CISC 867: Project 2
Fashion MNIST

By

Lomai Mohamed Saadeldin
ID: 20398049
Supervised by: Prof. Hazem Abbas

PROPLEM FORMULATION

- **Data Description:**
  - 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.
  - Zalando intends Fashion-MNIST to serve as a direct drop-in replacement for the original MNIST dataset for benchmarking machine learning algorithms. It shares the same image size and structure of training and testing splits.
  - 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 Problems:**

  we will use this data using a CNN neural network architecture.

- **Input:**

  (Fashion-MNIST) The training and test data set 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.

- **Output:**

  Predict the type of clothes.

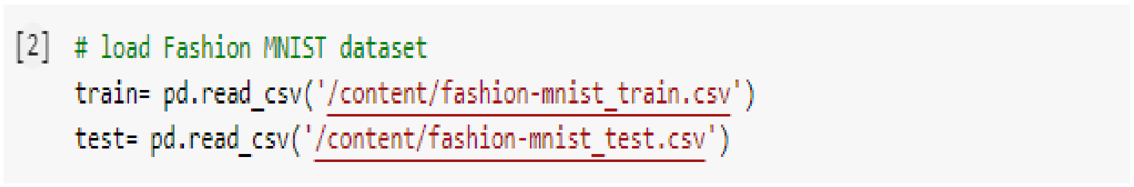
- **Challenges**
    - Dealing with duplication data
    - Reshape the data arrays to have a single color channel.
    - Prepare pixel data by Normalization.
    - Build LeNet-5 Model with best Hypermeters
    - Transfer learning

DATA EXPLORATORY

- **Import Some Modules to deal with data set**
- **Load the data set**
- **We can load data set across three ways:**

1. The tf.keras.datasets module provide a few toy datasets (already-vectorized, in Numpy format) that can be used for debugging a model or creating simple code examples.
2. Download the data set from Kaggle and load it on colab by pandas.
3. Import data from kaggle directly.

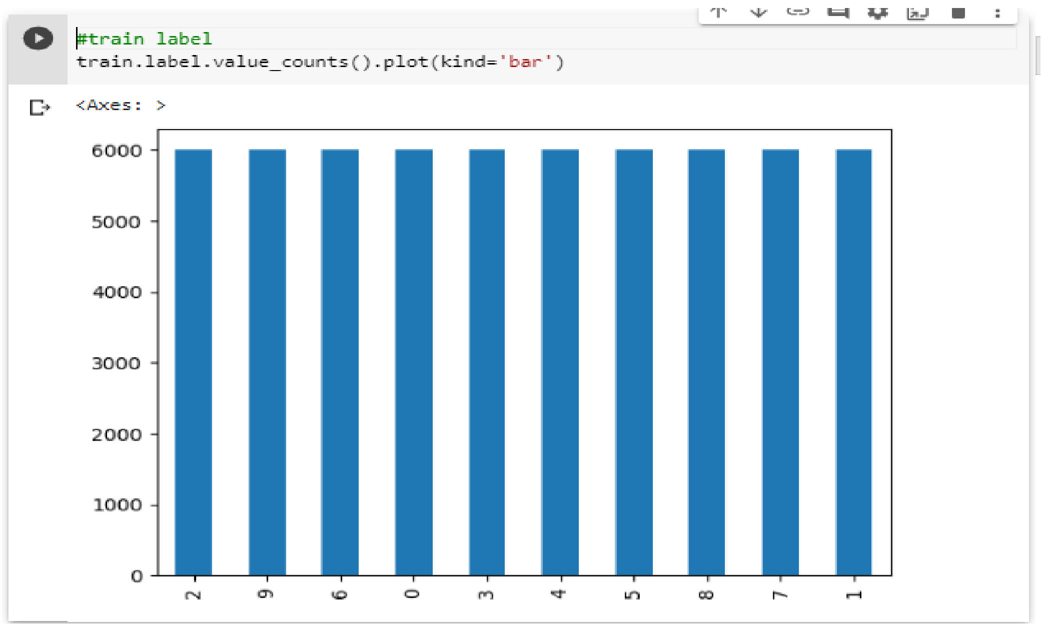
   I choice the second way.



```
[2]  # load Fashion MNIST dataset
     train= pd.read_csv('/content/fashion-mnist_train.csv')
     test= pd.read_csv('/content/fashion-mnist_test.csv')
```

DATA VISUALIZATION

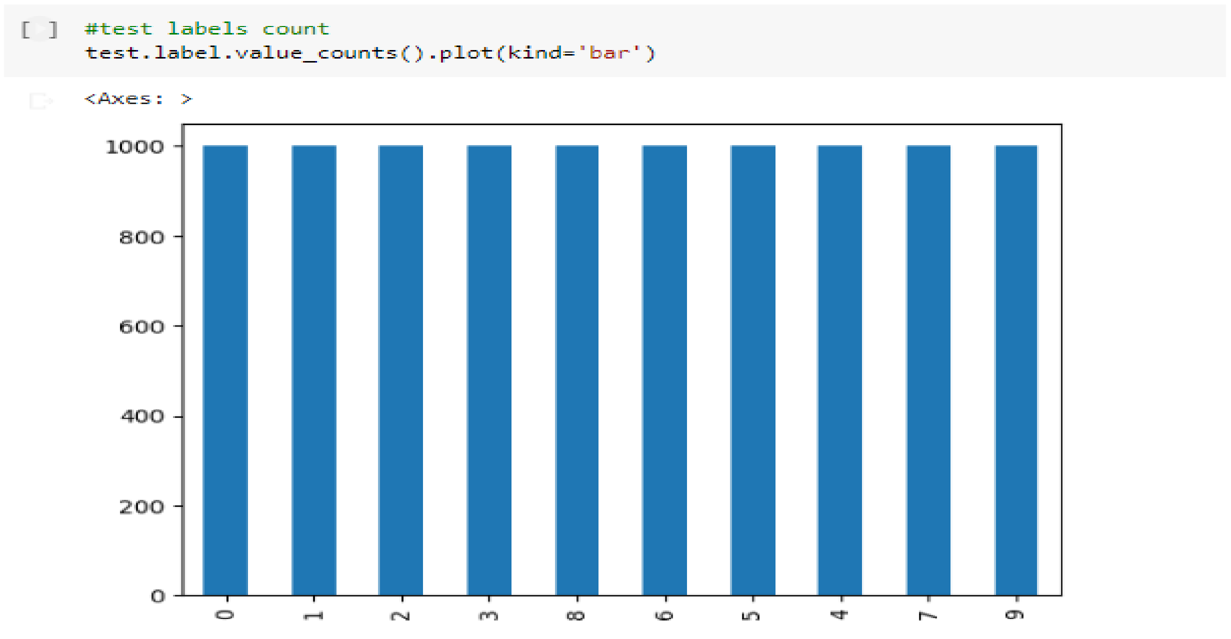
- **Number of labels for each class on train data.**



```
#train label
train.label.value_counts().plot(kind='bar')
```

```
<Axes: >
```

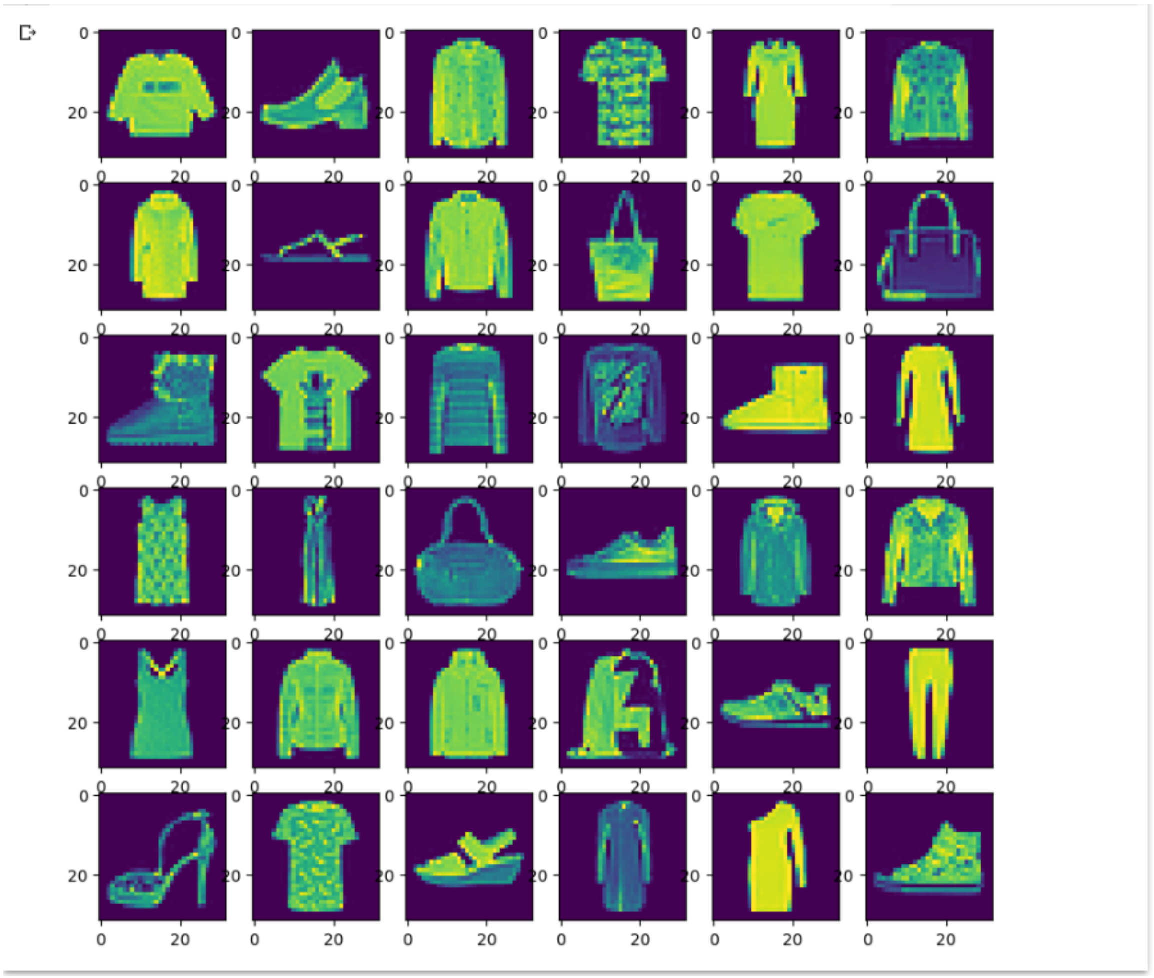
- **Number of labels for each class on test data.**



```
#test labels count
test.label.value_counts().plot(kind='bar')
```

```
<Axes: >
```

- **Visualize Some of Images. (after spilt the data set)**



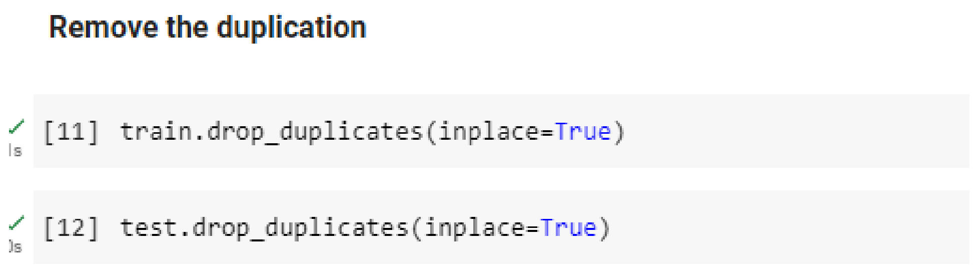
DATA PREPARATION

- Check the data for missing values:

  There is no missing valued on train or test set

- Check the duplication

  Remove the duplication from train and test data



Remove the duplication

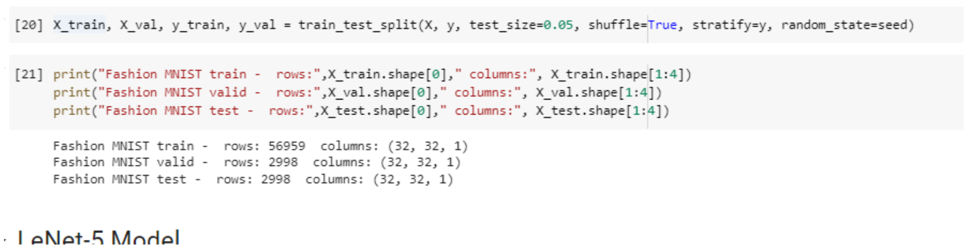
```
[11] train.drop_duplicates(inplace=True)
```

```
[12] test.drop_duplicates(inplace=True)
```

- Split Training data
- Spilt test Data
- Reshape data

  we can load the images and reshape the data arrays to have a single color channel.

- Prepare Pixel Data (Normalization)
- Split Training Data into Training and Validation.



```
[20] X_train, X_val, y_train, y_val = train_test_split(X, y, test_size=0.05, shuffle=True, stratify=y, random_state=seed)
```

```
[21] print("Fashion MNIST train -  rows:",X_train.shape[0]," columns:", X_train.shape[1:4])
     print("Fashion MNIST valid -  rows:",X_val.shape[0]," columns:", X_val.shape[1:4])
     print("Fashion MNIST test -  rows:",X_test.shape[0]," columns:", X_test.shape[1:4])

     Fashion MNIST train -  rows: 56959  columns: (32, 32, 1)
     Fashion MNIST valid -  rows: 2998  columns: (32, 32, 1)
     Fashion MNIST test -  rows: 2998  columns: (32, 32, 1)
```

LeNet-5 Model

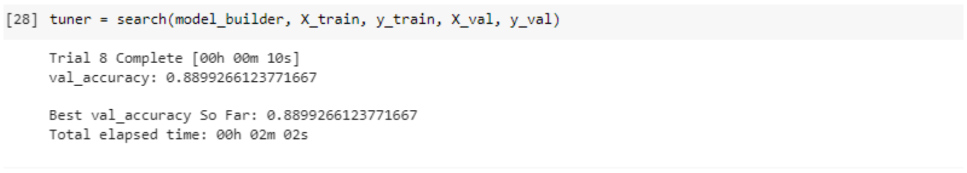
- **LeNet-5 network**

  build LeNet-5 Model: it's simple and straightforward architecture. It is a multi-layer convolution neural network for image classification.

- **LeNet-5 model with best Hyper parameters**

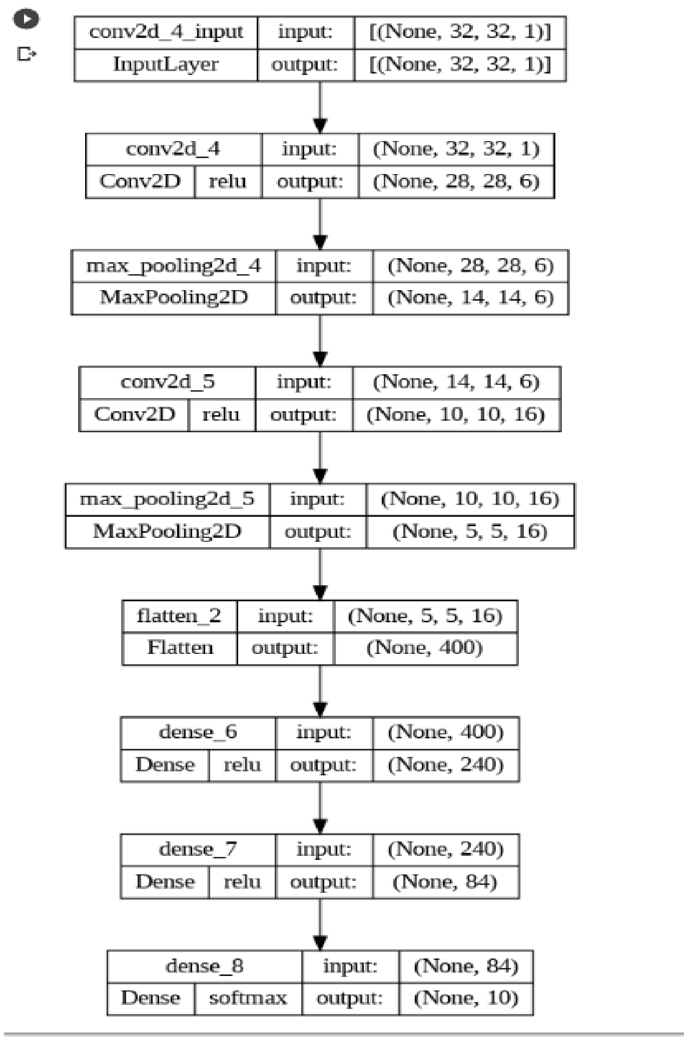
  Modification hyper parameters to get the best performance you can achieve.

  the Validation accuracy:

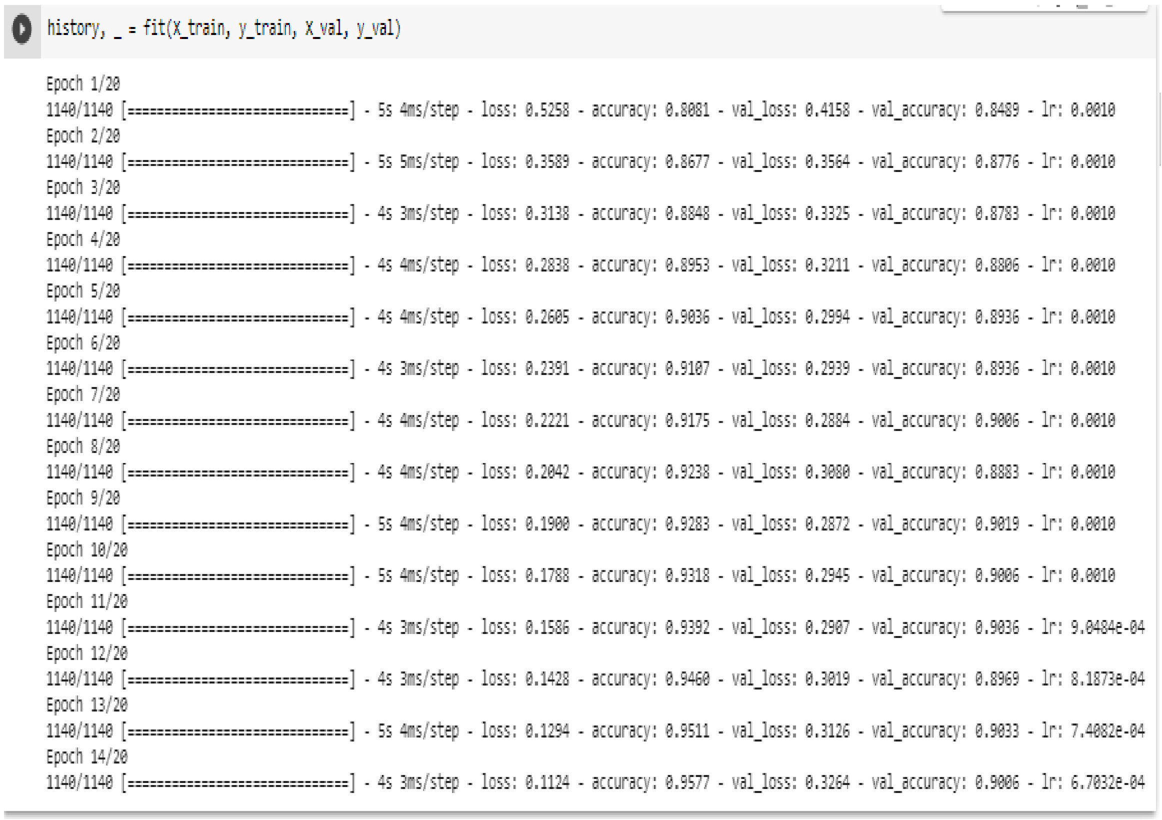


```
[28] tuner = search(model_builder, X_train, y_train, X_val, y_val)

     Trial 8 Complete [00h 00m 10s]
     val_accuracy: 0.8899266123771667

     Best val_accuracy So Far: 0.8899266123771667
     Total elapsed time: 00h 02m 02s
```

- **Build LeNet-5 model with optimal Hyper parameters**


conv2d_4_input | input: | [(None, 32, 32, 1)]
InputLayer | output: | [(None, 32, 32, 1)]
conv2d_4 | input: | (None, 32, 32, 1)
Conv2D | relu | output: | (None, 28, 28, 6)
max_pooling2d_4 | input: | (None, 28, 28, 6)
MaxPooling2D | output: | (None, 14, 14, 6)
conv2d_5 | input: | (None, 14, 14, 6)
Conv2D | relu | output: | (None, 10, 10, 16)
max_pooling2d_5 | input: | (None, 10, 10, 16)
MaxPooling2D | output: | (None, 5, 5, 16)
flatten_2 | input: | (None, 5, 5, 16)
Flatten | output: | (None, 400)
dense_6 | input: | (None, 400)
Dense | relu | output: | (None, 240)
dense_7 | input: | (None, 240)
Dense | relu | output: | (None, 84)
dense_8 | input: | (None, 84)
Dense | softmax | output: | (None, 10)

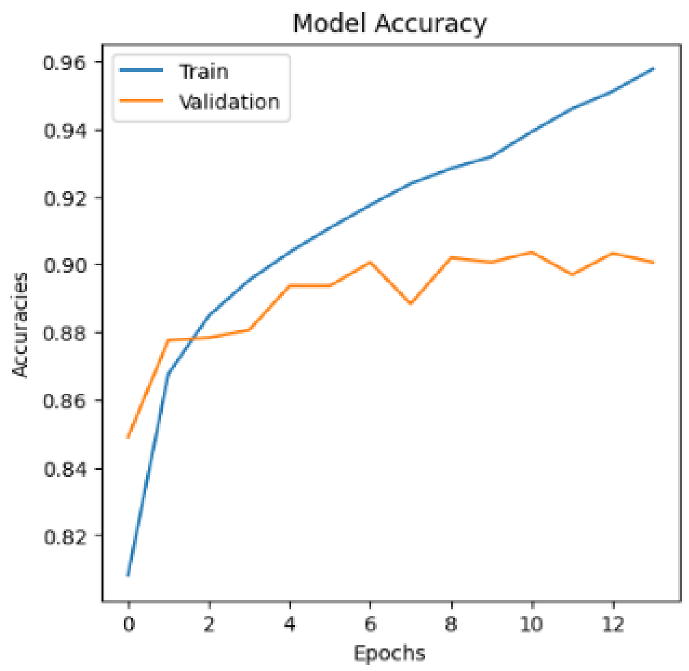
- **Train the LeNet-5 model with best Hyper parameters.**

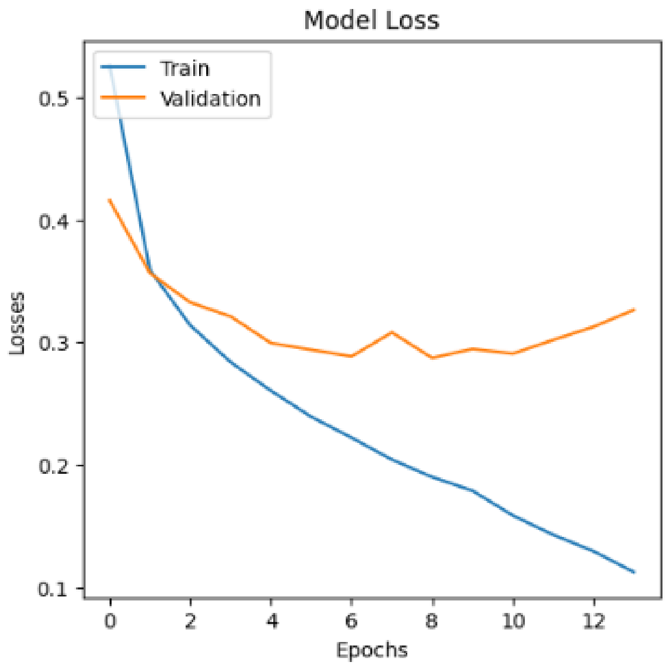


```
history, _ = fit(X_train, y_train, X_val, y_val)
```

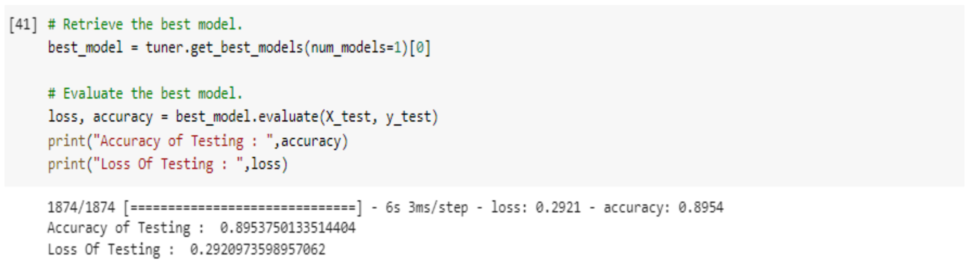
```
Epoch 1/20
1140/1140 [==============================] - 5s 4ms/step - loss: 0.5258 - accuracy: 0.8081 - val_loss: 0.4158 - val_accuracy: 0.8489 - lr: 0.0010
Epoch 2/20
1140/1140 [==============================] - 5s 5ms/step - loss: 0.3589 - accuracy: 0.8677 - val_loss: 0.3564 - val_accuracy: 0.8776 - lr: 0.0010
Epoch 3/20
1140/1140 [==============================] - 4s 3ms/step - loss: 0.3138 - accuracy: 0.8848 - val_loss: 0.3325 - val_accuracy: 0.8783 - lr: 0.0010
Epoch 4/20
1140/1140 [==============================] - 4s 4ms/step - loss: 0.2838 - accuracy: 0.8953 - val_loss: 0.3211 - val_accuracy: 0.8806 - lr: 0.0010
Epoch 5/20
1140/1140 [==============================] - 4s 4ms/step - loss: 0.2605 - accuracy: 0.9036 - val_loss: 0.2994 - val_accuracy: 0.8936 - lr: 0.0010
Epoch 6/20
1140/1140 [==============================] - 4s 3ms/step - loss: 0.2391 - accuracy: 0.9107 - val_loss: 0.2939 - val_accuracy: 0.8936 - lr: 0.0010
Epoch 7/20
1140/1140 [==============================] - 4s 4ms/step - loss: 0.2221 - accuracy: 0.9175 - val_loss: 0.2884 - val_accuracy: 0.9006 - lr: 0.0010
Epoch 8/20
1140/1140 [==============================] - 4s 4ms/step - loss: 0.2042 - accuracy: 0.9238 - val_loss: 0.3080 - val_accuracy: 0.8883 - lr: 0.0010
Epoch 9/20
1140/1140 [==============================] - 5s 4ms/step - loss: 0.1900 - accuracy: 0.9283 - val_loss: 0.2872 - val_accuracy: 0.9019 - lr: 0.0010
Epoch 10/20
1140/1140 [==============================] - 5s 4ms/step - loss: 0.1788 - accuracy: 0.9318 - val_loss: 0.2945 - val_accuracy: 0.9006 - lr: 0.0010
Epoch 11/20
1140/1140 [==============================] - 4s 3ms/step - loss: 0.1586 - accuracy: 0.9392 - val_loss: 0.2907 - val_accuracy: 0.9036 - lr: 9.0484e-04
Epoch 12/20
1140/1140 [==============================] - 4s 3ms/step - loss: 0.1428 - accuracy: 0.9460 - val_loss: 0.3019 - val_accuracy: 0.8969 - lr: 8.1873e-04
Epoch 13/20
1140/1140 [==============================] - 5s 4ms/step - loss: 0.1294 - accuracy: 0.9511 - val_loss: 0.3126 - val_accuracy: 0.9033 - lr: 7.4082e-04
Epoch 14/20
1140/1140 [==============================] - 4s 3ms/step - loss: 0.1124 - accuracy: 0.9577 - val_loss: 0.3264 - val_accuracy: 0.9006 - lr: 6.7032e-04
```

- **Plot graph to compare Accuracy and loss.**





- **Evaluate the model Using Test Data:**

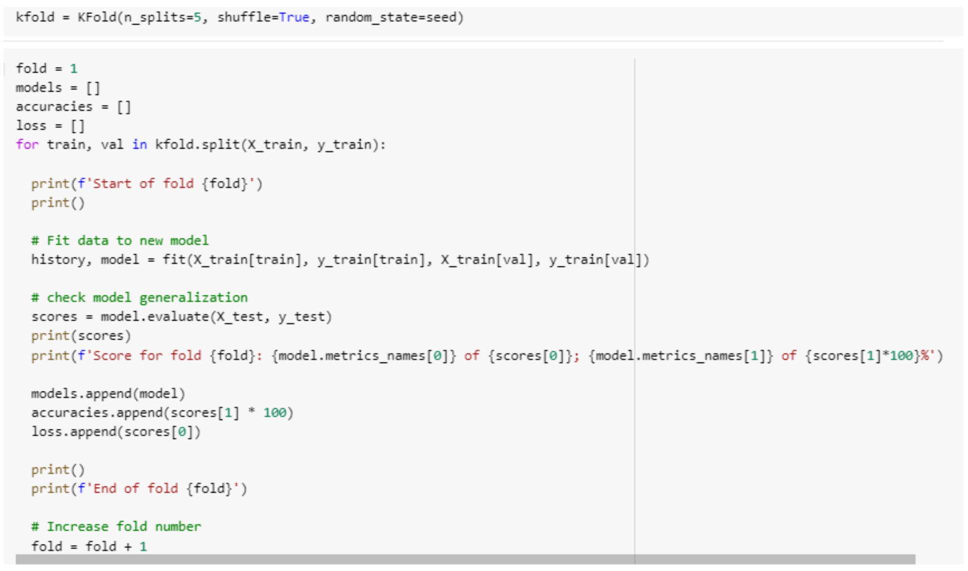


```
[41] # Retrieve the best model.
     best_model = tuner.get_best_models(num_models=1)[0]

     # Evaluate the best model.
     loss, accuracy = best_model.evaluate(X_test, y_test)
     print("Accuracy of Testing : ",accuracy)
     print("Loss Of Testing : ",loss)

     1874/1874 [==============================] - 6s 3ms/step - loss: 0.2921 - accuracy: 0.8954
     Accuracy of Testing :  0.8953750133514404
     Loss Of Testing :  0.2920973598957062
```

Accuracy of Testing: 0.8953750133514404

Loss of Testing: 0.2920973598957062

- **Cross-Validation (**using k-fold Cross validation).

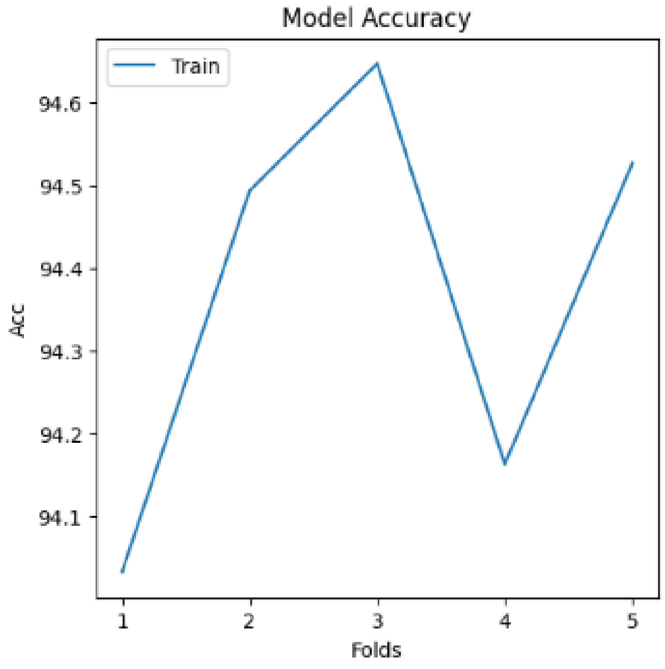


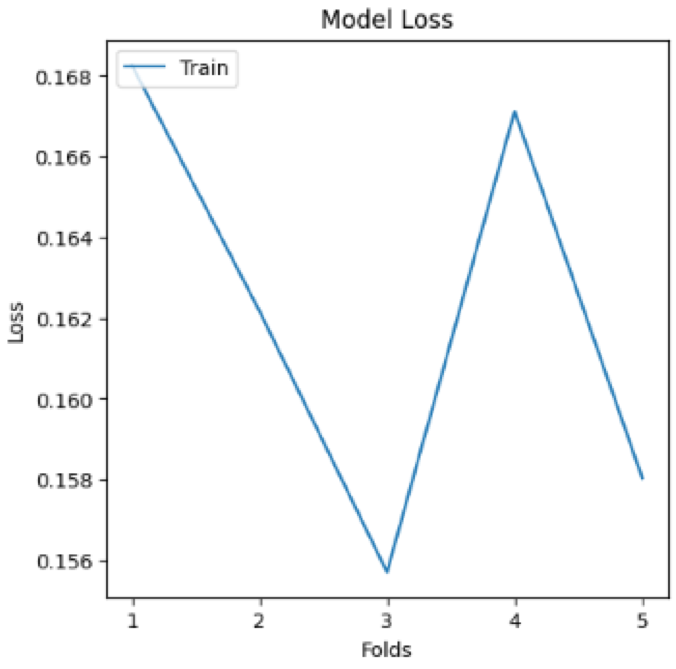
```
kfold = KFold(n_splits=5, shuffle=True, random_state=seed)
```

```
fold = 1
models = []
accuracies = []
loss = []
for train, val in kfold.split(X_train, y_train):

  print(f'Start of fold {fold}')
  print()

  # Fit data to new model
  history, model = fit(X_train[train], y_train[train], X_train[val], y_train[val])

  # check model generalization
  scores = model.evaluate(X_test, y_test)
  print(scores)
  print(f'Score for fold {fold}: {model.metrics_names[0]} of {scores[0]}; {model.metrics_names[1]} of {scores[1]*100}%')

  models.append(model)
  accuracies.append(scores[1] * 100)
  loss.append(scores[0])

  print()
  print(f'End of fold {fold}')

  # Increase fold number
  fold = fold + 1
```

- Score for fold 1: loss of 0.16825911402702332; accuracy of 94.03238892555237%
- Score for fold 2: loss of 0.16216780245304108; accuracy of 94.49272155761719%
- Score for fold 3: loss of 0.15570496022701263; accuracy of 94.64616179466248%
- Score for fold 4: loss of 0.16711124777793884; accuracy of 94.16248202323914%
- Score for fold 5: loss of 0.15803766250610352; accuracy of 94.5260763168335%

- **Plot graph to compare Accuracy and loss.**





- **Why do you think LeNet-5 further improves the accuracy if any at all? And if it doesn't, why not?**
- This network was a good introduction into the world of neural networks and is really simple to understand. It works well for character recognition images.
- It consists of 3 convolution layers and 3 pooling layers:

  -Pooling is used to reduce the number of features that we get after we have run our filters over the images. The filters cause us to get a large number of features compared to the image itself. So pooling is used to get a better generalized representation of the same.

  - “Convolution” here represents the understanding of features within an image. Extracting these features entails the entire process of convolution.

  However,
- This model was more specifically built for a certain use case. it does not do as well with color images. Most image recognition problems would require RGB images for better recognition.
- Since the model isn’t very deep, it struggles to scan for all features thus producing poor performing models. If the Neural Network isn’t fed with enough features from the training images, then it would be difficult for the model to generalize and create an accurate model.

- **Transfer learning**

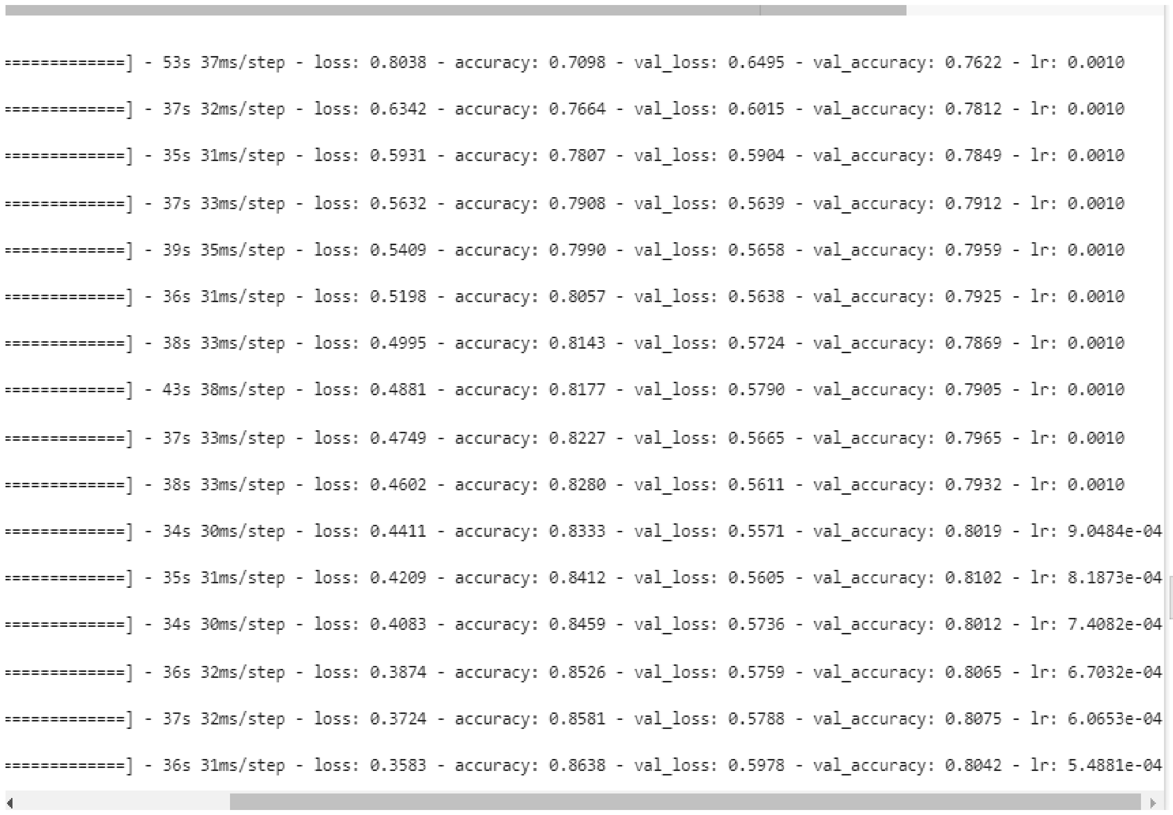
  We need to resize the MNIST data set.

  the minimum size actually depends on the ImageNet model.

- **Build RESNet152V2 model**

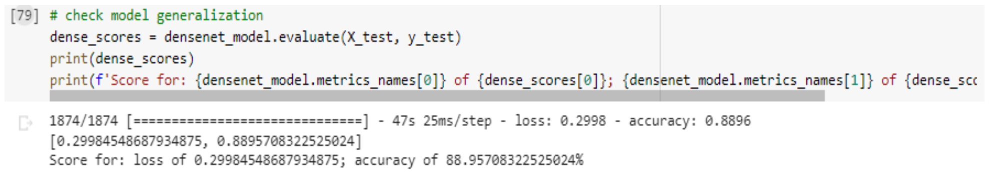
  It can help to gain acceptable accuracy faster than a traditional CNN.

- **Display RESNet152V2 model**
- **Train RESNet152V2 model**



```
=============] - 53s 37ms/step - loss: 0.8038 - accuracy: 0.7098 - val_loss: 0.6495 - val_accuracy: 0.7622 - lr: 0.0010
=============] - 37s 32ms/step - loss: 0.6342 - accuracy: 0.7664 - val_loss: 0.6015 - val_accuracy: 0.7812 - lr: 0.0010
=============] - 35s 31ms/step - loss: 0.5931 - accuracy: 0.7807 - val_loss: 0.5904 - val_accuracy: 0.7849 - lr: 0.0010
=============] - 37s 33ms/step - loss: 0.5632 - accuracy: 0.7908 - val_loss: 0.5639 - val_accuracy: 0.7912 - lr: 0.0010
=============] - 39s 35ms/step - loss: 0.5409 - accuracy: 0.7990 - val_loss: 0.5658 - val_accuracy: 0.7959 - lr: 0.0010
=============] - 36s 31ms/step - loss: 0.5198 - accuracy: 0.8057 - val_loss: 0.5638 - val_accuracy: 0.7925 - lr: 0.0010
=============] - 38s 33ms/step - loss: 0.4995 - accuracy: 0.8143 - val_loss: 0.5724 - val_accuracy: 0.7869 - lr: 0.0010
=============] - 43s 38ms/step - loss: 0.4881 - accuracy: 0.8177 - val_loss: 0.5790 - val_accuracy: 0.7905 - lr: 0.0010
=============] - 37s 33ms/step - loss: 0.4749 - accuracy: 0.8227 - val_loss: 0.5665 - val_accuracy: 0.7965 - lr: 0.0010
=============] - 38s 33ms/step - loss: 0.4602 - accuracy: 0.8280 - val_loss: 0.5611 - val_accuracy: 0.7932 - lr: 0.0010
=============] - 34s 30ms/step - loss: 0.4411 - accuracy: 0.8333 - val_loss: 0.5571 - val_accuracy: 0.8019 - lr: 9.0484e-04
=============] - 35s 31ms/step - loss: 0.4209 - accuracy: 0.8412 - val_loss: 0.5605 - val_accuracy: 0.8102 - lr: 8.1873e-04
=============] - 34s 30ms/step - loss: 0.4083 - accuracy: 0.8459 - val_loss: 0.5736 - val_accuracy: 0.8012 - lr: 7.4082e-04
=============] - 36s 32ms/step - loss: 0.3874 - accuracy: 0.8526 - val_loss: 0.5759 - val_accuracy: 0.8065 - lr: 6.7032e-04
=============] - 37s 32ms/step - loss: 0.3724 - accuracy: 0.8581 - val_loss: 0.5788 - val_accuracy: 0.8075 - lr: 6.0653e-04
=============] - 36s 31ms/step - loss: 0.3583 - accuracy: 0.8638 - val_loss: 0.5978 - val_accuracy: 0.8042 - lr: 5.4881e-04
```

- **Evaluate RESNet152V2 model Using Test Data:**

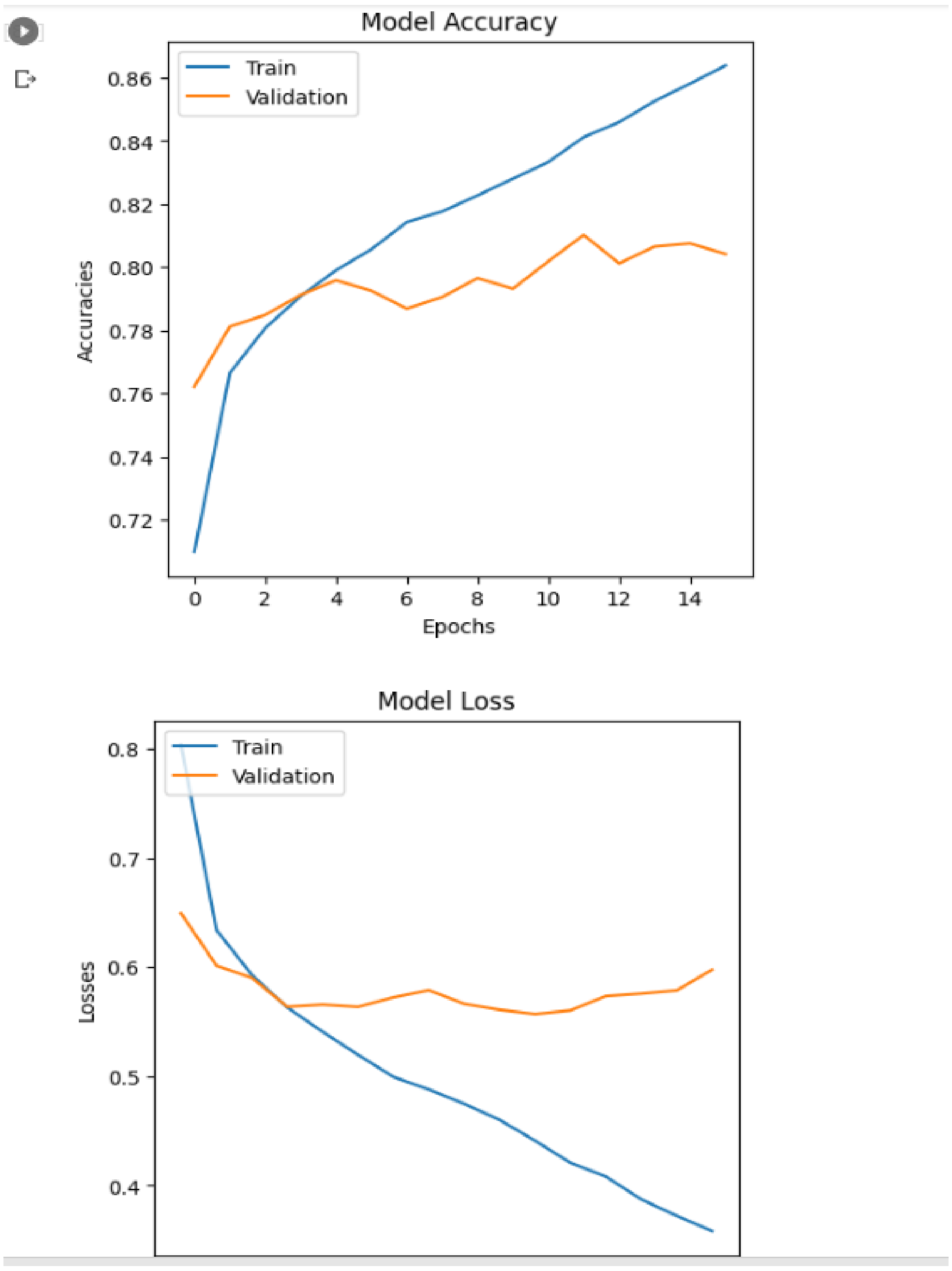


```
[79] # check model generalization
     dense_scores = densenet_model.evaluate(X_test, y_test)
     print(dense_scores)
     print(f'Score for: {densenet_model.metrics_names[0]} of {dense_scores[0]}; {densenet_model.metrics_names[1]} of {dense_scc

     1874/1874 [==============================] - 47s 25ms/step - loss: 0.2998 - accuracy: 0.8896
     [0.29984548687934875, 0.8895708322525024]
     Score for: loss of 0.29984548687934875; accuracy of 88.95708322525024%
```

Accuracy =88.95708322525024%

Loss =0.29984548687934875

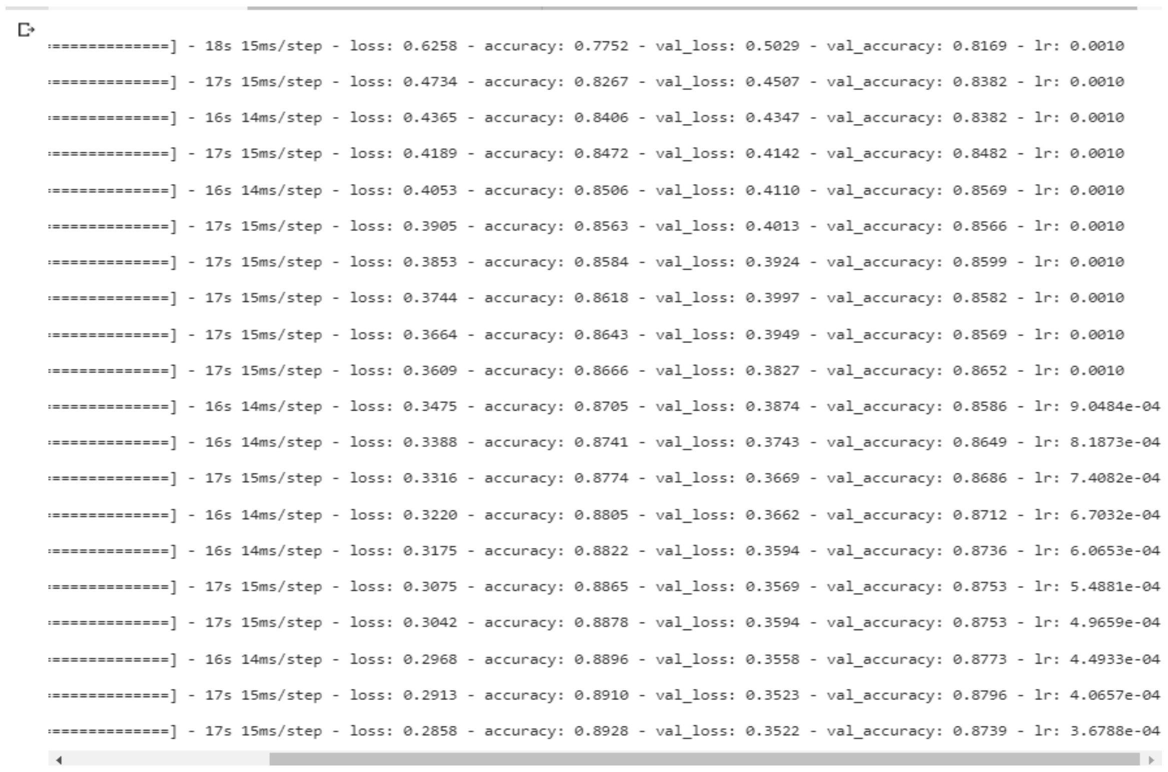
- **Plot graph to compare Accuracy and loss:**



- **Build VGG19 model**

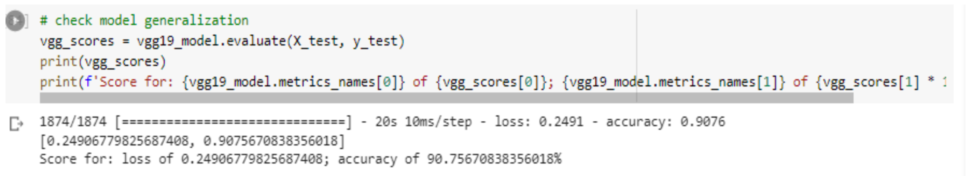
  Compared with existing methods, this method has faster training speed, fewer training samples per time, and higher accuracy.

- **Display VGG19 model**
- **Train VGG19 model**



```
:=============] - 18s 15ms/step - loss: 0.6258 - accuracy: 0.7752 - val_loss: 0.5029 - val_accuracy: 0.8169 - lr: 0.0010
:=============] - 17s 15ms/step - loss: 0.4734 - accuracy: 0.8267 - val_loss: 0.4507 - val_accuracy: 0.8382 - lr: 0.0010
:=============] - 16s 14ms/step - loss: 0.4365 - accuracy: 0.8406 - val_loss: 0.4347 - val_accuracy: 0.8382 - lr: 0.0010
:=============] - 17s 15ms/step - loss: 0.4189 - accuracy: 0.8472 - val_loss: 0.4142 - val_accuracy: 0.8482 - lr: 0.0010
:=============] - 16s 14ms/step - loss: 0.4053 - accuracy: 0.8506 - val_loss: 0.4110 - val_accuracy: 0.8569 - lr: 0.0010
:=============] - 17s 15ms/step - loss: 0.3905 - accuracy: 0.8563 - val_loss: 0.4013 - val_accuracy: 0.8566 - lr: 0.0010
:=============] - 17s 15ms/step - loss: 0.3853 - accuracy: 0.8584 - val_loss: 0.3924 - val_accuracy: 0.8599 - lr: 0.0010
:=============] - 17s 15ms/step - loss: 0.3744 - accuracy: 0.8618 - val_loss: 0.3997 - val_accuracy: 0.8582 - lr: 0.0010
:=============] - 17s 15ms/step - loss: 0.3664 - accuracy: 0.8643 - val_loss: 0.3949 - val_accuracy: 0.8569 - lr: 0.0010
:=============] - 17s 15ms/step - loss: 0.3609 - accuracy: 0.8666 - val_loss: 0.3827 - val_accuracy: 0.8652 - lr: 0.0010
:=============] - 16s 14ms/step - loss: 0.3475 - accuracy: 0.8705 - val_loss: 0.3874 - val_accuracy: 0.8586 - lr: 9.0484e-04
:=============] - 16s 14ms/step - loss: 0.3388 - accuracy: 0.8741 - val_loss: 0.3743 - val_accuracy: 0.8649 - lr: 8.1873e-04
:=============] - 17s 15ms/step - loss: 0.3316 - accuracy: 0.8774 - val_loss: 0.3669 - val_accuracy: 0.8686 - lr: 7.4082e-04
:=============] - 16s 14ms/step - loss: 0.3220 - accuracy: 0.8805 - val_loss: 0.3662 - val_accuracy: 0.8712 - lr: 6.7032e-04
:=============] - 16s 14ms/step - loss: 0.3175 - accuracy: 0.8822 - val_loss: 0.3594 - val_accuracy: 0.8736 - lr: 6.0653e-04
:=============] - 17s 15ms/step - loss: 0.3075 - accuracy: 0.8865 - val_loss: 0.3569 - val_accuracy: 0.8753 - lr: 5.4881e-04
:=============] - 17s 15ms/step - loss: 0.3042 - accuracy: 0.8878 - val_loss: 0.3594 - val_accuracy: 0.8753 - lr: 4.9659e-04
:=============] - 16s 14ms/step - loss: 0.2968 - accuracy: 0.8896 - val_loss: 0.3558 - val_accuracy: 0.8773 - lr: 4.4933e-04
:=============] - 17s 15ms/step - loss: 0.2913 - accuracy: 0.8910 - val_loss: 0.3523 - val_accuracy: 0.8796 - lr: 4.0657e-04
:=============] - 17s 15ms/step - loss: 0.2858 - accuracy: 0.8928 - val_loss: 0.3522 - val_accuracy: 0.8739 - lr: 3.6788e-04
```

- **Evaluate VGG19 model on test data set:**



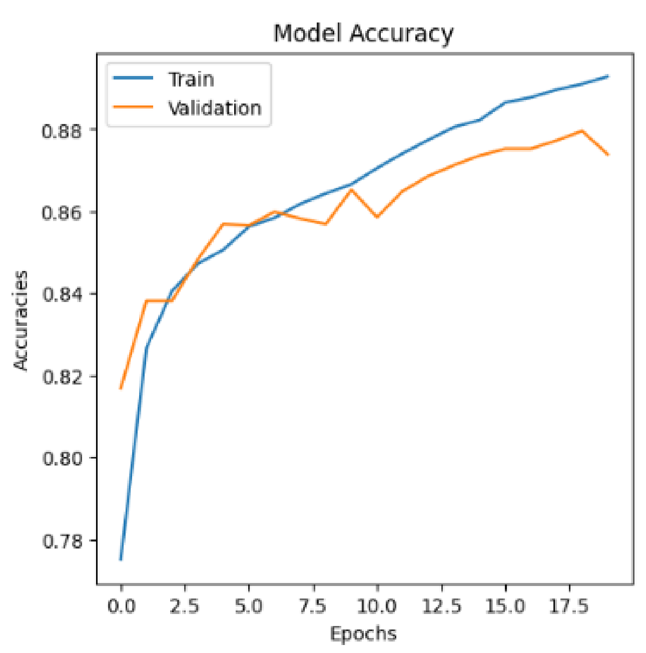
```
# check model generalization
vgg_scores = vgg19_model.evaluate(X_test, y_test)
print(vgg_scores)
print(f'Score for: {vgg19_model.metrics_names[0]} of {vgg_scores[0]}; {vgg19_model.metrics_names[1]} of {vgg_scores[1] * 1
```

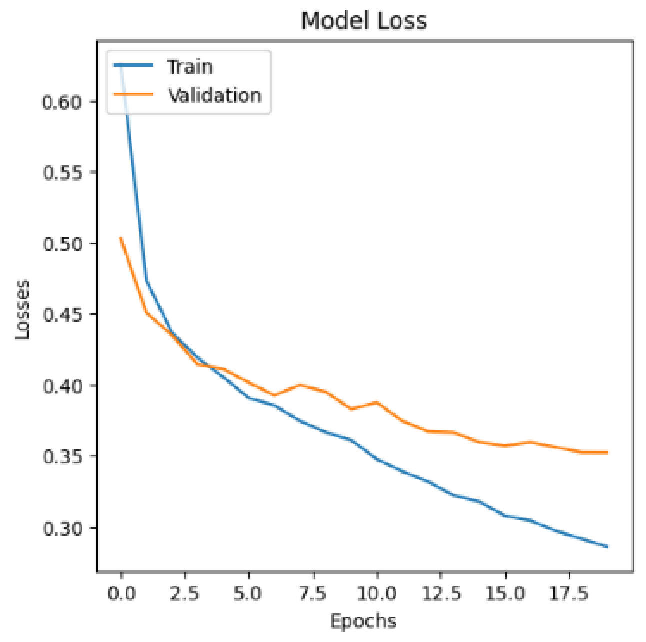
```
1874/1874 [==============================] - 20s 10ms/step - loss: 0.2491 - accuracy: 0.9076
[0.24906779825687408, 0.9075670838356018]
Score for: loss of 0.24906779825687408; accuracy of 90.75670838356018%
```

Accuracy =90.75670838356018%

Loss = 0.24906779825687408

- **Plot graph to compare Accuracy and loss:**





CONCLUSION

THE VGG19 MODEL GOT **THE BEST** ACCURACY

(Accuracy =90.75670838356018%)

(Loss = 0.24906779825687408)

THEN, THE **LENET-5 MODEL GOT ACCURACY**

(Accuracy =89. 53750133514404%)

(Loss = 0.2920973598957062)

THEN, THE **RESNET152V2 MODEL GOT ACCURACY**

(Accuracy =88.95708322525024%)

(Loss = 0.29984548687934875)

But, if I used the cross validation LeNet-5 model got the best Accuracy ≈ 94 %

REFERENCES

- **https://www.kaggle.com/datasets/zalando-research/fashionmnist**
- **https://www.analyticsvidhya.com/blog/2021/03/the-architecture-of-lenet-5/#:~:text=Lenet%2D5%20is%20one%20of,handwritten%20and%20machine%2Dprinted%20characters.**
- **https://d2l.ai/chapter_convolutional-neural-networks/lenet.html**
- **https://www.tensorflow.org/api_docs/python/tf/keras/preprocessing/image/ImageDataGenerator**
- **https://www.tensorflow.org/api_docs/python/tf/keras/callbacks/LearningRateScheduler**