# PS2 - Fashion MNIST

October 9, 2018

## 1 Classify the kind of clothing item (pant, shirt, etc.) shown in the Fashion MNIST dataset

- Do this using the best practices discussed in class (i.e. one hot encoding, ReLU, CNNS when you should, etc.)
- Have reasonable hyperparameters

```
In [16]: # Importing the libraries
         import numpy as np
         import keras
         from matplotlib import pyplot as plt
         import seaborn as sns
         from keras.models import Sequential
         from keras.layers import Dense, Dropout, Activation, Flatten, BatchNormalization
         from keras.layers import Conv2D, MaxPooling2D
         from sklearn.metrics import confusion_matrix
         from sklearn.model_selection import train_test_split
         # from keras import backend as K
         # K.tensorflow_backend._get_available_gpus()
```

## 2 Loading the Training and Testing Data

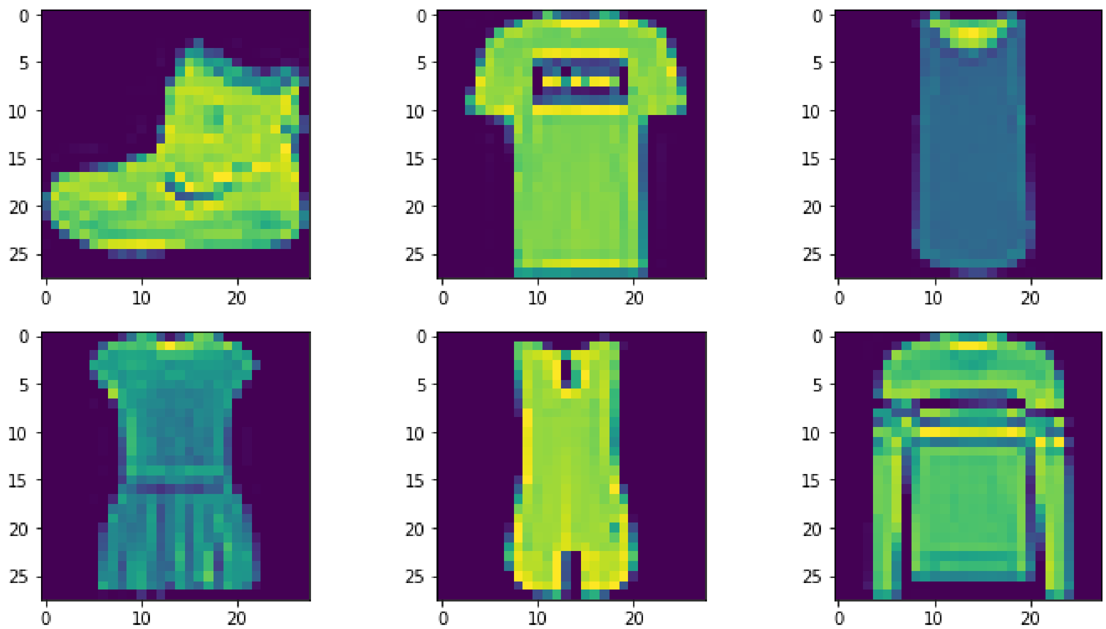
```
In [2]: # Loading the Training and Testing Data
        trainImages = np.load('./Fashion MNIST/trainImages.npy')
        testImages = np.load('./Fashion MNIST/testImages.npy')
        trainLabels = np.load('./Fashion MNIST/trainLabels.npy')
        testLabels = np.load('./Fashion MNIST/testLabels.npy')
```

## 3 Converting to Float32 to do input data Normalization

```
In [3]: # Converting to Float32 to do input data Normalization
        trainImages = trainImages.astype('float32') #images loaded in as int64, 0 to 255 integer
        testImages = testImages.astype('float32')
        # Normalization
        trainImages /= 255
        testImages /= 255
```

## 4 Showing the Input Data after Normalizing

In [4]:
```
# Preview the training data
plt.figure(figsize=(12,10))# Showing the Input Data after Normalizing
x, y = 3, 3
for i in range(6):
    plt.subplot(y, x, i+1)
    plt.imshow(trainImages[i].reshape((28,28)),interpolation='nearest')
plt.show()
```



## 5 Splitting the Data into Training and Validation Data

In [5]:
```
# Splitting the Training Data into training and Validation Data
X_train,X_val, Y_train,Y_val = train_test_split(trainImages, trainLabels, test_size=0.2)

print("For this problem the number of training data is:" + str(X_train.shape[0]) +"\nThe
" validation data is:" + str(X_val.shape[0]) + "\nand the number of test data is:" + str
```

```
For this problem the number of training data is:48000
The number of validation data is:12000
and the number of test data is:10000
```

## 6 Setting Hyper Paramters

In [6]:
```
# Hyper Parameters
batch_size = 64
num_classes = testLabels.shape[1]
epochs = 20
```

## 7 Creating Model

In [7]:
```
# create model
img_rows, img_cols = 28, 28
input_shape = (1,img_rows, img_cols)
model = Sequential()
model.add(Conv2D(32, kernel_size=(3, 3),
                 activation='relu',data_format='channels_first',use_bias=True ,
#                  bias_initializer = keras.initializers.glorot_uniform(seed=None),
                 input_shape=input_shape))          # Adding weights for tuning
# model.add(BatchNormalization())
model.add(Conv2D(64, (3, 3), activation='relu'))    # Added another layer as this
                                                    # smoothens the training accuracy cu
# model.add(BatchNormalization())
model.add(MaxPooling2D(pool_size=(2, 2)))           # Reducing the parmaters
model.add(Dropout(0.25))                             # Adding non linearity and randomne
model.add(Flatten())                                # Converting to 1D as required befor
                                                    # fully connected layer.
model.add(Dense(128, activation='relu'))            # fully connected layer with 128 hid
# model.add(BatchNormalization())
model.add(Dropout(0.5))                             # Adding non linearity and randomnes
model.add(Dense(num_classes, activation='softmax')) # Final fully connected layer.
# Compile model
model.compile(loss=keras.losses.categorical_crossentropy,
    optimizer=keras.optimizers.Adadelta(),metrics=['accuracy'])
# Get Model Summary
model.summary()
```

```
_________________________________________________________________
Layer (type)                 Output Shape              Param #
=================================================================
conv2d_1 (Conv2D)            (None, 32, 26, 26)        320
_________________________________________________________________
conv2d_2 (Conv2D)            (None, 30, 24, 64)        15040
_________________________________________________________________
max_pooling2d_1 (MaxPooling2 (None, 15, 12, 64)        0
_________________________________________________________________
dropout_1 (Dropout)          (None, 15, 12, 64)        0
_________________________________________________________________
flatten_1 (Flatten)          (None, 11520)             0
_________________________________________________________________
```

```
dense_1 (Dense)              (None, 128)               1474688
_________________________________________________________________
dropout_2 (Dropout)          (None, 128)               0
_________________________________________________________________
dense_2 (Dense)              (None, 10)                1290
=================================================================
Total params: 1,491,338
Trainable params: 1,491,338
Non-trainable params: 0
_________________________________________________________________
```

## 8 Weights and Biases for First Layer

In [8]:
```
# Before Training
weights = model.layers[0].get_weights()
w0 = np.array(weights[0])
b0 = np.array(weights[1])
print("The weights for first layer has dimensions of " + str(w0.shape))
print()
print("The Biases for first layer has dimensions of " + str(b0.shape)+" and the values a
print()
print("Just so we know that the weights have changed after training,\
 we will be comparing the sum of weights of he first layer before and after training.")
print()
print("The sum of weights of the first layer is :" + str(w0.sum()))
print("The sum of biases of the first layer is :" + str(b0.sum()))
```

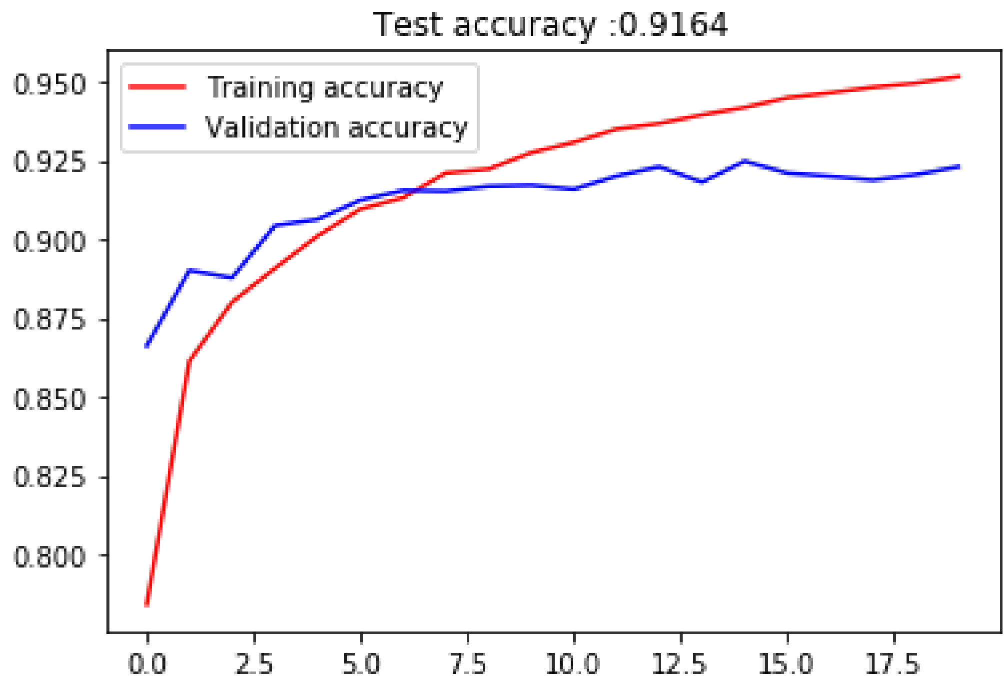
```
The weights for first layer has dimensions of (3, 3, 1, 32)

The Biases for first layer has dimensions of (32,) and the values are :
[0. 0. 0. 0. 0. 0. 0. 0. 0. 0. 0. 0. 0. 0. 0. 0. 0. 0. 0. 0. 0. 0. 0. 0.
 0. 0. 0. 0. 0. 0. 0. 0.]

Just so we know that the weights have changed after training, we will be comparing the sum of we

The sum of weights of the first layer is :-0.47791335
The sum of biases of the first layer is :0.0
```

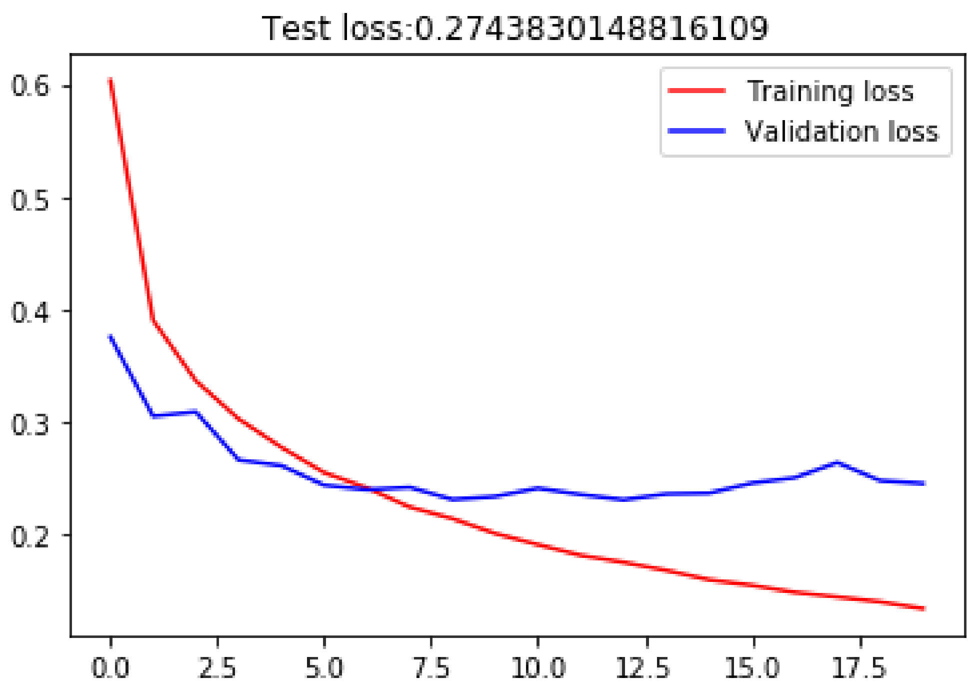
## 9 Training the Model

In [9]:
```
# Train
# Here I have considered the test data as the validation data.
h = model.fit(X_train, Y_train,
          batch_size=batch_size,
          epochs=epochs,
```

```
                    verbose=0,
        validation_data=(X_val, Y_val))

        # Evaluate Accuracy

        score = model.evaluate(testImages, testLabels, verbose=0)
        print('Training Loss:', h.history['loss'][-1])
        print('Training Accuracy:', h.history['acc'][-1])
        print()
        print('Validation Loss:', h.history['val_loss'][-1])
        print('Validation Accuracy:', h.history['val_acc'][-1])
        print()
        print('Test Loss:', score[0])
        print('Test Accuracy:', score[1])
        print()

        # Plot Graphs
        # print(h.history.keys())
        accuracy = h.history['acc']
        val_accuracy = h.history['val_acc']
        loss = h.history['loss']
        val_loss = h.history['val_loss']
        epochs = range(len(accuracy))
        plt.plot(epochs, accuracy, 'r', label='Training accuracy')
        plt.plot(epochs, val_accuracy, 'b', label='Validation accuracy')
        plt.title('Test accuracy :' + str(score[1]))
        plt.legend()
        plt.show()
        plt.figure()
        plt.plot(epochs, loss, 'r', label='Training loss')
        plt.plot(epochs, val_loss, 'b', label='Validation loss')
        plt.title('Test loss:'+str(score[0]))
        plt.legend()
        plt.show()
```

```
Training Loss: 0.13424885125209887
Training Accuracy: 0.951625

Validation Loss: 0.2452846585114797
Validation Accuracy: 0.923

Test Loss: 0.2743830148816109
Test Accuracy: 0.9164
```


Test accuracy :0.9164
Training accuracy
Validation accuracy
0.950
0.925
0.900
0.875
0.850
0.825
0.800
0.0
2.5
5.0
7.5
10.0
12.5
15.0
17.5


Test loss:0.2743830148816109
Training loss
Validation loss
0.6
0.5
0.4
0.3
0.2
0.0
2.5
5.0
7.5
10.0
12.5
15.0
17.5

## 10 Inferences

- The same model was used as used in MNIST. The model seems to be working good with only increase in number of Epochs to 20. the batch size was reduced to to 64 for more updates.
- Before using the same model I tried Fashion MNISt with other models as well. Below are the accuracy received with each model:
  - With 3 Dense (fully connected layers); Test Accuracy after 12 epochs was; 90.82 %
  - With 2 Dense and 1 convolution : Test Accuracy after 12 epochs: 91.11 %
  - With 2 Dense and 2 convolution: Test Accuracy is after 12 epochs : 91.74 %
  - Therefore, the final option was adopted.
- Therefore, when there is image data, use of conv2D layers help in feature exraction.
- There are 2 Convolution Layers and 2 Dense or Fully Connected Layers with subsequent Max Pooling and Dropout Layer.
- The number of hidden units in each layer are experimentally fine tuned. The units were experimented in 2^n multiples like 16, 32 , 64 ,128 , 256.
- ** Input Conv2D layer: ** The weights are initialized by Glorot uniform initializer, which is the default. This is set as the values are nicely zero centered and within -1 and 1.The biases are set to zero as it improves the accuracy by 0.14 as compared to randomly distribued values between (-1 and 1)(found out by experimnetation).This is also recommended as per in CS231 from Standord.
  - The kernel size is set to minimum of (3,3). This is because the image size is already small (28,28) and by using large kernel size there wont be enough data left to add more convolution layers.
  - the activation is set to RELU, as it is the most recommended and also as the input is normalized between 0 and 1.
- ** Max Pooling layer: ** for pooling Max Pooling is used. Again as the input size is very less. The maxpooling kernel size is set to (2,2). This works as there is still enough information in a (2,2) window and it also does not throw out a lot of valuable data.
- ** Drop Out: ** This is very Essential as it helps prevent over fitting. Two dropout layers are used.
  - The first one is in between convolution layers. The dropout of 0.25 was determined to be optimal. At this vlaue the convergence of Training accuracy to validation accuracy was smooth and gradual as compared to other values.
  - The second dropout was used just before the final fully connected layer. This is kept at 0.45 as experimnetally determined and is used to reduce overfitting.
- ** Dense Layers: ** Two fully connect layers are used with a droppout in between. This seems to give good performances in MNIST and Fashion MNIST.
  - For final fully connected layer, softmax is used as activation. This is because there are a total of 10 classes and the predicted output can exactly be any one of these classes.
- ** Loss ** : as one hot encoding is used, categorical_crossentropy loss is used as loss function. This is used as it is the most recommended one.
- ** Optimizer ** : Different Optimizers were tried out. Even for Fashion MNIST, for different optimizers there was not a lot of change in the convergence time or the number of epochs.

- For example, I used Adadelta and each epoch took 4.5 seconds less to finish as compared to Adam, also the final test accuracy was higher for Adadelta. Therefore I finally decided to use Adadelta as my optimizers. After playing with the parmaters, I come to the conclusion that the default value works the best.

## 11 Weights and Biases for First Layer After Training

In [13]:
```
# After Training
weights = model.layers[0].get_weights()
w0 = np.array(weights[0])
b0 = np.array(weights[1])
print("The weights for first layer has dimensions of " + str(w0.shape))
print()
print("The Biases for first layer has dimensions of " + str(b0.shape)+" and the values
print()
print("The sum of weights of the first layer is :" + str(w0.sum()))
print("The sum of biases of the first layer is :" + str(b0.sum()))
print("As we can see the sum has changed, indicating that the weights are now tuned.")
```

```
The weights for first layer has dimensions of (3, 3, 1, 32)

The Biases for first layer has dimensions of (32,) and the values are :
[ 0.03732191  0.03564902 -0.02086273  0.00827906 -0.01473734  0.01437863
 -0.00281613  0.00623995 -0.00739944 -0.06542691 -0.00145124 -0.14998387
 -0.01268976 -0.01216698 -0.01292941  0.02989011  0.01361851  0.01142375
  0.01032185 -0.03882701 -0.01248314  0.02152263 -0.09115795 -0.00246029
 -0.00022315 -0.00455045  0.04222413  0.00285289 -0.02125233  0.00818023
  0.05506296 -0.07067282]

The sum of weights of the first layer is :-0.62720954
The sum of biases of the first layer is :-0.24512535
As we can see the sum has changed, indicating that the weights are now tuned.
```

## 12 Calculate Confusion Matrix

In [14]:
```
# Predict the values from the validation dataset
Y_pred = model.predict(testImages)
# Convert predictions classes to one hot vectors
Y_pred_classes = np.argmax(Y_pred, axis = 1)
# Convert validation observations to one hot vectors
Y_true = np.argmax(testLabels, axis = 1)
# compute the confusion matrix
confusion_mtx = confusion_matrix(Y_true, Y_pred_classes)
print("Confusion Matrix is :\n"+str(confusion_mtx))
```

```
Confusion Matrix is :
[[886   2  18  13   4   1  70   0   6   0]
```

```
 [   2 983   0   9   3   0   2   0   1   0]
 [ 25   1 880   4  50   0  38   0   2   0]
 [ 11   5  11 901  44   0  26   0   2   0]
 [  3   1  52  18 895   0  30   0   1   0]
 [  0   0   0   0   0 976   0  18   0   6]
 [111   1  72  19  68   0 725   0   4   0]
 [  0   0   0   0   0   8   0 973   0  19]
 [  4   1   1   3   4   1   2   4 980   0]
 [  1   0   0   0   0   7   0  27   0 965]]
```

## 13 Incorrect Predictions

In [15]:
```
# Errors are difference between predicted labels and true labels
errors = (Y_pred_classes - Y_true != 0)
incorrect = (errors*1).sum()
print ("Number of Incorrect Predicitons are: " + str(incorrect)+ " out of "+ str (error
Y_pred_classes_errors = Y_pred_classes[errors]
Y_pred_errors = Y_pred[errors]
Y_true_errors = Y_true[errors]
X_val_errors = testImages[errors]

def display_errors(errors_index,img_errors,pred_errors, obs_errors):
    """ This function shows 6 images with their predicted and real labels"""
    a = ['T-shirt','Trouser','Pullover','Dress','Coat','Sandal','Shirt','Sneaker','Bag'
    n = 0
    nrows = 2
    ncols = 3
    fig, ax = plt.subplots(nrows,ncols,sharex=True,sharey=True)
    for row in range(nrows):
        for col in range(ncols):
            error = errors_index[n]
            ax[row,col].imshow((img_errors[error]).reshape((28,28)))
            ax[row,col].set_title("Predicted label :{}\nTrue label :{}".format(a[pred_e
                                                                            a[obs_er
            n += 1
    fig.tight_layout()



# Probabilities of the wrong predicted numbers
Y_pred_errors_prob = np.max(Y_pred_errors,axis = 1)

# Predicted probabilities of the true values in the error set
true_prob_errors = np.diagonal(np.take(Y_pred_errors, Y_true_errors, axis=1))

# Difference between the probability of the predicted label and the true label
```

```python
delta_pred_true_errors = Y_pred_errors_prob - true_prob_errors

# Sorted list of the delta prob errors
sorted_dela_errors = np.argsort(delta_pred_true_errors)

# Top 6 errors
most_important_errors = sorted_dela_errors[-6:]

# Show the top 6 errors
print()
print("Displaying the top 6 errors")
display_errors(most_important_errors, X_val_errors, Y_pred_classes_errors, Y_true_error
```

```
Number of Incorrect Predicitons are: 836 out of 10000 inputs

Displaying the top 6 errors
```