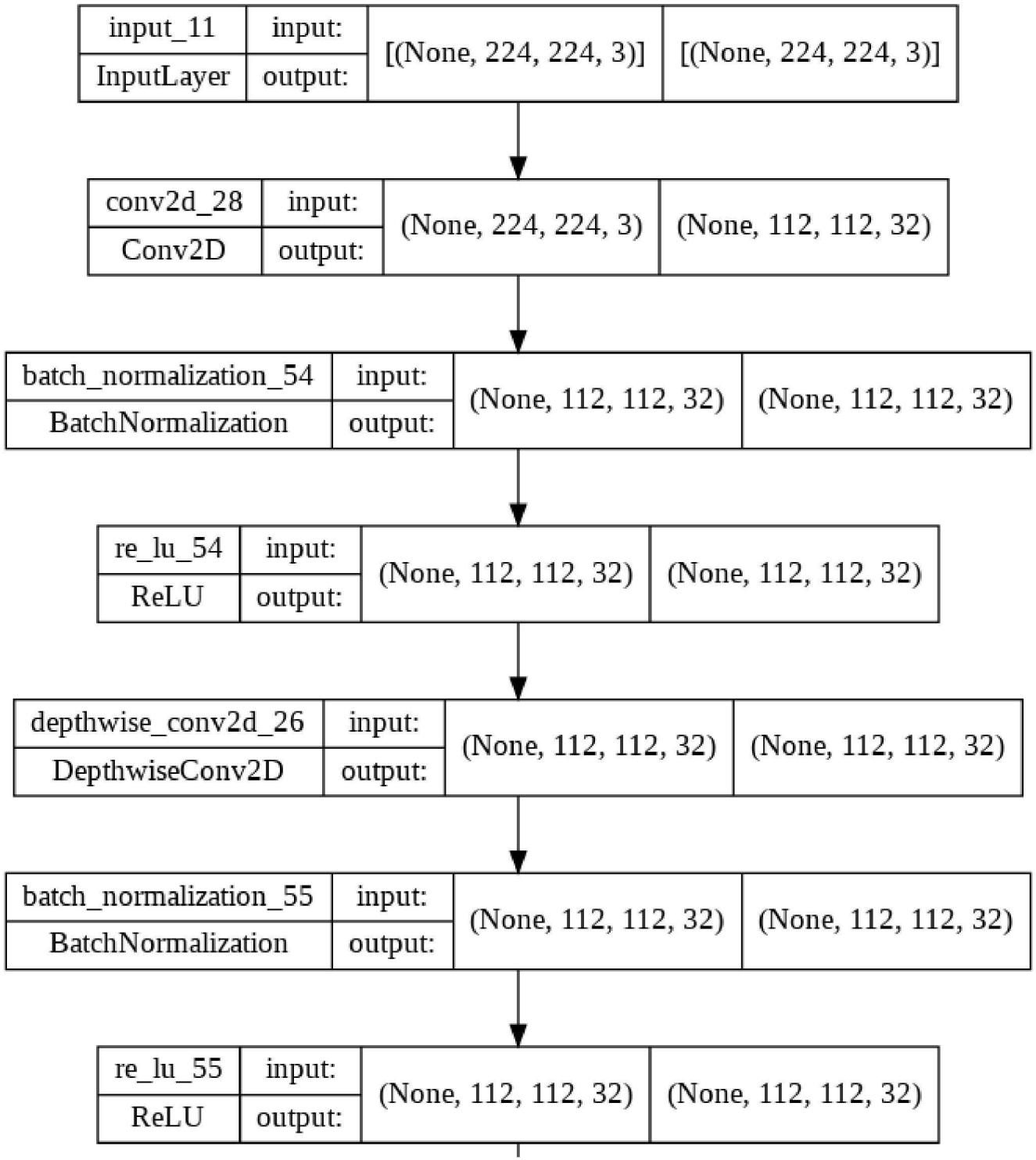
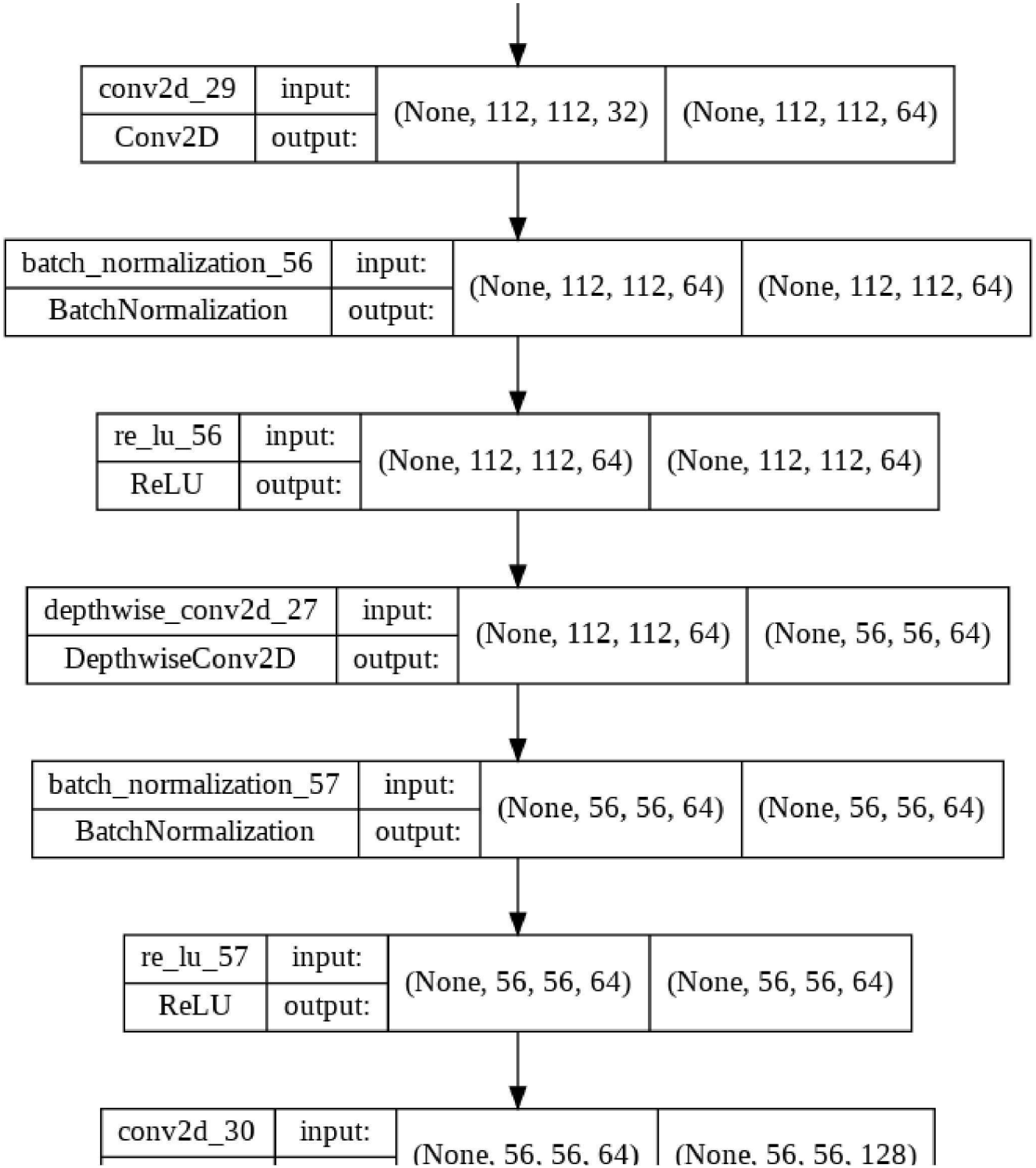
```
batch_normalization_77 (Bat  (None, 7, 7, 512)        2048
chNormalization)

re_lu_77 (ReLU)              (None, 7, 7, 512)        0

conv2d_40 (Conv2D)           (None, 7, 7, 1024)       525312

batch_normalization_78 (Bat  (None, 7, 7, 1024)       4096
chNormalization)

re_lu_78 (ReLU)              (None, 7, 7, 1024)       0

depthwise_conv2d_38 (Depthw  (None, 7, 7, 1024)       10240
iseConv2D)

batch_normalization_79 (Bat  (None, 7, 7, 1024)       4096
chNormalization)

re_lu_79 (ReLU)              (None, 7, 7, 1024)       0

conv2d_41 (Conv2D)           (None, 7, 7, 1024)       1049600

batch_normalization_80 (Bat  (None, 7, 7, 1024)       4096
chNormalization)

re_lu_80 (ReLU)              (None, 7, 7, 1024)       0

average_pooling2d_2 (Averag  (None, 7, 1, 1018)       0
ePooling2D)

dense_14 (Dense)             (None, 7, 1, 1000)       1019000

=================================================================
Total params: 4,258,808
Trainable params: 4,236,920
Non-trainable params: 21,888
_________________________________________________________________
```

In [ ]:
```
#plot the model
tf.keras.utils.plot_model(model, to_file='model.png', show_shapes=True, show_dtype=False,show_layer_names=True, rankdir
```

Out[ ]:


input_11
InputLayer
input:
output:
[(None, 224, 224, 3)]
[(None, 224, 224, 3)]
conv2d_28
Conv2D
input:
output:
(None, 224, 224, 3)
(None, 112, 112, 32)
batch_normalization_54
BatchNormalization
input:
output:
(None, 112, 112, 32)
(None, 112, 112, 32)
re_lu_54
ReLU
input:
output:
(None, 112, 112, 32)
(None, 112, 112, 32)
depthwise_conv2d_26
DepthwiseConv2D
input:
output:
(None, 112, 112, 32)
(None, 112, 112, 32)
batch_normalization_55
BatchNormalization
input:
output:
(None, 112, 112, 32)
(None, 112, 112, 32)
re_lu_55
ReLU
input:
output:
(None, 112, 112, 32)
(None, 112, 112, 32)


conv2d_29
Conv2D
input:
output:
(None, 112, 112, 32)
(None, 112, 112, 64)
batch_normalization_56
BatchNormalization
input:
output:
(None, 112, 112, 64)
(None, 112, 112, 64)
re_lu_56
ReLU
input:
output:
(None, 112, 112, 64)
(None, 112, 112, 64)
depthwise_conv2d_27
DepthwiseConv2D
input:
output:
(None, 112, 112, 64)
(None, 56, 56, 64)
batch_normalization_57
BatchNormalization
input:
output:
(None, 56, 56, 64)
(None, 56, 56, 64)
re_lu_57
ReLU
input:
output:
(None, 56, 56, 64)
(None, 56, 56, 64)
conv2d_30
input:
(None, 56, 56, 64)
(None, 56, 56, 128)

Conv2D	output:		

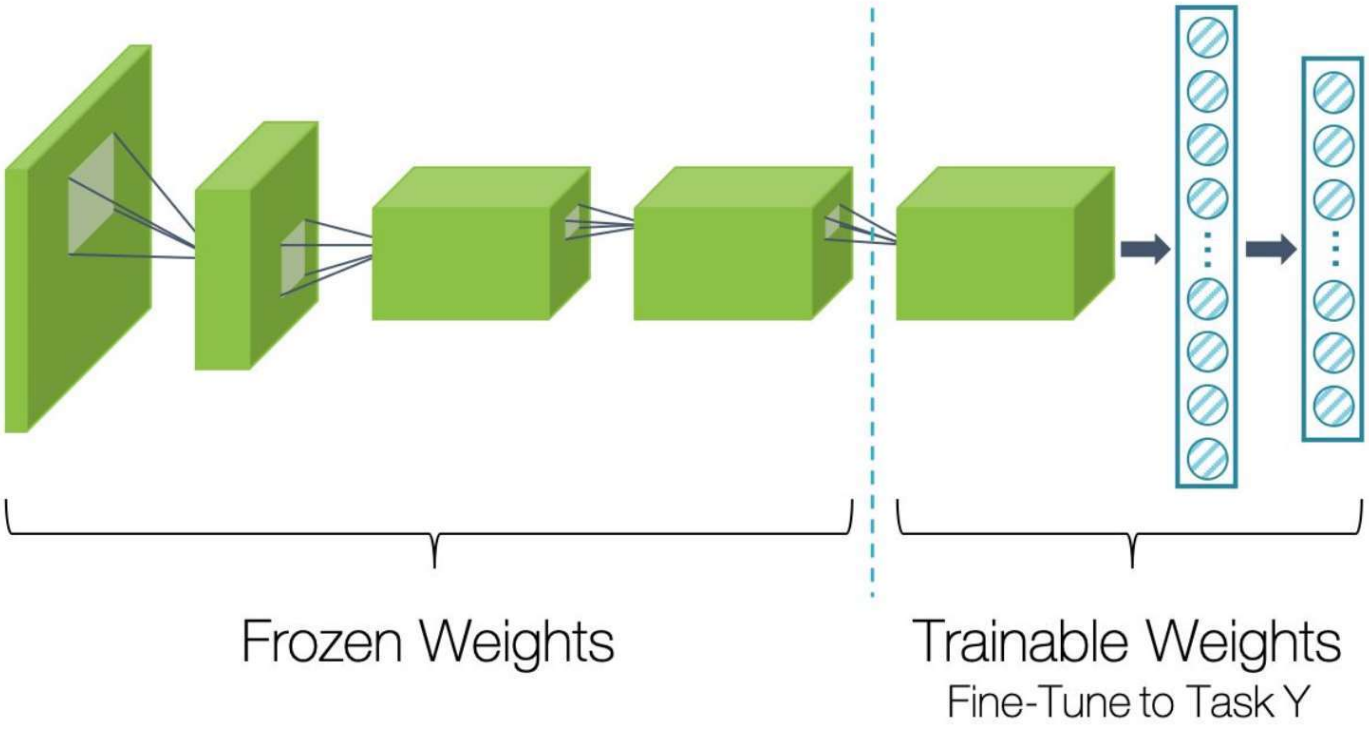
Fine Tuning for pneumonia xray images:

batch_normalization_58	input:	(None, 56, 56, 128)	(None, 56, 56, 128)
BatchNormalization	output:		

Now that we understand how aMobileNet works, lets use the Mobilenet with pretrained weights on ImageNet dataset.

- Basic transfer learning tells us only to train
  - We usually add only two to three fully connected layers after the generic layers to make the new classifier model.
- But if our task is identifying penumonia or different bones in an X-ray image and we want to start out from an ImageNet trained network, the high dissimilarity between regular ImageNet images and X-ray images
- So if you have large amount of data and want higher accuracy, we must allow more layers to be trained.
- This means unfreezing some of the layers that would have otherwise been frozen in transfer learning. This is known as fine tuning. Figure shows an example where some convolutional layers near the head/top are unfrozen and trained for the task at hand.

re_lu_58	input:	(None, 56, 56, 128)	(None, 56, 56, 128)
ReLU	output:		



- We often hear data scientists saying, "I fine tuned the model", which means that they took a pretrained model, removed task-specific layers and added new ones, froze the lower layers,
- and trained the upper part of the network on the new dataset they had.

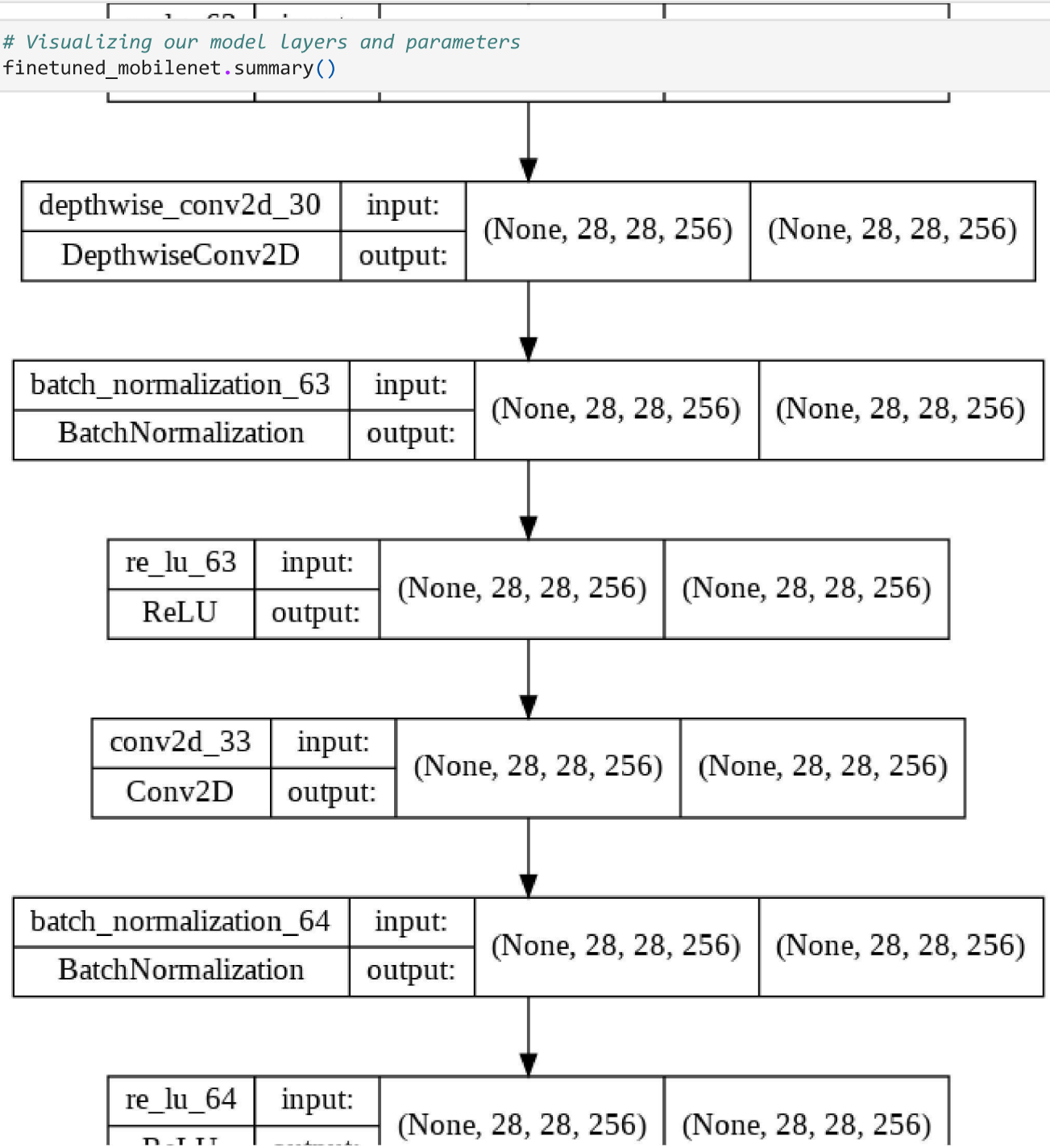
NOTE:
In daily lingo, transfer learning and fine tuning are used interchangeably. When spoken, transfer learning is used more as a general concept, whereas fine tuning is referred to as its implementation.

- But in our case since our dataset is small so performing finetuning will increase the number of parameters to train and result in overfitting.
- Lets see how to finetune using Mobilenet:

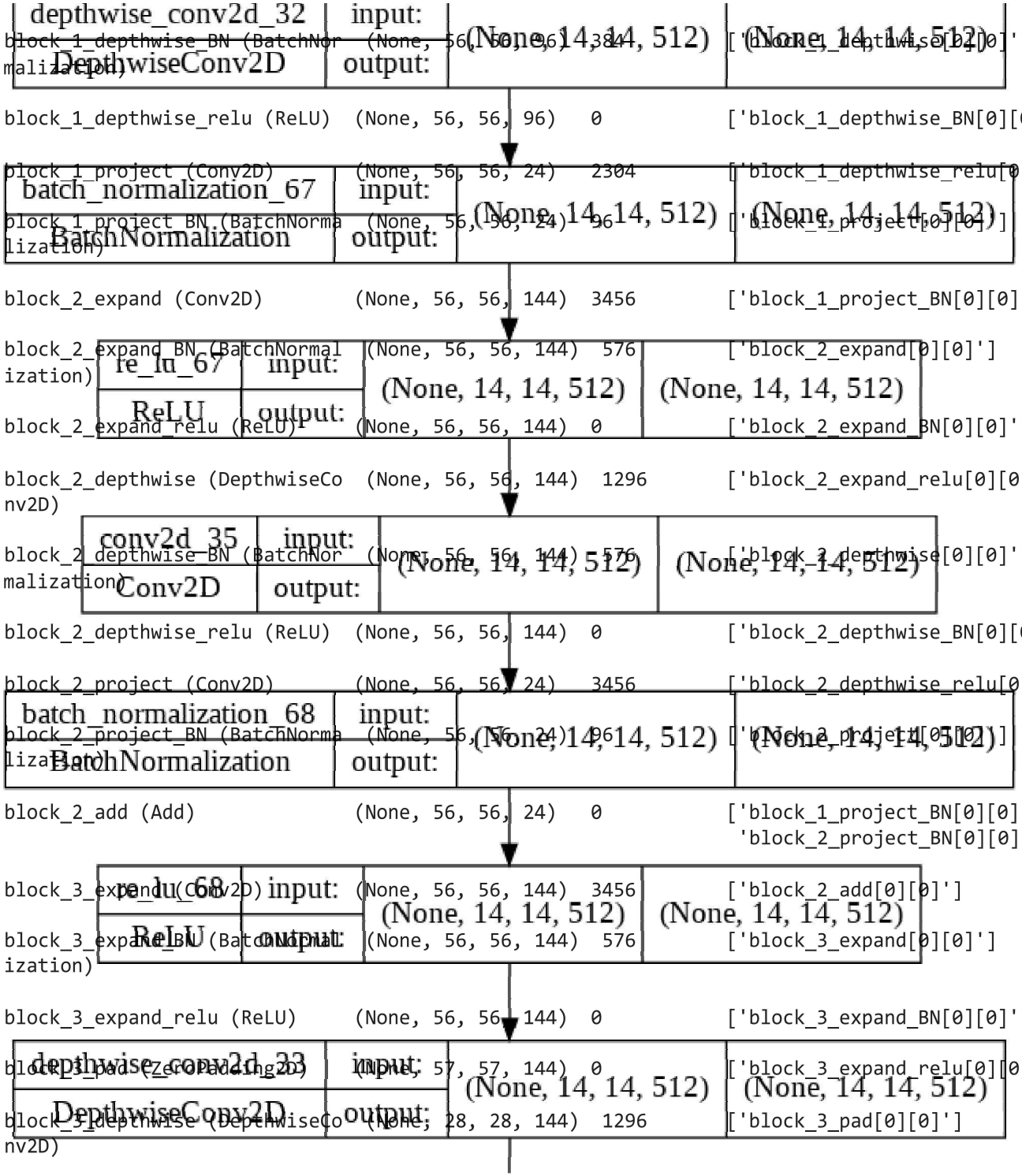
In [ ]:
```
def build_model():

    mobilenet_model = tf.keras.applications.MobileNetV2(
        weights ='imagenet',
        include_top = False,
        input_shape = (224,224,3)
    )

    #Freezing the pretrained mobilenet layers except the last  layer
    # Known as fintuning the model

    for layer in mobilenet_model.layers[:-2]:
        layer.trainable = False


    # for layer in mobilenet_model.layers:
    #     layer.trainable = False


    #Output of base model
    x = mobilenet_model.output
    x = tf.keras.layers.GlobalAveragePooling2D()(x)
    x = tf.keras.layers.Dense(128, activation = "relu")(x)
    output = tf.keras.layers.Dense(1, activation = 'sigmoid')(x)
    pretrained_model = tf.keras.Model(inputs = mobilenet_model.input, outputs = output)

    return pretrained_model
```

In [ ]:
```
finetuned_mobilenet = build_model()
```

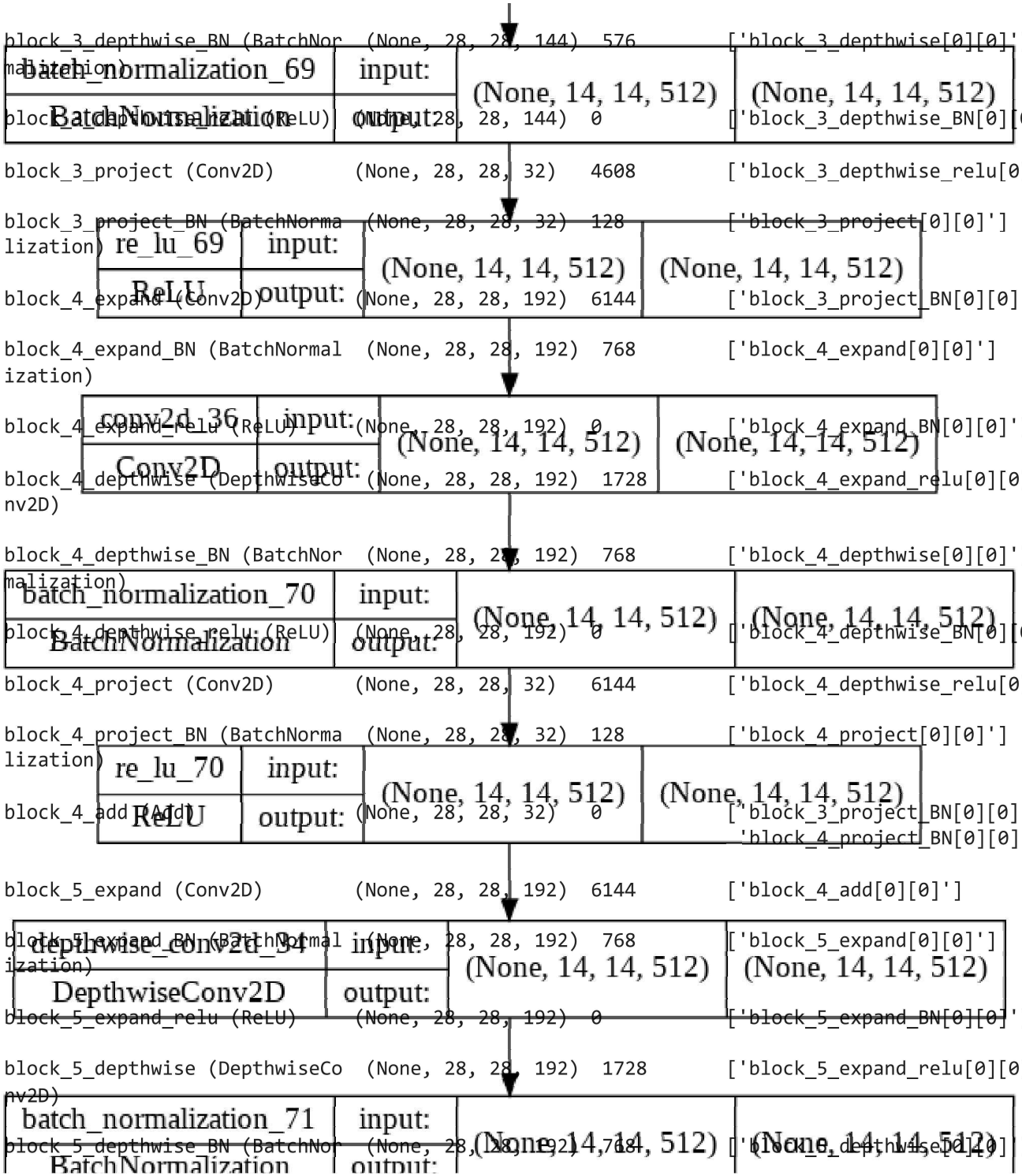
In [ ]:
```
# Visualizing our model layers and parameters
finetuned_mobilenet.summary()
```

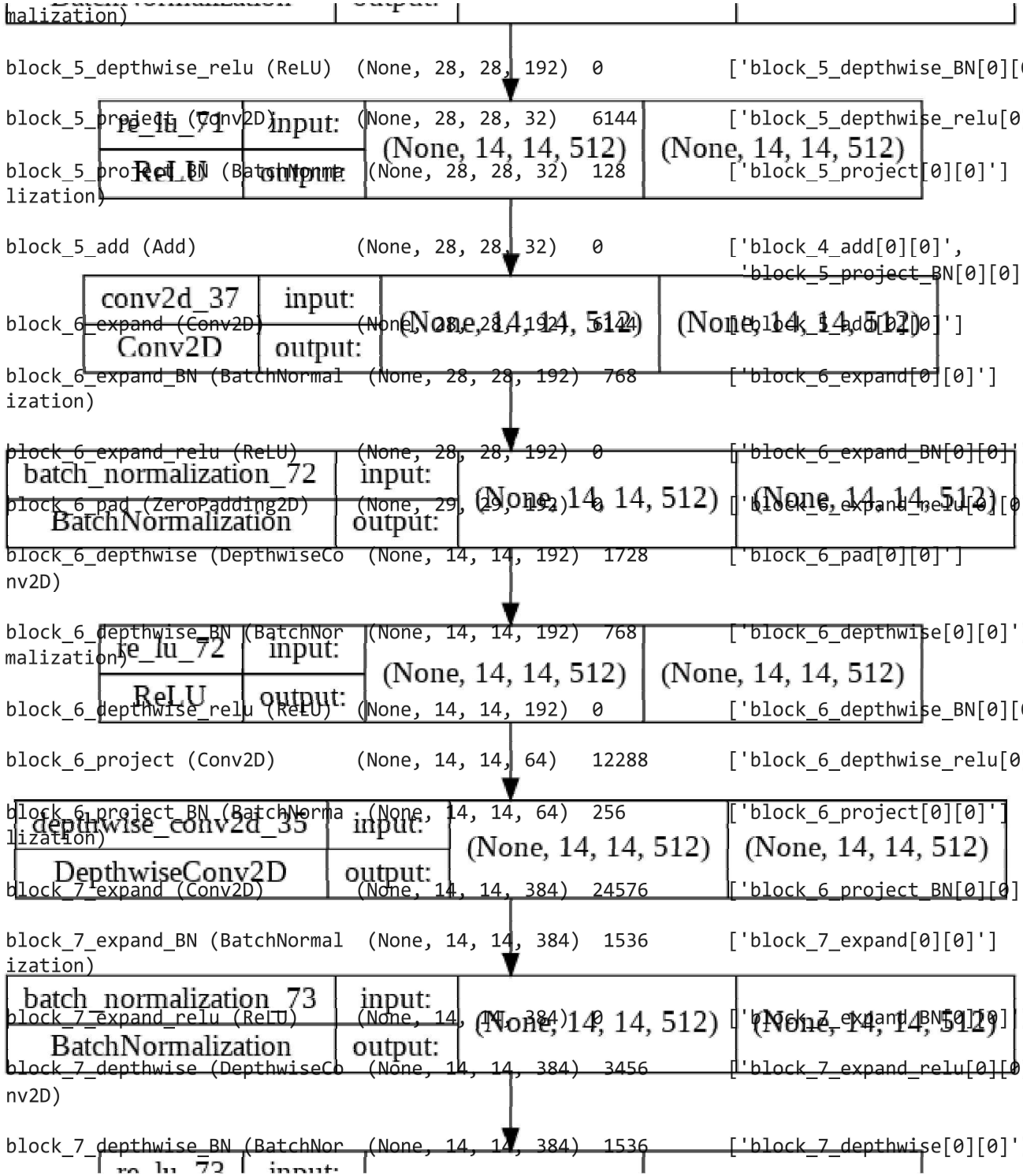


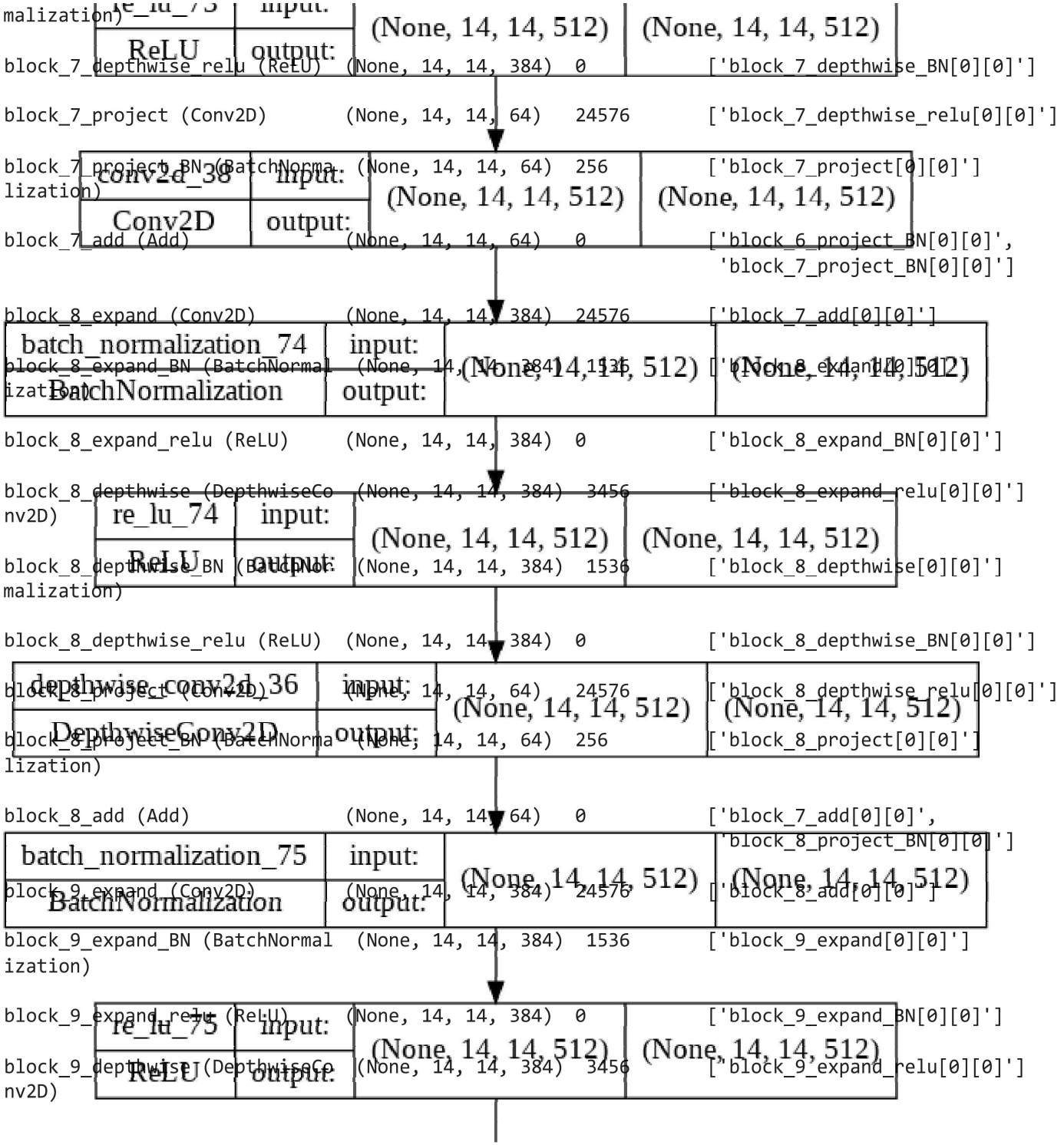
```
Model: "model_11"
__________________________________________________________________________________________________
Layer (type)                    Output Shape         Param #     Connected to
==================================================================================================
input_12 (InputLayer)           [(None, 224, 224, 3  0           []
                                )]

Conv1 (Conv2D)                  (None, 112, 112, 32  864         ['input_12[0][0]']
                                )

bn_Conv1 (BatchNormalization)   (None, 112, 112, 32  128         ['Conv1[0][0]']
                                )

Conv1_relu (ReLU)               (None, 112, 112, 32  0           ['bn_Conv1[0][0]']
                                )

expanded_conv_depthwise (Depth  (None, 112, 112, 32  288         ['Conv1_relu[0][0]']
wiseConv2D)                     )

expanded_conv_depthwise_BN (Ba  (None, 112, 112, 32  128         ['expanded_conv_depthwise[0][0]']
tchNormalization)               )

expanded_conv_depthwise_relu (  (None, 112, 112, 32  0           ['expanded_conv_depthwise_BN[0][0
ReLU)                           )                                ]']

expanded_conv_project (Conv2D)  (None, 112, 112, 16  512         ['expanded_conv_depthwise_relu[0]
                                )                                [0]']

expanded_conv_project_BN (Batc  (None, 112, 112, 16  64          ['expanded_conv_project[0][0]']
hNormalization)                 )

block_1_expand (Conv2D)         (None, 112, 112, 96  1536        ['expanded_conv_project_BN[0][0]'
                                )                                ]

block_1_expand_BN (BatchNormal  (None, 112, 112, 96  384         ['block_1_expand[0][0]']
ization)                        )

block_1_expand_relu (ReLU)      (None, 112, 112, 96  0           ['block_1_expand_BN[0][0]']
                                )

block_1_pad (ZeroPadding2D)     (None, 113, 113, 96  0           ['block_1_expand_relu[0][0]']
                                )

block_1_depthwise (DepthwiseCo  (None, 56, 56, 96)  864          ['block_1_pad[0][0]']
nv2D)
```



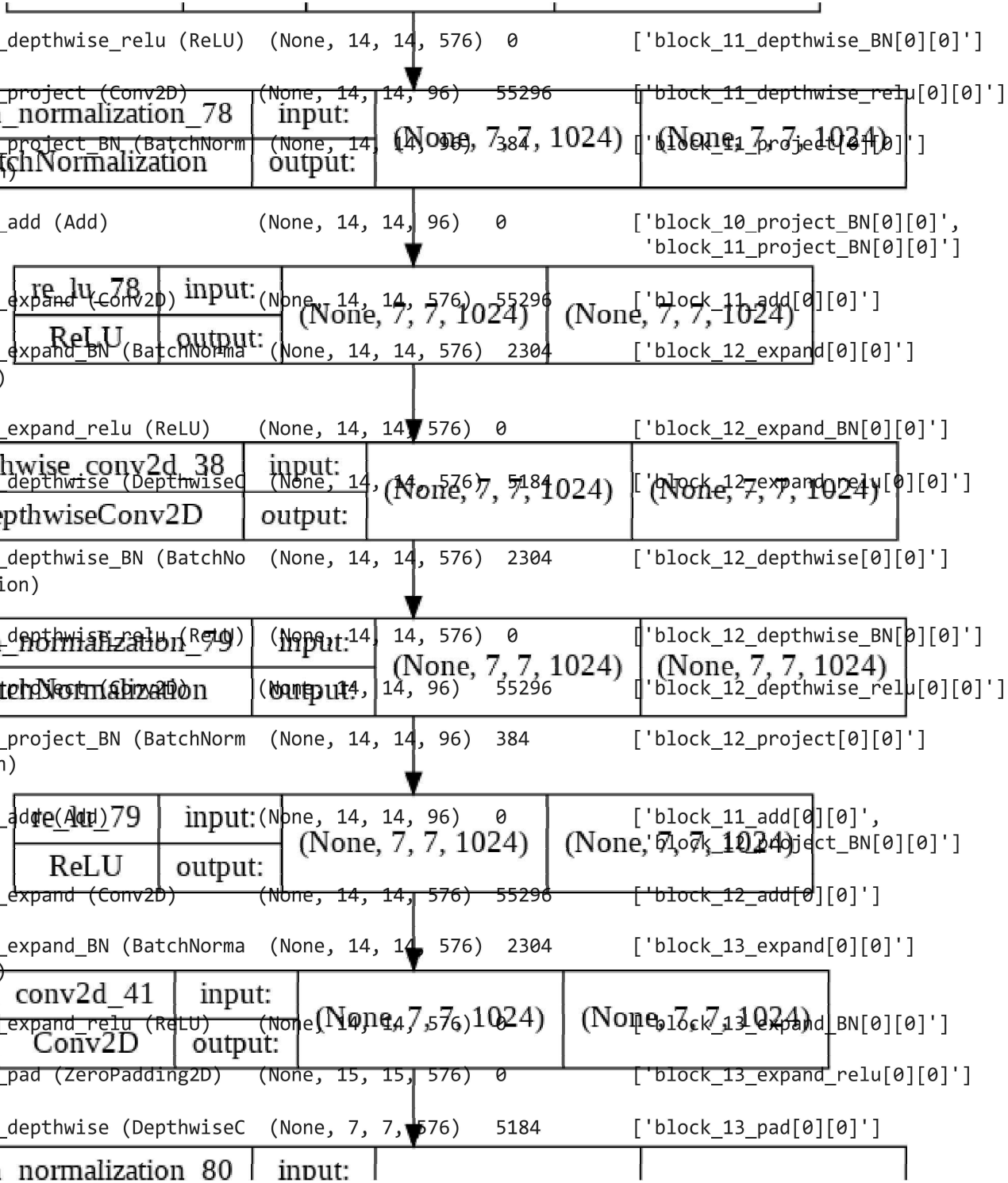

```
block_1_depthwise_BN (BatchNor  (None, 56, 56, 96)   384         ['block_1_depthwise[0][0]']
malization)

block_1_depthwise_relu (ReLU)  (None, 56, 56, 96)   0           ['block_1_depthwise_BN[0][0]']

block_1_project (Conv2D)       (None, 56, 56, 24)   2304        ['block_1_depthwise_relu[0][0]']

block_1_project_BN (BatchNorma  (None, 56, 56, 24)  96          ['block_1_project[0][0]']
lization)

block_2_expand (Conv2D)        (None, 56, 56, 144)  3456        ['block_1_project_BN[0][0]']

block_2_expand_BN (BatchNormal  (None, 56, 56, 144)  576         ['block_2_expand[0][0]']
ization)

block_2_expand_relu (ReLU)     (None, 56, 56, 144)  0           ['block_2_expand_BN[0][0]']

block_2_depthwise (DepthwiseCo  (None, 56, 56, 144)  1296        ['block_2_expand_relu[0][0]']
nv2D)

block_2_depthwise_BN (BatchNor  (None, 56, 56, 144)  576         ['block_2_depthwise[0][0]']
malization)

block_2_depthwise_relu (ReLU)  (None, 56, 56, 144)  0           ['block_2_depthwise_BN[0][0]']

block_2_project (Conv2D)       (None, 56, 56, 24)   3456        ['block_2_depthwise_relu[0][0]']

block_2_project_BN (BatchNorma  (None, 56, 56, 24)  96          ['block_2_project[0][0]']
lization)

block_2_add (Add)              (None, 56, 56, 24)   0           ['block_1_project_BN[0][0]',
                                                                 'block_2_project_BN[0][0]']

block_3_expand (Conv2D)        (None, 56, 56, 144)  3456        ['block_2_add[0][0]']

block_3_expand_BN (BatchNormal  (None, 56, 56, 144)  576         ['block_3_expand[0][0]']
ization)

block_3_expand_relu (ReLU)     (None, 56, 56, 144)  0           ['block_3_expand_BN[0][0]']

block_3_pad (ZeroPadding2D)    (None, 57, 57, 144)  0           ['block_3_expand_relu[0][0]']

block_3_depthwise (DepthwiseCo  (None, 28, 28, 144)  1296        ['block_3_pad[0][0]']
nv2D)
```

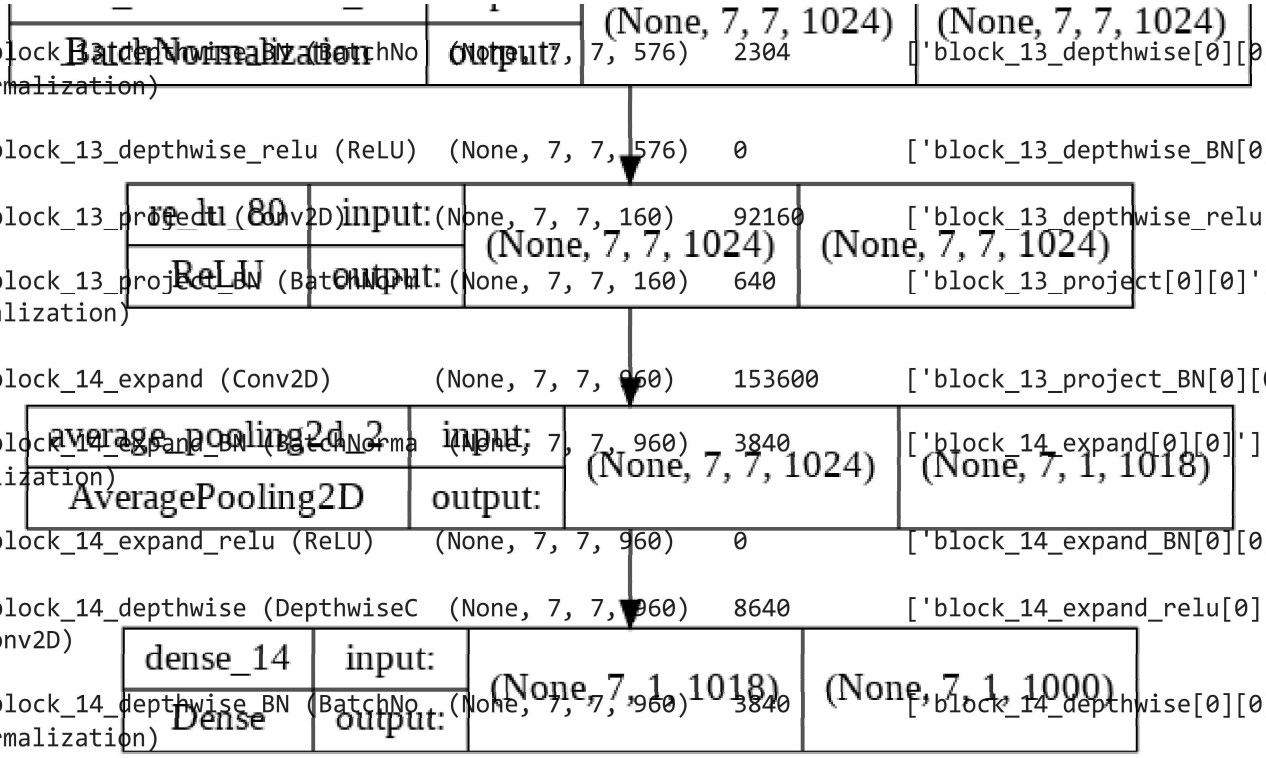

block_3_depthwise_BN (BatchNor (None, 28, 28, 144) 576 ['block_3_depthwise[0][0]']
malization)
batch_normalization_69
BatchNormalization
input:
output:
(None, 14, 14, 512)
(None, 14, 14, 512)
block_3_depthwise_relu (ReLU) (None, 28, 28, 144) 0 ['block_3_depthwise_BN[0][0]']
block_3_project (Conv2D) (None, 28, 28, 32) 4608 ['block_3_depthwise_relu[0][0]']
block_3_project_BN (BatchNorma (None, 28, 28, 32) 128 ['block_3_project[0][0]']
lization)
re_lu_69
ReLU
input:
output:
(None, 14, 14, 512)
(None, 14, 14, 512)
block_4_expand (Conv2D) (None, 28, 28, 192) 6144 ['block_3_project_BN[0][0]']
block_4_expand_BN (BatchNormal (None, 28, 28, 192) 768 ['block_4_expand[0][0]']
ization)
conv2d_36
Conv2D
input:
output:
(None, 14, 14, 512)
(None, 14, 14, 512)
block_4_expand_relu (ReLU) (None, 28, 28, 192) 0 ['block_4_expand_BN[0][0]']
block_4_depthwise (DepthwiseCo (None, 28, 28, 192) 1728 ['block_4_expand_relu[0][0]']
nv2D)
block_4_depthwise_BN (BatchNor (None, 28, 28, 192) 768 ['block_4_depthwise[0][0]']
malization)
batch_normalization_70
BatchNormalization
input:
output:
(None, 14, 14, 512)
(None, 14, 14, 512)
block_4_depthwise_relu (ReLU) (None, 28, 28, 192) 0 ['block_4_depthwise_BN[0][0]']
block_4_project (Conv2D) (None, 28, 28, 32) 6144 ['block_4_depthwise_relu[0][0]']
block_4_project_BN (BatchNorma (None, 28, 28, 32) 128 ['block_4_project[0][0]']
lization)
re_lu_70
ReLU
input:
output:
(None, 14, 14, 512)
(None, 14, 14, 512)
block_4_add (Add) (None, 28, 28, 32) 0 ['block_3_project_BN[0][0]',
'block_4_project_BN[0][0]']
block_5_expand (Conv2D) (None, 28, 28, 192) 6144 ['block_4_add[0][0]']
block_5_expand_BN (BatchNormal (None, 28, 28, 192) 768 ['block_5_expand[0][0]']
ization)
depthwise_conv2d_34
DepthwiseConv2D
input:
output:
(None, 14, 14, 512)
(None, 14, 14, 512)
block_5_expand_relu (ReLU) (None, 28, 28, 192) 0 ['block_5_expand_BN[0][0]']
block_5_depthwise (DepthwiseCo (None, 28, 28, 192) 1728 ['block_5_expand_relu[0][0]']
nv2D)
batch_normalization_71
BatchNormalization
input:
output:
(None, 14, 14, 512)
(None, 14, 14, 512)
block_5_depthwise_BN (BatchNor (None, 28, 28, 192) 768 ['block_5_depthwise[0][0]']


malization)
block_5_depthwise_relu (ReLU) (None, 28, 28, 192) 0 ['block_5_depthwise_BN[0][0]']
block_5_project (Conv2D) (None, 28, 28, 32) 6144 ['block_5_depthwise_relu[0][0]']
re_lu_71 input: (None, 14, 14, 512) (None, 14, 14, 512)
ReLU output:
block_5_project_BN (BatchNorm (None, 28, 28, 32) 128 ['block_5_project[0][0]']
lization)
block_5_add (Add) (None, 28, 28, 32) 0 ['block_4_add[0][0]',
'block_5_project_BN[0][0]']
conv2d_37 input: (None, 14, 14, 512) (None, 14, 14, 512)
Conv2D output:
block_6_expand (Conv2D) (None, 28, 28, 192) ['block_5_add[0][0]']
block_6_expand_BN (BatchNormal (None, 28, 28, 192) 768 ['block_6_expand[0][0]']
ization)
block_6_expand_relu (ReLU) (None, 28, 28, 192) 0 ['block_6_expand_BN[0][0]']
batch_normalization_72 input: (None, 14, 14, 512) (None, 14, 14, 512)
BatchNormalization output:
block_6_pad (ZeroPadding2D) (None, 29, 29, 192) 0 ['block_6_expand_relu[0][0]']
block_6_depthwise (DepthwiseCo (None, 14, 14, 192) 1728 ['block_6_pad[0][0]']
nv2D)
block_6_depthwise_BN (BatchNor (None, 14, 14, 192) 768 ['block_6_depthwise[0][0]']
malization)
re_lu_72 input: (None, 14, 14, 512) (None, 14, 14, 512)
ReLU output:
block_6_depthwise_relu (ReLU) (None, 14, 14, 192) 0 ['block_6_depthwise_BN[0][0]']
block_6_project (Conv2D) (None, 14, 14, 64) 12288 ['block_6_depthwise_relu[0][0]']
block_6_project_BN (BatchNorma (None, 14, 14, 64) 256 ['block_6_project[0][0]']
lization)
depthwise_conv2d_35 input: (None, 14, 14, 512) (None, 14, 14, 512)
DepthwiseConv2D output:
block_7_expand (Conv2D) (None, 14, 14, 384) 24576 ['block_6_project_BN[0][0]']
block_7_expand_BN (BatchNormal (None, 14, 14, 384) 1536 ['block_7_expand[0][0]']
ization)
batch_normalization_73 input: (None, 14, 14, 512) (None, 14, 14, 512)
BatchNormalization output:
block_7_expand_relu (ReLU) (None, 14, 14, 384) 0 ['block_7_expand_BN[0][0]']
block_7_depthwise (DepthwiseCo (None, 14, 14, 384) 3456 ['block_7_expand_relu[0][0]']
nv2D)
block_7_depthwise_BN (BatchNor (None, 14, 14, 384) 1536 ['block_7_depthwise[0][0]']


malization)
re_lu_73 input: (None, 14, 14, 512) (None, 14, 14, 512)
ReLU output:
block_7_depthwise_relu (ReLU) (None, 14, 14, 384) 0 ['block_7_depthwise_BN[0][0]']
block_7_project (Conv2D) (None, 14, 14, 64) 24576 ['block_7_depthwise_relu[0][0]']
block_7_project_BN (BatchNorma lization)
conv2d_38 input: (None, 14, 14, 512) (None, 14, 14, 512)
Conv2D output:
block_7_add (Add) (None, 14, 14, 64) 0 ['block_6_project_BN[0][0]', 'block_7_project_BN[0][0]']
block_8_expand (Conv2D) (None, 14, 14, 384) 24576 ['block_7_add[0][0]']
block_8_expand_BN (BatchNormal ization)
batch_normalization_74 input: (None, 14, 14, 512) (None, 14, 14, 512)
BatchNormalization output:
block_8_expand_relu (ReLU) (None, 14, 14, 384) 0 ['block_8_expand_BN[0][0]']
block_8_depthwise (DepthwiseCo nv2D) (None, 14, 14, 384) 3456 ['block_8_expand_relu[0][0]']
block_8_depthwise_BN (BatchNor malization)
re_lu_74 input: (None, 14, 14, 512) (None, 14, 14, 512)
ReLU output:
block_8_depthwise_relu (ReLU) (None, 14, 14, 384) 0 ['block_8_depthwise_BN[0][0]']
block_8_project (Conv2D) (None, 14, 14, 64) 24576 ['block_8_depthwise_relu[0][0]']
block_8_project_BN (BatchNorma lization)
depthwise_conv2d_36 input: (None, 14, 14, 512) (None, 14, 14, 512)
DepthwiseConv2D output:
block_8_add (Add) (None, 14, 14, 64) 0 ['block_7_add[0][0]', 'block_8_project_BN[0][0]']
block_9_expand (Conv2D) (None, 14, 14, 384) 24576 ['block_8_add[0][0]']
block_9_expand_BN (BatchNormal ization)
batch_normalization_75 input: (None, 14, 14, 512) (None, 14, 14, 512)
BatchNormalization output:
block_9_expand_relu (ReLU) (None, 14, 14, 384) 0 ['block_9_expand_BN[0][0]']
block_9_depthwise (DepthwiseCo nv2D) (None, 14, 14, 384) 3456 ['block_9_expand_relu[0][0]']
re_lu_75 input: (None, 14, 14, 512) (None, 14, 14, 512)
ReLU output:

```
block_9_depthwise_BN (BatchNor  (None, 14, 14, 384)  1536       ['block_9_depthwise[0][0]']
malization)
block_9_depthwise_relu (ReLU)  (None, 14, 14, 384)  0          ['block_9_depthwise_BN[0][0]']
block_9_project (Conv2D)       (None, 14, 14, 64)   24576      ['block_9_depthwise_relu[0][0]']
block_9_project_BN (BatchNorma  (None, 14, 14, 64)  256        ['block_9_project[0][0]']
lization)
block_9_add (Add)              (None, 14, 14, 64)   0          ['block_8_add[0][0]',
                                                                'block_9_project_BN[0][0]']
block_10_expand (Conv2D)       (None, 14, 14, 384)  24576      ['block_9_add[0][0]']
block_10_expand_BN (BatchNorma  (None, 14, 14, 384)  1536      ['block_10_expand[0][0]']
lization)
block_10_expand_relu (ReLU)    (None, 14, 14, 384)  0          ['block_10_expand_BN[0][0]']
block_10_depthwise (DepthwiseC  (None, 14, 14, 384)  3456      ['block_10_expand_relu[0][0]']
onv2D)
block_10_depthwise_BN (BatchNo  (None, 14, 14, 384)  1536      ['block_10_depthwise[0][0]']
rmalization)
block_10_depthwise_relu (ReLU)  (None, 14, 14, 384)  0        ['block_10_depthwise_BN[0][0]']
block_10_project (Conv2D)      (None, 14, 14, 96)   36864      ['block_10_depthwise_relu[0][0]']
block_10_project_BN (BatchNorm  (None, 14, 14, 96)  384        ['block_10_project[0][0]']
alization)
block_11_expand (Conv2D)       (None, 14, 14, 576)  55296      ['block_10_project_BN[0][0]']
block_11_expand_BN (BatchNorma  (None, 14, 14, 576)  2304      ['block_11_expand[0][0]']
lization)
block_11_expand_relu (ReLU)    (None, 14, 14, 576)  0          ['block_11_expand_BN[0][0]']
block_11_depthwise (DepthwiseC  (None, 14, 14, 576)  5184      ['block_11_expand_relu[0][0]']
onv2D)
block_11_depthwise_BN (BatchNo  (None, 14, 14, 576)  2304      ['block_11_depthwise[0][0]']
rmalization)
```


block_11_depthwise_relu (ReLU) (None, 14, 14, 576) 0 ['block_11_depthwise_BN[0][0]']
block_11_project (Conv2D) (None, 14, 14, 96) 55296 ['block_11_depthwise_relu[0][0]']
batch_normalization_78
input:
BatchNormalization
output:
(None, 7, 7, 1024)
(None, 7, 7, 1024)
block_11_project_BN (BatchNorm alization) (None, 14, 14, 96) 384 ['block_11_project[0][0]']
block_11_add (Add) (None, 14, 14, 96) 0 ['block_10_project_BN[0][0]', 'block_11_project_BN[0][0]']
re_lu_78
input:
ReLU
output:
(None, 7, 7, 1024)
(None, 7, 7, 1024)
block_12_expand (Conv2D) (None, 14, 14, 576) 55296 ['block_11_add[0][0]']
block_12_expand_BN (BatchNorma lization) (None, 14, 14, 576) 2304 ['block_12_expand[0][0]']
block_12_expand_relu (ReLU) (None, 14, 14, 576) 0 ['block_12_expand_BN[0][0]']
depthwise_conv2d_38
input:
DepthwiseConv2D
output:
(None, 7, 7, 1024)
(None, 7, 7, 1024)
block_12_depthwise (DepthwiseC onv2D) (None, 14, 14, 576) 5184 ['block_12_expand_relu[0][0]']
block_12_depthwise_BN (BatchNo rmalization) (None, 14, 14, 576) 2304 ['block_12_depthwise[0][0]']
block_12_depthwise_relu (ReLU) (None, 14, 14, 576) 0 ['block_12_depthwise_BN[0][0]']
batch_normalization_79
input:
BatchNormalization
output:
(None, 7, 7, 1024)
(None, 7, 7, 1024)
block_12_project (Conv2D) (None, 14, 14, 96) 55296 ['block_12_depthwise_relu[0][0]']
block_12_project_BN (BatchNorm alization) (None, 14, 14, 96) 384 ['block_12_project[0][0]']
re_lu_79
input:
ReLU
output:
(None, 7, 7, 1024)
(None, 7, 7, 1024)
block_12_add (Add) (None, 14, 14, 96) 0 ['block_11_add[0][0]', 'block_12_project_BN[0][0]']
block_13_expand (Conv2D) (None, 14, 14, 576) 55296 ['block_12_add[0][0]']
block_13_expand_BN (BatchNorma lization) (None, 14, 14, 576) 2304 ['block_13_expand[0][0]']
conv2d_41
input:
Conv2D
output:
(None, 7, 7, 1024)
(None, 7, 7, 1024)
block_13_expand_relu (ReLU) (None, 14, 14, 576) 0 ['block_13_expand_BN[0][0]']
block_13_pad (ZeroPadding2D) (None, 15, 15, 576) 0 ['block_13_expand_relu[0][0]']
block_13_depthwise (DepthwiseC onv2D) (None, 7, 7, 576) 5184 ['block_13_pad[0][0]']
batch_normalization_80
input:




```
block_13_depthwise_BN (BatchNo  (None, 7, 7, 576)   2304        ['block_13_depthwise[0][0]']
rmalization)

block_13_depthwise_relu (ReLU)  (None, 7, 7, 576)   0           ['block_13_depthwise_BN[0][0]']

block_13_project (Conv2D)       (None, 7, 7, 160)   92160       ['block_13_depthwise_relu[0][0]']

block_13_project_BN (BatchNorm  (None, 7, 7, 160)   640         ['block_13_project[0][0]']
alization)

block_14_expand (Conv2D)        (None, 7, 7, 960)   153600      ['block_13_project_BN[0][0]']

block_14_expand_BN (BatchNorma  (None, 7, 7, 960)   3840        ['block_14_expand[0][0]']
lization)

block_14_expand_relu (ReLU)     (None, 7, 7, 960)   0           ['block_14_expand_BN[0][0]']

block_14_depthwise (DepthwiseC  (None, 7, 7, 960)   8640        ['block_14_expand_relu[0][0]']
onv2D)

block_14_depthwise_BN (BatchNo  (None, 7, 7, 960)   3840        ['block_14_depthwise[0][0]']
rmalization)

block_14_depthwise_relu (ReLU)  (None, 7, 7, 960)   0           ['block_14_depthwise_BN[0][0]']

block_14_project (Conv2D)       (None, 7, 7, 160)   153600      ['block_14_depthwise_relu[0][0]']

block_14_project_BN (BatchNorm  (None, 7, 7, 160)   640         ['block_14_project[0][0]']
alization)

block_14_add (Add)              (None, 7, 7, 160)   0           ['block_13_project_BN[0][0]',
                                                                 'block_14_project_BN[0][0]']

block_15_expand (Conv2D)        (None, 7, 7, 960)   153600      ['block_14_add[0][0]']

block_15_expand_BN (BatchNorma  (None, 7, 7, 960)   3840        ['block_15_expand[0][0]']
lization)

block_15_expand_relu (ReLU)     (None, 7, 7, 960)   0           ['block_15_expand_BN[0][0]']

block_15_depthwise (DepthwiseC  (None, 7, 7, 960)   8640        ['block_15_expand_relu[0][0]']
onv2D)

block_15_depthwise_BN (BatchNo  (None, 7, 7, 960)   3840        ['block_15_depthwise[0][0]']
```

```
rmalization)

block_15_depthwise_relu (ReLU)  (None, 7, 7, 960)   0           ['block_15_depthwise_BN[0][0]']

block_15_project (Conv2D)       (None, 7, 7, 160)   153600      ['block_15_depthwise_relu[0][0]']

block_15_project_BN (BatchNorm  (None, 7, 7, 160)   640         ['block_15_project[0][0]']
alization)

block_15_add (Add)              (None, 7, 7, 160)   0           ['block_14_add[0][0]',
                                                                 'block_15_project_BN[0][0]']

block_16_expand (Conv2D)        (None, 7, 7, 960)   153600      ['block_15_add[0][0]']

block_16_expand_BN (BatchNorma  (None, 7, 7, 960)   3840        ['block_16_expand[0][0]']
lization)

block_16_expand_relu (ReLU)     (None, 7, 7, 960)   0           ['block_16_expand_BN[0][0]']

block_16_depthwise (DepthwiseC  (None, 7, 7, 960)   8640        ['block_16_expand_relu[0][0]']
onv2D)

block_16_depthwise_BN (BatchNo  (None, 7, 7, 960)   3840        ['block_16_depthwise[0][0]']
rmalization)

block_16_depthwise_relu (ReLU)  (None, 7, 7, 960)   0           ['block_16_depthwise_BN[0][0]']

block_16_project (Conv2D)       (None, 7, 7, 320)   307200      ['block_16_depthwise_relu[0][0]']

block_16_project_BN (BatchNorm  (None, 7, 7, 320)   1280        ['block_16_project[0][0]']
alization)

Conv_1 (Conv2D)                 (None, 7, 7, 1280)  409600      ['block_16_project_BN[0][0]']

Conv_1_bn (BatchNormalization)  (None, 7, 7, 1280)  5120        ['Conv_1[0][0]']

out_relu (ReLU)                 (None, 7, 7, 1280)  0           ['Conv_1_bn[0][0]']

global_average_pooling2d_8 (Gl  (None, 1280)        0           ['out_relu[0][0]']
obalAveragePooling2D)

dense_15 (Dense)                (None, 128)         163968      ['global_average_pooling2d_8[0][0
                                                                ]']

dense_16 (Dense)                (None, 1)           129         ['dense_15[0][0]']
```

```
=================================================================================================
Total params: 2,422,081
Trainable params: 166,657
Non-trainable params: 2,255,424
_________________________________________________________________________________________________
```

Defining callbacks

- The checkpoint callback saves the best weights of the model, so next time we want to use the model,
- we do not have to spend time training it. -The early stopping callback stops the training process when the model starts becoming stagnant, or even worse, when the model starts overfitting.

In [ ]:
```
# from google.colab import drive
# drive.mount('/content/gdrive/',force_remount=True)
```

In [ ]:
```
checkpoint_cb = tf.keras.callbacks.ModelCheckpoint("xray_model.h5", save_best_only=True)

early_stopping_cb = tf.keras.callbacks.EarlyStopping(
    patience=3, restore_best_weights=True
)
```

We also want to tune our learning rate.

- Too high of a learning rate will cause the model to diverge.
- Too small of a learning rate will cause the model to be too slow. We implement the exponential learning rate scheduling method below.

In [ ]:
```
def exponential_decay(lr0, s):
    def exponential_decay_fn(epoch):
        return lr0 * 0.1 **(epoch / s)
    return exponential_decay_fn

exponential_decay_fn = exponential_decay(0.01, 20)

lr_scheduler = tf.keras.callbacks.LearningRateScheduler(exponential_decay_fn)
```

In [ ]:
```
initial_learning_rate = 0.01

lr_schedule = tf.keras.optimizers.schedules.ExponentialDecay(
```

```
    initial_learning_rate, decay_steps=100000, decay_rate=0.96, staircase=True
)

lr_scheduler = tf.keras.callbacks.LearningRateScheduler(exponential_decay_fn)
```

Training the model:

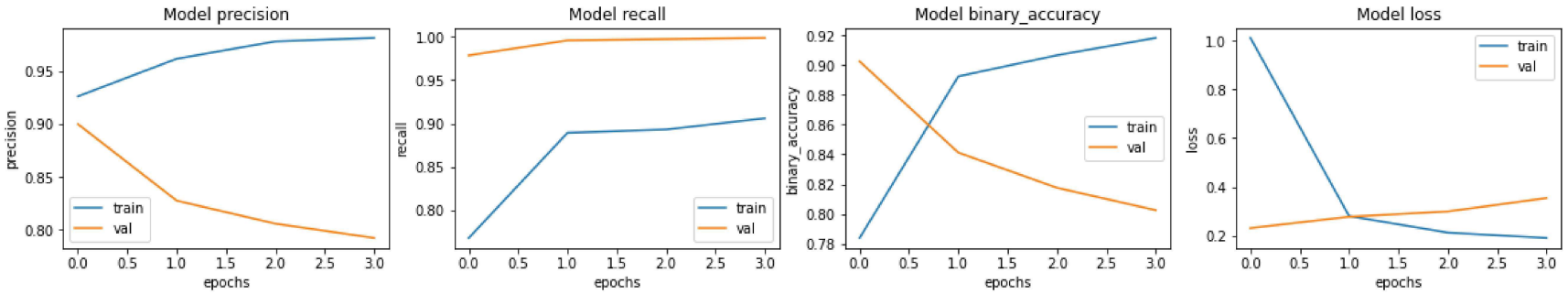
In [ ]:
```
with strategy.scope():
  finetuned_mobilenet = build_model()
  METRICS = [
  tf.keras.metrics.BinaryAccuracy(),
  tf.keras.metrics.Precision(name="precision"),
  tf.keras.metrics.Recall(name="recall"),
  ]
  finetuned_mobilenet.compile(
  optimizer=tf.keras.optimizers.Adam(),
  loss="binary_crossentropy",
  metrics=METRICS,
  )

  history = finetuned_mobilenet.fit(
  train_ds_batch,
  epochs = 10,
  validation_data = val_ds_batch,
  class_weight = class_weight,
  callbacks=[checkpoint_cb,early_stopping_cb,lr_scheduler]
  )
```

```
Epoch 1/10
17/17 [==============================] - 46s 2s/step - loss: 1.0123 - binary_accuracy: 0.7840 - precision: 0.9262 - rec
all: 0.7681 - val_loss: 0.2306 - val_binary_accuracy: 0.9024 - val_precision: 0.8999 - val_recall: 0.9785 - lr: 0.0100
Epoch 2/10
17/17 [==============================] - 3s 153ms/step - loss: 0.2799 - binary_accuracy: 0.8923 - precision: 0.9618 - r
ecall: 0.8892 - val_loss: 0.2771 - val_binary_accuracy: 0.8412 - val_precision: 0.8274 - val_recall: 0.9957 - lr: 0.008
9
Epoch 3/10
17/17 [==============================] - 4s 227ms/step - loss: 0.2120 - binary_accuracy: 0.9065 - precision: 0.9782 - r
ecall: 0.8930 - val_loss: 0.2991 - val_binary_accuracy: 0.8176 - val_precision: 0.8056 - val_recall: 0.9971 - lr: 0.007
9
Epoch 4/10
17/17 [==============================] - 6s 393ms/step - loss: 0.1902 - binary_accuracy: 0.9181 - precision: 0.9815 - r
ecall: 0.9060 - val_loss: 0.3540 - val_binary_accuracy: 0.8026 - val_precision: 0.7920 - val_recall: 0.9986 - lr: 0.007
1
```

Plotting model performance

Let's plot the model accuracy and loss for the training and the validating set. Note that no random seed is specified for this notebook. For your notebook, there might be slight variance.

In [ ]:
```
fig, ax = plt.subplots(1, 4, figsize=(20, 3))
ax = ax.ravel()

for i, met in enumerate(["precision", "recall", "binary_accuracy", "loss"]):
    ax[i].plot(history.history[met])
    ax[i].plot(history.history["val_" + met])
    ax[i].set_title("Model {}".format(met))
    ax[i].set_xlabel("epochs")
    ax[i].set_ylabel(met)
    ax[i].legend(["train", "val"])
```



Loading the Model

In [ ]:
```
loaded_mobilenet = tf.keras.models.load_model('xray_model.h5')
```

Evaluate model

Let's evaluate the model on our test data!

In [ ]:
```
loaded_mobilenet.evaluate(test_ds_batch, return_dict=True)
```

```
3/3 [==============================] - 3s 437ms/step - loss: 0.6547 - binary_accuracy: 0.6939 - precision: 0.6749 - rec
all: 0.9846
```

Out[ ]:
```
{'loss': 0.6546863913536072,
 'binary_accuracy': 0.6939102411270142,
 'precision': 0.6748681664466858,
 'recall': 0.9846153855323792}
```

- We see that our accuracy on our test data is 69%, which is lower than the accuracy for our validating set. This may indicate overfitting. ### Try **finetuning the model** further to decrease overfitting to the training and validation sets.
- [Refer lecture 2 to deal with overfitting.]
- Our recall is greater than our precision, indicating that almost all pneumonia images are correctly identified but some normal images are falsely identified.
- We should aim to increase our precision.
  - link to refer for solutions:
    1. https://stats.stackexchange.com/questions/134599/how-to-classify-a-unbalanced-dataset-by-convolutional-neural-networks-cnn
    2. https://www.tensorflow.org/tutorials/structured_data/imbalanced_data

Predicting on a test image using our trained MobileNet Model

In [ ]:
```
for image, label in test_ds_batch.take(2):
      plt.imshow(image[2] / 255.0)
      plt.title(CLASS_NAMES[label[2].numpy()])
      plt.axis('off')
prediction = finetuned_mobilenet.predict(test_ds_batch.take(2))[2]
scores = [1 - prediction, prediction]
for score, name in zip(scores, CLASS_NAMES):
    print("This image is %.2f percent %s" % ((100 * score), name))
%time
```

```
This image is 34.41 percent NORMAL
This image is 65.59 percent PNEUMONIA
CPU times: user 4 µs, sys: 1e+03 ns, total: 5 µs
Wall time: 10.5 µs
```

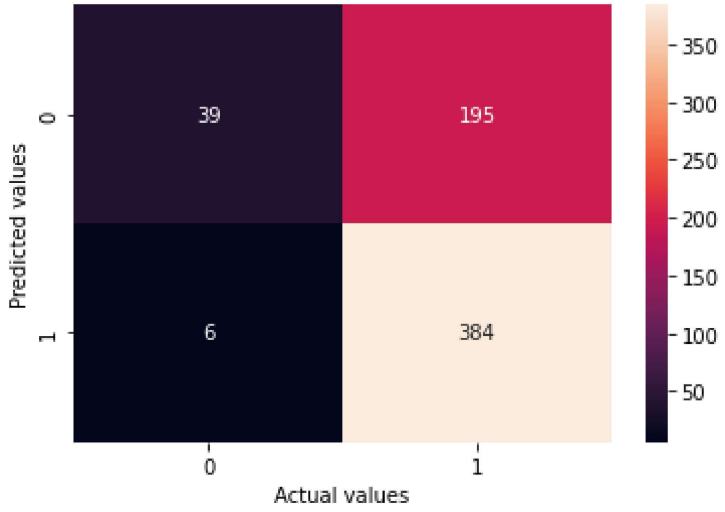
PNEUMONIA

Confusion Matrix

In [ ]:
```
label_list = []
prediction_list = []

for image, label in test_ds:
    image = tf.expand_dims(image, axis = 0)
    prediction = finetuned_mobilenet.predict(image)
    label = tf.where(label, 1, 0)

    if prediction > 0.5 :
        prediction = 1
    else :
        prediction = 0

    prediction_list.append(prediction)
    label_list.append(label)
```

In [ ]:
```
test_confusion_matrix = tf.math.confusion_matrix(label_list, prediction_list)
```

In [ ]:
```
ax = sns.heatmap(test_confusion_matrix, annot = True, fmt ='g')
ax.set(xlabel = 'Predicted values', ylabel = 'Actual values')

plt.show()
```



Can we figure out which part of the image our model focuses on to get a prediction ?

- We, **as humans**, when we try to classify an image.
- **we look at certain region in the image to make our judgement.**
- **For eg:** In case of cats & dog classification, we'll focus on face of the animal in the picture.
- In this case, i.e. pneunomnia classification, **the domain expert or doctor look at the specific region of the xray in the frame.**

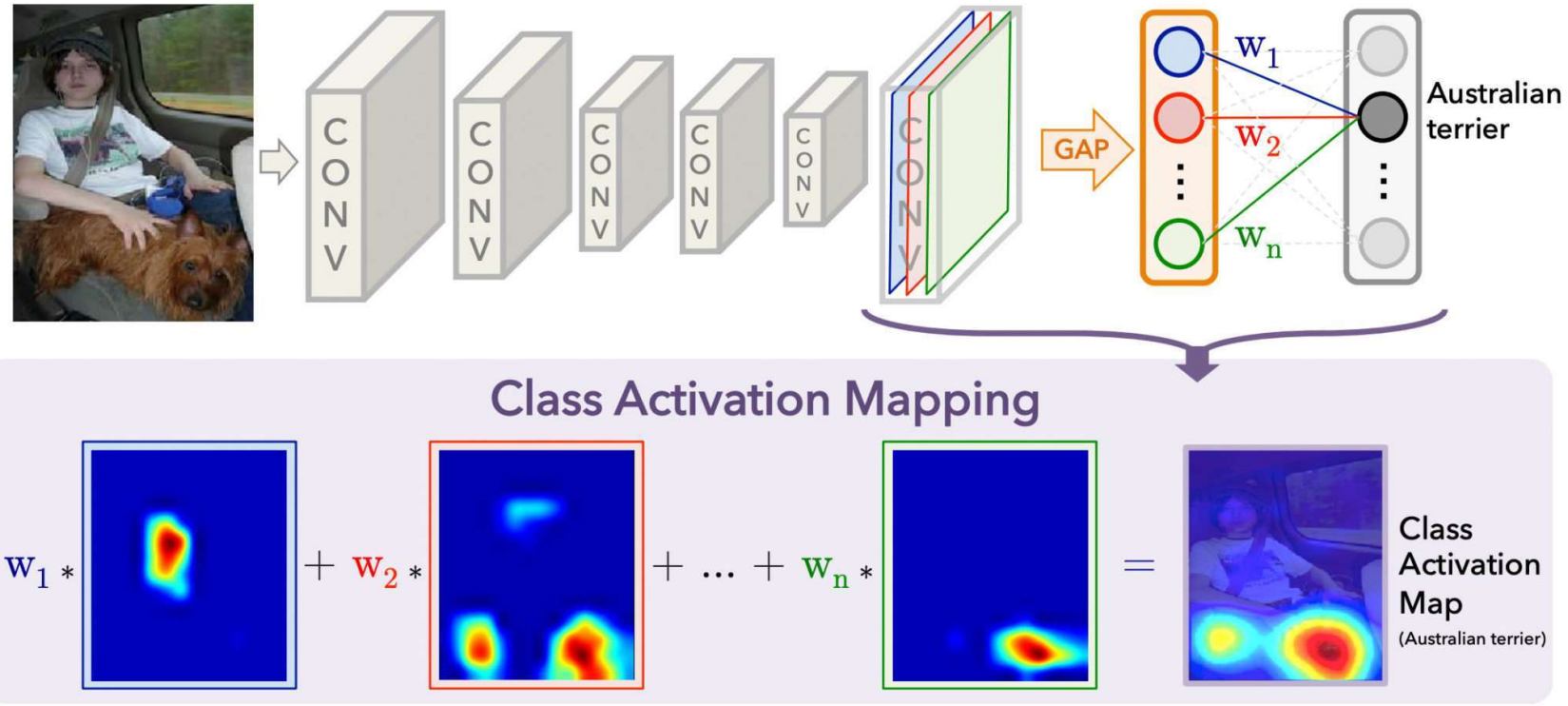
How can we check what region in the image our CNN is focusing on ?

- With the help of **GradCAM(Gradient-weighted Class Activation Map**) algorithm.
- It helps to find out the region on which **CNN is focusing on to predict particular class.**

GRADCAM (Post Read)

- GradCAM is an **improvement** over **CAM** in `versatility and accuracy`.
- It is complex but, luckily, **the output is intuitive.**
- **STEPS OF GRADCAM ALGO:**

1. Take an **image** as input
2. **Create a model** that is `cut off at the layer for which we want to create a GRAD-CAM heat-map`.
3. We attach the **fully-connected layers** for `prediction`.
4. We then run the input through the model, **grab the layer output, and loss**.
5. Next, we find the **gradient of the output of our desired model layer w.r.t. the model loss**.
6. From there, `we take sections of the gradient which contribute to the prediction`, reduce, resize, and rescale so that the heat-map can be overlaid with the original image.



Since we need the last Convolution layer for implementing our GRADCAM Algorithm, </br> we print every Conv. layer name in our pretrained_modbilenet model

In [ ]:
```
for i in range(len(finetuned_mobilenet.layers)):
    layer = finetuned_mobilenet.layers[i]
    if 'conv' not in layer.name:
        continue
    print(i , layer.name , layer.output.shape)
```

```
4 expanded_conv_depthwise (None, 112, 112, 32)
5 expanded_conv_depthwise_BN (None, 112, 112, 32)
6 expanded_conv_depthwise_relu (None, 112, 112, 32)
7 expanded_conv_project (None, 112, 112, 16)
8 expanded_conv_project_BN (None, 112, 112, 16)
```

Last convolutional layer for GRADCAM is expanded_conv_project_BN

In [ ]:
```
last_conv_layer_name = finetuned_mobilenet.layers[8].name
last_conv_layer_name
```

Out[ ]:
```
'expanded_conv_project_BN'
```

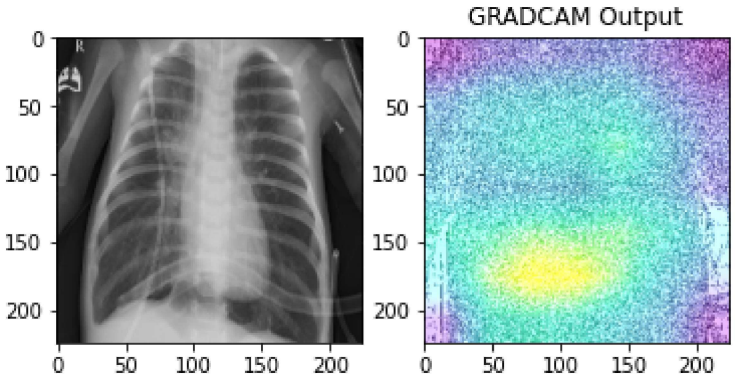
In [ ]:
```
!pip install tf-explain
```

```
Looking in indexes: https://pypi.org/simple, https://us-python.pkg.dev/colab-wheels/public/simple/
Requirement already satisfied: tf-explain in /usr/local/lib/python3.7/dist-packages (0.3.1)
```

In [ ]:
```
from tf_explain.core.grad_cam import GradCAM
import random

for image, label in test_ds_batch.take(2):

    data = ([image[2].numpy()], None)
    # print('Original label',CLASS_NAMES[label[2].numpy()])
    image = image[2] / 255

prediction = finetuned_mobilenet.predict(test_ds_batch.take(2))[2]
scores = [1 - prediction, prediction]

for score, name in zip(scores, CLASS_NAMES):
    print("This image is %.2f percent %s" % ((100 * score), name))

# Start explainer
explainer = GradCAM()
grid = explainer.explain(data, finetuned_mobilenet,class_index = 0)

#subplot(r,c) provide the no. of rows and columns
f, axarr = plt.subplots(1, 2)

axarr[0].imshow(image)
axarr[1].imshow(grid)
# axarr[0].title.set_text(f"{float(prediction)*100:.2f}% penumonia predicted")
```

```
axarr[1].title.set_text('GRADCAM Output')
plt.show()
```

```
This image is 34.41 percent NORMAL
This image is 65.59 percent PNEUMONIA
```



Question :

```
What is GRADCAM ?

A) Colourises our black and white images
B) Plots the part of input where model focuses
C) Is used for xray images specifically
D) Used for model evaluation
```

Answer :

B)

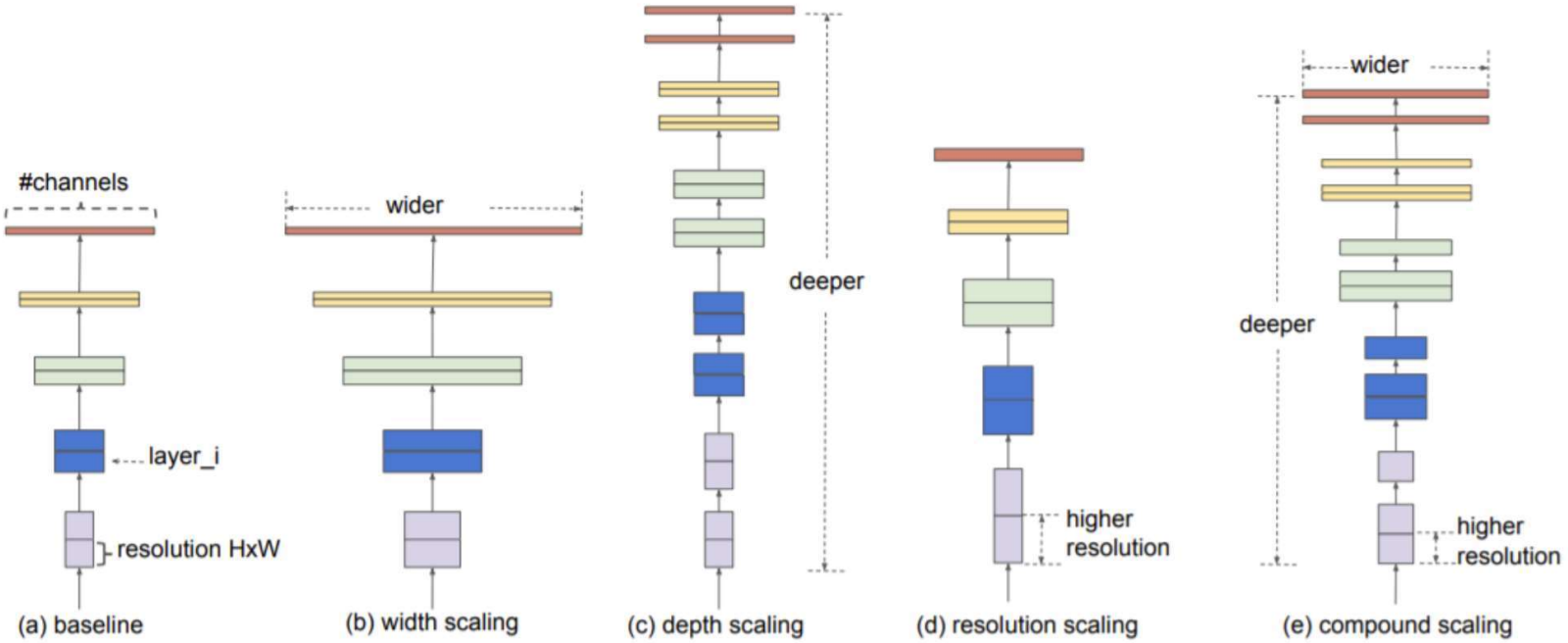
GRADCAM :

- Gradient-weighted Class Activation Mapping (Grad-CAM)
- Used to visualize which part of our input the model focuses while prediction
- Makes a heatmap around the focused area(s)
- uses the gradients of any target image flowing into the final convolutional layer to produce a heatmap highlighting the important regions in the image

Rethinking Model Design (Width, Depth, Resolution):

Now that we have studied about MobileNet and its working lets scale things up.

- Generally, when we design an architecture, we think in three dimensions
  - Width Scaling : Increasing number of features in conv layers
  - Depth Scaling : Increasing number of layers
  - Resolution Scaling : Adjusting Input shape
- ResNet / MobileNet optimizes one dimension at a time
- Couldn't we search for the ideal combination of all three dimsion at once
- Just like we perform hyperparameter tuning in desion tree, chosing tree depth, min_samples_leaf, min_samples_split, etc?
- a brute-force search through the entire set of possible operations would be a very large a task.
- Next model (EfficientNet) has simplified the problem in a clever way.



Intro to EfficientNet

- Original Paper: https://arxiv.org/abs/1905.11946
- **EfficientNet** is actually a family of neural networks of different sizes, where a lot of attention was paid to the scaling of the networks in the family.
- Convolutional archi- tectures have three main ways of scaling:
  - Use more layers.
  - Use more channels in each layer.
  - Use higher-resolution input images.
- The EfficientNet paper points out that these three scaling axes are not independent:
  - "If the input image is bigger, then the network needs more layers to increase the receptive field and more channels to capture more fine-grained patterns on the bigger image."
- The novelty in the EfficientNetB0 through EfficientNetB7 family of neural networks is that they are scaled along all three scaling axes rather than just one, as was the case in earlier architecture families such as ResNet50/ResNet101/ResNet152.
- The Effi- cientNet family is today the workhorse of many applied machine learning teams because it offers optimal performance levels for every weight count.

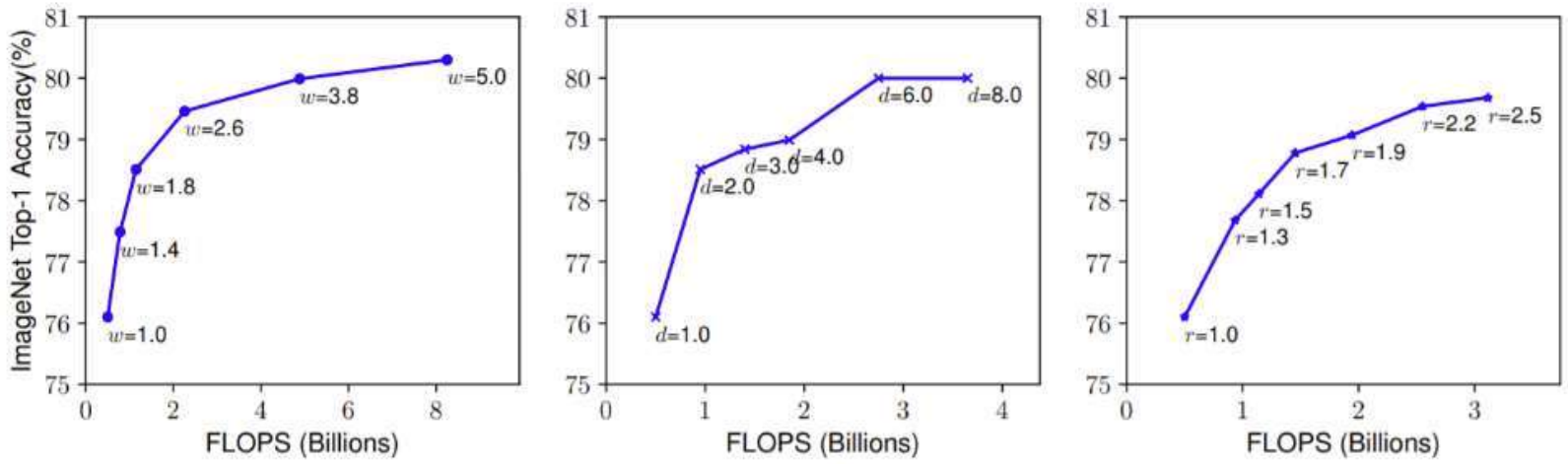
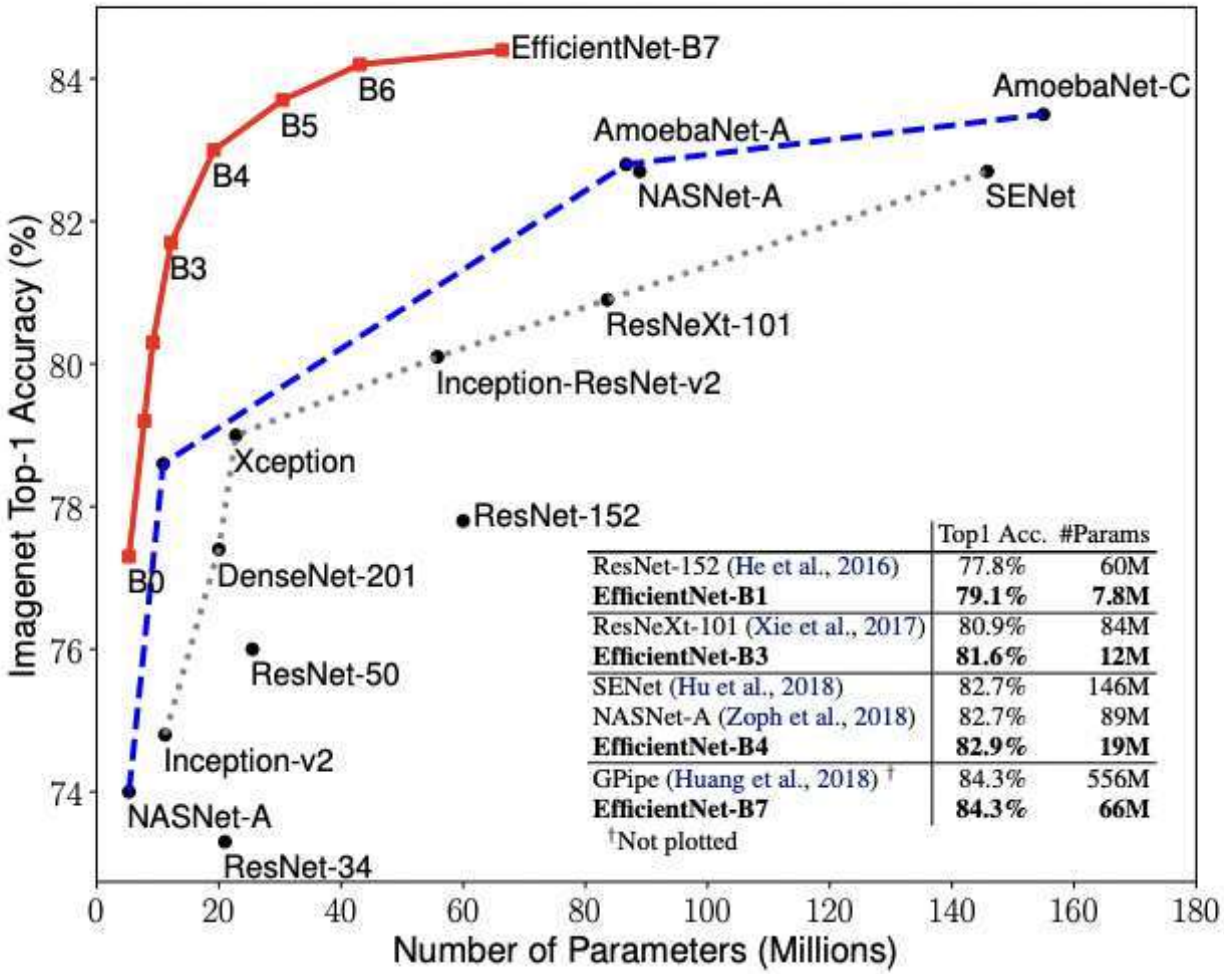


Figure 3. **Scaling Up a Baseline Model with Different Network Width (w), Depth (d), and Resolution (r) Coefficients.** Bigger networks with larger width, depth, or resolution tend to achieve higher accuracy, but the accuracy gain quickly saturate after reaching 80%, demonstrating the limitation of single dimension scaling. Baseline network is described in Table 1.

- EfficientNet models generally use an order of magnitude fewer parameters and FLOPS than other ConvNets with similar accuracy

- They Came up with eights models B0 to B7 (with increasing size)

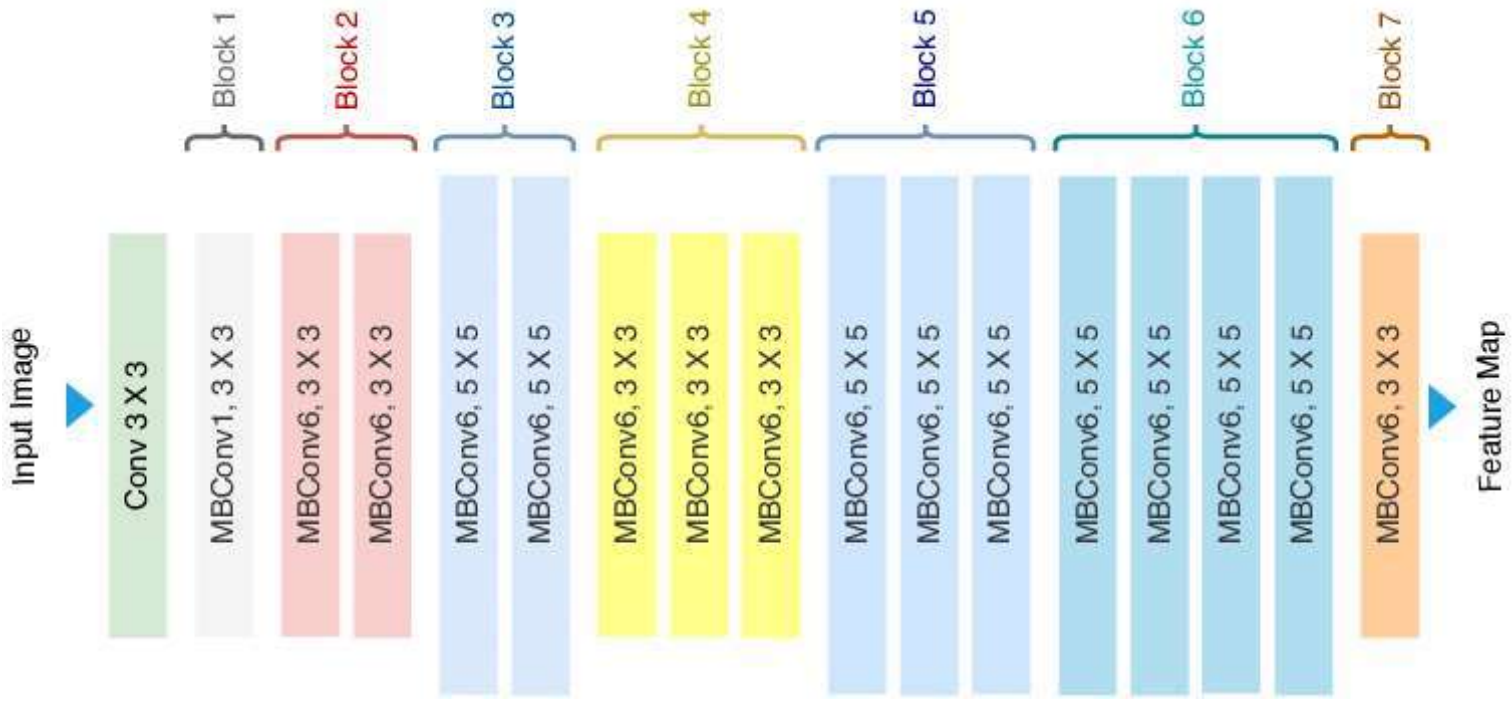


	Top1 Acc.	#Params
ResNet-152 (He et al., 2016)	77.8%	60M
EfficientNet-B1	**79.1%**	**7.8M**
ResNeXt-101 (Xie et al., 2017)	80.9%	84M
EfficientNet-B3	**81.6%**	**12M**
SENet (Hu et al., 2018)	82.7%	146M
NASNet-A (Zoph et al., 2018)	82.7%	89M
EfficientNet-B4	**82.9%**	**19M**
GPipe (Huang et al., 2018) †	84.3%	556M
EfficientNet-B7	**84.3%**	**66M**

†Not plotted

Figure 1. **Model Size vs. ImageNet Accuracy.** All numbers are for single-crop, single-model. Our EfficientNets significantly outperform other ConvNets. In particular, EfficientNet-B7 achieves new state-of-the-art 84.3% top-1 accuracy but being 8.4x smaller and 6.1x faster than GPipe. EfficientNet-B1 is 7.6x smaller and 5.7x faster than ResNet-152. Details are in Table 2 and 4.

- For example, **EfficientNet-B3 achieves higher accuracy than ResNet101 using 18x fewer FLOPS.**
- Full efficientnet code from scracth:
  https://github.com/qubvel/efficientnet/blob/f7f3e736c113b872caf53dae9fbbda996a8eb87d/efficientnet/model.py



Loading pretrained efficientnet:

https://keras.io/api/applications/efficientnet/

In [ ]:
```
with strategy.scope():
    pretrained_model = tf.keras.applications.EfficientNetB6(weights='imagenet', include_top=False, input_shape=[*IMAGE_
    pretrained_model.trainable = True # fine-tuning

    model = tf.keras.Sequential([
        # convert image format from int [0,255] to the format expected by this model
        tf.keras.layers.Lambda(lambda data: tf.keras.applications.efficientnet.preprocess_input(tf.cast(data, tf.float3
        pretrained_model,
        tf.keras.layers.GlobalAveragePooling2D(),
```

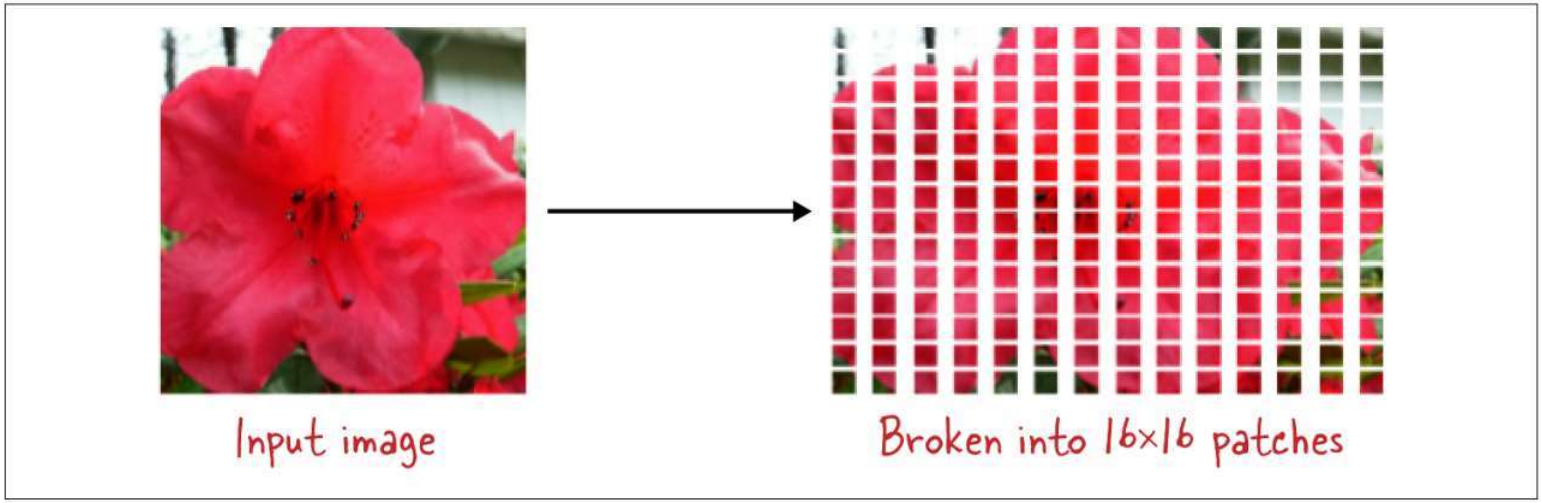
```
        tf.keras.layers.Dense(2, activation='softmax')
    ])
```

Beyond Convolution: The Transformer Architecture:[Optional to read]

- The architectures for computer vision that are discussed till now all rely on convolutional filters.
- Compared to the naive dense neural networks discussed in lecture 1, convolutional filters reduce the number of weights necessary to learn how to extract information from images.
- However, as dataset sizes keep increasing, there comes a point where this weight reduction is no longer necessary.
- Ashish Vaswani et al. proposed the Transformer architecture for natural language processing in a 2017 paper with the catchy title **"Attention Is All You Need."**
- As the title indicates, the key innovation in the Transformer architecture is the concept of attention—having the model focus on some part of the input text sequence when pre- dicting each word.
- For example, consider a model that needs to translate the French phrase "ma chemise rouge" into English ("my red shirt").
  - The model would learn to focus on the word rouge when predicting the second word of the English translation, red.
- The Transformer model achieves this by using positional encodings.
- Instead of simply representing the input phrase by its words, it adds the position of each word as an input: (ma, 1), (chemise, 2), (rouge, 3).
- The model then learns from the training dataset which word of the input it needs to focus on when predicting a specific word of the output.

- **The Vision Transformer (ViT) model** adapts the Transformer idea to work on images.
- The equivalent of words in images are square patches, so the first step is to take the input image and break it into patches,

In [ ]:
```
patches = tf.image.extract_patches(
        images=images,
        sizes=[1, self.patch_size, self.patch_size, 1],
        strides=[1, self.patch_size, self.patch_size, 1],
        rates=[1, 1, 1, 1],
        padding="VALID",
)
```



- Here if you observed The input image is broken into patches that are treated as the sequence input to the Transformer.
- The patches are represented by concatenating the patch pixel values and the patch position within the image:

In [ ]:
```
encoded = (tf.keras.layers.Dense(...)(patch) +
               tf.keras.layers.Embedding(...)(position))
```

- Note that the patch position is the ordinal number (5th, 6th, etc.) of the patch and is treated as a categorical variable.
- A learnable embedding is employed to capture close- ness relationships between patches that have related content.
- The patch representation is passed through multiple transformer blocks, each of which consists of an attention head (to learn which parts of the input to focus on)

In [ ]:
```
x1 = tf.keras.layers.LayerNormalization()(encoded)
#Just for sake of basic overview
attention_output = tf.keras.layers.MultiHeadAttention(
    num_heads=num_heads, key_dim=projection_dim, dropout=0.1
)(x1, x1)
```

The attention output is used to add emphasis to the patch representation:

In [ ]:
```
# Skip connection 1.
x2 = tf.keras.layers.Add()([attention_output, encoded])
```

```
x3 = tf.keras.layers.LayerNormalization()(x2)# Layer normalization 2.
```

and passed through a set of dense layers:

In [ ]:
```
# multilayer perceptron (mlp), a set of dense layers.
x3 = mlp(x3, hidden_units=transformer_units,
         dropout_rate=0.1)
# Skip connection 2 forms input to next block
encoded = tf.keras.layers.Add()([x3, x2])
```

- The training loop is similar to that of any of the convolutional network architectures as we discussed.
- Note that the ViT architecture requires a lot more data than convolutional network models—the authors suggest pretraining the ViT model on large amounts of data and then fine-tuning on smaller datasets.
- Even though not particularly promising at present for our relatively small dataset, the idea of applying the Transformer architecture to images is interesting, and a potential source of new innovations in computer vision.

We will discuss about transformers and attention in lot more details in NLP module.

Summary of pretrained models:

Model Name	Number of params	Top 1 Acc	Top 5 Acc
EfficientnetB0	5.3M	77.3	93.5
MobileNet	2.3M	71.0	90.5
ResNet50	25.6M	76	93
Inception	11.2M	74.8	92.2
VGG16	138M	74.4	91.9
AlexNet	62M	63.3	84.6

How do you decide which model you should chose on a new dataset?

- If you want to roll your own layers, start with VggNet. It's the simplest model that will perform well.

- For edge devices, you typically want to optimize for models that can be downloaded fast, occupy very little space on the device, and don't incur high latencies during prediction. For a small model that runs fast on low-power devices, con- sider MobileNetV2.
- If you don't have size/speed restrictions (such as if inference will be done on autoscaling cloud systems) and want the best/fanciest model, consider Efficient- Net.
- If you belong to a conservative organization that wants to stick with something tried and true, choose ResNet50 or one of its larger variants. -if training cost and prediction latency are not of concern, or if small improvements in model accuracy bring outside rewards, consider an ensemble of three complementary models.

Points to remember before applying transfer learning:

- Mainly there are two factors

1. size of the new dataset (small or big), and
2. its similarity to the original dataset (e.g. ImageNet-like in terms of the content of images and the classes, or very different)

- New dataset is small and similar to original dataset.
    - Since the data is small, it is not a good idea to fine-tune the ConvNet due to overfitting concerns.
    - data is similar to the original data, we expect higher-level features in the ConvNet to be relevant to this dataset as well.
- New dataset is large and similar to the original dataset.
    - Since we have more data, we can have more confidence that we won't overfit if we were to try to fine-tune through the full network.
- New dataset is large and very different from the original dataset.
    - Since the dataset is very large, we may expect that we can afford to train a ConvNet from scratch.
    - However, in practice it is very often still beneficial to initialize with weights from a pretrained model.

Supplementary material

Other models to follow :-

DenseNet : https://arxiv.org/pdf/1608.06993.pdf

SqueezeNet : https://arxiv.org/pdf/1602.07360.pdf

NASNet : https://arxiv.org/pdf/1707.07012.pdf