Business Problem:

- More than 1 million people are hospitalized with pneumonia, which is a very serious problem
- Chest X-rays are currently the best available method for diagnosing it
- Suppose You are working as a Data Scientist at Qure.ai (a Medical startup) and want to Classify if a person has pneumonia or not.
    - You also have to deploy the model on mobile device for real time inferences
    - Please go through this link to know about the existing apps in medical domain :

      https://www.grantsformedical.com/apps-for-medical-diagnosis.html
- This was especially useful during the times when COVID-19 was known to cause pneumonia.

Normal

Pneumonia

- Image on the left is a normal, but on the right we can see severe glass opacity mainly due to air displacement by fluids

Brief intro:

What is Pneumonia?

- Pneumonia is an infection that inflames the air sacs in one or both lungs.

- The air sacs may fill with fluid or pus (purulent material), causing cough with phlegm or pus, fever, chills, and difficulty breathing.
- A variety of organisms, including bacteria, viruses and fungi, can cause pneumonia.
- Pneumonia can range in seriousness from mild to life-threatening.
- It is most serious for infants and young children, people older than age 65, and people with health problems or weakened immune systems.

Agenda & Motivation:

Computer Vision has a lot of applications in medical diagnosis:

- In this lecture we will explain the complete pipeline from loading data to predicting results, and
- It will explain how to build an X-ray image classification model using CNN to predict whether an X-ray scan shows presence of pneumonia.

Real time constraints:

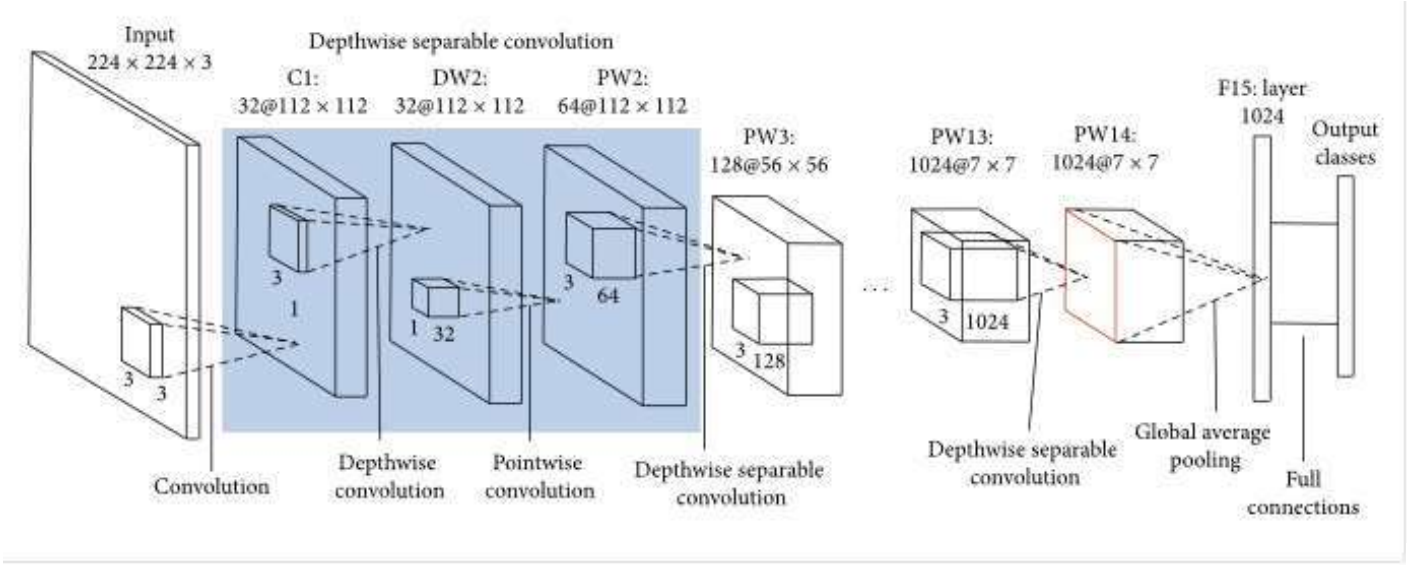
- Low latency requirements
- False negative or positives can be risky
  - If a person has Pneumonia and your model predicted as Normal
  - If a person is Normal and your model predicted as Pneumonia
- Model should be confident in deciding the class
- Model explainability through visualizations

How we are going to solve the problem :

- In previous classes you have studied about state-of-the-art models like VGG16, ResNet, etc.
- Models like these are computationally heavy and cannot be deployed on mobile devices

Which model to use here ?

- We are going to introduce mobilenet in this session
  - MobileNet achieves sky high accuracies and is 100x smaller compared to these models
  - MobileNet is used on mobile apps for object detection, image classficiation, etc. and provide low latency outputs for any use case



MobileNet Architecture

- We will again through the Architecture in detail but before that -
- Lets check our DATA !

In [ ]:
```
# Importing required libraries

import warnings
import numpy as np
import seaborn as sns
import tensorflow as tf
import matplotlib.cm as cm
import matplotlib.pyplot as plt
warnings.filterwarnings('ignore')
```

Here we will also see how to utilize TPUs efficiently-

What are TPUs?

- TPUs are tensor processing units developed by Google to accelerate operations on a Tensorflow Graph.
  - Each TPU packs up to 180 teraflops of floating-point performance and 64 GB of high-bandwidth memory onto a single board.
  - These are Google's custom-developed application-specific integrated circuits (ASICs) used to accelerate machine learning workloads.
- TPUs were only available on Google cloud but now they are available for free in Colab.

In [ ]:
```
#tf.distribute.Strategy is a TensorFlow API to distribute training across multiple GPUs,
#multiple machines, or TPUs. Using this API, you can distribute your existing models and
#training code with minimal code changes.
try:
    tpu = tf.distribute.cluster_resolver.TPUClusterResolver.connect()
    print("Device:", tpu.master())
    strategy = tf.distribute.TPUStrategy(tpu)
except:
    strategy = tf.distribute.get_strategy()
print("Number of replicas:", strategy.num_replicas_in_sync)
```

```
WARNING:tensorflow:TPU system grpc://10.52.223.82:8470 has already been initialized. Reinitializing the TPU can cause p
reviously created variables on TPU to be lost.
Device: grpc://10.52.223.82:8470
Number of replicas: 8
```

- Below, we define key configuration parameters we'll use in this example.
- To run on TPU, this example must be on Colab with the TPU runtime selected.

In [ ]:
```
AUTOTUNE = tf.data.AUTOTUNE

# Specifying our training batch size
BATCH_SIZE = 32 * strategy.num_replicas_in_sync

# Specifying the image size
IMAGE_SIZE = [224, 224]

# List containing class names, which will be used to index on our model output
# 0 = NORMAL, 1 = PNEUMONIA
CLASS_NAMES = ["NORMAL", "PNEUMONIA"]
```

tf.data.AUTOTUNE

- tf.data builds a performance model of the input pipeline
- runs an optimization algorithm to find a good allocation of its CPU budget across all parameters specified as **AUTOTUNE**.
- While the input pipeline is running, tf.data tracks time spent in each operation, so that these times can be fed to the optimization algorithm.

Load the data:

- The Chest X-ray data we are using from *Cell*30154-5) divides the data into training and test files.
- Here we are downloading the data using Google Cloud link

Download dataset using google cloud link:

Here the dataset is present in the TFRecord file format format:

What is TFRecord file format format ?

- A TFRecord file stores your data as a sequence of binary strings.
- It is Tensorflow's own binary storage format.
- Advantages of tf.record format :
  - Binary data takes up less space on disk,
  - takes less time to copy
  - can be read much more efficiently from disk.
  - If you are working with large datasets, using a binary file format for storage of your data can have a significant impact on the performance of your import pipeline

In [ ]:
```
train_images = tf.data.TFRecordDataset(
    "gs://download.tensorflow.org/data/ChestXRay2017/train/images.tfrec"
)
train_paths = tf.data.TFRecordDataset(
    "gs://download.tensorflow.org/data/ChestXRay2017/train/paths.tfrec"
)

ds = tf.data.Dataset.zip((train_images, train_paths))
```

In [ ]:
```
COUNT_NORMAL = len([filename for filename in train_paths if "NORMAL" in filename.numpy().decode("utf-8")])
print("Normal images count in training set: " + str(COUNT_NORMAL))

COUNT_PNEUMONIA = len([filename for filename in train_paths if "PNEUMONIA" in filename.numpy().decode("utf-8")])
print("Pneumonia images count in training set: " + str(COUNT_PNEUMONIA))

print('Total Count of images:', COUNT_NORMAL+COUNT_PNEUMONIA )
```

```
Normal images count in training set: 1349
Pneumonia images count in training set: 3883
Total Count of images: 5232
```

- Notice that there are way more images that are classified as pneumonia than normal.
- This shows that we have an **imbalance** in our data i.e our model can be biased towards high majority class (Pneumonia in our case)

How we are going to solve the class imbalance here ?

Can we apply augmentation here ?

- Here Data augmentation will not be useful because X-ray scans are only taken in a specific orientation, and variations such as flips and rotations will not exist in real X-ray images.
- We will correct for this imbalance later using class weights

In [ ]:
```
TRAIN_IMG_COUNT = COUNT_NORMAL + COUNT_PNEUMONIA
# Scaling by total/2 helps keep the loss to a similar magnitude.
# The sum of the weights of all examples stays the same.

weight_for_0 = (1 / COUNT_NORMAL) * (TRAIN_IMG_COUNT)  / 2.0
weight_for_1 = (1 / COUNT_PNEUMONIA) * (TRAIN_IMG_COUNT) / 2.0

class_weight = {0: weight_for_0, 1: weight_for_1}

print("Weight for class 0: {:.2f}".format(weight_for_0))
print("Weight for class 1: {:.2f}".format(weight_for_1))
```

```
Weight for class 0: 1.94
Weight for class 1: 0.67
```

- The weight for class 0 (Normal) is a lot higher than the weight for class 1 (Pneumonia).
- Because there are less normal images, each normal image will be weighted more to balance the data as the CNN works best when the training data is balanced.

Creating (Image, Label) pair :

- We want to map each filename to the corresponding (image, label) pair. The following methods will help us do that.

- As we only have two labels, we will encode the label so that `1` or `True` indicates pneumonia and `0` or `False` indicates normal.

In [ ]:
```
def get_label(file_path):
    # convert the path to a list of path components
    parts = tf.strings.split(file_path, "/")
    # The second to last is the class-directory
    return parts[-2] == "PNEUMONIA"


def decode_img(img):
    # convert the compressed string to a 3D uint8 tensor
    img = tf.image.decode_jpeg(img, channels=3)
    # resize the image to the desired size.
    return tf.image.resize(img, IMAGE_SIZE)


def process_path(image, path):
    label = get_label(path)
    # load the raw data from the file as a string
    img = decode_img(image)
    return img, label


ds = ds.map(process_path, num_parallel_calls = AUTOTUNE)
```

Let's split the data into a training and validation datasets.

In [ ]:
```
ds = ds.shuffle(10000)
train_ds = ds.take(4300)
val_ds = ds.skip(4300)
```

Let's visualize the shape of an (image, label) pair.

In [ ]:
```
for image, label in train_ds.take(1):
    print("Image shape: ", image.numpy().shape)
```

```
Image shape:  (224, 224, 3)
```

Load and format the test data as well.

In [ ]:
```
test_images = tf.data.TFRecordDataset(
    "gs://download.tensorflow.org/data/ChestXRay2017/test/images.tfrec"
)
test_paths = tf.data.TFRecordDataset(
    "gs://download.tensorflow.org/data/ChestXRay2017/test/paths.tfrec"
)

COUNT_NORMAL = len([filename for filename in test_paths if "NORMAL" in filename.numpy().decode("utf-8")])
print("Normal images count in test set: " + str(COUNT_NORMAL))

COUNT_PNEUMONIA = len([filename for filename in test_paths if "PNEUMONIA" in filename.numpy().decode("utf-8")])
print("Pneumonia images count in test set: " + str(COUNT_PNEUMONIA))
```

```
Normal images count in test set: 234
Pneumonia images count in test set: 390
```

In [ ]:
```
test_ds = tf.data.Dataset.zip((test_images, test_paths))

test_ds = test_ds.map(process_path, num_parallel_calls=AUTOTUNE)
test_ds_batch = test_ds.batch(BATCH_SIZE)
```

Visualize the dataset

- First, let's use buffered prefetching so we can yield data from disk without having I/O become blocking.
- Please note that large image datasets should not be cached in memory.
- We do it here because the dataset is not very large and we want to train on TPU.

In [ ]:
```
def prepare_for_training(ds, cache=True):
    # This is a small dataset, only load it once, and keep it in memory.
    # use `.cache(filename)` to cache preprocessing work for datasets that don't
    # fit in memory.

    ds = ds.cache()
    ds = ds.batch(BATCH_SIZE)

    # `prefetch` lets the dataset fetch batches in the background while the model
    # is training.
    ds = ds.prefetch(buffer_size=AUTOTUNE)

    return ds
```

Call the next batch iteration of the training data.

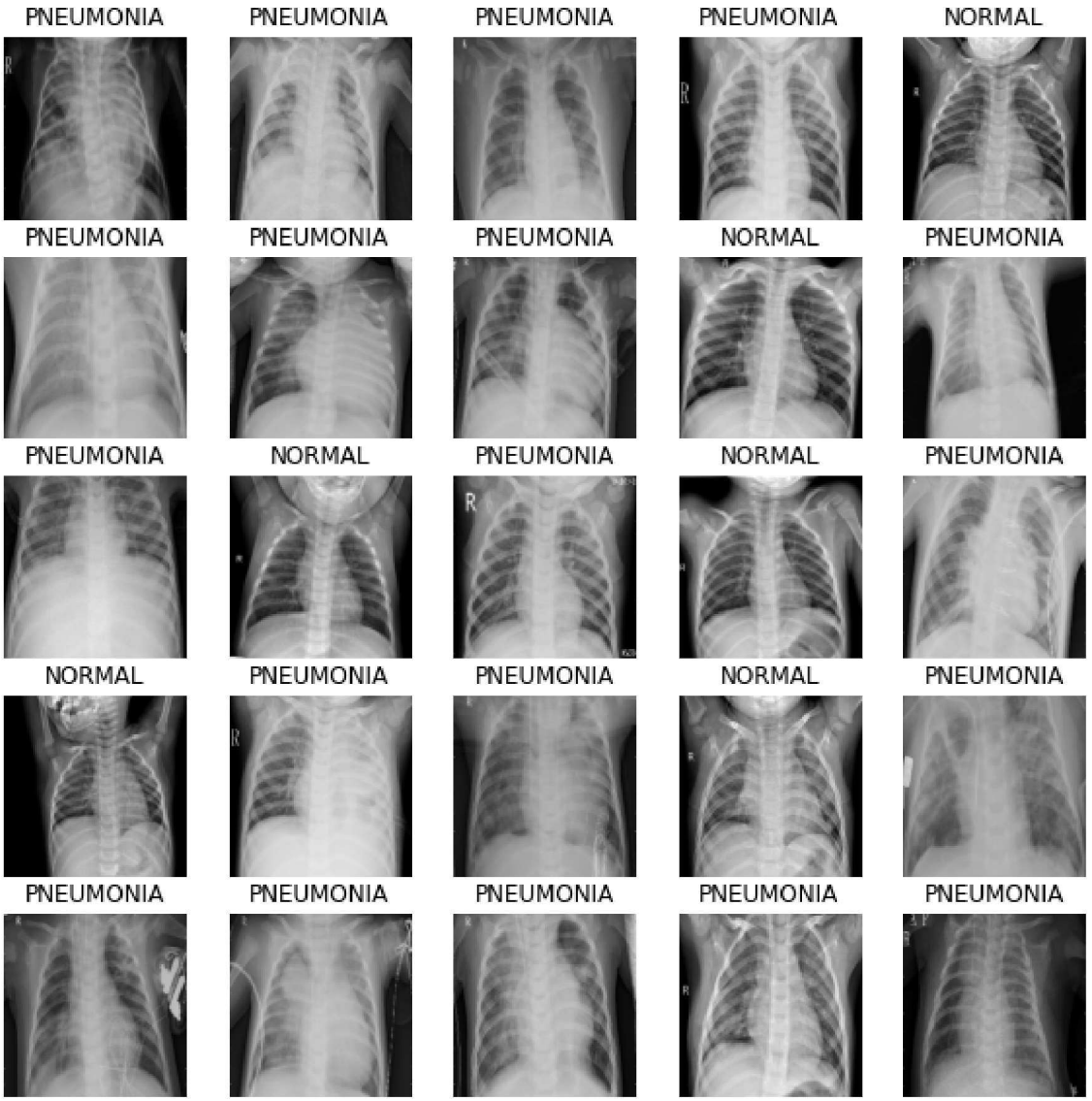
In [ ]:
```
train_ds_batch = prepare_for_training(train_ds)
val_ds_batch = prepare_for_training(val_ds)

image_batch, label_batch = next(iter(train_ds_batch))
```

Define the method to show the images in the batch.

In [ ]:
```
def show_batch(image_batch, label_batch):
    plt.figure(figsize=(10, 10))
    for n in range(25):
        ax = plt.subplot(5, 5, n + 1)
        plt.imshow(image_batch[n] / 255)
        if label_batch[n]:
            plt.title("PNEUMONIA")
        else:
            plt.title("NORMAL")
        plt.axis("off")
```

As the method takes in NumPy arrays as its parameters, call the numpy function on the batches to return the tensor in NumPy array form.

In [ ]:
```
show_batch(image_batch.numpy(), label_batch.numpy())
```



Question:

```
Why didn't we apply augmentation to our data before training ?

A) Augmentation increases computational cost for our mobilenet
B) Augmentation decreases mobilenet accuracy as it has less parameters
C) Xray images becomes useless after augmentation
```

```
D) MobileNet does not accept augmented images
```

Answer:

C)

- Here Data augmentation will not be useful because X-ray scans are only taken in a specific orientation
- variations such as flips and rotations will not exist in real X-ray images.

MobileNet: Why and How ?

Now that our dataset is ready !

We move on to building and training our model.

- While working with Image datasets, some state-of-the-art models include ResNet50, InceptionNet, VGG16, AlexNet, etc.
- These model architectures do achieve groundbreaking accuracies on ImageNet Dataset but are computationally heavy to train and perform model inference
- This called for need of a lightweight model but reliable model
- MobileNet, introduced in 2017 as a lightweight deep neural network, MobileNet has fewer parameters and high accuracy
    - Original Paper: https://arxiv.org/abs/1704.04861

Using MobileNet:

Model Name	Number of params	Top 1 Acc	Top 5 Acc
MobileNet	2.3M	71.0	90.5
ResNet50	25.6M	76	93
Inception	11.2M	74.8	92.2

Model Name	Number of params	Top 1 Acc	Top 5 Acc
VGG16	138M	74.4	91.9
AlexNet	62M	63.3	84.6

- We need to design networks with low parameters, MobileNet is one such model
- Before jumping into details, Lets compare metric of ResNet50 and MobileNet
- MobileNet has 90.5 top 5 acc which is still good, given the fact that it has only 2.5 M parameters
- MobileNet is majorily used for on-device mobile inference, running on Tensorflow Lite (TFlite): https://www.tensorflow.org/lite/guide
- TFlite is a subset of TF, having only necessary functions so we can install the package on low resource devices

Where is MobileNet used ?

- MobileNet, being an efficient yet reliable deep learning network is being used in Mobile apps
- On-device Computer vision for mobile devices to perform object detection, image classfication, etc.
- Low-latency inference allows for quick inference
- MobileNet can be easily integrated for medical use in nobile apps, to give precise results on, assume a xray test for pneumonia
- We will learn how to train a MobileNet Model on pneumonia chest xray data which can be deployed in a mobile app

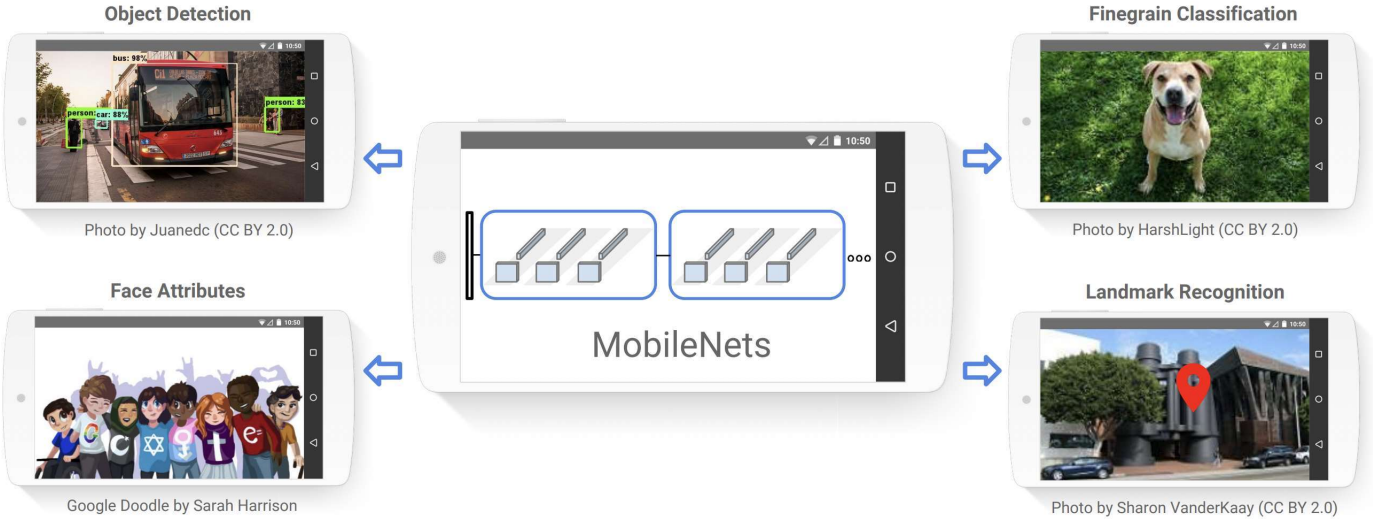


Figure 1. MobileNet models can be applied to various recognition tasks for efficient on device intelligence.

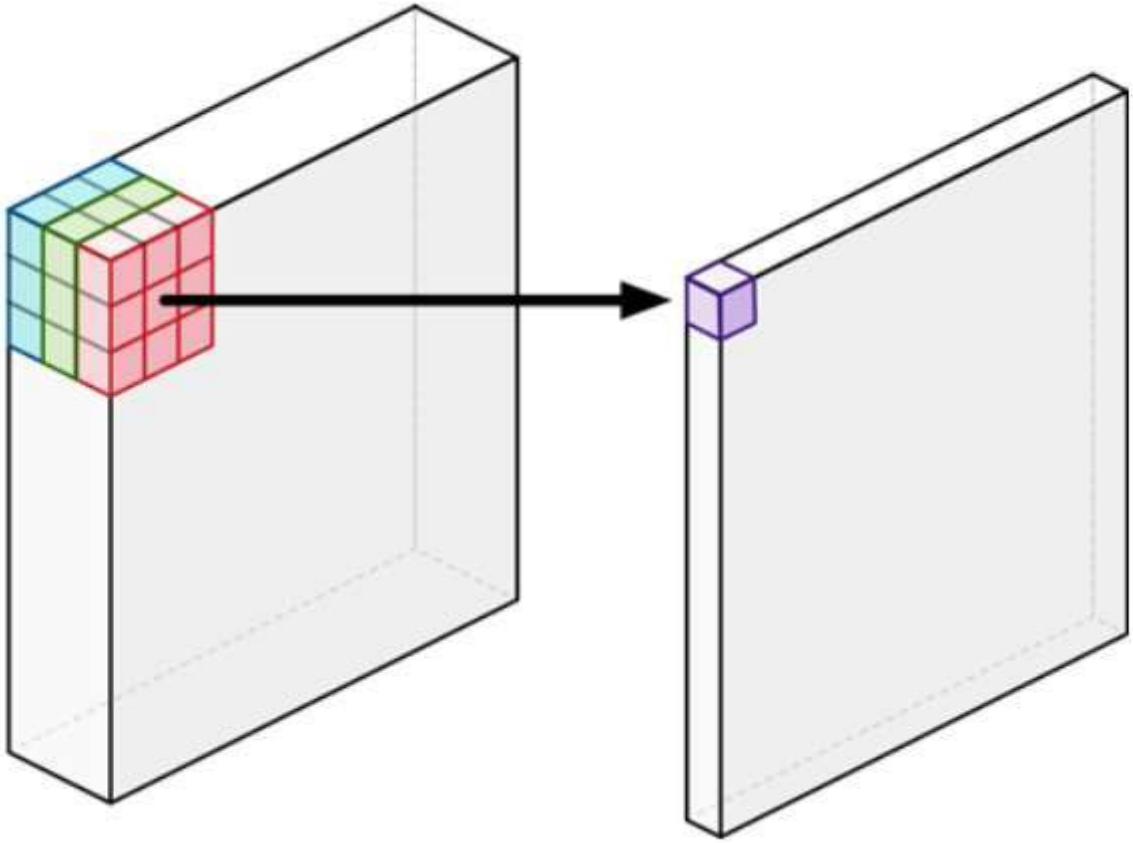
MobileNet : Behind the Scenes

MobileNet core is backed by Depthwise and Pointwise Convolutions which are computationally efficient yet produce significant results.
</br>

How? Lets find out

- MobileNet has aprox 30 layers and ResNet50 also has 50 layers
- So how did mobilenet have 10x low parameters?
- Mobilenet does not use normal convolutional layers
- They use something called Depthwise & pointwise convolutions which is very compute inexpensive

- If the image has 3 input channels, then running a single kernel
- across this image results in an output image with only 1 channel per pixel.



- So for each input pixel, no matter how many channels it has
- the convolution writes a new output pixel with only a single channel

- MobileNet only uses these standard conv layer once, the first conv layers
- after that all other layers do “depthwise separable” convolution
- which is a combination of
    1. depthwise convolution
    2. pointwise convolution

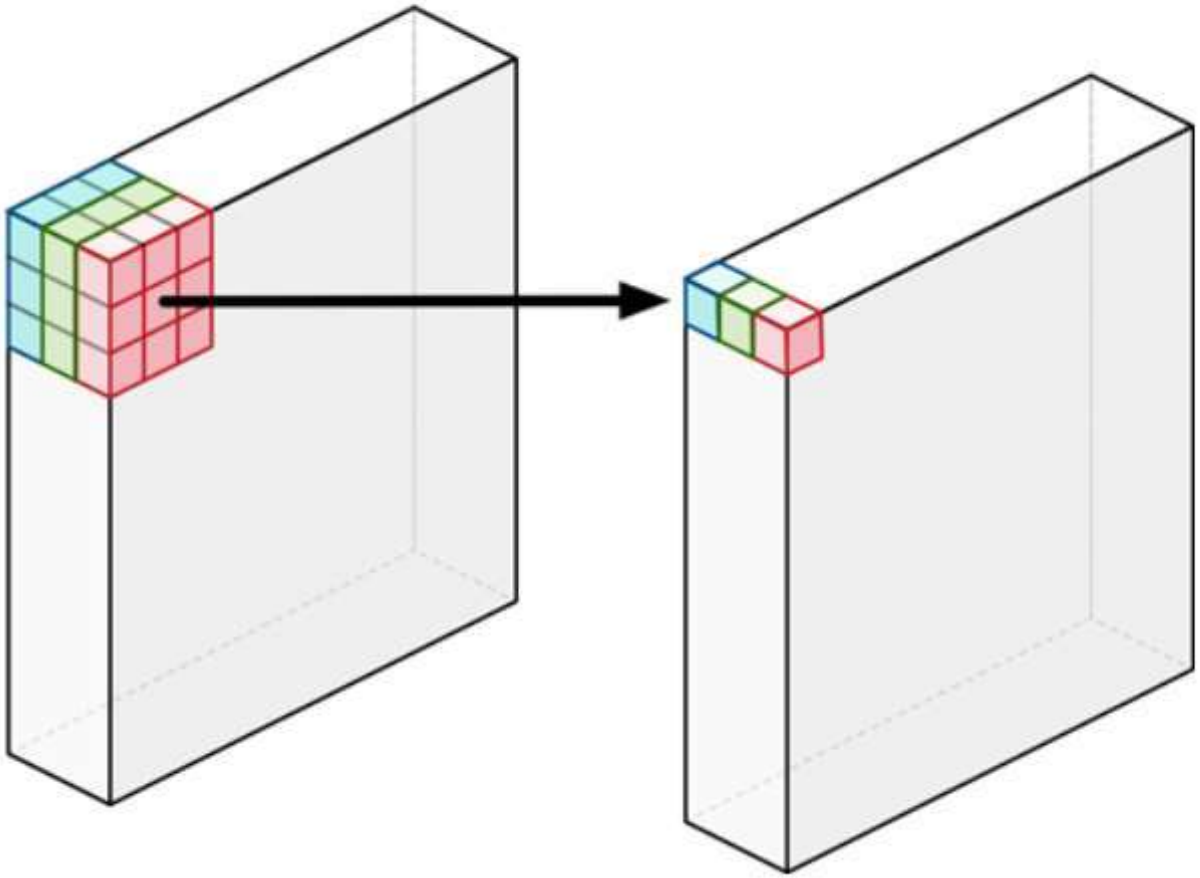
Depthwise Convolution is a type of convolution where we apply a single convolutional filter/kernel for each input channel in the image (3 in RGB). </br>In the regular 2D convolution performed over multiple input channels, the filter is as deep as the input and lets us freely mix channels to generate each element in the output.

In contrast, depthwise convolutions keep each channel separate. To summarize the steps, we:

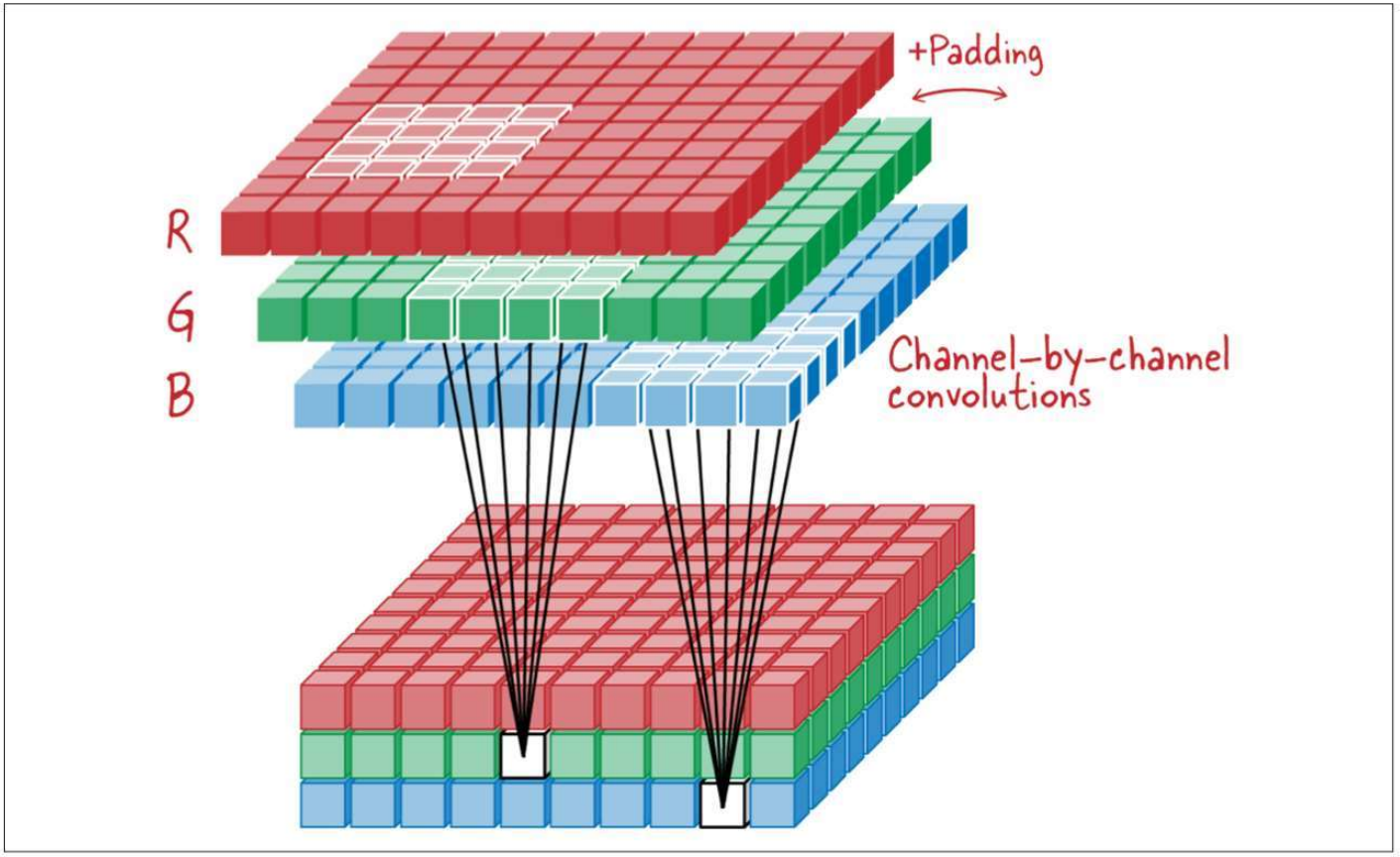
- For MobileNets the depthwise convolution applies a single filter to each input channel.
- The pointwise convolution then applies a 1×1 convolution to combine the outputs the depthwise convolution.
- A standard convolution both filters and combines inputs into a new set of outputs in one step.
- The depthwise separable convolution splits this into two layers, a separate layer for filtering and a separate layer for combining.
- This factorization has the effect of drastically reducing computation and model size.

Summary:

- Split the input and filter into channels.
- We convolve each input with the respective filter.
- We stack the convolved outputs together.

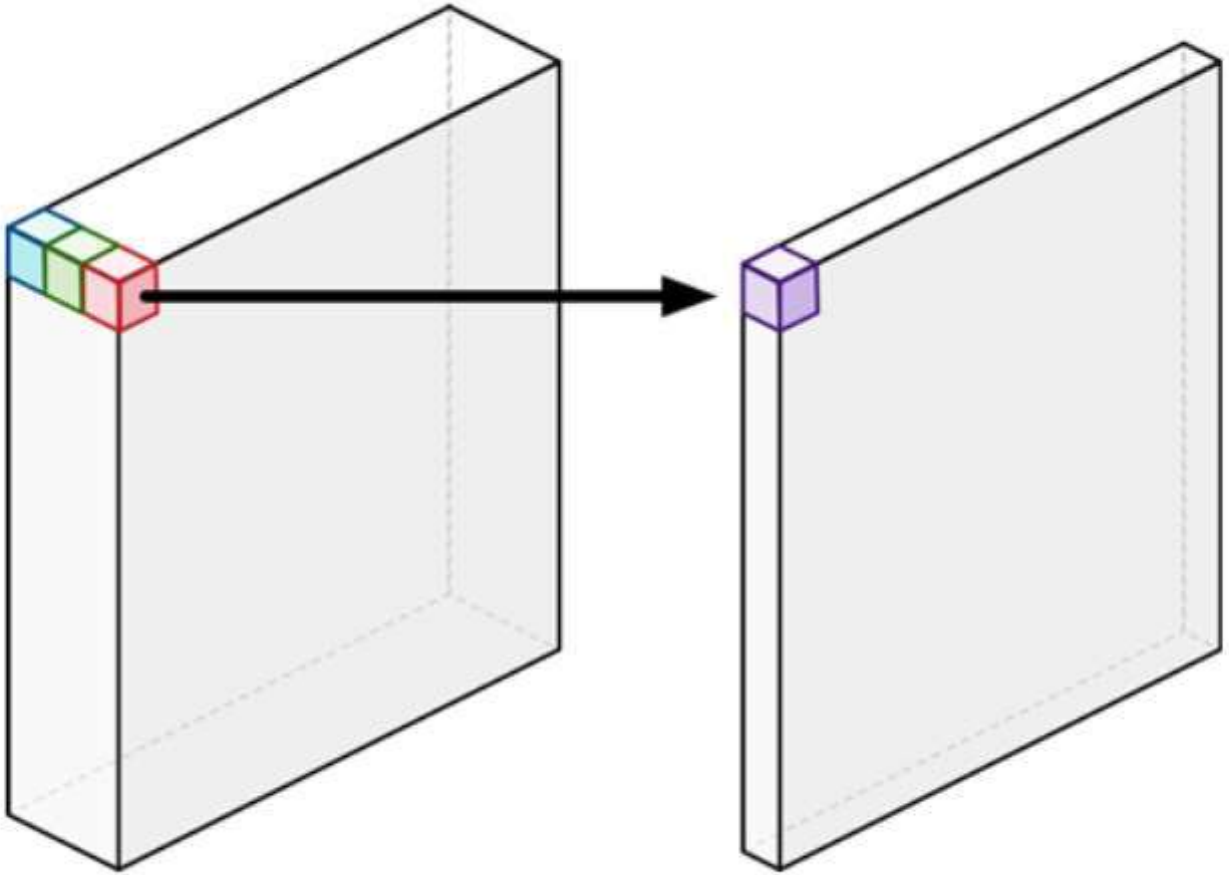


- Unlike a regular convolution it does not combine the input channels
- but it performs convolution on each channel separately.
- For an **input image with 3 channels**,a depthwise convolution creates an **output image that also has 3 channels.**
- Each channel gets its own set of weights.
- The purpose of the depthwise convolution is to filter the input channels

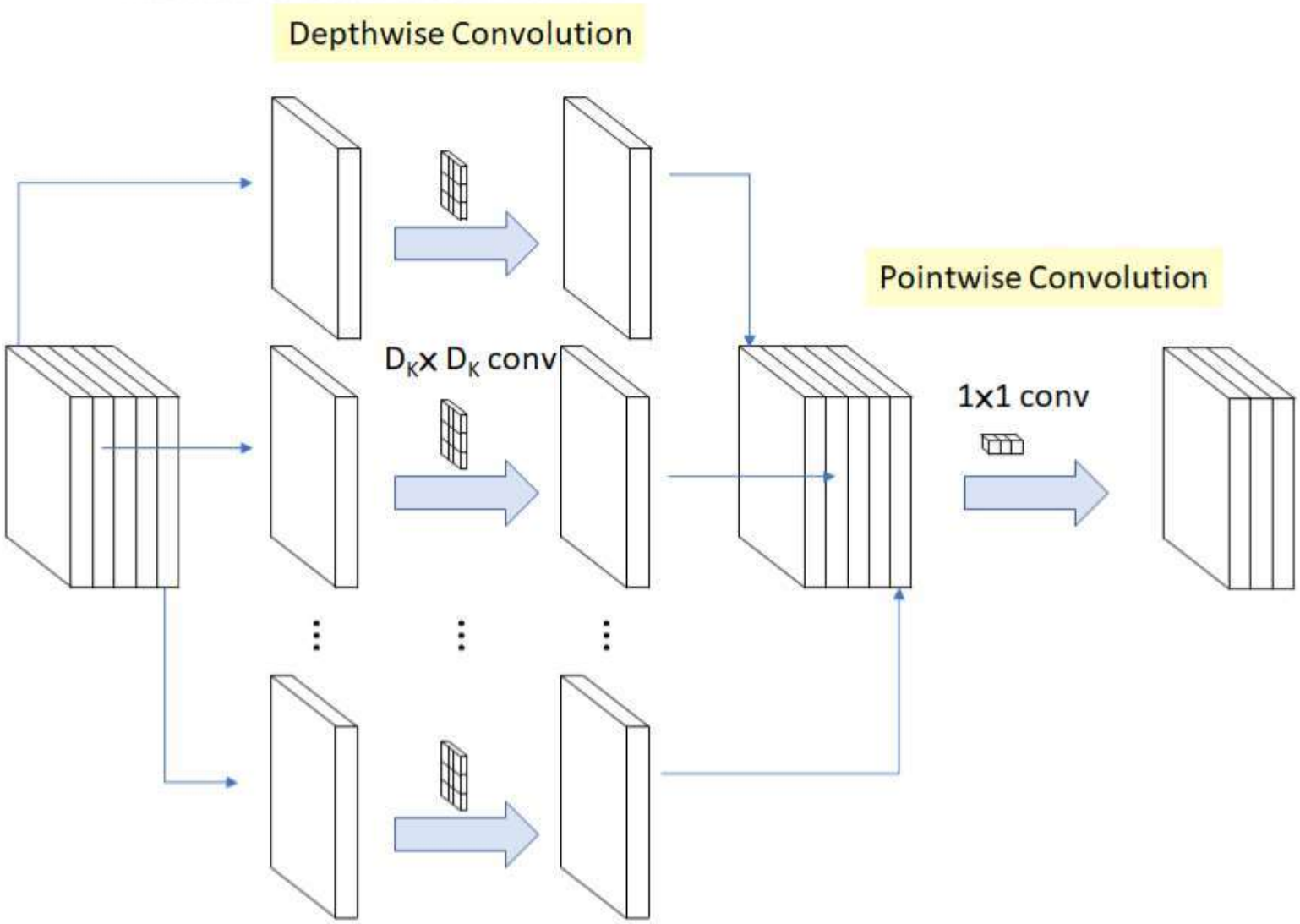


Pointwise Convolution is a type of convolution that uses a 1x1 kernel: a kernel that iterates through every single point. This kernel has a depth of however many channels the input image has.

- It is just a conv with 1x1 kernel, this simply adds up all the channels (as a weighted sum)
- we usually stack together many of these pointwise kernels to create an output image with many channels.
- The purpose of this pointwise convolution is to
- combine the output channels of the depthwise convolution to create new features.

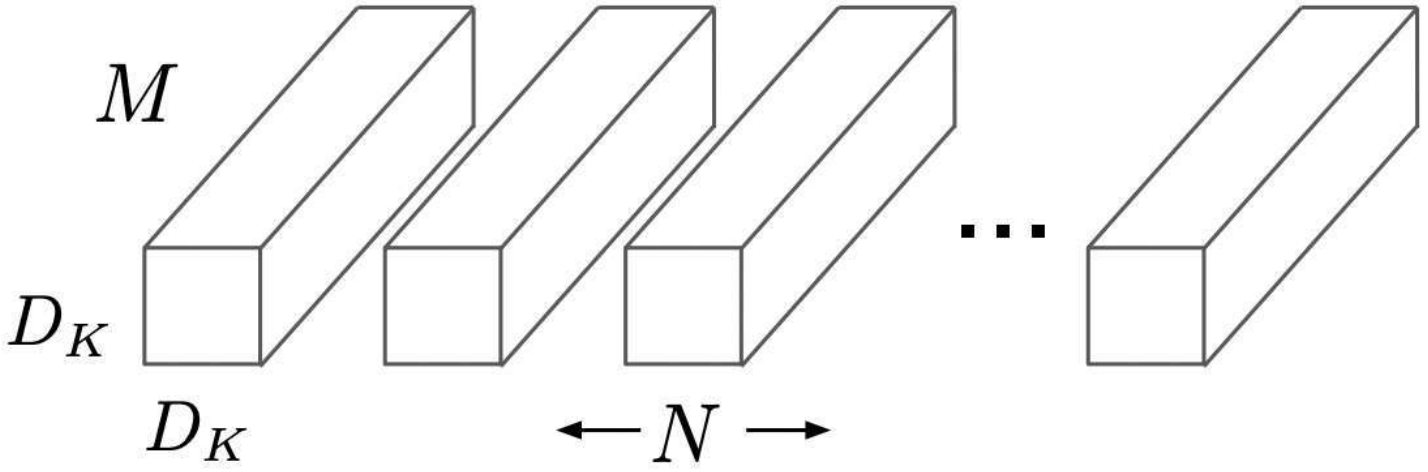


- When we put these two things together a depthwise convolution followed by a pointwise convolution
- the result is called a depthwise separable convolution.
- A regular convolution does both filtering and combining in a single go,
- but with a depthwise separable convolution these two operations are done as separate steps

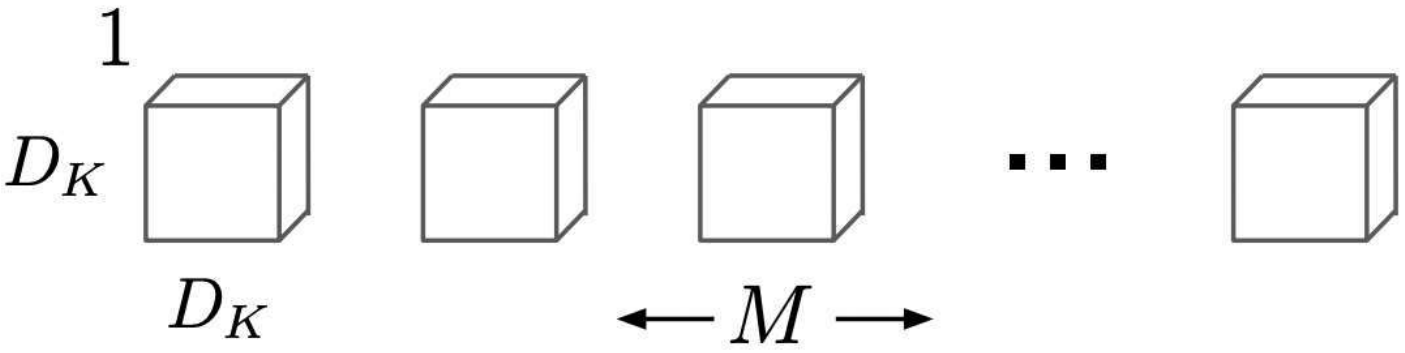


How do Depthwise & pointwise convolutions helps?

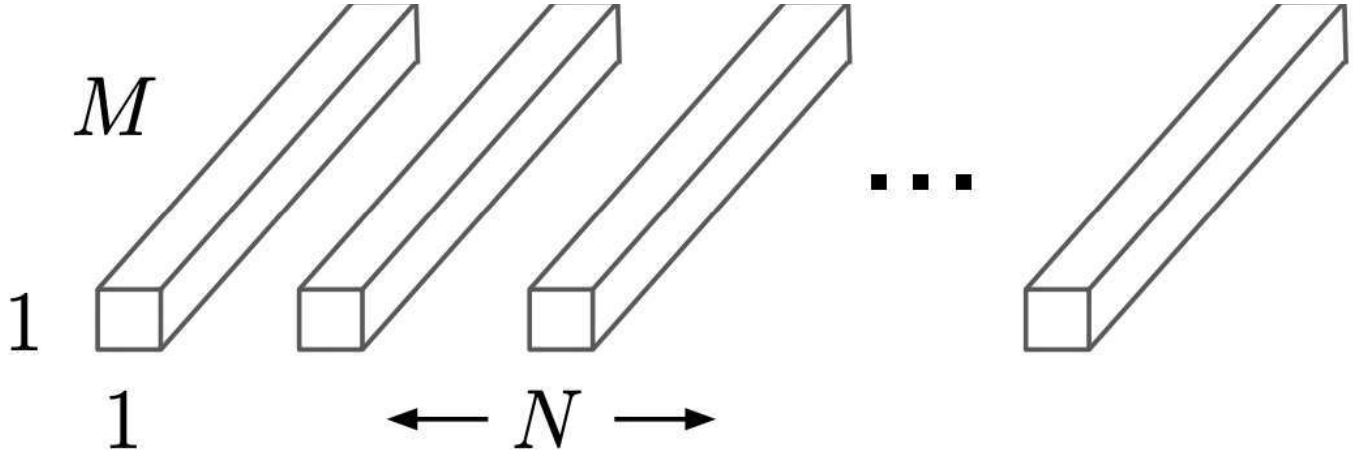
- The end results of both (Normal & MobileNet) approaches are pretty similar they both filter the data and make new features
- but a regular convolution has to do much more computational work to get there and needs to learn more weights.
- By this we are able to lower down the # of params & FLOPs



(a) Standard Convolution Filters



(b) Depthwise Convolutional Filters



(c) 1×1 Convolutional Filters called Pointwise Convolution in the context of Depthwise Separable Convolution

Figure 2. The standard convolutional filters in (a) are replaced by two layers: depthwise convolution in (b) and pointwise convolution in (c) to build a depthwise separable filter.

- Now lets look at the whole network

MobileNets architecture

Table 1. MobileNet Body Architecture

Type / Stride	Filter Shape	Input Size
Conv / s2	$3 \times 3 \times 3 \times 32$	$224 \times 224 \times 3$
Conv dw / s1	$3 \times 3 \times 32$ dw	$112 \times 112 \times 32$
Conv / s1	$1 \times 1 \times 32 \times 64$	$112 \times 112 \times 32$
Conv dw / s2	$3 \times 3 \times 64$ dw	$112 \times 112 \times 64$
Conv / s1	$1 \times 1 \times 64 \times 128$	$56 \times 56 \times 64$
Conv dw / s1	$3 \times 3 \times 128$ dw	$56 \times 56 \times 128$
Conv / s1	$1 \times 1 \times 128 \times 128$	$56 \times 56 \times 128$
Conv dw / s2	$3 \times 3 \times 128$ dw	$56 \times 56 \times 128$
Conv / s1	$1 \times 1 \times 128 \times 256$	$28 \times 28 \times 128$
Conv dw / s1	$3 \times 3 \times 256$ dw	$28 \times 28 \times 256$
Conv / s1	$1 \times 1 \times 256 \times 256$	$28 \times 28 \times 256$
Conv dw / s2	$3 \times 3 \times 256$ dw	$28 \times 28 \times 256$
Conv / s1	$1 \times 1 \times 256 \times 512$	$14 \times 14 \times 256$
$5\times$ Conv dw / s1 Conv / s1	$3 \times 3 \times 512$ dw $1 \times 1 \times 512 \times 512$	$14 \times 14 \times 512$ $14 \times 14 \times 512$
Conv dw / s2	$3 \times 3 \times 512$ dw	$14 \times 14 \times 512$
Conv / s1	$1 \times 1 \times 512 \times 1024$	$7 \times 7 \times 512$

Conv dw / s2	$3 \times 3 \times 1024$ dw	$7 \times 7 \times 1024$
Conv / s1	$1 \times 1 \times 1024 \times 1024$	$7 \times 7 \times 1024$
Avg Pool / s1	Pool 7×7	$7 \times 7 \times 1024$
FC / s1	1024×1000	$1 \times 1 \times 1024$
Softmax / s1	Classifier	$1 \times 1 \times 1000$

- The MobileNets network has approx 30 layers.
- The design of the network is quite straightforward:

    1. convolutional layer with stride 2
    2. depthwise layer
    3. pointwise layer that doubles the number of channels
    4. depthwise layer with stride 2
    5. pointwise layer that doubles the number of channels
    6. depthwise layer
    7. pointwise layer
    8. depthwise layer with stride 2
    9. pointwise layer that doubles the number of channels
- then reapeat this structure 5 times

MobileNet versions:

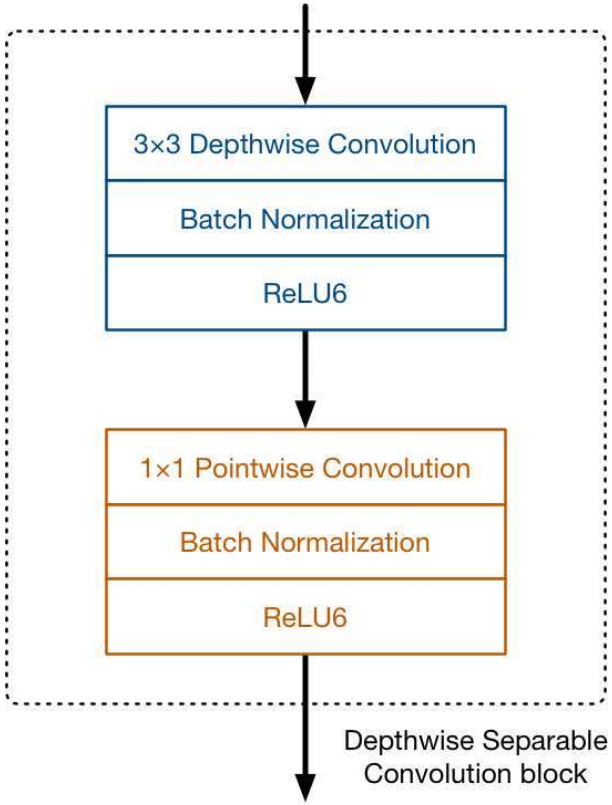
We primarily have three versions of MobileNet, which over the years improved on-device machine learning for Mobile devices.

Refer to this First MobileNet paper

MobileNetV1

The first version of MobileNet was introduced in 2017 by Google. The idea behind it was primarily develop TensorFlow based comoputer vision models that are suitable for low-resource on device inference on mobile devices.

The core architecture of MobileNet V1 is based on depthwise and pointwise convolutions that helps in reducing the computational of the model.

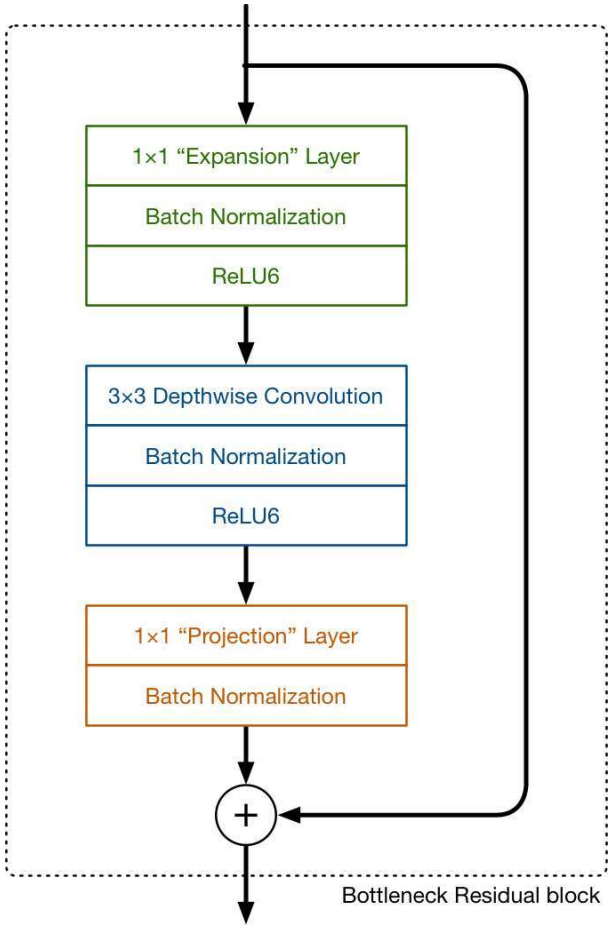


MobileNetV2[POST READ]

The second version of MobileNet architecture was released in 2018, a ramped up version of V1, to optimize the model architecture for object detection, classfication and segmentation tasks.

The architecture introduced Linear bottlenecks (a BottleNeck Block without the last activation) and residual connections (known as shortcuts) which enable higher accuracy and faster training.

MobileNetV2 Paper



Architecture using keras or tf:

- MobileNet is one of the smallest Deep Neural networks that are fast and efficient and can be run on devices without high-end GPUs.
- Implementation of these networks is very simple when using a framework such as Keras (on TensorFlow).

In [ ]:
```
import tensorflow as tf

#import all necessary layers
from tensorflow.keras.layers import Input, DepthwiseConv2D
from tensorflow.keras.layers import Conv2D, BatchNormalization
```

```
from tensorflow.keras.layers import ReLU, AvgPool2D, Flatten, Dense

from tensorflow.keras import Model
```

In [ ]:
```
# MobileNet block
def mobilnet_block (x, filters, strides):

    x = DepthwiseConv2D(kernel_size = 3, strides = strides, padding = 'same')(x)
    x = BatchNormalization()(x)
    x = ReLU()(x)

    x = Conv2D(filters = filters, kernel_size = 1, strides = 1)(x)
    x = BatchNormalization()(x)
    x = ReLU()(x)

    return x
```

In [ ]:
```
#stem of the model
input = Input(shape = (224, 224, 3))
x = Conv2D(filters = 32, kernel_size = 3, strides = 2, padding = 'same')(input)
x = BatchNormalization()(x)
x = ReLU()(x)
```

In [ ]:
```
# main part of the model
x = mobilnet_block(x, filters = 64, strides = 1)
x = mobilnet_block(x, filters = 128, strides = 2)
x = mobilnet_block(x, filters = 128, strides = 1)
x = mobilnet_block(x, filters = 256, strides = 2)
x = mobilnet_block(x, filters = 256, strides = 1)
x = mobilnet_block(x, filters = 512, strides = 2)
for _ in range (5):
     x = mobilnet_block(x, filters = 512, strides = 1)
x = mobilnet_block(x, filters = 1024, strides = 2)
x = mobilnet_block(x, filters = 1024, strides = 1)
x = AvgPool2D (pool_size = 7, strides = 1, data_format='channels_first')(x)
output = Dense (units = 1000, activation = 'softmax')(x)
model = Model(inputs=input, outputs=output)
model.summary()
```

```
Model: "model_10"
_________________________________________________________________
 Layer (type)                Output Shape              Param #
=================================================================
 input_11 (InputLayer)       [(None, 224, 224, 3)]     0

 conv2d_28 (Conv2D)          (None, 112, 112, 32)      896

 batch_normalization_54 (Bat  (None, 112, 112, 32)     128
 chNormalization)

 re_lu_54 (ReLU)             (None, 112, 112, 32)      0

 depthwise_conv2d_26 (Depthw  (None, 112, 112, 32)     320
 iseConv2D)

 batch_normalization_55 (Bat  (None, 112, 112, 32)     128
 chNormalization)

 re_lu_55 (ReLU)             (None, 112, 112, 32)      0

 conv2d_29 (Conv2D)          (None, 112, 112, 64)      2112

 batch_normalization_56 (Bat  (None, 112, 112, 64)     256
 chNormalization)

 re_lu_56 (ReLU)             (None, 112, 112, 64)      0

 depthwise_conv2d_27 (Depthw  (None, 56, 56, 64)       640
 iseConv2D)

 batch_normalization_57 (Bat  (None, 56, 56, 64)       256
 chNormalization)

 re_lu_57 (ReLU)             (None, 56, 56, 64)        0

 conv2d_30 (Conv2D)          (None, 56, 56, 128)       8320

 batch_normalization_58 (Bat  (None, 56, 56, 128)      512
 chNormalization)

 re_lu_58 (ReLU)             (None, 56, 56, 128)       0

 depthwise_conv2d_28 (Depthw  (None, 56, 56, 128)      1280
 iseConv2D)
```

```
batch_normalization_59 (Bat  (None, 56, 56, 128)      512
chNormalization)

re_lu_59 (ReLU)              (None, 56, 56, 128)      0

conv2d_31 (Conv2D)           (None, 56, 56, 128)      16512

batch_normalization_60 (Bat  (None, 56, 56, 128)      512
chNormalization)

re_lu_60 (ReLU)              (None, 56, 56, 128)      0

depthwise_conv2d_29 (Depthw  (None, 28, 28, 128)      1280
iseConv2D)

batch_normalization_61 (Bat  (None, 28, 28, 128)      512
chNormalization)

re_lu_61 (ReLU)              (None, 28, 28, 128)      0

conv2d_32 (Conv2D)           (None, 28, 28, 256)      33024

batch_normalization_62 (Bat  (None, 28, 28, 256)      1024
chNormalization)

re_lu_62 (ReLU)              (None, 28, 28, 256)      0

depthwise_conv2d_30 (Depthw  (None, 28, 28, 256)      2560
iseConv2D)

batch_normalization_63 (Bat  (None, 28, 28, 256)      1024
chNormalization)

re_lu_63 (ReLU)              (None, 28, 28, 256)      0

conv2d_33 (Conv2D)           (None, 28, 28, 256)      65792

batch_normalization_64 (Bat  (None, 28, 28, 256)      1024
chNormalization)

re_lu_64 (ReLU)              (None, 28, 28, 256)      0

depthwise_conv2d_31 (Depthw  (None, 14, 14, 256)      2560
iseConv2D)
```

```
batch_normalization_65 (Bat  (None, 14, 14, 256)      1024
chNormalization)

re_lu_65 (ReLU)              (None, 14, 14, 256)      0

conv2d_34 (Conv2D)           (None, 14, 14, 512)      131584

batch_normalization_66 (Bat  (None, 14, 14, 512)      2048
chNormalization)

re_lu_66 (ReLU)              (None, 14, 14, 512)      0

depthwise_conv2d_32 (Depthw  (None, 14, 14, 512)      5120
iseConv2D)

batch_normalization_67 (Bat  (None, 14, 14, 512)      2048
chNormalization)

re_lu_67 (ReLU)              (None, 14, 14, 512)      0

conv2d_35 (Conv2D)           (None, 14, 14, 512)      262656

batch_normalization_68 (Bat  (None, 14, 14, 512)      2048
chNormalization)

re_lu_68 (ReLU)              (None, 14, 14, 512)      0

depthwise_conv2d_33 (Depthw  (None, 14, 14, 512)      5120
iseConv2D)

batch_normalization_69 (Bat  (None, 14, 14, 512)      2048
chNormalization)

re_lu_69 (ReLU)              (None, 14, 14, 512)      0

conv2d_36 (Conv2D)           (None, 14, 14, 512)      262656

batch_normalization_70 (Bat  (None, 14, 14, 512)      2048
chNormalization)

re_lu_70 (ReLU)              (None, 14, 14, 512)      0

depthwise_conv2d_34 (Depthw  (None, 14, 14, 512)      5120
iseConv2D)
```

```
batch_normalization_71 (Bat  (None, 14, 14, 512)      2048
chNormalization)

re_lu_71 (ReLU)              (None, 14, 14, 512)      0

conv2d_37 (Conv2D)           (None, 14, 14, 512)      262656

batch_normalization_72 (Bat  (None, 14, 14, 512)      2048
chNormalization)

re_lu_72 (ReLU)              (None, 14, 14, 512)      0

depthwise_conv2d_35 (Depthw  (None, 14, 14, 512)      5120
iseConv2D)

batch_normalization_73 (Bat  (None, 14, 14, 512)      2048
chNormalization)

re_lu_73 (ReLU)              (None, 14, 14, 512)      0

conv2d_38 (Conv2D)           (None, 14, 14, 512)      262656

batch_normalization_74 (Bat  (None, 14, 14, 512)      2048
chNormalization)

re_lu_74 (ReLU)              (None, 14, 14, 512)      0

depthwise_conv2d_36 (Depthw  (None, 14, 14, 512)      5120
iseConv2D)

batch_normalization_75 (Bat  (None, 14, 14, 512)      2048
chNormalization)

re_lu_75 (ReLU)              (None, 14, 14, 512)      0

conv2d_39 (Conv2D)           (None, 14, 14, 512)      262656

batch_normalization_76 (Bat  (None, 14, 14, 512)      2048
chNormalization)

re_lu_76 (ReLU)              (None, 14, 14, 512)      0

depthwise_conv2d_37 (Depthw  (None, 7, 7, 512)        5120
iseConv2D)
```

```
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
```