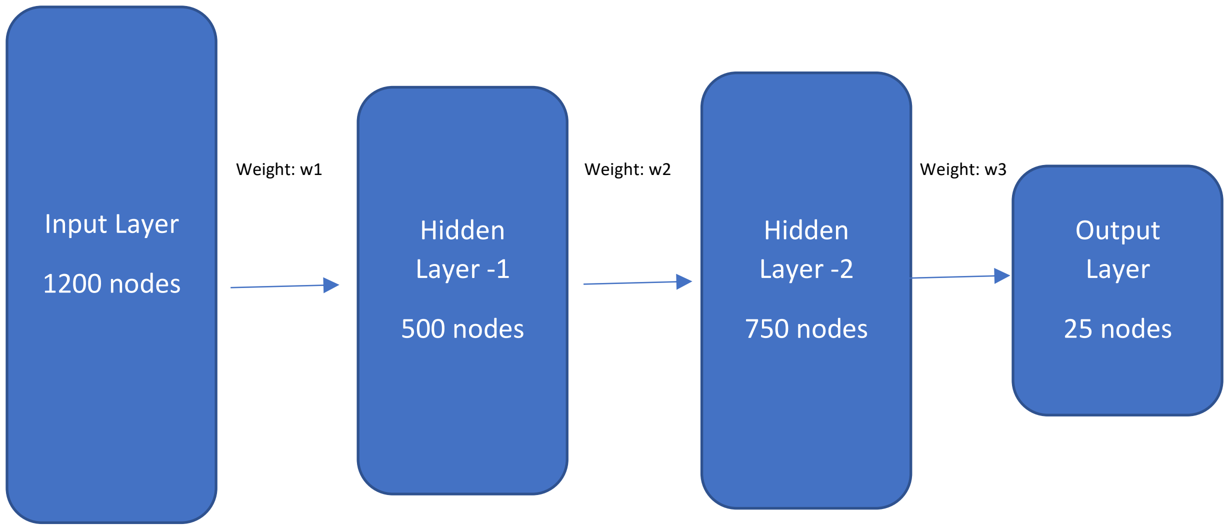
HW-3

Prithul Sarker

CS 687 Fundamentals of Deep Learning

1.



(a) Size of input feature space is 1200 as input layer has 1200 nodes.

(b) The number of parameters in 1st hidden layer is 500x1200 = 600000, as the input and 1st hidden layer have 1200 and 500 nodes, respectively.

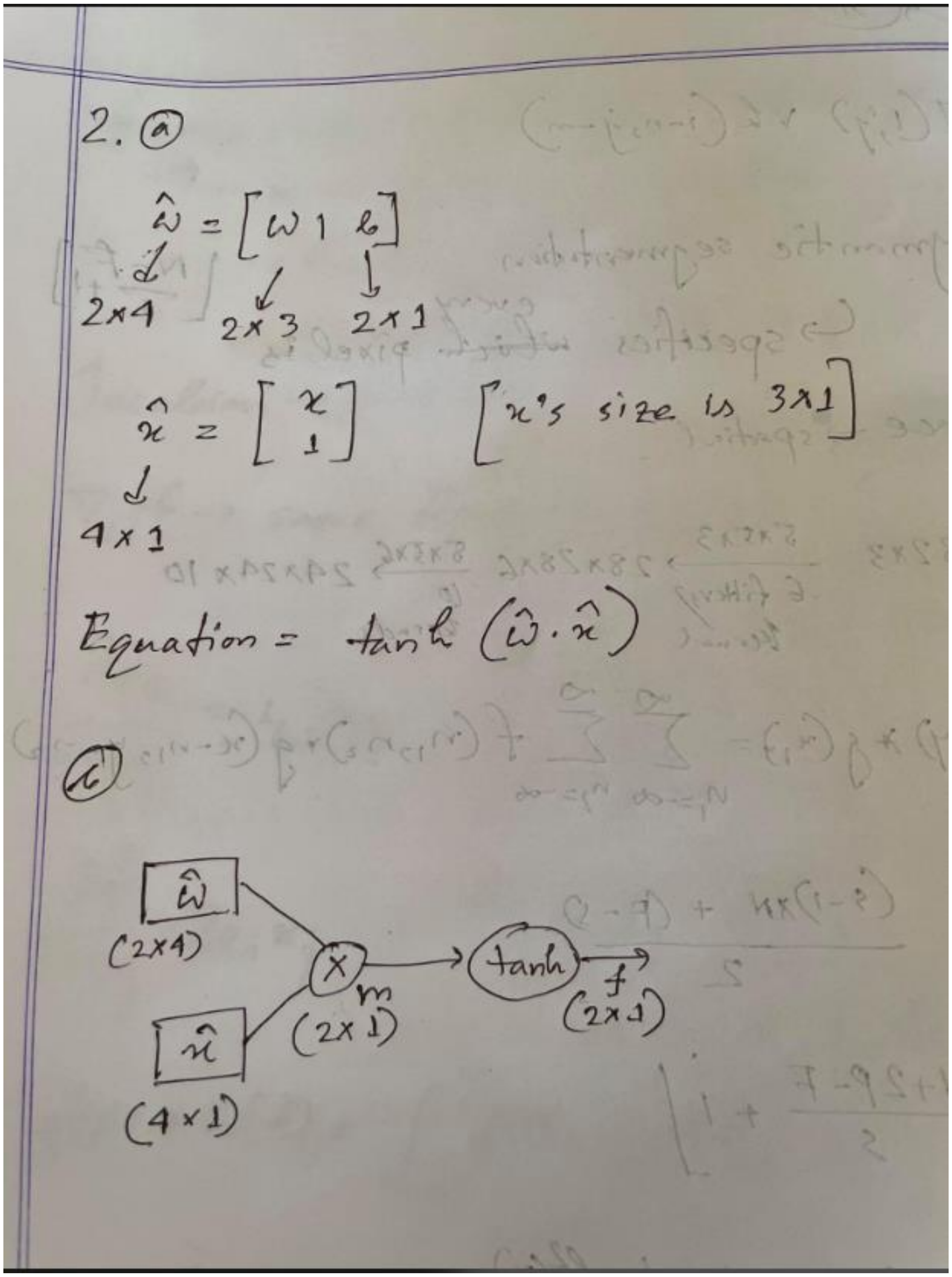
(c) The number of parameters in 2nd hidden layer is 750x500 = 375000, as the 1st and 2nd hidden layer have 500 and 750 nodes, respectively.

(d) The number of parameters in output layer is 25x750 = 18750, as the 2nd hidden layer and output layer have 750 and 25 nodes, respectively.

(e) The number of parameters of the overall network is 25x750x500x1200 = 11,250,000,000, since the network is densely connected and so, every node is connected to connected to all nodes of previous layer and next layer.

(f) H-1 layer equation: f1 = tanh(w1. x) [where x is the input and w1 is the weight of the H-1 layer]
H-2 layer equation: f2 = tanh(w2. f1) [where w2 is the weight of the H-2 layer]
Output layer equation: f = tanh(w3.f2) [where w3 is the weight of the output layer]

2.



2. (a)

$$\underset{2\times 4}{\hat{\omega}} = \left[\underset{2\times 3}{\omega} \mid \underset{2\times 1}{b}\right]$$

$$\underset{4\times 1}{\hat{x}} = \begin{bmatrix} x \\ 1 \end{bmatrix} \qquad [x\text{'s size is } 3\times 1]$$

Equation = $\tanh(\hat{\omega}\cdot\hat{x})$

(b)

$$\begin{tikzcd} \underset{(2\times 4)}{\boxed{\hat{\omega}}} \arrow[dr] & & & \\ & \times \arrow[r, "\underset{(2\times 1)}{m}"] & \tanh \arrow[r, "\underset{(2\times 1)}{f}"] & {} \\ \underset{(4\times 1)}{\boxed{\hat{x}}} \arrow[ur] & & & \end{tikzcd}$$

(c) $f = \tanh(m)$

$$\frac{\partial f}{\partial f} = 1$$

$$\frac{\partial f}{\partial m} = 1 - \tanh^2(m) = \begin{bmatrix} 1 - \tanh^2(m_1) \\ 1 - \tanh^2(m_2) \end{bmatrix}$$

$m = \hat{\omega} \cdot \hat{x}$

$$\frac{\partial m}{\partial \hat{x}} = \hat{\omega}^T \qquad \frac{\partial m}{\partial x} = \omega^T \qquad [\text{where } m = \omega x + b]$$

$$\frac{\partial m}{\partial \hat{\omega}} = \hat{x}^T \qquad \frac{\partial m}{\partial \omega} = x^T$$

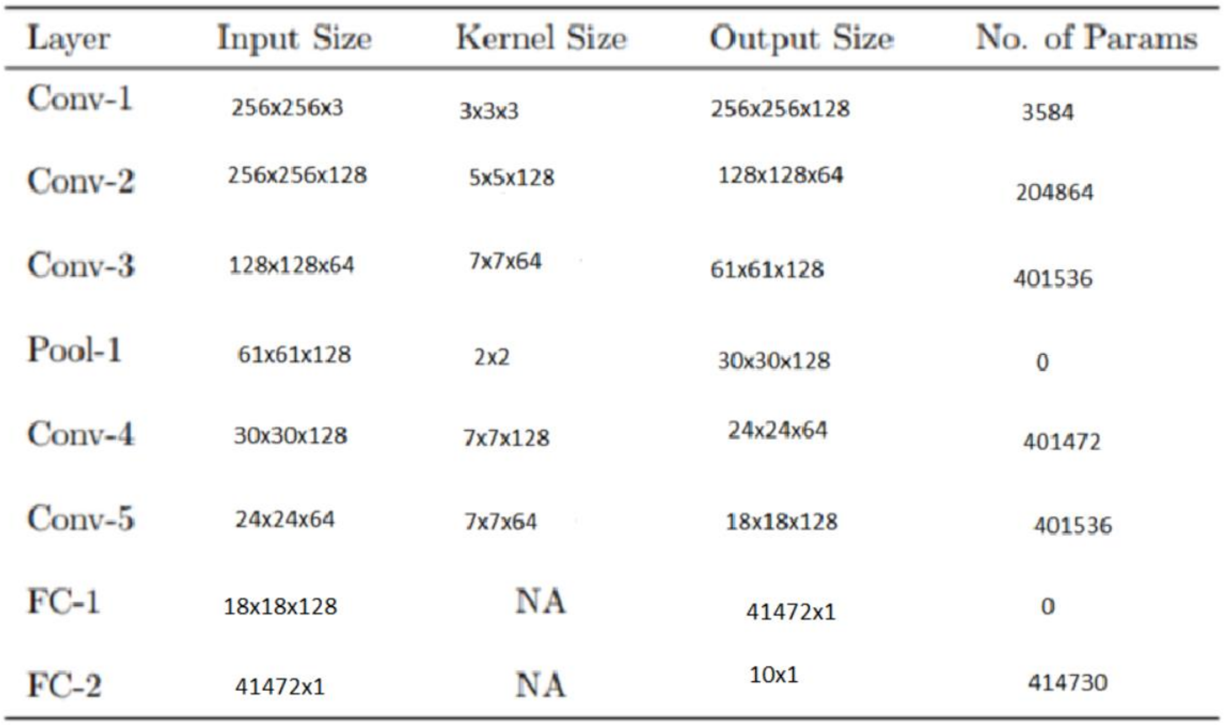
$$\frac{\partial f}{\partial \hat{x}} = \frac{\partial m}{\partial \hat{x}} \cdot \frac{\partial f}{\partial m} = \hat{\omega}^T \cdot \frac{\partial f}{\partial m}$$

[size of $\hat{\omega}^T$ is 4×2
and size of $\frac{\partial f}{\partial m}$ is 2×1.
The overall size of $\frac{\partial f}{\partial \hat{x}}$ is 4×1]

$$\frac{\partial f}{\partial \hat{\omega}} = \frac{\partial f}{\partial m} \cdot \frac{\partial m}{\partial \hat{\omega}} = \frac{\partial f}{\partial m} \cdot \hat{x}^T \qquad [\text{final size is } 2 \times 4]$$

$$\frac{\partial f}{\partial x} = \frac{\partial m}{\partial x} \cdot \frac{\partial f}{\partial m} = \omega^T \cdot \frac{\partial f}{\partial m} \qquad [\text{final size is } 3 \times 1]$$

3.



Layer	Input Size	Kernel Size	Output Size	No. of Params
Conv-1	256x256x3	3x3x3	256x256x128	3584
Conv-2	256x256x128	5x5x128	128x128x64	204864
Conv-3	128x128x64	7x7x64	61x61x128	401536
Pool-1	61x61x128	2x2	30x30x128	0
Conv-4	30x30x128	7x7x128	24x24x64	401472
Conv-5	24x24x64	7x7x64	18x18x128	401536
FC-1	18x18x128	NA	41472x1	0
FC-2	41472x1	NA	10x1	414730

4. (a)

In GoogLeNet, at the inception layer, 1 x 1 convolution layer is used before applying larger convolution. The main reason for this is the dimension reduction and eventually lesser operation which results in faster neural network. A parallel max pooling layer is used directly to the input and then 1 x 1 convolution is used to make its dimension lower. Here, less filter channel is used. Finally, all the result is concatenated. To make the concatenation work, all previous convolution goes through same padding convolution method.

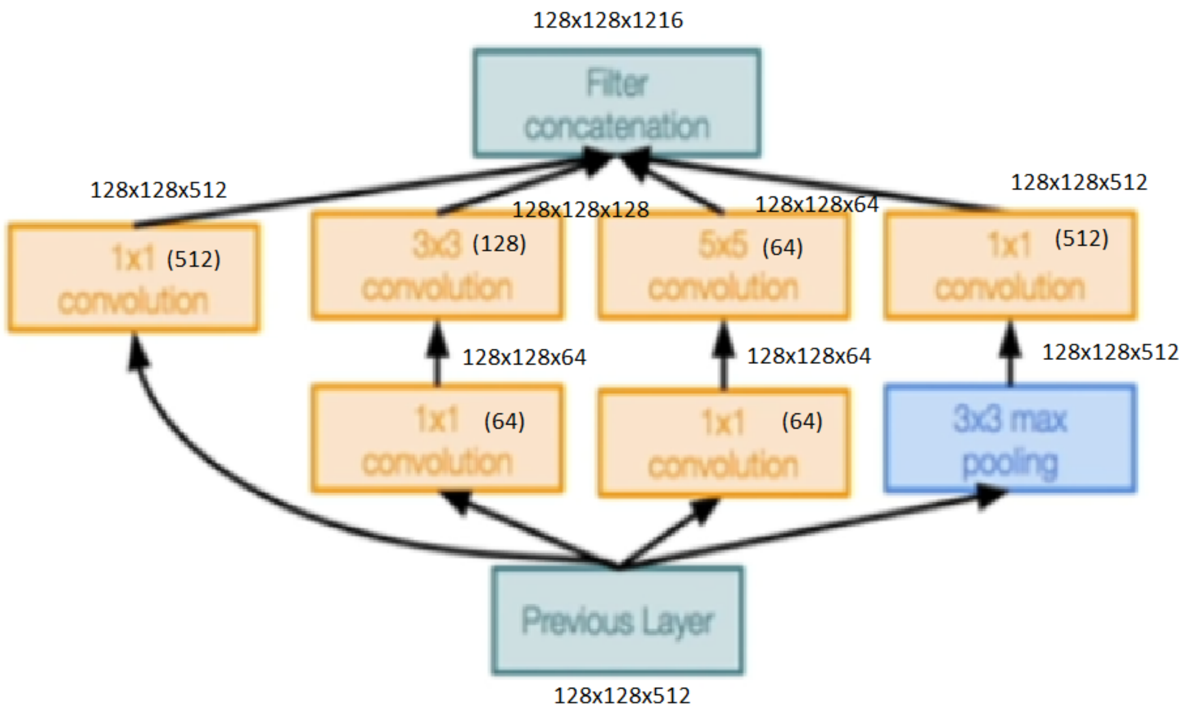
4. (b)

1. Considering s = 1, the padding required to have same dimension is = (F – 1) / 2

The padding used in both sides for 3 x 3 convolution is (3 – 1)/ 2 = 1

The padding used in both sides for 5 x 5 convolution is (5 – 1)/ 2 = 2

2.



3. Number of operations:

[1x1 conv, 64] - > 128 x 128 x 64 x 1 x 1 x 512 = 536,870,912

[1x1 conv, 64] - > 128 x 128 x 64 x 1 x 1 x 512 = 536,870,912

[3x3 max pooling] - > No operation

[1x1 conv, 512] - > 128 x 128 x 512 x 1 x 1 x 512 = 4,294,967,296

[3x3 conv, 128] - > 128 x 128 x 128 x 3 x 3 x 64 = 1,207,959,552

[5x5 conv, 64] - > 128 x 128 x 64 x 5 x 5 x 64 = 1,677,721,600

[1x1 conv, 512] - > 128 x 128 x 512 x 1 x 1 x 512 = 4,294,967,296

4. Disregarding bias, number of parameters:

[1x1 conv, 64] - > 1 x 1 x 512 x 64 = 32768

[1x1 conv, 64] - > 1 x 1 x 512 x 64 = 32768

[3x3 max pooling] - > No parameter

[1x1 conv, 512] - > 1 x 1 x 512 x 512 = 262144

[3x3 conv, 128] - > 3 x 3 x 64 x 128 = 73728

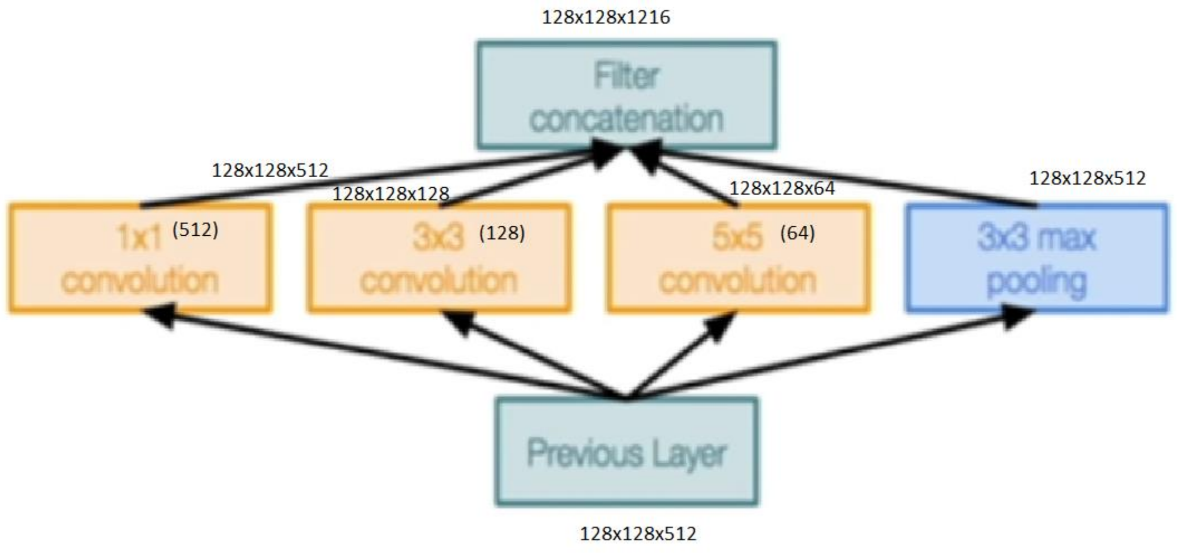
[5x5 conv, 64] - > 5 x 5 x 64 x 64 = 102400

[1x1 conv, 512] - > 1 x 1 x 512 x 512 = 262144

5. Number of operations of the whole module = 12,549 M

6. Number of parameters of the whole module = 765952

4. (c)



(Considering the 1x1 module has a depth of 512)

The number of operations = 128x128x512x1x1x512 + 128x128x128x3x3x512 + 128x128x64x5x5x512

= 27,380 M

The number of parameters = 1 x 1 x 512 x 512 + 3 x 3 x 512 x 128 + 5 x 5 x 512 x 64

= 1,671,168

GoogLeNet inception module needs to make 12.5 billion operations and has 765,952 parameters, whereas the naïve inception module needs to do 27.3 billion operations has 1,671,168 parameters. Naïve module has more than 2 times of parameters and operations to execute which result in more time consumption and more complexity with every layer. Therefore, GoogLeNet is more efficient than the naïve module with respect to cost of operations.

4. (d)

In GoogLeNet, the inception module has 1x1 convolution before doing any larger convolution. This is done because of the reduction of dimension and it preserves all necessary information in reduced dimensions. So, when the layer with lower dimension is gone through larger convolution, it produces less parameters even though it passes through two convolutions. On the other hand, for the naïve inception module, the convolution is done with the input as the way it was which results in more parameters as a high number of channels is multiplied with another high number.

5.

Report:

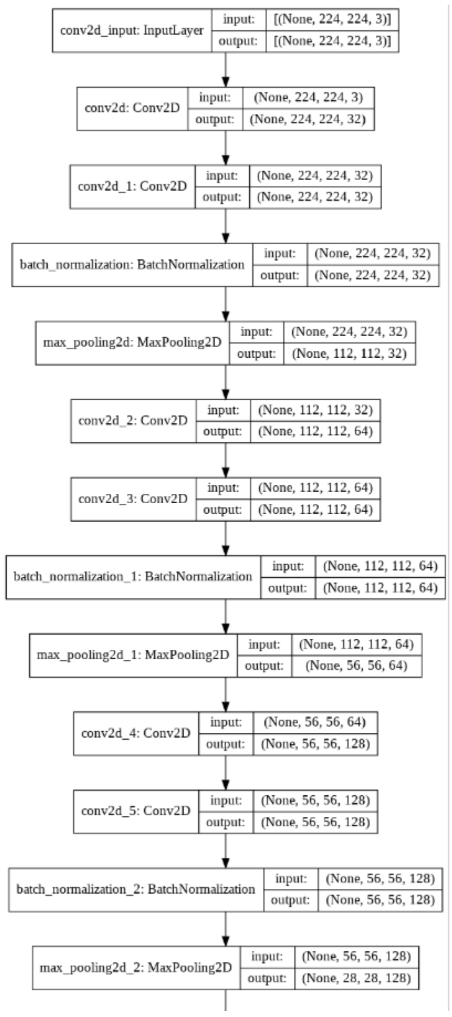
The task of this report is to create an architecture similar like VGG network. Karen Simonyan and Andrew Zisserman introduced this architecture in their paper- “Very Deep Convolutional Networks for Large Scale Image Recognition”. They used several layers of combination of multiple convolution layer and max-pooling layer before dense layers. This was a state-of-the-art architecture when the paper was published. For this task, we have a dataset of cats and dogs. So, we are to create an architecture for the classification of the dogs and cats. The training dataset contains 4000 images for training, 600 images for validation and 200 images for testing. The images of different classes are given in different subfolders.

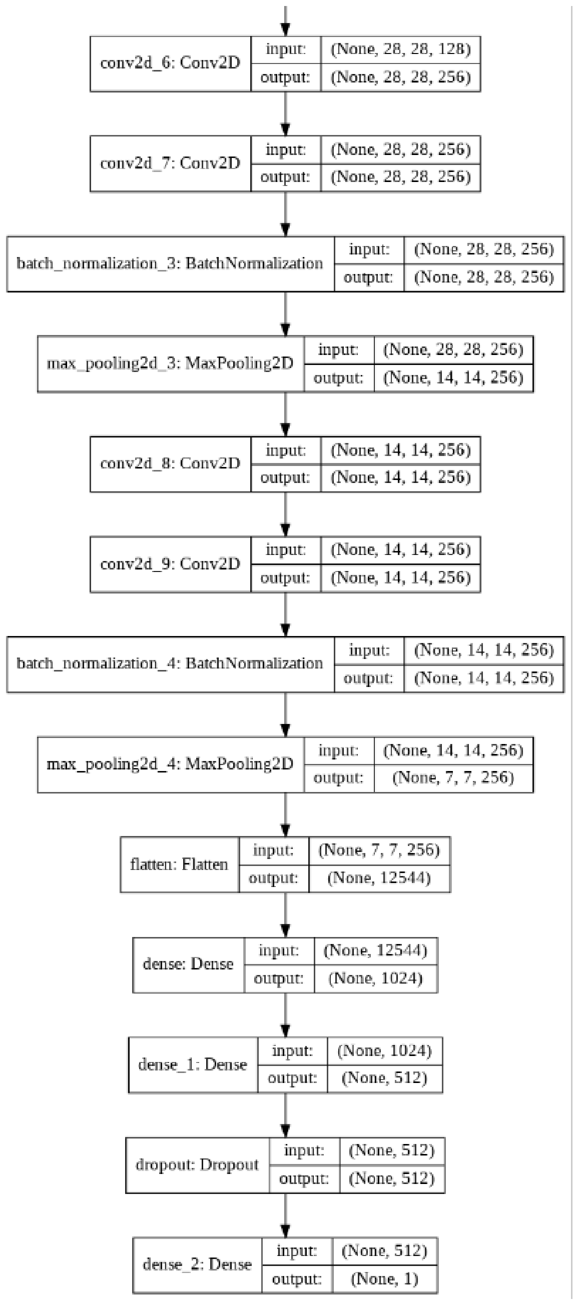
I used two approaches with different kernel size and dense layers.

a. Same kernel size and different dense layers with input shape 152x152x3
b. Different kernel size and dense layers with input shape 224x224x3

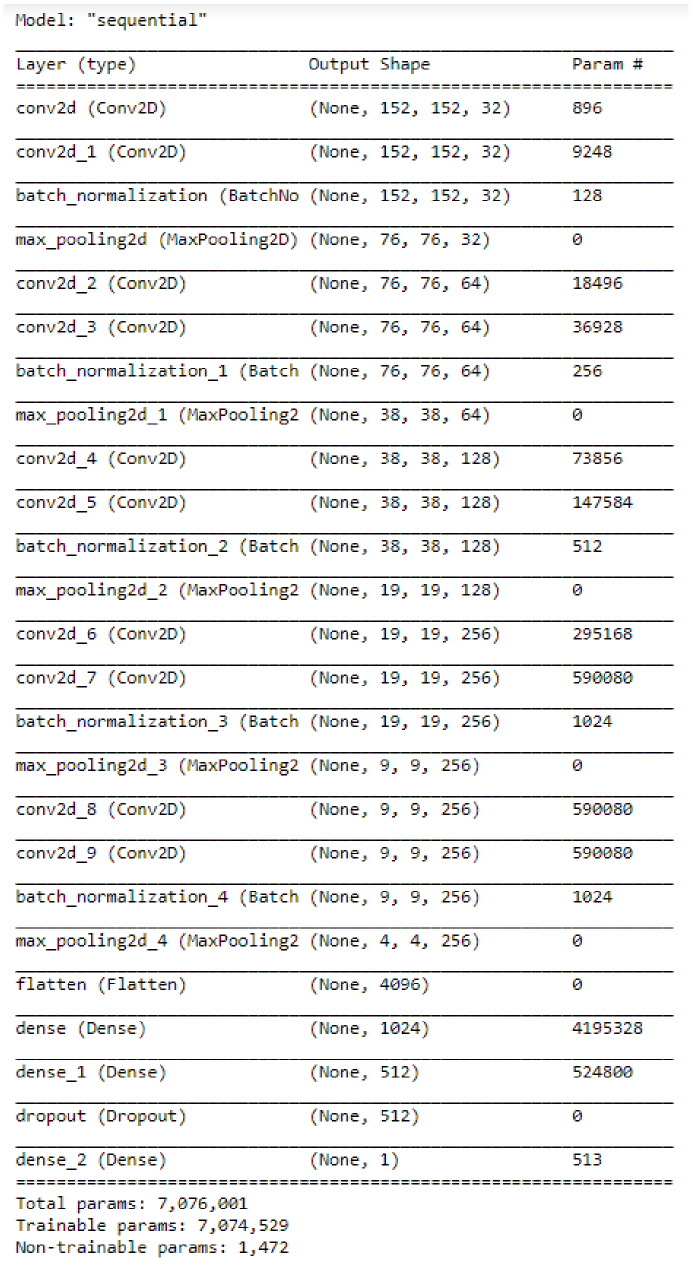
<u>Network Architecture:</u>

<u>First approach:</u> using same kernel sizes and different dense layers changing input size to 152x152x3.


conv2d_input: InputLayer | input: [(None, 224, 224, 3)] | output: [(None, 224, 224, 3)]
conv2d: Conv2D | input: (None, 224, 224, 3) | output: (None, 224, 224, 32)
conv2d_1: Conv2D | input: (None, 224, 224, 32) | output: (None, 224, 224, 32)
batch_normalization: BatchNormalization | input: (None, 224, 224, 32) | output: (None, 224, 224, 32)
max_pooling2d: MaxPooling2D | input: (None, 224, 224, 32) | output: (None, 112, 112, 32)
conv2d_2: Conv2D | input: (None, 112, 112, 32) | output: (None, 112, 112, 64)
conv2d_3: Conv2D | input: (None, 112, 112, 64) | output: (None, 112, 112, 64)
batch_normalization_1: BatchNormalization | input: (None, 112, 112, 64) | output: (None, 112, 112, 64)
max_pooling2d_1: MaxPooling2D | input: (None, 112, 112, 64) | output: (None, 56, 56, 64)
conv2d_4: Conv2D | input: (None, 56, 56, 64) | output: (None, 56, 56, 128)
conv2d_5: Conv2D | input: (None, 56, 56, 128) | output: (None, 56, 56, 128)
batch_normalization_2: BatchNormalization | input: (None, 56, 56, 128) | output: (None, 56, 56, 128)
max_pooling2d_2: MaxPooling2D | input: (None, 56, 56, 128) | output: (None, 28, 28, 128)


conv2d_6: Conv2D | input: (None, 28, 28, 128) | output: (None, 28, 28, 256)
conv2d_7: Conv2D | input: (None, 28, 28, 256) | output: (None, 28, 28, 256)
batch_normalization_3: BatchNormalization | input: (None, 28, 28, 256) | output: (None, 28, 28, 256)
max_pooling2d_3: MaxPooling2D | input: (None, 28, 28, 256) | output: (None, 14, 14, 256)
conv2d_8: Conv2D | input: (None, 14, 14, 256) | output: (None, 14, 14, 256)
conv2d_9: Conv2D | input: (None, 14, 14, 256) | output: (None, 14, 14, 256)
batch_normalization_4: BatchNormalization | input: (None, 14, 14, 256) | output: (None, 14, 14, 256)
max_pooling2d_4: MaxPooling2D | input: (None, 14, 14, 256) | output: (None, 7, 7, 256)
flatten: Flatten | input: (None, 7, 7, 256) | output: (None, 12544)
dense: Dense | input: (None, 12544) | output: (None, 1024)
dense_1: Dense | input: (None, 1024) | output: (None, 512)
dropout: Dropout | input: (None, 512) | output: (None, 512)
dense_2: Dense | input: (None, 512) | output: (None, 1)

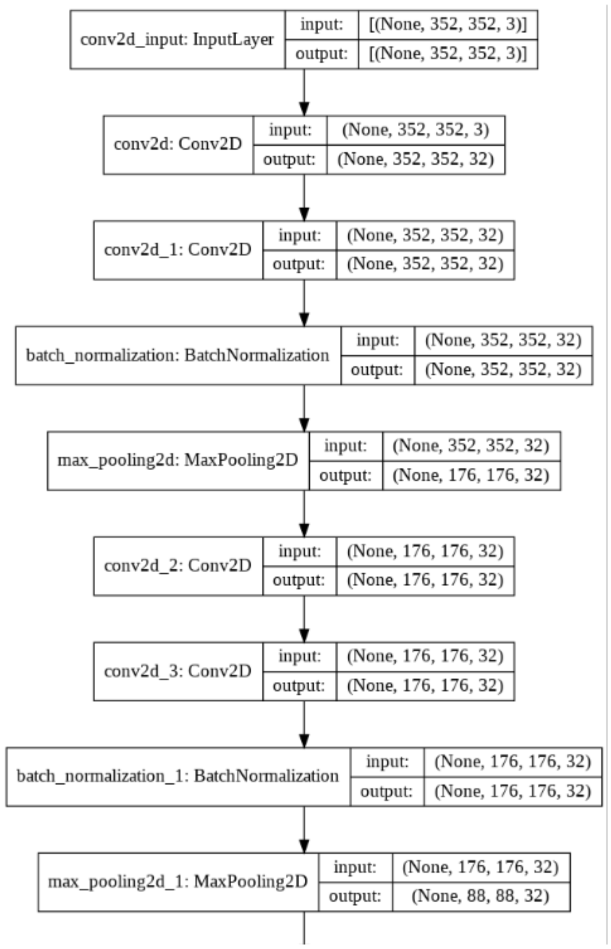
The summary of the network (first approach) is as follows:

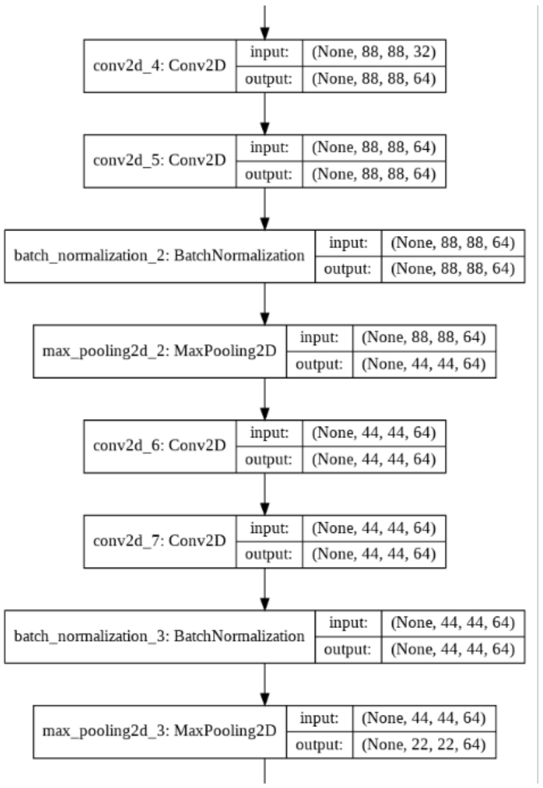


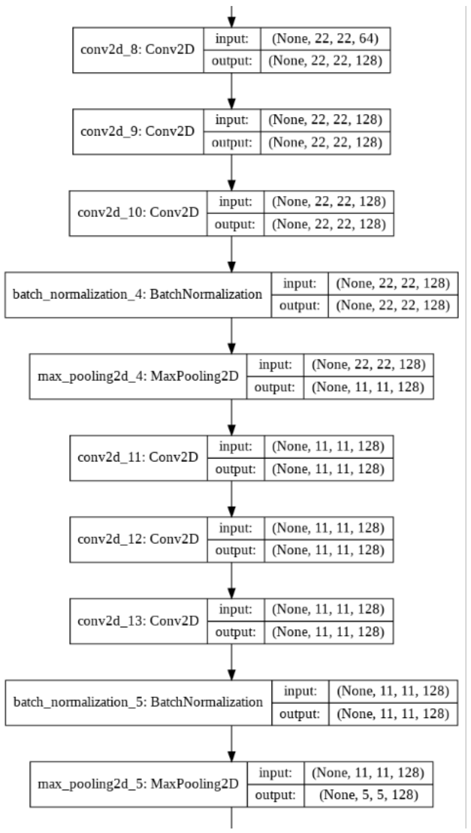
```
Model: "sequential"
_________________________________________________________________
Layer (type)                 Output Shape              Param #
=================================================================
conv2d (Conv2D)              (None, 152, 152, 32)      896
_________________________________________________________________
conv2d_1 (Conv2D)            (None, 152, 152, 32)      9248
_________________________________________________________________
batch_normalization (BatchNo (None, 152, 152, 32)      128
_________________________________________________________________
max_pooling2d (MaxPooling2D) (None, 76, 76, 32)        0
_________________________________________________________________
conv2d_2 (Conv2D)            (None, 76, 76, 64)        18496
_________________________________________________________________
conv2d_3 (Conv2D)            (None, 76, 76, 64)        36928
_________________________________________________________________
batch_normalization_1 (Batch (None, 76, 76, 64)        256
_________________________________________________________________
max_pooling2d_1 (MaxPooling2 (None, 38, 38, 64)        0
_________________________________________________________________
conv2d_4 (Conv2D)            (None, 38, 38, 128)       73856
_________________________________________________________________
conv2d_5 (Conv2D)            (None, 38, 38, 128)       147584
_________________________________________________________________
batch_normalization_2 (Batch (None, 38, 38, 128)       512
_________________________________________________________________
max_pooling2d_2 (MaxPooling2 (None, 19, 19, 128)       0
_________________________________________________________________
conv2d_6 (Conv2D)            (None, 19, 19, 256)       295168
_________________________________________________________________
conv2d_7 (Conv2D)            (None, 19, 19, 256)       590080
_________________________________________________________________
batch_normalization_3 (Batch (None, 19, 19, 256)       1024
_________________________________________________________________
max_pooling2d_3 (MaxPooling2 (None, 9, 9, 256)         0
_________________________________________________________________
conv2d_8 (Conv2D)            (None, 9, 9, 256)         590080
_________________________________________________________________
conv2d_9 (Conv2D)            (None, 9, 9, 256)         590080
_________________________________________________________________
batch_normalization_4 (Batch (None, 9, 9, 256)         1024
_________________________________________________________________
max_pooling2d_4 (MaxPooling2 (None, 4, 4, 256)         0
_________________________________________________________________
flatten (Flatten)            (None, 4096)              0
_________________________________________________________________
dense (Dense)                (None, 1024)              4195328
_________________________________________________________________
dense_1 (Dense)              (None, 512)               524800
_________________________________________________________________
dropout (Dropout)            (None, 512)               0
_________________________________________________________________
dense_2 (Dense)              (None, 1)                 513
=================================================================
Total params: 7,076,001
Trainable params: 7,074,529
Non-trainable params: 1,472
```

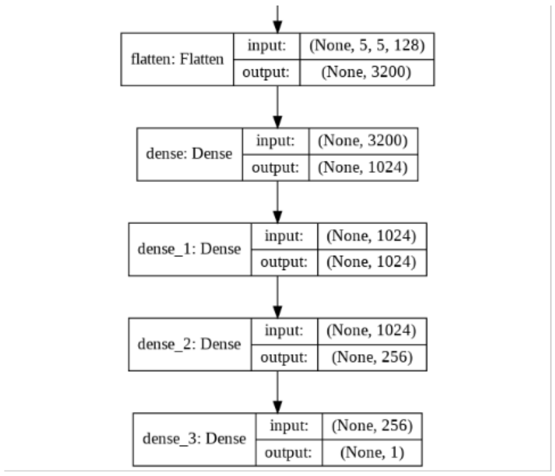
Here, there are over 7 million parameters and of that 1472 parameters are non-trainable and these come from the batch normalization layer. Batch normalization is used to make the values smaller. In the network (first approach), at first we got two 32 filters of 3x3 convolution, followed by batch normalization and max pooling layer. Then we added two 64 filters of 3x3 convolution, followed by batch normalization and max pooling layer. Next, we added two 128 filters of 3x3 convolution and then batch normalization and max pooling layer. For final convolution layer, the architecture has got two 256 filters of 3x3 convolution, followed by batch normalization and max pooling layer. The similar convolution, batch normalization and max pooling layer is again used. Then we used flatten function before making the layers fully connected. Finally, we have two dense layers of units 1024 and 512 before the last dense layer of unit 1 for binary classification. Rather than using random initializer, we used “he_uniform” initializer and l2 regularizer to get better result. At the final dense layer, we used sigmoid activation function because sigmoid performs better for binary classification. This model performs with best accuracy. However, to complete this model, we had to change the input size from 224x224x3 to 152x152x3. This model has more parameter than my second approach.

<u>Second approach</u>: Maintaining the input size 224x224x3 and using different kernel size and dense layers. The architecture of the network is as follows:

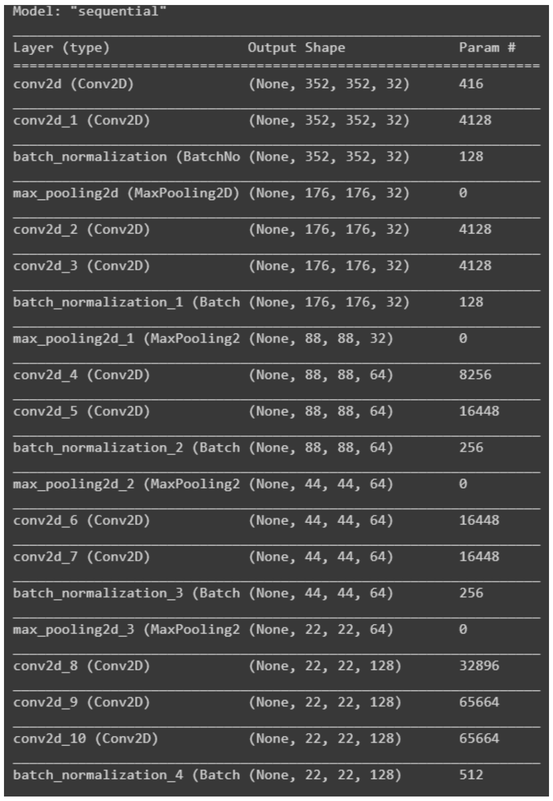

conv2d_input: InputLayer | input: [(None, 352, 352, 3)] | output: [(None, 352, 352, 3)]
conv2d: Conv2D | input: (None, 352, 352, 3) | output: (None, 352, 352, 32)
conv2d_1: Conv2D | input: (None, 352, 352, 32) | output: (None, 352, 352, 32)
batch_normalization: BatchNormalization | input: (None, 352, 352, 32) | output: (None, 352, 352, 32)
max_pooling2d: MaxPooling2D | input: (None, 352, 352, 32) | output: (None, 176, 176, 32)
conv2d_2: Conv2D | input: (None, 176, 176, 32) | output: (None, 176, 176, 32)
conv2d_3: Conv2D | input: (None, 176, 176, 32) | output: (None, 176, 176, 32)
batch_normalization_1: BatchNormalization | input: (None, 176, 176, 32) | output: (None, 176, 176, 32)
max_pooling2d_1: MaxPooling2D | input: (None, 176, 176, 32) | output: (None, 88, 88, 32)


conv2d_4: Conv2D
input: (None, 88, 88, 32)
output: (None, 88, 88, 64)
conv2d_5: Conv2D
input: (None, 88, 88, 64)
output: (None, 88, 88, 64)
batch_normalization_2: BatchNormalization
input: (None, 88, 88, 64)
output: (None, 88, 88, 64)
max_pooling2d_2: MaxPooling2D
input: (None, 88, 88, 64)
output: (None, 44, 44, 64)
conv2d_6: Conv2D
input: (None, 44, 44, 64)
output: (None, 44, 44, 64)
conv2d_7: Conv2D
input: (None, 44, 44, 64)
output: (None, 44, 44, 64)
batch_normalization_3: BatchNormalization
input: (None, 44, 44, 64)
output: (None, 44, 44, 64)
max_pooling2d_3: MaxPooling2D
input: (None, 44, 44, 64)
output: (None, 22, 22, 64)

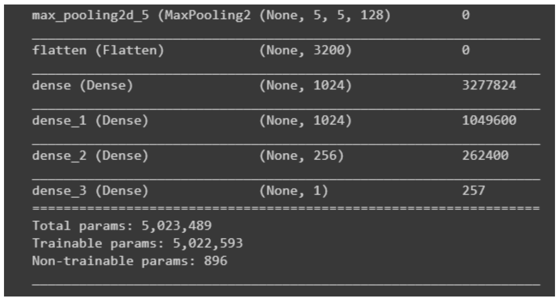

conv2d_8: Conv2D
input: (None, 22, 22, 64)
output: (None, 22, 22, 128)
conv2d_9: Conv2D
input: (None, 22, 22, 128)
output: (None, 22, 22, 128)
conv2d_10: Conv2D
input: (None, 22, 22, 128)
output: (None, 22, 22, 128)
batch_normalization_4: BatchNormalization
input: (None, 22, 22, 128)
output: (None, 22, 22, 128)
max_pooling2d_4: MaxPooling2D
input: (None, 22, 22, 128)
output: (None, 11, 11, 128)
conv2d_11: Conv2D
input: (None, 11, 11, 128)
output: (None, 11, 11, 128)
conv2d_12: Conv2D
input: (None, 11, 11, 128)
output: (None, 11, 11, 128)
conv2d_13: Conv2D
input: (None, 11, 11, 128)
output: (None, 11, 11, 128)
batch_normalization_5: BatchNormalization
input: (None, 11, 11, 128)
output: (None, 11, 11, 128)
max_pooling2d_5: MaxPooling2D
input: (None, 11, 11, 128)
output: (None, 5, 5, 128)



The summary of the network is as follows:



```
Model: "sequential"
_________________________________________________________________
Layer (type)                 Output Shape              Param #
=================================================================
conv2d (Conv2D)              (None, 352, 352, 32)      416
_________________________________________________________________
conv2d_1 (Conv2D)            (None, 352, 352, 32)      4128
_________________________________________________________________
batch_normalization (BatchNo (None, 352, 352, 32)      128
_________________________________________________________________
max_pooling2d (MaxPooling2D) (None, 176, 176, 32)      0
_________________________________________________________________
conv2d_2 (Conv2D)            (None, 176, 176, 32)      4128
_________________________________________________________________
conv2d_3 (Conv2D)            (None, 176, 176, 32)      4128
_________________________________________________________________
batch_normalization_1 (Batch (None, 176, 176, 32)      128
_________________________________________________________________
max_pooling2d_1 (MaxPooling2 (None, 88, 88, 32)        0
_________________________________________________________________
conv2d_4 (Conv2D)            (None, 88, 88, 64)        8256
_________________________________________________________________
conv2d_5 (Conv2D)            (None, 88, 88, 64)        16448
_________________________________________________________________
batch_normalization_2 (Batch (None, 88, 88, 64)        256
_________________________________________________________________
max_pooling2d_2 (MaxPooling2 (None, 44, 44, 64)        0
_________________________________________________________________
conv2d_6 (Conv2D)            (None, 44, 44, 64)        16448
_________________________________________________________________
conv2d_7 (Conv2D)            (None, 44, 44, 64)        16448
_________________________________________________________________
batch_normalization_3 (Batch (None, 44, 44, 64)        256
_________________________________________________________________
max_pooling2d_3 (MaxPooling2 (None, 22, 22, 64)        0
_________________________________________________________________
conv2d_8 (Conv2D)            (None, 22, 22, 128)       32896
_________________________________________________________________
conv2d_9 (Conv2D)            (None, 22, 22, 128)       65664
_________________________________________________________________
conv2d_10 (Conv2D)           (None, 22, 22, 128)       65664
_________________________________________________________________
batch_normalization_4 (Batch (None, 22, 22, 128)       512
```



```
max_pooling2d_5 (MaxPooling2 (None, 5, 5, 128)         0
_________________________________________________________________
flatten (Flatten)            (None, 3200)              0
_________________________________________________________________
dense (Dense)                (None, 1024)              3277824
_________________________________________________________________
dense_1 (Dense)              (None, 1024)              1049600
_________________________________________________________________
dense_2 (Dense)              (None, 256)               262400
_________________________________________________________________
dense_3 (Dense)              (None, 1)                 257
=================================================================
Total params: 5,023,489
Trainable params: 5,022,593
Non-trainable params: 896
_________________________________________________________________
```

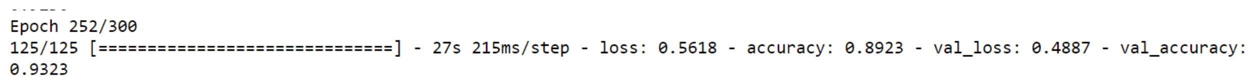
The second VGG-lite architecture has about 5 million parameters and 896 of them are non-trainable. This is because of the batch normalization layers. Batch normalization is used to make the values smaller. In the network, at first we got two 32 filters of 2x2 convolution, followed by batch normalization and max pooling layer. The similar convolution, batch normalization and max pooling layer is again used. Then we got two 64 filters of 2x2 convolution, followed by batch normalization and max pooling layer. The similar convolution, batch normalization and max pooling layer is again used. For final convolution layer, the architecture has got three 128 filters of 2x2 convolution, followed by batch normalization and max pooling layer. The similar convolution, batch normalization and max pooling layer is again used. Then we used flatten function before making the layers fully connected. Finally, we have three dense layers of units 1024, 1024 an 256 before the last dense layer of unit 1 for binary classification. This light model is used because we wanted to have less trainable parameters so that at the end we can have better accuracy. Rather than using random initializer, we used "he_uniform" initializer and l2 regularizer to get better result. At the final dense layer, we used sigmoid activation function because sigmoid performs better for binary classification.

As stated earlier, the second approach has less parameters than the first approach and this method converges faster than the first approach. However, in terms of accuracy the first one goes over 90% for the test dataset.

For both cases, SGD (Stochastic Gradient Descent) with starting learning rate 0.001 and momentum 0.9 is used. ReduceLROnPlateau is used to reduce learning rate when the loss is stuck for 5 consecutive epochs.

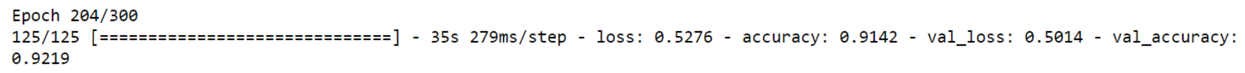
<u>Results:</u>

My first VGG-lite architecture for input size 152x152x3 got more than 93.23% accuracy for the validation dataset, 91.5% accuracy for the test dataset and, over 91% accuracy for the training dataset for the best epoch after 200 epochs. At epoch 252, the validation accuracy hit the maximum: 93.23%.



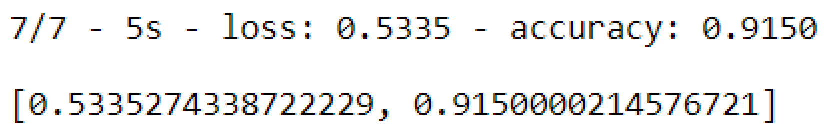
```
Epoch 252/300
125/125 [==============================] - 27s 215ms/step - loss: 0.5618 - accuracy: 0.8923 - val_loss: 0.4887 - val_accuracy:
0.9323
```

At epoch 204, the training accuracy is maximum: 91.42%.



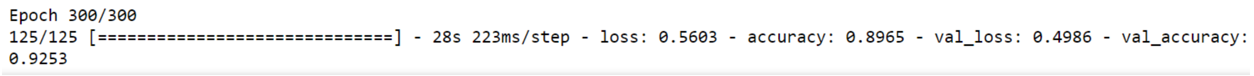
```
Epoch 204/300
125/125 [==============================] - 35s 279ms/step - loss: 0.5276 - accuracy: 0.9142 - val_loss: 0.5014 - val_accuracy:
0.9219
```

At 300 epoch, the accuracy for the test dataset is 91.50%. And the accuracy at epoch 300 for the training and validation dataset is 89.65% and 92.53% respectively.



```
7/7 - 5s - loss: 0.5335 - accuracy: 0.9150

[0.5335274338722229, 0.9150000214576721]
```

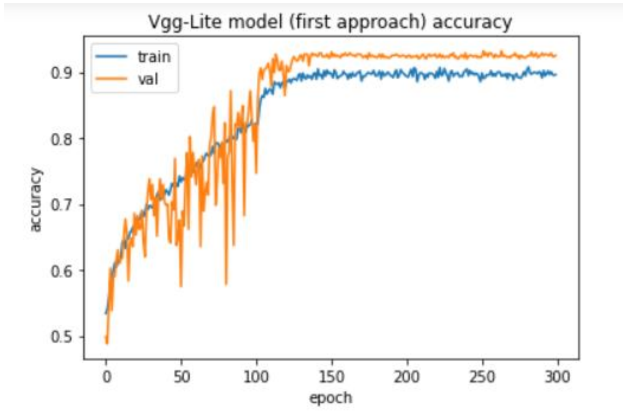


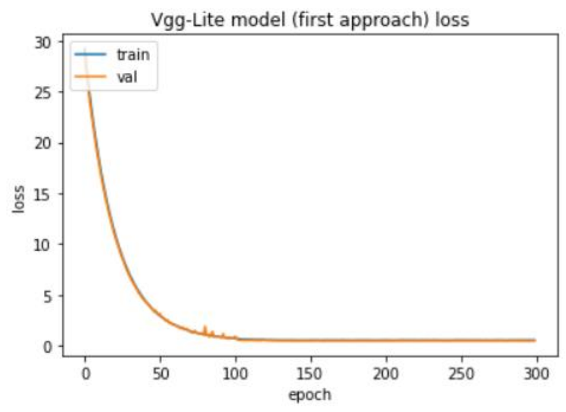
```
Epoch 300/300
125/125 [==============================] - 28s 223ms/step - loss: 0.5603 - accuracy: 0.8965 - val_loss: 0.4986 - val_accuracy:
0.9253
```

For the first approach, as 3x3 kernels is used and highly dense layer is used, there are more parameters which take more time to learn and take up more memory. The main reason for higher accuracy for images with lower dimension is the dataset size. More learning is possible from convolution with lower dimension images since more information is concentrated in less space.

My second VGG-lite architecture for input size 224x224x3 got more than 90% accuracy for the validation dataset, 87% accuracy for the test dataset and, 85% accuracy for the training dataset after 200 epochs.

At first, at the final stage, softmax was used for classification. In that case, the accuracy only gets about 50%. But since this is a binary classifier, the sigmoid activation function suits the best. Then again, the accuracy would go up to maximum 75%. This is because of the larger size of kernels. So when the kernel size is reduced to 2x2, the accuracy increases. Now again, since the dataset is small, the large size of dense layer at the end makes difference. So we needed to make the dense layer smaller which result in less number of parameters. Because of the small dataset, the less the number of parameters, the easier it would be to train the dataset. At that point, the accuracy hit the maximum with less filters and smaller size of the kernel.





From the graph, we can notice that the training and validation loss decreased in a steep manner till 100 epoch and then the value got saturated. For the accuracy, the validation accuracy is higher than training accuracy. The higher accuracy suggests that this architecture have done better than that of the second approach.

Time for each epoch at the starting for the first approach was 88 second whereas time for each epoch for the second approach at the starting was around 59 second.

For lower accuracy, the first concern is the small dataset. Even though we are doing multiple data augmentation using tensorflow generator, the network still needs more training data. If we do not augment the training data, the network accuracy would be less than 80%. The second concern for lower accuracy is the explicitness of training dataset. There are some images which are not related to the classification such as this one:

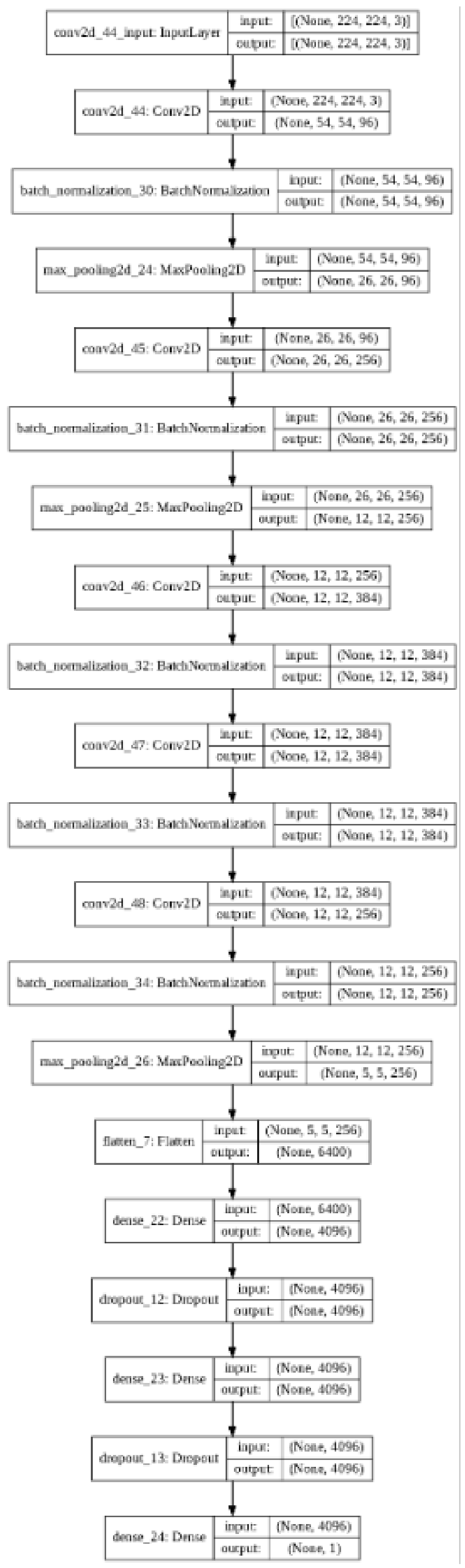
dog.1043.jpg

There are some images where the animal is not easily identifiable. This kind of images would be more appropriate if the training dataset were larger. This could be other reason for the lower accuracy.

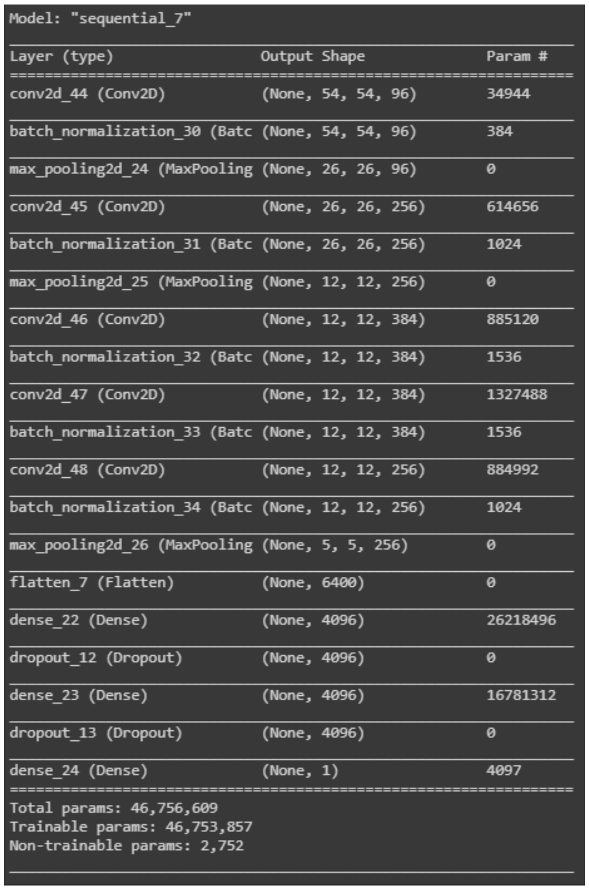
Being in 2021, we know we can use many other networks such as inception or residual network. Those would make better accuracy. Even though the VGG is a bit outdated, over 90% accuracy in validation and test dataset is more than what can be expected with this small dataset.

5. (h)

The architecture for AlexNet-Lite is as follows:

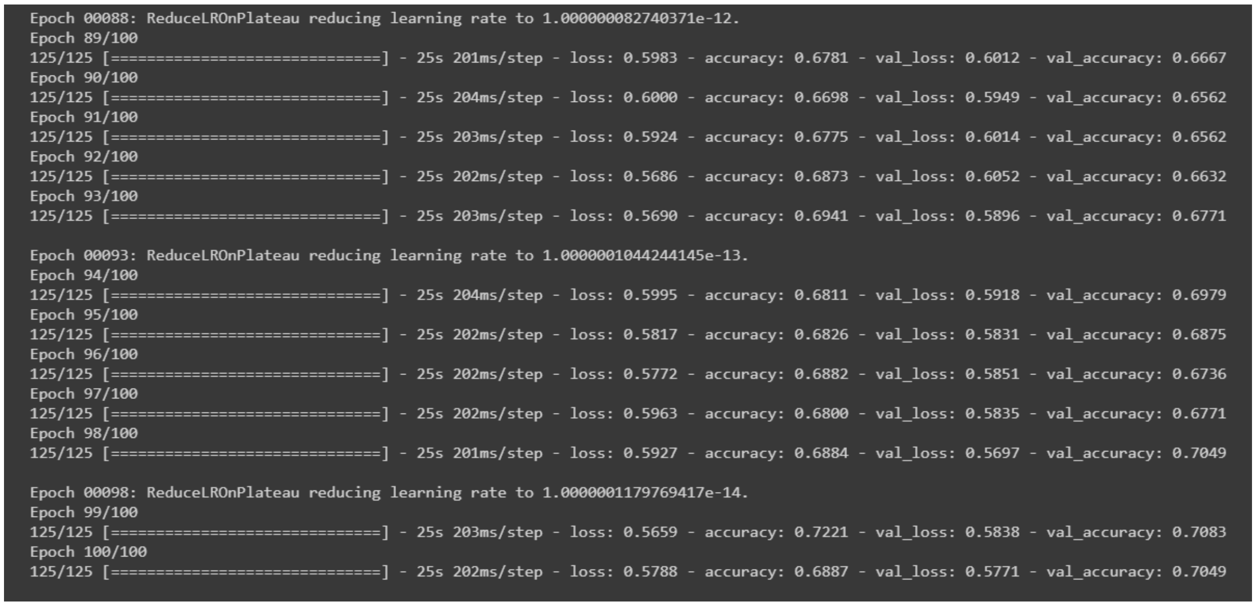

conv2d_44_input: InputLayer | input: [(None, 224, 224, 3)] | output: [(None, 224, 224, 3)]
conv2d_44: Conv2D | input: (None, 224, 224, 3) | output: (None, 54, 54, 96)
batch_normalization_30: BatchNormalization | input: (None, 54, 54, 96) | output: (None, 54, 54, 96)
max_pooling2d_24: MaxPooling2D | input: (None, 54, 54, 96) | output: (None, 26, 26, 96)
conv2d_45: Conv2D | input: (None, 26, 26, 96) | output: (None, 26, 26, 256)
batch_normalization_31: BatchNormalization | input: (None, 26, 26, 256) | output: (None, 26, 26, 256)
max_pooling2d_25: MaxPooling2D | input: (None, 26, 26, 256) | output: (None, 12, 12, 256)
conv2d_46: Conv2D | input: (None, 12, 12, 256) | output: (None, 12, 12, 384)
batch_normalization_32: BatchNormalization | input: (None, 12, 12, 384) | output: (None, 12, 12, 384)
conv2d_47: Conv2D | input: (None, 12, 12, 384) | output: (None, 12, 12, 384)
batch_normalization_33: BatchNormalization | input: (None, 12, 12, 384) | output: (None, 12, 12, 384)
conv2d_48: Conv2D | input: (None, 12, 12, 384) | output: (None, 12, 12, 256)
batch_normalization_34: BatchNormalization | input: (None, 12, 12, 256) | output: (None, 12, 12, 256)
max_pooling2d_26: MaxPooling2D | input: (None, 12, 12, 256) | output: (None, 5, 5, 256)
flatten_7: Flatten | input: (None, 5, 5, 256) | output: (None, 6400)
dense_22: Dense | input: (None, 6400) | output: (None, 4096)
dropout_12: Dropout | input: (None, 4096) | output: (None, 4096)
dense_23: Dense | input: (None, 4096) | output: (None, 4096)
dropout_13: Dropout | input: (None, 4096) | output: (None, 4096)
dense_24: Dense | input: (None, 4096) | output: (None, 1)

Following is the summary of AlexNet-Lite:



```
Model: "sequential_7"
_________________________________________________________________
Layer (type)                 Output Shape              Param #
=================================================================
conv2d_44 (Conv2D)           (None, 54, 54, 96)        34944
_________________________________________________________________
batch_normalization_30 (Batc (None, 54, 54, 96)        384
_________________________________________________________________
max_pooling2d_24 (MaxPooling (None, 26, 26, 96)        0
_________________________________________________________________
conv2d_45 (Conv2D)           (None, 26, 26, 256)       614656
_________________________________________________________________
batch_normalization_31 (Batc (None, 26, 26, 256)       1024
_________________________________________________________________
max_pooling2d_25 (MaxPooling (None, 12, 12, 256)       0
_________________________________________________________________
conv2d_46 (Conv2D)           (None, 12, 12, 384)       885120
_________________________________________________________________
batch_normalization_32 (Batc (None, 12, 12, 384)       1536
_________________________________________________________________
conv2d_47 (Conv2D)           (None, 12, 12, 384)       1327488
_________________________________________________________________
batch_normalization_33 (Batc (None, 12, 12, 384)       1536
_________________________________________________________________
conv2d_48 (Conv2D)           (None, 12, 12, 256)       884992
_________________________________________________________________
batch_normalization_34 (Batc (None, 12, 12, 256)       1024
_________________________________________________________________
max_pooling2d_26 (MaxPooling (None, 5, 5, 256)         0
_________________________________________________________________
flatten_7 (Flatten)          (None, 6400)              0
_________________________________________________________________
dense_22 (Dense)             (None, 4096)              26218496
_________________________________________________________________
dropout_12 (Dropout)         (None, 4096)              0
_________________________________________________________________
dense_23 (Dense)             (None, 4096)              16781312
_________________________________________________________________
dropout_13 (Dropout)         (None, 4096)              0
_________________________________________________________________
dense_24 (Dense)             (None, 1)                 4097
=================================================================
Total params: 46,756,609
Trainable params: 46,753,857
Non-trainable params: 2,752
_________________________________________________________________
```

Here is a screenshot of the final epochs:



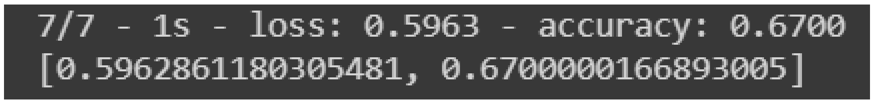
```
Epoch 00088: ReduceLROnPlateau reducing learning rate to 1.000000082740371e-12.
Epoch 89/100
125/125 [==============================] - 25s 201ms/step - loss: 0.5983 - accuracy: 0.6781 - val_loss: 0.6012 - val_accuracy: 0.6667
Epoch 90/100
125/125 [==============================] - 25s 204ms/step - loss: 0.6000 - accuracy: 0.6698 - val_loss: 0.5949 - val_accuracy: 0.6562
Epoch 91/100
125/125 [==============================] - 25s 203ms/step - loss: 0.5924 - accuracy: 0.6775 - val_loss: 0.6014 - val_accuracy: 0.6562
Epoch 92/100
125/125 [==============================] - 25s 202ms/step - loss: 0.5686 - accuracy: 0.6873 - val_loss: 0.6052 - val_accuracy: 0.6632
Epoch 93/100
125/125 [==============================] - 25s 203ms/step - loss: 0.5690 - accuracy: 0.6941 - val_loss: 0.5896 - val_accuracy: 0.6771

Epoch 00093: ReduceLROnPlateau reducing learning rate to 1.0000001044244145e-13.
Epoch 94/100
125/125 [==============================] - 25s 204ms/step - loss: 0.5995 - accuracy: 0.6811 - val_loss: 0.5918 - val_accuracy: 0.6979
Epoch 95/100
125/125 [==============================] - 25s 202ms/step - loss: 0.5817 - accuracy: 0.6826 - val_loss: 0.5831 - val_accuracy: 0.6875
Epoch 96/100
125/125 [==============================] - 25s 202ms/step - loss: 0.5772 - accuracy: 0.6882 - val_loss: 0.5851 - val_accuracy: 0.6736
Epoch 97/100
125/125 [==============================] - 25s 202ms/step - loss: 0.5963 - accuracy: 0.6800 - val_loss: 0.5835 - val_accuracy: 0.6771
Epoch 98/100
125/125 [==============================] - 25s 201ms/step - loss: 0.5927 - accuracy: 0.6884 - val_loss: 0.5697 - val_accuracy: 0.7049

Epoch 00098: ReduceLROnPlateau reducing learning rate to 1.0000001179769417e-14.
Epoch 99/100
125/125 [==============================] - 25s 203ms/step - loss: 0.5659 - accuracy: 0.7221 - val_loss: 0.5838 - val_accuracy: 0.7083
Epoch 100/100
125/125 [==============================] - 25s 202ms/step - loss: 0.5788 - accuracy: 0.6887 - val_loss: 0.5771 - val_accuracy: 0.7049
```

As it can be seen, the learning rate has reduced to 1e-14 because the architecture cannot get less loss and this is why learning rate is reducing after 5 consecutive epochs of stuck loss.

AlexNet got about 70-72% accuracy for the dataset after about 100 epochs. After that, the accuracy swings back and forth between 68 to 72% for both training and validation dataset. The training and validation loss for the AlexNet architecture is around 0.56. Here's the final evaluation on the test set.

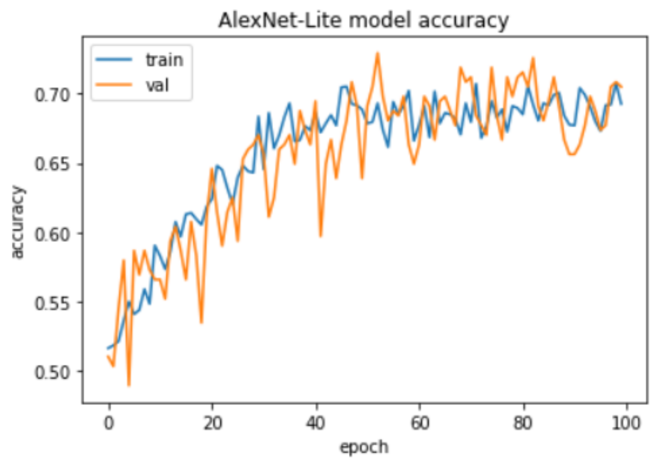


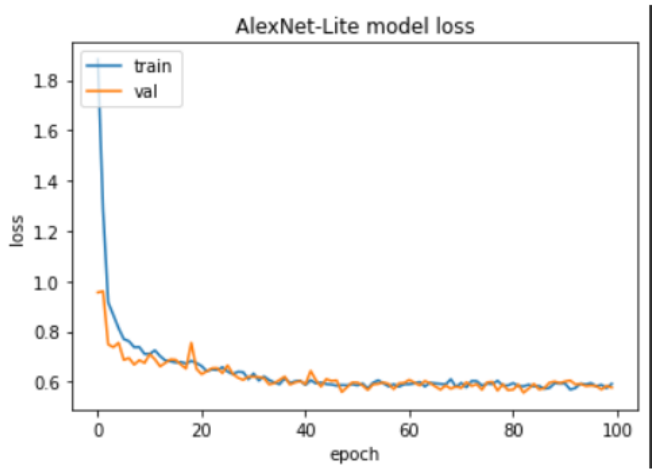
```
7/7 - 1s - loss: 0.5963 - accuracy: 0.6700
[0.5962861180305481, 0.6700000166893005]
```

The loss on the test dataset is 0.596 and the accuracy for the test dataset for the AlexNet-Lite is about 67%.

SGD (Stochastic Gradient Descent) with starting learning rate 0.001 and momentum 0.9 is used. ReduceLROnPlateau is used to reduce learning rate when the loss is stuck for 5 consecutive epochs. After 43 epochs, the learning rate reduced to 1e-5 and loss was stuck at 0.6029.

Below is the plot of loss and accuracy graph for training and validation dataset for AlexNet-Lite architecture:





From the graph, we can see that the loss gets saturated near 0.6 after about 50 epochs. Similar case is for the accuracy, it somewhat gets stuck after 50 epochs to 70%.

AlexNet-Lite has over 46 million parameters and VGGNet-Lite has about 7 million parameters. This is because AlexNet uses higher convolution values which take up more parameters. In terms of accuracy, the AlexNet does not reach over 75%. The accuracy could be better if we could optimize the kernel sizes and dense layers. The original AlexNet architecture is used here to complete the network.

To compare the accuracy, VGG=Lite did about 17% better than AlexNet-Lite. This is because AlexNetLite uses higher convolution which drops out many useful information and takes the average information. For VGG-Lite it uses smaller convolution with less strides and max pooling which keep most useful information. VGG-Lite model makes more generalized output than AlexNet-Lite. Again, in terms of speed, time for each epoch for AlexNet-Lite took only 28 second whereas it took over 85 second for each epoch for VGG-Lite.

Final loss for AlexNet-Lite is around 0.60 and final loss for VGG-Lite is around 0.90. In both cases, SGD with 0.9 momentum was used for optimizer. The loss for both cases was binary crossentropy.

Both models were then state-of-the-art models back in their published years. Although for now, these models might seem outdated, these models are the pioneer of the faster models of this time.