CS 395
Homework 3 – Fashion MNIST
35 Points Total
Due in Canvas by 11:59 PM on Thursday, February 7, 2019

Train your first neural network: basic classification… This lab follows the activities outlined here:

https://www.tensorflow.org/tutorials/keras/basic_classification

But follow the instructions in this lab because they differ slightly from that exercise.

We will do some basic classification on the MNIST dataset. We will use tf.keras, a high-level API to build and train models in TensorFlow.

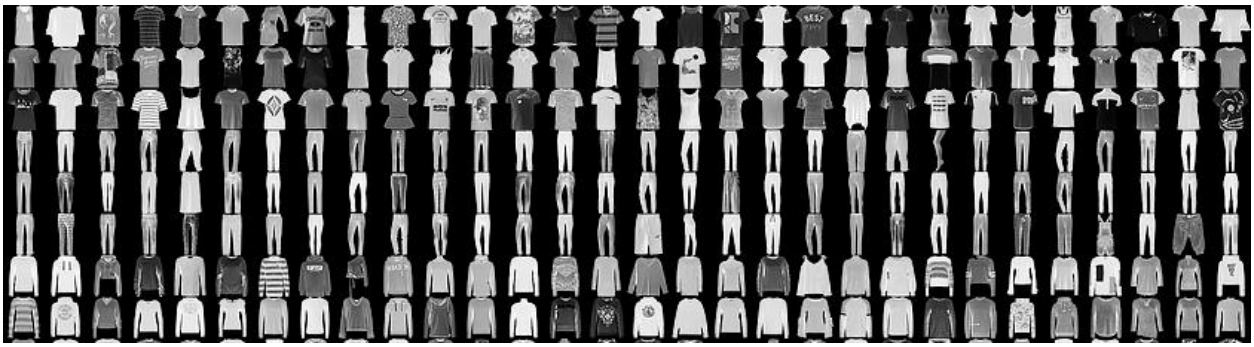
1. First let's see if we have TensorFlow installed. Run the following and provide the output (1 point).

```
# TensorFlow and tf.keras
import tensorflow as tf
from tensorflow import keras

# Helper libraries
import numpy as np
import matplotlib.pyplot as plt

print(tf.__version__)
```

2. Next we are going to **install the MNIST fashion dataset**, which contains 70,000 grayscale images in 10 categories. The images show individual articles of clothing at low resolution (28 by 28 pixels), as seen here:



Fashion MNIST is intended as a drop-in replacement for the classic MNIST dataset—often used as the "Hello, World" of machine learning programs for computer vision. The classic MNIST dataset contains images of handwritten digits (0, 1, 2, etc) in an identical format to the articles of clothing we'll use here.

This assignment uses Fashion MNIST for variety, and because it's a slightly more challenging problem than regular MNIST. Both datasets are relatively small and are used to verify that an algorithm works as expected. They're good starting points to test and debug code.

We will use 60,000 images to train the network and 10,000 images to evaluate how accurately the network learned to classify images. (We can also use a smaller training set and a larger test set, but for this assignment, the 6:1 split is okay.) You can access the Fashion MNIST directly from TensorFlow, just import and load the data:

```
fashion_mnist = keras.datasets.fashion_mnist

(train_images, train_labels), (test_images, test_labels) =
fashion_mnist.load_data()
```

We should get something like the following:

```
Downloading data from https://storage.googleapis.com/tensorflow/tf-keras-datasets/train-labels-idx1-ubyte.gz

32768/29515 [================================] - 0s 0us/step

Downloading data from https://storage.googleapis.com/tensorflow/tf-keras-datasets/train-images-idx3-ubyte.gz

26427392/26421880 [==============================] - 0s 0us/step

Downloading data from https://storage.googleapis.com/tensorflow/tf-keras-datasets/t10k-labels-idx1-ubyte.gz

8192/5148 [===============================================] - 0s 0us/step

Downloading data from https://storage.googleapis.com/tensorflow/tf-keras-datasets/t10k-images-idx3-ubyte.gz

4423680/4422102 [==============================] - 0s 0us/step
```

Loading the dataset returns four NumPy arrays:

- The `train_images` and `train_labels` arrays are the *training set*—the data the model uses to learn.
- The model is tested against the *test set*, the `test_images`, and `test_labels` arrays.

The images are 28x28 NumPy arrays, with pixel values ranging between 0 and 255.
The *labels* are an array of integers, ranging from 0 to 9. These correspond to the *class* of clothing the image represents:

Label	Class
0	T-shirt/top
1	Trouser

2	Pullover
3	Dress
4	Coat
5	Sandal
6	Shirt
7	Sneaker
8	Bag
9	Ankle boot

Each image is mapped to a single label. Since the *class names* are not included with the dataset, store them here to use later when plotting the images. Run the following:

```
class_names = ['T-shirt/top', 'Trouser', 'Pullover', 'Dress', 'Coat',
               'Sandal', 'Shirt', 'Sneaker', 'Bag', 'Ankle boot']
```

3. Next, let's **explore the data**:

Let's explore the format of the dataset before training the model. The following shows there are 60,000 images in the training set, with each image represented as 28 x 28 pixels.

a. Issue the following:

```
train_images.shape
```

We should get the following:

```
60000, 28, 28)
```

b. Likewise, there are 60,000 labels in the training set. If we issue the following:

```
len(train_labels)
```

We should get the following:

```
60000
```

c. Each label is an integer between 0 and 9. Issue the following:

```
train_labels
```

We should get the following:

```
array([9, 0, 0, ..., 3, 0, 5], dtype=uint8)
```

There are 10,000 images in the test set. Again, each image is represented as 28 x 28 pixels.

d. Issue the following:

```
test_images.shape
```

We should get the following:

```
(10000, 28, 28)
```

And the test set contains 10,000 images labels.

e. Issue the following:

```
len(test_labels)
```

We should get the following:

```
10000
```

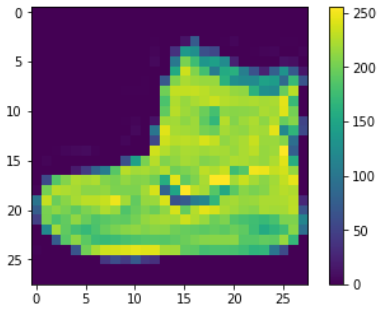
4. If you got the above answers, things look good. Now we need to **pre-process the data**.

The data must be preprocessed before training the network. If you inspect the first image in the training set, you will see that the pixel values fall in the range of 0 to 255:

a. Issue the following:

```
plt.figure()
plt.imshow(train_images[0])
plt.colorbar()
plt.grid(False)
plt.show()
```

The following should appear:



Nice, huh?

b. Now change the second line to see a separate image (use any other number in the train_images[] array between 1 and 9):

```
plt.figure()
plt.imshow(train_images[5])
plt.colorbar()
plt.grid(False)
plt.show()
```

Show what appears (1 point).

Next, we need to **normalize** the data. We scale these values to a range of 0 to 1 before feeding to the neural network model. For this, we divide the values by 255. It's important that the training set and the testing set are preprocessed in the same way.

c. Run the following:

```
train_images = train_images / 255.0
test_images = test_images / 255.0
```

Now if we issue the following, we should see the scale between 0 and 1. Show the output from the following (1 point).

```
plt.figure()
plt.imshow(train_images[0])
plt.colorbar()
plt.grid(False)
plt.show()
```

Let's display the first 25 images from the training set and display the class name below each image. Verify that the data is in the correct format and we're ready to build and train the network. Run the following:

```
plt.figure(figsize=(10,10))
for i in range(25):
    plt.subplot(5,5,i+1)
    plt.xticks([])
    plt.yticks([])
    plt.grid(False)
    plt.imshow(train_images[i], cmap=plt.cm.binary)
    plt.xlabel(class_names[train_labels[i]])
plt.show()
```

This just shows the first 25 images. You can play around with the figsize or the range or even the subplot values. From the above code, you should get something like the following:

All of our data is currently correctly labeled, but when we do the test set, we will remove the labels and try to calculate them.

5. Building the neural network requires configuring the layers of the model, then compiling the model.

The basic building block of a neural network is the **layer**. Layers extract representations from the data fed into them. And, hopefully, these representations are more meaningful for the problem at hand.

Most of deep learning consists of chaining together simple layers. Most layers, like `tf.keras.layers.Dense`, have parameters that are learned during training.

```
model = keras.Sequential([
    keras.layers.Flatten(input_shape=(28, 28)),
    keras.layers.Dense(128, activation=tf.nn.relu),
    keras.layers.Dense(10, activation=tf.nn.softmax)
])
```

Notice the activation functions? This is where we start experimenting.

We could also include dropout

```
keras.layers.Dropout(0.2),
```

The first layer in this network, `tf.keras.layers.Flatten`, transforms the format of the images from a 2d-array (of 28 by 28 pixels), to a 1d-array of 28 * 28 = 784 pixels. Think of this layer as unstacking rows of pixels in the image and lining them up. This layer has no parameters to learn; it only reformats the data. It just helps us with the processing.

After the pixels are flattened, the network consists of a sequence of two `tf.keras.layers.Dense` layers. These are densely-connected, or fully-connected, neural layers. The first `Dense` layer has 128 nodes (or neurons). The second (and last) layer is a 10-node **softmax** layer—this returns an array of 10 probability scores that sum to 1. We need to keep this at 10 because we have 10 classes of clothing items. Each node contains a score that indicates the probability that the current image belongs to one of the 10 classes.

However, the command:

```
keras.layers.Dense(128, activation=tf.nn.relu),
```

can be modified. The types of activation functions we use can be found here:

https://www.tensorflow.org/api_docs/python/tf/nn

We will come back to this a bit later and modify this line.

6. Next we compile the model:

Before the model is ready for training, it needs a few more settings. These are added during the model's **compile step**:

Loss function —This measures how accurate the model is during training. We want to minimize this function to "steer" the model in the right direction.

These can be found here:

https://www.tensorflow.org/api_docs/python/tf/keras/losses

Optimizer —This is how the model is updated based on the data it sees and its loss function.

https://www.tensorflow.org/api_docs/python/tf/keras/optimizers

Again, lots of choices here!

Metrics —Used to monitor the training and testing steps. The following example uses accuracy, the fraction of the images that are correctly classified.

Since we want to have an accurately classified model, we choose accuracy.

The available metrics can be found here:

https://www.tensorflow.org/api_docs/python/tf/keras/metrics

For now, run the following:

```
model.compile(optimizer='adam',
              loss='sparse_categorical_crossentropy',
              metrics=['accuracy'])
```

7. Next we **train the model**. Training the neural network model requires the following steps:

   I. Feed the **training data** to the model—in this example, the `train_images` and `train_labels` arrays.

   II. The model learns to associate images and labels.

   III. We ask the model to make predictions about a **test set**—in this example, the `test_images` array. We verify that the predictions match the labels from the `test_labels` array.

      a. To start training, call the `model.fit` method—the model is then "fit" to the training data:

```
model.fit(train_images, train_labels, epochs=5)
```

So we use 5 passes through the data (5 epochs) to learn about the training data. Will a smaller or greater number of epochs give better results? We will investigate that later.

Your results will look something like the following:

```
Epoch 1/5

60000/60000 [==============================] - 7s 112us/step - loss: 0.5029 - acc: 0.8219

Epoch 2/5

60000/60000 [==============================] - 6s 107us/step - loss: 0.3782 - acc: 0.8649

Epoch 3/5

60000/60000 [==============================] - 6s 104us/step - loss: 0.3393 - acc: 0.8767

Epoch 4/5

60000/60000 [==============================] - 6s 105us/step - loss: 0.3150 - acc: 0.8859

Epoch 5/5

60000/60000 [==============================] - 6s 106us/step - loss: 0.2978 - acc: 0.8910
```

As this model trains, the loss and accuracy metrics are displayed. This model reaches an accuracy of about 0.88 (or 88%) on the training data.

But that is on the **training set**. What about the unlabeled **test set**?

8. Next, we compare how the model performs on the test dataset:

```
test_loss, test_acc = model.evaluate(test_images, test_labels)
print('Test accuracy:', test_acc)
```

And we get something like:

```
10000/10000 [==============================] - 0s 49us/step

Test accuracy: 0.8727
```

It turns out, in the above example, the accuracy on the test dataset is a little less than the accuracy on the training dataset. This gap between training accuracy and test accuracy is an example of *overfitting*. Overfitting is when a machine learning model performs worse on new data than on their training data. Remember dropout helps us avoid this.

Assignment:

1. Provide TensorFlow version info in step 1. (1 point)
2. Provide image sample image in step 4 b. (use any other number in the train_images[] array between 1 and 9) (1 point)
3. Provide normalized data for 1 sample image in step 4 c. (1 point)
4. Perform steps 5-8 and show outputs (1 point)
5. Effects of Dropout. Perform steps 5-8 again, but this time include the dropout

   ```
   keras.layers.Dropout(0.2),
   ```

   in step 5. Show all outputs (2 points)
   How did it affect the results? (1 point)
6. Effects of Activation Functions. Perform steps 5-8 again, but each time include 2 different activation functions in the first layer other than relu.

   ```
   keras.layers.Dense(128, activation=tf.nn.choose_other_activation),
   ```

   in step 5. Choose reasonable options (you might need to do some research here). Show all outputs each time (4 points).
   How did it affect the results? (1 point)
7. Effects of Loss Function. Perform steps 5-8 again, but each time include 2 different loss functions other than 'sparse_categorical_crossentropy' in step 6. Choose reasonable options (you might need to do some research here). Show all outputs. (4 points).
   How did it affect the results? (1 point)

8. Effects of Optimizer. Perform steps 5-8 again, but each time include 2 different loss functions other than 'adam' in step 6. Choose reasonable options (you might need to do some research here). Show all outputs. (4 points).
   How did it affect the results? (1 point)
9. Effects of Number of Epochs. Perform steps 5-8 again but change the number of epochs in step 7. Instead of 5, use values of 2 and 8. Show all outputs. (4 points).
   How did it affect the results? (1 point)
10. Tie it all together. Make a table to show the results for accuracy for each method. (3 points).
    What did you learn overall? Which changes improved accuracy, and which did not? Explain. (5 points)