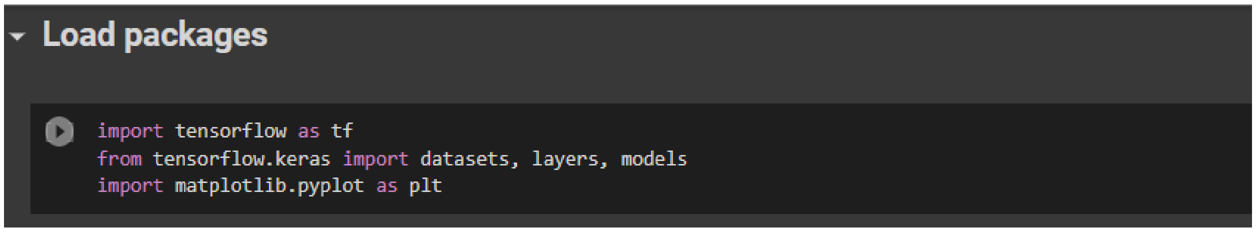
INTORDUCTION TO BIGDATA AND DATA SCIENCE

BONUS ICE

Task 1

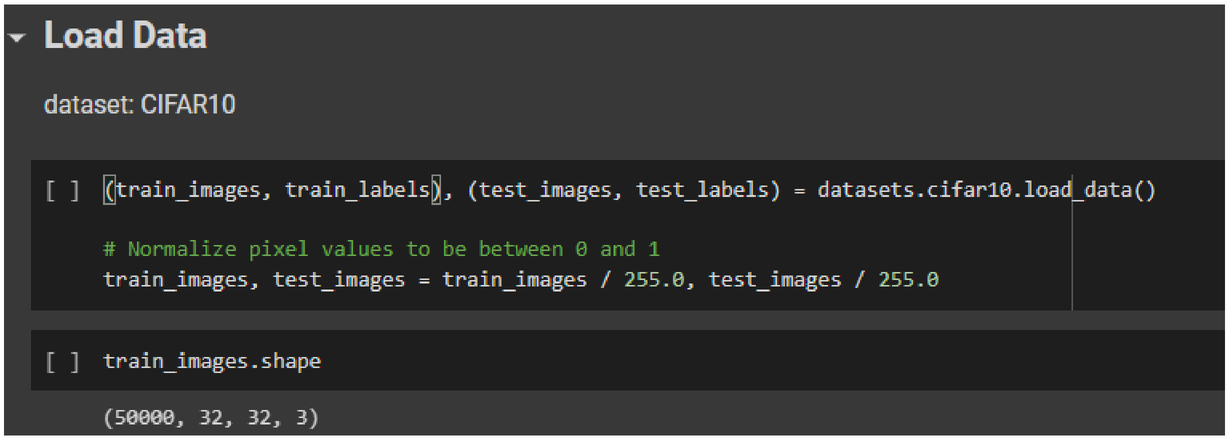
Loading packages that are needed to run the model and plot the graphs for loss and accuracy of the model.



Load packages

```
import tensorflow as tf
from tensorflow.keras import datasets, layers, models
import matplotlib.pyplot as plt
```

Convolutional neural networks are generally used on images. Images that are used in this model are imported from the dataset cifar10 of tenserflow.keras. The images are then split into train images and test images datasets. The images are normalized by dividing the datasets with number 255.0. Then identified the image’s shape. The images are with a shape of height 32, length of 32 and depth of 3. The total count train images are 50000.



Load Data

dataset: CIFAR10

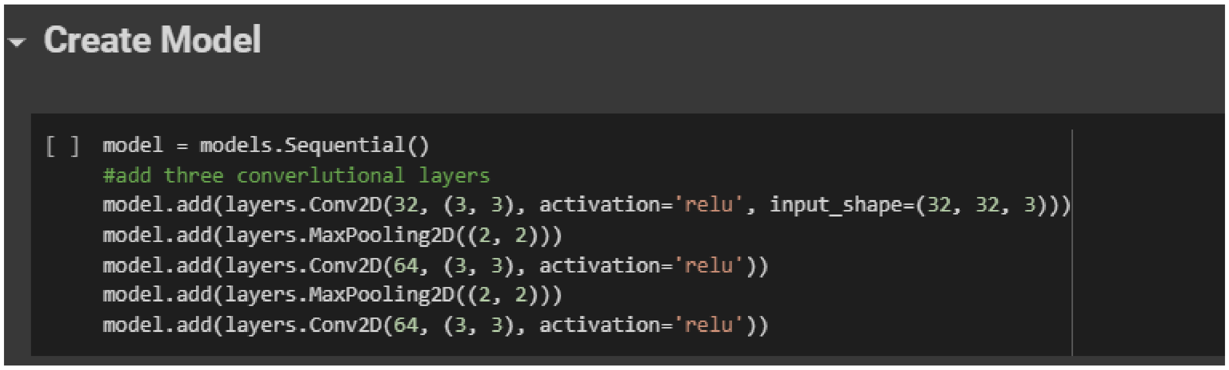
```
(train_images, train_labels), (test_images, test_labels) = datasets.cifar10.load_data()

# Normalize pixel values to be between 0 and 1
train_images, test_images = train_images / 255.0, test_images / 255.0
```

```
train_images.shape
```

```
(50000, 32, 32, 3)
```

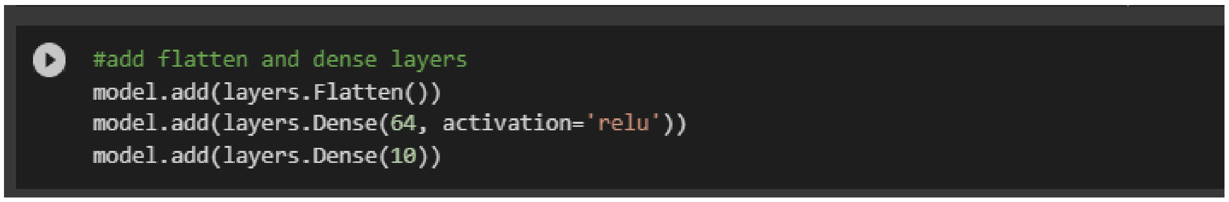
A sequential model is selected for this scenario. The model is added with layers, 2d convolution layer with size of each matrix is 32, 3, 3. This matrix is obtained from the input image. The input image is converted to a matrix based on the pixels. The height and width of the matrix is taken from the number of pixels in a row and number of pixels in columns. The 3rd dimension depth is calculated based on the color of the pixel at that position. If the image is a grey scale image then the value in the 3rd dimension will be either 1 or -1. If the image is a color image then the 3rd dimension will have 3 values. One of R – red, one for G - green and other for B – blue. This is because a any color can be formed with the combination of red, green and blue colors in different propotions. After convolution, various filter operations will be performed on the matrix to identify the patterns of objects in the images. After filtering, maxpooling is applied on the filtered image to downsample the image. MaxPooling in this example is taken as 2X2.

After adding convolution and maxpooling this process is repeated 3 times in this example. This means this model will have 3 layers.



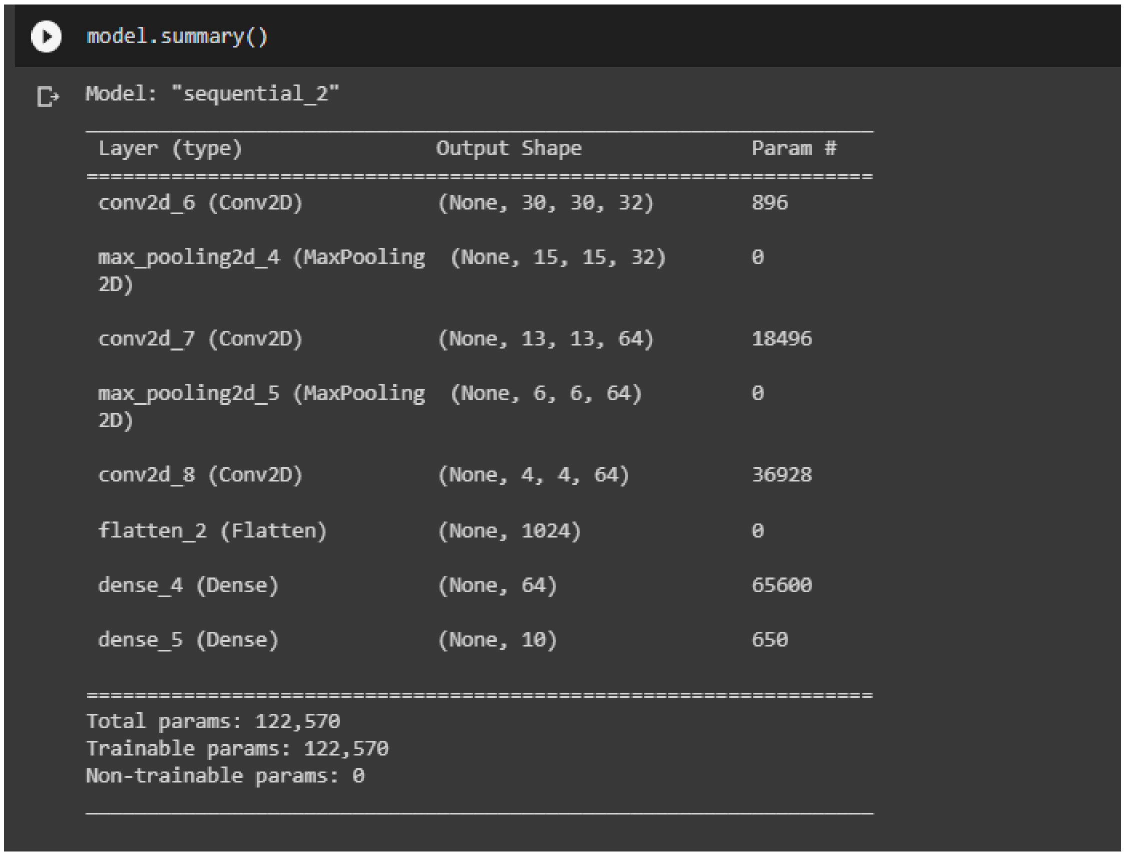
```
model = models.Sequential()
#add three converlutional layers
model.add(layers.Conv2D(32, (3, 3), activation='relu', input_shape=(32, 32, 3)))
model.add(layers.MaxPooling2D((2, 2)))
model.add(layers.Conv2D(64, (3, 3), activation='relu'))
model.add(layers.MaxPooling2D((2, 2)))
model.add(layers.Conv2D(64, (3, 3), activation='relu'))
```

The purpose of using a dense layer in convolution neural networks is to perform non-linear transformation the outputs of all the previous layers and connect them.



```
#add flatten and dense layers
model.add(layers.Flatten())
model.add(layers.Dense(64, activation='relu'))
model.add(layers.Dense(10))
```

The below figure shows the summary of the model created so far. From the image it is evident that the total params in the model are 122,570 and all the parameters can be used for training the model.

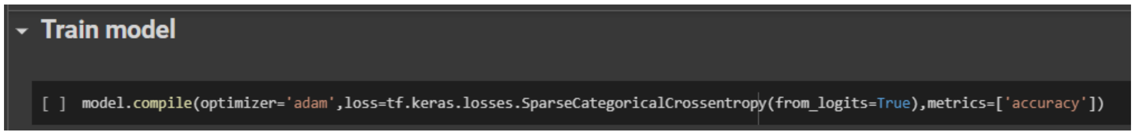


```
model.summary()

Model: "sequential_2"
_________________________________________________________________
 Layer (type)                Output Shape              Param #
=================================================================
 conv2d_6 (Conv2D)           (None, 30, 30, 32)        896

 max_pooling2d_4 (MaxPooling  (None, 15, 15, 32)       0
 2D)

 conv2d_7 (Conv2D)           (None, 13, 13, 64)        18496

 max_pooling2d_5 (MaxPooling  (None, 6, 6, 64)         0
 2D)

 conv2d_8 (Conv2D)           (None, 4, 4, 64)          36928

 flatten_2 (Flatten)         (None, 1024)              0

 dense_4 (Dense)             (None, 64)                65600

 dense_5 (Dense)             (None, 10)                650

=================================================================
Total params: 122,570
Trainable params: 122,570
Non-trainable params: 0
_________________________________________________________________
```

The model is compiled with adam optimizer and sparceCategoricalCrossentropy loss function. Accuracy metric is calculated on compiling the model.

Adam optimizer is an extension of SGD optimizer that adapts learning behavior from the previous output.

Sparse Categorical cross entropy loss function is used in the cases of multi class classification problems.



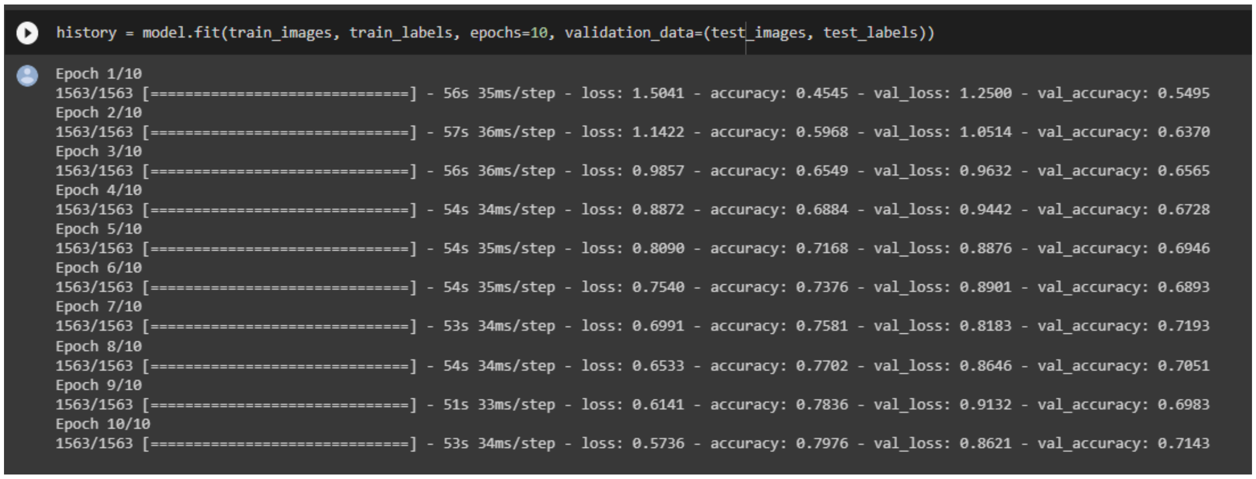
Train model

```
model.compile(optimizer='adam',loss=tf.keras.losses.SparseCategoricalCrossentropy(from_logits=True),metrics=['accuracy'])
```

The training images are given to the model with their labels and test images with their labels for validation.

Epoch is a parameter which says about how many times the model needs to repeat the training process with the same dataset. Increasing epoch will increase the performance of the model as the model will get trained more times with the same dataset. This is only up to some value of the epoch. If we further increase the epoch by that critical value, the model will be over fitted, and it will not perform well.

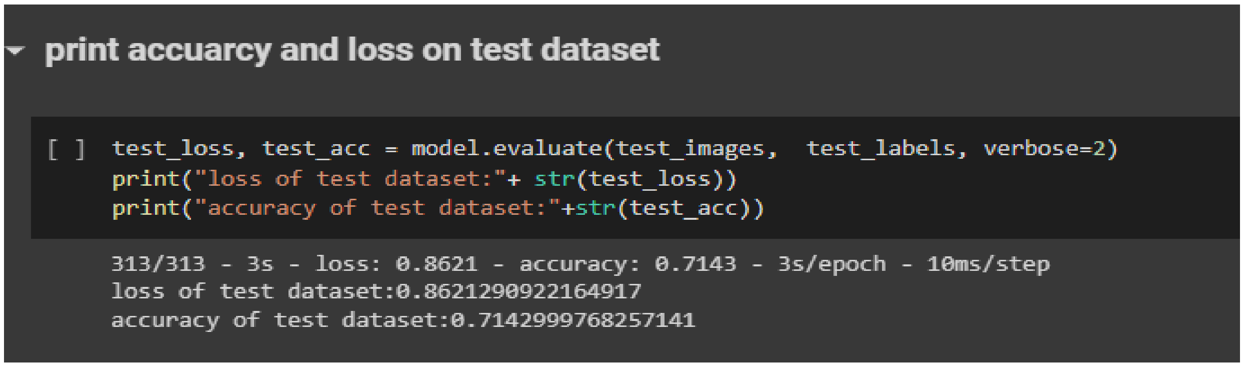
In this case the epoch used is 10 and the accuracy of the model after training the model with same train data for 10 times is 0.7143 which was 0.5495 on epoch 1.



```
history = model.fit(train_images, train_labels, epochs=10, validation_data=(test_images, test_labels))

Epoch 1/10
1563/1563 [==============================] - 56s 35ms/step - loss: 1.5041 - accuracy: 0.4545 - val_loss: 1.2500 - val_accuracy: 0.5495
Epoch 2/10
1563/1563 [==============================] - 57s 36ms/step - loss: 1.1422 - accuracy: 0.5968 - val_loss: 1.0514 - val_accuracy: 0.6370
Epoch 3/10
1563/1563 [==============================] - 56s 36ms/step - loss: 0.9857 - accuracy: 0.6549 - val_loss: 0.9632 - val_accuracy: 0.6565
Epoch 4/10
1563/1563 [==============================] - 54s 34ms/step - loss: 0.8872 - accuracy: 0.6884 - val_loss: 0.9442 - val_accuracy: 0.6728
Epoch 5/10
1563/1563 [==============================] - 54s 35ms/step - loss: 0.8090 - accuracy: 0.7168 - val_loss: 0.8876 - val_accuracy: 0.6946
Epoch 6/10
1563/1563 [==============================] - 54s 35ms/step - loss: 0.7540 - accuracy: 0.7376 - val_loss: 0.8901 - val_accuracy: 0.6893
Epoch 7/10
1563/1563 [==============================] - 53s 34ms/step - loss: 0.6991 - accuracy: 0.7581 - val_loss: 0.8183 - val_accuracy: 0.7193
Epoch 8/10
1563/1563 [==============================] - 54s 34ms/step - loss: 0.6533 - accuracy: 0.7702 - val_loss: 0.8646 - val_accuracy: 0.7051
Epoch 9/10
1563/1563 [==============================] - 51s 33ms/step - loss: 0.6141 - accuracy: 0.7836 - val_loss: 0.9132 - val_accuracy: 0.6983
Epoch 10/10
1563/1563 [==============================] - 53s 34ms/step - loss: 0.5736 - accuracy: 0.7976 - val_loss: 0.8621 - val_accuracy: 0.7143
```

The loss and accuracy of the model have been evaluated using the evaluate method by providing the test images and labels to evaluation method. Verbose is an optional parameter that is given to the evaluation method. This will describe how the evaluation metrics need to be printed. Verbose 2 means that the metrics for each batch has to be printed.

The accuracy and loss of the model are printed after applying the evaluation method on the test data. The accuracy is 0.714 and loss is 0.8621.

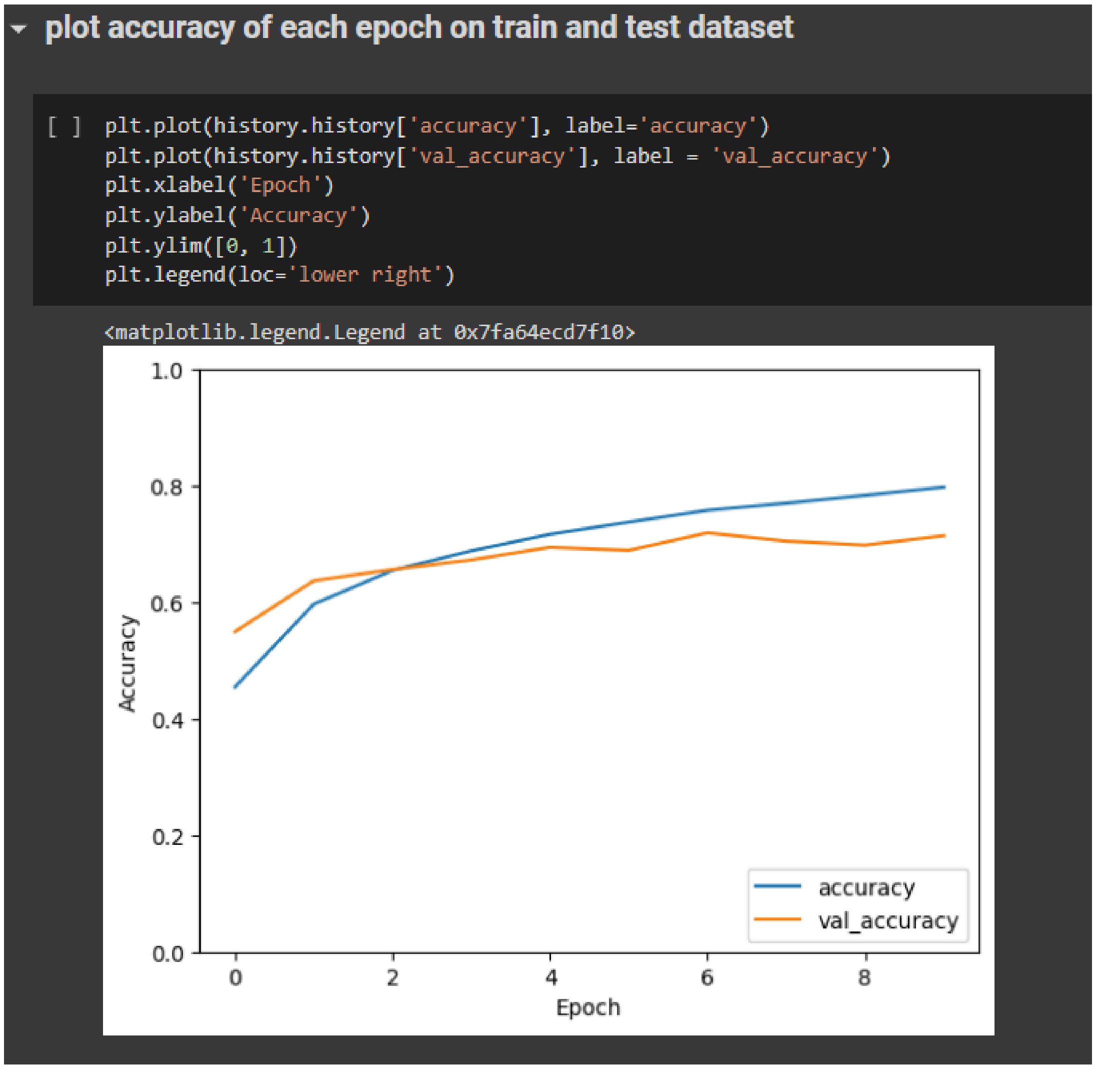


print accuarcy and loss on test dataset

```
[ ] test_loss, test_acc = model.evaluate(test_images,  test_labels, verbose=2)
    print("loss of test dataset:"+ str(test_loss))
    print("accuracy of test dataset:"+str(test_acc))

    313/313 - 3s - loss: 0.8621 - accuracy: 0.7143 - 3s/epoch - 10ms/step
    loss of test dataset:0.8621290922164917
    accuracy of test dataset:0.7142999768257141
```

Accuracy vs epoch plot for the data has been plotted and the screenshot of the result is attached below. From the figure it is evident that the raise in epoch increased the accuracy of the model.

Accuracy of the model is the percentage is performed on the training dataset and val_accuracy is performed on the testing dataset. The val_accuracy is less than accuracy because the model is trained again and again and the model have seen the training data for 10 times as epoch is 10. But the model has not seen the testing data till is asked to validate. So val_accuracy is less compared to accuracy.



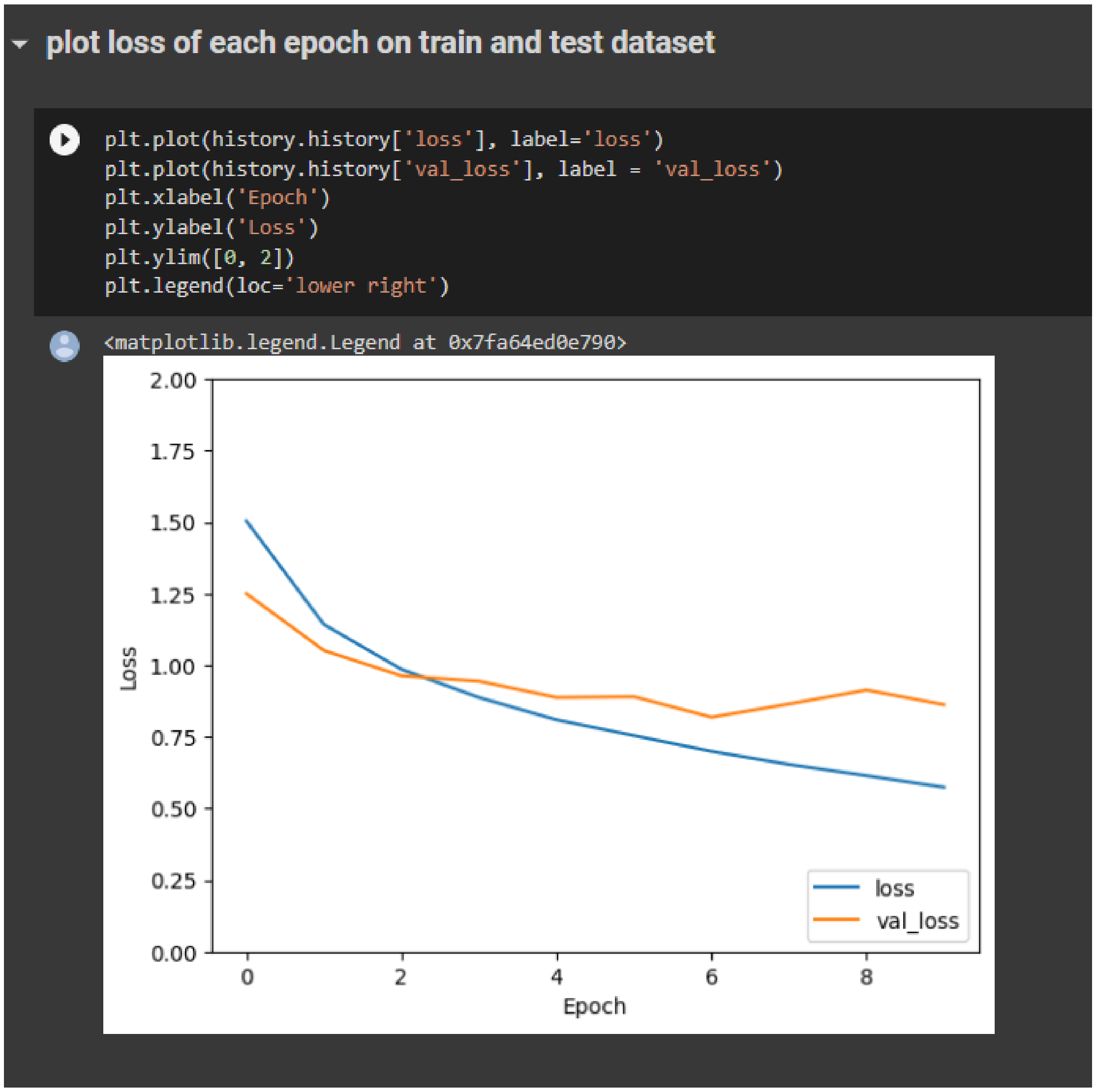
plot accuracy of each epoch on train and test dataset

```
[ ] plt.plot(history.history['accuracy'], label='accuracy')
    plt.plot(history.history['val_accuracy'], label = 'val_accuracy')
    plt.xlabel('Epoch')
    plt.ylabel('Accuracy')
    plt.ylim([0, 1])
    plt.legend(loc='lower right')
```

```
<matplotlib.legend.Legend at 0x7fa64ecd7f10>
```

Loss vs epoch plot for the data has been plotted and the screenshot of the result is attached below. From the figure it is evident that the raise in epoch reduced the loss and val_loss of the model.

Loss of the model is the percentage is performed on the training dataset and val_loss is performed on the testing dataset. The val_loss is higher than loss because the model is trained again and again and the model have seen the training data for 10 times as epoch is 10. But the model has not seen the testing data till is asked to validate. So val_loss is high compared to loss.

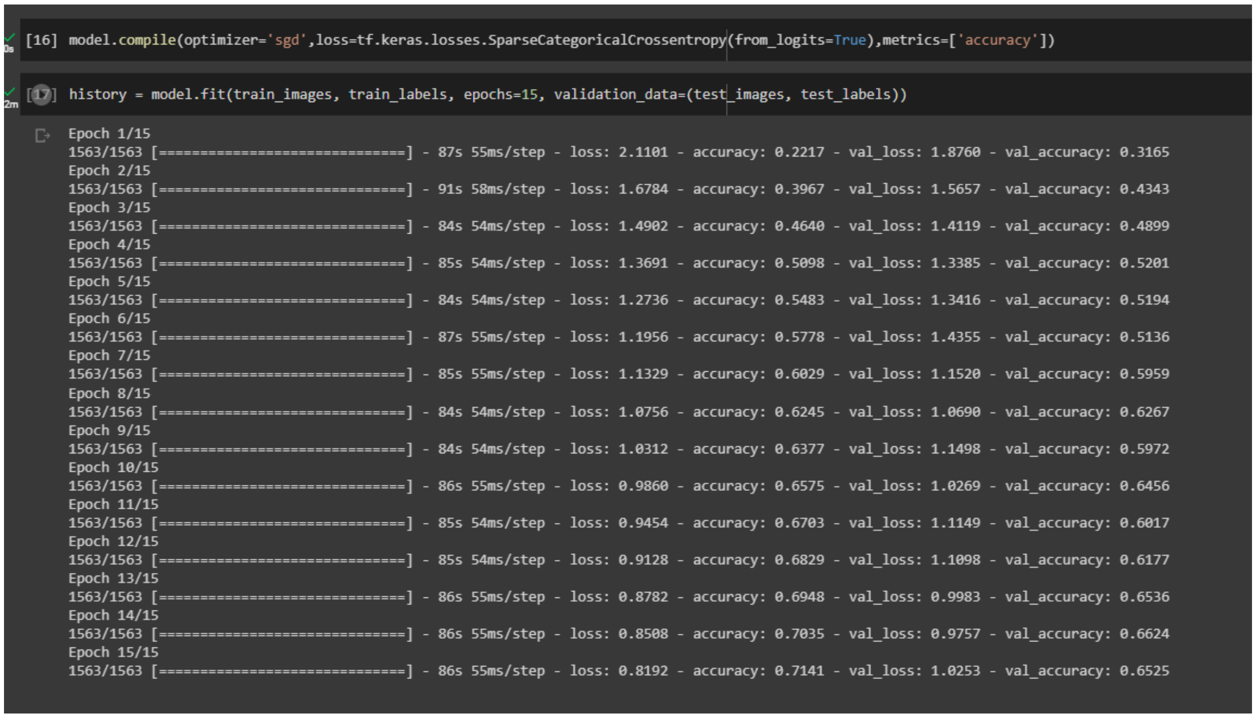


plot loss of each epoch on train and test dataset

```
plt.plot(history.history['loss'], label='loss')
plt.plot(history.history['val_loss'], label = 'val_loss')
plt.xlabel('Epoch')
plt.ylabel('Loss')
plt.ylim([0, 2])
plt.legend(loc='lower right')
```

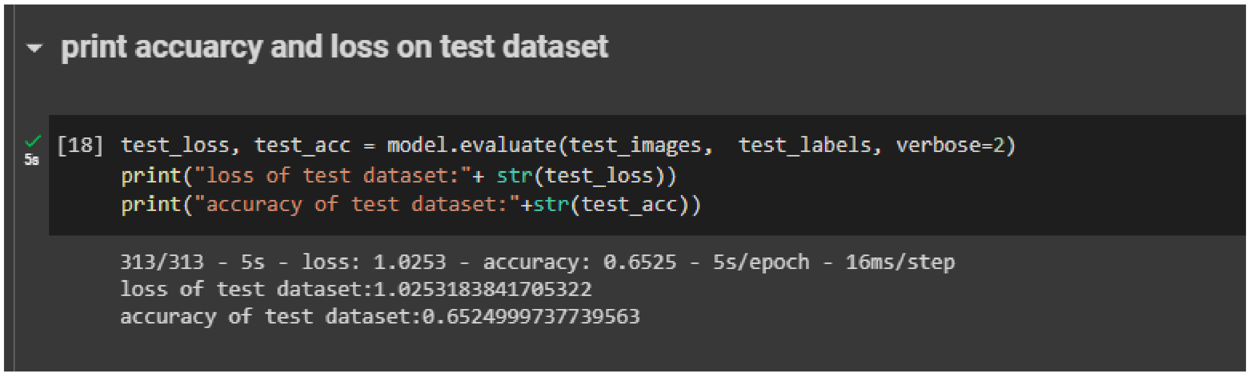
```
<matplotlib.legend.Legend at 0x7fa64ed0e790>
```

The model is trained again by changing the optimizer to sgd which is a minor version of adam optimizer. The epoch is changed to 15. The accuracy of the model is 0.7141 which is close to the accuracy achieved by adam optimizer with epoch 10. But the loss value of the model is at 0.8192 which is higher than that achieved by the previous model.



```
[16] model.compile(optimizer='sgd',loss=tf.keras.losses.SparseCategoricalCrossentropy(from_logits=True),metrics=['accuracy'])

[17] history = model.fit(train_images, train_labels, epochs=15, validation_data=(test_images, test_labels))

Epoch 1/15
1563/1563 [==============================] - 87s 55ms/step - loss: 2.1101 - accuracy: 0.2217 - val_loss: 1.8760 - val_accuracy: 0.3165
Epoch 2/15
1563/1563 [==============================] - 91s 58ms/step - loss: 1.6784 - accuracy: 0.3967 - val_loss: 1.5657 - val_accuracy: 0.4343
Epoch 3/15
1563/1563 [==============================] - 84s 54ms/step - loss: 1.4902 - accuracy: 0.4640 - val_loss: 1.4119 - val_accuracy: 0.4899
Epoch 4/15
1563/1563 [==============================] - 85s 54ms/step - loss: 1.3691 - accuracy: 0.5098 - val_loss: 1.3385 - val_accuracy: 0.5201
Epoch 5/15
1563/1563 [==============================] - 84s 54ms/step - loss: 1.2736 - accuracy: 0.5483 - val_loss: 1.3416 - val_accuracy: 0.5194
Epoch 6/15
1563/1563 [==============================] - 87s 55ms/step - loss: 1.1956 - accuracy: 0.5778 - val_loss: 1.4355 - val_accuracy: 0.5136
Epoch 7/15
1563/1563 [==============================] - 85s 55ms/step - loss: 1.1329 - accuracy: 0.6029 - val_loss: 1.1520 - val_accuracy: 0.5959
Epoch 8/15
1563/1563 [==============================] - 84s 54ms/step - loss: 1.0756 - accuracy: 0.6245 - val_loss: 1.0690 - val_accuracy: 0.6267
Epoch 9/15
1563/1563 [==============================] - 84s 54ms/step - loss: 1.0312 - accuracy: 0.6377 - val_loss: 1.1498 - val_accuracy: 0.5972
Epoch 10/15
1563/1563 [==============================] - 86s 55ms/step - loss: 0.9860 - accuracy: 0.6575 - val_loss: 1.0269 - val_accuracy: 0.6456
Epoch 11/15
1563/1563 [==============================] - 85s 54ms/step - loss: 0.9454 - accuracy: 0.6703 - val_loss: 1.1149 - val_accuracy: 0.6017
Epoch 12/15
1563/1563 [==============================] - 85s 54ms/step - loss: 0.9128 - accuracy: 0.6829 - val_loss: 1.1098 - val_accuracy: 0.6177
Epoch 13/15
1563/1563 [==============================] - 86s 55ms/step - loss: 0.8782 - accuracy: 0.6948 - val_loss: 0.9983 - val_accuracy: 0.6536
Epoch 14/15
1563/1563 [==============================] - 86s 55ms/step - loss: 0.8508 - accuracy: 0.7035 - val_loss: 0.9757 - val_accuracy: 0.6624
Epoch 15/15
1563/1563 [==============================] - 86s 55ms/step - loss: 0.8192 - accuracy: 0.7141 - val_loss: 1.0253 - val_accuracy: 0.6525
```

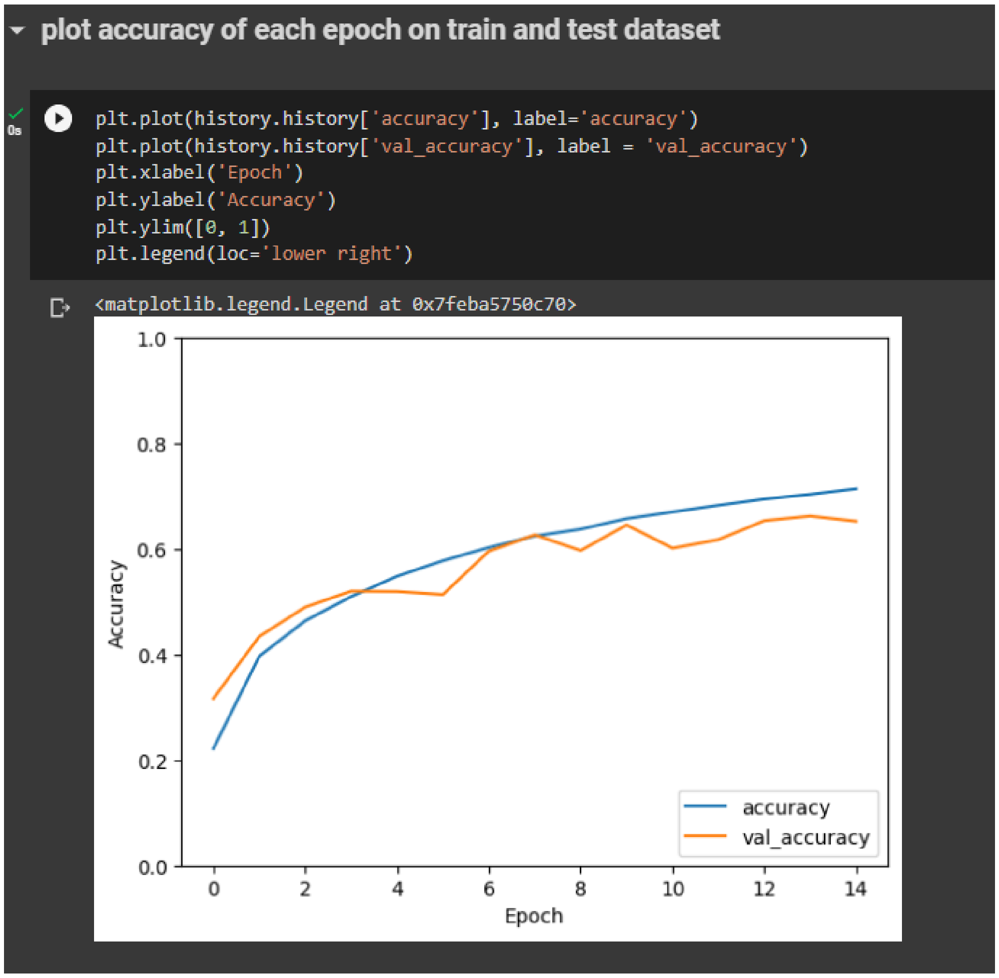
The accuracy and loss of the model is evaluated using the evaluate method on the model. 1.0253 is the loss given by this model where as the previous model is given the loss as 0.853. The accuracy of the model is 0.6525 which is less than the the accuracy of the previous model 0.7143.



print accuarcy and loss on test dataset

```
[18] test_loss, test_acc = model.evaluate(test_images,  test_labels, verbose=2)
     print("loss of test dataset:"+ str(test_loss))
     print("accuracy of test dataset:"+str(test_acc))

313/313 - 5s - loss: 1.0253 - accuracy: 0.6525 - 5s/epoch - 16ms/step
loss of test dataset:1.0253183841705322
accuracy of test dataset:0.6524999737739563
```

The accuracy vs epoch plot of the model is plotted and the screenshot is pasted below.

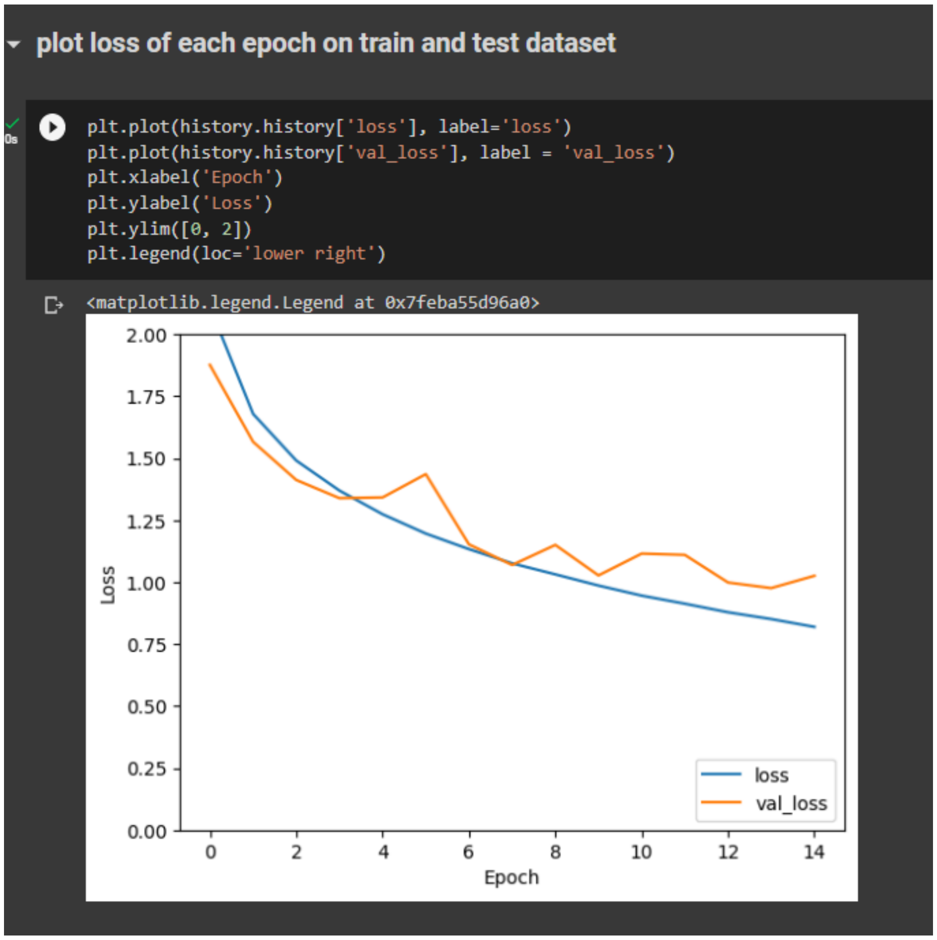


plot accuracy of each epoch on train and test dataset

```
plt.plot(history.history['accuracy'], label='accuracy')
plt.plot(history.history['val_accuracy'], label = 'val_accuracy')
plt.xlabel('Epoch')
plt.ylabel('Accuracy')
plt.ylim([0, 1])
plt.legend(loc='lower right')
```

```
<matplotlib.legend.Legend at 0x7feba5750c70>
```

The loss vs epoch plot of the model is plotted and the screenshot is attached below.



plot loss of each epoch on train and test dataset

```
plt.plot(history.history['loss'], label='loss')
plt.plot(history.history['val_loss'], label = 'val_loss')
plt.xlabel('Epoch')
plt.ylabel('Loss')
plt.ylim([0, 2])
plt.legend(loc='lower right')
```

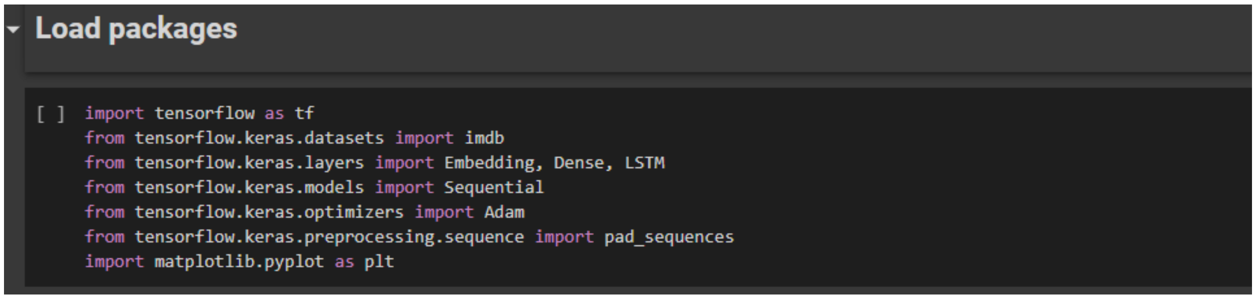
```
<matplotlib.legend.Legend at 0x7feba55d96a0>
```

From the results of both the models, it is clear that the model with epoch 10 and optimizer adam is a clear winner on the model with epoch 15 and optimizer sdg. The reason for it is the optimizer. Adam optimizer is an advanced version of sdg optimizer. The Adam optimizer has a distinct learning rate. For each parameter in the network and dynamically changes the learning rate based on the gradient's first and second moments. This helps to prevent the oscillations and sluggish convergence that might occur with regular SGD, allowing the optimizer to reach a suitable solution more rapidly.

Task 2

RNN stands for Recurrent Neural Network. This model is generally used on sequential data or time series data. Some of the use cases of Recurrent Neural Networks in the fields of natural language processing, time series analysis, speech recognition and image captioning.

The below screenshot about importing the necessary packages that are needed in building the RNN model and plotting the results of the model on a graph.



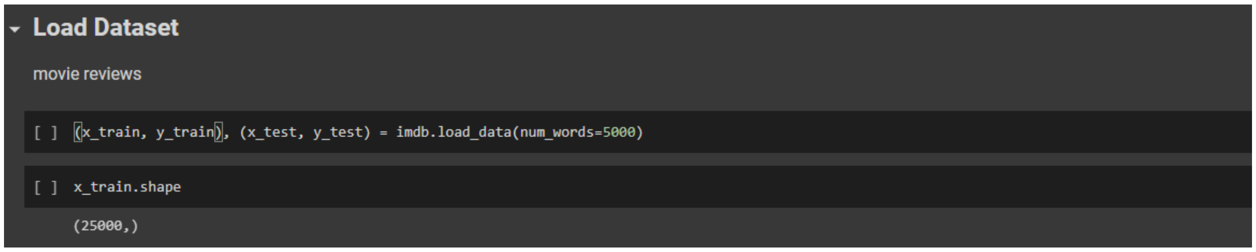
Load packages

```
[ ] import tensorflow as tf
    from tensorflow.keras.datasets import imdb
    from tensorflow.keras.layers import Embedding, Dense, LSTM
    from tensorflow.keras.models import Sequential
    from tensorflow.keras.optimizers import Adam
    from tensorflow.keras.preprocessing.sequence import pad_sequences
    import matplotlib.pyplot as plt
```

The dataset is fetched from IMBD which is a popular website for movies that captures movie reviews from viewers.

Here we are loading 5000 words from IMBD. The fetched data is then split for testing and training. As LSTM is a supervised learning model the input data should have 2 parameters. One of them acting as a reference point and the other as the data at instance. Incase of a time series data, we can assume one parameter as time or date and the other parameter as the data at that moment. Here in the below screenshot or snippet of code, x_train will act as a reference point and y_train will act as the data at x_train for the training data. Similarly, x_test and y_test will act as reference point and actual data respectively.

‘x_train.shape’ will find the shape of the training data. Here it is (25000,). That means there are 25000 samples.



Load Dataset

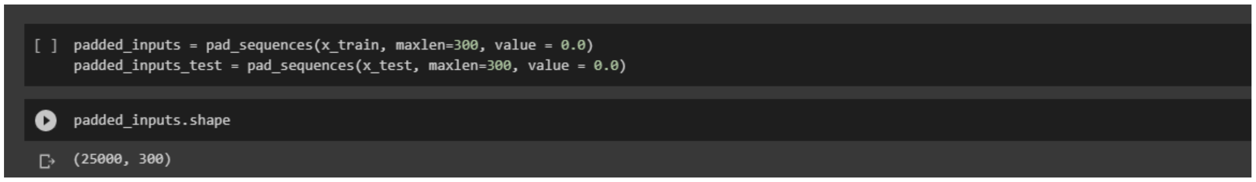
movie reviews

```
[ ] (x_train, y_train), (x_test, y_test) = imdb.load_data(num_words=5000)
```

```
[ ] x_train.shape

    (25000,)
```

Pad_sequence is used on both the inputs ‘x_test’ and ‘x_train’ to maintain the equal length for all the sequences. If there are any sequences that have less number of elements in it, then we need to pad the data with some value to maintain the equal length in all the sequences. Here the sequences are padded with 0 and the maximum length of the sequence is considered as 300.

Then the shape of the x_train data is observed. The shape of x train data after padding is (25000, 300). That means there are 25000 sequences, and each have 300 samples in it.

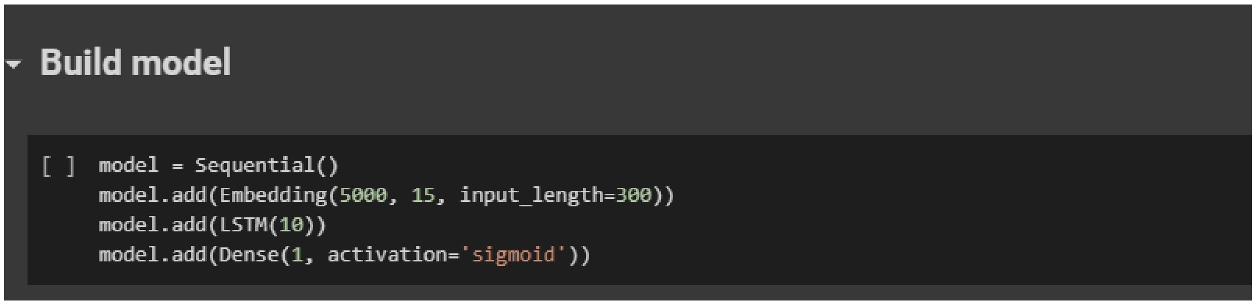


```
[ ] padded_inputs = pad_sequences(x_train, maxlen=300, value = 0.0)
    padded_inputs_test = pad_sequences(x_test, maxlen=300, value = 0.0)

    padded_inputs.shape

    (25000, 300)
```

Created a model using sequential class from keras. Sequential model in keras allows to add layers one after the other by using the add method. The model is added with an embedding layer. This layer will take the encoded sequence input integers and embeds each integer with a dense vector of size 15. The dense in machine learning is used to represent the categorical data in the form of most meaningful vectors that are better understood by machine learning algorithms. They help the ML algorithms to identify the words in a much better way. Here all the 5000 words of input data is embedded int a sequence of dense vectors of size 15. Size of each input sequence is also given as an input.

For this model we are taking a LSTM layer with 10 units in it. Each unit is responsible for processing 1 step at time. Increasing the number of models will help in increasing the performance of the model and will allow the model to store more previous units' data. There are very much useful in the cases of language modeling and speech recognition.

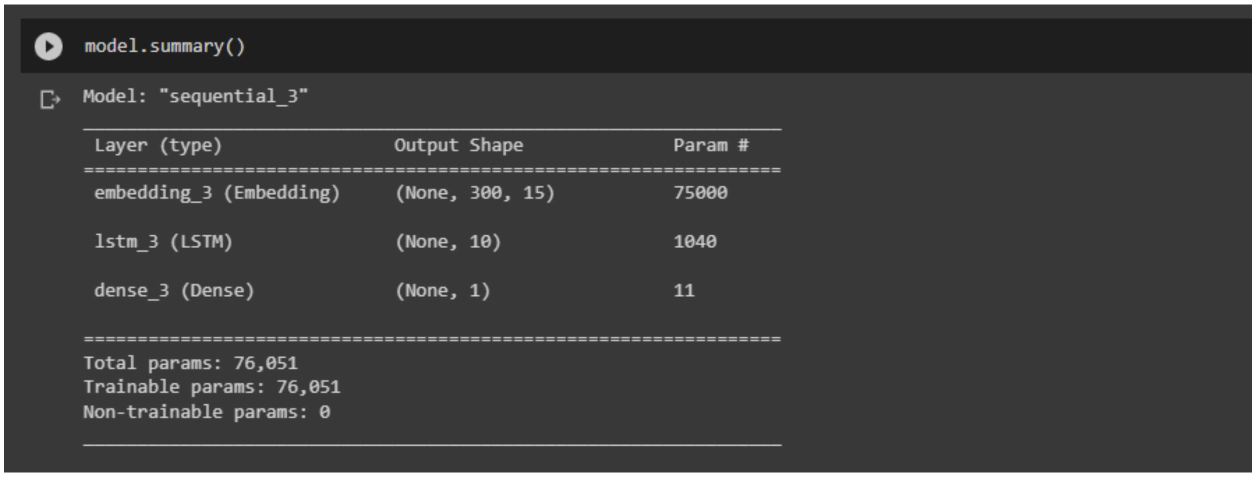
The model is added with a dense layer. The number 1 in the dense method says that the output of the model should be only one unit. A dense layer with 1 output unit is usually used in binary classification tasks. The result of the dense function is a value that may be ranging above one. The binary classification problems needs a probability value that will be between 0 and 1. So the value is passed over a sigmoid function. Sigmoid is a mathematical expression s(x) = 1/(1+exponential(-x)). The will bring any value to a value in between 0 and 1.



Build model

```
[ ] model = Sequential()
    model.add(Embedding(5000, 15, input_length=300))
    model.add(LSTM(10))
    model.add(Dense(1, activation='sigmoid'))
```

The below figure shows the summary of the model created so far. From the image it is evident that the total params in the model are 76,051 and all the parameters can be used for training the model.

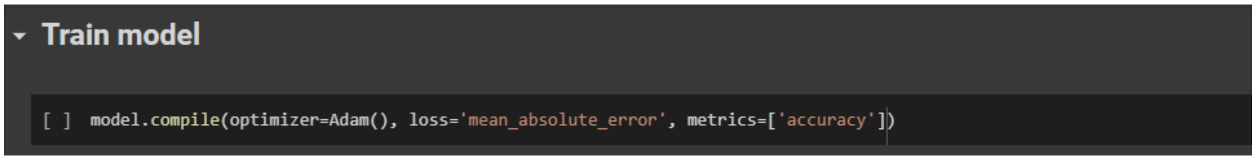


```
model.summary()

Model: "sequential_3"
_________________________________________________________________
 Layer (type)                Output Shape              Param #
=================================================================
 embedding_3 (Embedding)     (None, 300, 15)           75000

 lstm_3 (LSTM)               (None, 10)                1040

 dense_3 (Dense)             (None, 1)                 11

=================================================================
Total params: 76,051
Trainable params: 76,051
Non-trainable params: 0
_________________________________________________________________
```

The model is compiled with adam optimizer and ‘mean_absolute_error’ loss function.

Accuracy metric is calculated on compiling the model. Adam optimizer is an extension of SGD optimizer that adapts learning behavior from the previous output.

Mean Absolute Error loss function is used to calculate the average absolute difference actual and predicted values.



Train model

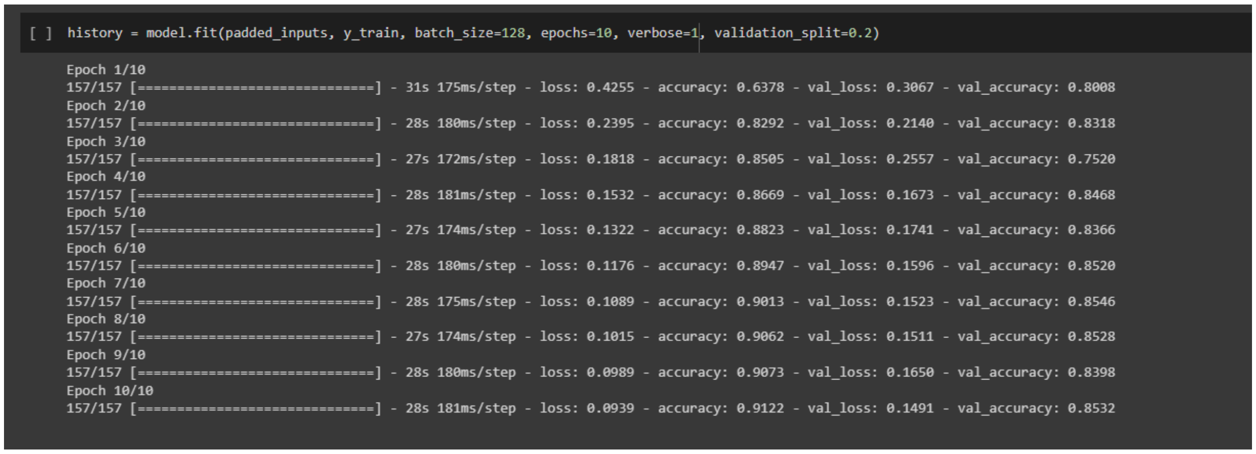
```
[ ] model.compile(optimizer=Adam(), loss='mean_absolute_error', metrics=['accuracy'])
```

The model is fed with padded input data the reference labels in y_train. The data is fed in batches. Each batch will have 128 units of data. That means in one step 128 units of data is used to train the model. There will be a total of 157 batches.

Epoch is a parameter which says about how many times the model needs to repeat the training process with the same dataset. Increasing epoch will increase the performance of the model as the model will get trained more times with the same dataset. This is only up to some value of the epoch. If we further increase the epoch by that critical value, the model will be over fitted, and it will not perform well.

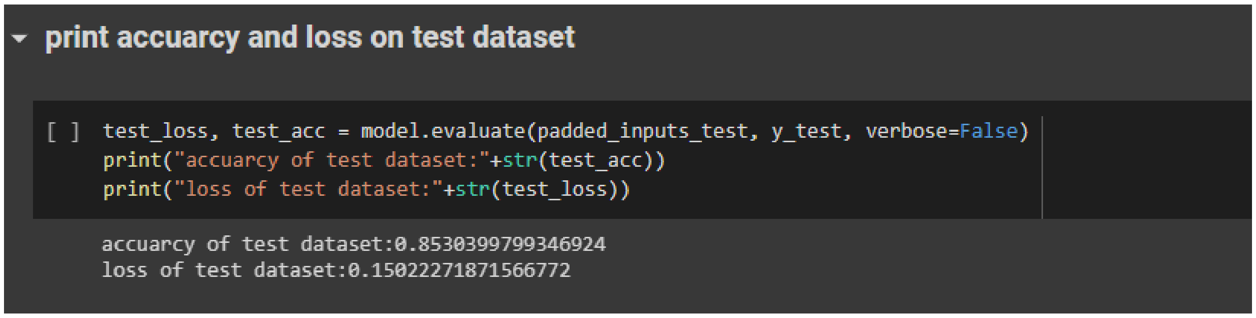
In this case the epoch used is 10 and the accuracy of the model after training the model with same train data for 10 times is 0.9122 which was 0.63778 on epoch 1.

Twenty percent of the input data is used for validation testing. The data split is provided to the parameter validation_split.



```
[ ]  history = model.fit(padded_inputs, y_train, batch_size=128, epochs=10, verbose=1, validation_split=0.2)

     Epoch 1/10
     157/157 [==============================] - 31s 175ms/step - loss: 0.4255 - accuracy: 0.6378 - val_loss: 0.3067 - val_accuracy: 0.8008
     Epoch 2/10
     157/157 [==============================] - 28s 180ms/step - loss: 0.2395 - accuracy: 0.8292 - val_loss: 0.2140 - val_accuracy: 0.8318
     Epoch 3/10
     157/157 [==============================] - 27s 172ms/step - loss: 0.1818 - accuracy: 0.8505 - val_loss: 0.2557 - val_accuracy: 0.7520
     Epoch 4/10
     157/157 [==============================] - 28s 181ms/step - loss: 0.1532 - accuracy: 0.8669 - val_loss: 0.1673 - val_accuracy: 0.8468
     Epoch 5/10
     157/157 [==============================] - 27s 174ms/step - loss: 0.1322 - accuracy: 0.8823 - val_loss: 0.1741 - val_accuracy: 0.8366
     Epoch 6/10
     157/157 [==============================] - 28s 180ms/step - loss: 0.1176 - accuracy: 0.8947 - val_loss: 0.1596 - val_accuracy: 0.8520
     Epoch 7/10
     157/157 [==============================] - 28s 175ms/step - loss: 0.1089 - accuracy: 0.9013 - val_loss: 0.1523 - val_accuracy: 0.8546
     Epoch 8/10
     157/157 [==============================] - 27s 174ms/step - loss: 0.1015 - accuracy: 0.9062 - val_loss: 0.1511 - val_accuracy: 0.8528
     Epoch 9/10
     157/157 [==============================] - 28s 180ms/step - loss: 0.0989 - accuracy: 0.9073 - val_loss: 0.1650 - val_accuracy: 0.8398
     Epoch 10/10
     157/157 [==============================] - 28s 181ms/step - loss: 0.0939 - accuracy: 0.9122 - val_loss: 0.1491 - val_accuracy: 0.8532
```

The accuracy and loss of the model is obtained using evaluate method. The method is provided with the test data and an additional parameter verbose is also provided. Verbose false means that we are not requesting the model to provide the performance metrics for each set of data. The accuracy of the model is 0.85 and the loss of the model is 0.15.

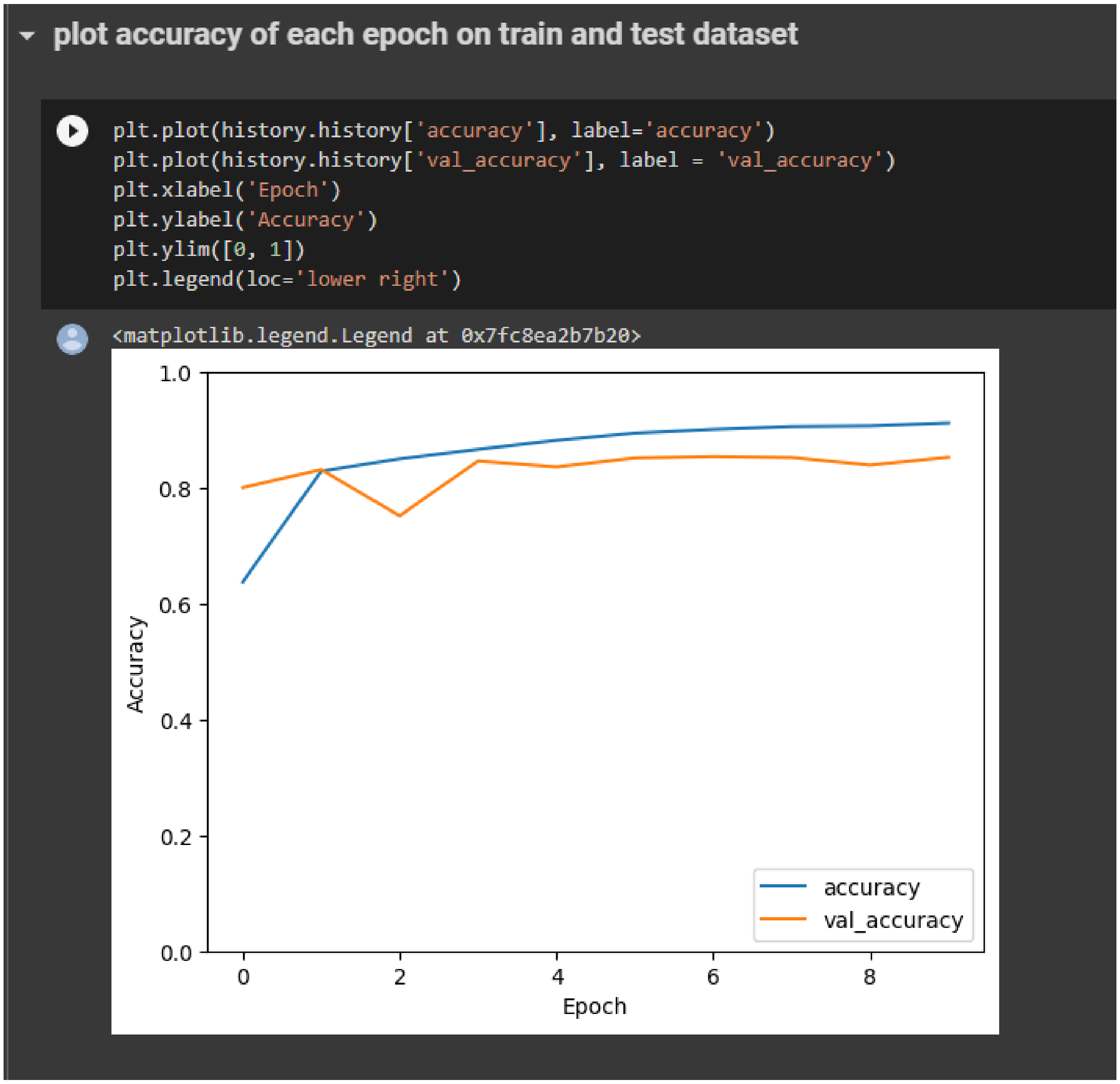


print accuarcy and loss on test dataset

```
[ ]  test_loss, test_acc = model.evaluate(padded_inputs_test, y_test, verbose=False)
     print("accuarcy of test dataset:"+str(test_acc))
     print("loss of test dataset:"+str(test_loss))

     accuarcy of test dataset:0.8530399799346924
     loss of test dataset:0.15022271871566772
```

Accuracy vs epoch plot for the data has been plotted and the screenshot of the result is attached below. From the figure it is evident that the raise in epoch increased the accuracy of the model.

Accuracy of the model is the percentage is performed on the training dataset and val_accuracy is performed on the testing dataset. The val_accuracy is less than accuracy because the model is trained again and again and the model have seen the training data for 10 times as epoch is 10. But the model has not seen the testing data till is asked to validate. So val_accuracy is less compared to accuracy.



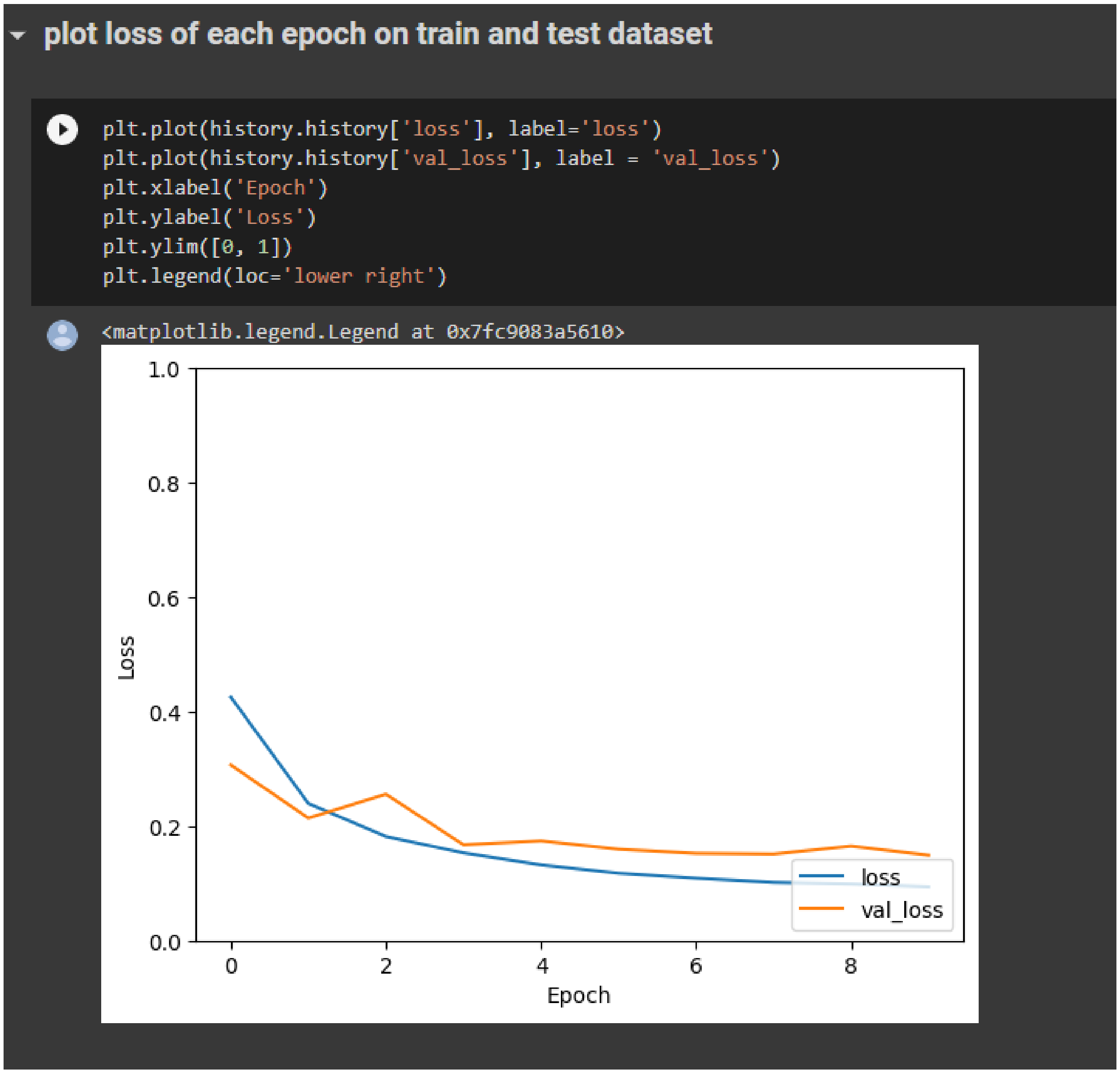
plot accuracy of each epoch on train and test dataset

```
plt.plot(history.history['accuracy'], label='accuracy')
plt.plot(history.history['val_accuracy'], label = 'val_accuracy')
plt.xlabel('Epoch')
plt.ylabel('Accuracy')
plt.ylim([0, 1])
plt.legend(loc='lower right')
```

```
<matplotlib.legend.Legend at 0x7fc8ea2b7b20>
```

Loss vs epoch plot for the data has been plotted and the screenshot of the result is attached below. From the figure it is evident that the raise in epoch reduced the loss and val_loss of the model.

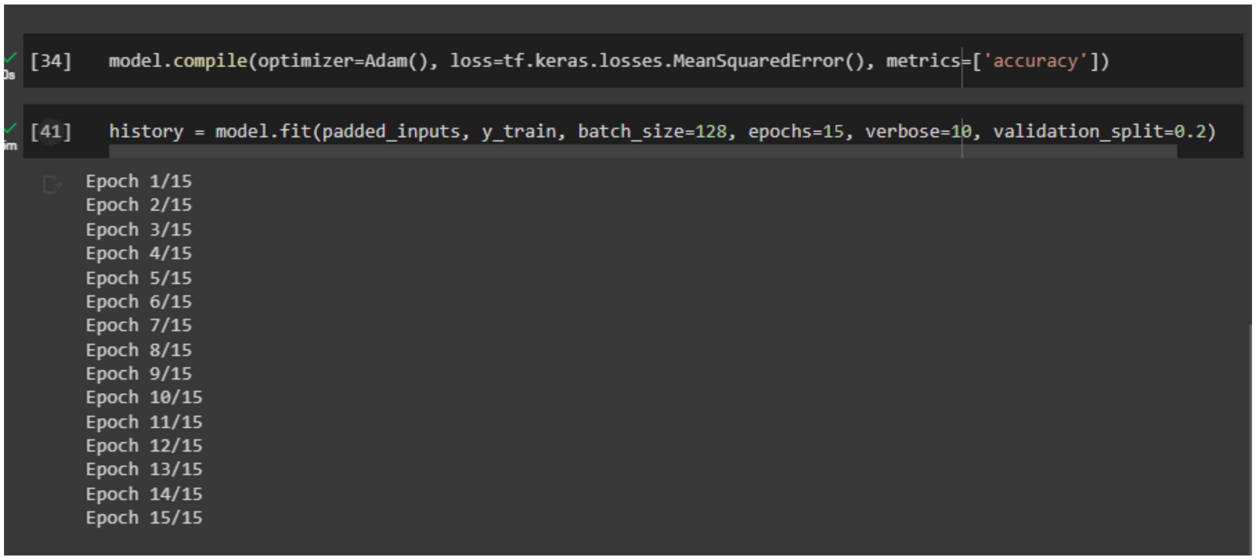
Loss of the model is the percentage is performed on the training dataset and val_loss is performed on the testing dataset. The val_loss is higher than loss because the model is trained again and again and the model have seen the training data for 10 times as epoch is 10. But the model has not seen the testing data till is asked to validate. So val_loss is high compared to loss.


plot loss of each epoch on train and test dataset

```
plt.plot(history.history['loss'], label='loss')
plt.plot(history.history['val_loss'], label = 'val_loss')
plt.xlabel('Epoch')
plt.ylabel('Loss')
plt.ylim([0, 1])
plt.legend(loc='lower right')
```

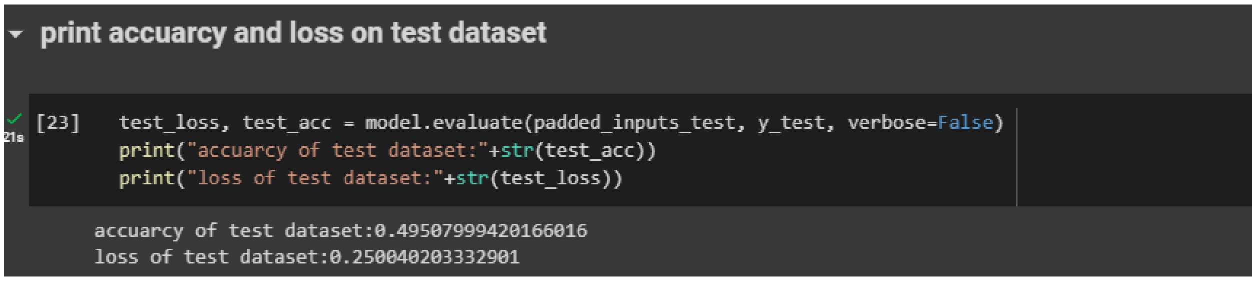
```
<matplotlib.legend.Legend at 0x7fc9083a5610>
```

The model is run again with the initially given loss function, mean square error and the epoch as 15. The mean square error loss function measures the average squared difference between the predicted and actual values. Here we don't have the breakdown of accuracy and loss as we are using verbose as 10 which equivalents to verbose false.



```
[34]    model.compile(optimizer=Adam(), loss=tf.keras.losses.MeanSquaredError(), metrics=['accuracy'])

[41]    history = model.fit(padded_inputs, y_train, batch_size=128, epochs=15, verbose=10, validation_split=0.2)

        Epoch 1/15
        Epoch 2/15
        Epoch 3/15
        Epoch 4/15
        Epoch 5/15
        Epoch 6/15
        Epoch 7/15
        Epoch 8/15
        Epoch 9/15
        Epoch 10/15
        Epoch 11/15
        Epoch 12/15
        Epoch 13/15
        Epoch 14/15
        Epoch 15/15
```

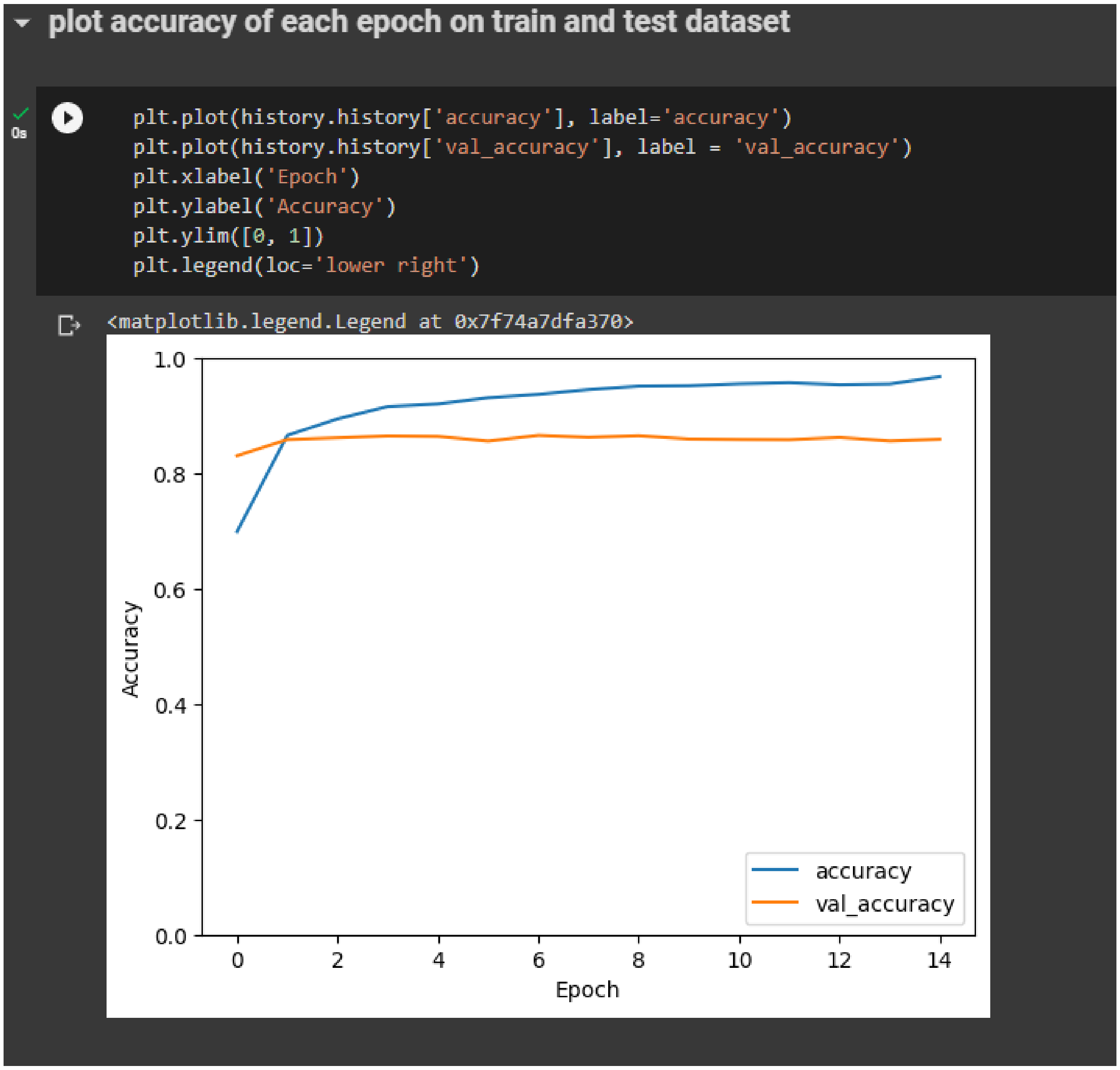
The accuracy and loss of the model is obtained by using the below code. The accuracy obtained for the model with epoch as 15 and loss function as mean square error is 0.49 and the loss is 0.25. This means the previous model performed much better compared to this model.



print accuarcy and loss on test dataset

```
[23]    test_loss, test_acc = model.evaluate(padded_inputs_test, y_test, verbose=False)
        print("accuarcy of test dataset:"+str(test_acc))
        print("loss of test dataset:"+str(test_loss))

        accuarcy of test dataset:0.49507999420166016
        loss of test dataset:0.250040203332901
```

The accuracy vs epoch plot of the model is plotted and the screenshot is pasted below.

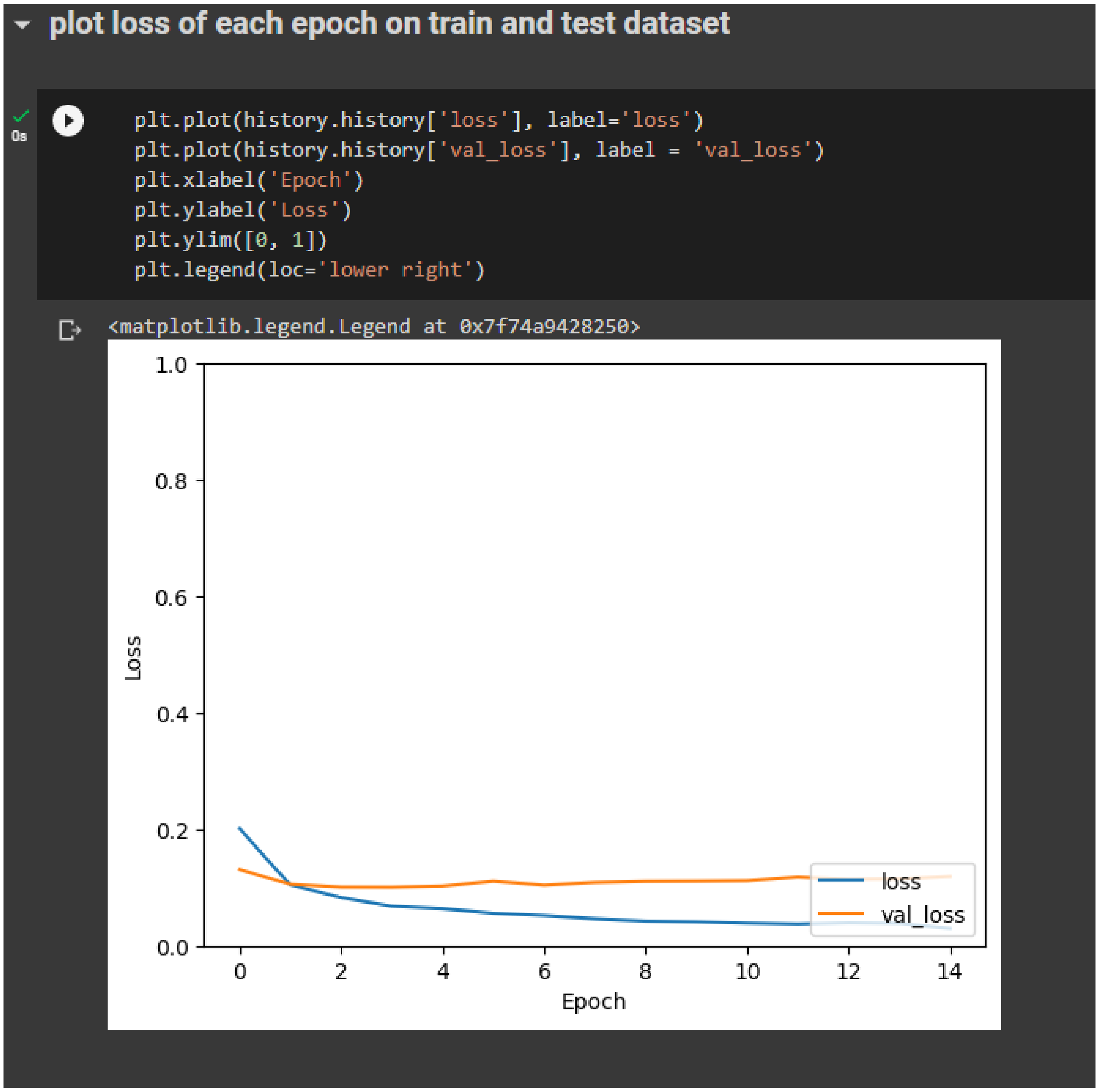


plot accuracy of each epoch on train and test dataset

```
plt.plot(history.history['accuracy'], label='accuracy')
plt.plot(history.history['val_accuracy'], label = 'val_accuracy')
plt.xlabel('Epoch')
plt.ylabel('Accuracy')
plt.ylim([0, 1])
plt.legend(loc='lower right')
```

```
<matplotlib.legend.Legend at 0x7f74a7dfa370>
```

The loss vs epoch plot of the model is plotted and the screenshot is pasted below.



plot loss of each epoch on train and test dataset

```
plt.plot(history.history['loss'], label='loss')
plt.plot(history.history['val_loss'], label = 'val_loss')
plt.xlabel('Epoch')
plt.ylabel('Loss')
plt.ylim([0, 1])
plt.legend(loc='lower right')
```

```
<matplotlib.legend.Legend at 0x7f74a9428250>
```

Though the plots of both the models are almost close, the model with epoch 15 and loss function as mean squared error performed slightly better that the other model. The reason for the model with mean square error as loss function performed better may be, squaring of the errors in mean square error can reduce the impact of small errors while amplifying the large errors. This means that MSE may be better suited for tasks where the goal is to minimize large errors, such as in image segmentation or object detection.