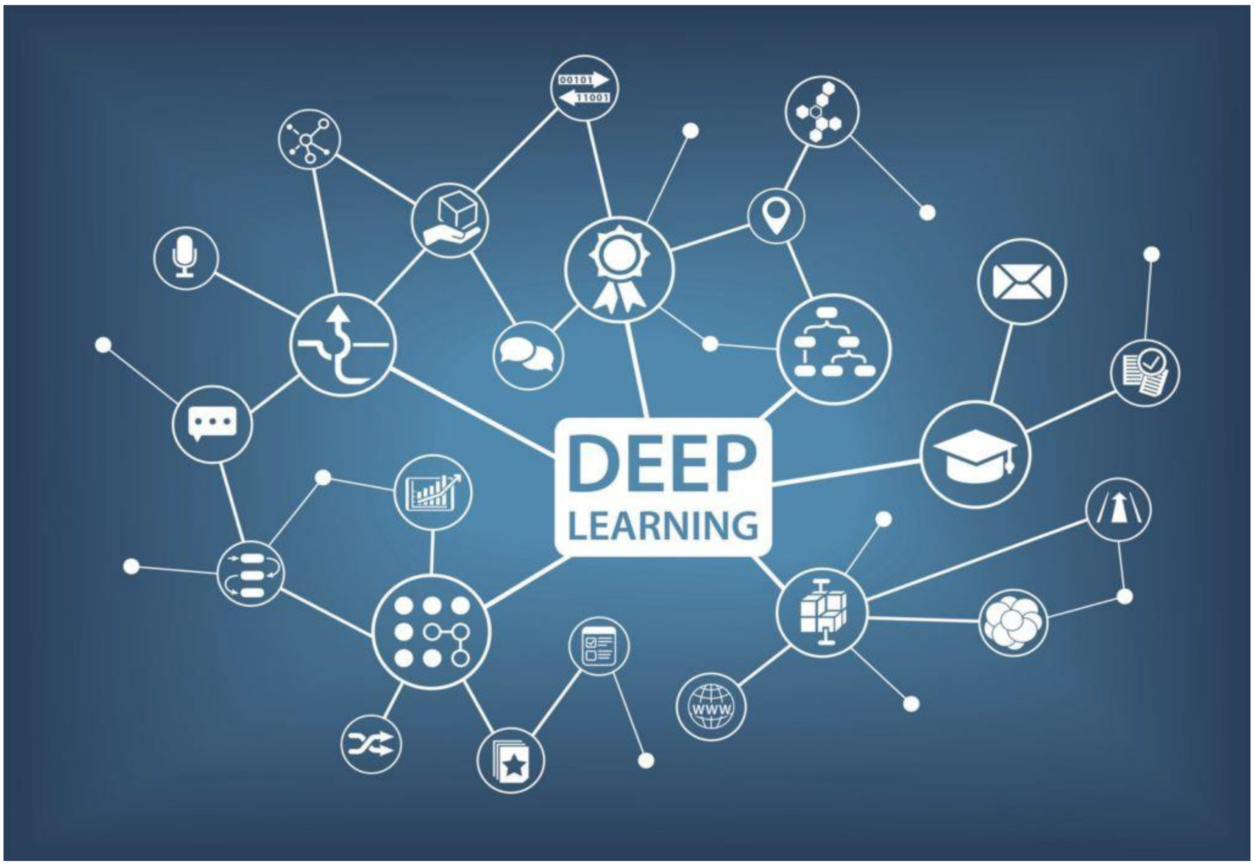
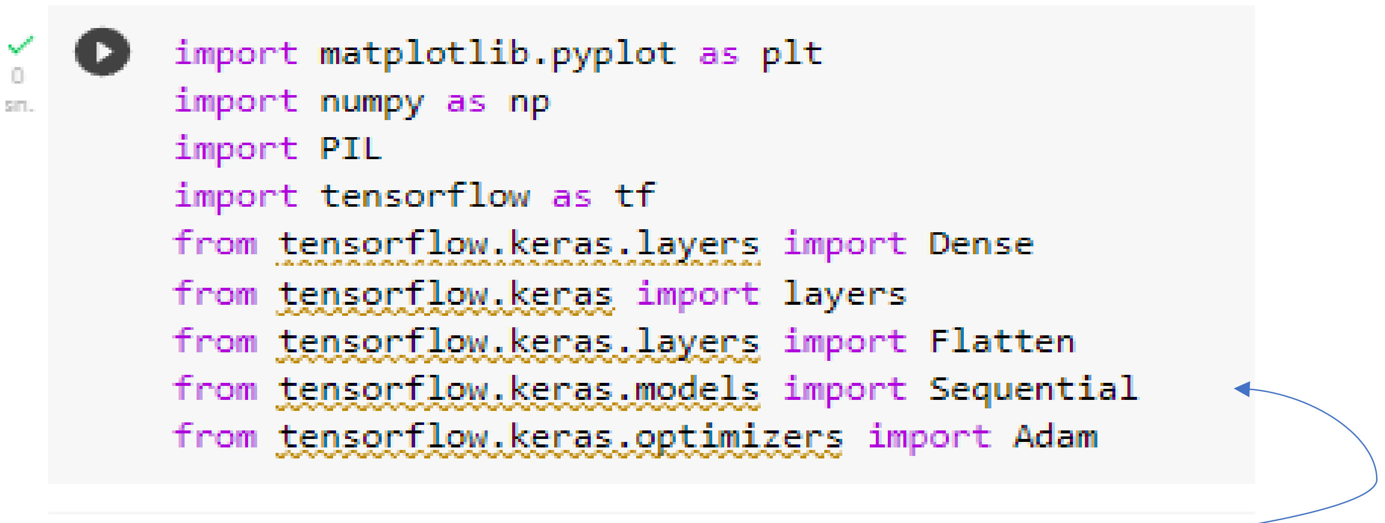


ELİF BAYIR

Deep Learning

Question 1

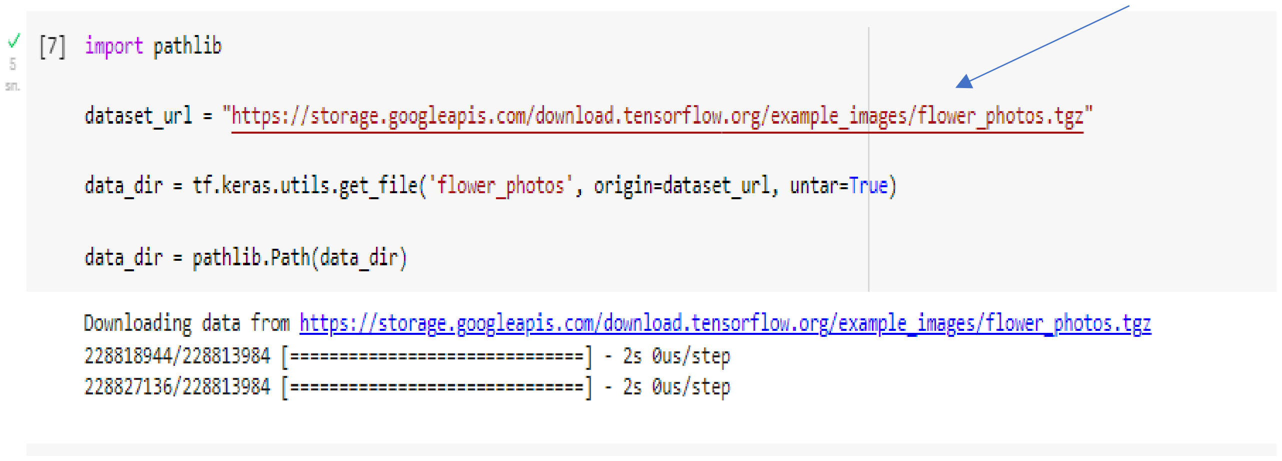


```
import matplotlib.pyplot as plt
import numpy as np
import PIL
import tensorflow as tf
from tensorflow.keras.layers import Dense
from tensorflow.keras import layers
from tensorflow.keras.layers import Flatten
from tensorflow.keras.models import Sequential
from tensorflow.keras.optimizers import Adam
```

Import all the required libraries

Use the same approach for any data you want. For the purpose of this tutorial, we'll be working on the Tensorflow flower classification problem. The data set contains of 5 classes of flowers, for which we will try to build a classifier.

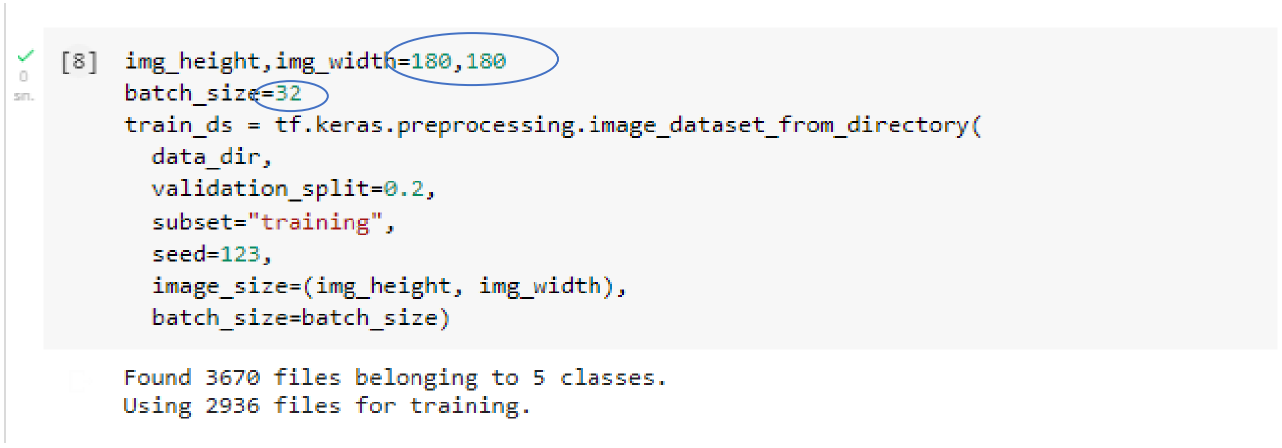
Execute the following lines to import the data on your workspace.



```
[7] import pathlib

    dataset_url = "https://storage.googleapis.com/download.tensorflow.org/example_images/flower_photos.tgz"

    data_dir = tf.keras.utils.get_file('flower_photos', origin=dataset_url, untar=True)

    data_dir = pathlib.Path(data_dir)
```

```
Downloading data from https://storage.googleapis.com/download.tensorflow.org/example_images/flower_photos.tgz
228818944/228813984 [==============================] - 2s 0us/step
228827136/228813984 [==============================] - 2s 0us/step
```

Split our image data into training and validation. With each epoch, our model will get trained on the training subset, while it checks its performance on the validation data at each epoch.



```
[8]  img_height,img_width=180,180
     batch_size=32
     train_ds = tf.keras.preprocessing.image_dataset_from_directory(
       data_dir,
       validation_split=0.2,
       subset="training",
       seed=123,
       image_size=(img_height, img_width),
       batch_size=batch_size)
```

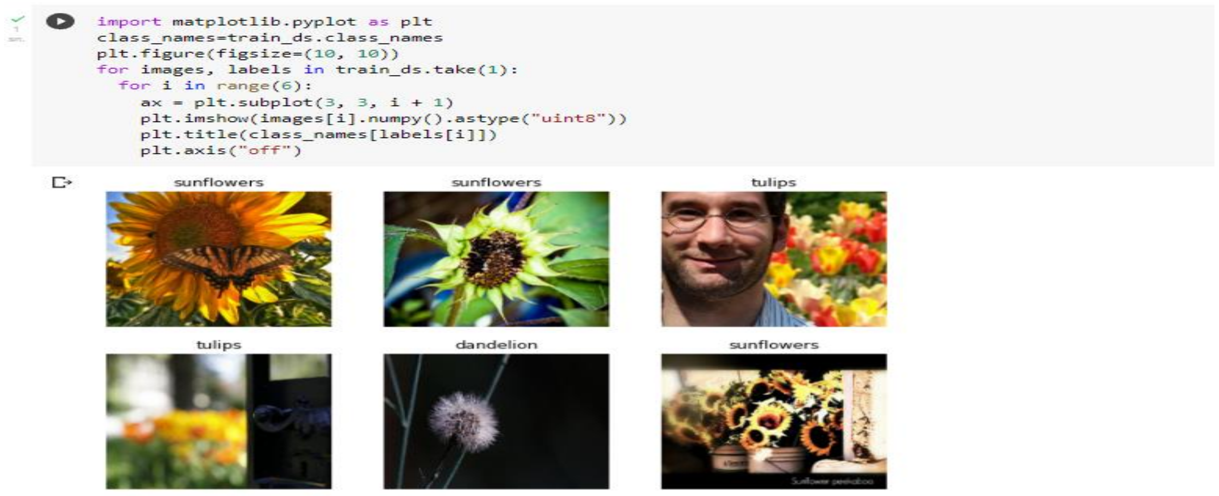
```
Found 3670 files belonging to 5 classes.
Using 2936 files for training.
```

input to a dimension of 180,180

mention the validation split as 0.2. This means that 80% of data will reserved for training while 20% will be used for validation

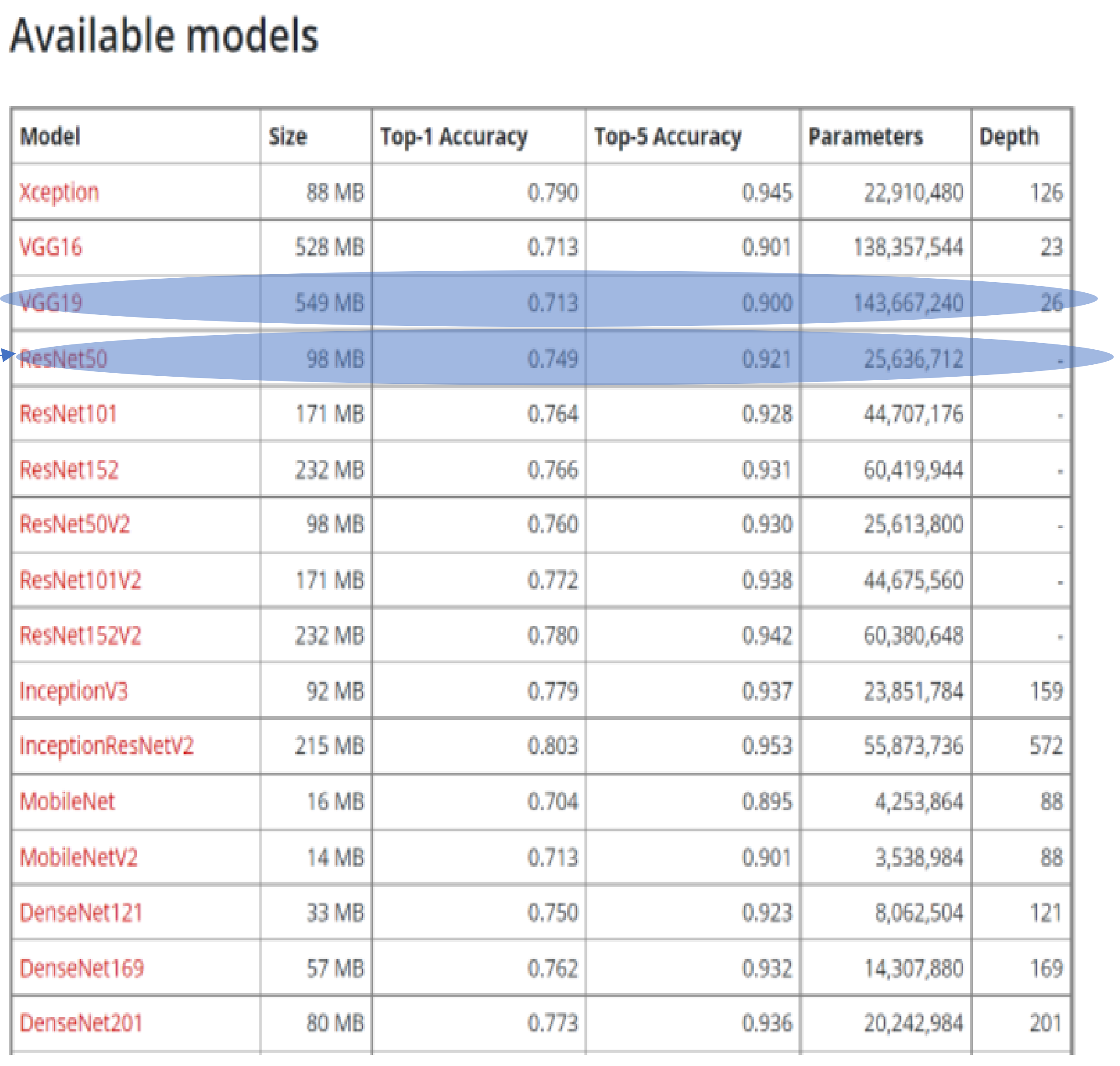
batch size as 32. If you are working on a system with lower ram configuration, you can reduce the batch size further

<u>using the matplotlib library to visualize 6 images in our data</u>



```
import matplotlib.pyplot as plt
class_names=train_ds.class_names
plt.figure(figsize=(10, 10))
for images, labels in train_ds.take(1):
  for i in range(6):
    ax = plt.subplot(3, 3, i + 1)
    plt.imshow(images[i].numpy().astype("uint8"))
    plt.title(class_names[labels[i]])
    plt.axis("off")
```

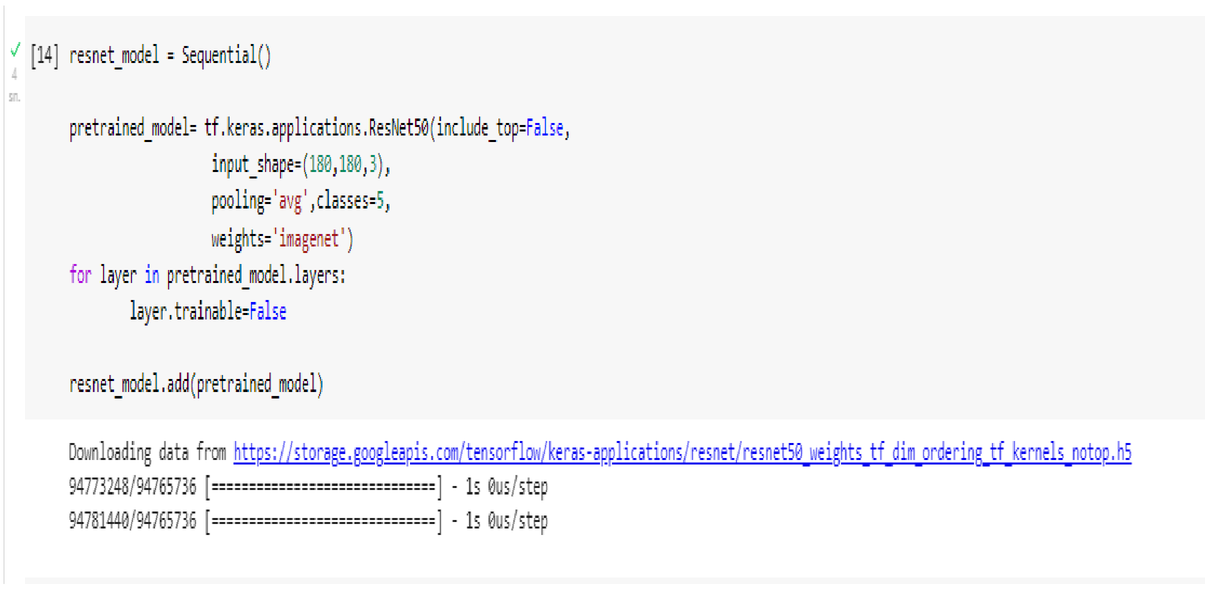
COMPARE ResNet vs VGG19



Available models

Model	Size	Top-1 Accuracy	Top-5 Accuracy	Parameters	Depth
Xception	88 MB	0.790	0.945	22,910,480	126
VGG16	528 MB	0.713	0.901	138,357,544	23
VGG19	549 MB	0.713	0.900	143,667,240	26
ResNet50	98 MB	0.749	0.921	25,636,712	-
ResNet101	171 MB	0.764	0.928	44,707,176	-
ResNet152	232 MB	0.766	0.931	60,419,944	-
ResNet50V2	98 MB	0.760	0.930	25,613,800	-
ResNet101V2	171 MB	0.772	0.938	44,675,560	-
ResNet152V2	232 MB	0.780	0.942	60,380,648	-
InceptionV3	92 MB	0.779	0.937	23,851,784	159
InceptionResNetV2	215 MB	0.803	0.953	55,873,736	572
MobileNet	16 MB	0.704	0.895	4,253,864	88
MobileNetV2	14 MB	0.713	0.901	3,538,984	88
DenseNet121	33 MB	0.750	0.923	8,062,504	121
DenseNet169	57 MB	0.762	0.932	14,307,880	169
DenseNet201	80 MB	0.773	0.936	20,242,984	201

List of available models in keras



```
[14] resnet_model = Sequential()

     pretrained_model= tf.keras.applications.ResNet50(include_top=False,
                        input_shape=(180,180,3),
                        pooling='avg',classes=5,
                        weights='imagenet')
     for layer in pretrained_model.layers:
             layer.trainable=False

     resnet_model.add(pretrained_model)
```

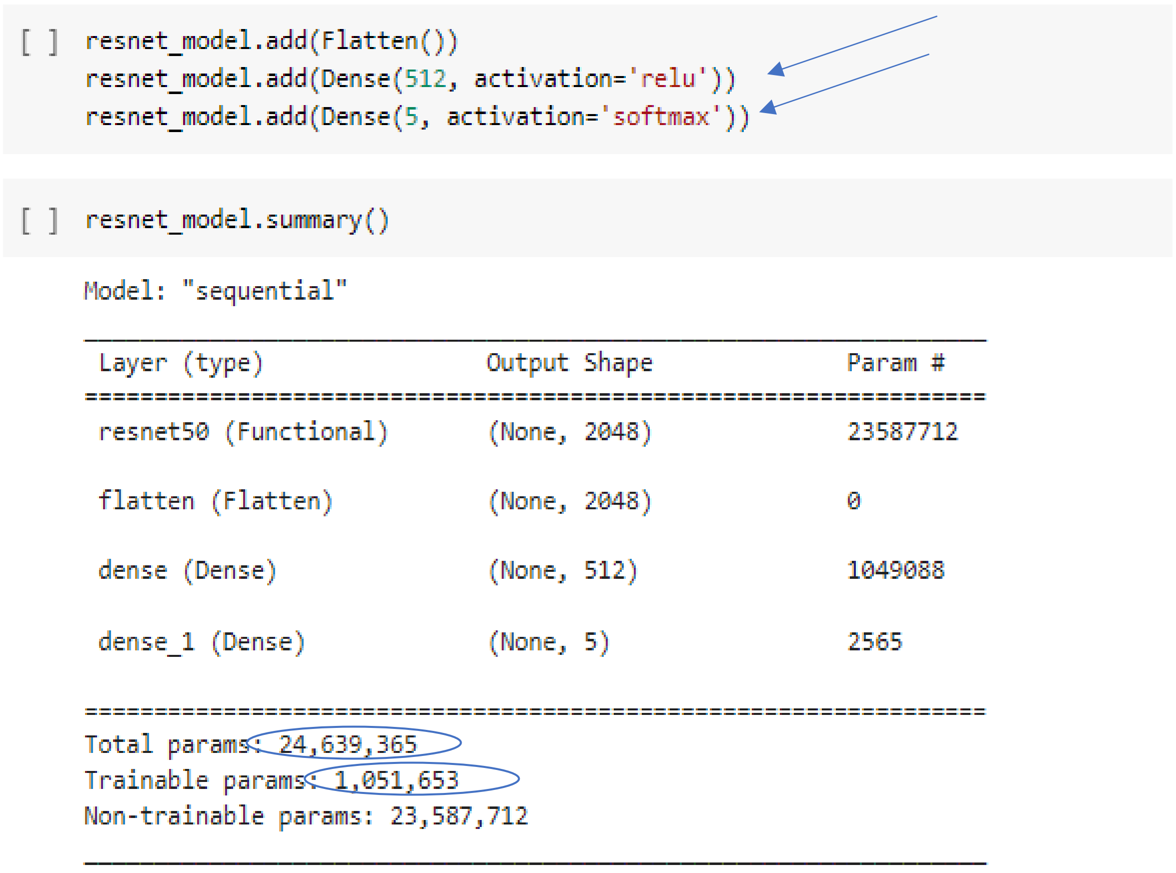
```
Downloading data from https://storage.googleapis.com/tensorflow/keras-applications/resnet/resnet50_weights_tf_dim_ordering_tf_kernels_notop.h5
94773248/94765736 [==============================] - 1s 0us/step
94781440/94765736 [==============================] - 1s 0us/step
```

Importing the ResNet class, we mention include_top=False. This ensures that we can add our own custom input and output layers according to our data.

Mention the weights='imagenet'. This means that the Resnet50 model will use the weights it learnt while being trained on the imagenet data.

layer.trainable= False in the pretrained model. This ensures that the model does not learn the weights again, saving us a lot of time and space complexity.

In the output layer we use the softmax activation function and we have 5 output neurons corresponding to the 5 classes in our data



```
[ ] resnet_model.add(Flatten())
    resnet_model.add(Dense(512, activation='relu'))
    resnet_model.add(Dense(5, activation='softmax'))
```

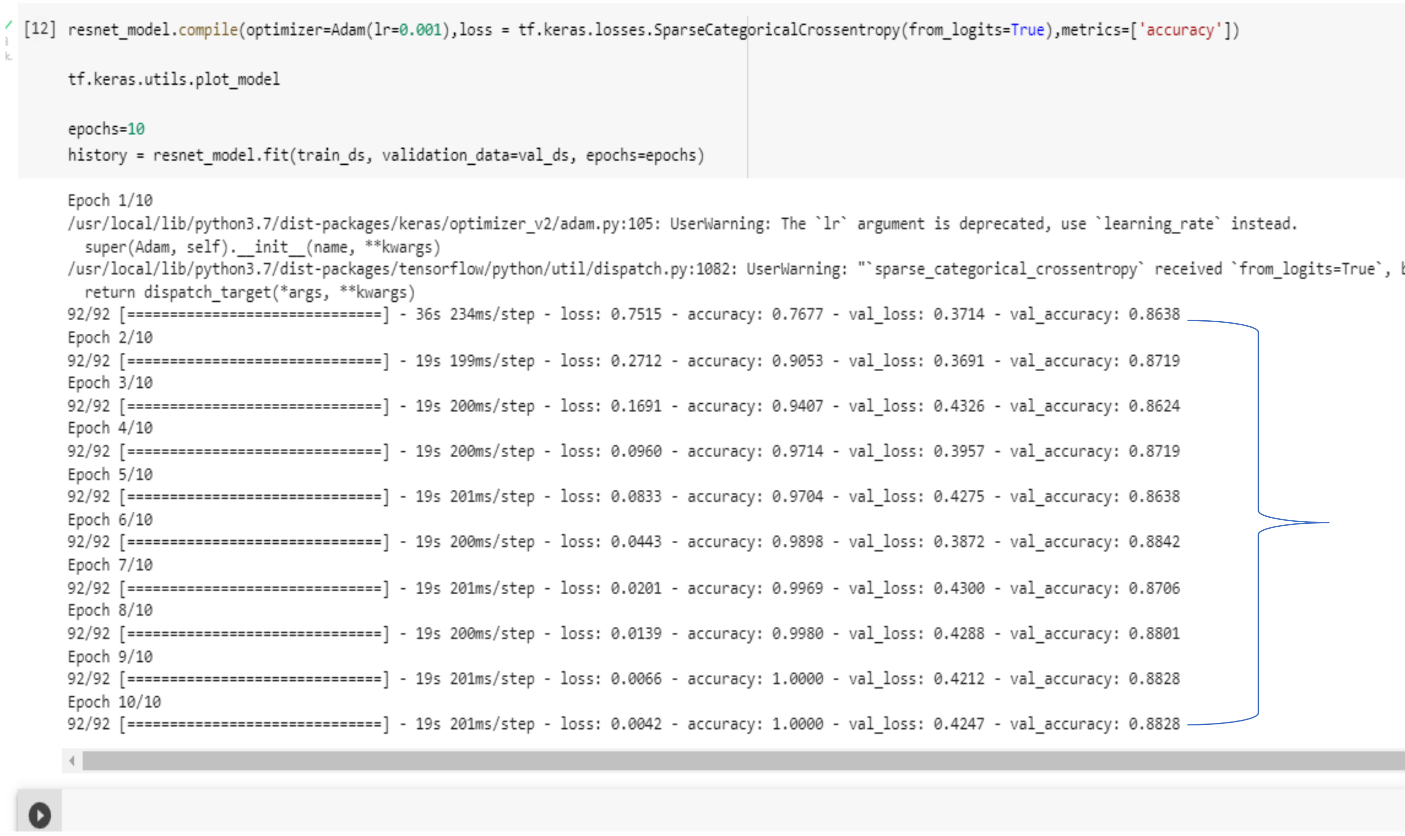
```
[ ] resnet_model.summary()
```

```
Model: "sequential"
_________________________________________________________________
 Layer (type)                Output Shape              Param #
=================================================================
 resnet50 (Functional)       (None, 2048)              23587712

 flatten (Flatten)           (None, 2048)              0

 dense (Dense)               (None, 512)               1049088

 dense_1 (Dense)             (None, 5)                 2565

=================================================================
Total params: 24,639,365
Trainable params: 1,051,653
Non-trainable params: 23,587,712
_________________________________________________________________
```

The key point to note over here is that the total number of parameters in the Resnet50 model is 24 million. But the trainable parameters are only 1 million.

That is precisely how transfer learning saves us massive time,space and computational complexity.

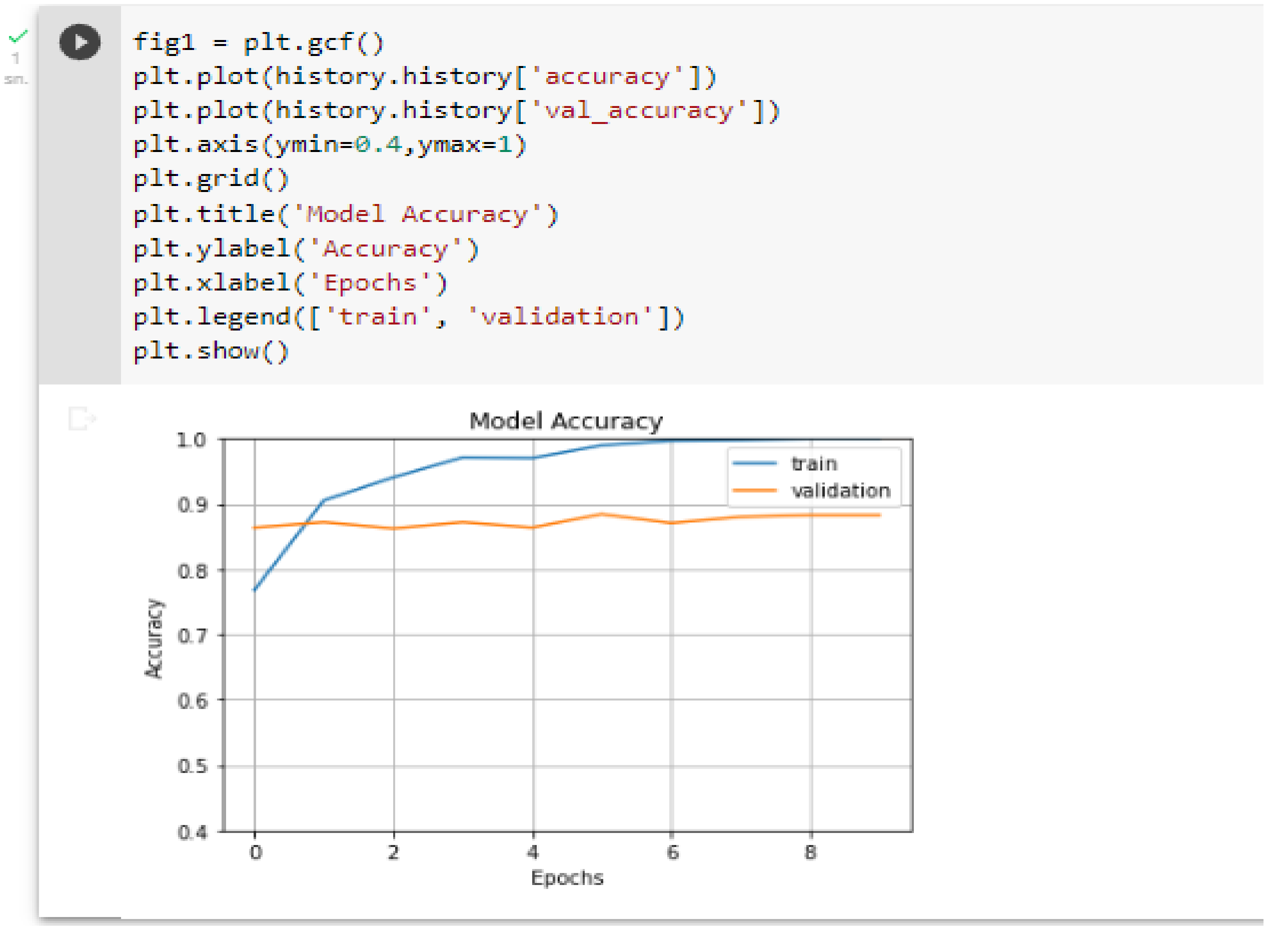
Now that our model is ready we simply compile it and train it over 10 epochs for now



```
[12] resnet_model.compile(optimizer=Adam(lr=0.001),loss = tf.keras.losses.SparseCategoricalCrossentropy(from_logits=True),metrics=['accuracy'])

     tf.keras.utils.plot_model

     epochs=10
     history = resnet_model.fit(train_ds, validation_data=val_ds, epochs=epochs)
```

```
Epoch 1/10
/usr/local/lib/python3.7/dist-packages/keras/optimizer_v2/adam.py:105: UserWarning: The `lr` argument is deprecated, use `learning_rate` instead.
  super(Adam, self).__init__(name, **kwargs)
/usr/local/lib/python3.7/dist-packages/tensorflow/python/util/dispatch.py:1082: UserWarning: "`sparse_categorical_crossentropy` received `from_logits=True`, b
  return dispatch_target(*args, **kwargs)
92/92 [==============================] - 36s 234ms/step - loss: 0.7515 - accuracy: 0.7677 - val_loss: 0.3714 - val_accuracy: 0.8638
Epoch 2/10
92/92 [==============================] - 19s 199ms/step - loss: 0.2712 - accuracy: 0.9053 - val_loss: 0.3691 - val_accuracy: 0.8719
Epoch 3/10
92/92 [==============================] - 19s 200ms/step - loss: 0.1691 - accuracy: 0.9407 - val_loss: 0.4326 - val_accuracy: 0.8624
Epoch 4/10
92/92 [==============================] - 19s 200ms/step - loss: 0.0960 - accuracy: 0.9714 - val_loss: 0.3957 - val_accuracy: 0.8719
Epoch 5/10
92/92 [==============================] - 19s 201ms/step - loss: 0.0833 - accuracy: 0.9704 - val_loss: 0.4275 - val_accuracy: 0.8638
Epoch 6/10
92/92 [==============================] - 19s 200ms/step - loss: 0.0443 - accuracy: 0.9898 - val_loss: 0.3872 - val_accuracy: 0.8842
Epoch 7/10
92/92 [==============================] - 19s 201ms/step - loss: 0.0201 - accuracy: 0.9969 - val_loss: 0.4300 - val_accuracy: 0.8706
Epoch 8/10
92/92 [==============================] - 19s 200ms/step - loss: 0.0139 - accuracy: 0.9980 - val_loss: 0.4288 - val_accuracy: 0.8801
Epoch 9/10
92/92 [==============================] - 19s 201ms/step - loss: 0.0066 - accuracy: 1.0000 - val_loss: 0.4212 - val_accuracy: 0.8828
Epoch 10/10
92/92 [==============================] - 19s 201ms/step - loss: 0.0042 - accuracy: 1.0000 - val_loss: 0.4247 - val_accuracy: 0.8828
```

Once your model is trained, we move over to the next step which is evaluating the model

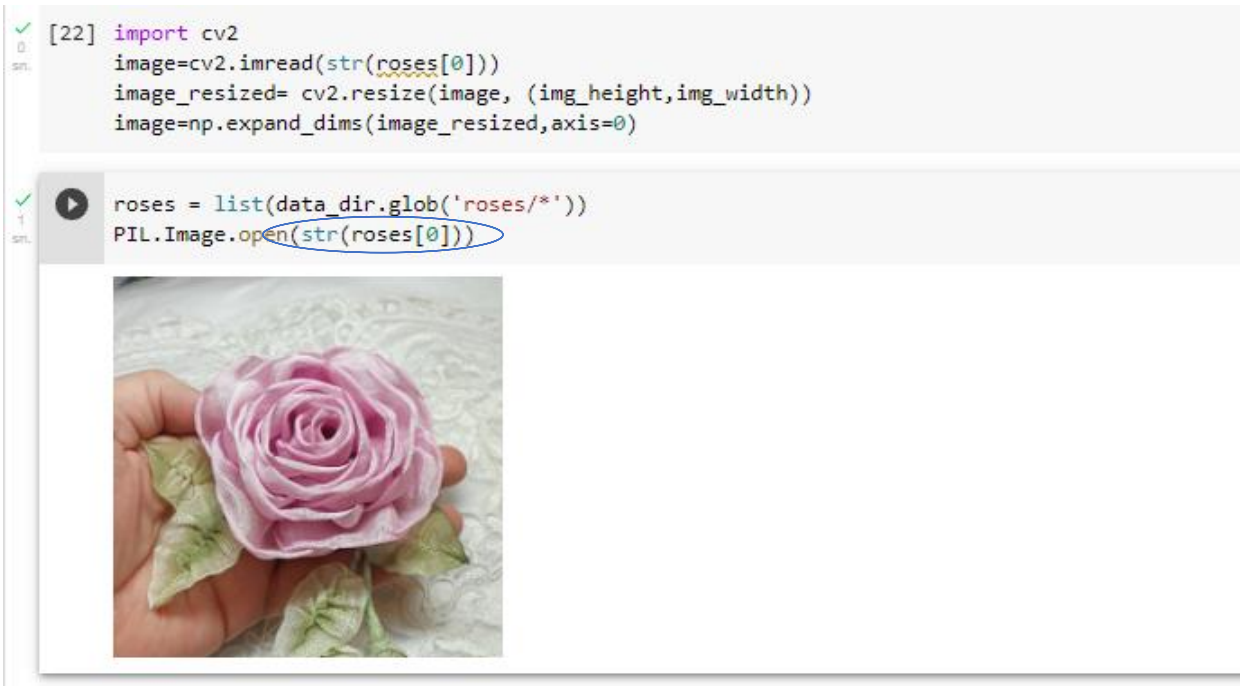


```
fig1 = plt.gcf()
plt.plot(history.history['accuracy'])
plt.plot(history.history['val_accuracy'])
plt.axis(ymin=0.4,ymax=1)
plt.grid()
plt.title('Model Accuracy')
plt.ylabel('Accuracy')
plt.xlabel('Epochs')
plt.legend(['train', 'validation'])
plt.show()
```

- matplotlib library to plot the train and validation accuracy with respect to each epoch. These logs had been stored in the history variable during the time of training.

The model does seem to have overfit a little bit, but will talk about measures to prevent over fitting in some other blog. For now since the validation accuracy is good enough(around 90%), we will proceed with the final step of making predictions with our mode.

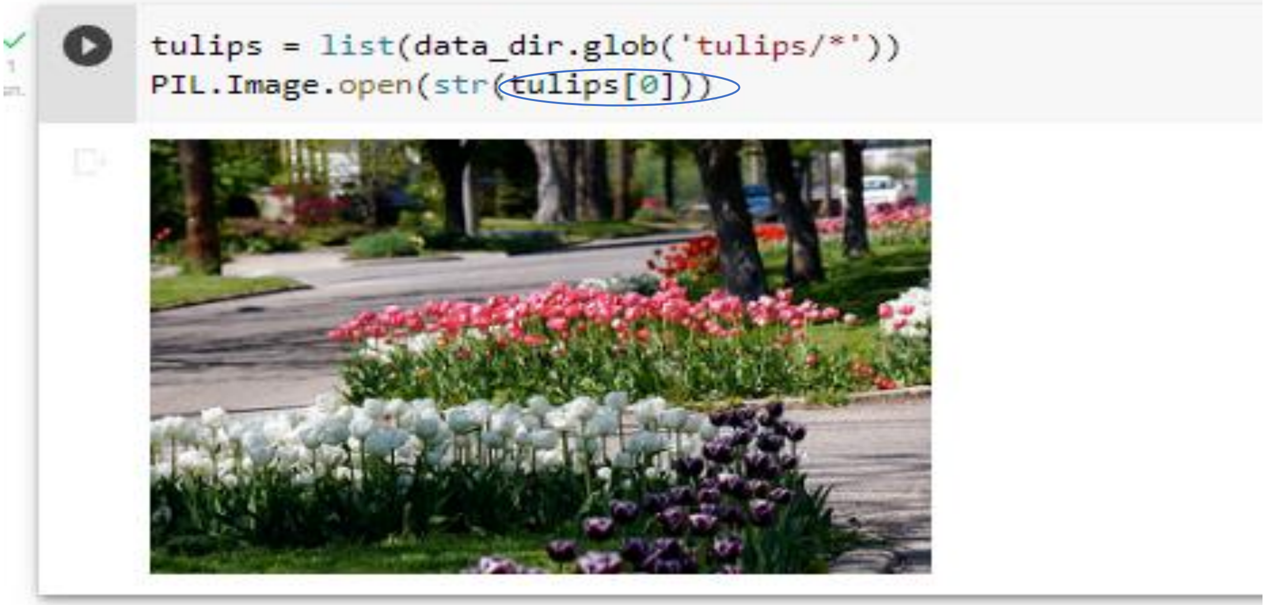
For now we will run predictions on a sample image of rose from our data



```
[22] import cv2
     image=cv2.imread(str(roses[0]))
     image_resized= cv2.resize(image, (img_height,img_width))
     image=np.expand_dims(image_resized,axis=0)
```

```
roses = list(data_dir.glob('roses/*'))
PIL.Image.open(str(roses[0]))
```

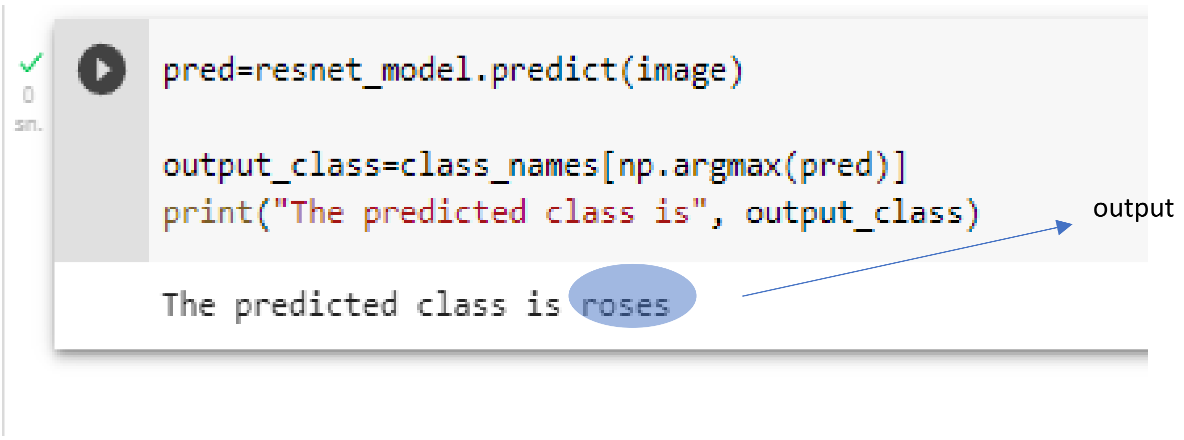
Let's take a look at our tulips, for example.



```
tulips = list(data_dir.glob('tulips/*'))
PIL.Image.open(str(tulips[0]))
```

To make predictions, we simply call the predict method

However, when trying print the predictions, you will receive an array of 5 numbers, since we used the softmax classifier. To get an output label prediction we execute the following code

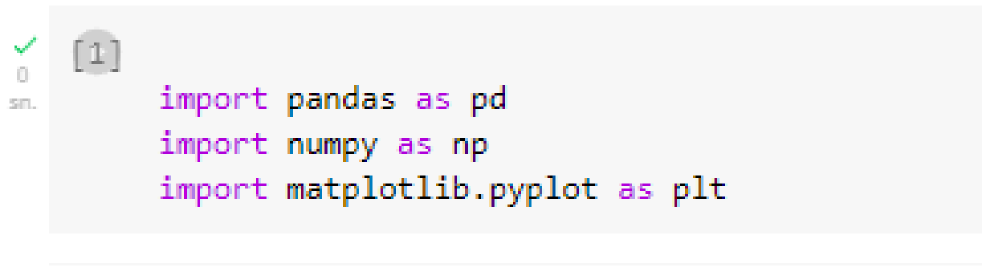


```
pred=resnet_model.predict(image)

output_class=class_names[np.argmax(pred)]
print("The predicted class is", output_class)
```

```
The predicted class is roses
```

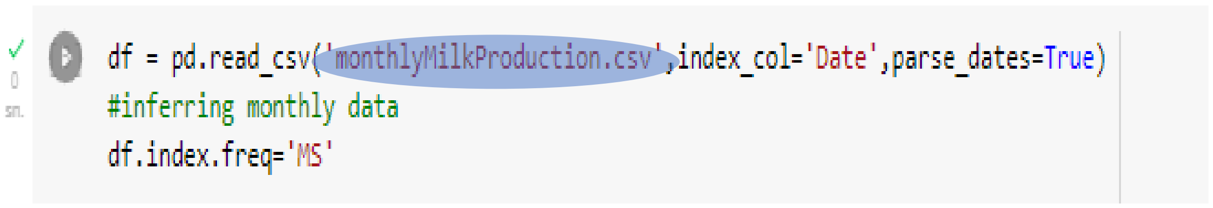
Question 2

Importing required libraries.

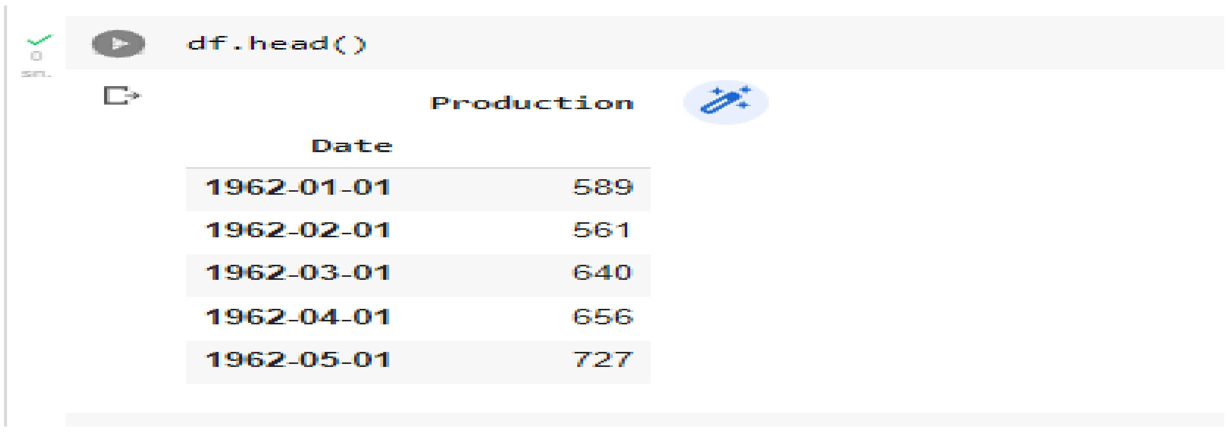


```
[1]
        import pandas as pd
        import numpy as np
        import matplotlib.pyplot as plt
```

Load our dataset and read head(5) data.



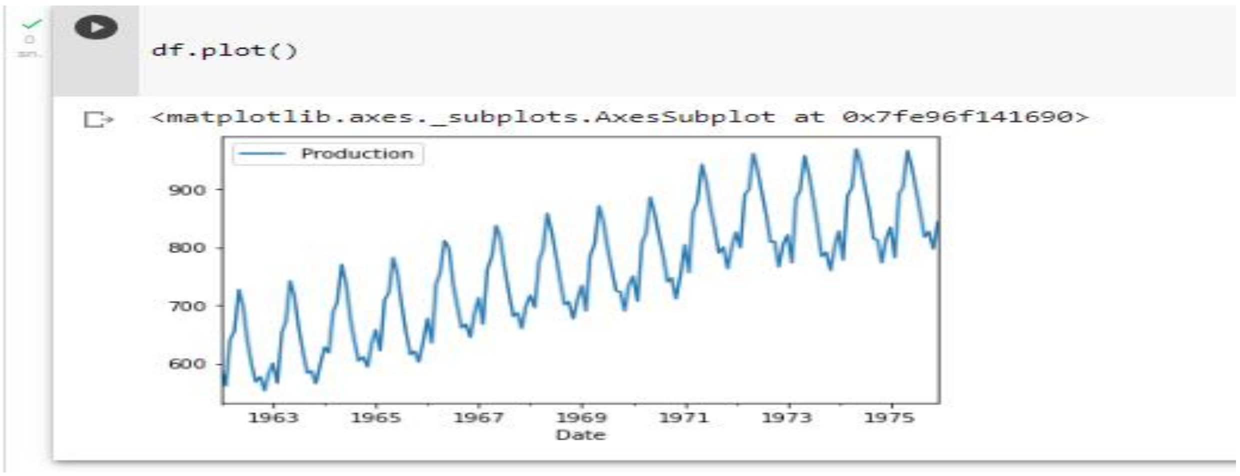
```
df = pd.read_csv('monthlyMilkProduction.csv',index_col='Date',parse_dates=True)
#inferring monthly data
df.index.freq='MS'
```



```
df.head()
```

Date	Production
1962-01-01	589
1962-02-01	561
1962-03-01	640
1962-04-01	656
1962-05-01	727

<u>Plotting data</u>

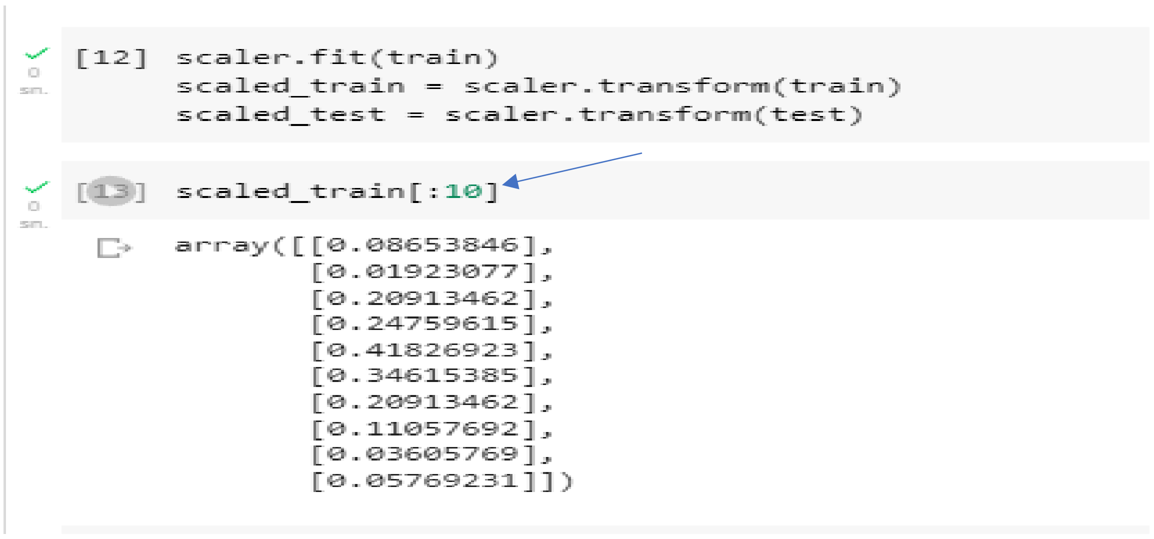


```
df.plot()
```

```
<matplotlib.axes._subplots.AxesSubplot at 0x7fe96f141690>
```

Scaling Data

Then just reshaping it to make just one column and n no. of rows where n represents the no. of elements in the array

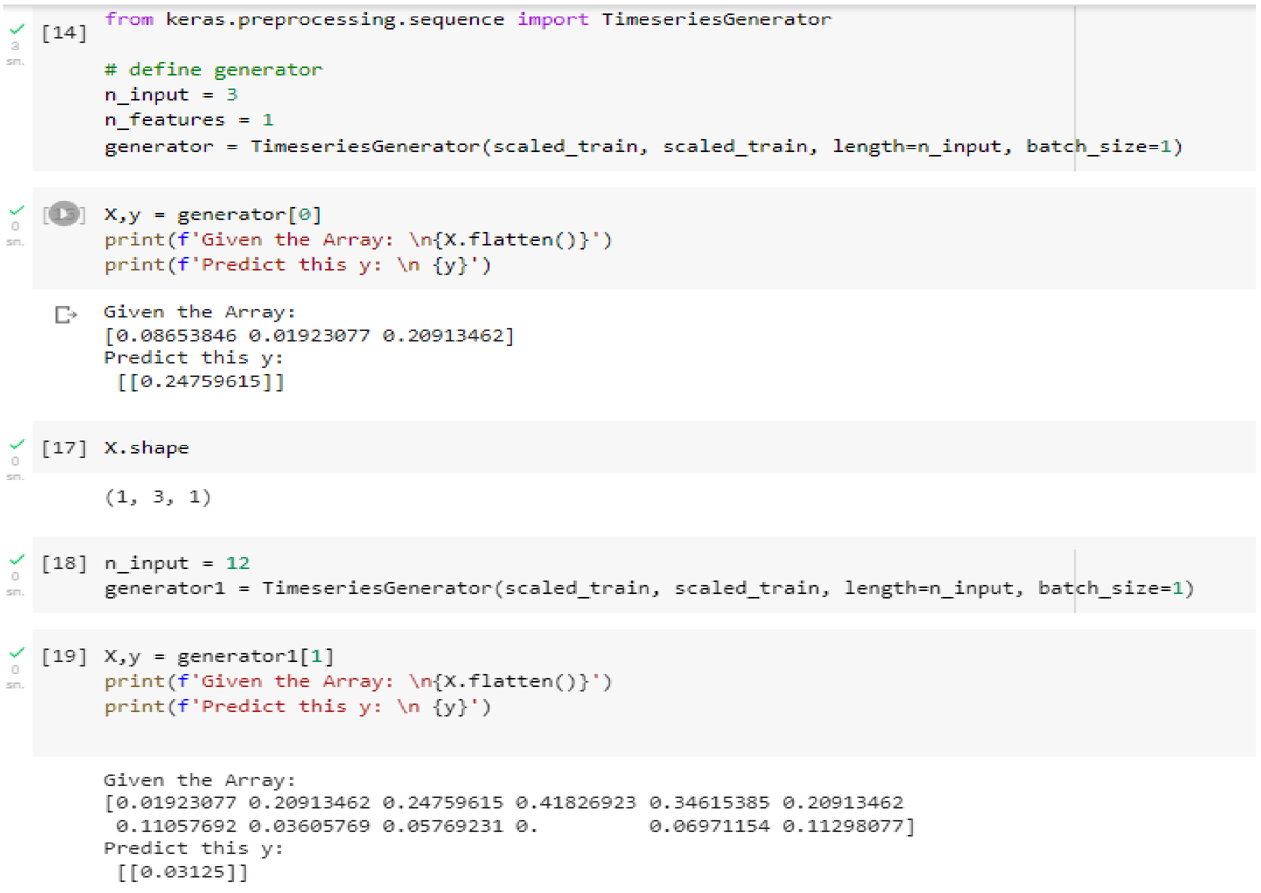


```
[12] scaler.fit(train)
     scaled_train = scaler.transform(train)
     scaled_test = scaler.transform(test)
```

```
[13] scaled_train[:10]
```

```
array([[0.08653846],
       [0.01923077],
       [0.20913462],
       [0.24759615],
       [0.41826923],
       [0.34615385],
       [0.20913462],
       [0.11057692],
       [0.03605769],
       [0.05769231]])
```

- Creating training data
- Getting train data in shape
- Checking train data and train labels



```
[14] from keras.preprocessing.sequence import TimeseriesGenerator

     # define generator
     n_input = 3
     n_features = 1
     generator = TimeseriesGenerator(scaled_train, scaled_train, length=n_input, batch_size=1)
```

```
[15] X,y = generator[0]
     print(f'Given the Array: \n{X.flatten()}')
     print(f'Predict this y: \n {y}')
```

```
Given the Array:
[0.08653846 0.01923077 0.20913462]
Predict this y:
 [[0.24759615]]
```

```
[17] X.shape
```

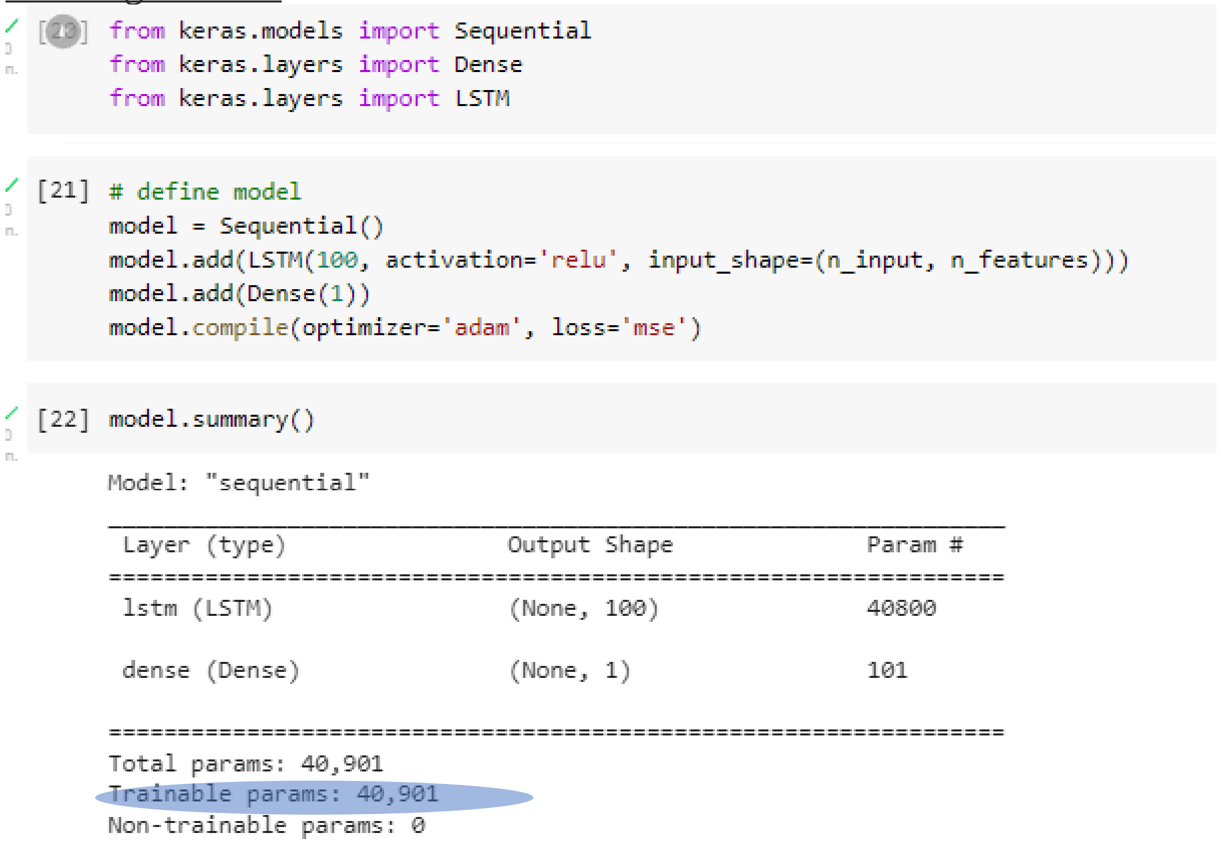
```
(1, 3, 1)
```

```
[18] n_input = 12
     generator1 = TimeseriesGenerator(scaled_train, scaled_train, length=n_input, batch_size=1)
```

```
[19] X,y = generator1[1]
     print(f'Given the Array: \n{X.flatten()}')
     print(f'Predict this y: \n {y}')
```

```
Given the Array:
[0.01923077 0.20913462 0.24759615 0.41826923 0.34615385 0.20913462
 0.11057692 0.03605769 0.05769231 0.         0.06971154 0.11298077]
Predict this y:
 [[0.03125]]
```

Creating a model



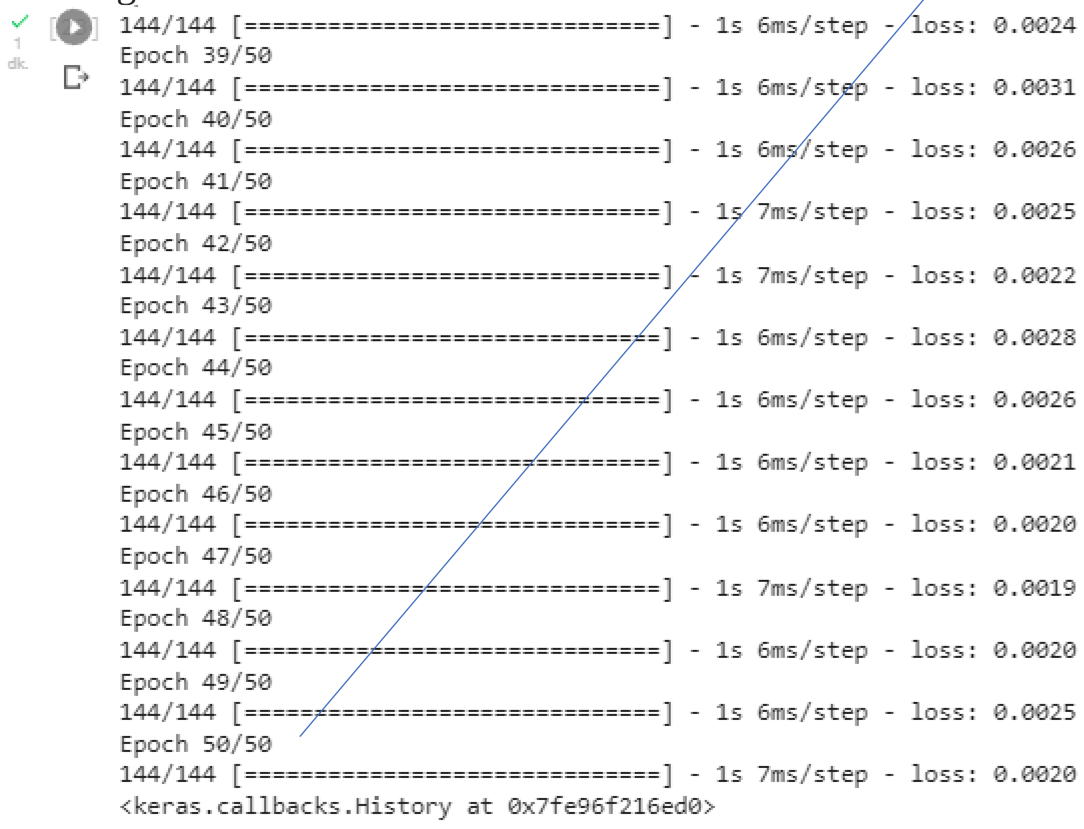
```
from keras.models import Sequential
from keras.layers import Dense
from keras.layers import LSTM
```

```
[21] # define model
     model = Sequential()
     model.add(LSTM(100, activation='relu', input_shape=(n_input, n_features)))
     model.add(Dense(1))
     model.compile(optimizer='adam', loss='mse')
```

```
[22] model.summary()

     Model: "sequential"
     _________________________________________________________________
      Layer (type)                Output Shape              Param #
     =================================================================
      lstm (LSTM)                 (None, 100)               40800

      dense (Dense)               (None, 1)                 101

     =================================================================
     Total params: 40,901
     Trainable params: 40,901
     Non-trainable params: 0
```

epochs=50

Training model

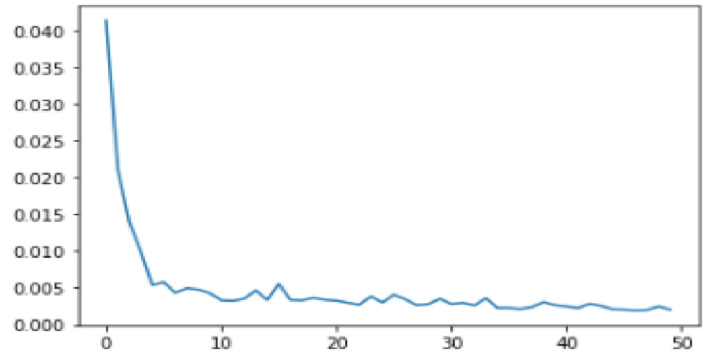


```
144/144 [==============================] - 1s 6ms/step - loss: 0.0024
Epoch 39/50
144/144 [==============================] - 1s 6ms/step - loss: 0.0031
Epoch 40/50
144/144 [==============================] - 1s 6ms/step - loss: 0.0026
Epoch 41/50
144/144 [==============================] - 1s 7ms/step - loss: 0.0025
Epoch 42/50
144/144 [==============================] - 1s 7ms/step - loss: 0.0022
Epoch 43/50
144/144 [==============================] - 1s 6ms/step - loss: 0.0028
Epoch 44/50
144/144 [==============================] - 1s 6ms/step - loss: 0.0026
Epoch 45/50
144/144 [==============================] - 1s 6ms/step - loss: 0.0021
Epoch 46/50
144/144 [==============================] - 1s 6ms/step - loss: 0.0020
Epoch 47/50
144/144 [==============================] - 1s 7ms/step - loss: 0.0019
Epoch 48/50
144/144 [==============================] - 1s 6ms/step - loss: 0.0020
Epoch 49/50
144/144 [==============================] - 1s 6ms/step - loss: 0.0025
Epoch 50/50
144/144 [==============================] - 1s 7ms/step - loss: 0.0020
<keras.callbacks.History at 0x7fe96f216ed0>
```

Plotting loss curve for Milk Production prediction model

```
loss_per_epoch = model.history.history['loss']
plt.plot(range(len(loss_per_epoch)),loss_per_epoch)
```

```
[<matplotlib.lines.Line2D at 0x7fe96c8d6dd0>]
```



```
[25]
```

- Creating a seed for next year's Milk Production prediction.
- Next year's Milk Production prediction

```
[25] last_train_batch = scaled_train[-12:]

     last_train_batch = last_train_batch.reshape((1, n_input, n_features))
```

```
[26] model.predict(last_train_batch)

     array([[0.6502957]], dtype=float32)
```

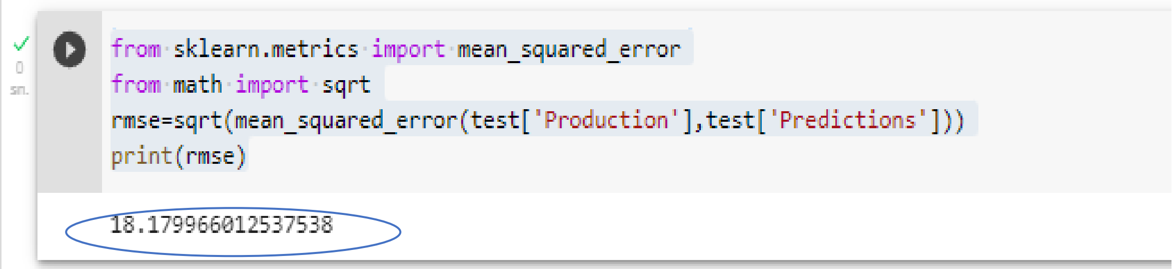
```
[27] scaled_test[0]

     test_predictions = []
```

```
[28] first_eval_batch = scaled_train[-n_input:]
     current_batch = first_eval_batch.reshape((1, n_input, n_features))
```

```
[29] for i in range(len(test)):

         # get the prediction value for the first batch
         current_pred = model.predict(current_batch)[0]

         # append the prediction into the array
         test_predictions.append(current_pred)

         # use the prediction to update the batch and remove the first value
         current_batch = np.append(current_batch[:,1:,:],[[current_pred]],axis=1)
```

```
[30] test_predictions

     [array([0.6502957], dtype=float32),
      array([0.6048693], dtype=float32),
      array([0.79876465], dtype=float32),
      array([0.8663778], dtype=float32),
      array([0.98100835], dtype=float32),
      array([0.96684647], dtype=float32),
      array([0.89425874], dtype=float32),
      array([0.79253477], dtype=float32),
      array([0.6775114], dtype=float32),
      array([0.64138794], dtype=float32),
      array([0.58192366], dtype=float32),
      array([0.620713], dtype=float32)]
```

regression loss



```
from sklearn.metrics import mean_squared_error
from math import sqrt
rmse=sqrt(mean_squared_error(test['Production'],test['Predictions']))
print(rmse)
```

```
18.179966012537538
```

Compare LSTM vs StackedLSTM

There are some cases, however, we can see the sequence in 2 dimensions, or more. For example imagine you have a photo, and you want to predict the values of pixel (x,y), you would probably think this pixel depends on both (x-1, y) and (x, y-1). So the "sequence" of pixels grows in both dimensions, and in this case it is in 2D space, instead of in the time dimension.

Concretely, multidimensional LSTM units have recurrent connections in multiple dimensions. For example, in 2D LSTM units (used in state-of-the-art handwriting recognition systems) the hidden value h_{x, y} depends on the values of h_{x-1, y} and h_{x,y-1}. So you can imagine the formulas of LSTM gates and cells have to be updated to take into account those recurrent connections.

Stack of LSTM, on other hands, is simply a stack of several LSTM layers, so you can stack any LSTM layers you want. For example, you can stack several 2D-LSTM layers and form a deep network, not only in terms of layers, but also in the recurrent dimensions. The intuition is that higher LSTM layers can capture abstract concepts in the sequences, which might help for the task at hand.