“Neural Networks”

Shervin Halat

98131018

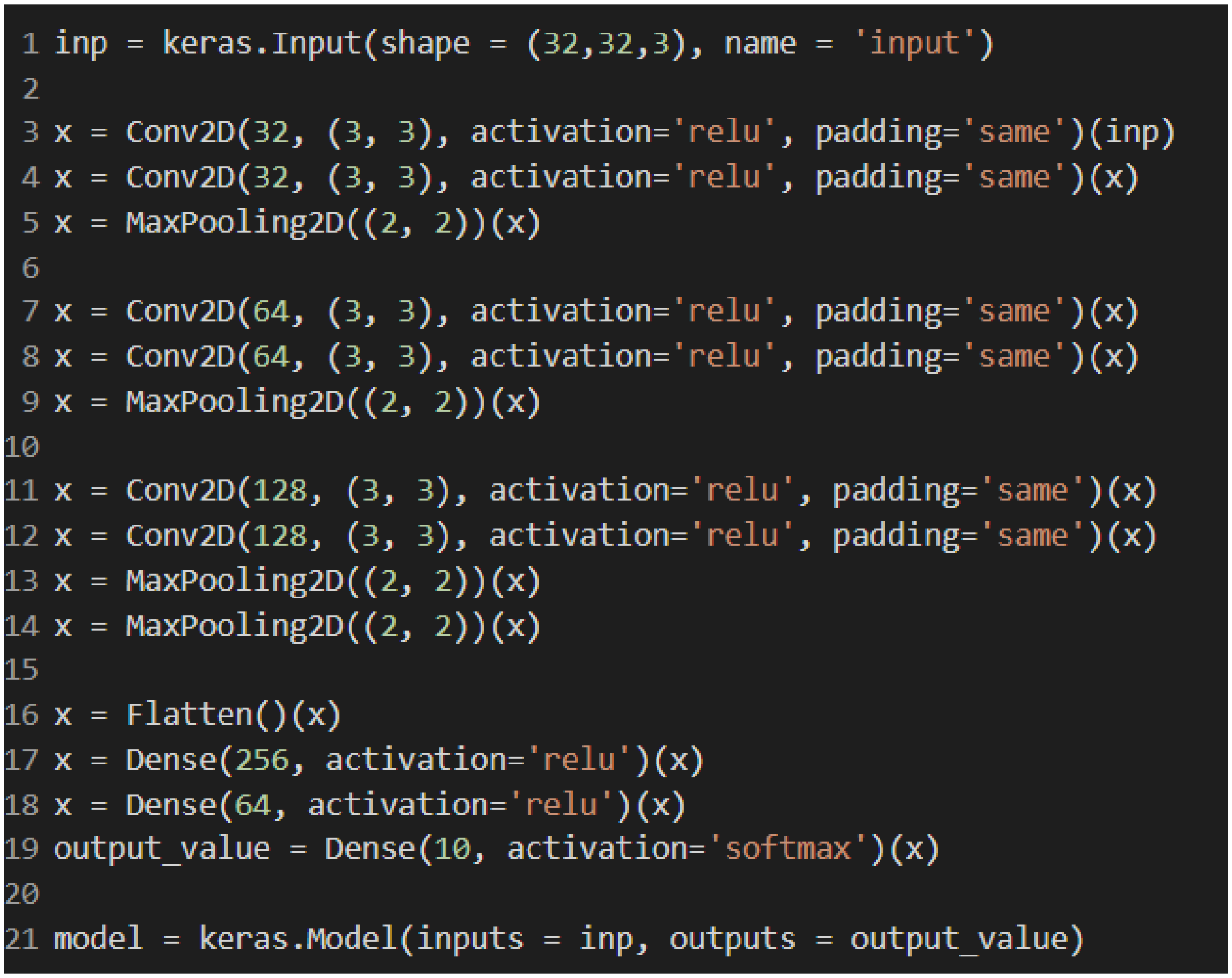
Homework 3

“First Part”

1.

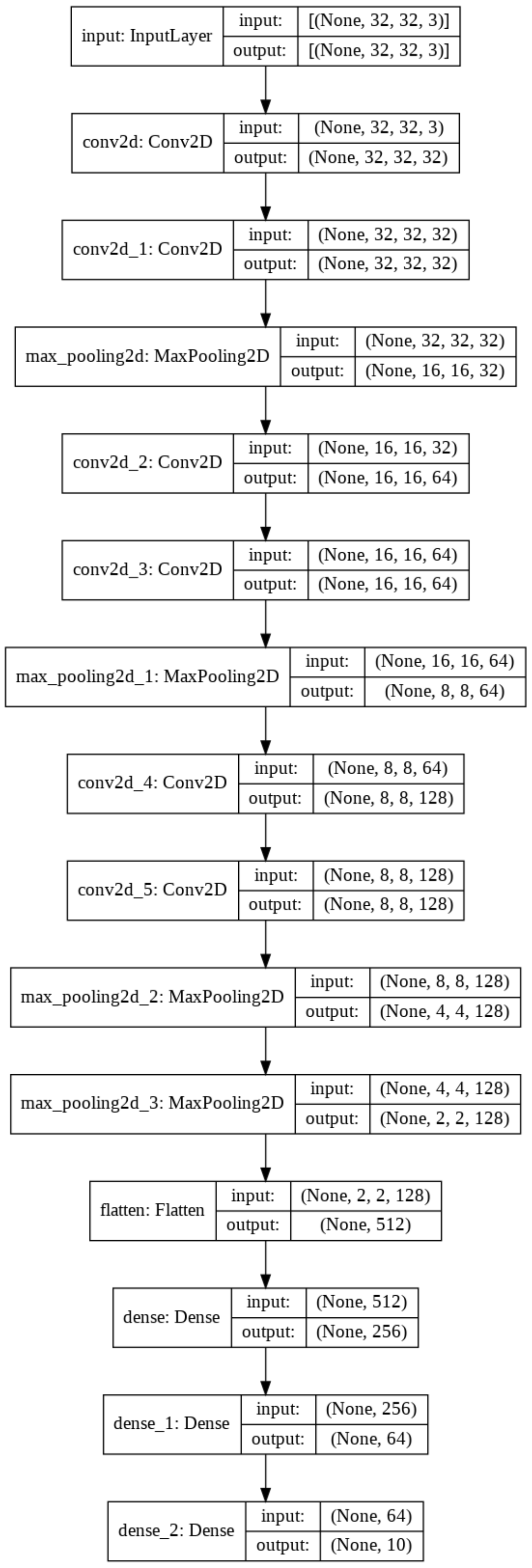
Considering the following model as the base-model, which is inspired by VGG net, required investigations on different properties were conducted.

The base model consists of three blocks with the following configurations:

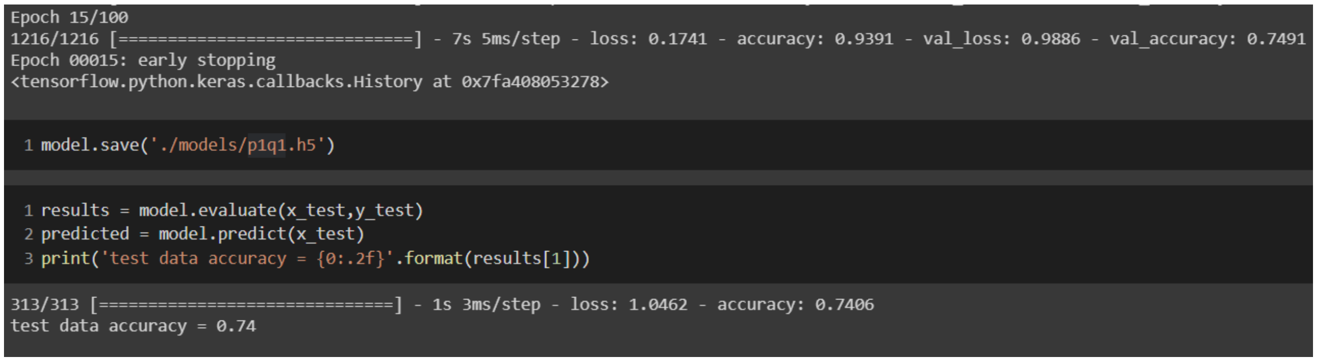


```python
inp = keras.Input(shape = (32,32,3), name = 'input')

x = Conv2D(32, (3, 3), activation='relu', padding='same')(inp)
x = Conv2D(32, (3, 3), activation='relu', padding='same')(x)
x = MaxPooling2D((2, 2))(x)

x = Conv2D(64, (3, 3), activation='relu', padding='same')(x)
x = Conv2D(64, (3, 3), activation='relu', padding='same')(x)
x = MaxPooling2D((2, 2))(x)

x = Conv2D(128, (3, 3), activation='relu', padding='same')(x)
x = Conv2D(128, (3, 3), activation='relu', padding='same')(x)
x = MaxPooling2D((2, 2))(x)
x = MaxPooling2D((2, 2))(x)

x = Flatten()(x)
x = Dense(256, activation='relu')(x)
x = Dense(64, activation='relu')(x)
output_value = Dense(10, activation='softmax')(x)

model = keras.Model(inputs = inp, outputs = output_value)
```

And the base-model is shown below:


input: InputLayer
input: [(None, 32, 32, 3)]
output: [(None, 32, 32, 3)]
conv2d: Conv2D
input: (None, 32, 32, 3)
output: (None, 32, 32, 32)
conv2d_1: Conv2D
input: (None, 32, 32, 32)
output: (None, 32, 32, 32)
max_pooling2d: MaxPooling2D
input: (None, 32, 32, 32)
output: (None, 16, 16, 32)
conv2d_2: Conv2D
input: (None, 16, 16, 32)
output: (None, 16, 16, 64)
conv2d_3: Conv2D
input: (None, 16, 16, 64)
output: (None, 16, 16, 64)
max_pooling2d_1: MaxPooling2D
input: (None, 16, 16, 64)
output: (None, 8, 8, 64)
conv2d_4: Conv2D
input: (None, 8, 8, 64)
output: (None, 8, 8, 128)
conv2d_5: Conv2D
input: (None, 8, 8, 128)
output: (None, 8, 8, 128)
max_pooling2d_2: MaxPooling2D
input: (None, 8, 8, 128)
output: (None, 4, 4, 128)
max_pooling2d_3: MaxPooling2D
input: (None, 4, 4, 128)
output: (None, 2, 2, 128)
flatten: Flatten
input: (None, 2, 2, 128)
output: (None, 512)
dense: Dense
input: (None, 512)
output: (None, 256)
dense_1: Dense
input: (None, 256)
output: (None, 64)
dense_2: Dense
input: (None, 64)
output: (None, 10)

The trained base-model reached 74% accuracy on test set after 15 epochs which stopped by early stopping callback.



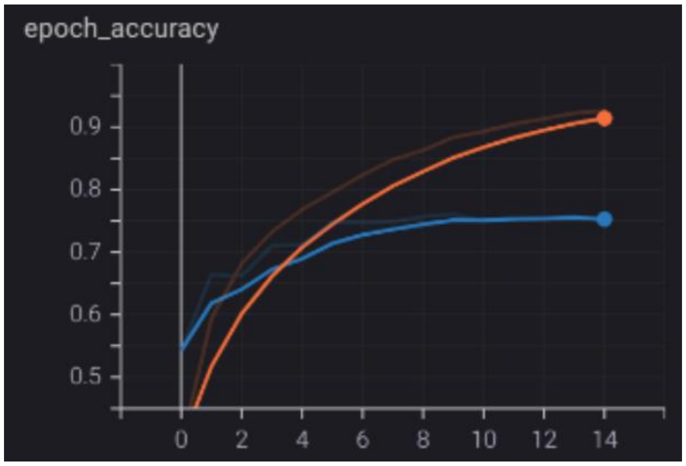
```
Epoch 15/100
1216/1216 [==============================] - 7s 5ms/step - loss: 0.1741 - accuracy: 0.9391 - val_loss: 0.9886 - val_accuracy: 0.7491
Epoch 00015: early stopping
<tensorflow.python.keras.callbacks.History at 0x7fa408053278>
```

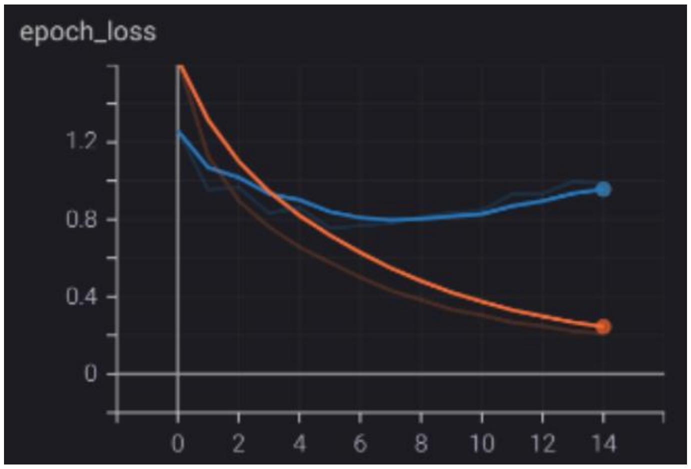
```
model.save('./models/p1q1.h5')
```

```
results = model.evaluate(x_test,y_test)
predicted = model.predict(x_test)
print('test data accuracy = {0:.2f}'.format(results[1]))
```

```
313/313 [==============================] - 1s 3ms/step - loss: 1.0462 - accuracy: 0.7406
test data accuracy = 0.74
```

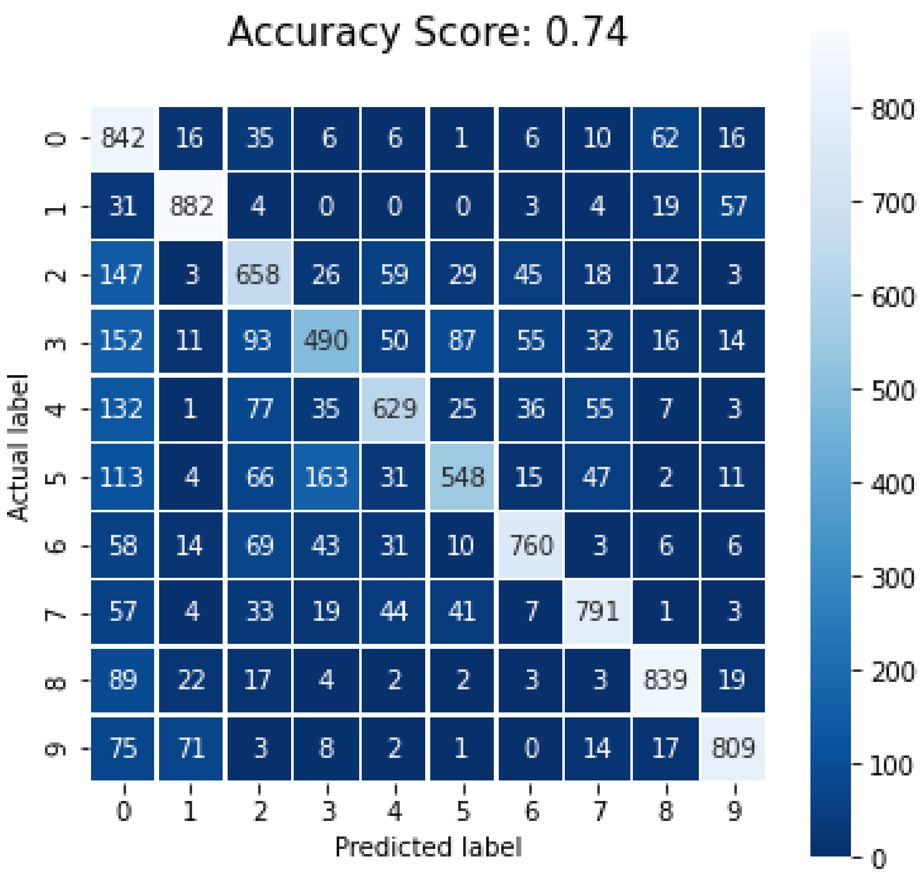
The training logs are as follows:





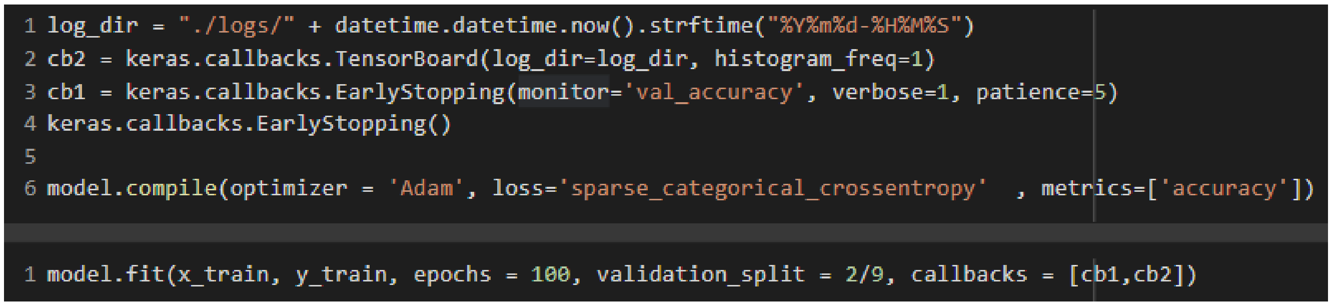
As it can be figured out from figures above, the model has overfitted on the training set. This problem is addressed by regularization in the next question (Regularization).

The confusion matrix of the trained base-model is as follows:



Actual \ Predicted	0	1	2	3	4	5	6	7	8	9
0	842	16	35	6	6	1	6	10	62	16
1	31	882	4	0	0	0	3	4	19	57
2	147	3	658	26	59	29	45	18	12	3
3	152	11	93	490	50	87	55	32	16	14
4	132	1	77	35	629	25	36	55	7	3
5	113	4	66	163	31	548	15	47	2	11
6	58	14	69	43	31	10	760	3	6	6
7	57	4	33	19	44	41	7	791	1	3
8	89	22	17	4	2	2	3	3	839	19
9	75	71	3	8	2	1	0	14	17	809

All models of this part, including the base-model, are trained by the following configurations:



```
log_dir = "./logs/" + datetime.datetime.now().strftime("%Y%m%d-%H%M%S")
cb2 = keras.callbacks.TensorBoard(log_dir=log_dir, histogram_freq=1)
cb1 = keras.callbacks.EarlyStopping(monitor='val_accuracy', verbose=1, patience=5)
keras.callbacks.EarlyStopping()

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

```
model.fit(x_train, y_train, epochs = 100, validation_split = 2/9, callbacks = [cb1,cb2])
```

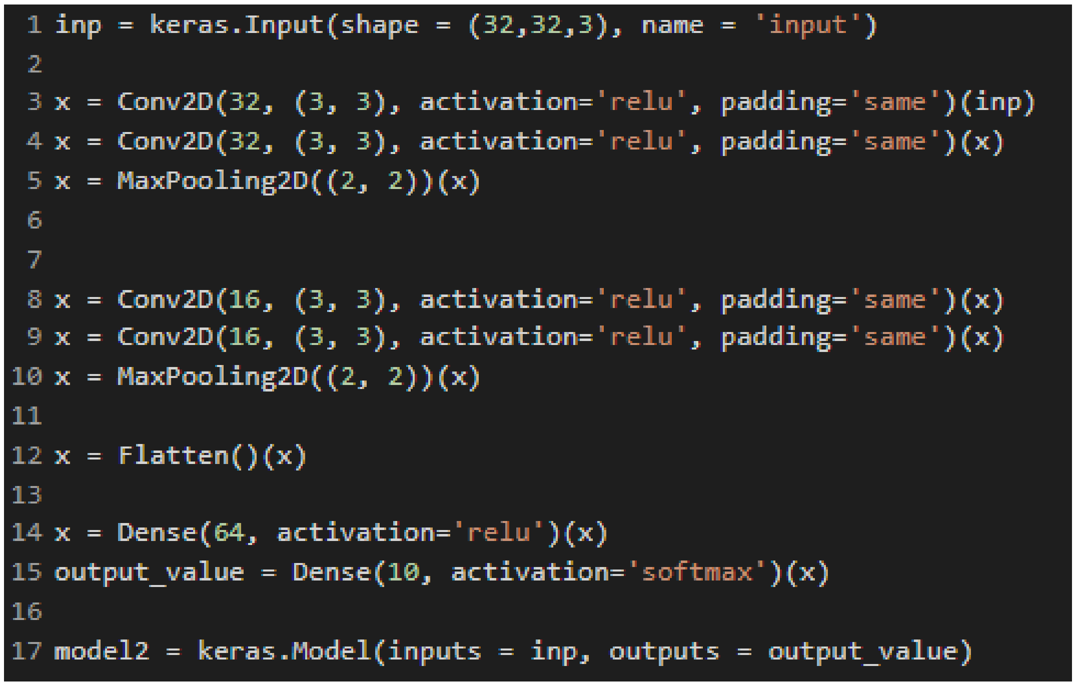
Remembering base-model’s test accuracy of 74%, now we go through the investigation on the effect of multiple properties:

- Number and sizes of hidden layers:

  To investigate these properties, number of blocks was changed from three blocks to two blocks with lower sizes and five blocks with higher sizes. The obtained test accuracies along with the corresponding configurations and logs are as follows:

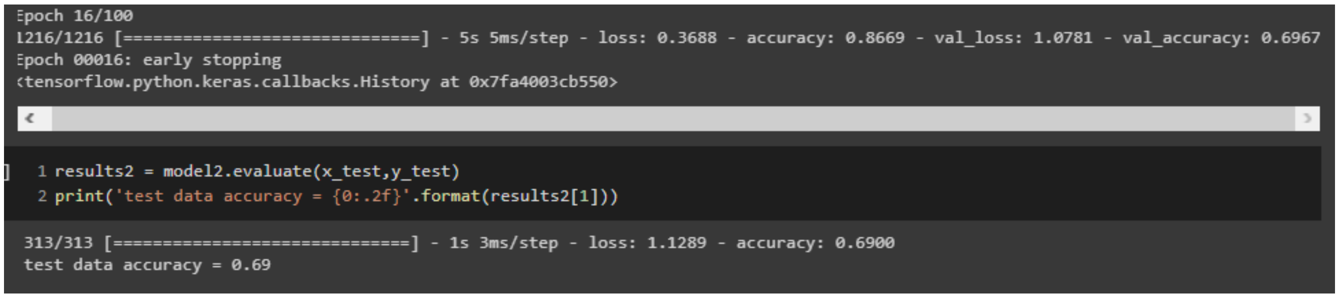
‘2 blocks’

Considering following configuration with less hidden layers and smaller sizes,



```python
inp = keras.Input(shape = (32,32,3), name = 'input')

x = Conv2D(32, (3, 3), activation='relu', padding='same')(inp)
x = Conv2D(32, (3, 3), activation='relu', padding='same')(x)
x = MaxPooling2D((2, 2))(x)


x = Conv2D(16, (3, 3), activation='relu', padding='same')(x)
x = Conv2D(16, (3, 3), activation='relu', padding='same')(x)
x = MaxPooling2D((2, 2))(x)

x = Flatten()(x)

x = Dense(64, activation='relu')(x)
output_value = Dense(10, activation='softmax')(x)

model2 = keras.Model(inputs = inp, outputs = output_value)
```

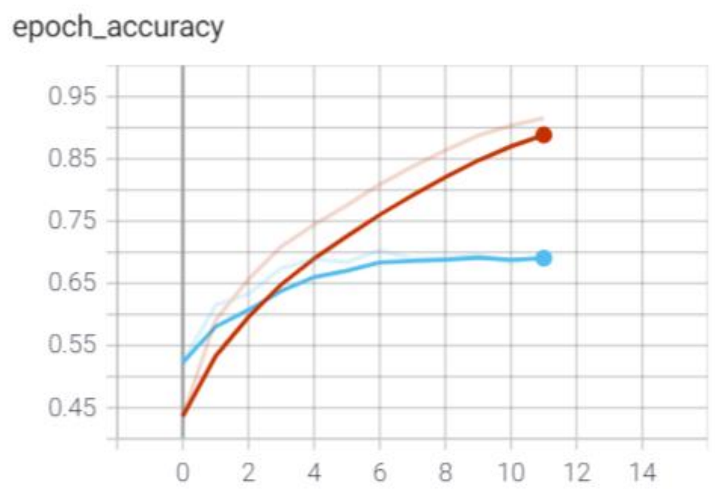
Following results will be obtained:

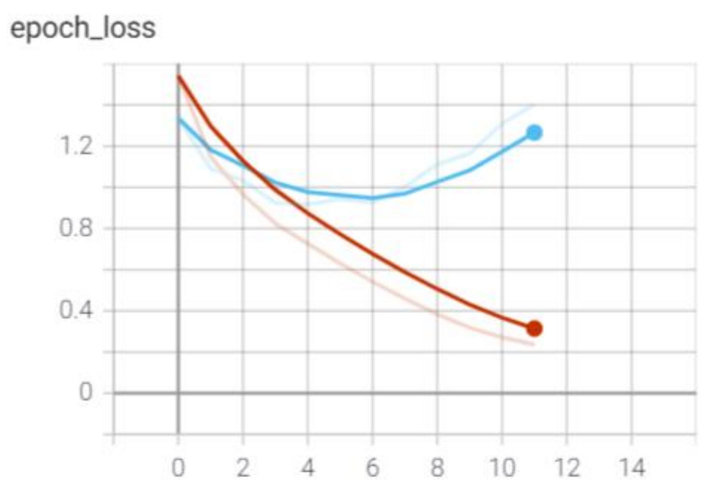


```
Epoch 16/100
1216/1216 [==============================] - 5s 5ms/step - loss: 0.3688 - accuracy: 0.8669 - val_loss: 1.0781 - val_accuracy: 0.6967
Epoch 00016: early stopping
<tensorflow.python.keras.callbacks.History at 0x7fa4003cb550>
```

```python
results2 = model2.evaluate(x_test,y_test)
print('test data accuracy = {0:.2f}'.format(results2[1]))
```

```
313/313 [==============================] - 1s 3ms/step - loss: 1.1289 - accuracy: 0.6900
test data accuracy = 0.69
```



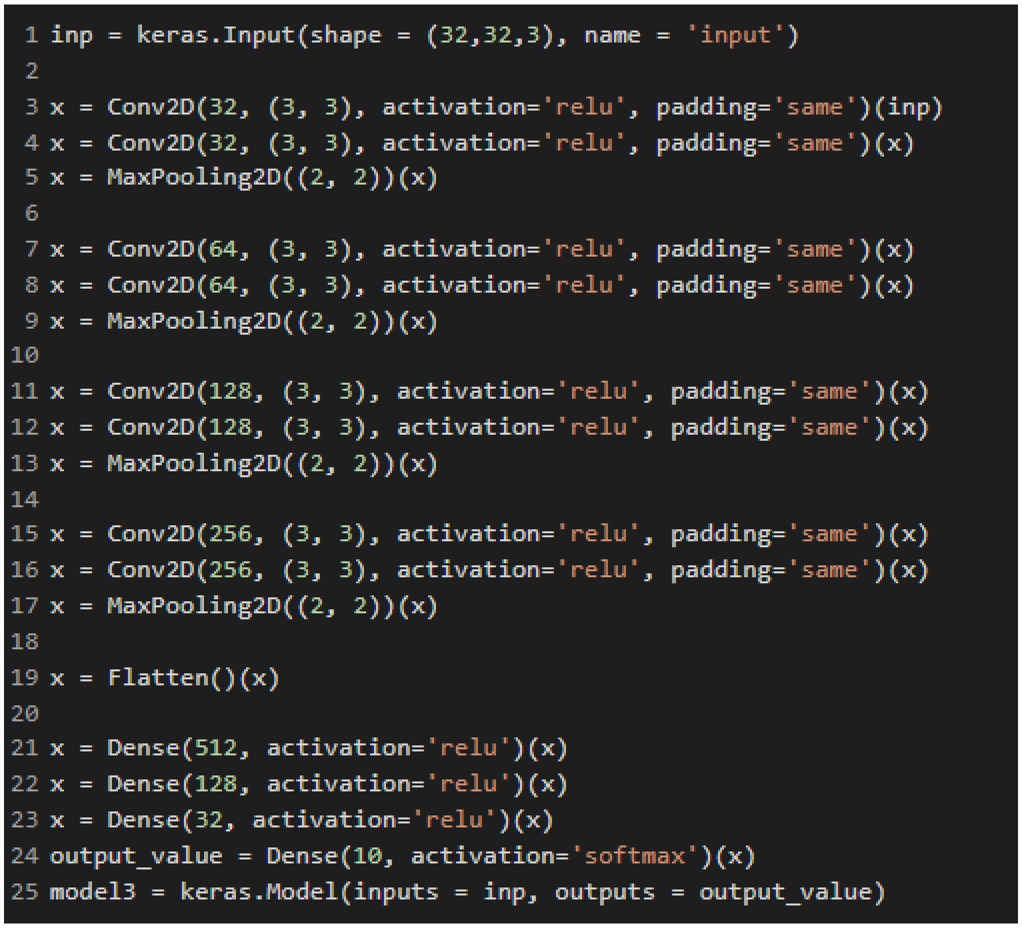


As we can see in the images above, the new model with 2 blocks has inferior test accuracy compared to the base-model with 3 blocks. The new model has reached 69% test accuracy after 16 epochs of training which is stopped by early stopping callback.

Overall, less hidden layers along with lower sizes results in lower performance compared to base-model.

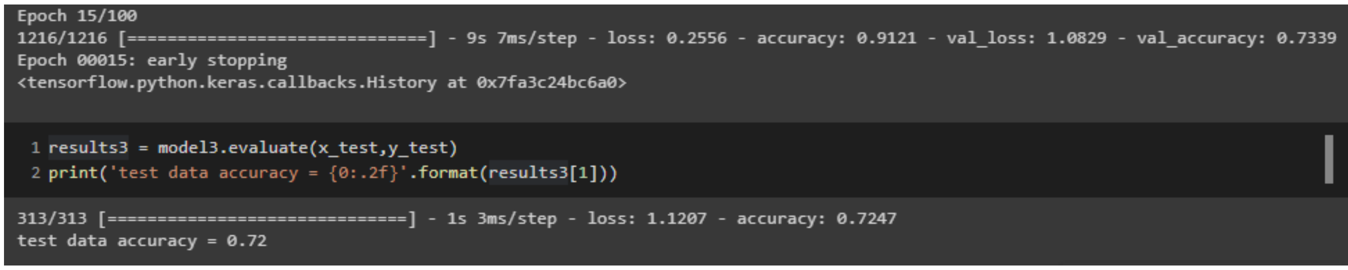
‘4 blocks’

Considering following configuration with more hidden layers and larger sizes,



```python
inp = keras.Input(shape = (32,32,3), name = 'input')

x = Conv2D(32, (3, 3), activation='relu', padding='same')(inp)
x = Conv2D(32, (3, 3), activation='relu', padding='same')(x)
x = MaxPooling2D((2, 2))(x)

x = Conv2D(64, (3, 3), activation='relu', padding='same')(x)
x = Conv2D(64, (3, 3), activation='relu', padding='same')(x)
x = MaxPooling2D((2, 2))(x)

x = Conv2D(128, (3, 3), activation='relu', padding='same')(x)
x = Conv2D(128, (3, 3), activation='relu', padding='same')(x)
x = MaxPooling2D((2, 2))(x)

x = Conv2D(256, (3, 3), activation='relu', padding='same')(x)
x = Conv2D(256, (3, 3), activation='relu', padding='same')(x)
x = MaxPooling2D((2, 2))(x)

x = Flatten()(x)

x = Dense(512, activation='relu')(x)
x = Dense(128, activation='relu')(x)
x = Dense(32, activation='relu')(x)
output_value = Dense(10, activation='softmax')(x)
model3 = keras.Model(inputs = inp, outputs = output_value)
```

Following results will be obtained:



```
Epoch 15/100
1216/1216 [==============================] - 9s 7ms/step - loss: 0.2556 - accuracy: 0.9121 - val_loss: 1.0829 - val_accuracy: 0.7339
Epoch 00015: early stopping
<tensorflow.python.keras.callbacks.History at 0x7fa3c24bc6a0>
```

```python
results3 = model3.evaluate(x_test,y_test)
print('test data accuracy = {0:.2f}'.format(results3[1]))
```

```
313/313 [==============================] - 1s 3ms/step - loss: 1.1207 - accuracy: 0.7247
test data accuracy = 0.72
```

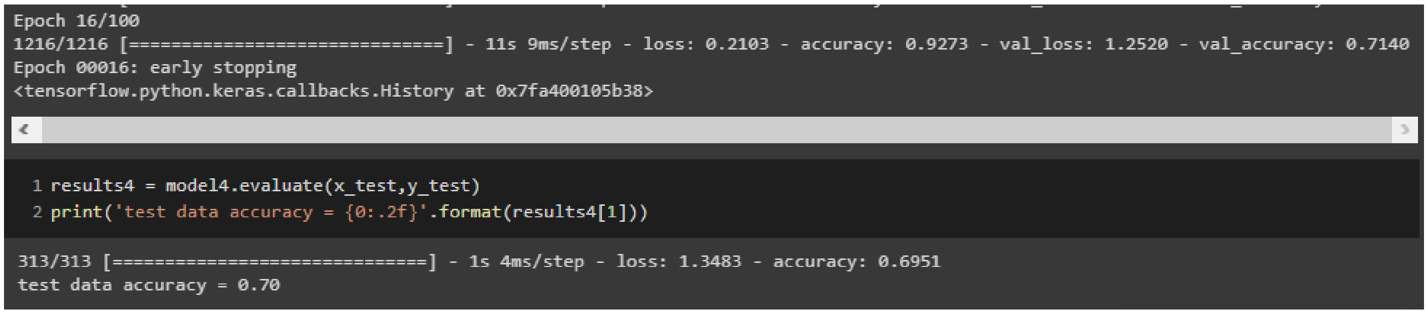
As we can see in the images above, the new model with 4 blocks has also inferior test accuracy compared to the base-model with 3 blocks. The new model has reached 72% test accuracy after 15 epochs of training which is stopped by early stopping callback.

In conclusion, it seems that there is an optimal number and sizes of hidden layers at which model has the best performance and lower or higher values result in lower performance of the model.

- Kernel sizes:

'Kernel size of 5'

By only increasing the base-models kernel size of 3 to 5, the following test accuracy of 70% will be obtained after 16 epochs.

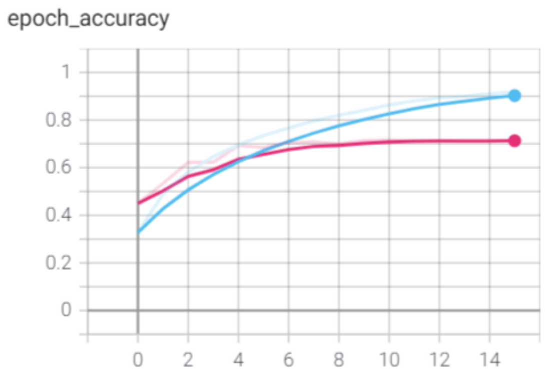


```
Epoch 16/100
1216/1216 [==============================] - 11s 9ms/step - loss: 0.2103 - accuracy: 0.9273 - val_loss: 1.2520 - val_accuracy: 0.7140
Epoch 00016: early stopping
<tensorflow.python.keras.callbacks.History at 0x7fa400105b38>
```

```
results4 = model4.evaluate(x_test,y_test)
print('test data accuracy = {0:.2f}'.format(results4[1]))
```

```
313/313 [==============================] - 1s 4ms/step - loss: 1.3483 - accuracy: 0.6951
test data accuracy = 0.70
```

It seems that increasing the kernel size leads to a slight decrease in model's performance.

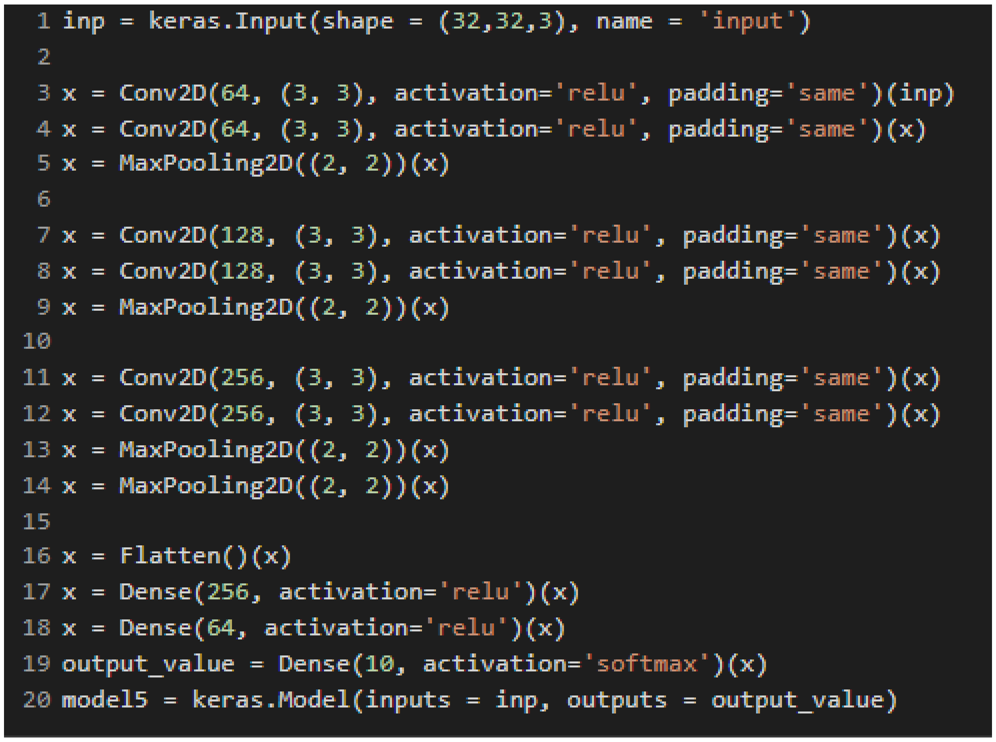


- Number of kernels:

  For this part, three conditions of doubling, halving, and quartering the number of kernels of each convolutional layer of the base-model has been investigated:

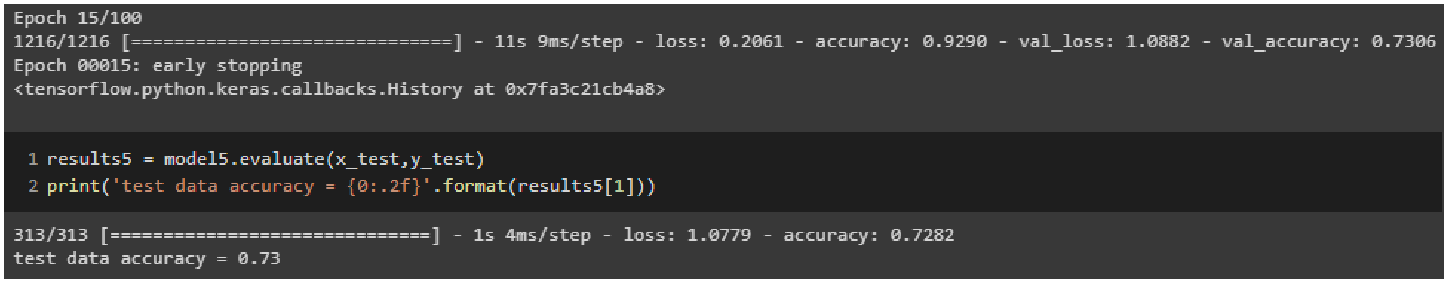
‘Doubling’

Considering the following configuration:



```
inp = keras.Input(shape = (32,32,3), name = 'input')

x = Conv2D(64, (3, 3), activation='relu', padding='same')(inp)
x = Conv2D(64, (3, 3), activation='relu', padding='same')(x)
x = MaxPooling2D((2, 2))(x)

x = Conv2D(128, (3, 3), activation='relu', padding='same')(x)
x = Conv2D(128, (3, 3), activation='relu', padding='same')(x)
x = MaxPooling2D((2, 2))(x)

x = Conv2D(256, (3, 3), activation='relu', padding='same')(x)
x = Conv2D(256, (3, 3), activation='relu', padding='same')(x)
x = MaxPooling2D((2, 2))(x)
x = MaxPooling2D((2, 2))(x)

x = Flatten()(x)
x = Dense(256, activation='relu')(x)
x = Dense(64, activation='relu')(x)
output_value = Dense(10, activation='softmax')(x)
model5 = keras.Model(inputs = inp, outputs = output_value)
```

As we can see in the following image, doubling number of kernels has had approximately no effect on the performance of the model since both test accuracy and number of epochs are rather the same.

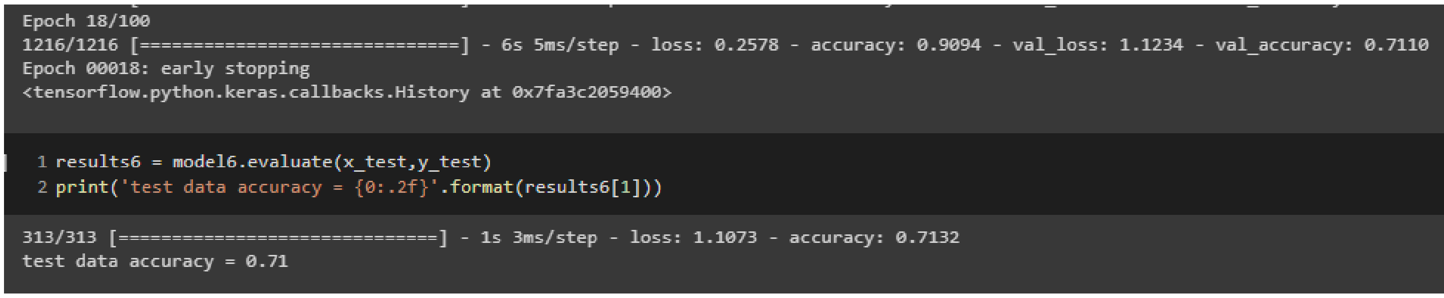


```
Epoch 15/100
1216/1216 [==============================] - 11s 9ms/step - loss: 0.2061 - accuracy: 0.9290 - val_loss: 1.0882 - val_accuracy: 0.7306
Epoch 00015: early stopping
<tensorflow.python.keras.callbacks.History at 0x7fa3c21cb4a8>
```

```
results5 = model5.evaluate(x_test,y_test)
print('test data accuracy = {0:.2f}'.format(results5[1]))
```

```
313/313 [==============================] - 1s 4ms/step - loss: 1.0779 - accuracy: 0.7282
test data accuracy = 0.73
```

‘Halving’

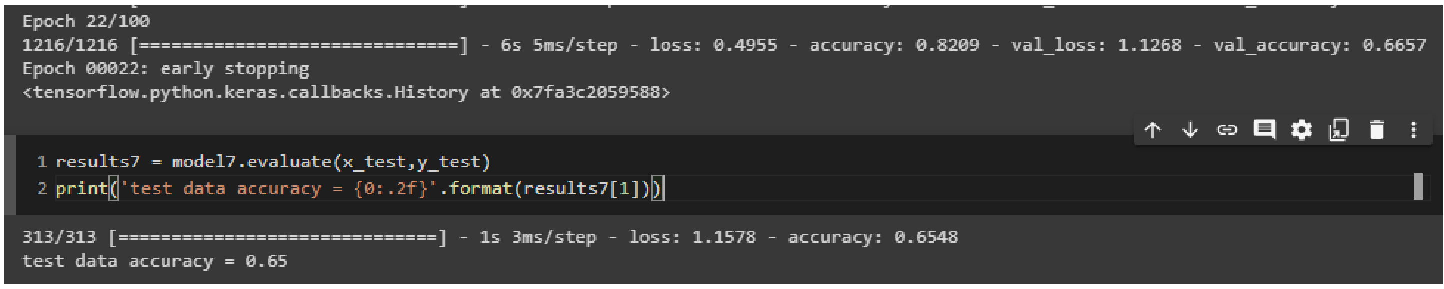


```
Epoch 18/100
1216/1216 [==============================] - 6s 5ms/step - loss: 0.2578 - accuracy: 0.9094 - val_loss: 1.1234 - val_accuracy: 0.7110
Epoch 00018: early stopping
<tensorflow.python.keras.callbacks.History at 0x7fa3c2059400>
```

```
results6 = model6.evaluate(x_test,y_test)
print('test data accuracy = {0:.2f}'.format(results6[1]))
```

```
313/313 [==============================] - 1s 3ms/step - loss: 1.1073 - accuracy: 0.7132
test data accuracy = 0.71
```

‘Quartering’



```
Epoch 22/100
1216/1216 [==============================] - 6s 5ms/step - loss: 0.4955 - accuracy: 0.8209 - val_loss: 1.1268 - val_accuracy: 0.6657
Epoch 00022: early stopping
<tensorflow.python.keras.callbacks.History at 0x7fa3c2059588>
```

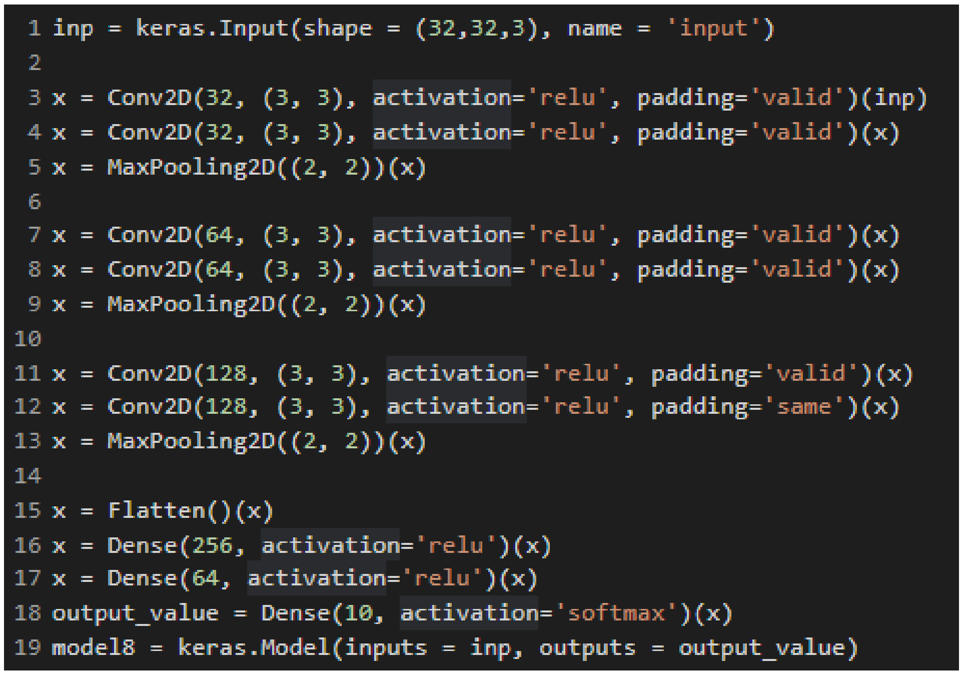
```
results7 = model7.evaluate(x_test,y_test)
print('test data accuracy = {0:.2f}'.format(results7[1]))
```

```
313/313 [==============================] - 1s 3ms/step - loss: 1.1578 - accuracy: 0.6548
test data accuracy = 0.65
```

As we can see in the results obtained above, as the number of kernels is divided by a higher value, the model’s performance drops more significantly.

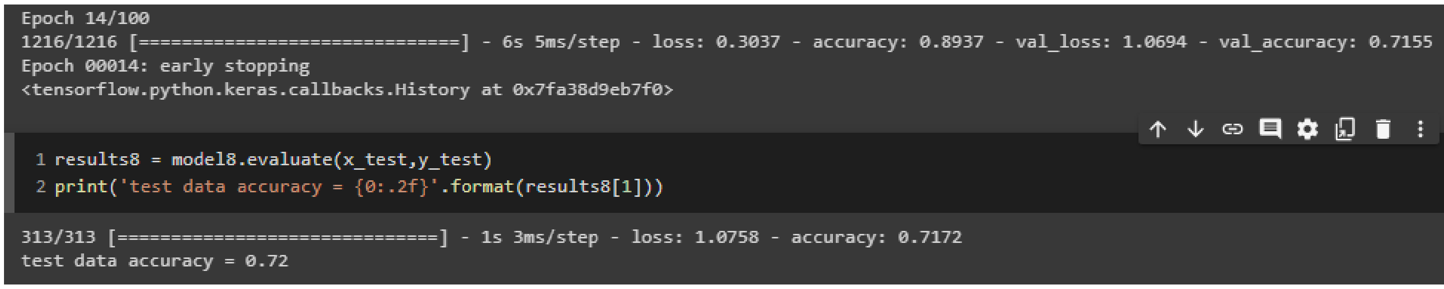
- Padding:

  Contrast to the base-model which had ‘same’ padding in all convolutional layers, this time ‘valid’ padding will be chosen. Therefore, the configuration will be modified to the following:



```python
inp = keras.Input(shape = (32,32,3), name = 'input')

x = Conv2D(32, (3, 3), activation='relu', padding='valid')(inp)
x = Conv2D(32, (3, 3), activation='relu', padding='valid')(x)
x = MaxPooling2D((2, 2))(x)

x = Conv2D(64, (3, 3), activation='relu', padding='valid')(x)
x = Conv2D(64, (3, 3), activation='relu', padding='valid')(x)
x = MaxPooling2D((2, 2))(x)

x = Conv2D(128, (3, 3), activation='relu', padding='valid')(x)
x = Conv2D(128, (3, 3), activation='relu', padding='same')(x)
x = MaxPooling2D((2, 2))(x)

x = Flatten()(x)
x = Dense(256, activation='relu')(x)
x = Dense(64, activation='relu')(x)
output_value = Dense(10, activation='softmax')(x)
model8 = keras.Model(inputs = inp, outputs = output_value)
```

As we can see in the following, 'valid' padding results in a slight decrease in model's performance:



```
Epoch 14/100
1216/1216 [==============================] - 6s 5ms/step - loss: 0.3037 - accuracy: 0.8937 - val_loss: 1.0694 - val_accuracy: 0.7155
Epoch 00014: early stopping
<tensorflow.python.keras.callbacks.History at 0x7fa38d9eb7f0>
```

```python
results8 = model8.evaluate(x_test,y_test)
print('test data accuracy = {0:.2f}'.format(results8[1]))
```

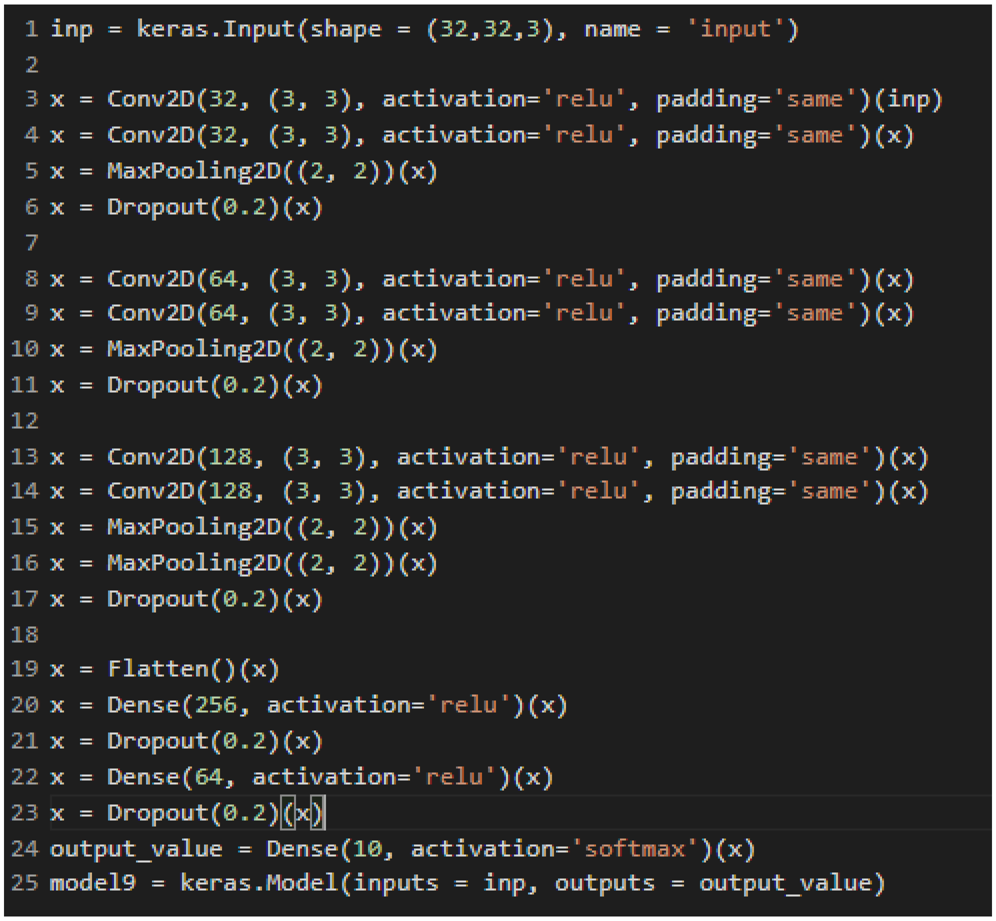
```
313/313 [==============================] - 1s 3ms/step - loss: 1.0758 - accuracy: 0.7172
test data accuracy = 0.72
```

2.

Generally, many regularization approaches have been introduced mostly in order to address the problem of model’s overfitting on the training data. Generally, these approaches slow down convergence rate of the model along with other techniques which have both pros and cons. The pros and cons of L1, L2 and dropout regularization methods are discussed in the following:

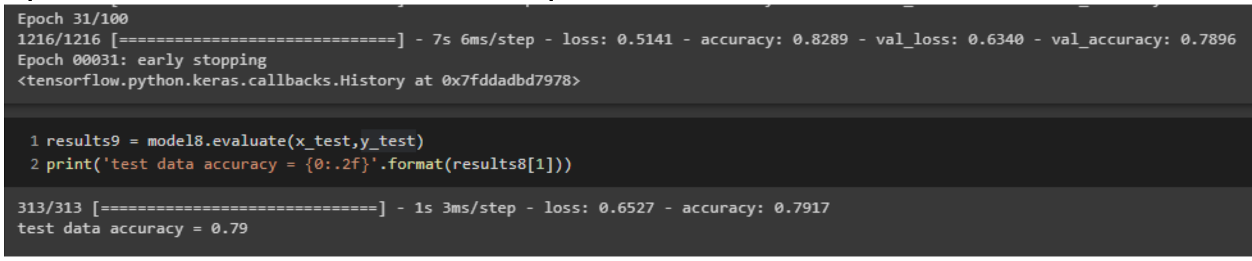
- Dropout Regularization:

  Probabilistically dropping out nodes of layers in the network is a simple and effective regularization method. In fact, by this approach, a single model simulates an Ensemble of neural networks with different configurations which also reduce overfitting but requires the additional computational cost. In contrast, dropout approach is very computationally cheap and also remarkably effective regularization method. Dropout may be applied on any or all hidden layers in the network and It can be used with most types of layers, such as dense fully connected layers, convolutional layers and so on. Without dropout, the network may learn wrong biases or would over-saturate. Lastly, dropout approach like the rest of approaches has the downside of decrease in model’s converge rate which results in more required epochs to converge.
  In our case, the dropout layers are added after each block and Dense layers with same rate of 0.2. The mentioned configuration is as follows:



```
inp = keras.Input(shape = (32,32,3), name = 'input')

x = Conv2D(32, (3, 3), activation='relu', padding='same')(inp)
x = Conv2D(32, (3, 3), activation='relu', padding='same')(x)
x = MaxPooling2D((2, 2))(x)
x = Dropout(0.2)(x)

x = Conv2D(64, (3, 3), activation='relu', padding='same')(x)
x = Conv2D(64, (3, 3), activation='relu', padding='same')(x)
x = MaxPooling2D((2, 2))(x)
x = Dropout(0.2)(x)

x = Conv2D(128, (3, 3), activation='relu', padding='same')(x)
x = Conv2D(128, (3, 3), activation='relu', padding='same')(x)
x = MaxPooling2D((2, 2))(x)
x = MaxPooling2D((2, 2))(x)
x = Dropout(0.2)(x)

x = Flatten()(x)
x = Dense(256, activation='relu')(x)
x = Dropout(0.2)(x)
x = Dense(64, activation='relu')(x)
x = Dropout(0.2)(x)
output_value = Dense(10, activation='softmax')(x)
model9 = keras.Model(inputs = inp, outputs = output_value)
```

As we can see in the following, the test accuracy of the model has increased to 79% after adding Dropout layers but after 31 epochs which has doubled compared to the base-model.

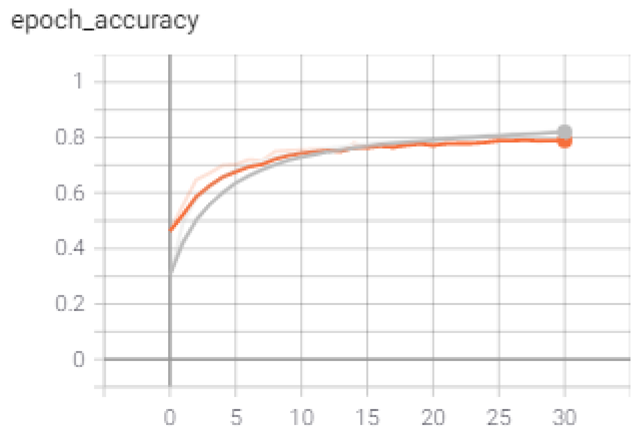


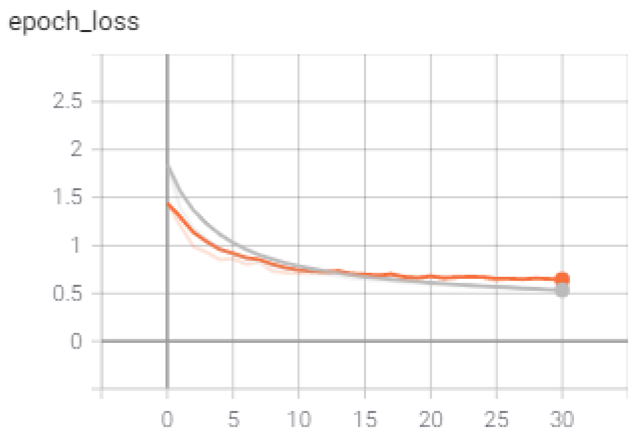
```
Epoch 31/100
1216/1216 [==============================] - 7s 6ms/step - loss: 0.5141 - accuracy: 0.8289 - val_loss: 0.6340 - val_accuracy: 0.7896
Epoch 00031: early stopping
<tensorflow.python.keras.callbacks.History at 0x7fddadbd7978>
```

```
results9 = model8.evaluate(x_test,y_test)
print('test data accuracy = {0:.2f}'.format(results8[1]))
```

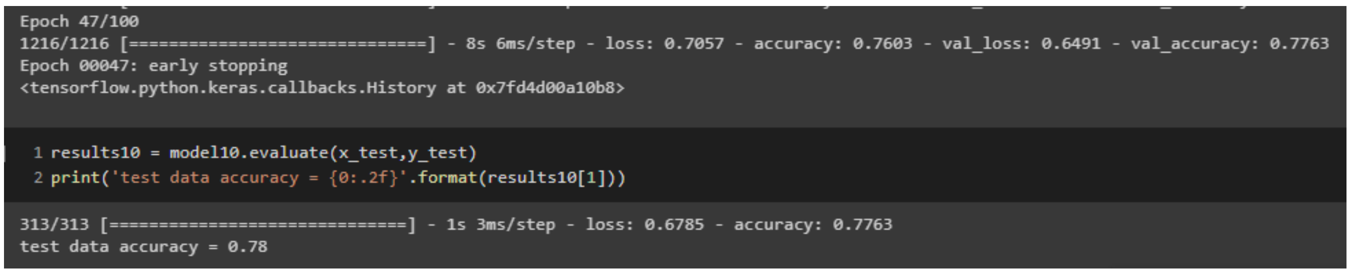
```
313/313 [==============================] - 1s 3ms/step - loss: 0.6527 - accuracy: 0.7917
test data accuracy = 0.79
```

The log of the new model is shown below; as we can see, the problem of overfitting is addressed in the new model by dropout but the training time has doubled.



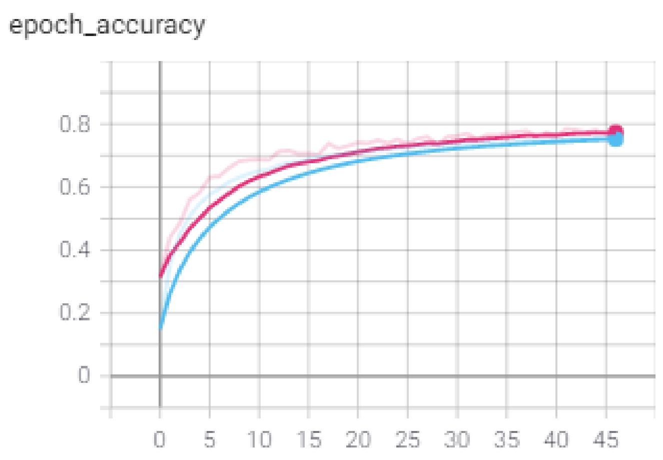


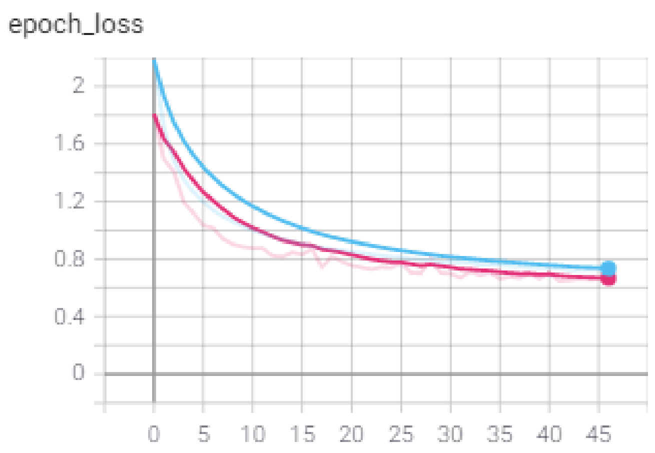
Another the same experiment was conducted, this time with dropout rate of 0.3 which resulted in approximately equal outputs but after more epochs:


```
Epoch 47/100
1216/1216 [==============================] - 8s 6ms/step - loss: 0.7057 - accuracy: 0.7603 - val_loss: 0.6491 - val_accuracy: 0.7763
Epoch 00047: early stopping
<tensorflow.python.keras.callbacks.History at 0x7fd4d00a10b8>
```

```
results10 = model10.evaluate(x_test,y_test)
print('test data accuracy = {0:.2f}'.format(results10[1]))
```

```
313/313 [==============================] - 1s 3ms/step - loss: 0.6785 - accuracy: 0.7763
test data accuracy = 0.78
```

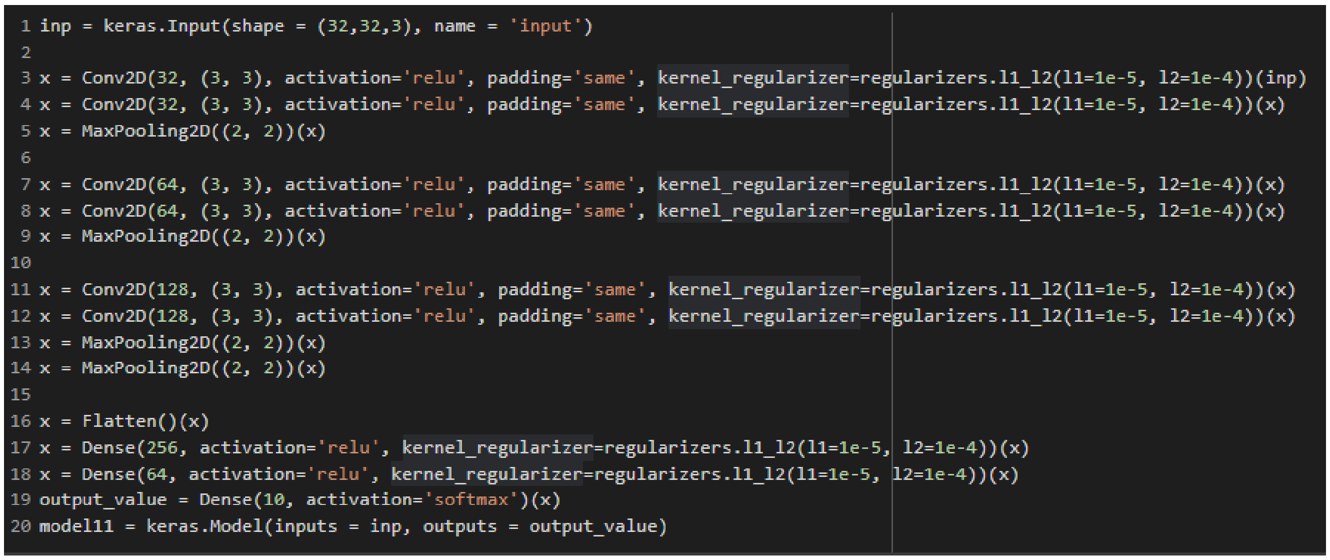




Lastly, the model had the best accuracy with the dropout of 0.2 considering number of epochs compared to values of 0.1 and 0.3.

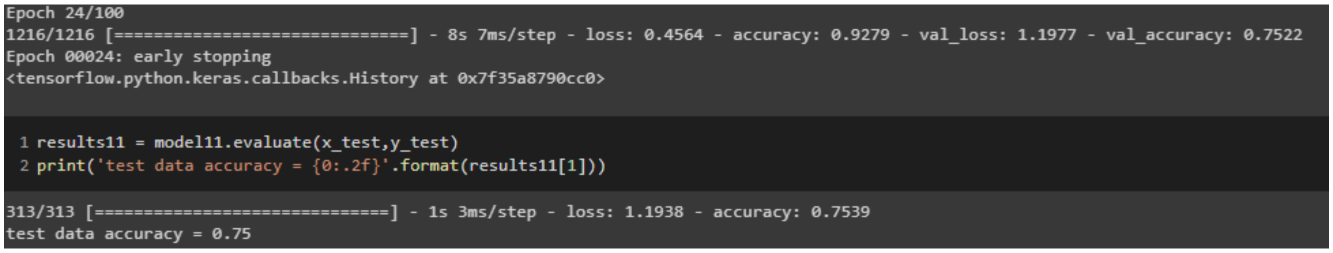
- L1, L2 Regularization:
  Generally, a network with large network weights can be a sign of an unstable network where small changes in the input can lead to large changes in the output. A solution to this problem is to update the cost function to encourage the network to keep the weights small. This method is called weight regularization which has subsets named L1 and L2. Similar to dropout approach, weight regularization may be used with all sort of networks. Also, it is suggested to use both L1 and L2 methods together in order to benefit from advantages of both methods. Similar to dropout method, weight regularization is computationally cheap as well but is less effective compared to dropout method.

  In our case, wight regularization of l1_l2 is added to both convolutional layers and Dense layers of the base-model with the same values. The mentioned configuration is as follows:



```python
inp = keras.Input(shape = (32,32,3), name = 'input')

x = Conv2D(32, (3, 3), activation='relu', padding='same', kernel_regularizer=regularizers.l1_l2(l1=1e-5, l2=1e-4))(inp)
x = Conv2D(32, (3, 3), activation='relu', padding='same', kernel_regularizer=regularizers.l1_l2(l1=1e-5, l2=1e-4))(x)
x = MaxPooling2D((2, 2))(x)

x = Conv2D(64, (3, 3), activation='relu', padding='same', kernel_regularizer=regularizers.l1_l2(l1=1e-5, l2=1e-4))(x)
x = Conv2D(64, (3, 3), activation='relu', padding='same', kernel_regularizer=regularizers.l1_l2(l1=1e-5, l2=1e-4))(x)
x = MaxPooling2D((2, 2))(x)

x = Conv2D(128, (3, 3), activation='relu', padding='same', kernel_regularizer=regularizers.l1_l2(l1=1e-5, l2=1e-4))(x)
x = Conv2D(128, (3, 3), activation='relu', padding='same', kernel_regularizer=regularizers.l1_l2(l1=1e-5, l2=1e-4))(x)
x = MaxPooling2D((2, 2))(x)
x = MaxPooling2D((2, 2))(x)

x = Flatten()(x)
x = Dense(256, activation='relu', kernel_regularizer=regularizers.l1_l2(l1=1e-5, l2=1e-4))(x)
x = Dense(64, activation='relu', kernel_regularizer=regularizers.l1_l2(l1=1e-5, l2=1e-4))(x)
output_value = Dense(10, activation='softmax')(x)
model11 = keras.Model(inputs = inp, outputs = output_value)
```

With the following outputs:



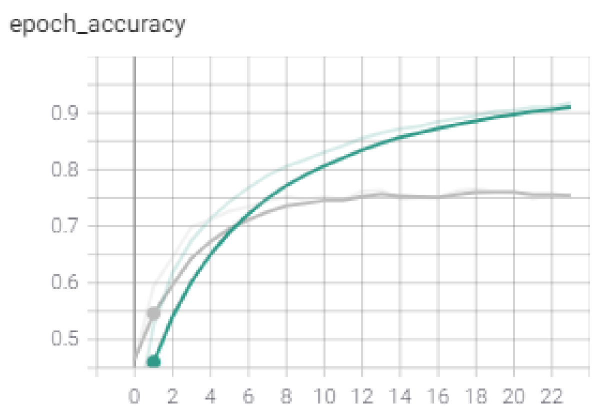
```
Epoch 24/100
1216/1216 [==============================] - 8s 7ms/step - loss: 0.4564 - accuracy: 0.9279 - val_loss: 1.1977 - val_accuracy: 0.7522
Epoch 00024: early stopping
<tensorflow.python.keras.callbacks.History at 0x7f35a8790cc0>
```

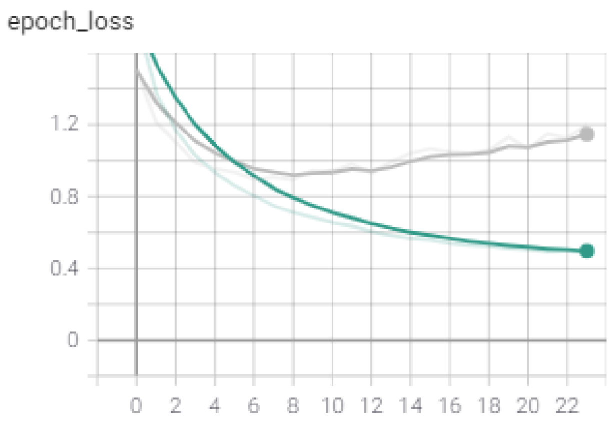
```
results11 = model11.evaluate(x_test,y_test)
print('test data accuracy = {0:.2f}'.format(results11[1]))
```

```
313/313 [==============================] - 1s 3ms/step - loss: 1.1938 - accuracy: 0.7539
test data accuracy = 0.75
```

As we can see, the model has reached 75% test accuracy after 24 epochs which is similar to the base-model.

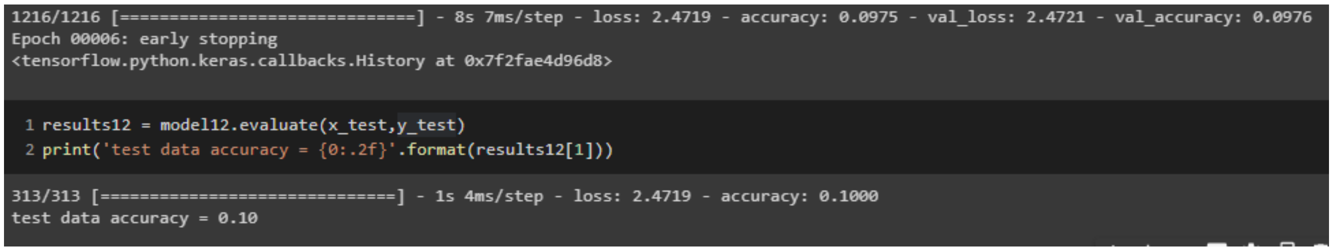
Also, as we can see in the following logs, the configuration couldn't address overfitting.





The first experiment was conducted with the small values of 10^{-5} and 10^{-4} for l1 and l2 respectively. Which had no sensible effect on the base-model.

The next experiment was conducted with higher values of 0.3 and 0.3 for l1 and l2 which resulted in no training at all:



```
1216/1216 [==============================] - 8s 7ms/step - loss: 2.4719 - accuracy: 0.0975 - val_loss: 2.4721 - val_accuracy: 0.0976
Epoch 00006: early stopping
<tensorflow.python.keras.callbacks.History at 0x7f2fae4d96d8>
```

```
results12 = model12.evaluate(x_test,y_test)
print('test data accuracy = {0:.2f}'.format(results12[1]))
```

```
313/313 [==============================] - 1s 4ms/step - loss: 2.4719 - accuracy: 0.1000
test data accuracy = 0.10
```

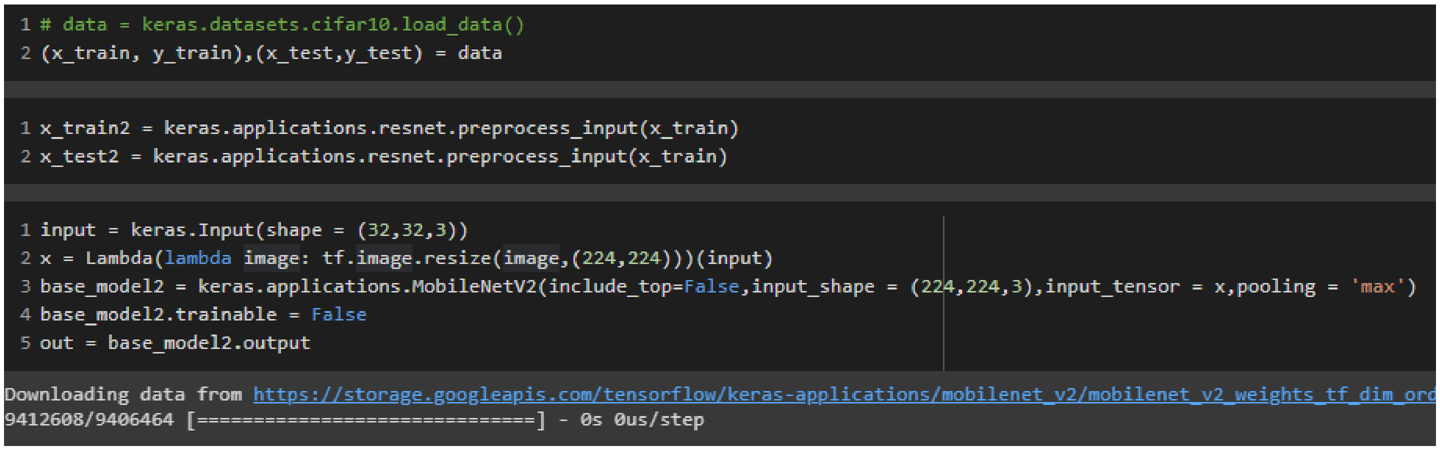
To sum up, as we saw above, the dropout regularization approach significantly improved the model as against the weight regularization approach which had no sensible effect.

“Second Part”

1.

First step:

Importing the model and preprocessing the input data with the imported model’s corresponding preprocessing method.



```
# data = keras.datasets.cifar10.load_data()
(x_train, y_train),(x_test,y_test) = data
```

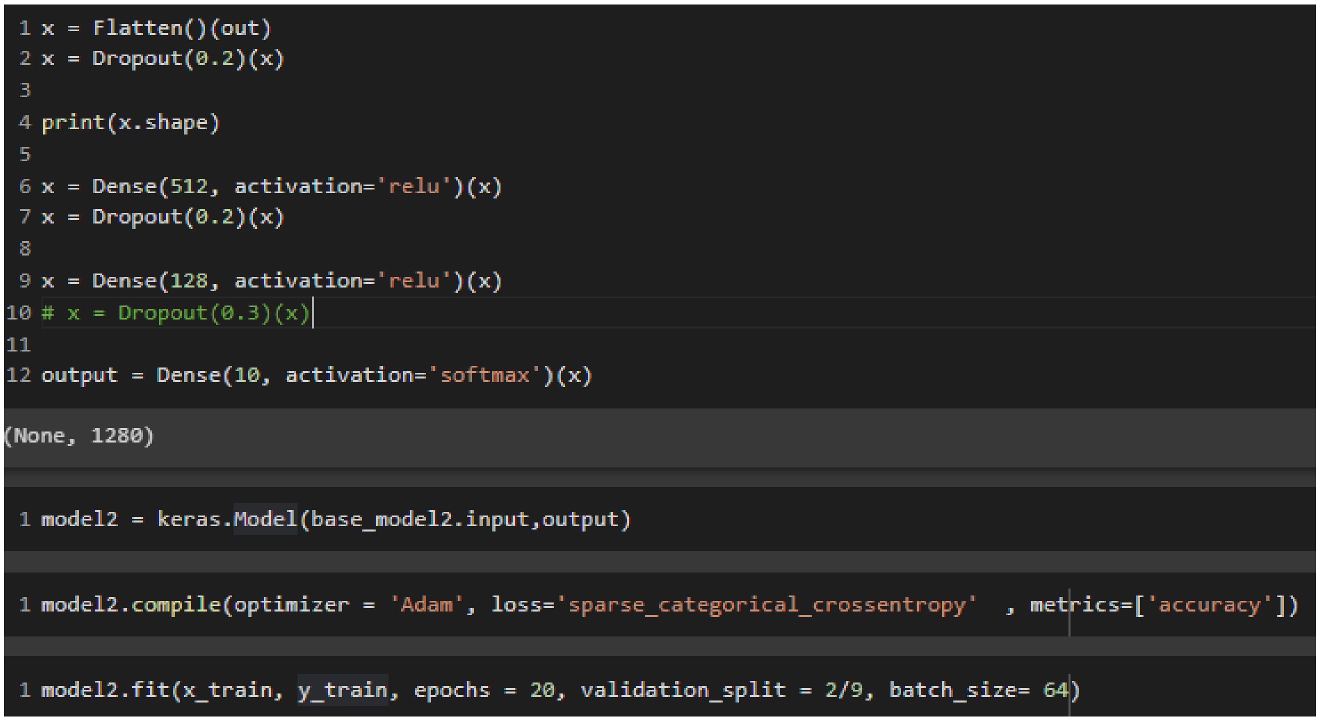
```
x_train2 = keras.applications.resnet.preprocess_input(x_train)
x_test2 = keras.applications.resnet.preprocess_input(x_train)
```

```
input = keras.Input(shape = (32,32,3))
x = Lambda(lambda image: tf.image.resize(image,(224,224)))(input)
base_model2 = keras.applications.MobileNetV2(include_top=False,input_shape = (224,224,3),input_tensor = x,pooling = 'max')
base_model2.trainable = False
out = base_model2.output
```

```
Downloading data from https://storage.googleapis.com/tensorflow/keras-applications/mobilenet_v2/mobilenet_v2_weights_tf_dim_ord
9412608/9406464 [==============================] - 0s 0us/step
```

Second step (Feature extracting):

Now, we first upscale our data in order to make it compatible with the imported model’s input size by adding a lambda layer at the beginning of the model. Then, by taking only the feature extractor part of the model and setting its layers’ trainable parameter to ‘false’, and adding some custom Dense layers similar to our previously defined base-model as the classifier part of the final model, we begin to train the final model for few numbers of epochs (8 epochs) to train the initial weights of the classifier part of it.



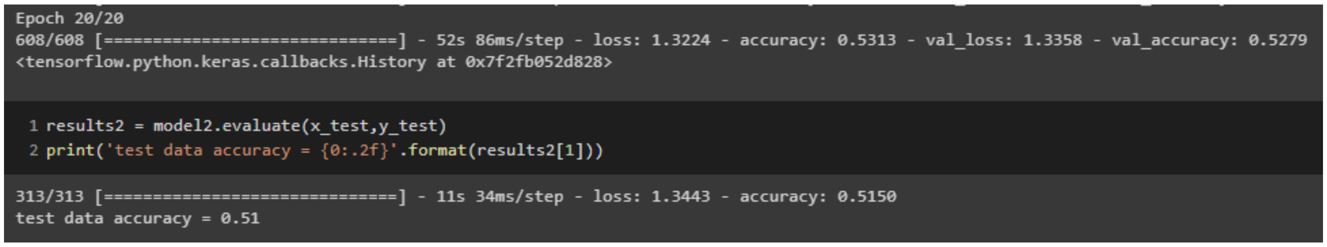
```
x = Flatten()(out)
x = Dropout(0.2)(x)

print(x.shape)

x = Dense(512, activation='relu')(x)
x = Dropout(0.2)(x)

x = Dense(128, activation='relu')(x)
# x = Dropout(0.3)(x)

output = Dense(10, activation='softmax')(x)
```

```
(None, 1280)
```

```
model2 = keras.Model(base_model2.input,output)
```

```
model2.compile(optimizer = 'Adam', loss='sparse_categorical_crossentropy'  , metrics=['accuracy'])
```

```
model2.fit(x_train, y_train, epochs = 20, validation_split = 2/9, batch_size= 64)
```



```
Epoch 20/20
608/608 [==============================] - 52s 86ms/step - loss: 1.3224 - accuracy: 0.5313 - val_loss: 1.3358 - val_accuracy: 0.5279
<tensorflow.python.keras.callbacks.History at 0x7f2fb052d828>
```

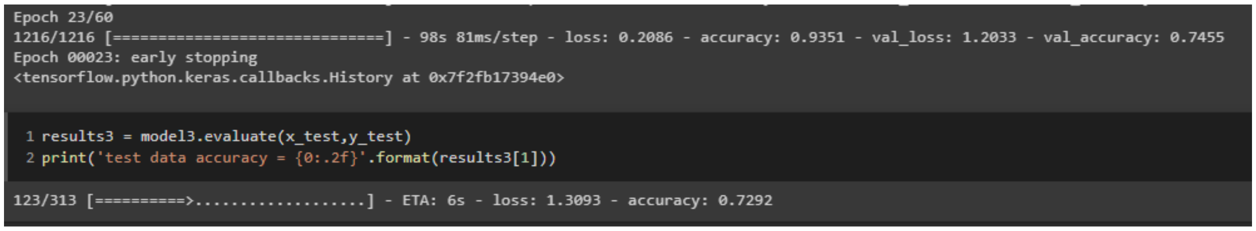
```
results2 = model2.evaluate(x_test,y_test)
print('test data accuracy = {0:.2f}'.format(results2[1]))
```

```
313/313 [==============================] - 11s 34ms/step - loss: 1.3443 - accuracy: 0.5150
test data accuracy = 0.51
```

Third step (Fine-Tuning):

Now that we have trained initial weights of classifier part, we begin the fine-tuning process in order to further improve the weight of both feature extractor and classifier parts of our model and reach higher accuracy. For feature extractor part, a portion of the latest layers (120 out of 156 layers) will become trainable.

The obtained results are as follows:



```
Epoch 23/60
1216/1216 [==============================] - 98s 81ms/step - loss: 0.2086 - accuracy: 0.9351 - val_loss: 1.2033 - val_accuracy: 0.7455
Epoch 00023: early stopping
<tensorflow.python.keras.callbacks.History at 0x7f2fb17394e0>
```

```python
results3 = model3.evaluate(x_test,y_test)
print('test data accuracy = {0:.2f}'.format(results3[1]))
```

```
123/313 [==========>...................] - ETA: 6s - loss: 1.3093 - accuracy: 0.7292
```

The poor accuracy is due to the poor regularization, which could not be addressed by adding dropout layers with rate of 0.3.

Therefore, although the fine-tuned model had superior accuracy on training data but failed in test accuracy compared to the base-model due to the poor regularization.