5 different types of image Classification(AlexNet,ResNet,InceptionV3,GoogleNet,Vgg16)

Rahaman Md. Mashiur (id: 18-37227-1), Riaz, Abdul Al Mahmud (id: 18-36989-1), MD SANOWAR HOSSAIN (id: 18-36896-1), Moktadir, Md. Golam (id: 18-37211-1),Masum Bepari (id: 18-36780-1)

[a]*Department of Computer Sciences, American International University-Bangladesh*

1. Introduction

1.Capture reader's interest - According to the Internet Center (IDC), the whole sum of worldwide information will reach 42ZB in 2020. And more than 70% of the data is transmitted by picture or video. To extricate valuable data from these pictures and video information, computer vision developed as the times require. At present, computer vision technology has developed rapidly in the field of image classification, face recognition, object detection, motion recognition, medicine, and target tracking. In our everyday life, classification makes a difference us in making choices. The require for classification emerges at whatever point a question is set in a particular bunch or course depending upon the qualities comparing to that protest. Most of the mechanical issues are classification issues. Researchers have concocted progressed classification procedures for moving forward classification precision. Each single day various pictures are created, which makes the need to classify them so that openness is less demanding and speedier. The data preparing which is done amid the classification makes a difference in picture categorization into different bunches. For example, stock market prediction, weather forecasting, bankruptcy prediction, medical diagnosis, speech recognition, character recognition, etc. The AlexNet show at the 2012 ILSVRC conference, which was optimized over the conventional Convolutional Neural Systems (CNN). It mainly includes building a deeper model structure, sampling under the overlap, ReLU activation function, and adopting the Dropout method. It is applied to image classification, which reduces the image classification Top-5 error rate from 25.8% to 16.4%. The Visual Geometry Group of Oxford University proposed the VGG model and achieved second place in the ILSVRC image classification competition. It reduces the Top-5 error rate for image classification to 7.3%. Its structure is similar to the AlexNet model but uses more convolutional layers. ResNet, short for Residual Network is a specific type of neural network that was introduced in 2015 by Kaiming He, Xiangyu Zhang, Shaoqing Ren, and Jian Sun in their paper "Deep Residual Learning for Image Recognition". Won 1st place in the ILSVRC 2015 classification competition with a top-5 error rate of 3.57% (An ensemble model). GoogLeNet is a 22-layer deep convolutional neural organize that's a variation of the Initiation Arrange, a Profound Convolutional Neural Organize created by researchers at Google. Initiation v3 may be a convolutional neural organize for helping in picture examination and protest discovery and got its begin as a module for Googlenet. It is the third version of Google's Beginning Convolutional Neural Organize, initially presented amid the ImageNet Acknowledgment Challenge.

Although the deep learning theory has achieved good application results in image classification, it has problems such as excessive gradient propagation path and over-fitting. Given this, this paper introduces the idea of sparse representation into the architecture of the deep learning network and comprehensively utilizes the sparse representation of good multidimensional data linear decomposition ability and the deep structural advantages of multilayer nonlinear mapping. It will complete the approximation of complex functions and build a deep learning model with adaptive approximation capabilities. It solves the problem of function approximation in the deep learning model. At the same time, a sparse representation classification method using the optimized kernel function is proposed to replace the classifier in the deep learning model. It will improve the image classification effect. So, this paper proposes to describe five image classification algorithms.

2. Literature Review :

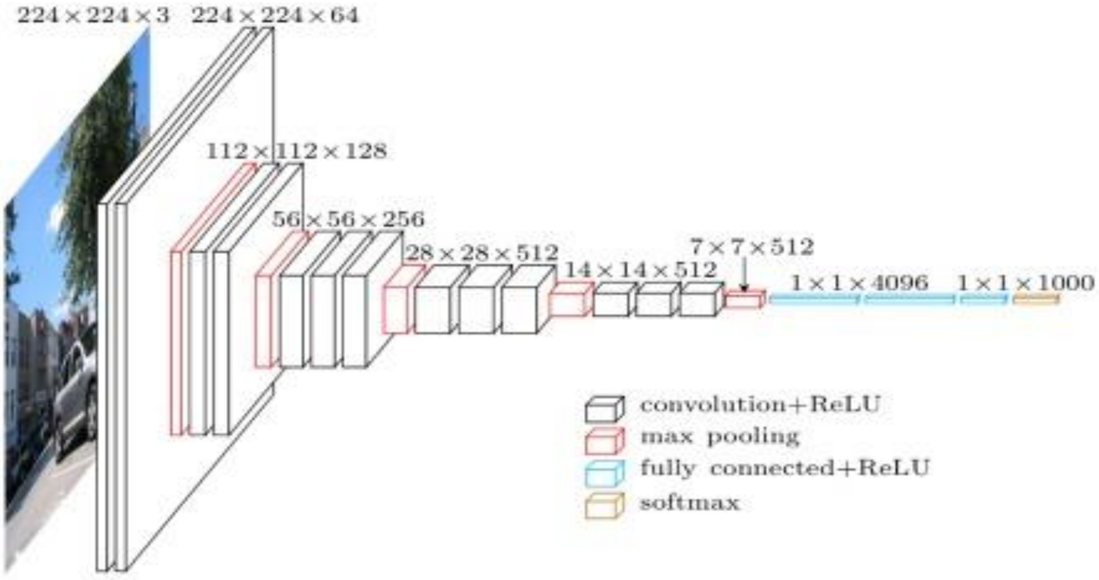
VGG16 and VGG19 Image classification

VGG is a Convolutional Neural Network (CNN) architecture that secured first and second positions in the localisation and classification tasks respectively in ImageNet challenge 2014.

All the CNNs have more or less similar architecture, stack of convolution and pooling layers at start and ending with fully connected and soft-max layers.

VGG16

Vgg-16 is a type of convolutional neural network. In general the number after the network name indicates the number of layers the architecture comprises of. The idea was to stack up layers to form a very deep convolutional neural network that would perform extremely well on tasks. Vgg-16 bagged one of the awards in ImageNet 2014 challenge. 16 indicates that the network has 16 weight layers.



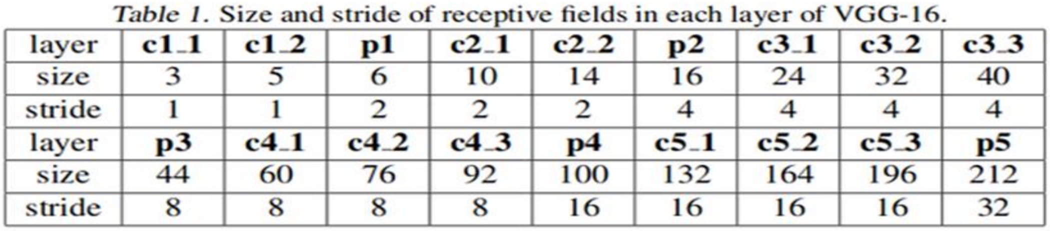
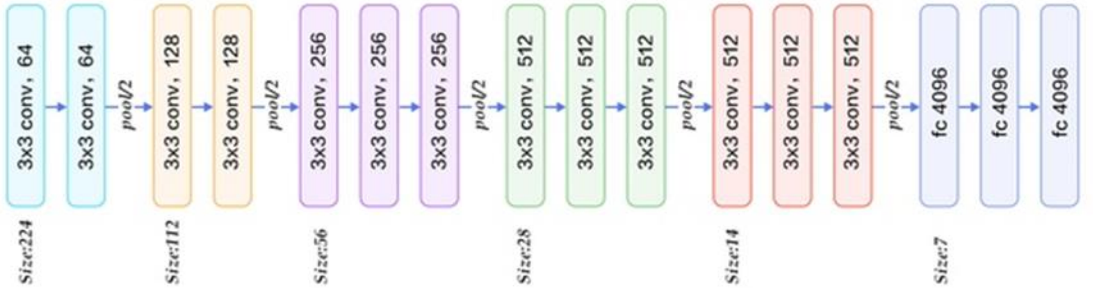


Table 1. Size and stride of receptive fields in each layer of VGG-16.

layer	c1_1	c1_2	p1	c2_1	c2_2	p2	c3_1	c3_2	c3_3
size	3	5	6	10	14	16	24	32	40
stride	1	1	2	2	2	4	4	4	4
layer	p3	c4_1	c4_2	c4_3	p4	c5_1	c5_2	c5_3	p5
size	44	60	76	92	100	132	164	196	212
stride	8	8	8	8	16	16	16	16	32



VGG 16 Application:

- Given image → find object name in the image
- It can detect any one of 1000 images
- It takes input image of size 224 * 224 * 3 (RGB image)

Built using:

- Convolutions layers (used only 3*3 size)
- Max pooling layers (used only 2*2 size)
- Fully connected layers at end
- Total 16 layers

Description of layers:

- Convolution using 64 filters
- Convolution using 64 filters + Max pooling
- Convolution using 128 filters
- Convolution using 128 filters + Max pooling
- Convolution using 256 filters

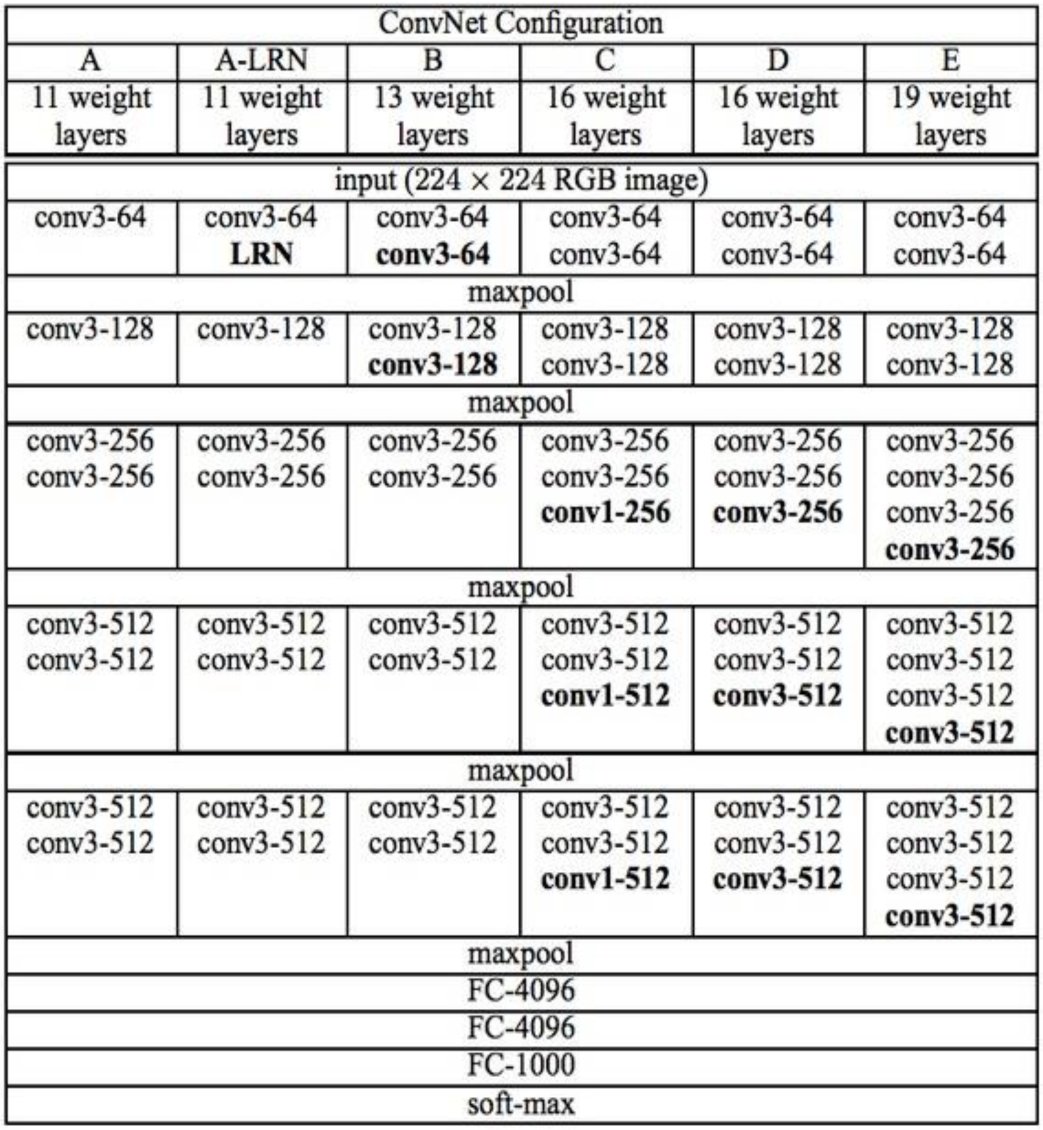
- Convolution using 256 filters
- Convolution using 256 filters + Max pooling
- Convolution using 512 filters
- Convolution using 512 filters
- Convolution using 512 filters + Max pooling
- Convolution using 512 filters
- Convolution using 512 filters
- Convolution using 512 filters + Max pooling
- Fully connected with 4096 nodes
- Fully connected with 4096 nodes
- Output layer with Softmax activation with 1000 nodes

VGG19

VGG-19 is a trained Convolutional Neural Network, from Visual Geometry Group, Department of Engineering Science, University of Oxford. The number 19 stands for the number of layers with trainable weights. 16 Convolutional layers and 3 Fully Connected layers.

The above diagram is from the original research paper. The different columns A, A-LRN to E shows the different architectures tried by the VGG team. The column E refers to VGG-19 architecture. The VGG-19 was trained on the ImageNet challenge (ILSVRC) 1000-class classification task. The network takes a (224, 224, 3) RBG image as the input.

Please look at how the layers are denoted. conv<size of the filter>-<number of such filters>. Thus, conv3-64 means 64 (3, 3) square filters. Note that all the conv layers in VGG-19 use (3, 3) filters and that the number of filters increases in powers of two (64, 128, 256, 512). In all the Conv layers, stride length used in 1 (pixel) with a padding of 1 (pixel) on each side. There are 5 sets of conv layers, 2 of them have 64 filters, next set has 2 conv layers with 128 filters, next set has 4 conv layers with 256 filters, and next 2 sets have 4 conv layers each, with 512 filters. There are max pooling layers in between each set of conv layers. max pooling layers have 2x2 filters with stride of 2 (pixels). The output of last pooling layer is flattened an fed to a

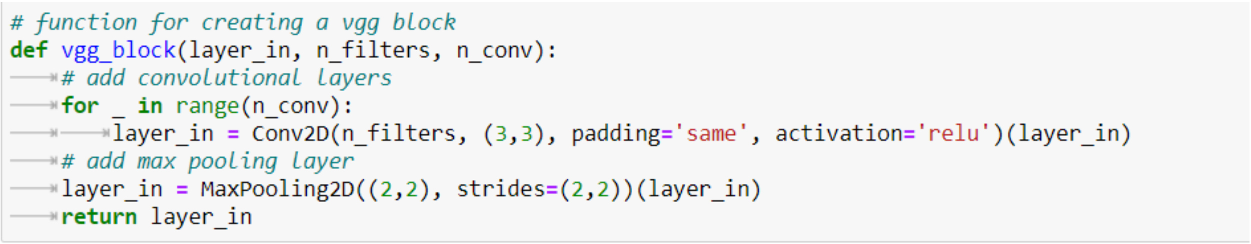


ConvNet Configuration					
A	A-LRN	B	C	D	E
11 weight layers	11 weight layers	13 weight layers	16 weight layers	16 weight layers	19 weight layers
input (224 × 224 RGB image)					
conv3-64	conv3-64 **LRN**	conv3-64 **conv3-64**	conv3-64 conv3-64	conv3-64 conv3-64	conv3-64 conv3-64
maxpool					
conv3-128	conv3-128	conv3-128 **conv3-128**	conv3-128 conv3-128	conv3-128 conv3-128	conv3-128 conv3-128
maxpool					
conv3-256 conv3-256	conv3-256 conv3-256	conv3-256 conv3-256	conv3-256 conv3-256 **conv1-256**	conv3-256 conv3-256 **conv3-256**	conv3-256 conv3-256 conv3-256 **conv3-256**
maxpool					
conv3-512 conv3-512	conv3-512 conv3-512	conv3-512 conv3-512	conv3-512 conv3-512 **conv1-512**	conv3-512 conv3-512 **conv3-512**	conv3-512 conv3-512 conv3-512 **conv3-512**
maxpool					
conv3-512 conv3-512	conv3-512 conv3-512	conv3-512 conv3-512	conv3-512 conv3-512 **conv1-512**	conv3-512 conv3-512 **conv3-512**	conv3-512 conv3-512 conv3-512 **conv3-512**
maxpool					
FC-4096					
FC-4096					
FC-1000					
soft-max					

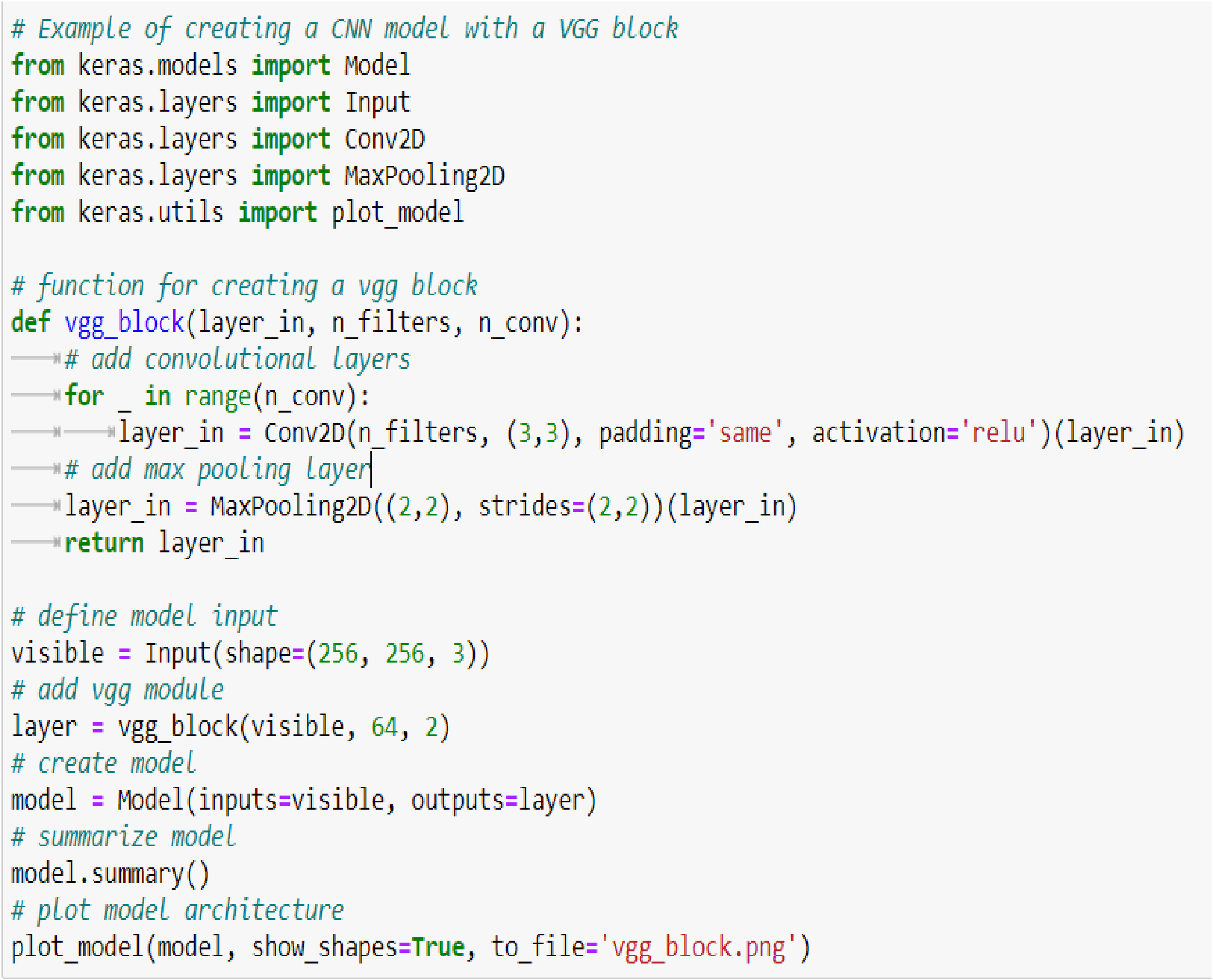
fully connected layer with 4096 neurons. The output goes to another fully connected layer with 4096 neurons, whose output is fed into another fully connected layer with 1000 neurons. All these layers are ReLU activated. Finally there is a softmax layer which uses cross entropy loss.

The layers with trainable weights are only the convolution layers and the Fully Connected layers. maxpool layer is used to reduce the size of the input image where softmax is used to make the final decision.

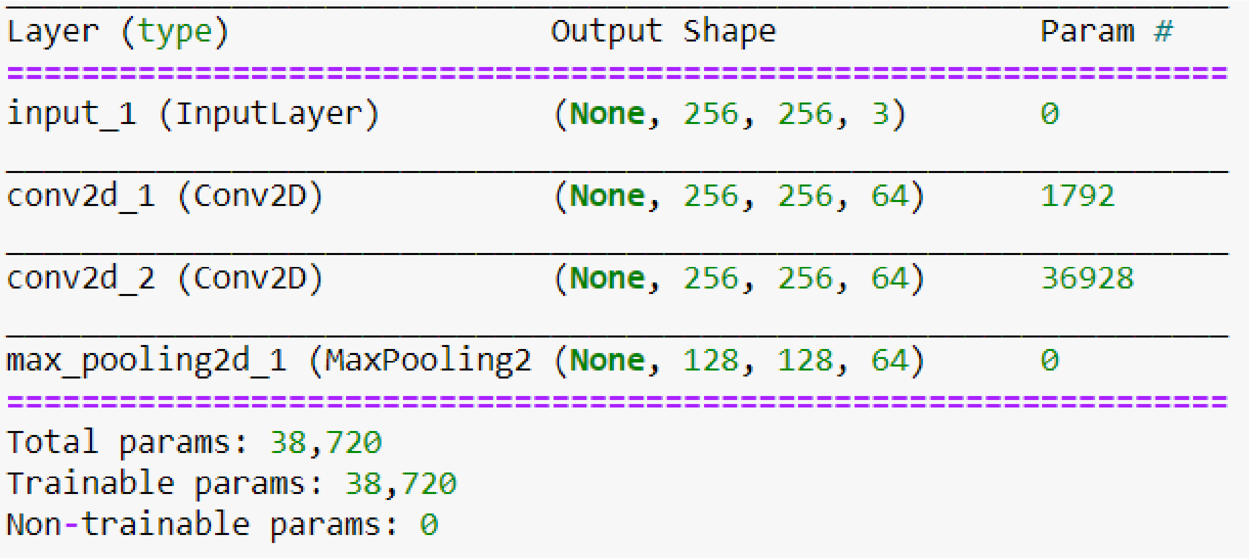
How to Implement VGG Blocks



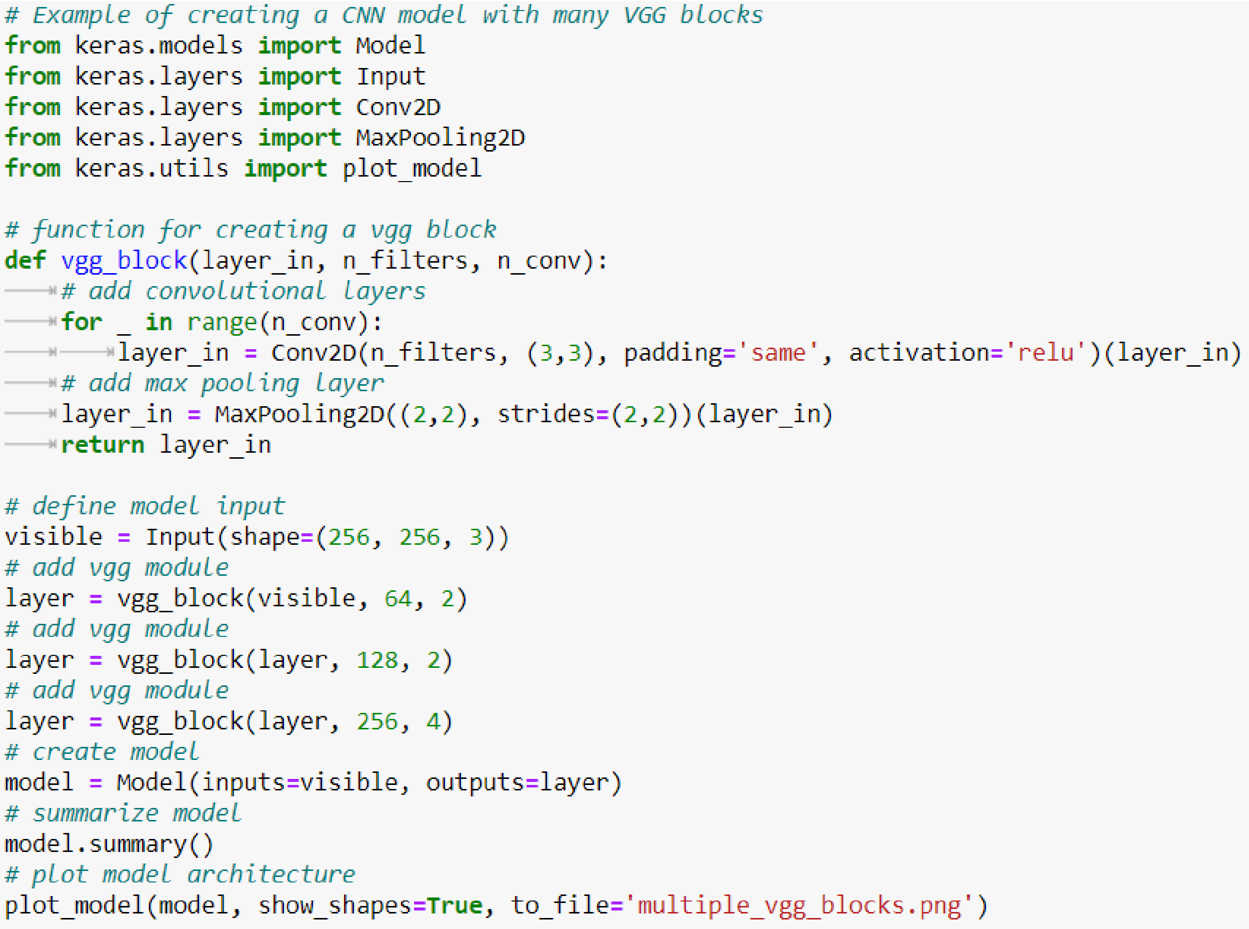
```python
# function for creating a vgg block
def vgg_block(layer_in, n_filters, n_conv):
    # add convolutional layers
    for _ in range(n_conv):
        layer_in = Conv2D(n_filters, (3,3), padding='same', activation='relu')(layer_in)
    # add max pooling layer
    layer_in = MaxPooling2D((2,2), strides=(2,2))(layer_in)
    return layer_in
```



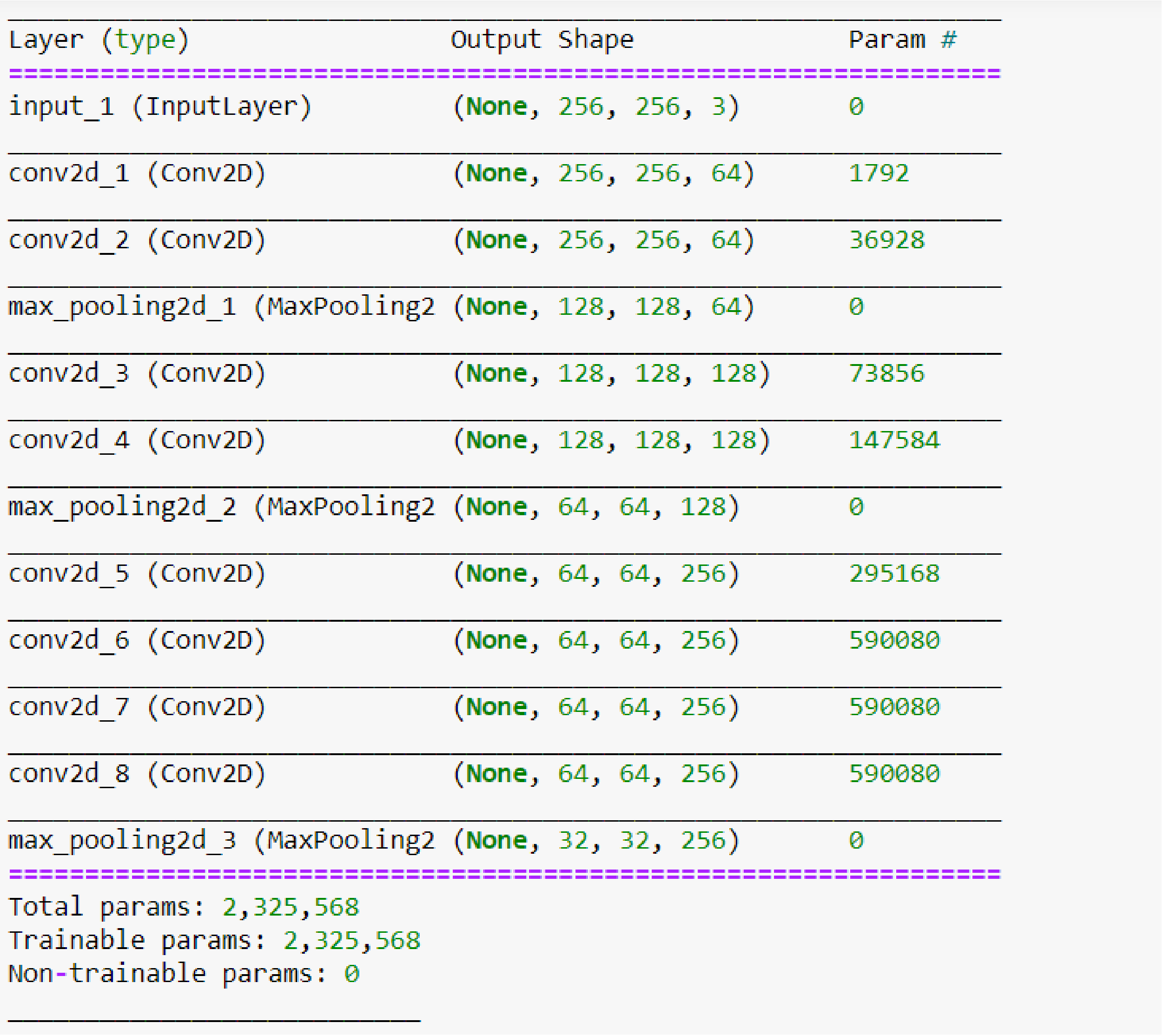
```python
# Example of creating a CNN model with a VGG block
from keras.models import Model
from keras.layers import Input
from keras.layers import Conv2D
from keras.layers import MaxPooling2D
from keras.utils import plot_model

# function for creating a vgg block
def vgg_block(layer_in, n_filters, n_conv):
    # add convolutional layers
    for _ in range(n_conv):
        layer_in = Conv2D(n_filters, (3,3), padding='same', activation='relu')(layer_in)
    # add max pooling layer
    layer_in = MaxPooling2D((2,2), strides=(2,2))(layer_in)
    return layer_in

# define model input
visible = Input(shape=(256, 256, 3))
# add vgg module
layer = vgg_block(visible, 64, 2)
# create model
model = Model(inputs=visible, outputs=layer)
# summarize model
model.summary()
# plot model architecture
plot_model(model, show_shapes=True, to_file='vgg_block.png')
```



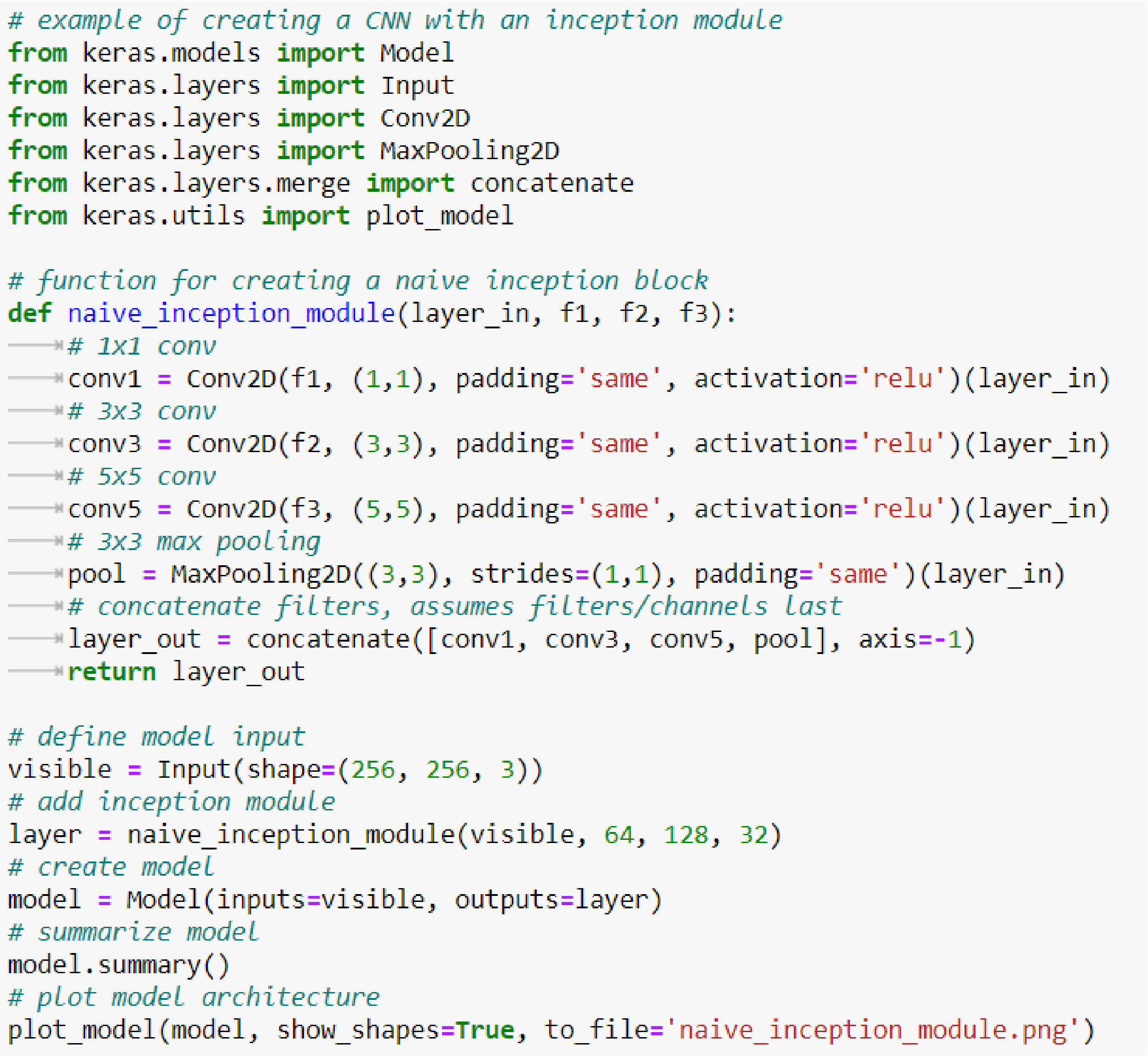
```
_________________________________________________________________
Layer (type)                 Output Shape              Param #
=================================================================
input_1 (InputLayer)         (None, 256, 256, 3)       0
_________________________________________________________________
conv2d_1 (Conv2D)            (None, 256, 256, 64)      1792
_________________________________________________________________
conv2d_2 (Conv2D)            (None, 256, 256, 64)      36928
_________________________________________________________________
max_pooling2d_1 (MaxPooling2 (None, 128, 128, 64)      0
=================================================================
Total params: 38,720
Trainable params: 38,720
Non-trainable params: 0
```



```python
# Example of creating a CNN model with many VGG blocks
from keras.models import Model
from keras.layers import Input
from keras.layers import Conv2D
from keras.layers import MaxPooling2D
from keras.utils import plot_model

# function for creating a vgg block
def vgg_block(layer_in, n_filters, n_conv):
    # add convolutional layers
    for _ in range(n_conv):
        layer_in = Conv2D(n_filters, (3,3), padding='same', activation='relu')(layer_in)
    # add max pooling layer
    layer_in = MaxPooling2D((2,2), strides=(2,2))(layer_in)
    return layer_in

# define model input
visible = Input(shape=(256, 256, 3))
# add vgg module
layer = vgg_block(visible, 64, 2)
# add vgg module
layer = vgg_block(layer, 128, 2)
# add vgg module
layer = vgg_block(layer, 256, 4)
# create model
model = Model(inputs=visible, outputs=layer)
# summarize model
model.summary()
# plot model architecture
plot_model(model, show_shapes=True, to_file='multiple_vgg_blocks.png')
```



```
_________________________________________________________________
Layer (type)                 Output Shape              Param #
=================================================================
input_1 (InputLayer)         (None, 256, 256, 3)       0
_________________________________________________________________
conv2d_1 (Conv2D)            (None, 256, 256, 64)      1792
_________________________________________________________________
conv2d_2 (Conv2D)            (None, 256, 256, 64)      36928
_________________________________________________________________
max_pooling2d_1 (MaxPooling2 (None, 128, 128, 64)      0
_________________________________________________________________
conv2d_3 (Conv2D)            (None, 128, 128, 128)     73856
_________________________________________________________________
conv2d_4 (Conv2D)            (None, 128, 128, 128)     147584
_________________________________________________________________
max_pooling2d_2 (MaxPooling2 (None, 64, 64, 128)       0
_________________________________________________________________
conv2d_5 (Conv2D)            (None, 64, 64, 256)       295168
_________________________________________________________________
conv2d_6 (Conv2D)            (None, 64, 64, 256)       590080
_________________________________________________________________
conv2d_7 (Conv2D)            (None, 64, 64, 256)       590080
_________________________________________________________________
conv2d_8 (Conv2D)            (None, 64, 64, 256)       590080
_________________________________________________________________
max_pooling2d_3 (MaxPooling2 (None, 32, 32, 256)       0
=================================================================
Total params: 2,325,568
Trainable params: 2,325,568
Non-trainable params: 0
___________________________
```



```
# example of creating a CNN with an inception module
from keras.models import Model
from keras.layers import Input
from keras.layers import Conv2D
from keras.layers import MaxPooling2D
from keras.layers.merge import concatenate
from keras.utils import plot_model

# function for creating a naive inception block
def naive_inception_module(layer_in, f1, f2, f3):
    # 1x1 conv
    conv1 = Conv2D(f1, (1,1), padding='same', activation='relu')(layer_in)
    # 3x3 conv
    conv3 = Conv2D(f2, (3,3), padding='same', activation='relu')(layer_in)
    # 5x5 conv
    conv5 = Conv2D(f3, (5,5), padding='same', activation='relu')(layer_in)
    # 3x3 max pooling
    pool = MaxPooling2D((3,3), strides=(1,1), padding='same')(layer_in)
    # concatenate filters, assumes filters/channels last
    layer_out = concatenate([conv1, conv3, conv5, pool], axis=-1)
    return layer_out

# define model input
visible = Input(shape=(256, 256, 3))
# add inception module
layer = naive_inception_module(visible, 64, 128, 32)
# create model
model = Model(inputs=visible, outputs=layer)
# summarize model
model.summary()
# plot model architecture
plot_model(model, show_shapes=True, to_file='naive_inception_module.png')
```

RESNET 50

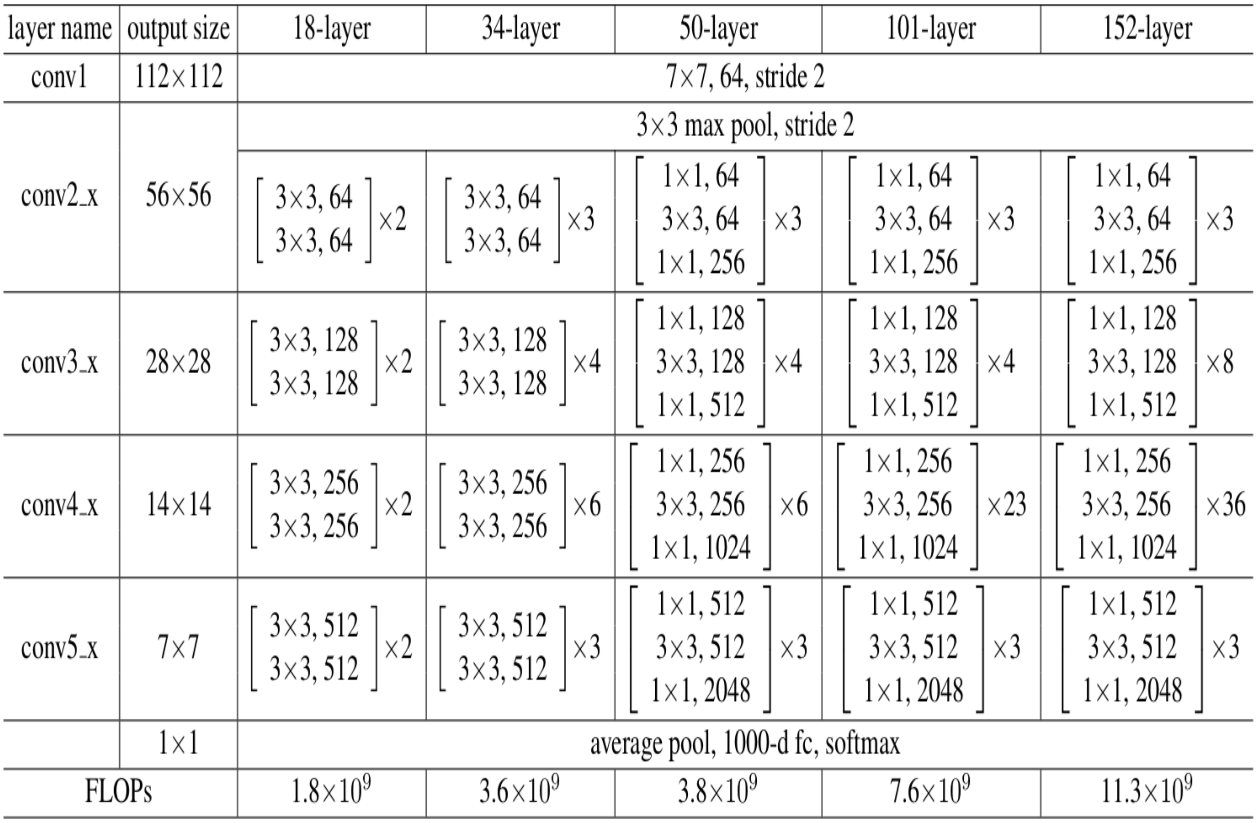
In 2012 at the LSVRC2012 classification contest AlexNet won the the first price, After that ResNet was the most interesting thing that happened to the computer vision and the deep learning world.

Because of the framework that ResNets presented it was made possible to train ultra deep neural networks and by that i mean that i network can contain hundreds or thousands of layers and still achieve great performance.

ResNet50 is a variant of ResNet model which has 48 Convolution layers along with 1 MaxPool and 1 Average Pool layer. It has 3.8 x 10^9 Floating points operations. It

is a widely used ResNet model and we have explored ResNet50 architecture in depth.
We start with some background information, comparison with other models and then, dive directly into ResNet50 architecture

ResNet50 Architecture



layer name	output size	18-layer	34-layer	50-layer	101-layer	152-layer
conv1	112×112	7×7, 64, stride 2				
conv2_x	56×56	3×3 max pool, stride 2				
		$\begin{bmatrix}3\times3, 64\\3\times3, 64\end{bmatrix}\times2$	$\begin{bmatrix}3\times3, 64\\3\times3, 64\end{bmatrix}\times3$	$\begin{bmatrix}1\times1, 64\\3\times3, 64\\1\times1, 256\end{bmatrix}\times3$	$\begin{bmatrix}1\times1, 64\\3\times3, 64\\1\times1, 256\end{bmatrix}\times3$	$\begin{bmatrix}1\times1, 64\\3\times3, 64\\1\times1, 256\end{bmatrix}\times3$
conv3_x	28×28	$\begin{bmatrix}3\times3, 128\\3\times3, 128\end{bmatrix}\times2$	$\begin{bmatrix}3\times3, 128\\3\times3, 128\end{bmatrix}\times4$	$\begin{bmatrix}1\times1, 128\\3\times3, 128\\1\times1, 512\end{bmatrix}\times4$	$\begin{bmatrix}1\times1, 128\\3\times3, 128\\1\times1, 512\end{bmatrix}\times4$	$\begin{bmatrix}1\times1, 128\\3\times3, 128\\1\times1, 512\end{bmatrix}\times8$
conv4_x	14×14	$\begin{bmatrix}3\times3, 256\\3\times3, 256\end{bmatrix}\times2$	$\begin{bmatrix}3\times3, 256\\3\times3, 256\end{bmatrix}\times6$	$\begin{bmatrix}1\times1, 256\\3\times3, 256\\1\times1, 1024\end{bmatrix}\times6$	$\begin{bmatrix}1\times1, 256\\3\times3, 256\\1\times1, 1024\end{bmatrix}\times23$	$\begin{bmatrix}1\times1, 256\\3\times3, 256\\1\times1, 1024\end{bmatrix}\times36$
conv5_x	7×7	$\begin{bmatrix}3\times3, 512\\3\times3, 512\end{bmatrix}\times2$	$\begin{bmatrix}3\times3, 512\\3\times3, 512\end{bmatrix}\times3$	$\begin{bmatrix}1\times1, 512\\3\times3, 512\\1\times1, 2048\end{bmatrix}\times3$	$\begin{bmatrix}1\times1, 512\\3\times3, 512\\1\times1, 2048\end{bmatrix}\times3$	$\begin{bmatrix}1\times1, 512\\3\times3, 512\\1\times1, 2048\end{bmatrix}\times3$
	1×1	average pool, 1000-d fc, softmax				
FLOPs		1.8×10^9	3.6×10^9	3.8×10^9	7.6×10^9	11.3×10^9

Table 1

Now we are going to discuss about Resnet 50 and also the architecture for the above talked 18 and 34 layer ResNet is also given residual mapping and not shown for simplicity.
There was a small change that was made for the ResNet 50 and above that before this the shortcut connections skipped two layers but now they skip three layers and also there was 1 * 1 convolution layers added that we are going to see in detail with the ResNet 50 Architecture.
So as we can see in the table 1 the resnet 50 architecture contains the following element:

- A convoultion with a kernel size of 7 * 7 and 64 different kernels all with a stride of size 2 giving us 1 layer.

- Next we see max pooling with also a stride size of 2.

- In the next convolution there is a 1 * 1,64 kernel following this a 3 * 3,64 kernel and at last a 1 * 1,256 kernel, These three layers are repeated in total 3 time so giving us 9 layers in this step.

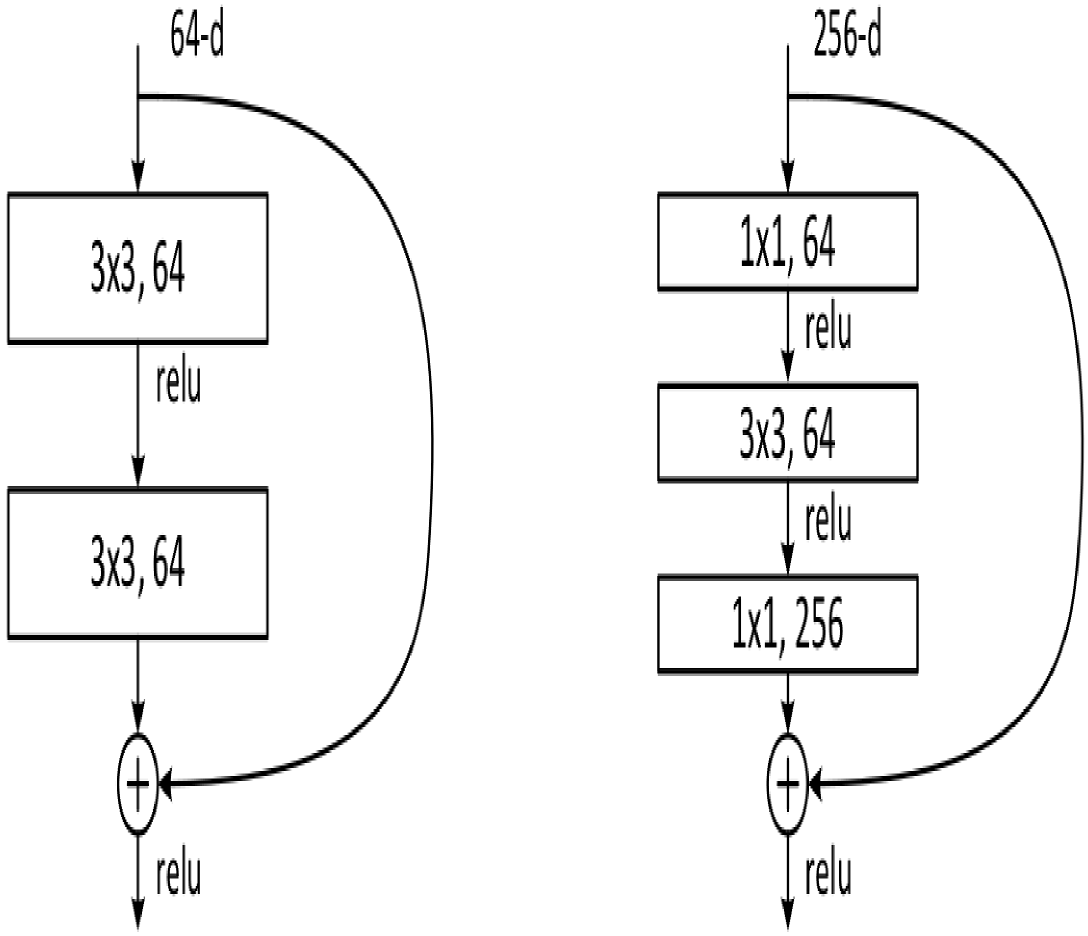
- Next we see kernel of 1 * 1,128 after that a kernel of 3 * 3,128 and at last a kernel of 1 * 1,512 this step was repeated 4 time so giving us 12 layers in this step.

- After that there is a kernal of 1 * 1,256 and two more kernels with 3 * 3,256 and 1 * 1,1024 and this is repeated 6 time giving us a total of 18 layers.

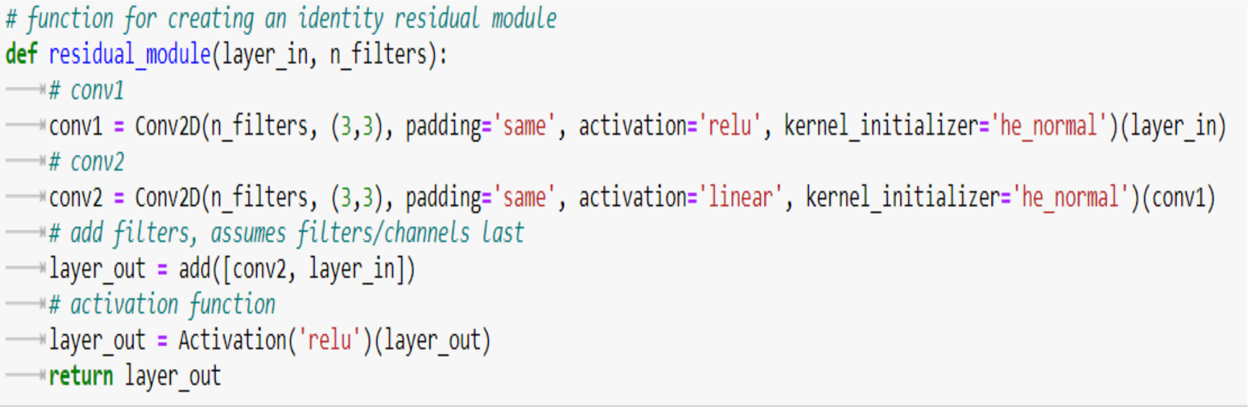
- And then again a 1 * 1,512 kernel with two more of 3 * 3,512 and 1 * 1,2048 and this was repeated 3 times giving us a total of 9 layers.

- After that we do a average pool and end it with a fully connected layer containing 1000 nodes and at the end a softmax function so this gives us 1 layer.

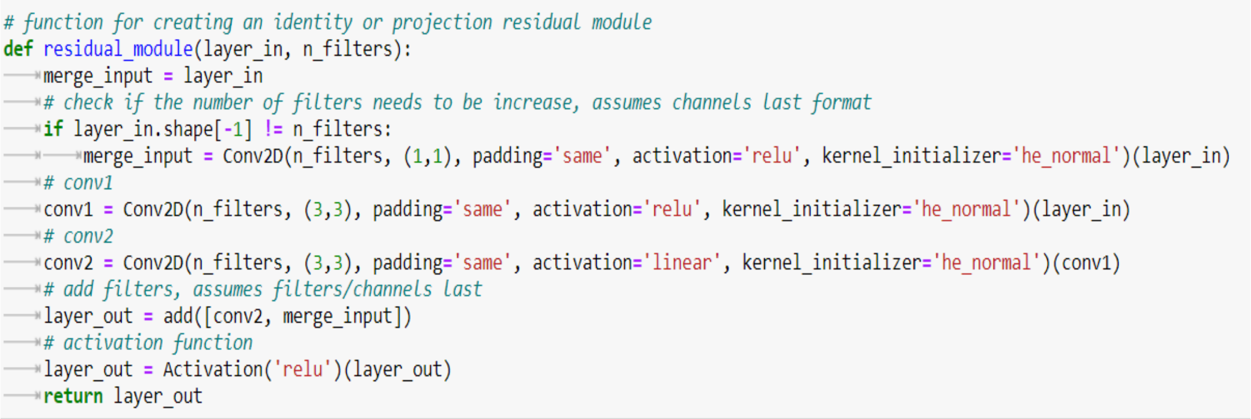
- We don't actually count the activation functions and the max/ average pooling layers. So totaling this it gives us a 1 + 9 + 12 + 18 + 9 + 1 = 50 layers Deep Convolutional network.


64-d
3x3, 64
relu
3x3, 64
relu
256-d
1x1, 64
relu
3x3, 64
relu
1x1, 256
relu

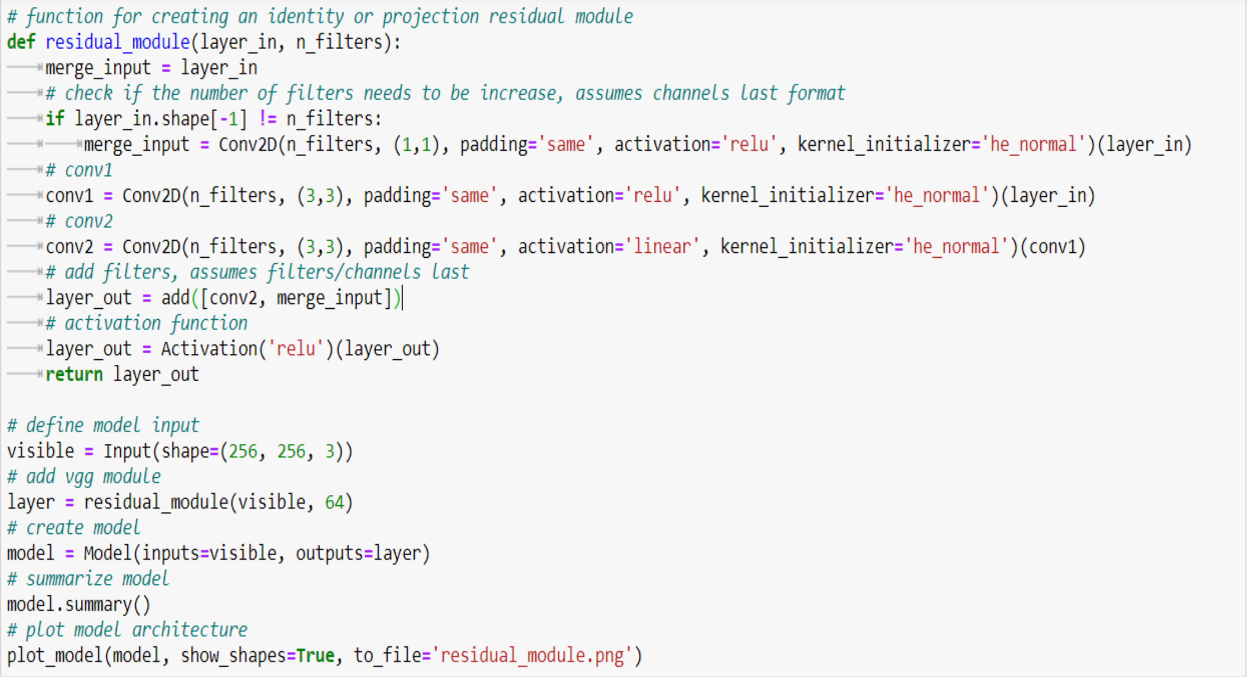
How to Implement the ResNet Module



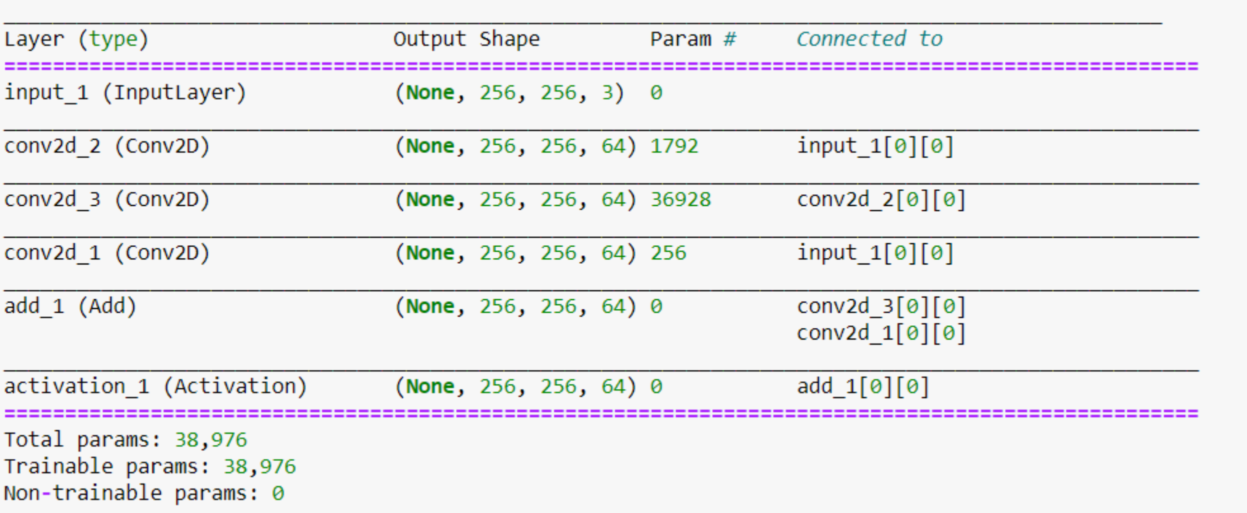
```python
# function for creating an identity residual module
def residual_module(layer_in, n_filters):
    # conv1
    conv1 = Conv2D(n_filters, (3,3), padding='same', activation='relu', kernel_initializer='he_normal')(layer_in)
    # conv2
    conv2 = Conv2D(n_filters, (3,3), padding='same', activation='linear', kernel_initializer='he_normal')(conv1)
    # add filters, assumes filters/channels last
    layer_out = add([conv2, layer_in])
    # activation function
    layer_out = Activation('relu')(layer_out)
    return layer_out
```



```python
# function for creating an identity or projection residual module
def residual_module(layer_in, n_filters):
    merge_input = layer_in
    # check if the number of filters needs to be increase, assumes channels last format
    if layer_in.shape[-1] != n_filters:
        merge_input = Conv2D(n_filters, (1,1), padding='same', activation='relu', kernel_initializer='he_normal')(layer_in)
    # conv1
    conv1 = Conv2D(n_filters, (3,3), padding='same', activation='relu', kernel_initializer='he_normal')(layer_in)
    # conv2
    conv2 = Conv2D(n_filters, (3,3), padding='same', activation='linear', kernel_initializer='he_normal')(conv1)
    # add filters, assumes filters/channels last
    layer_out = add([conv2, merge_input])
    # activation function
    layer_out = Activation('relu')(layer_out)
    return layer_out
```

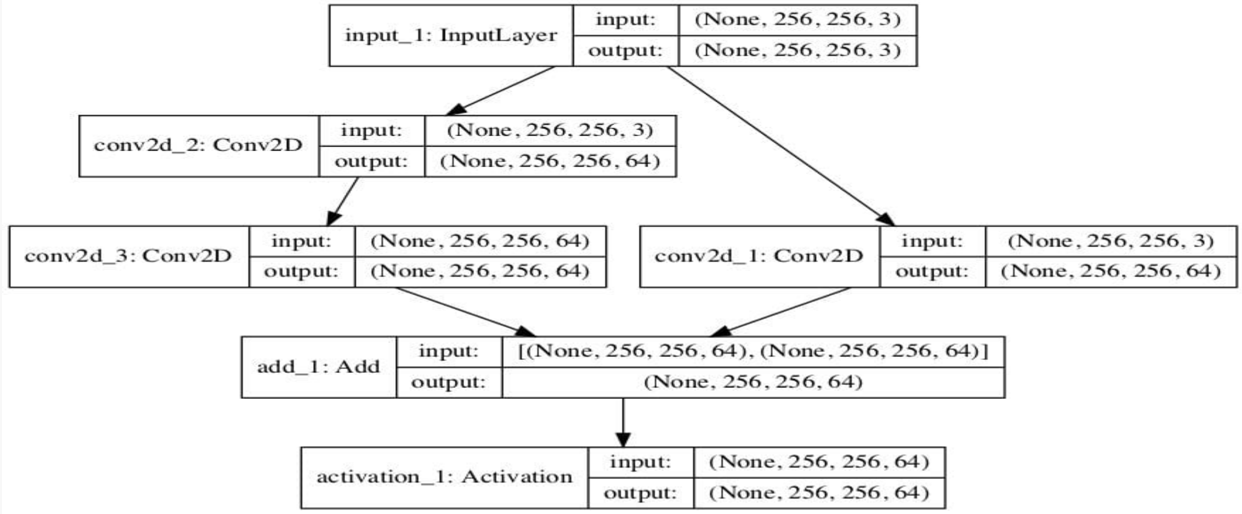


```python
# function for creating an identity or projection residual module
def residual_module(layer_in, n_filters):
    merge_input = layer_in
    # check if the number of filters needs to be increase, assumes channels last format
    if layer_in.shape[-1] != n_filters:
        merge_input = Conv2D(n_filters, (1,1), padding='same', activation='relu', kernel_initializer='he_normal')(layer_in)
    # conv1
    conv1 = Conv2D(n_filters, (3,3), padding='same', activation='relu', kernel_initializer='he_normal')(layer_in)
    # conv2
    conv2 = Conv2D(n_filters, (3,3), padding='same', activation='linear', kernel_initializer='he_normal')(conv1)
    # add filters, assumes filters/channels last
    layer_out = add([conv2, merge_input])
    # activation function
    layer_out = Activation('relu')(layer_out)
    return layer_out

# define model input
visible = Input(shape=(256, 256, 3))
# add vgg module
layer = residual_module(visible, 64)
# create model
model = Model(inputs=visible, outputs=layer)
# summarize model
model.summary()
# plot model architecture
plot_model(model, show_shapes=True, to_file='residual_module.png')
```



```
________________________________________________________________________________________________
Layer (type)                    Output Shape         Param #     Connected to
================================================================================================
input_1 (InputLayer)            (None, 256, 256, 3)  0
________________________________________________________________________________________________
conv2d_2 (Conv2D)               (None, 256, 256, 64) 1792        input_1[0][0]
________________________________________________________________________________________________
conv2d_3 (Conv2D)               (None, 256, 256, 64) 36928       conv2d_2[0][0]
________________________________________________________________________________________________
conv2d_1 (Conv2D)               (None, 256, 256, 64) 256         input_1[0][0]
________________________________________________________________________________________________
add_1 (Add)                     (None, 256, 256, 64) 0           conv2d_3[0][0]
                                                                 conv2d_1[0][0]
________________________________________________________________________________________________
activation_1 (Activation)       (None, 256, 256, 64) 0           add_1[0][0]
================================================================================================
Total params: 38,976
Trainable params: 38,976
Non-trainable params: 0
```

We can see the module with the inflation of the number of filters in the input and the addition of the two elements at the end of the module.



Plot of Convolutional Neural Network Architecture With an Residual Module

Inception Module

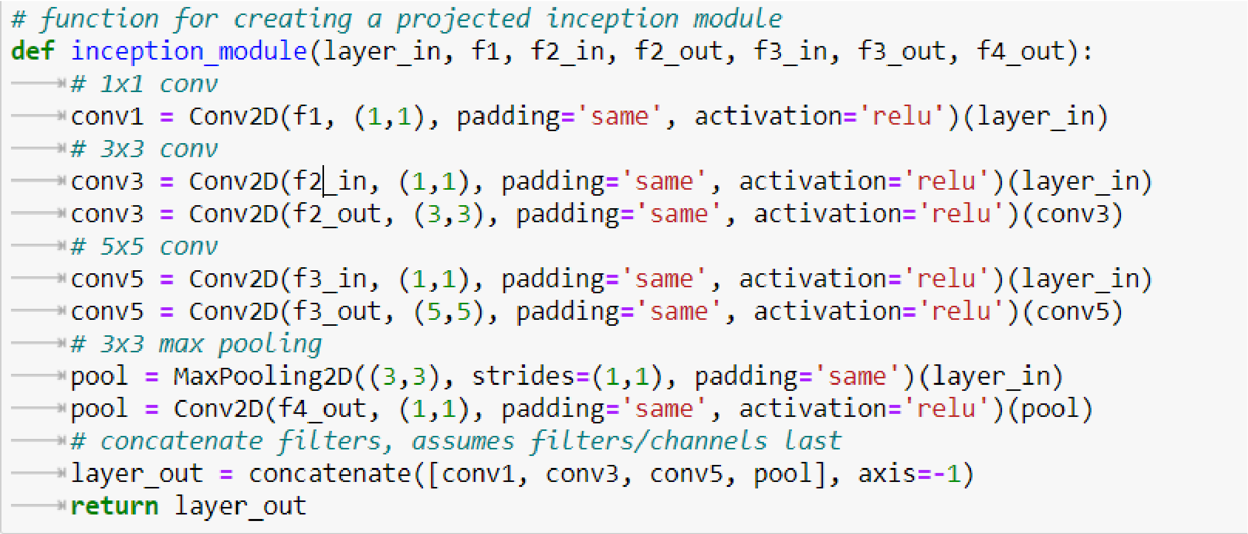
The inception module was described and used in the GoogLeNet model in the 2015 paper by Christian Szegedy, et al. titled “Going Deeper with Convolutions.”

Like the VGG model, the GoogLeNet model achieved top results in the 2014 version of the ILSVRC challenge.

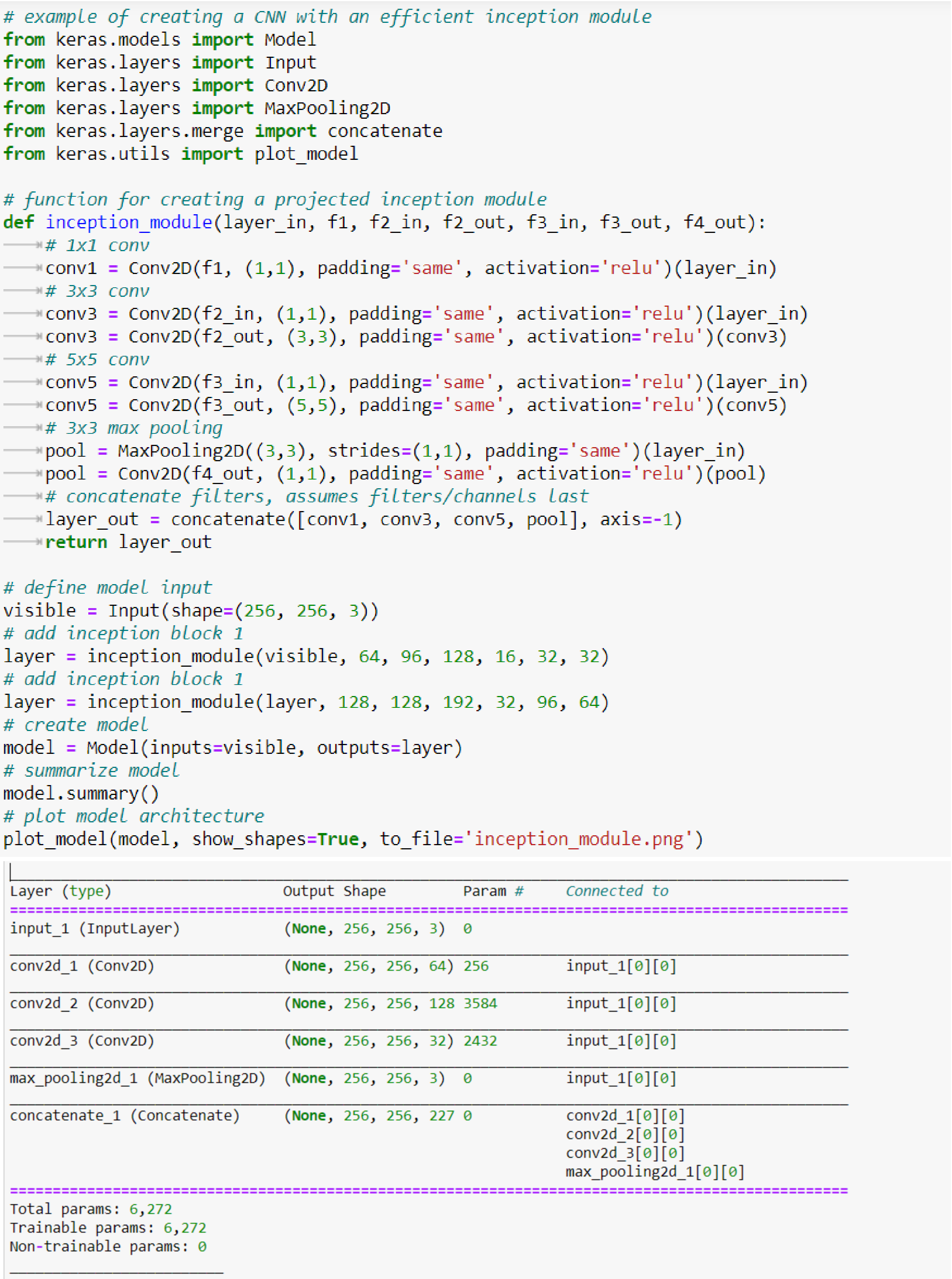
The key innovation on the inception model is called the inception module. This is a block of parallel convolutional layers with different sized filters (e.g. 1×1, 3×3, 5×5) and a and 3×3 max pooling layer, the results of which are then concatenated.
This is a very simple and powerful architectural unit that allows the model to learn not only parallel filters of the same size, but parallel filters of differing sizes, allowing learning at multiple scales.

We can implement an inception module directly using the Keras functional API. The function below will create a single inception module with a fixed number of filters for each of the parallel convolutional layers. From the GoogLeNet architecture described in the paper, it does not appear to use a systematic sizing of filters for parallel convolutional layers as the model is highly optimized. As such, we can parameterize the module definition so that we can specify the number of filters to use in each of the 1×1, 3×3, and 5×5 filters.

How to Implement the Inception Module



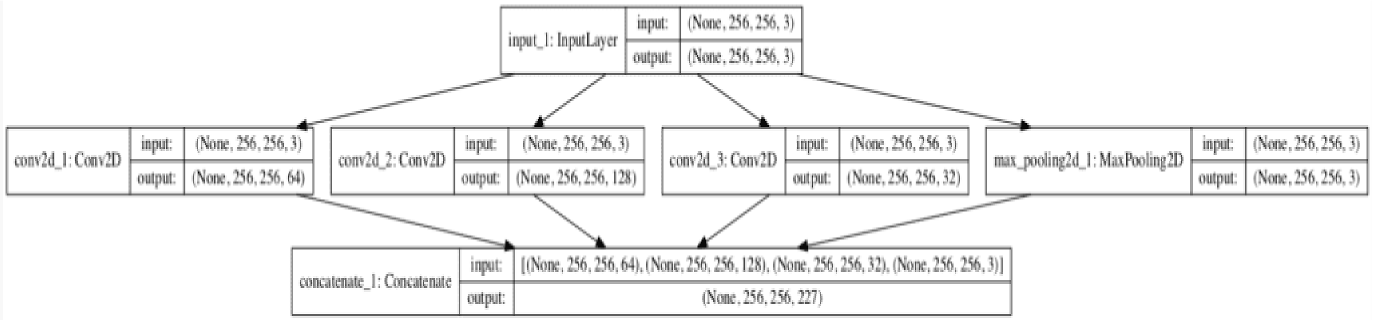
```
# function for creating a projected inception module
def inception_module(layer_in, f1, f2_in, f2_out, f3_in, f3_out, f4_out):
    # 1x1 conv
    conv1 = Conv2D(f1, (1,1), padding='same', activation='relu')(layer_in)
    # 3x3 conv
    conv3 = Conv2D(f2_in, (1,1), padding='same', activation='relu')(layer_in)
    conv3 = Conv2D(f2_out, (3,3), padding='same', activation='relu')(conv3)
    # 5x5 conv
    conv5 = Conv2D(f3_in, (1,1), padding='same', activation='relu')(layer_in)
    conv5 = Conv2D(f3_out, (5,5), padding='same', activation='relu')(conv5)
    # 3x3 max pooling
    pool = MaxPooling2D((3,3), strides=(1,1), padding='same')(layer_in)
    pool = Conv2D(f4_out, (1,1), padding='same', activation='relu')(pool)
    # concatenate filters, assumes filters/channels last
    layer_out = concatenate([conv1, conv3, conv5, pool], axis=-1)
    return layer_out
```



```python
# example of creating a CNN with an efficient inception module
from keras.models import Model
from keras.layers import Input
from keras.layers import Conv2D
from keras.layers import MaxPooling2D
from keras.layers.merge import concatenate
from keras.utils import plot_model

# function for creating a projected inception module
def inception_module(layer_in, f1, f2_in, f2_out, f3_in, f3_out, f4_out):
	# 1x1 conv
	conv1 = Conv2D(f1, (1,1), padding='same', activation='relu')(layer_in)
	# 3x3 conv
	conv3 = Conv2D(f2_in, (1,1), padding='same', activation='relu')(layer_in)
	conv3 = Conv2D(f2_out, (3,3), padding='same', activation='relu')(conv3)
	# 5x5 conv
	conv5 = Conv2D(f3_in, (1,1), padding='same', activation='relu')(layer_in)
	conv5 = Conv2D(f3_out, (5,5), padding='same', activation='relu')(conv5)
	# 3x3 max pooling
	pool = MaxPooling2D((3,3), strides=(1,1), padding='same')(layer_in)
	pool = Conv2D(f4_out, (1,1), padding='same', activation='relu')(pool)
	# concatenate filters, assumes filters/channels last
	layer_out = concatenate([conv1, conv3, conv5, pool], axis=-1)
	return layer_out

# define model input
visible = Input(shape=(256, 256, 3))
# add inception block 1
layer = inception_module(visible, 64, 96, 128, 16, 32, 32)
# add inception block 1
layer = inception_module(layer, 128, 128, 192, 32, 96, 64)
# create model
model = Model(inputs=visible, outputs=layer)
# summarize model
model.summary()
# plot model architecture
plot_model(model, show_shapes=True, to_file='inception_module.png')
```

```
__________________________________________________________________________________________________
Layer (type)                    Output Shape         Param #     Connected to
==================================================================================================
input_1 (InputLayer)            (None, 256, 256, 3)  0
__________________________________________________________________________________________________
conv2d_1 (Conv2D)               (None, 256, 256, 64) 256         input_1[0][0]
__________________________________________________________________________________________________
conv2d_2 (Conv2D)               (None, 256, 256, 128 3584        input_1[0][0]
__________________________________________________________________________________________________
conv2d_3 (Conv2D)               (None, 256, 256, 32) 2432        input_1[0][0]
__________________________________________________________________________________________________
max_pooling2d_1 (MaxPooling2D)  (None, 256, 256, 3)  0           input_1[0][0]
__________________________________________________________________________________________________
concatenate_1 (Concatenate)     (None, 256, 256, 227 0           conv2d_1[0][0]
                                                                 conv2d_2[0][0]
                                                                 conv2d_3[0][0]
                                                                 max_pooling2d_1[0][0]
==================================================================================================
Total params: 6,272
Trainable params: 6,272
Non-trainable params: 0
_________________________
```

A plot of the model architecture is also created that helps to clearly see the parallel structure of the module as well as the matching shapes of the output of each element of the module that allows their direct concatenation by the third dimension (filters or channels).


input_1: InputLayer
input: (None, 256, 256, 3)
output: (None, 256, 256, 3)
conv2d_1: Conv2D
input: (None, 256, 256, 3)
output: (None, 256, 256, 64)
conv2d_2: Conv2D
input: (None, 256, 256, 3)
output: (None, 256, 256, 128)
conv2d_3: Conv2D
input: (None, 256, 256, 3)
output: (None, 256, 256, 32)
max_pooling2d_1: MaxPooling2D
input: (None, 256, 256, 3)
output: (None, 256, 256, 3)
concatenate_1: Concatenate
input: [(None, 256, 256, 64), (None, 256, 256, 128), (None, 256, 256, 32), (None, 256, 256, 3)]
output: (None, 256, 256, 227)

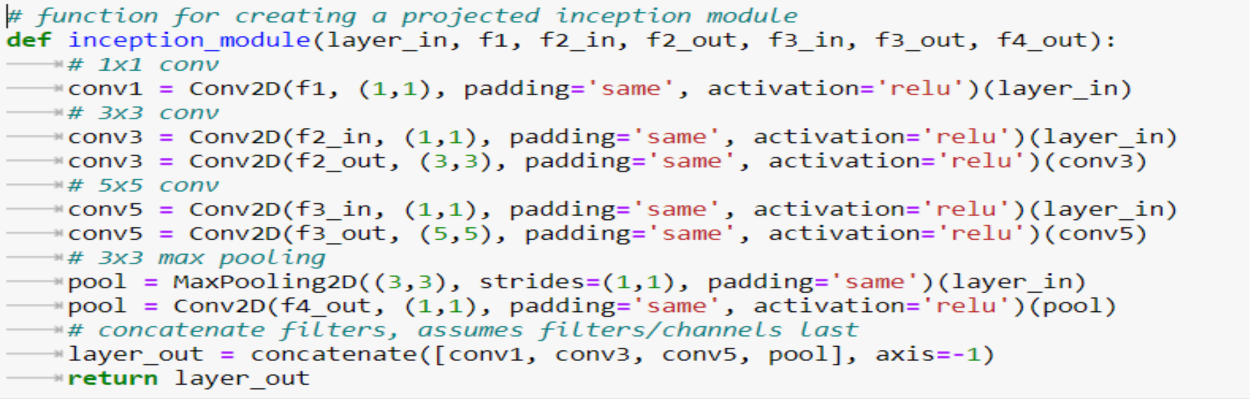
Plot of Convolutional Neural Network Architecture With a Naive Inception Module

The version of the inception module that we have implemented is called the naive inception module.

A modification to the module was made in order to reduce the amount of computation required. Specifically, 1×1 convolutional layers were added to reduce the number of filters before the 3×3 and 5×5 convolutional layers, and to increase the number of filters after the pooling layer.

If you intend to use many inception modules in your model, you may require this computational performance-based modification.

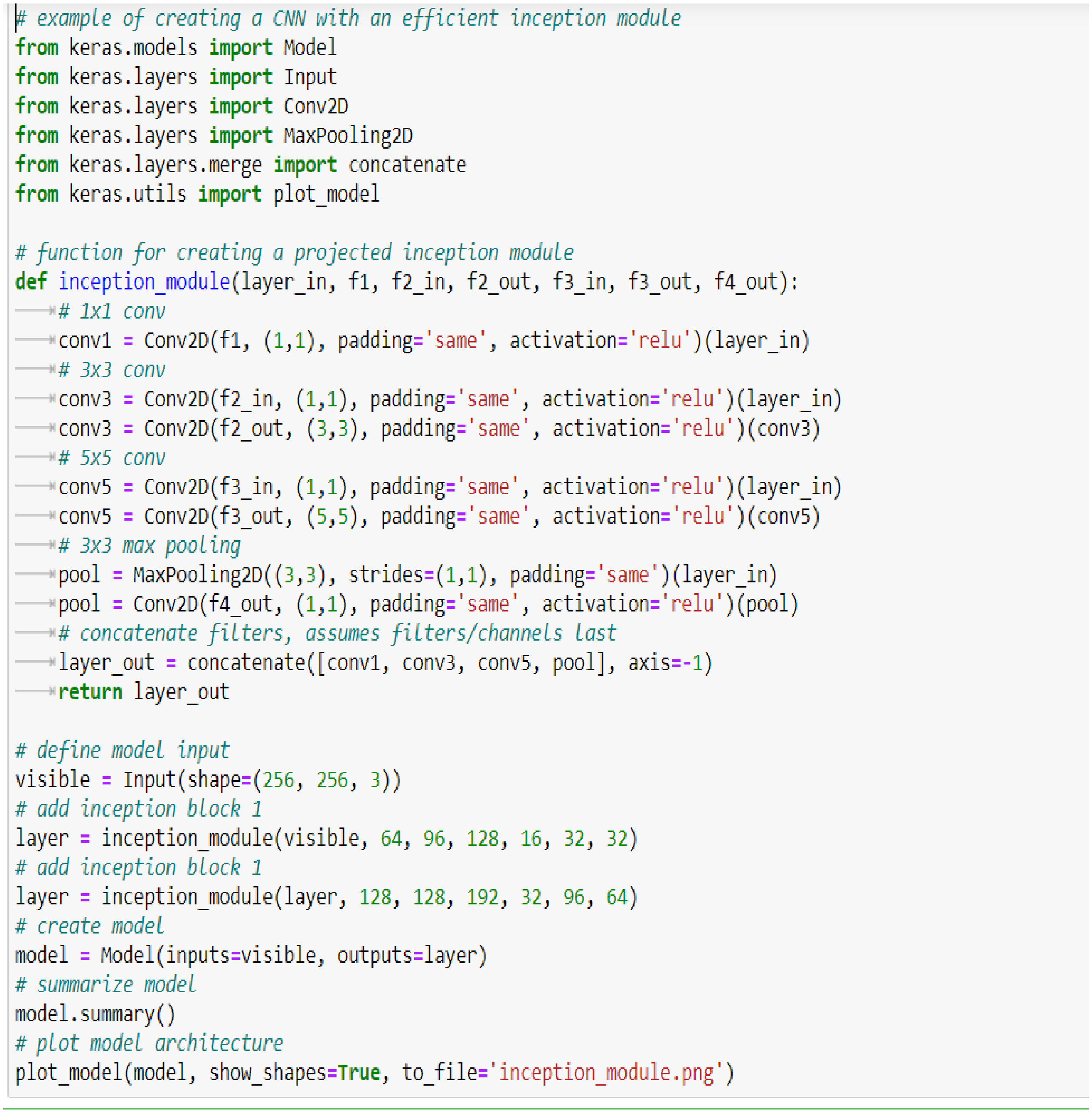
The function below implements this optimization improvement with parameterization so that you can control the amount of reduction in the number of filters prior to the 3×3 and 5×5 convolutional layers and the number of increased filters after max pooling.



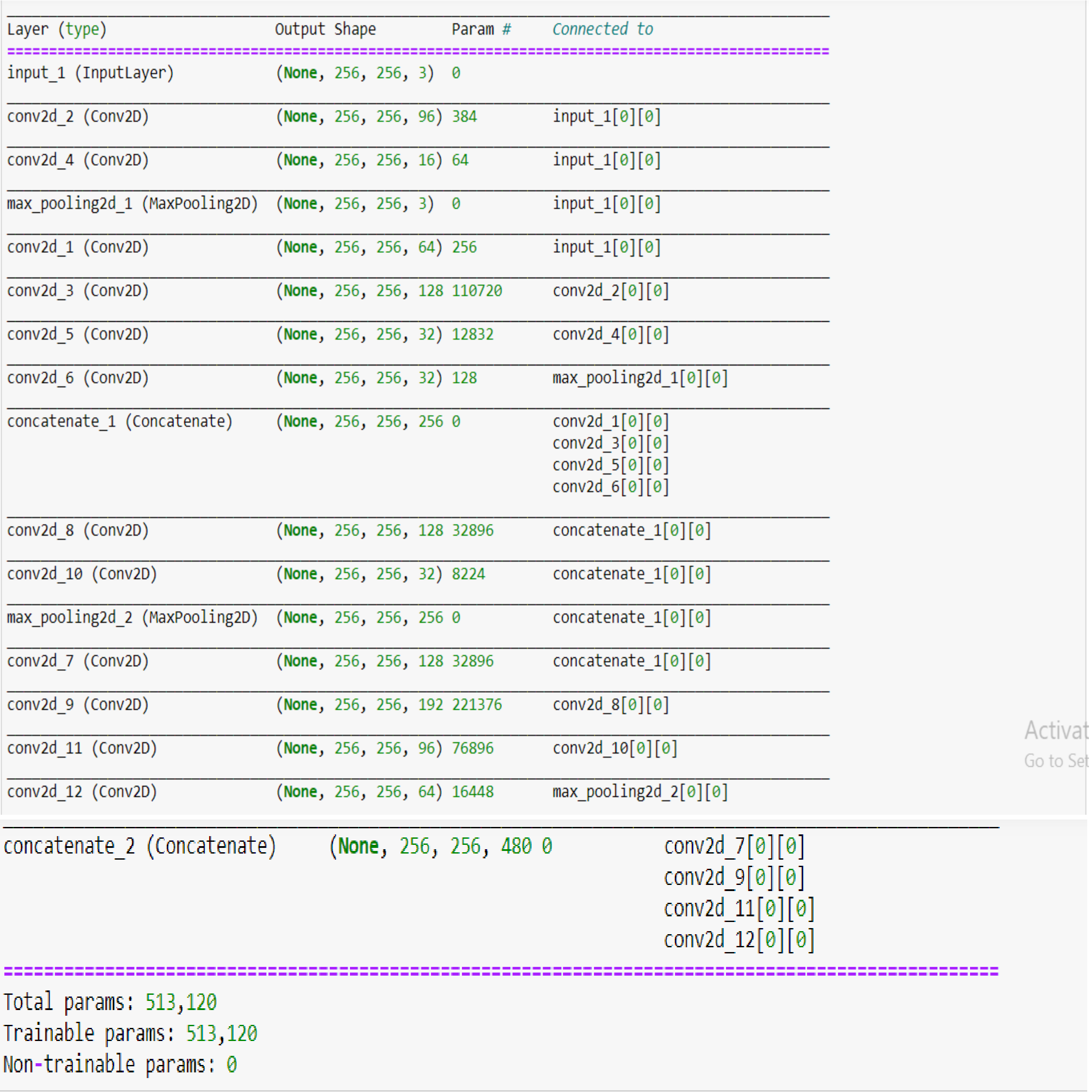
```
# function for creating a projected inception module
def inception_module(layer_in, f1, f2_in, f2_out, f3_in, f3_out, f4_out):
    # 1x1 conv
    conv1 = Conv2D(f1, (1,1), padding='same', activation='relu')(layer_in)
    # 3x3 conv
    conv3 = Conv2D(f2_in, (1,1), padding='same', activation='relu')(layer_in)
    conv3 = Conv2D(f2_out, (3,3), padding='same', activation='relu')(conv3)
    # 5x5 conv
    conv5 = Conv2D(f3_in, (1,1), padding='same', activation='relu')(layer_in)
    conv5 = Conv2D(f3_out, (5,5), padding='same', activation='relu')(conv5)
    # 3x3 max pooling
    pool = MaxPooling2D((3,3), strides=(1,1), padding='same')(layer_in)
    pool = Conv2D(f4_out, (1,1), padding='same', activation='relu')(pool)
    # concatenate filters, assumes filters/channels last
    layer_out = concatenate([conv1, conv3, conv5, pool], axis=-1)
    return layer_out
```

We can create a model with two of these optimized inception modules to get a concrete idea of how the architecture looks in practice.

In this case, the number of filter configurations are based on “*inception (3a)*” and “*inception (3b)*” from Table 1 in the paper.
The complete example is listed below.



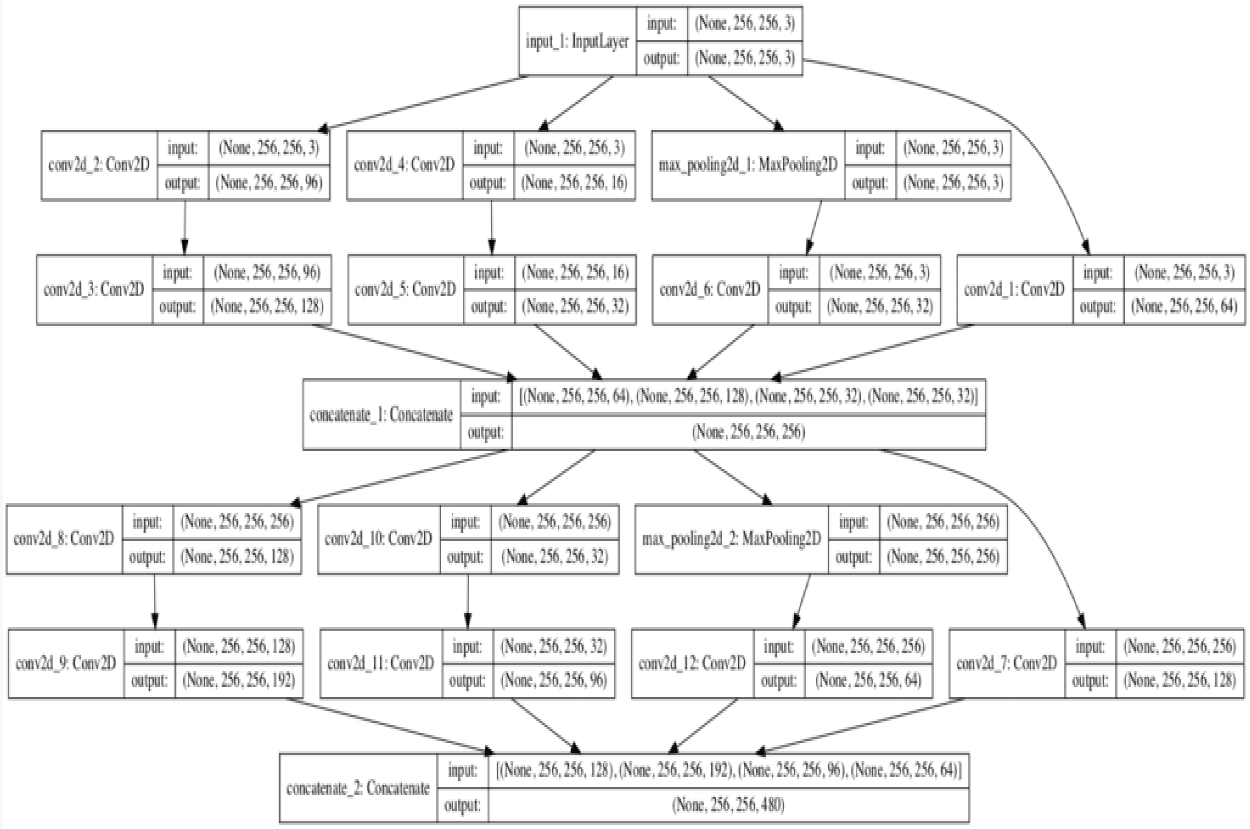
```
# example of creating a CNN with an efficient inception module
from keras.models import Model
from keras.layers import Input
from keras.layers import Conv2D
from keras.layers import MaxPooling2D
from keras.layers.merge import concatenate
from keras.utils import plot_model

# function for creating a projected inception module
def inception_module(layer_in, f1, f2_in, f2_out, f3_in, f3_out, f4_out):
    # 1x1 conv
    conv1 = Conv2D(f1, (1,1), padding='same', activation='relu')(layer_in)
    # 3x3 conv
    conv3 = Conv2D(f2_in, (1,1), padding='same', activation='relu')(layer_in)
    conv3 = Conv2D(f2_out, (3,3), padding='same', activation='relu')(conv3)
    # 5x5 conv
    conv5 = Conv2D(f3_in, (1,1), padding='same', activation='relu')(layer_in)
    conv5 = Conv2D(f3_out, (5,5), padding='same', activation='relu')(conv5)
    # 3x3 max pooling
    pool = MaxPooling2D((3,3), strides=(1,1), padding='same')(layer_in)
    pool = Conv2D(f4_out, (1,1), padding='same', activation='relu')(pool)
    # concatenate filters, assumes filters/channels last
    layer_out = concatenate([conv1, conv3, conv5, pool], axis=-1)
    return layer_out

# define model input
visible = Input(shape=(256, 256, 3))
# add inception block 1
layer = inception_module(visible, 64, 96, 128, 16, 32, 32)
# add inception block 1
layer = inception_module(layer, 128, 128, 192, 32, 96, 64)
# create model
model = Model(inputs=visible, outputs=layer)
# summarize model
model.summary()
# plot model architecture
plot_model(model, show_shapes=True, to_file='inception_module.png')
```



```
__________________________________________________________________________________________________
Layer (type)                    Output Shape         Param #     Connected to
==================================================================================================
input_1 (InputLayer)            (None, 256, 256, 3)  0
__________________________________________________________________________________________________
conv2d_2 (Conv2D)               (None, 256, 256, 96) 384         input_1[0][0]
__________________________________________________________________________________________________
conv2d_4 (Conv2D)               (None, 256, 256, 16) 64          input_1[0][0]
__________________________________________________________________________________________________
max_pooling2d_1 (MaxPooling2D)  (None, 256, 256, 3)  0           input_1[0][0]
__________________________________________________________________________________________________
conv2d_1 (Conv2D)               (None, 256, 256, 64) 256         input_1[0][0]
__________________________________________________________________________________________________
conv2d_3 (Conv2D)               (None, 256, 256, 128 110720      conv2d_2[0][0]
__________________________________________________________________________________________________
conv2d_5 (Conv2D)               (None, 256, 256, 32) 12832       conv2d_4[0][0]
__________________________________________________________________________________________________
conv2d_6 (Conv2D)               (None, 256, 256, 32) 128         max_pooling2d_1[0][0]
__________________________________________________________________________________________________
concatenate_1 (Concatenate)     (None, 256, 256, 256 0           conv2d_1[0][0]
                                                                 conv2d_3[0][0]
                                                                 conv2d_5[0][0]
                                                                 conv2d_6[0][0]
__________________________________________________________________________________________________
conv2d_8 (Conv2D)               (None, 256, 256, 128 32896       concatenate_1[0][0]
__________________________________________________________________________________________________
conv2d_10 (Conv2D)              (None, 256, 256, 32) 8224        concatenate_1[0][0]
__________________________________________________________________________________________________
max_pooling2d_2 (MaxPooling2D)  (None, 256, 256, 256 0           concatenate_1[0][0]
__________________________________________________________________________________________________
conv2d_7 (Conv2D)               (None, 256, 256, 128 32896       concatenate_1[0][0]
__________________________________________________________________________________________________
conv2d_9 (Conv2D)               (None, 256, 256, 192 221376      conv2d_8[0][0]
__________________________________________________________________________________________________
conv2d_11 (Conv2D)              (None, 256, 256, 96) 76896       conv2d_10[0][0]
__________________________________________________________________________________________________
conv2d_12 (Conv2D)              (None, 256, 256, 64) 16448       max_pooling2d_2[0][0]
__________________________________________________________________________________________________
concatenate_2 (Concatenate)     (None, 256, 256, 480 0           conv2d_7[0][0]
                                                                 conv2d_9[0][0]
                                                                 conv2d_11[0][0]
                                                                 conv2d_12[0][0]
==================================================================================================
Total params: 513,120
Trainable params: 513,120
Non-trainable params: 0
```

A plot of the model architecture is created that does make the layout of each module clear and how the first model feeds the second module.

Note that the first 1×1 convolution in each inception module is on the far right for space reasons, but besides that, the other layers are organized left to right within each module.

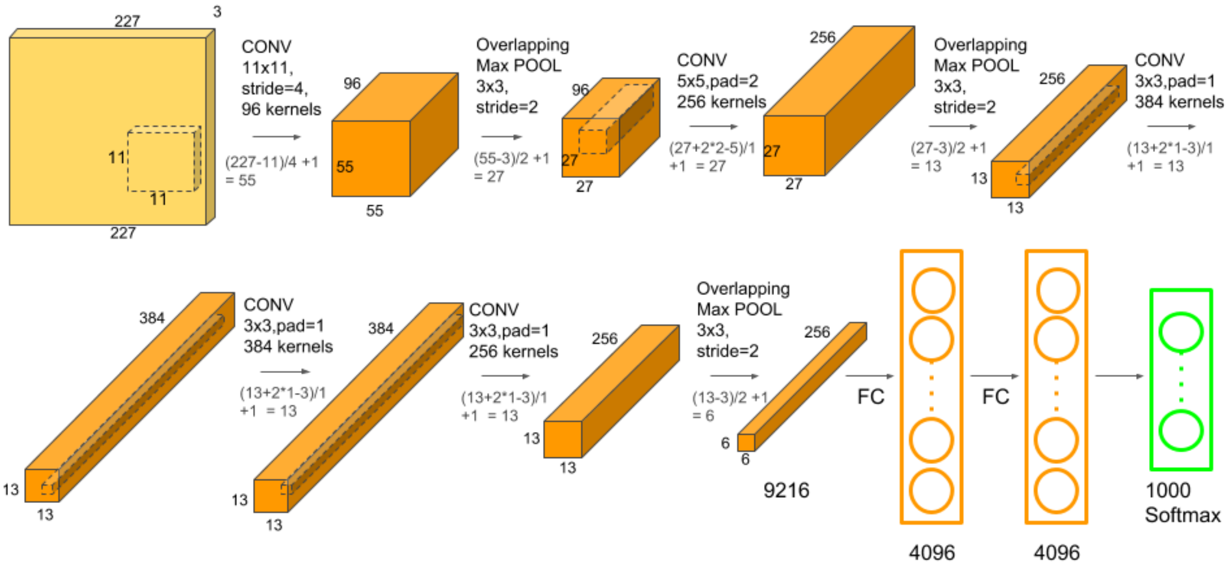

input_1: InputLayer | input: (None, 256, 256, 3) | output: (None, 256, 256, 3)
conv2d_2: Conv2D | input: (None, 256, 256, 3) | output: (None, 256, 256, 96)
conv2d_4: Conv2D | input: (None, 256, 256, 3) | output: (None, 256, 256, 16)
max_pooling2d_1: MaxPooling2D | input: (None, 256, 256, 3) | output: (None, 256, 256, 3)
conv2d_3: Conv2D | input: (None, 256, 256, 96) | output: (None, 256, 256, 128)
conv2d_5: Conv2D | input: (None, 256, 256, 16) | output: (None, 256, 256, 32)
conv2d_6: Conv2D | input: (None, 256, 256, 3) | output: (None, 256, 256, 32)
conv2d_1: Conv2D | input: (None, 256, 256, 3) | output: (None, 256, 256, 64)
concatenate_1: Concatenate | input: [(None, 256, 256, 64), (None, 256, 256, 128), (None, 256, 256, 32), (None, 256, 256, 32)] | output: (None, 256, 256, 256)
conv2d_8: Conv2D | input: (None, 256, 256, 256) | output: (None, 256, 256, 128)
conv2d_10: Conv2D | input: (None, 256, 256, 256) | output: (None, 256, 256, 32)
max_pooling2d_2: MaxPooling2D | input: (None, 256, 256, 256) | output: (None, 256, 256, 256)
conv2d_9: Conv2D | input: (None, 256, 256, 128) | output: (None, 256, 256, 192)
conv2d_11: Conv2D | input: (None, 256, 256, 32) | output: (None, 256, 256, 96)
conv2d_12: Conv2D | input: (None, 256, 256, 256) | output: (None, 256, 256, 64)
conv2d_7: Conv2D | input: (None, 256, 256, 256) | output: (None, 256, 256, 128)
concatenate_2: Concatenate | input: [(None, 256, 256, 128), (None, 256, 256, 192), (None, 256, 256, 96), (None, 256, 256, 64)] | output: (None, 256, 256, 480)

Plot of Convolutional Neural Network Architecture With a Efficient Inception Module

AlexNet:

AlexNet is the name of a convolutional neural network (CNN) architecture designed by Alex Krizhevsky in collaboration with Sutskever and Geoffrey Hinton, who was Krizhevsky's Ph.D. advisor.

AlexNet has eight layers; the first five were convolutional layers, some of them followed by max-pooling layers, and the last three were fully connected layers. The activation function used in this architecture is the non-saturating ReLU activation function, which showed improved training performance over tanh and sigmoid.[1]



The input Layer of the alexNet are 256*256*3 it means it supports 3 dimension pictures of 256*256 pixels. To reduce overfitting problem while training the model use drop out layers. The neurons do not contribute to the forward pass and do not participate in backpropagation. [1]

This model uses stochastic gradient descent as a optimization function with batch size, momentum and weight decay set to 128, 0.9, 0.00005 respectively. All the layers use an equal learning rate of 0.0001.

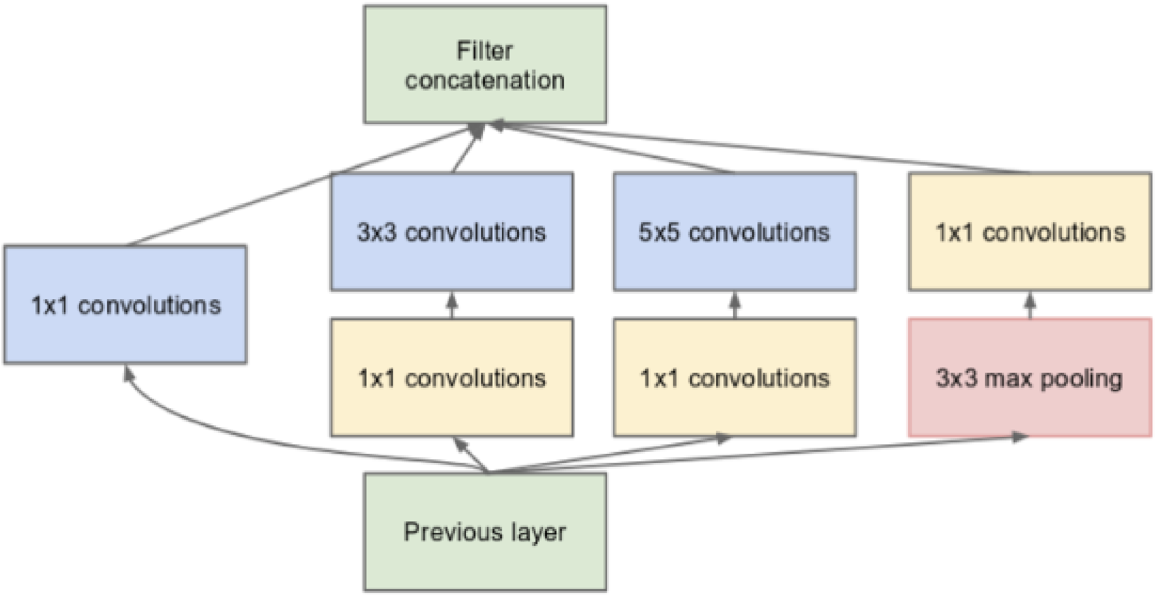
Popular deep learning framework like PyTorch and Tensorflow have the basic implementation of AlexNet architecture.

GoogleNet:

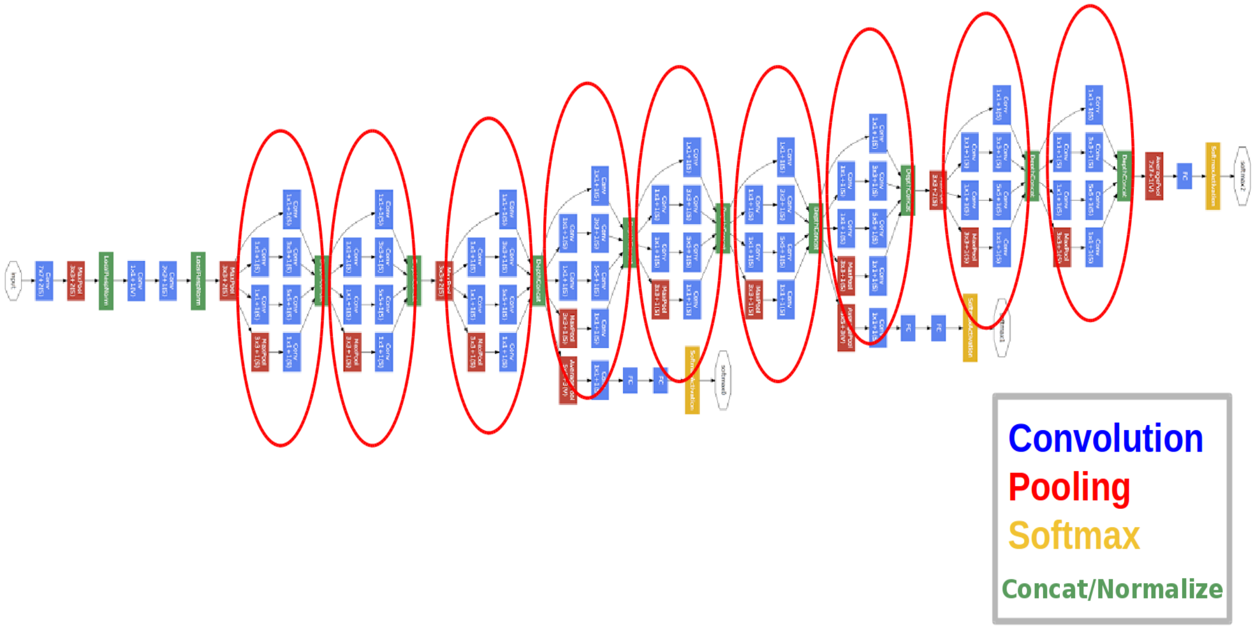
GoogleNet is one kind of Inception Network Which makes a huge impact in the field of neural network. So far there are three kind of Inception networks which are inception version 1,2 & 3. GooogleNet is the first version of Inception Network. It was developed by a team at google.

The GoogleNet architecture is not like AlexNet and ZF-Net. It uses many different kinds of methods such as 1*1 convolution and global average pooling that enables it to create deeper architecture.

A network with many deep layers may facing overfitting problem. So the author of the google net proposed with the idea of having filters on the same level and with multiple sizes. So it becomes a wider network than a deeper network.[1]



The googleNet has 22 deep layers with 27 pooling layer included. There are also 9 inception layer stacked linearly. At the end the inception modules are attached with the global average pooling layer. [1]



The googleNet is trained using distributed machine learning system with a modest amount of model and data parallelism. The training used asynchronous stochastic gradient descent with a momentum of 0.9 and a fixed learning rate schedule decreasing the learning rate by 4% every 8 epochs.

Discussion

AlexNet was the first central CNN model that used GPUs for training. This leads to faster movement of models. AlexNet is a more profound architecture with eight layers, which means that it can better extract features compared to LeNet. It also worked well for the time with color images. The ReLu activation function used in this network has two advantages. It does not limit the output, unlike other activation functions. This means there isn't too much loss of features. It negates the negative result of the summation of gradients and not the dataset itself. This means that it will further improve model training speed since not all perceptions are active. It has some disadvantages also. Compared to models used in this article, the depth of this model is significantly less, and hence it struggles to learn features from image sets. We can see that it needs more extra time to achieve higher accuracy results compared to future models.

VGG brought with it a massive improvement in accuracy and a speed improvement as well. This was primarily because of improving the depth of the model and also introducing pre-trained models. The increase in the number of layers with smaller kernels saw a rise in non-linearity, which is always a positive in deep learning. VGG brought with it various architectures built on a similar concept. This gives more options to us as to which architecture could best fit our application. It has some disadvantages also. One major check that I found was that this model experiences the vanishing gradient problem. If we look at my validation loss graph, we see it increasing as a trend. This wasn't the problem with any of the other models. The vanishing gradient problem was solved with the ResNet architecture. VGG is slower than the newer ResNet architecture that introduced residual learning, which was another breakthrough.

The ResNet architecture does not need to fire all neurons in every epoch. This significantly reduces the training time and improves accuracy. Once a feature is learned, it does not try to understand it again but instead focuses on learning newer features. A brilliant approach that greatly improved model training performance. The complexity of an identical VGG network caused the degradation problem, which was solved by residual learning.

GoogleNet trains faster than VGG. The area of a pre-trained GoogleNet is approximately less than VGG. A VGG reduction can have >500 MBs, whereas GoogleNet has a length of only 96 MB. GoogleNet does not have an actual problem per se, but further architecture changes are proposed, making the design perform better. One such modification is termed an Xception Network, in which the end of divergence of conception module (4 in GoogleNet as we saw in the image above) is improved. It can now theoretically be infinite (hence called extreme inception!)

Inception v3 mainly focuses on consuming less computational power by modifying the past Inception architectures. In connection to VGGNet, Inception Networks

(GoogLeNet/Inception v1) have proved to be more computationally effective, both in times of the number of parameters created by the network and the reasonable cost incurred (memory and other resources). If any changes are made to an Inception Network, care needs to be needed to ensure that the computational advantages aren't lost. Thus, adopting an Inception network for different use cases turns out to be a problem due to the uncertainty of the new network's response. In an Initiation v3 model, various techniques for optimizing the network have been suggested to loosen the constraints for more accessible model adaptation. The methods include factorized convolutions, regularization, dimension reduction, and parallelized computations.

Conclusion:

Among all of these structures that we have discussed about, some of the architectures are designed especially for large scale data analysis (such as GoogLeNet and ResNet), whereas the VGG network is considered a general architecture. The use of Convolutional Neural Network with fewer layers has the advantage of lower hardware requirements and shorter training times compared to their deeper counterparts. Shorter training times allow to test more hyper-parameters and facilitates the overall training process. Lower hardware requirements also enable the use of increased image resolutions.

Learning capacity of Convolutional Neural Network is significantly improved over the years by doing some structural modifications. It is observed in recent literature that the main boost in Convolutional Neural Network performance has been achieved by replacing the conventional layer structure with blocks. These blocks play a significant role in boosting of Convolutional Neural Network performance by making problem aware learning. Moreover, block based architecture of Convolutional Neural Network encourages learning in a modular fashion and thereby, making architecture more simple and understandable. The concept of block being a structural unit is going to persist and further enhance Convolutional Neural Network performance.

References

[1] Vihar Kurama, “A review of popular deep learning Architecture,” PaperspaceBlog [Online], May 19,2020 Available: https://blog.paperspace.com/popular-deep-learning-architectures-alexnet-vgg-googlenet/?fbclid=IwAR3YwAXmq0U0jDXZriCloBuZreDk_Xs58GRwNlWn7RyG2wDADRY_nTT0v_Y

[2] https://machinelearningmastery.com/how-to-implement-major-architecture-innovations-for-convolutional-neural-networks/?fbclid=IwAR37wbDfSNecrTW8ToG8TYzHroq8Z-GVdFM3c-HTBCeAdoGthRR8nDJqdOk