 0.3385

Final Test Accuracy: 0.8824
```