DEEP LEARNING - CNN PROJECT
FASHION MNIST DATASET

ROLL NO: 2018103592 **DATE: 20 April 2021**
NAME: SHRUTHI M

PROBLEM STATEMENT

More than 25% of entire revenue in E-Commerce is attributed to apparels & accessories. A major problem they face is categorizing these apparels from just the images especially when the categories provided by the brands are inconsistent.

The Fashion-MNIST clothing classification problem is a standard dataset used in computer vision and deep learning. Although the dataset is relatively simple, it can be used as the basis for learning and practicing how to develop, evaluate, and use deep convolutional neural networks for image classification from scratch. Although the original dataset contains only around 60000 images, we improve the performance by data augmentation. Significant regularization techniques are applied to the model to ensure high accuracy.

This problem statement will help companies to automatically categorize the clothes based on the image which significantly reduces the workload of humans.

DATASET DETAILS

https://www.kaggle.com/zalando-research/fashionmnist

Fashion-MNIST is a dataset of Zalando's article images—consisting of a training set of 60,000 examples and a test set of 10,000 examples. Each example is a 28x28 grayscale image, associated with a label from 10 classes.

Each image is 28 pixels in height and 28 pixels in width, for a total of 784 pixels in total. Each pixel has a single pixel-value associated with it, indicating the lightness or darkness of that pixel, with higher numbers meaning darker. This pixel-value is an integer between 0 and 255. The training and test data sets have 785 columns. The first column consists of the class labels (see above), and represents the article of clothing. The rest of the columns contain the pixel-values of the associated image.

Labels

Each training and test example is assigned to one of the following labels:

- 0 T-shirt/top
- 1 Trouser
- 2 Pullover
- 3 Dress
- 4 Coat
- 5 Sandal
- 6 Shirt
- 7 Sneaker
- 8 Bag
- 9 Ankle boot

MODULES

EXPLORING AND VISUALISING DATASET

Here, we import the dataset and analyse the data. The dataset contains label and pixel values.
Label: The Target variable
Pixels: The smallest unit of a Digital Image or Graphic that can be displayed on Digital Display Device.

NORMALISATION

The Pixel Values are often stored as Integer Numbers in the range 0 to 255. They need to be scaled down to [0,1] in order for Optimization Algorithms to work much faster. Here, we use Zero Mean and Unit Variance for standardizing. The formula is (x-mean)/variance. Along with normalisation, we also create a validation set to monitor the evolution of our model.

ONE HOT ENCODING

The labels are given as integers between 0-9. We need to one hot encode them. Eg 8 [0, 0, 0, 0, 0, 0, 0, 0, 1, 0] . We have 10 digits [0-9] or classes, therefore we one-hot-encode the target variable with 10 classes

DATA AUGMENTATION

Data Augmentation increases the dataset size by applying various transformations on the original dataset. This not only increases the size but also helps in creating the data with different perspectives. These are some of the transformations that are performed.

RandomRotation(degrees) : Rotate the image by angle

degrees : Range of degrees to select from. If degrees is a number instead of a sequence like (min, max), the range of degrees will be (-degrees, +degrees)

RandomResizedCrop(size) : Crop the given PIL Image to random size and aspect ratio.

size – expected output size of each edge

RandomHorizontalFlip(p) : Horizontally flip the given PIL Image randomly with a given probability p.

Resize(size) : Resize the input PIL Image to the given size.

size (sequence or int) – Desired output size. If size is a sequence like (h, w), output size will be matched to this. If size is an int, the smaller edge of the image will be matched to this number. i.e, if height > width, then image will be rescaled to (size * height / width, size)

CenterCrop(size): Crops the given PIL Image at the center.

BUILDING MODEL

Here, we come up with our own idea of CNN model. The next section gives a detailed description of our model

FITTING THE MODEL

The images are fit into the model and trained

PLOTTING GRAPHS

We plot training and validation errors and accuracy graph

PREDICTION & EVALUATION

The model is evaluated with the testing set and confusion matrix is plotted.

CNN MODEL SUMMARY

```
from keras.regularizers import l2
model = Sequential()

# Hidden layer 1
model.add(Conv2D(filters=32, kernel_size=(3, 3), activation='relu', strides=1, padding='same', input_shape=(28,28,1)))
model.add(BatchNormalization())
model.add(Dropout(0.3))

# Hidden layer 2
model.add(Conv2D(filters=64, kernel_size=(3, 3), activation='relu', strides=1, padding='same'))
model.add(BatchNormalization())
model.add(Dropout(0.4))
model.add(MaxPooling2D(pool_size=(2, 2)))

# Hidden layer 3
model.add(Conv2D(filters=64, kernel_size=(5, 5), activation='relu', strides=1, padding='same', kernel_regularizer=l2(0.01)))
model.add(BatchNormalization())
model.add(Dropout(0.5))
model.add(MaxPooling2D(pool_size=(2, 2)))

# Hidden layer 4
model.add(Conv2D(filters=128, kernel_size=(3, 3), activation='relu', strides=1, padding='same', kernel_regularizer=l2(0.01)))
model.add(BatchNormalization())
model.add(Dropout(0.25))
```

```
# Output Layer
model.add(Flatten())
model.add(Dense(512, activation='relu'))
model.add(BatchNormalization())
model.add(Dropout(0.5))
model.add(Dense(128, activation='relu'))
model.add(BatchNormalization())
model.add(Dropout(0.5))
model.add(Dense(10, activation='softmax'))
```

In the **first hidden layer**, we have **32 input channels** and the kernel shape is **3*3** with a stride value of 1. Padding is of type '**Same'** which ensures output dimensions are the same as input dimensions. Activation function that is used is **'relu'**. We also perform few regularization techniques such as **'Batch Normalisation'** and **dropout of 0.3** (30% of nodes are dropped)

In the **second hidden layer**, we have **64 input channels** and the kernel shape is **3*3** with a stride value of 1. Padding is of type '**Same'** which ensures output dimensions are the same as input dimensions. Activation function that is used is **'relu'**. **'Batch Normalisation'** is performed and this time we set **dropout to 0.4** (40% of nodes are dropped). After this, **'max pooling'** is done with pooling size of **2*2** which suppresses noise as well as reduces the dimensions.

In the **third hidden layer**, we have **64 input channels** and the kernel shape is **5*5** with a stride value of 1. Padding is of type '**Same'** which ensures output dimensions are the same as input dimensions. Activation function that is used is **'relu'**. **'Batch Normalisation'** is performed and this time we set **dropout to 0.5** (50% of nodes are dropped). Along with that, we also penalize by applying **L2 Norm Regularization** with **alpha value of 0.01**. After this, **'max pooling'** is done with pooling size of **2*2** which suppresses noise as well as reduces the dimensions.

In the **fourth (final) hidden layer**, we have **128 input channels** and the kernel shape is **3*3** with a stride value of 1. Padding is of type '**Same'** which ensures output dimensions are the same as input dimensions. Activation function that is used is **'relu'**. **'Batch Normalisation'** is performed and this time we set **dropout to 0.25** (25% of nodes are dropped). Along with that, we also penalize by applying **L2 Norm Regularization** with **alpha value of 0.01**.

In the **output layer**, we flatten the nodes and change the shape to **512 nodes** with **'relu'** activation and then perform **'batch normalisation'** and **'dropout' of 0.5.** We further reduce this to **128** nodes and then apply **'batch normalisation'** and **'dropout' of 0.5.** Finally the nodes are reduced to the size of the class set (i.e. 10) and **'softmax'** activation is used.

```
[38] Summary of the model:
Model: "sequential"
_________________________________________________________________
Layer (type)                 Output Shape              Param #
=================================================================
conv2d (Conv2D)              (None, 28, 28, 32)        320
_________________________________________________________________
batch_normalization (BatchNo (None, 28, 28, 32)        128
_________________________________________________________________
dropout (Dropout)            (None, 28, 28, 32)        0
_________________________________________________________________
conv2d_1 (Conv2D)            (None, 28, 28, 64)        18496
_________________________________________________________________
batch_normalization_1 (Batch (None, 28, 28, 64)        256
_________________________________________________________________
dropout_1 (Dropout)          (None, 28, 28, 64)        0
_________________________________________________________________
max_pooling2d (MaxPooling2D) (None, 14, 14, 64)        0
_________________________________________________________________
conv2d_2 (Conv2D)            (None, 14, 14, 64)        102464
_________________________________________________________________
batch_normalization_2 (Batch (None, 14, 14, 64)        256
_________________________________________________________________
dropout_2 (Dropout)          (None, 14, 14, 64)        0
_________________________________________________________________
max_pooling2d_1 (MaxPooling2 (None, 7, 7, 64)          0
_________________________________________________________________
```

[38]

```
_________________________________________________________________
conv2d_3 (Conv2D)            (None, 7, 7, 128)         73856
_________________________________________________________________
batch_normalization_3 (Batch (None, 7, 7, 128)         512
_________________________________________________________________
dropout_3 (Dropout)          (None, 7, 7, 128)         0
_________________________________________________________________
flatten (Flatten)            (None, 6272)              0
_________________________________________________________________
dense (Dense)                (None, 512)               3211776
_________________________________________________________________
batch_normalization_4 (Batch (None, 512)               2048
_________________________________________________________________
dropout_4 (Dropout)          (None, 512)               0
_________________________________________________________________
dense_1 (Dense)              (None, 128)               65664
_________________________________________________________________
batch_normalization_5 (Batch (None, 128)               512
_________________________________________________________________
dropout_5 (Dropout)          (None, 128)               0
_________________________________________________________________
dense_2 (Dense)              (None, 10)                1290
=================================================================
Total params: 3,477,578
Trainable params: 3,475,722
Non-trainable params: 1,856
_________________________________________________________________
```

This is the final summary of the model we created

CODING SNAPSHOTS

MODULE I - EXPLORING AND VISUALISING DATASET

Importing packages

Importing Packages

```
import numpy as np
import pandas as pd
import seaborn as sns
import matplotlib.pyplot as plt
import os
from glob import glob
import cv2

from keras.models import Sequential
from keras.layers import Conv2D, MaxPooling2D, Activation, Dropout, Flatten, Dense
from keras.preprocessing.image import ImageDataGenerator, img_to_array, load_img
```

```
[4] from keras.models import Sequential
    from keras.layers import Conv2D, MaxPooling2D, Activation, Dropout, Flatten, Dense
    from keras.preprocessing.image import ImageDataGenerator, img_to_array, load_img
    import matplotlib.pyplot as plt
    from PIL import Image
    from glob import glob
    import tensorflow as tf
```

Since, training takes a lot of time with CPU, we use GPU

```
print(tf.test.gpu_device_name())
from tensorflow.python.client import device_lib
print(device_lib.list_local_devices())
!cat /proc/meminfo
```

```
, name: "/device:GPU:0"
device_type: "GPU"
memory_limit: 11154422528
locality {
  bus_id: 1
  links {
  }
}
incarnation: 9190846040501557904
physical_device_desc: "device: 0, name: Tesla K80, pci bus id: 0000:00:04.0, compute capability: 3.7"
]
MemTotal:       13333564 kB
MemFree:         9671036 kB
MemAvailable:   12111652 kB
Buffers:           85616 kB
Cached:          2403048 kB
SwapCached:            0 kB
Active:          1248500 kB
Inactive:        2026676 kB
Active(anon):     659376 kB
Inactive(anon):     2412 kB
Active(file):     589124 kB
```

Importing dataset

```
[20] train = pd.read_csv('./fashion-mnist_train.csv')
     test = pd.read_csv('./fashion-mnist_test.csv')
     df_train = train.copy()
     df_test = test.copy()
```

```
[21] df_train.head()
```

	label	pixel1	pixel2	pixel3	pixel4	pixel5	pixel6	pixel7	pixel8	pixel9	pixel10	pixel11	pixel12	pixel13	pixel14	pixel15	pixel16
0	2	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0
1	9	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0
2	6	0	0	0	0	0	0	0	5	0	0	0	105	92	101	107	100
3	0	0	0	0	1	2	0	0	0	0	0	114	183	112	55	23	72
4	3	0	0	0	0	0	0	0	0	0	0	0	0	46	0	21	68

5 rows × 785 columns

Shape and info of dataset

Find the shape and other details about image

```
[22] print("Training data shape: ", df_train.shape)
     print("Testing data shape: ", df_test.shape)
```

```
Training data shape:  (60000, 785)
Testing data shape:  (10000, 785)
```

```
[23] print("Dataset information type: ")
     print(df_train.info())
```

```
Dataset information type:
<class 'pandas.core.frame.DataFrame'>
RangeIndex: 60000 entries, 0 to 59999
Columns: 785 entries, label to pixel784
dtypes: int64(785)
memory usage: 359.3 MB
None
```

Finding the no of labels

```
[24] print("The total no of labels: ", (df_train.label.unique().shape))
     print("The unique labels are: ", df_train.label.unique())
```

```
The total no of labels:  (10,)
The unique labels are:  [2 9 6 0 3 4 5 8 7 1]
```

```
[25] print("The ditribution of labels: ", df_train['label'].value_counts())
```

```
The ditribution of labels:  9    6000
8    6000
7    6000
6    6000
5    6000
4    6000
3    6000
2    6000
1    6000
0    6000
Name: label, dtype: int64
```

Separating input and label

```
[26] X_train = df_train.iloc[:,1:]
     y_train = df_train.iloc[:,0:1]
     print("Shape of X: ", X_train.shape)
     print("Shape of Y: ",y_train.shape)
```

```
Shape of X:  (60000, 784)
Shape of Y:  (60000, 1)
```

Since shape is 784, we reshape that to 28*28

```
[27] X_train = X_train.values.reshape(X_train.shape[0],28,28,1)
```

```
[28] print("X shape: ", X_train.shape)
```

```
X shape:  (60000, 28, 28, 1)
```

```
[29] X_test = df_test.iloc[:,1:]
     y_test = df_test.iloc[:,0:1]
     print("Shape of X: ", X_test.shape)
     print("Shape of Y: ",y_test.shape)
```

```
Shape of X:  (10000, 784)
Shape of Y:  (10000, 1)
```

```
[30] X_test = X_test.values.reshape(X_test.shape[0],28,28,1)
```

```
[31] print("X shape: ", X_test.shape)
```

```
X shape:  (10000, 28, 28, 1)
```

MODULE II - NORMALISATION

Normalisation

Since pixels are of the range 0 to 255, we need to scale them

```
[32] X_train = X_train.astype("float32")/255
     X_test = X_test.astype("float32")/255
```

We create validation set with the training set so that we can monitor and perform EarlyStopping if needed

Create validation set also

```
[41] from sklearn.model_selection import train_test_split
     X_train, X_val, y_train, y_val = train_test_split(X_train, y_train, test_size=0.15)

     print('Training set shape: ', X_train.shape, y_train.shape)
     print('Validation set shape: ', X_val.shape, y_val.shape)
```

```
Training set shape:  (51000, 28, 28, 1) (51000, 10)
Validation set shape:  (9000, 28, 28, 1) (9000, 10)
```

MODULE III - ONE HOT ENCODING

Labels are turned to 1-hot encoded form. For example: label 3 is converted to [0,0,0,1,0,0,0,0,0,0]

One hot encoding

```
y_train = to_categorical(y_train, num_classes=10)
y_test = to_categorical(y_test, num_classes=10)
```

```
[ ] print("Shape of y train: ", y_train.shape)
    print("Shape of y test: ",y_test.shape)

    Shape of y train:  (60000, 10)
    Shape of y test:  (10000, 10)
```

```
[ ] print("Example: ", y_train[0])

    Example:  [0. 0. 1. 0. 0. 0. 0. 0. 0. 0.]
```

MODULE IV - DATA AUGMENTATION

To perform augmentation, we use the inbuilt Keras ImageDataGenerator where we specify a set of transformations that should be applied to our original dataset. Data augmentation helps to prevent overfitting and trains the model with images from different perspectives. Here, we give rotation_range to be 0.5, zoom_range as 0.1 and set width_shift_range and height_shift_range to 0.3 and 0.1 respectively.

Data Augmentation

```
from keras.preprocessing.image import ImageDataGenerator
datagen = ImageDataGenerator(
        featurewise_center=False,  # set input mean to 0 over the dataset
        samplewise_center=False,  # set each sample mean to 0
        featurewise_std_normalization=False,  # divide inputs by std of the dataset
        samplewise_std_normalization=True,  # divide each input by its std
        zca_whitening=False,  # dimesion reduction
        rotation_range=0.5,  # randomly rotate images in the range
        zoom_range = 0.1, # Randomly zoom image
        width_shift_range=0.3,  # randomly shift images horizontally
        height_shift_range=0.1,  # randomly shift images vertically
        horizontal_flip=False,  # randomly flip images
        vertical_flip=False)  # randomly flip images

datagen.fit(X_train)
```

MODULE V - BUILDING MODEL

```
from keras.regularizers import l2
model = Sequential()

# Hidden layer 1
model.add(Conv2D(filters=32, kernel_size=(3, 3), activation='relu', strides=1, padding='same', input_shape=(28,28,1)))
model.add(BatchNormalization())
model.add(Dropout(0.3))

# Hidden layer 2
model.add(Conv2D(filters=64, kernel_size=(3, 3), activation='relu', strides=1, padding='same'))
model.add(BatchNormalization())
model.add(Dropout(0.4))
model.add(MaxPooling2D(pool_size=(2, 2)))

# Hidden layer 3
model.add(Conv2D(filters=64, kernel_size=(5, 5), activation='relu', strides=1, padding='same', kernel_regularizer=l2(0.01)))
model.add(BatchNormalization())
model.add(Dropout(0.5))
model.add(MaxPooling2D(pool_size=(2, 2)))

# Hidden layer 4
model.add(Conv2D(filters=128, kernel_size=(3, 3), activation='relu', strides=1, padding='same', kernel_regularizer=l2(0.01)))
model.add(BatchNormalization())
model.add(Dropout(0.25))
```

```
# Output Layer
model.add(Flatten())
model.add(Dense(512, activation='relu'))
model.add(BatchNormalization())
model.add(Dropout(0.5))
model.add(Dense(128, activation='relu'))
model.add(BatchNormalization())
model.add(Dropout(0.5))
model.add(Dense(10, activation='softmax'))
```

MODULE VI - TRAINING MODEL

Here, the learning rate is changed as we move between layers, This is because having a larger value of learning rate can overshoot the gradient and does not update weights properly.

We change the learning rate as the model evolves

```
[39] reduce_lr = LearningRateScheduler(lambda x: 1e-3 * 0.9 ** x)
```

Early stopping is done to prevent overfitting. Patience level is set to 5 and the no of epochs is 75 and batch_size is 80. The

Training the Model

```
from tensorflow.keras.callbacks import EarlyStopping
early_stopping = EarlyStopping(monitor='loss', patience=5, verbose=1)
history = model.fit_generator(datagen.flow(X_train, y_train, batch_size = 80), epochs = 75,
                              validation_data = (X_val, y_val), verbose=1,
                              callbacks = [reduce_lr, early_stopping])
```

We use **Adam optimizer** for the model and loss function is **categorical_crossentropy.**

Trying to use Adam Propagation

```
model.compile(optimizer = 'Adam', loss = 'categorical_crossentropy', metrics = ['accuracy'])
```

MODULE VII - PLOTTING GRAPHS

Plotting graphs

```
[64] # Check how loss & mae went down
epoch_loss = history.history['loss']
epoch_val_loss = history.history['val_loss']
epoch_acc = history.history['accuracy']
epoch_val_acc = history.history['val_accuracy']

plt.figure(figsize=(20,6))
plt.subplot(1,2,1)
plt.plot(range(0,len(epoch_loss)), epoch_loss, 'b-', linewidth=2, label='Train Loss')
plt.plot(range(0,len(epoch_val_loss)), epoch_val_loss, 'r-', linewidth=2, label='Val Loss')
plt.title('Evolution of loss on train & validation datasets over epochs')
plt.legend(loc='best')

plt.subplot(1,2,2)
plt.plot(range(0,len(epoch_acc)), epoch_acc, 'b-', linewidth=2, label='Train Accuracy')
plt.plot(range(0,len(epoch_val_acc)), epoch_val_acc, 'r-', linewidth=2,label='Val Accuracy')
plt.title('Evolution of MAE on train & validation datasets over epochs')
plt.legend(loc='best')

plt.show()
```

MODULE VIII - PREDICTION AND EVALUATION

Evaluating model

```
[65] score = model.evaluate(X_test, y_test)

313/313 [==============================] - 2s 7ms/step - loss: 0.2002 - accuracy: 0.9340
```

```
[66] print('Loss of the model: {:.4f}'.format(score[0]))
     print('Accuracy of the model: {:.4f}'.format(score[1]))

Loss of the model: 0.2002
Accuracy of the model: 0.9340
```

Plotting confusion matrix

```
[68] from sklearn.metrics import confusion_matrix, classification_report
     # Predict the values from the validation dataset
     y_pred = model.predict(X_test)

     # Convert predictions classes to one hot vectors
     y_pred_classes = np.argmax(y_pred,axis = 1)
     y_true = np.argmax(y_test,axis = 1)

     confusion_mtx = confusion_matrix(y_true, y_pred_classes)
```

RESULTS

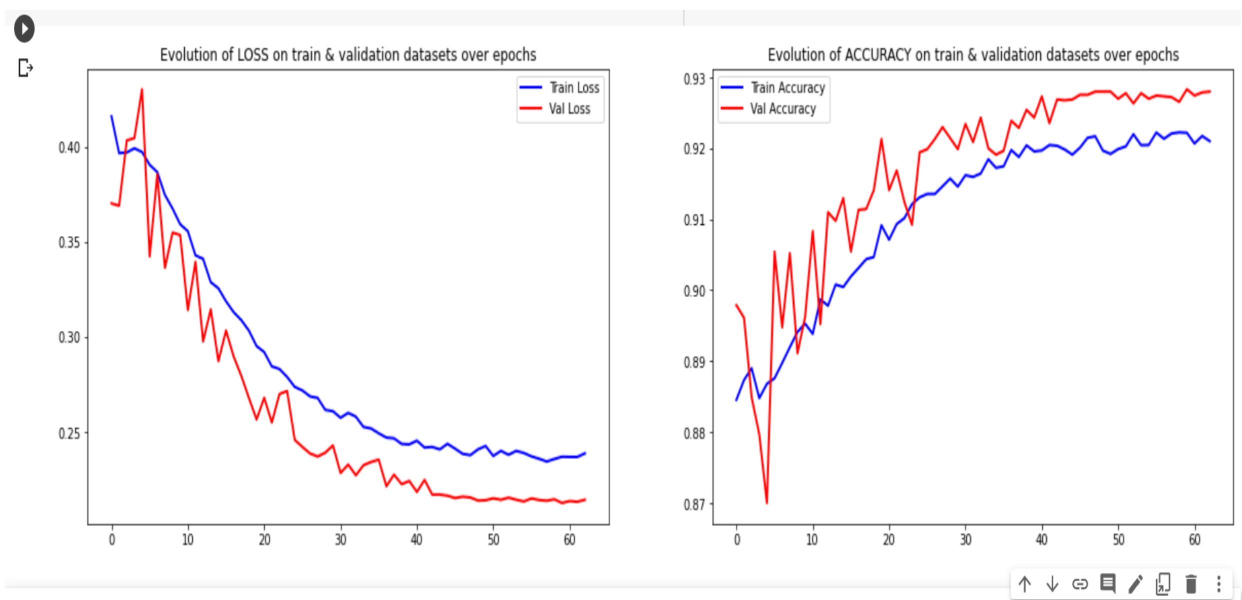
Model result

```
/usr/local/lib/python3.7/dist-packages/tensorflow/python/keras/engine/training.py:1844: UserWarning: `Model.fit_generator` is deprecated and will be
  warnings.warn('`Model.fit_generator` is deprecated and '
Epoch 1/75
638/638 [==============================] - 29s 45ms/step - loss: 0.4159 - accuracy: 0.8845 - val_loss: 0.3702 - val_accuracy: 0.8979
Epoch 2/75
638/638 [==============================] - 28s 44ms/step - loss: 0.3966 - accuracy: 0.8873 - val_loss: 0.3690 - val_accuracy: 0.8961
Epoch 3/75
638/638 [==============================] - 29s 45ms/step - loss: 0.3970 - accuracy: 0.8890 - val_loss: 0.4033 - val_accuracy: 0.8849
Epoch 4/75
638/638 [==============================] - 28s 44ms/step - loss: 0.3991 - accuracy: 0.8848 - val_loss: 0.4045 - val_accuracy: 0.8797
Epoch 5/75
638/638 [==============================] - 28s 44ms/step - loss: 0.3972 - accuracy: 0.8868 - val_loss: 0.4302 - val_accuracy: 0.8700
Epoch 6/75
638/638 [==============================] - 28s 45ms/step - loss: 0.3905 - accuracy: 0.8876 - val_loss: 0.3423 - val_accuracy: 0.9054
Epoch 7/75
638/638 [==============================] - 28s 44ms/step - loss: 0.3866 - accuracy: 0.8897 - val_loss: 0.3856 - val_accuracy: 0.8948
Epoch 8/75
638/638 [==============================] - 28s 44ms/step - loss: 0.3746 - accuracy: 0.8920 - val_loss: 0.3363 - val_accuracy: 0.9052
Epoch 9/75
638/638 [==============================] - 28s 44ms/step - loss: 0.3675 - accuracy: 0.8941 - val_loss: 0.3548 - val_accuracy: 0.8911
```

```
Epoch 40/75
638/638 [==============================] - 27s 42ms/step - loss: 0.2434 - accuracy: 0.9196 - val_loss: 0.2242 - val_accuracy: 0.9243
Epoch 41/75
638/638 [==============================] - 27s 42ms/step - loss: 0.2454 - accuracy: 0.9197 - val_loss: 0.2184 - val_accuracy: 0.9273
Epoch 42/75
638/638 [==============================] - 27s 43ms/step - loss: 0.2418 - accuracy: 0.9205 - val_loss: 0.2248 - val_accuracy: 0.9236
Epoch 43/75
638/638 [==============================] - 28s 44ms/step - loss: 0.2421 - accuracy: 0.9204 - val_loss: 0.2171 - val_accuracy: 0.9269
Epoch 44/75
638/638 [==============================] - 28s 45ms/step - loss: 0.2408 - accuracy: 0.9198 - val_loss: 0.2171 - val_accuracy: 0.9268
Epoch 45/75
638/638 [==============================] - 28s 44ms/step - loss: 0.2437 - accuracy: 0.9191 - val_loss: 0.2164 - val_accuracy: 0.9269
```

```
Epoch 57/75
638/638 [==============================] - 28s 44ms/step - loss: 0.2359 - accuracy: 0.9214 - val_loss: 0.2142 - val_accuracy: 0.9273
Epoch 58/75
638/638 [==============================] - 28s 44ms/step - loss: 0.2344 - accuracy: 0.9221 - val_loss: 0.2138 - val_accuracy: 0.9272
Epoch 59/75
638/638 [==============================] - 28s 44ms/step - loss: 0.2358 - accuracy: 0.9223 - val_loss: 0.2146 - val_accuracy: 0.9266
Epoch 60/75
638/638 [==============================] - 28s 44ms/step - loss: 0.2369 - accuracy: 0.9222 - val_loss: 0.2126 - val_accuracy: 0.9283
Epoch 61/75
638/638 [==============================] - 28s 44ms/step - loss: 0.2368 - accuracy: 0.9207 - val_loss: 0.2136 - val_accuracy: 0.9274
Epoch 62/75
638/638 [==============================] - 28s 44ms/step - loss: 0.2368 - accuracy: 0.9217 - val_loss: 0.2133 - val_accuracy: 0.9279
Epoch 63/75
638/638 [==============================] - 28s 44ms/step - loss: 0.2387 - accuracy: 0.9210 - val_loss: 0.2143 - val_accuracy: 0.9280
Epoch 00063: early stopping
```

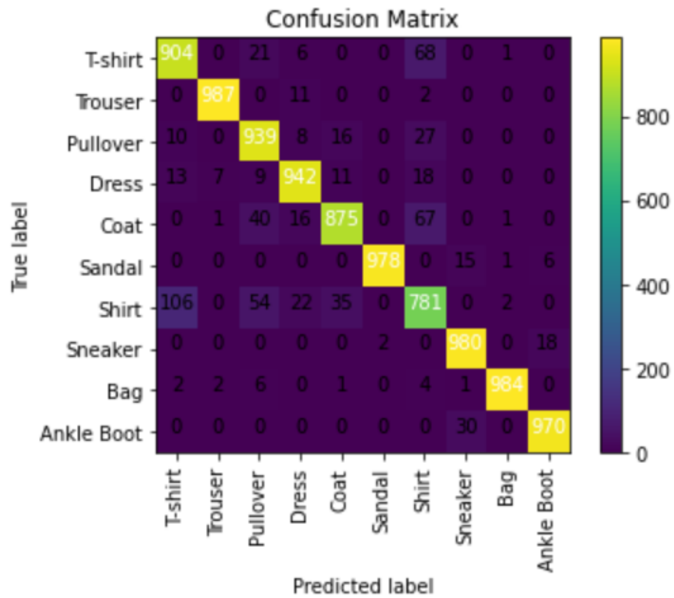
We can infer that the accuracy has improved steadily from 88% to around **92%**.We also observe that **Early Stopping** has taken place at **epoch 63** and the model is not trained any further.



From the above graphs, we can observe that there is not much generalization error and thus our model will not overfit.

Confusion Matrix

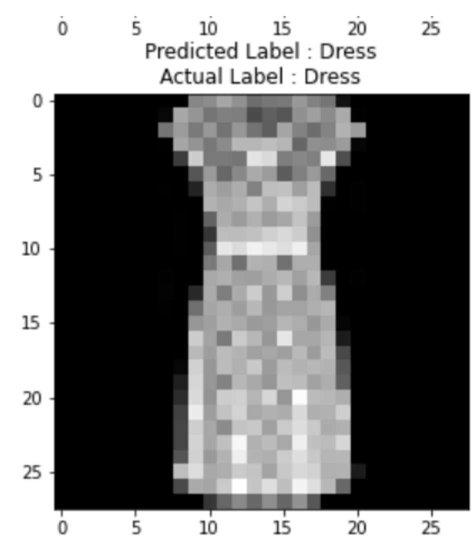
```
Text(0.5, 0, 'Predicted label')
```

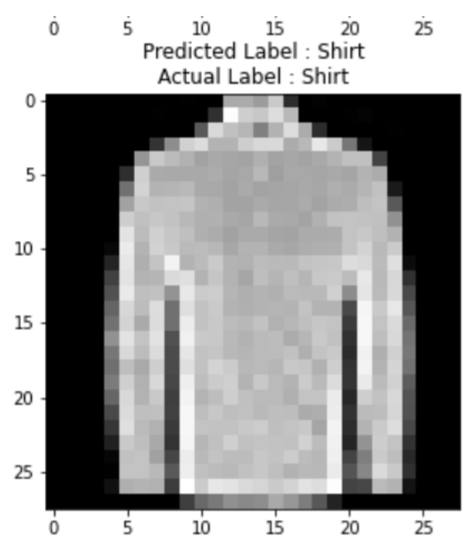


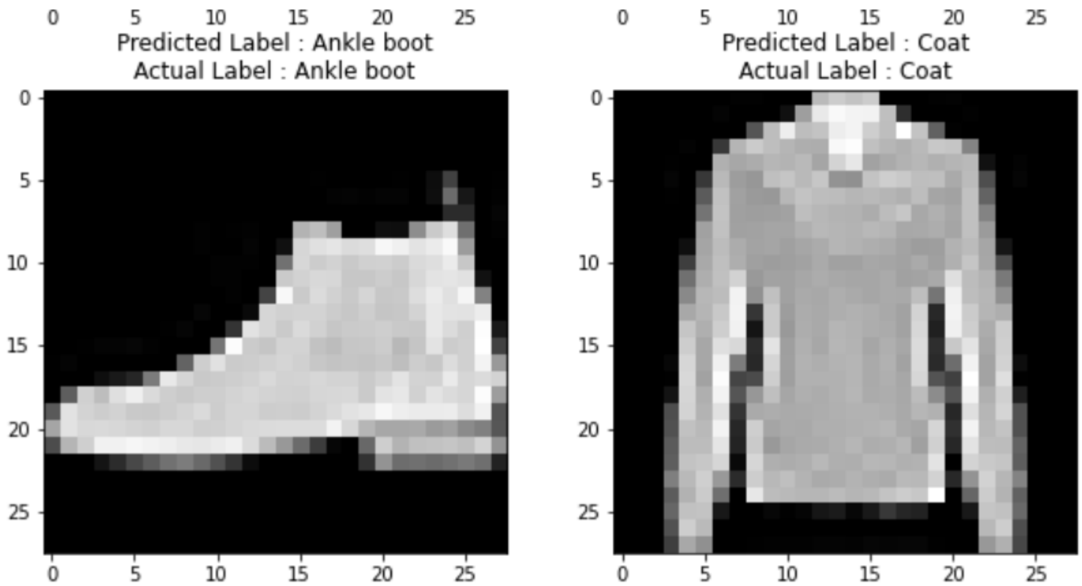
From the confusion matrix, we can conclude that Shirts are mostly misclassified as Tshirt. Nearly 104 shirts are wrongly predicted as Tshirt.

CONCLUSION

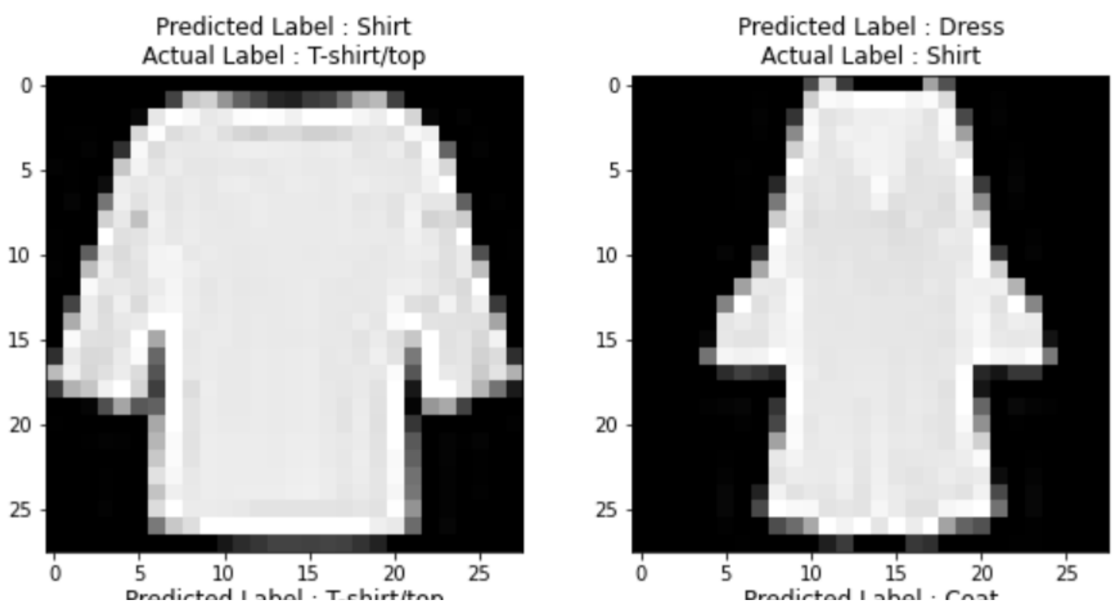
Examples of correctly predicted classes







Examples of incorrectly predicted classes



We have created our own CNN model with many regularization parameters (Batch Normalisation, Data Augmentation, Dropout, L2 Norm Regularisation, Early Stopping) and have obtained an accuracy of 92.4 %. With further more training, we can achieve better results.

REFERENCES

https://paperswithcode.com/dataset/fashion-mnist
https://rviews.rstudio.com/2019/11/11/a-comparison-of-methods-for-predicting-clothing-classes-using-the-fashion-mnist-dataset-in-rstudio-and-python-part-1/