HW1

Yasaman Emami Deep Learning Oct 5,2022

The CNN models are developed using “keras” package in python.

Common steps for all models part1 :

1. Load data from CIFAR10 dataset from keras sample datasets which is colored images with 10 classes
2. Split data into train and test(50k train, 10k test)
3. showSamples(train) => shows 25 samples of train dataset

CNN Model for Q1 part a and b:

4. cnn(shap) => this function build CNN model required in Q1 part a which starts with input layer and “shap” is the shape of the input which is the input parameter of this function then other layers stack on top of each other to build the structure the following layers repeat 3 times as required by Q1 part a:
    a. Conv2D layer => to create 2D convolutional layer which applies to images Conv2D(filters, kernel_size=3, activation='relu', strides =1, padding = 'same') For filters value first round 32 was selected then following processes used 64 as the number for filters the reason is that 16 or 32 for the first layer is common to set. Since the first several layers' filters commonly replicate edge detectors, blob detectors, and other less abstract filter types. Since there is a limit to how much information can be extracted from the raw input layer, you typically don't want to apply too many filters to the input layer. If you add too many filters, most of them will be unnecessary.
    b. BatchNormalization layer => The internal covariate shift is a significant issue that batch normalization resolves. By applying batchnormalization a greater learning rate can be selected since it makes the data travel between intermediate layers of the neural network. Since it has a regularizing impact, dropouts are frequently eliminated.
    c. MaxPooling layer => Max Pooling pooling is a technique used in computer programming to reduce the computational cost by lowering the number of parameters to learn. A feature map is downsampled (pooled) by using the max pooling procedure, which determines the maximum value for patches of the feature map.
5. Flatten Layed => After that to feed the input to fully connected layers it should be flattened which reduces the input tensors' many dimensions to a single one
6. Dense Layer => After the step5 two Dense layers will be applied first one with 512 units and the second one with 128 units and “reLu activation” function. Based on the results of the convolutional layers, a dense layer is utilized to categorize the images. The neurons in each layer of the neural network calculate the weighted average of their input, and this average is then passed through a non-linear function known as a "activation function."
7. Dropout layer => which is kind of regularization sets some input weights to zero with a frequency of rate = 0.5 this step would prevent overfitting
8. number of trainable parameters also was counted for part d

VGG Model for Q1 part c:

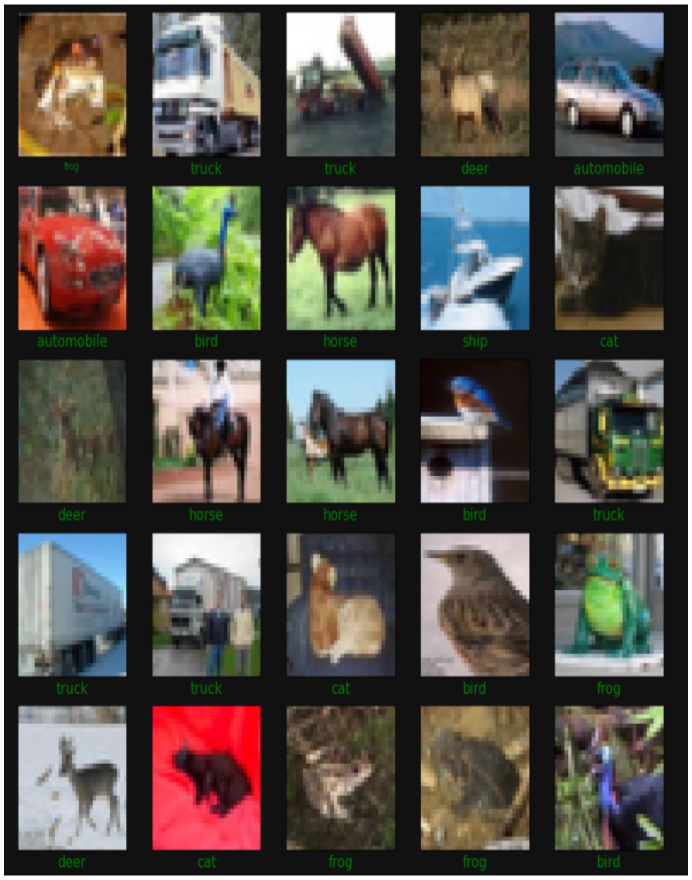
9. In this part first VGG19 model from keras.applications was built as the base model with 3 input parameters: include_top=False, pooling='avg', input_shape = (32,32,3) the reason for selecting these parameter values are include_top is False because it's not needed to add 3 fully connected on top of the network. Pooling is avg because it applies 1 global pooling to the last convolutional block which makes the output to be 2D tensor.
10. myVggModel(base) => filters the layer in a cusom way as requested by question1 part c the last layer which is output layed is added to the model with softmax activation and 10 number of classes
11. In myVggModel func number of trainable parameters also was counted for part d

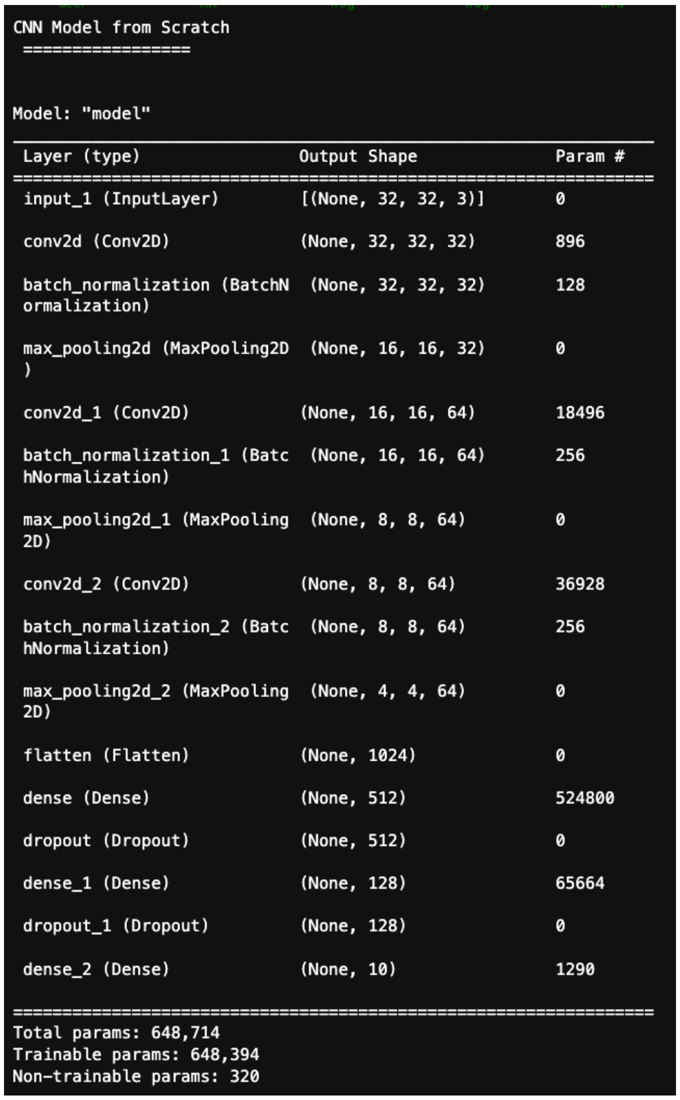
Common steps for all models - part 2:

12. In trainModel(x_train, y_train, x_test, y_test, model) function => a model that was built with cnn/vgg functions and train and test sets are inputs then the model gets compile the optimizer is Adam which was showing better results comparing to other ones like sgd. The loss is sparse categorical entropy as the crossentropy loss between the labels and the predictions is calculated. In situations when there are two or more label classes, use this crossentropy loss function. Labels must be supplied as integers, as expected. The metric is accuracy because FP and FN has no difference for us to use recall and precision. Accuracy = (TP+TN)/(TP+FP+TN+FN)
13. model.fit() => to train the model that was built and complied, batch_size is the default value of 32 which is the number of samples per gradient update, number of epochs is set to 10 as it takes very long-running on cpu and few starting epochs are not showing very good val_accuracy so better to have more epochs and steps_per_epoch is the batches of samples in each epoch as the name offers.
14. plots(hoistory) => plots the performance of the model in different epochs and evaluate the test set accuracy

Q1 – a :

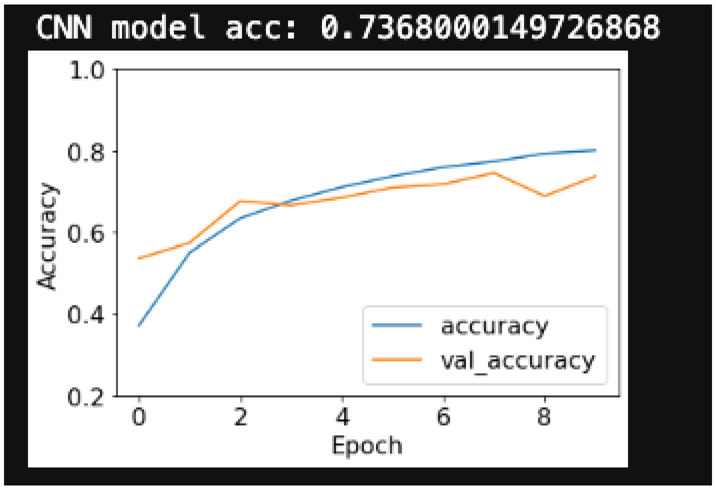
15. Calling the above-mentioned functions related to this part and calculating the execution time

Samples from dataset

Model Summary



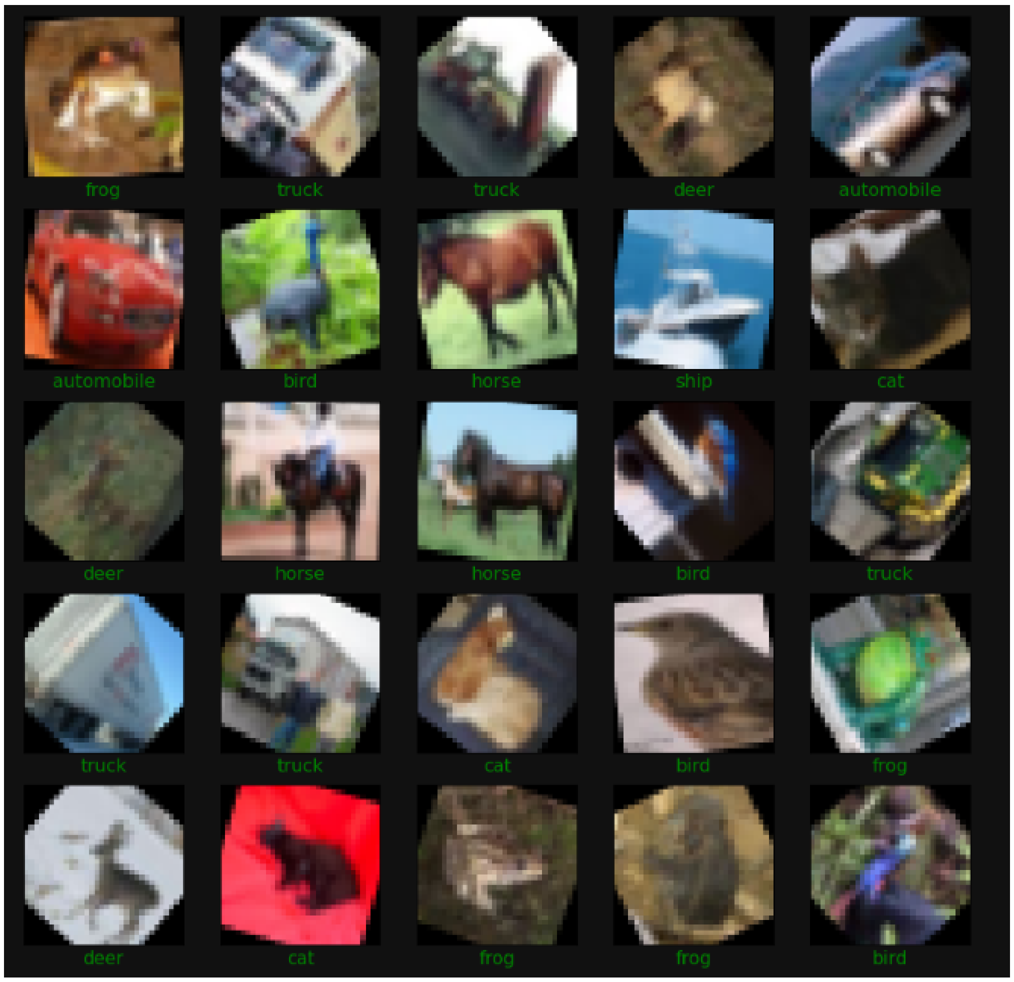
```
CNN Model from Scratch
 ================

Model: "model"
_________________________________________________________________
 Layer (type)                Output Shape              Param #
=================================================================
 input_1 (InputLayer)        [(None, 32, 32, 3)]       0

 conv2d (Conv2D)             (None, 32, 32, 32)        896

 batch_normalization (BatchN  (None, 32, 32, 32)       128
 ormalization)

 max_pooling2d (MaxPooling2D  (None, 16, 16, 32)       0
 )

 conv2d_1 (Conv2D)           (None, 16, 16, 64)        18496

 batch_normalization_1 (Batc  (None, 16, 16, 64)       256
 hNormalization)

 max_pooling2d_1 (MaxPooling  (None, 8, 8, 64)         0
 2D)

 conv2d_2 (Conv2D)           (None, 8, 8, 64)          36928

 batch_normalization_2 (Batc  (None, 8, 8, 64)         256
 hNormalization)

 max_pooling2d_2 (MaxPooling  (None, 4, 4, 64)         0
 2D)

 flatten (Flatten)           (None, 1024)              0

 dense (Dense)               (None, 512)               524800

 dropout (Dropout)           (None, 512)               0

 dense_1 (Dense)             (None, 128)               65664

 dropout_1 (Dropout)         (None, 128)               0

 dense_2 (Dense)             (None, 10)                1290

=================================================================
Total params: 648,714
Trainable params: 648,394
Non-trainable params: 320
```

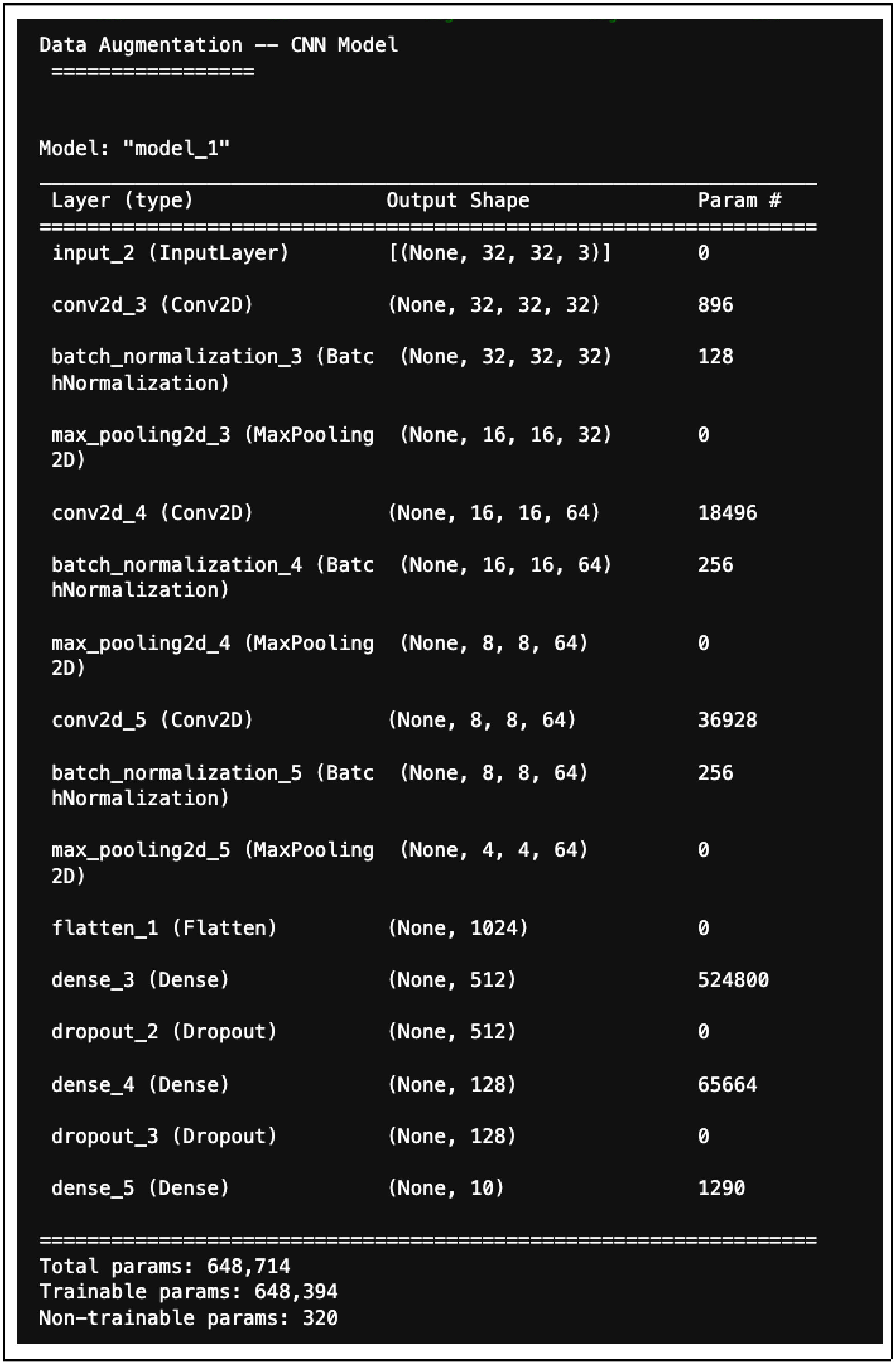
Performance plot



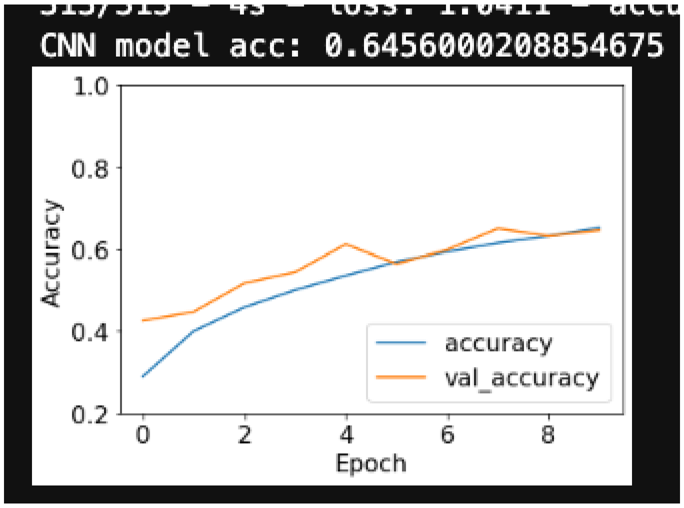
Q1 – b:

16. Data augmentation: rotate images between -70 & 70 degrees (just training set)
17. Then showing 25 samples from train dataset and building the CNN model, train it and evaluate and plot besides calculating the execution time

Samples from dataset
 frog truck truck deer automobile automobile bird horse ship cat deer horse horse bird truck truck truck cat bird frog deer cat frog frog bird
Model Summary



```
Data Augmentation -- CNN Model
 =================


Model: "model_1"
_________________________________________________________________
 Layer (type)                Output Shape              Param #
=================================================================
 input_2 (InputLayer)        [(None, 32, 32, 3)]       0

 conv2d_3 (Conv2D)           (None, 32, 32, 32)        896

 batch_normalization_3 (Batc  (None, 32, 32, 32)       128
 hNormalization)

 max_pooling2d_3 (MaxPooling  (None, 16, 16, 32)       0
 2D)

 conv2d_4 (Conv2D)           (None, 16, 16, 64)        18496

 batch_normalization_4 (Batc  (None, 16, 16, 64)       256
 hNormalization)

 max_pooling2d_4 (MaxPooling  (None, 8, 8, 64)         0
 2D)

 conv2d_5 (Conv2D)           (None, 8, 8, 64)          36928

 batch_normalization_5 (Batc  (None, 8, 8, 64)         256
 hNormalization)

 max_pooling2d_5 (MaxPooling  (None, 4, 4, 64)         0
 2D)

 flatten_1 (Flatten)         (None, 1024)              0

 dense_3 (Dense)             (None, 512)               524800

 dropout_2 (Dropout)         (None, 512)               0

 dense_4 (Dense)             (None, 128)               65664

 dropout_3 (Dropout)         (None, 128)               0

 dense_5 (Dense)             (None, 10)                1290

=================================================================
Total params: 648,714
Trainable params: 648,394
Non-trainable params: 320
```

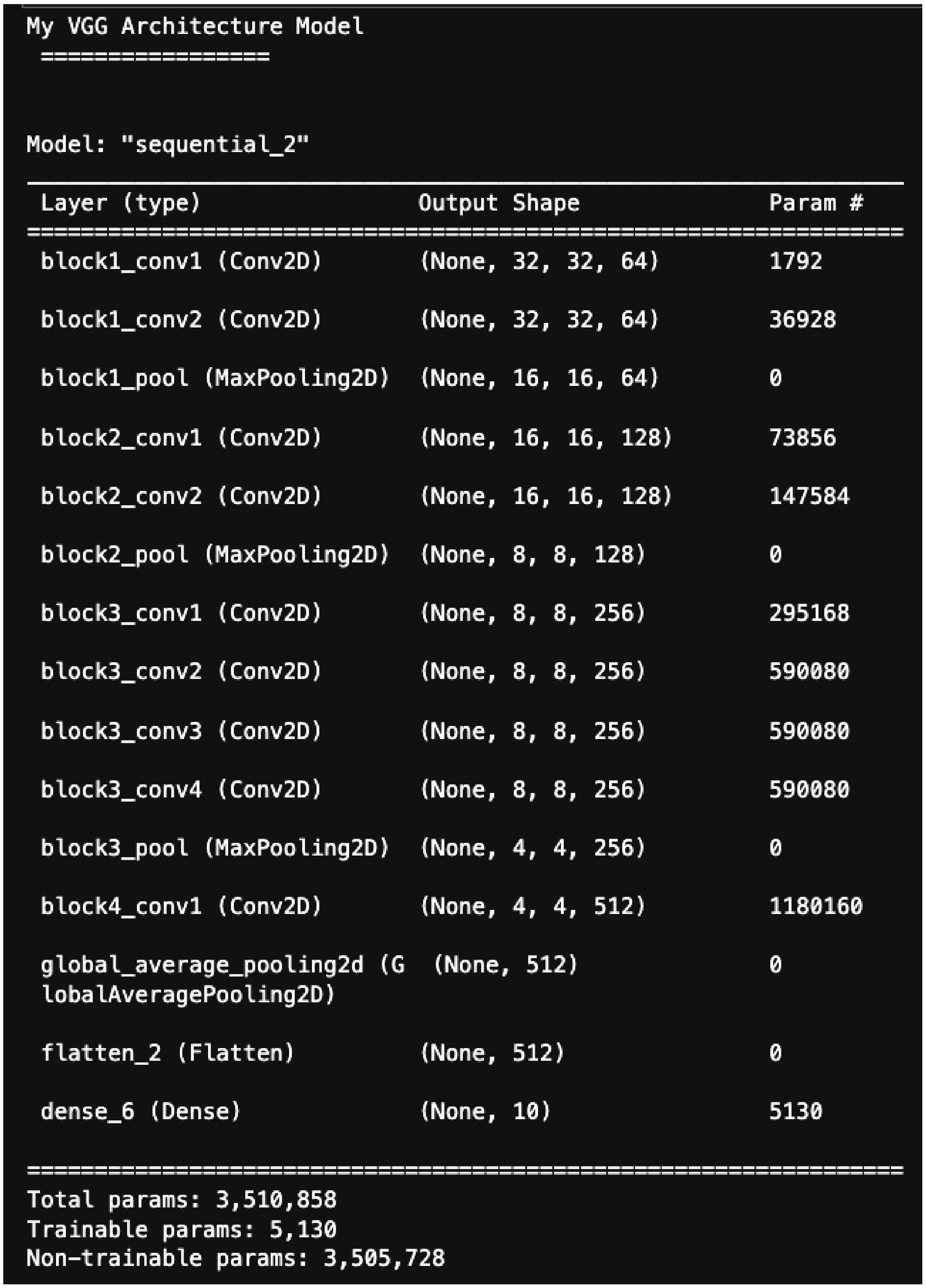
Performance plot


```
313/313 - 4s - loss: 1.0411 - accu
CNN model acc: 0.6456000208854675
```

Q1 – c:

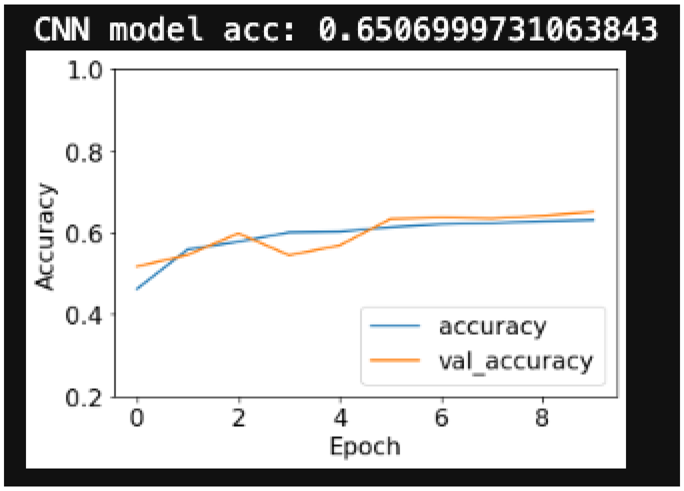
18. building the custom VGG model, train it and evaluate and plot besides calculating the execution time

Model Summary



```
My VGG Architecture Model
 =================


Model: "sequential_2"
_________________________________________________________________
 Layer (type)                Output Shape              Param #
=================================================================
 block1_conv1 (Conv2D)       (None, 32, 32, 64)        1792

 block1_conv2 (Conv2D)       (None, 32, 32, 64)        36928

 block1_pool (MaxPooling2D)  (None, 16, 16, 64)        0

 block2_conv1 (Conv2D)       (None, 16, 16, 128)       73856

 block2_conv2 (Conv2D)       (None, 16, 16, 128)       147584

 block2_pool (MaxPooling2D)  (None, 8, 8, 128)         0

 block3_conv1 (Conv2D)       (None, 8, 8, 256)         295168

 block3_conv2 (Conv2D)       (None, 8, 8, 256)         590080

 block3_conv3 (Conv2D)       (None, 8, 8, 256)         590080

 block3_conv4 (Conv2D)       (None, 8, 8, 256)         590080

 block3_pool (MaxPooling2D)  (None, 4, 4, 256)         0

 block4_conv1 (Conv2D)       (None, 4, 4, 512)         1180160

 global_average_pooling2d (G  (None, 512)              0
 lobalAveragePooling2D)

 flatten_2 (Flatten)         (None, 512)               0

 dense_6 (Dense)             (None, 10)                5130

=================================================================
Total params: 3,510,858
Trainable params: 5,130
Non-trainable params: 3,505,728
```

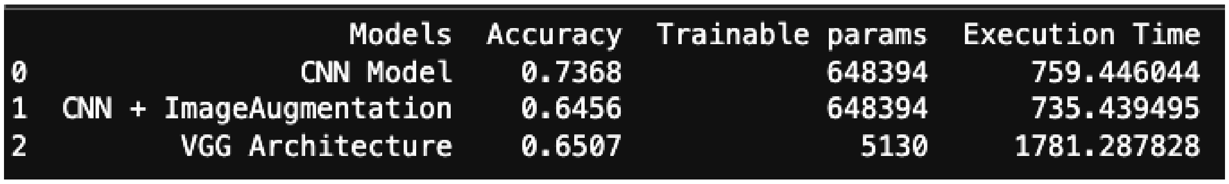
Performance plot



Q1 – d:

19. Showing the table of models, related accuracies, number of trainable parameters, and execution time of each model with the same number of epochs

Model performance comparison



```
                      Models  Accuracy  Trainable params  Execution Time
0                  CNN Model    0.7368            648394      759.446044
1  CNN + ImageAugmentation    0.6456            648394      735.439495
2           VGG Architecture    0.6507              5130     1781.287828
```

As the data shows in the above table CNN model with original data is showing better accuracy among all of the other architectures and augmented data, the trainable parameter is very more than VGG architecture. The number of computations in a computer model increases with the number of trainable parameters, indicating an increase in the hardware requirements. This makes the optimization process more challenging by introducing issues like overfitting or even the choice of the optimization technique itself. Although the number of trainable params is much more than VGG it takes less time to execute.

Q2:

For Q2 layers of network stacks on top of each other then the results of residual blocks would get added only if it improves the accuracy.

1. Using initializer of HeNormal for deterministic behavior of initializer. With stddev = sqrt(2 / fan-in), where fan-in is the number of input units in the weight tensor, it pulls samples from a truncated normal distribution centered on 0.
2. Using piecewise_constant_decay for changing the learning rate of optimizers the values are the rates.
3. Optimizer is selected as SGD this time to experiment with lr defined above and optimizer_momentum as 0.9 as its offered by some researchers to accelerate the gradient in the right direction for faster converge.
4. Then y_train and y_test is onehot encoded to be compatible tpo use with categorical crossentropy loss function
5. Residual_block () => takes a copy of input at the beginning then creates layers in the block as required by question. Then before feeding to relu activation it benefits from the add function in keras to add the result of RB and initial input which gets a list of tensors of same shape and returns one tensor before ADD matching filter should be applied on initial input which is called x_skip to match the shape of output of this block
6. ResidualBlock() => invoke the method in 5 with filter size 32 which is asked in the question and doubles the size of filer every 3 times
7. my_model() => create base model and invoke ResidualBlock method for RB part and softmax with 10 classes is applied as the last(output)layer. Softmax because we have multi classes
8. Train_model => is the same as trainModel in Q1 in this part built model is also saved with two approaches keras model.save() and with the help of joblib library
9. evaluate _model => is same as plots method in Q1
10. At the end all required methods are invoked to initialize train and evaluate ResNet model

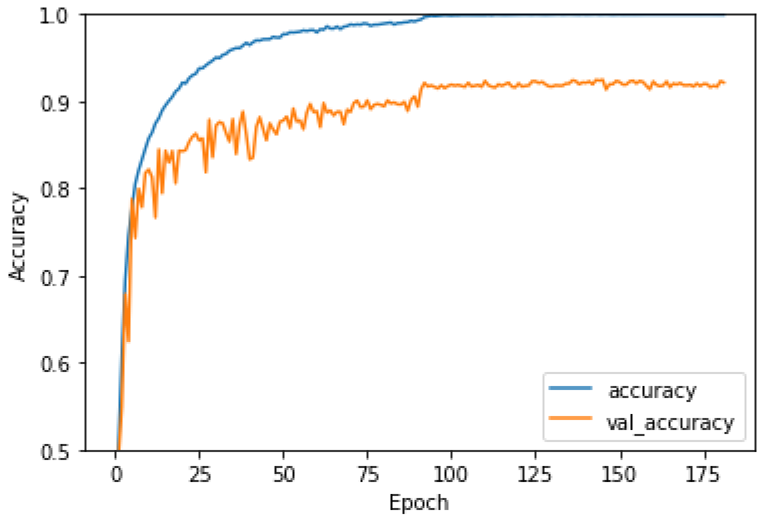
This application was run on google colab to use GPU resources and the final model is saved and shared in
"https://drive.google.com/file/d/1QieO4b2Udrl2bKQIqwNl8a-KPCqbBcZy/view?usp=sharing"

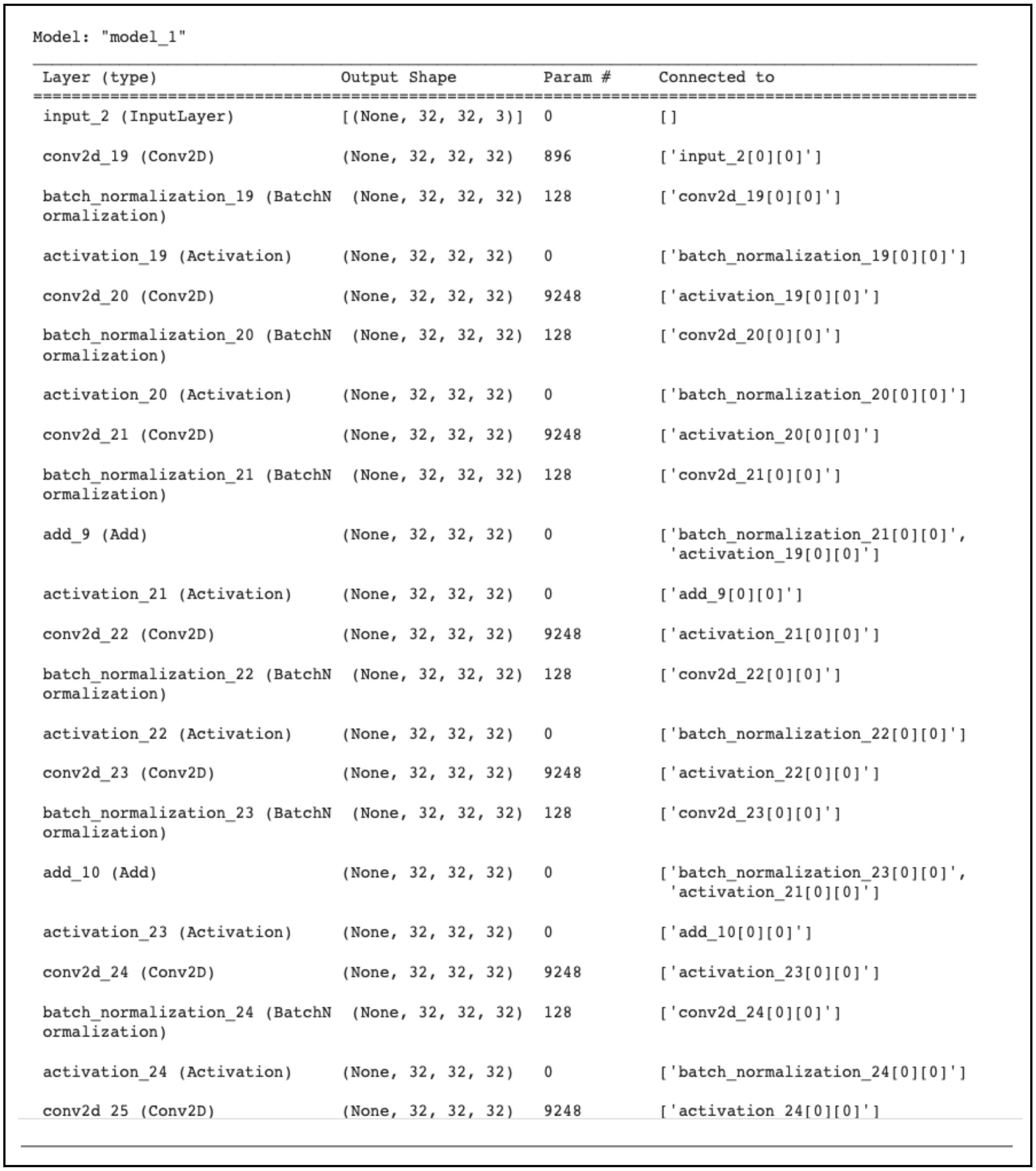
This model was trained for 182 epochs:
Following is the result of last epoch:

```
Epoch 182/182
351/351 [==============================] - 29s 82ms/step - loss: 0.0019 - accuracy: 0.9996 - val_loss: 0.4153 - val_accuracy: 0.9215
```

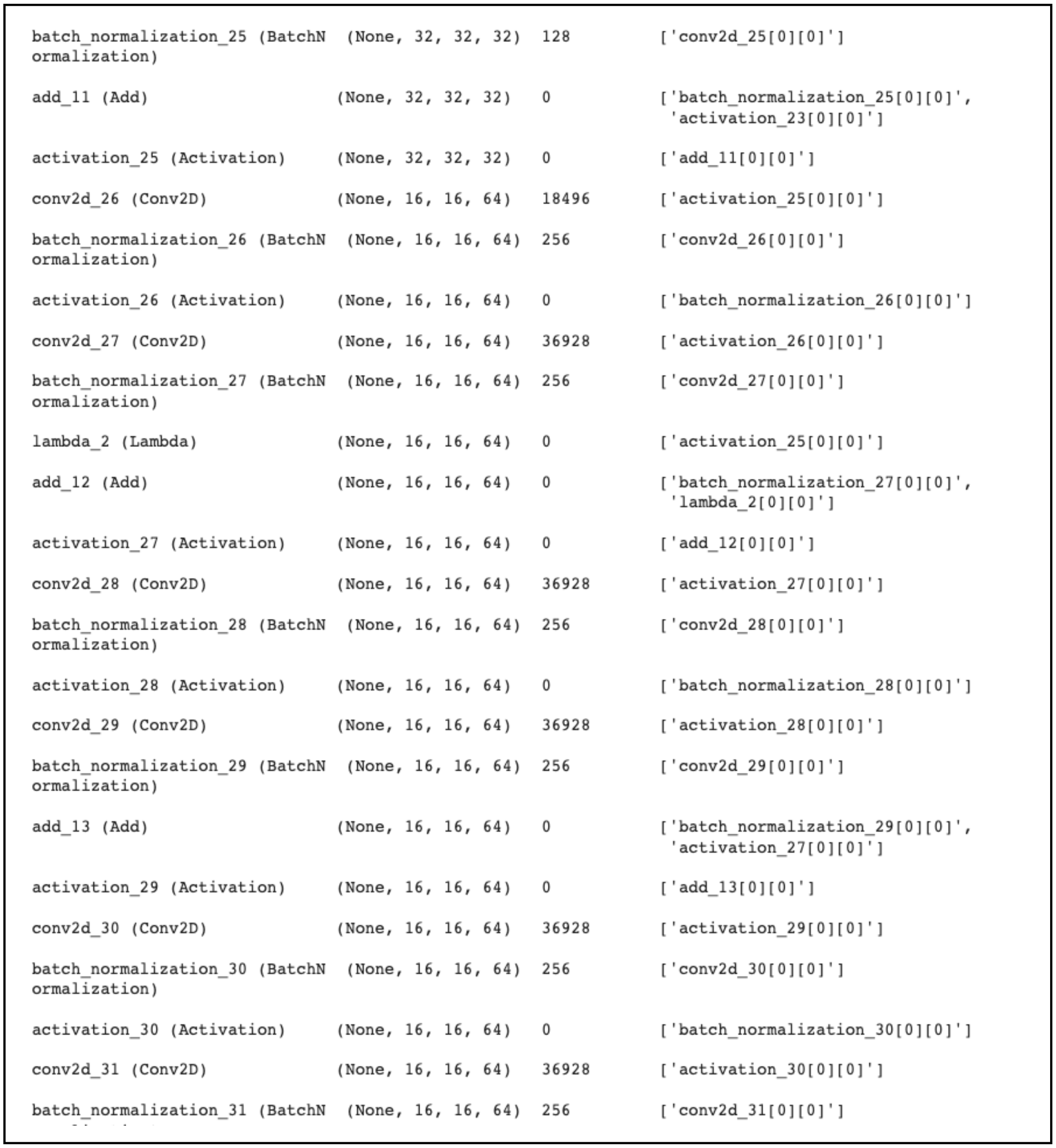
Here is the accuracy of test set:

```
Test loss: 0.49421802163124084 / Test accuracy: 0.9175000190734863
```

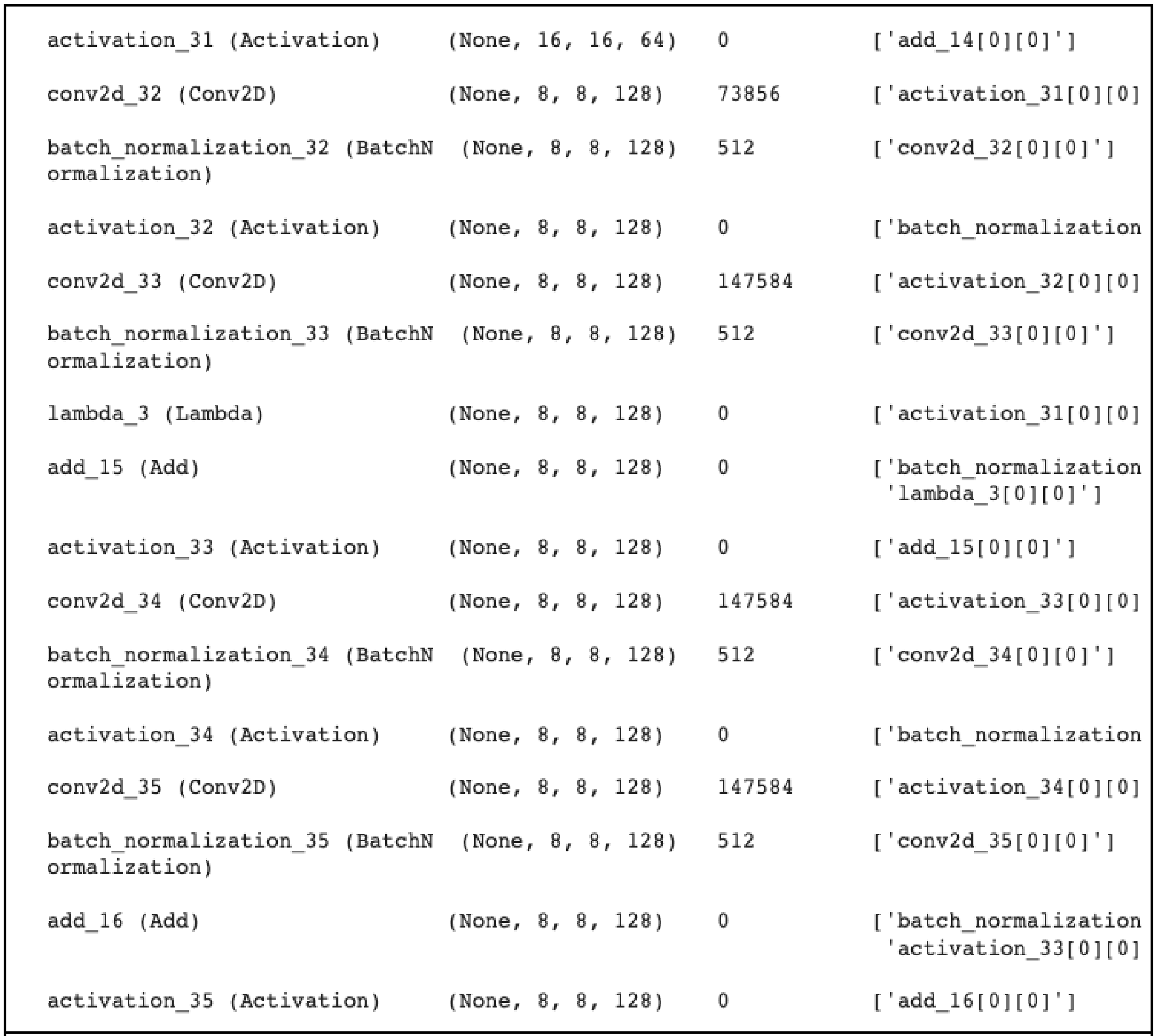
Performance plot

Some screenshots of the model summary



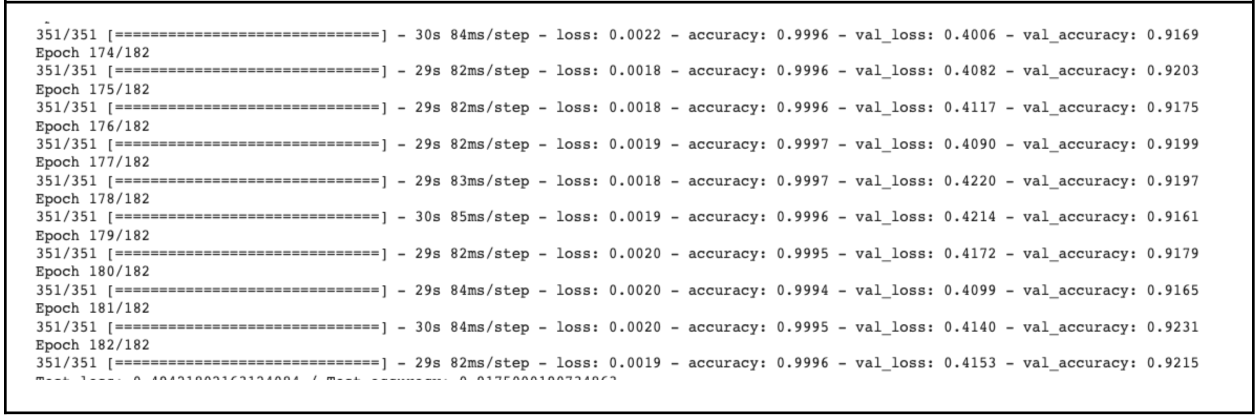
```
Model: "model_1"
__________________________________________________________________________________________________
 Layer (type)                   Output Shape         Param #     Connected to
==================================================================================================
 input_2 (InputLayer)           [(None, 32, 32, 3)]  0           []

 conv2d_19 (Conv2D)             (None, 32, 32, 32)   896         ['input_2[0][0]']

 batch_normalization_19 (BatchN  (None, 32, 32, 32)  128         ['conv2d_19[0][0]']
 ormalization)

 activation_19 (Activation)     (None, 32, 32, 32)   0           ['batch_normalization_19[0][0]']

 conv2d_20 (Conv2D)             (None, 32, 32, 32)   9248        ['activation_19[0][0]']

 batch_normalization_20 (BatchN  (None, 32, 32, 32)  128         ['conv2d_20[0][0]']
 ormalization)

 activation_20 (Activation)     (None, 32, 32, 32)   0           ['batch_normalization_20[0][0]']

 conv2d_21 (Conv2D)             (None, 32, 32, 32)   9248        ['activation_20[0][0]']

 batch_normalization_21 (BatchN  (None, 32, 32, 32)  128         ['conv2d_21[0][0]']
 ormalization)

 add_9 (Add)                    (None, 32, 32, 32)   0           ['batch_normalization_21[0][0]',
                                                                  'activation_19[0][0]']

 activation_21 (Activation)     (None, 32, 32, 32)   0           ['add_9[0][0]']

 conv2d_22 (Conv2D)             (None, 32, 32, 32)   9248        ['activation_21[0][0]']

 batch_normalization_22 (BatchN  (None, 32, 32, 32)  128         ['conv2d_22[0][0]']
 ormalization)

 activation_22 (Activation)     (None, 32, 32, 32)   0           ['batch_normalization_22[0][0]']

 conv2d_23 (Conv2D)             (None, 32, 32, 32)   9248        ['activation_22[0][0]']

 batch_normalization_23 (BatchN  (None, 32, 32, 32)  128         ['conv2d_23[0][0]']
 ormalization)

 add_10 (Add)                   (None, 32, 32, 32)   0           ['batch_normalization_23[0][0]',
                                                                  'activation_21[0][0]']

 activation_23 (Activation)     (None, 32, 32, 32)   0           ['add_10[0][0]']

 conv2d_24 (Conv2D)             (None, 32, 32, 32)   9248        ['activation_23[0][0]']

 batch_normalization_24 (BatchN  (None, 32, 32, 32)  128         ['conv2d_24[0][0]']
 ormalization)

 activation_24 (Activation)     (None, 32, 32, 32)   0           ['batch_normalization_24[0][0]']

 conv2d_25 (Conv2D)             (None, 32, 32, 32)   9248        ['activation_24[0][0]']
```



```
batch_normalization_25 (BatchN  (None, 32, 32, 32)  128        ['conv2d_25[0][0]']
ormalization)

add_11 (Add)                   (None, 32, 32, 32)  0          ['batch_normalization_25[0][0]',
                                                               'activation_23[0][0]']

activation_25 (Activation)     (None, 32, 32, 32)  0          ['add_11[0][0]']

conv2d_26 (Conv2D)             (None, 16, 16, 64)  18496      ['activation_25[0][0]']

batch_normalization_26 (BatchN  (None, 16, 16, 64)  256        ['conv2d_26[0][0]']
ormalization)

activation_26 (Activation)     (None, 16, 16, 64)  0          ['batch_normalization_26[0][0]']

conv2d_27 (Conv2D)             (None, 16, 16, 64)  36928      ['activation_26[0][0]']

batch_normalization_27 (BatchN  (None, 16, 16, 64)  256        ['conv2d_27[0][0]']
ormalization)

lambda_2 (Lambda)              (None, 16, 16, 64)  0          ['activation_25[0][0]']

add_12 (Add)                   (None, 16, 16, 64)  0          ['batch_normalization_27[0][0]',
                                                               'lambda_2[0][0]']

activation_27 (Activation)     (None, 16, 16, 64)  0          ['add_12[0][0]']

conv2d_28 (Conv2D)             (None, 16, 16, 64)  36928      ['activation_27[0][0]']

batch_normalization_28 (BatchN  (None, 16, 16, 64)  256        ['conv2d_28[0][0]']
ormalization)

activation_28 (Activation)     (None, 16, 16, 64)  0          ['batch_normalization_28[0][0]']

conv2d_29 (Conv2D)             (None, 16, 16, 64)  36928      ['activation_28[0][0]']

batch_normalization_29 (BatchN  (None, 16, 16, 64)  256        ['conv2d_29[0][0]']
ormalization)

add_13 (Add)                   (None, 16, 16, 64)  0          ['batch_normalization_29[0][0]',
                                                               'activation_27[0][0]']

activation_29 (Activation)     (None, 16, 16, 64)  0          ['add_13[0][0]']

conv2d_30 (Conv2D)             (None, 16, 16, 64)  36928      ['activation_29[0][0]']

batch_normalization_30 (BatchN  (None, 16, 16, 64)  256        ['conv2d_30[0][0]']
ormalization)

activation_30 (Activation)     (None, 16, 16, 64)  0          ['batch_normalization_30[0][0]']

conv2d_31 (Conv2D)             (None, 16, 16, 64)  36928      ['activation_30[0][0]']

batch_normalization_31 (BatchN  (None, 16, 16, 64)  256        ['conv2d_31[0][0]']
```



```
activation_31 (Activation)     (None, 16, 16, 64)   0           ['add_14[0][0]']

conv2d_32 (Conv2D)             (None, 8, 8, 128)    73856       ['activation_31[0][0]

batch_normalization_32 (BatchN  (None, 8, 8, 128)   512         ['conv2d_32[0][0]']
ormalization)

activation_32 (Activation)     (None, 8, 8, 128)    0           ['batch_normalization

conv2d_33 (Conv2D)             (None, 8, 8, 128)    147584      ['activation_32[0][0]

batch_normalization_33 (BatchN  (None, 8, 8, 128)   512         ['conv2d_33[0][0]']
ormalization)

lambda_3 (Lambda)              (None, 8, 8, 128)    0           ['activation_31[0][0]

add_15 (Add)                   (None, 8, 8, 128)    0           ['batch_normalization
                                                                 'lambda_3[0][0]']

activation_33 (Activation)     (None, 8, 8, 128)    0           ['add_15[0][0]']

conv2d_34 (Conv2D)             (None, 8, 8, 128)    147584      ['activation_33[0][0]

batch_normalization_34 (BatchN  (None, 8, 8, 128)   512         ['conv2d_34[0][0]']
ormalization)

activation_34 (Activation)     (None, 8, 8, 128)    0           ['batch_normalization

conv2d_35 (Conv2D)             (None, 8, 8, 128)    147584      ['activation_34[0][0]

batch_normalization_35 (BatchN  (None, 8, 8, 128)   512         ['conv2d_35[0][0]']
ormalization)

add_16 (Add)                   (None, 8, 8, 128)    0           ['batch_normalization
                                                                 'activation_33[0][0]

activation_35 (Activation)     (None, 8, 8, 128)    0           ['add_16[0][0]']
```



```
351/351 [==============================] - 30s 84ms/step - loss: 0.0022 - accuracy: 0.9996 - val_loss: 0.4006 - val_accuracy: 0.9169
Epoch 174/182
351/351 [==============================] - 29s 82ms/step - loss: 0.0018 - accuracy: 0.9996 - val_loss: 0.4082 - val_accuracy: 0.9203
Epoch 175/182
351/351 [==============================] - 29s 82ms/step - loss: 0.0018 - accuracy: 0.9996 - val_loss: 0.4117 - val_accuracy: 0.9175
Epoch 176/182
351/351 [==============================] - 29s 82ms/step - loss: 0.0019 - accuracy: 0.9997 - val_loss: 0.4090 - val_accuracy: 0.9199
Epoch 177/182
351/351 [==============================] - 29s 83ms/step - loss: 0.0018 - accuracy: 0.9997 - val_loss: 0.4220 - val_accuracy: 0.9197
Epoch 178/182
351/351 [==============================] - 30s 85ms/step - loss: 0.0019 - accuracy: 0.9996 - val_loss: 0.4214 - val_accuracy: 0.9161
Epoch 179/182
351/351 [==============================] - 29s 82ms/step - loss: 0.0020 - accuracy: 0.9995 - val_loss: 0.4172 - val_accuracy: 0.9179
Epoch 180/182
351/351 [==============================] - 29s 84ms/step - loss: 0.0020 - accuracy: 0.9994 - val_loss: 0.4099 - val_accuracy: 0.9165
Epoch 181/182
351/351 [==============================] - 30s 84ms/step - loss: 0.0020 - accuracy: 0.9995 - val_loss: 0.4140 - val_accuracy: 0.9231
Epoch 182/182
351/351 [==============================] - 29s 82ms/step - loss: 0.0019 - accuracy: 0.9996 - val_loss: 0.4153 - val_accuracy: 0.9215
[illegible]
```