Deep Learning

Deep Learning

Lecture: Convolutional Neural Networks –
Common Architectures
Checkpointing
Data Augmentation
Feature Extraction

Ted Scully

RECAP

5	1	3	5	7	9
3	4	4	0	7	8
6	1	8	6	3	4
1	3	5	8	4	6
1	7	5	3	1	2
7	2	2	6	4	6

Original Image

Feature Map

(Result of Convolution)

-1	0	1
-1	0	1
-1	0	1

Filter (Kernel)

- A convolution involves repeatedly applying a **filter** (kernel) to an original image.
- A filter allows us to extract specific features from the original image.

RECAP

5 **-1**	1 **0**	3 **1**	5	7	9
3 **-1**	4 **0**	4 **1**	0	7	8
6 **-1**	1 **0**	8 **1**	6	3	4
1	3	5	8	4	6
1	7	5	3	1	2
7	2	2	6	4	6

Original Image

-1	0	1
-1	0	1
-1	0	1

Feature Map
(Result of Convolution)

1			

(5*-1) + (1*0) + (3*1) +
(3*-1) + (4*0) + (4*1) +
(6*-1) + (1*0) + (8*1)

RECAP

Notice that we move (slide) the filter over one pixel width to the right

5	1 **-1**	3 **0**	5 **1**	7	9
3	4 **-1**	4 **0**	0 **1**	7	8
6	1 **-1**	8 **0**	6 **1**	3	4
1	3	5	8	4	6
1	7	5	3	1	2
7	2	2	6	4	6

Original Image

Feature Map
(Result of Convolution)

1	5		

-1	0	1
-1	0	1
-1	0	1

(1*-1) + (3*0) + (5*1) +
(4*-1) + (4*0) + (0*1) +
(1*-1) + (8*0) + (6*1)

RECAP

5	1	3	5	7	9
3	4	4	0	7	8
6	1	8	6	3	4
1	3	5	8	4	6
1	7	5	3	1	2
7	2	2	6	4	6

Original Image

?	?	?
?	?	?
?	?	?

Filter (Kernel)

Feature Map
(Result of Convolution)

1	5	2	10
7	.	.	.
.	.	.	.
.	.	.	.

RECAP

5	1	3	5	7	9
3	4	4	0	7	8
6	1	8	6	3	4
1	3	5	8	4	6
1	7	5	3	1	2
7	2	2	6	4	6

Original Image

?	?	?
?	?	?
?	?	?

Filter (Kernel)

Feature Map
(Result of Convolution)

RELU(

1	5	2	10
7	.	.	.
.	.	.	.
.	.	.	.

+ bias)

Components of CNNs - Padding

RECAP

- Padding is a simple technique where we pad the boundary of an image with zero value pixels before performing a convolution (The actual **<u>image pixels will get more exposure to the convolutions</u>**) and feature maps don't get progressively smaller.

- Two types of padding (**Valid** and **Same**)

0	0	0	0	0	0	0	0
0	5	1	3	5	7	9	0
0	3	4	4	0	7	8	0
0	6	1	8	6	3	4	0
0	1	3	5	8	4	6	0
0	1	7	5	3	1	2	0
0	7	2	2	6	4	6	0
0	0	0	0	0	0	0	0

Original Image

?	?	?
?	?	?
?	?	?

Filter (Kernel)

Result of Convolution

RECAP

Stride

- By default when performing our convolution we slide the filter in steps of **one pixel at a time**.
- The stride is a integer parameter that is used to specify the **step size in pixels** when performing convolutions (in the previous example our stride s = 1).
- Notice below we are moving the filter in two pixels steps (s = 2)

0 ?	0 ?	0 ?	0	0	0	0
0 ?	5 ?	1 ?	3	5	7	0
0 ?	3 ?	4 ?	4	0	7	0
0	6	1	8	6	3	0
0	1	3	5	8	4	0
0	1	7	5	3	1	0
0	0	0	0	0	0	0

RECAP

Stride

- By default when performing our convolution we slide the filter in steps of one pixel at a time.
- The stride is a integer parameter that is used to specify the **step size in pixels** when performing convolutions (in the previous example our stride s = 1).
- Notice below we are moving the filter in two pixels steps (s = 2)

0	0	0 ?	0 ?	0 ?	0	0
0	5	1 ?	3 ?	5 ?	7	0
0	3	4 ?	4 ?	0 ?	7	0
0	6	1	8	6	3	0
0	1	3	5	8	4	0
0	1	7	5	3	1	0
0	0	0	0	0	0	0

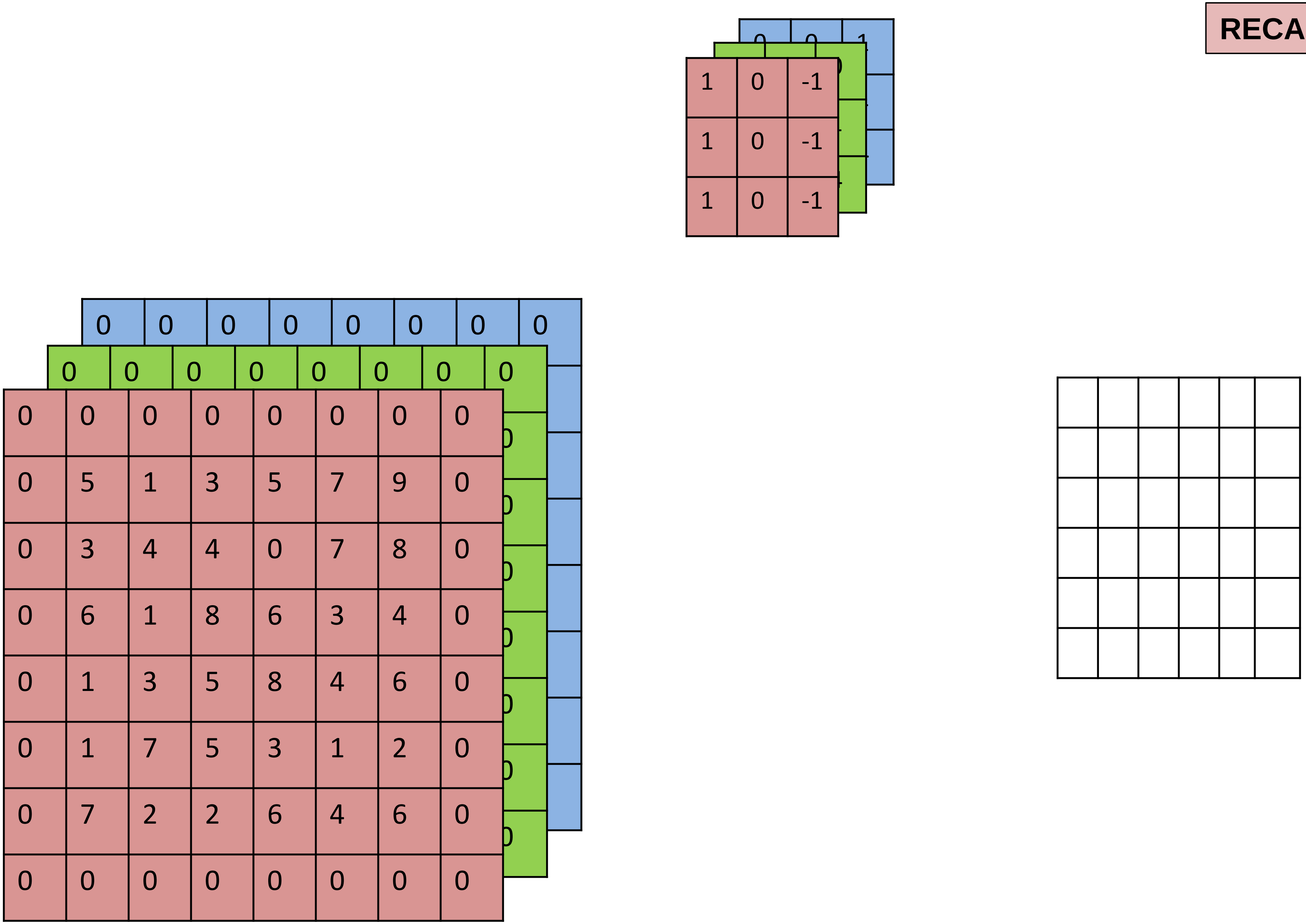
RECAP

Stride

- By default when performing our convolution we slide the filter in steps of one pixel at a time.
- The stride is a integer parameter that is used to specify the **step size in pixels** when performing convolutions (in the previous example our stride s = 1).
- Notice below we are moving the filter in two pixels steps (s = 2)

0	0	0	0	0 ?	0 ?	0 ?
0	5	1	3	5 ?	7 ?	0 ?
0	3	4	4	0 ?	7 ?	0 ?
0	6	1	8	6	3	0
0	1	3	5	8	4	0
0	1	7	5	3	1	0
0	0	0	0	0	0	0

RECAP



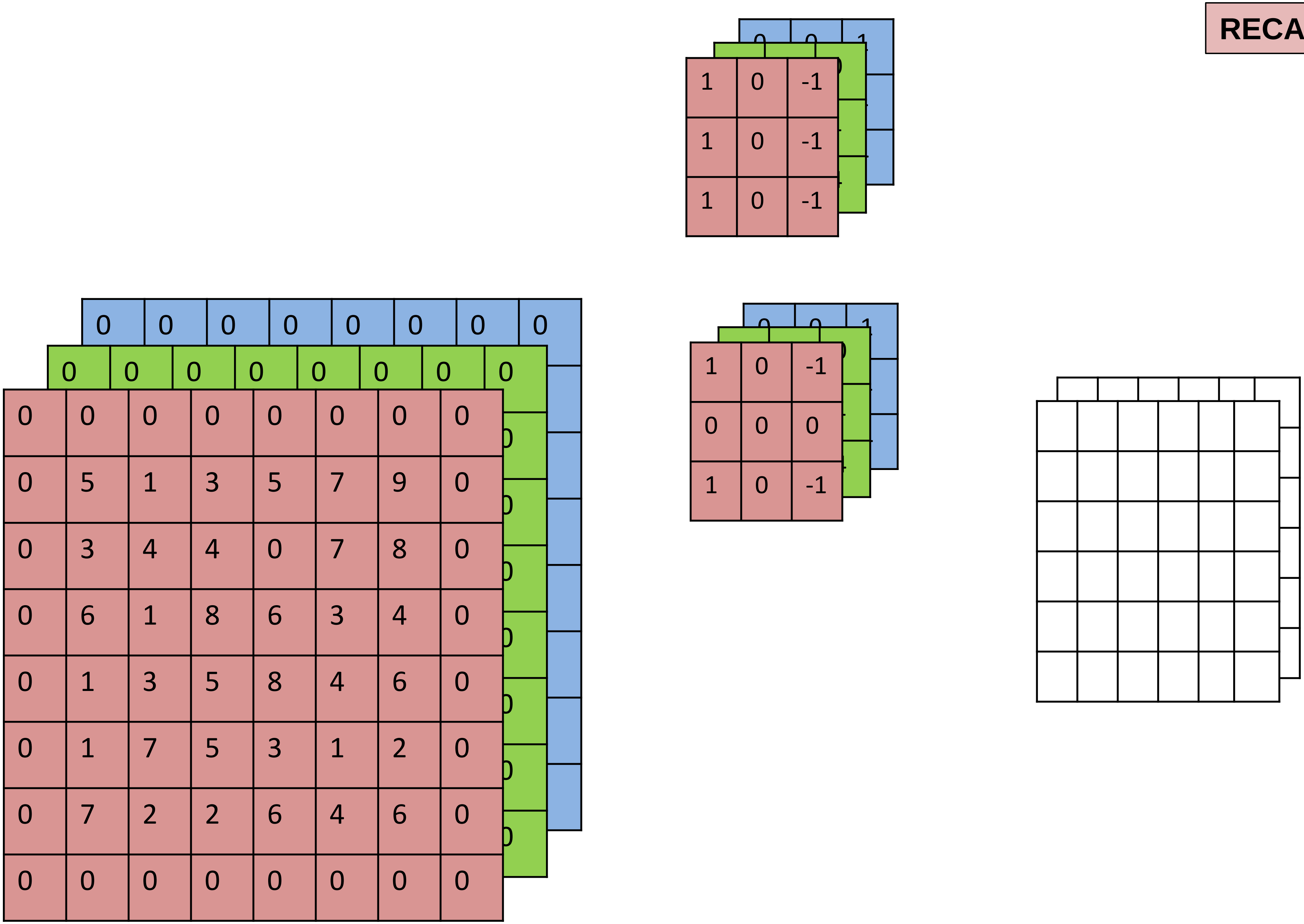
1	0	-1
1	0	-1
1	0	-1

0	0	0	0	0	0	0	0

0	0	0	0	0	0	0	0

0	0	0	0	0	0	0	0
0	5	1	3	5	7	9	0
0	3	4	4	0	7	8	0
0	6	1	8	6	3	4	0
0	1	3	5	8	4	6	0
0	1	7	5	3	1	2	0
0	7	2	2	6	4	6	0
0	0	0	0	0	0	0	0

RECAP

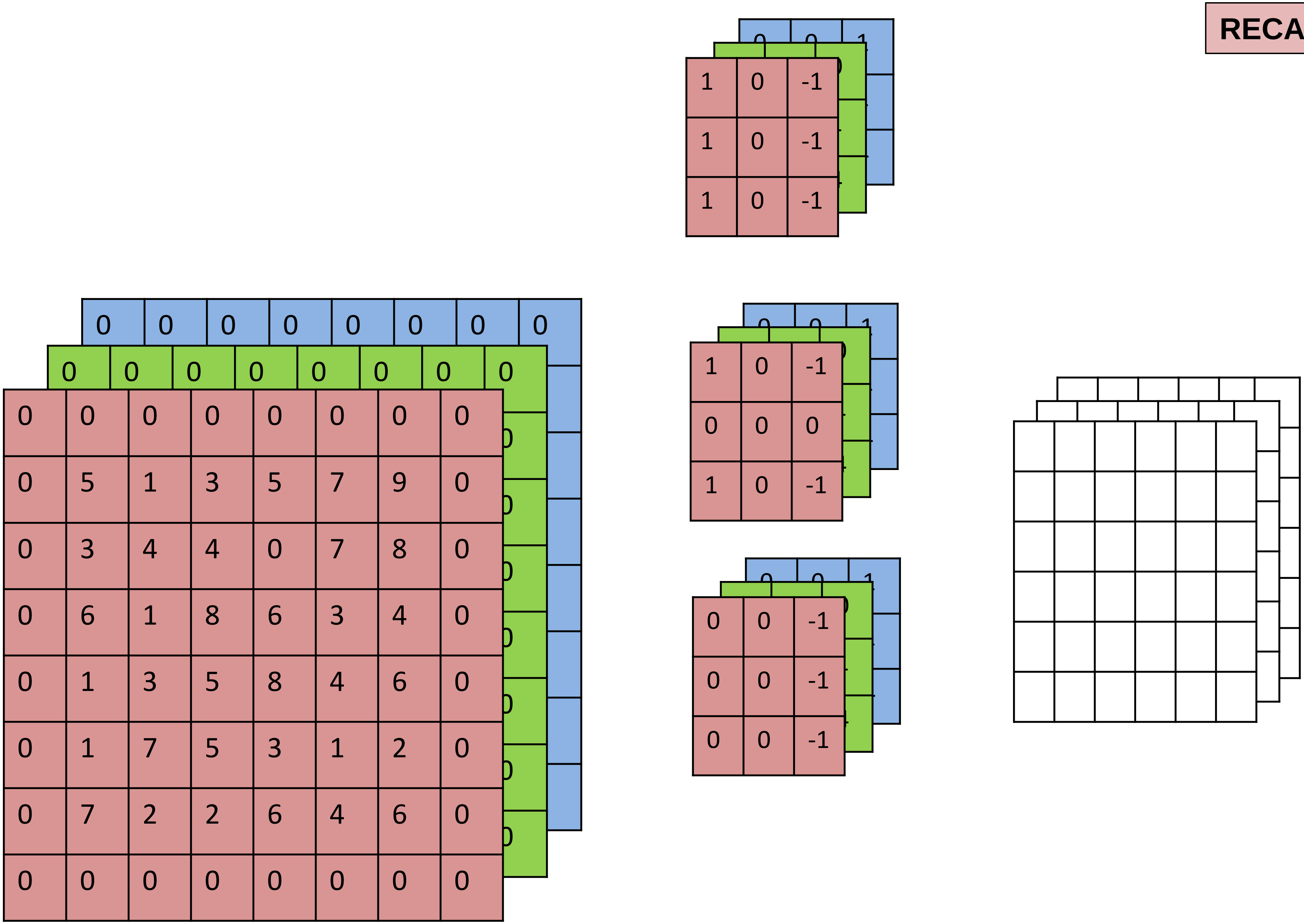


0	0	0	0	0	0	0	0
0	5	1	3	5	7	9	0
0	3	4	4	0	7	8	0
0	6	1	8	6	3	4	0
0	1	3	5	8	4	6	0
0	1	7	5	3	1	2	0
0	7	2	2	6	4	6	0
0	0	0	0	0	0	0	0

1	0	-1
1	0	-1
1	0	-1

1	0	-1
0	0	0
1	0	-1

RECAP



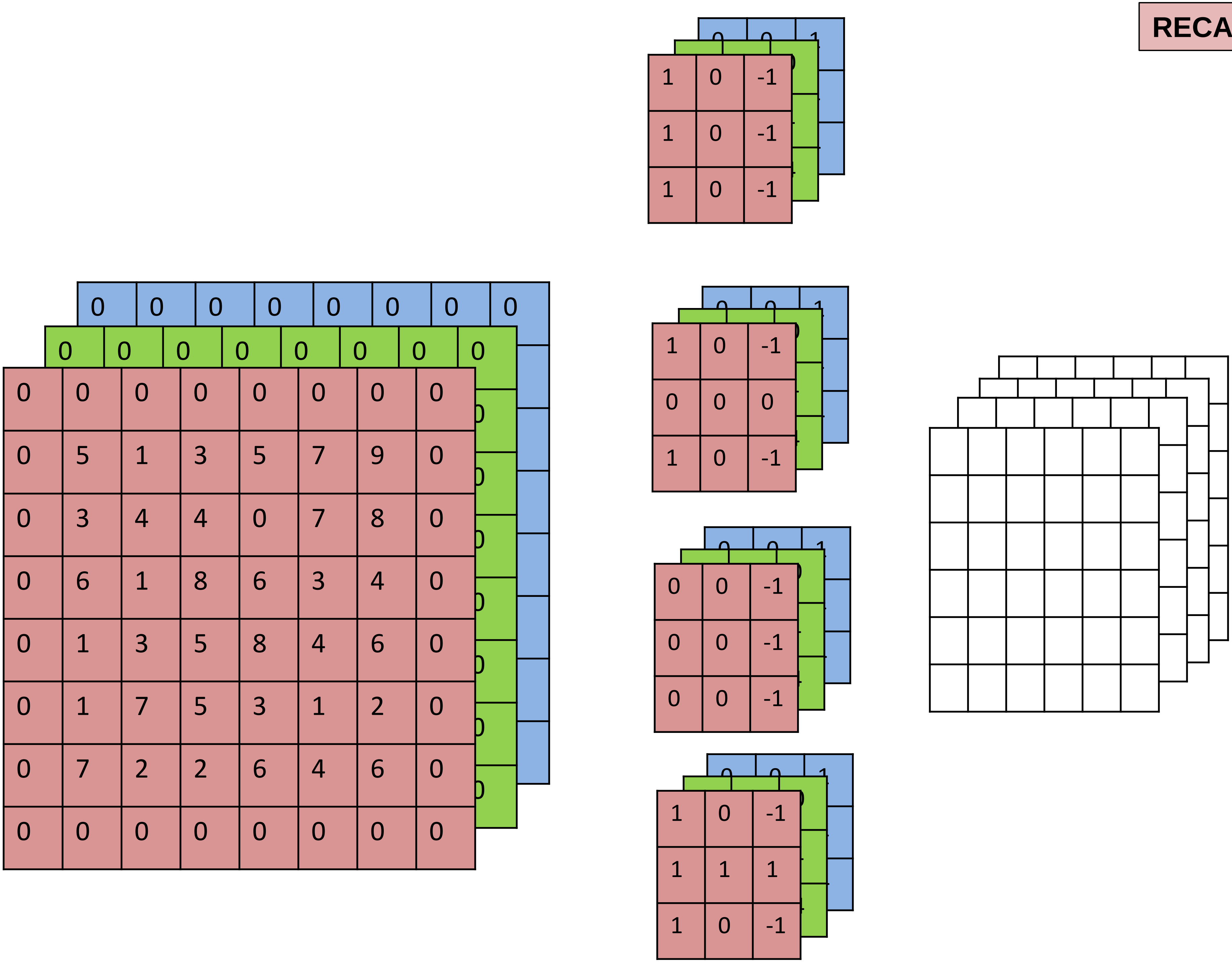
0	0	0	0	0	0	0	0
0	5	1	3	5	7	9	0
0	3	4	4	0	7	8	0
0	6	1	8	6	3	4	0
0	1	3	5	8	4	6	0
0	1	7	5	3	1	2	0
0	7	2	2	6	4	6	0
0	0	0	0	0	0	0	0

1	0	-1
1	0	-1
1	0	-1

1	0	-1
0	0	0
1	0	-1

0	0	-1
0	0	-1
0	0	-1

RECAP



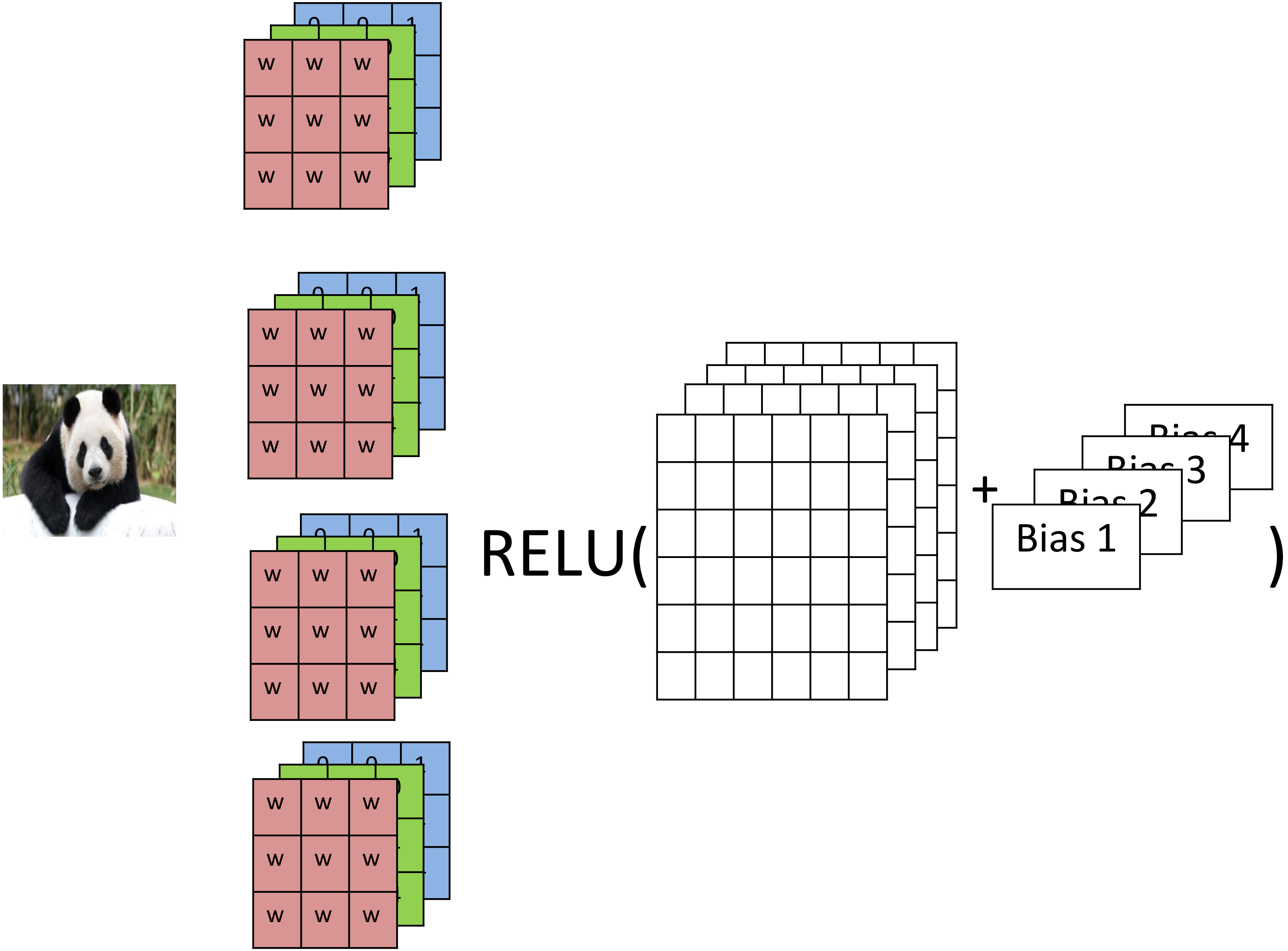
0	0	0	0	0	0	0	0
0	5	1	3	5	7	9	0
0	3	4	4	0	7	8	0
0	6	1	8	6	3	4	0
0	1	3	5	8	4	6	0
0	1	7	5	3	1	2	0
0	7	2	2	6	4	6	0
0	0	0	0	0	0	0	0

1	0	-1
1	0	-1
1	0	-1

1	0	-1
0	0	0
1	0	-1

0	0	-1
0	0	-1
0	0	-1

1	0	-1
1	1	1
1	0	-1


w w w
w w w
w w w
w w w
w w w
w w w
w w w
w w w
w w w
w w w
w w w
w w w
RELU(
+
Bias 1
Bias 2
Bias 3
Bias 4
)

RECAP

Pooling Layers (Downsampling)

- The primary objective of the pooling layer is to **reduce the size of the representation**, which will in turn speeds up computation.
- The pooling layer itself **has no learnable parameters**. However, it has two hyper-parameters:
  - The first is the **stride length** (s)
  - The second is the **pool size** (w)
- A common configuration is s=2 and w =2
- The most common type of pooling is called **max pooling (once the feature is detected in a particular quadrant then it is retained and sent to the resulting matrix)**.

5	1	3	5	7	9
3	4	4	0	7	8
6	1	8	6	3	4
1	3	5	8	4	6
1	1	5	3	1	2
1	2	2	6	4	6

Applying Pool Size (w=2, s=2)

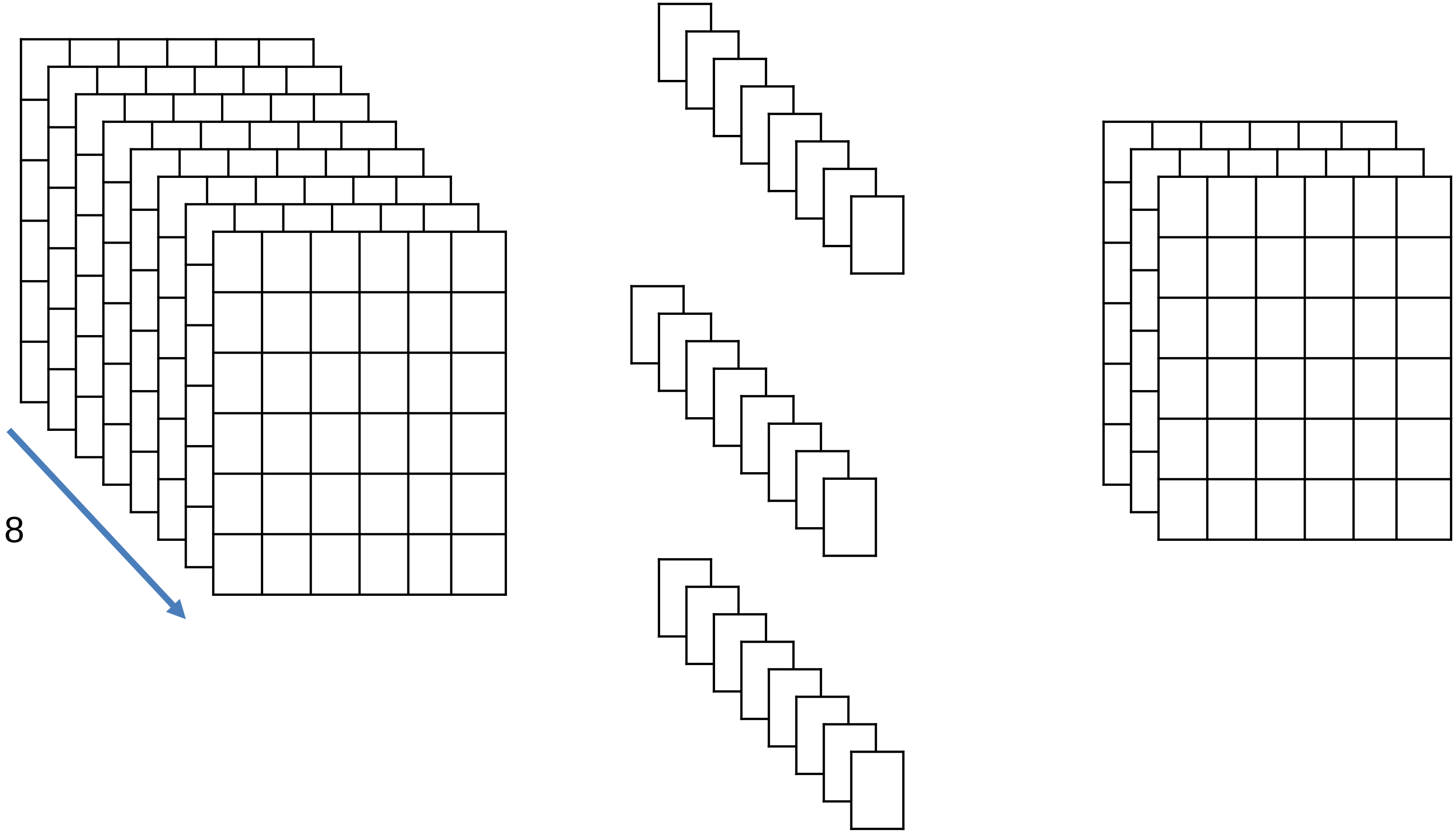
5	5	9
6	8	6
2	6	6

Result of Max Pooling Layer

1*1 Convolution

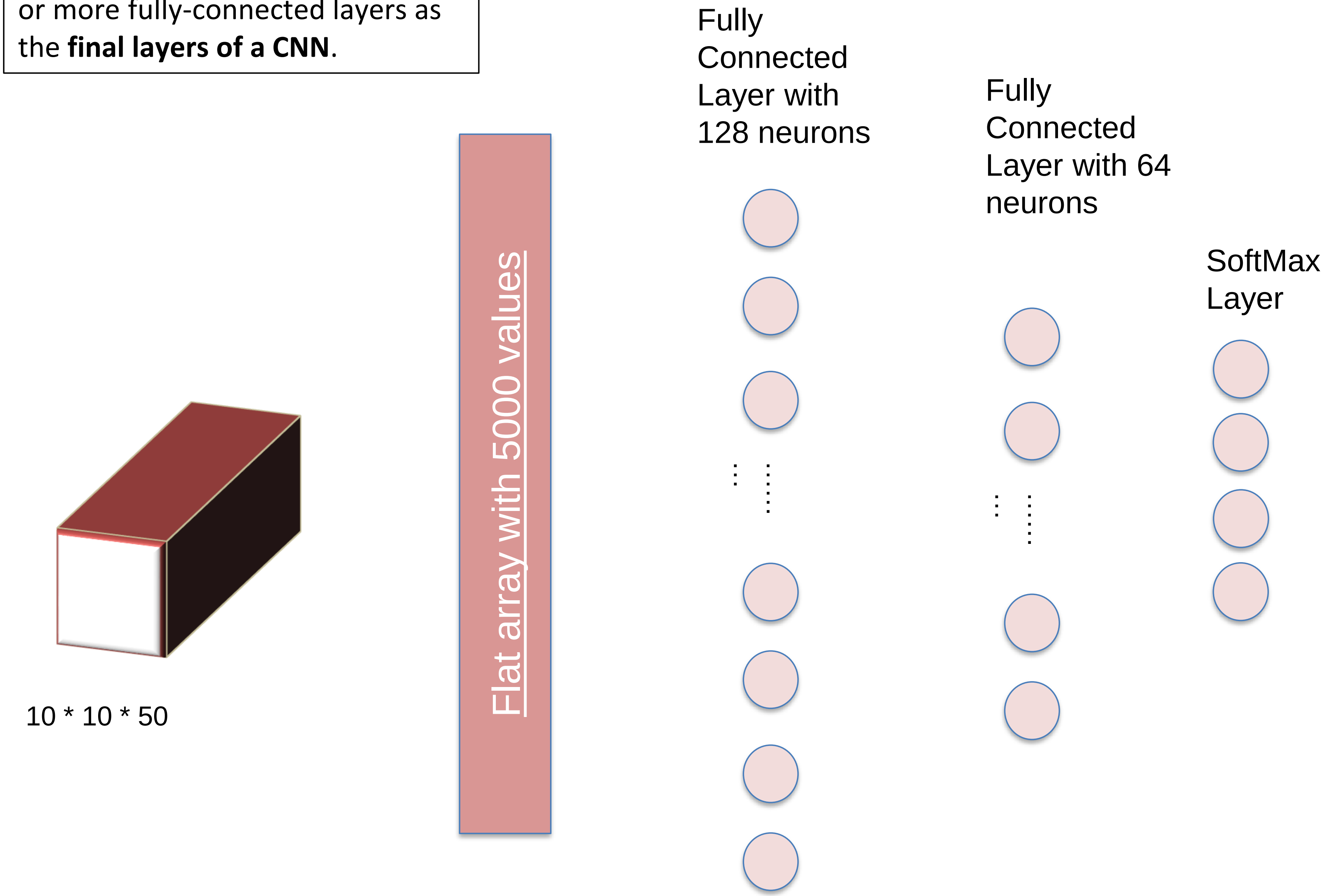
RECAP

- While a pooling layer is useful for reducing the width and height of the resulting feature map representation, a 1*1 convolutional layer will reduce the depth.
- This is very useful for reducing the computational complexity in large network (see inception network)



RECAP

It is very common to include one or more fully-connected layers as the **final layers of a CNN**.

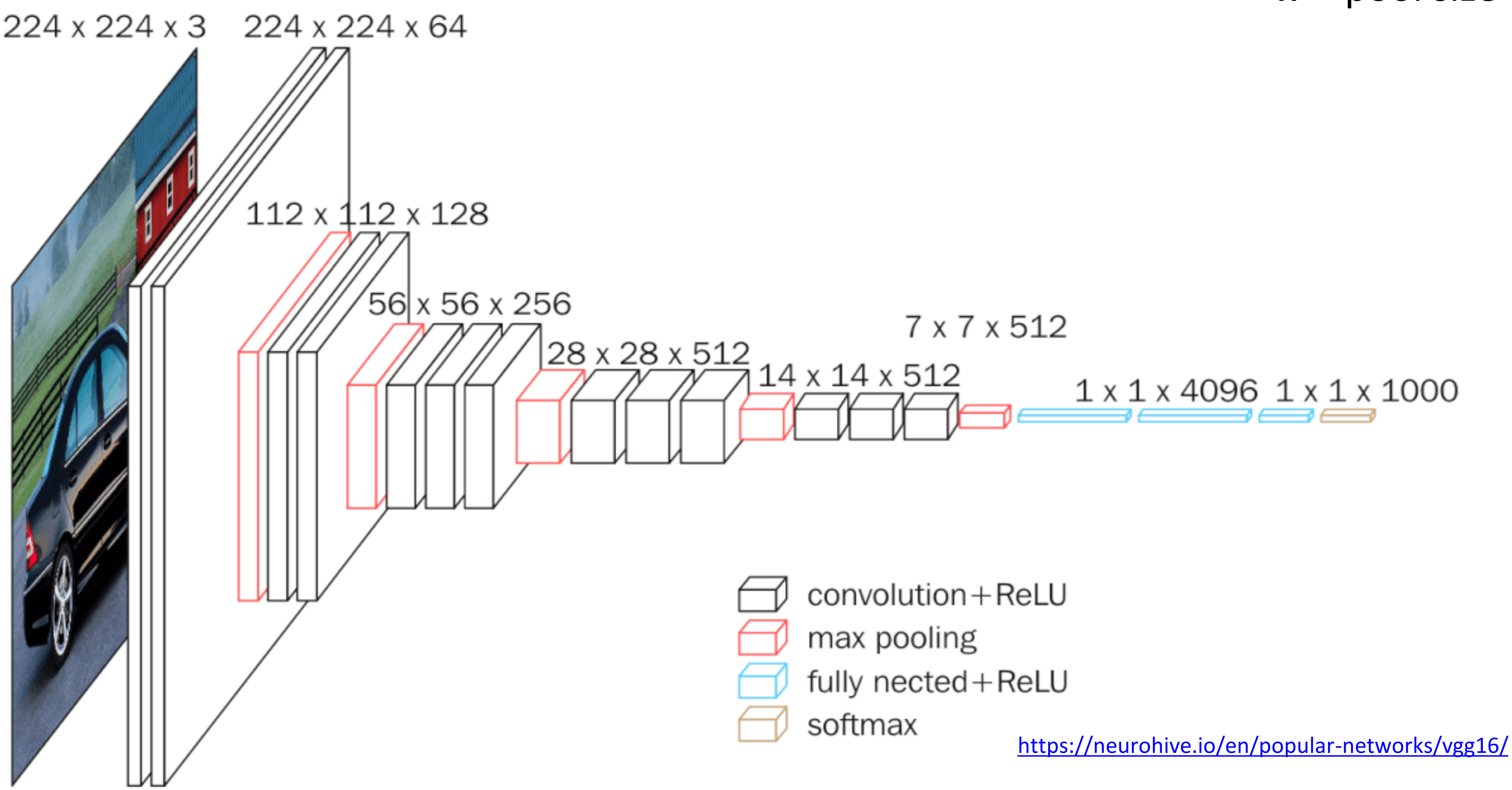

10 * 10 * 50
Flat array with 5000 values
Fully Connected Layer with 128 neurons
Fully Connected Layer with 64 neurons
SoftMax Layer

RECAP

VGG 16 Architecture

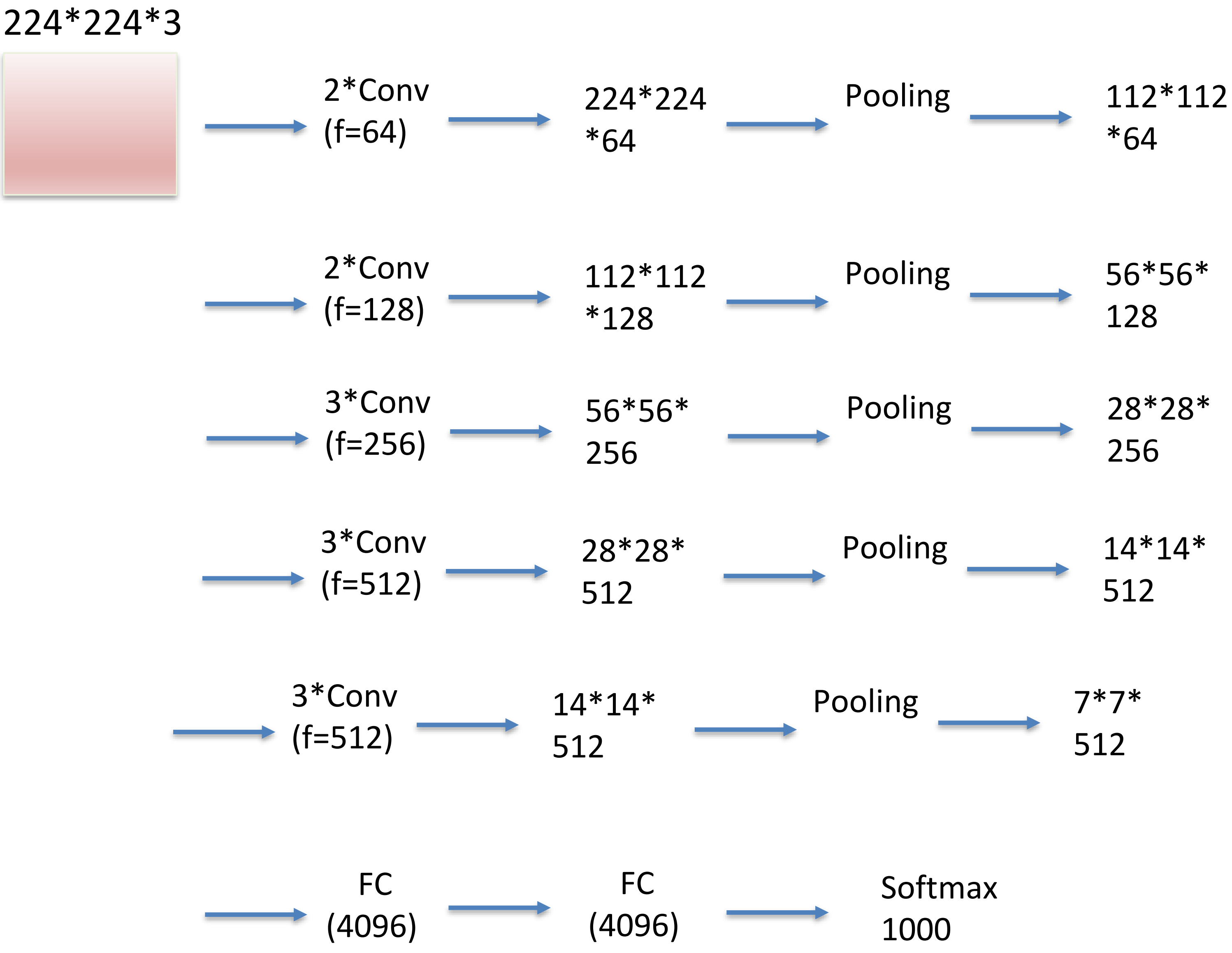
- For **convolution layers** VGG16 uses the following **k=3, s=1, p**
- For **pooling** it uses the following **w=3, s=2**
- Notice this network structure is quite regular (2* (2 convolutions followed by a pooling layer) followed by 3* (3 convolutions followed by a pooling layer).
- Also notice the number of filters **doubles** after each “block”

- s – stride length
- k – filter size
- p – padding
- f – number of filters
- w – pool size



https://neurohive.io/en/popular-networks/vgg16/

RECAP



224*224*3

2*Conv (f=64) → 224*224 *64 → Pooling → 112*112 *64

2*Conv (f=128) → 112*112 *128 → Pooling → 56*56* 128

3*Conv (f=256) → 56*56* 256 → Pooling → 28*28* 256

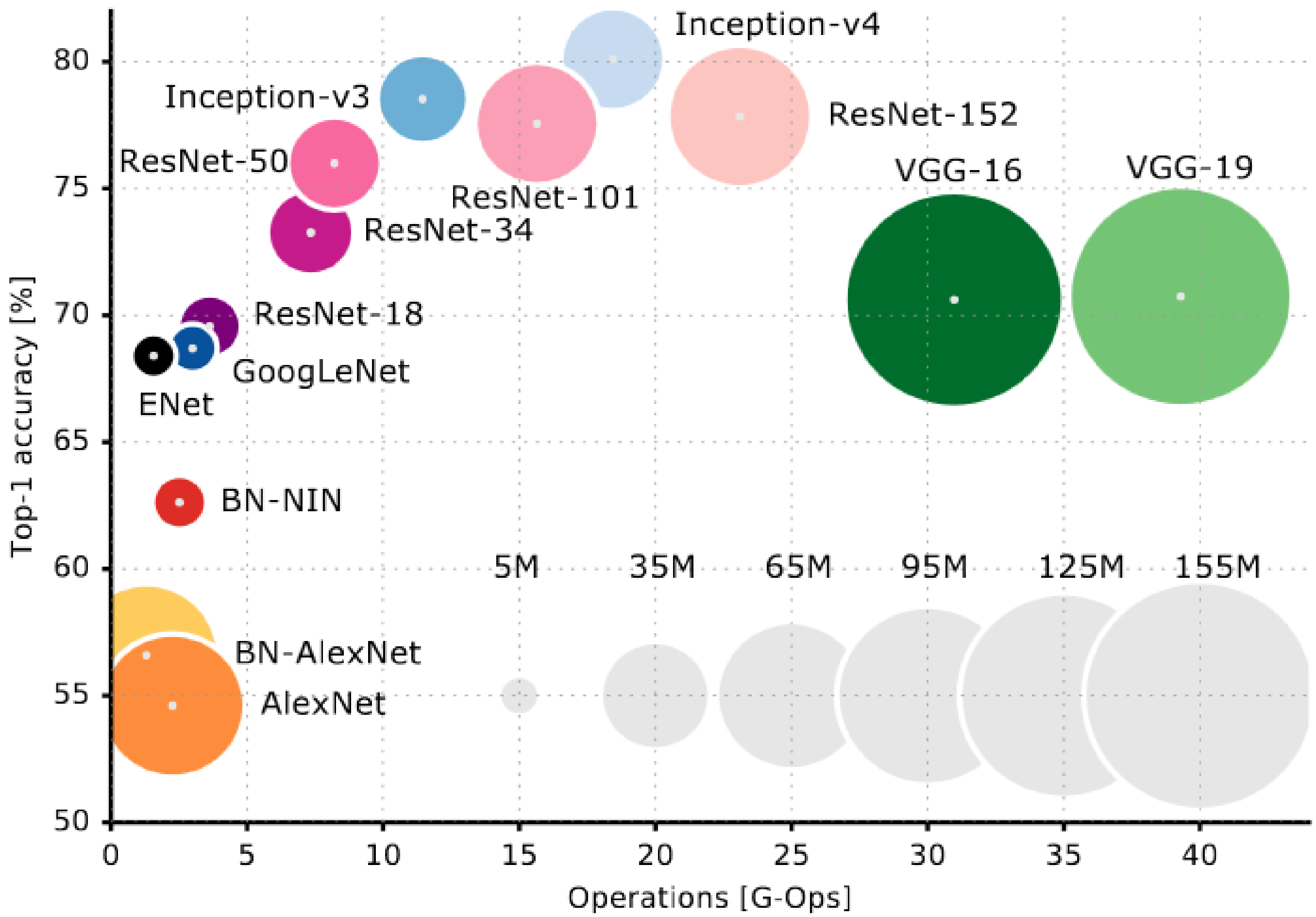
3*Conv (f=512) → 28*28* 512 → Pooling → 14*14* 512

3*Conv (f=512) → 14*14* 512 → Pooling → 7*7* 512

FC (4096) → FC (4096) → Softmax 1000

This graph represents the **rank 1 accuracy** versus the **number of operations** required for a single forward pass.
The size of the circles is proportional to the **number of network parameters**.

Interestingly we can see that VGG, even though it is widely used in many applications, is by far the most expensive architecture — both in terms of computational requirements and number of parameters (**138M**). [An Analysis of Deep Neural Network Models for Practical Applications]



Inception (GoogleNet) CNNs

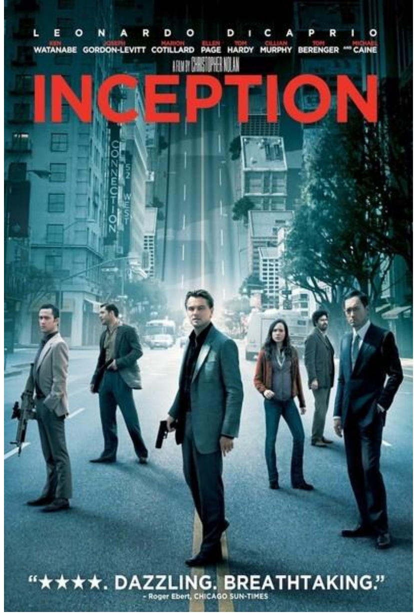
- The results shown on the previous slide also mean that the **inference time** per image for VGG is higher than the other networks.

- While the organized and regular structure of VGG is appealing a major **disadvantage** is that it requires significant **computational resources** (the number of learnable parameters, the total number of operations that need to be performed for a single forward pass and well as the inference time).

- The original VGG16 network has approximately **138M** parameters that need to be trained.

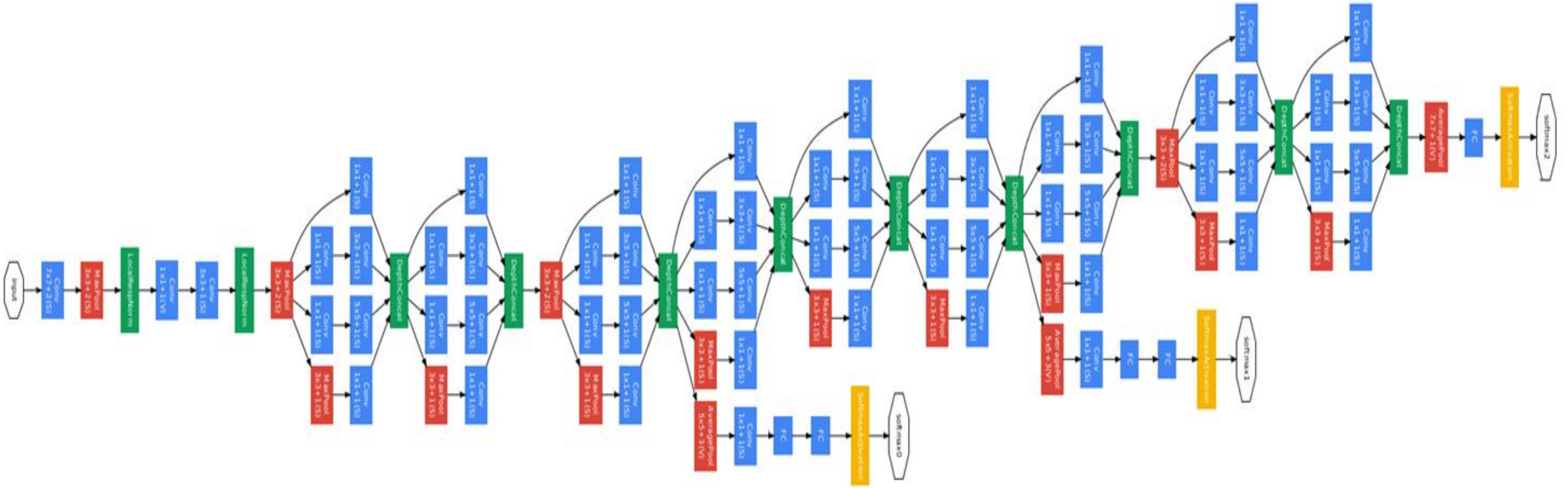
- In contrast the original **inception network** dramatically reduces the number **of learnable parameters to approx. 7 million**.

Inception Network

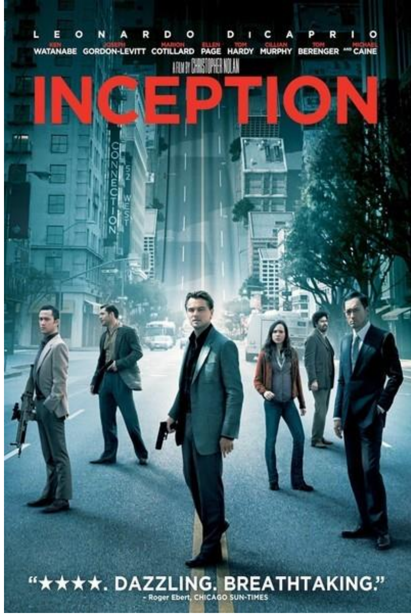

LEONARDO DICAPRIO
KEN WATANABE JOSEPH GORDON-LEVITT MARION COTILLARD ELLEN PAGE TOM HARDY CILLIAN MURPHY TOM BERENGER AND MICHAEL CAINE
A FILM BY CHRISTOPHER NOLAN
INCEPTION
"★★★★. DAZZLING. BREATHTAKING."
- Roger Ebert, CHICAGO SUN-TIMES

The model below is taken from the original paper on inception networks (Going Deeper with Convolutions 2014).
This is the original inception network (also called GoogLeNet)

You may notice that there is a repetitive pattern in the model. module. We take a closer look at this over the next few slides.



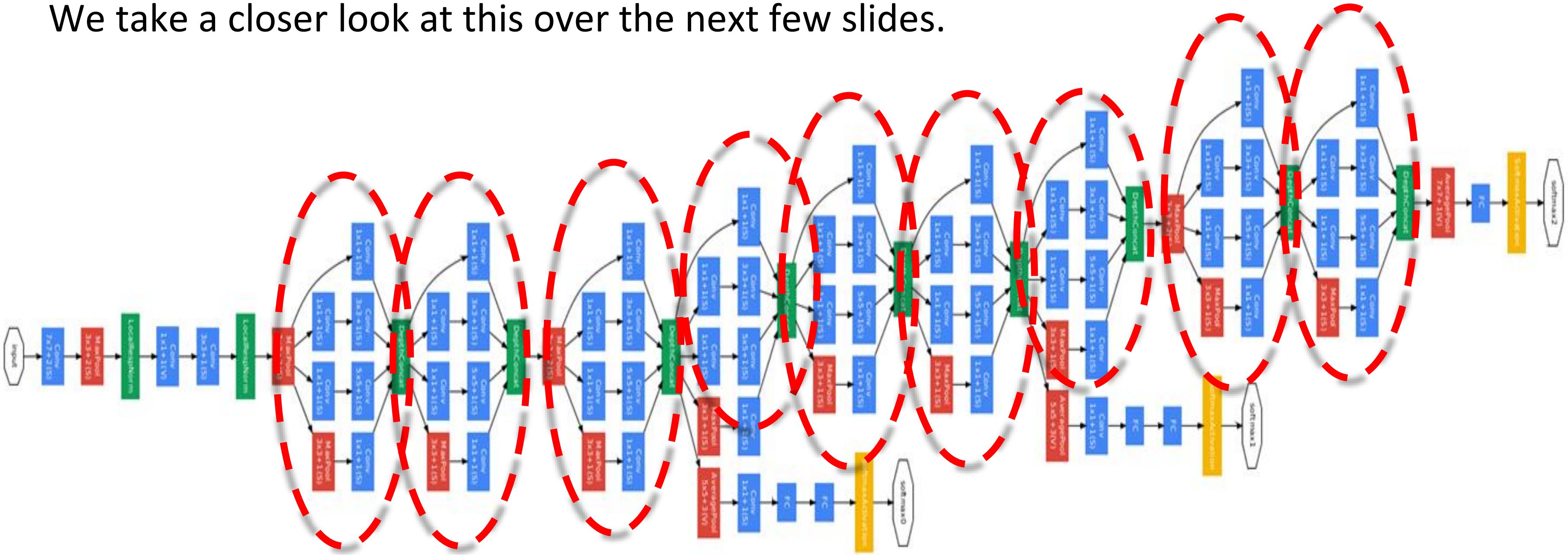
Inception Network


LEONARDO DICAPRIO
INCEPTION
"★★★★. DAZZLING. BREATHTAKING."
- Roger Ebert, CHICAGO SUN-TIMES

The model below is taken from the original paper on inception networks (Going Deeper with Convolutions 2014).
This is the original inception network (also called GoogLeNet)

You may notice that there is a repetitive pattern in the model. module. We take a closer look at this over the next few slides

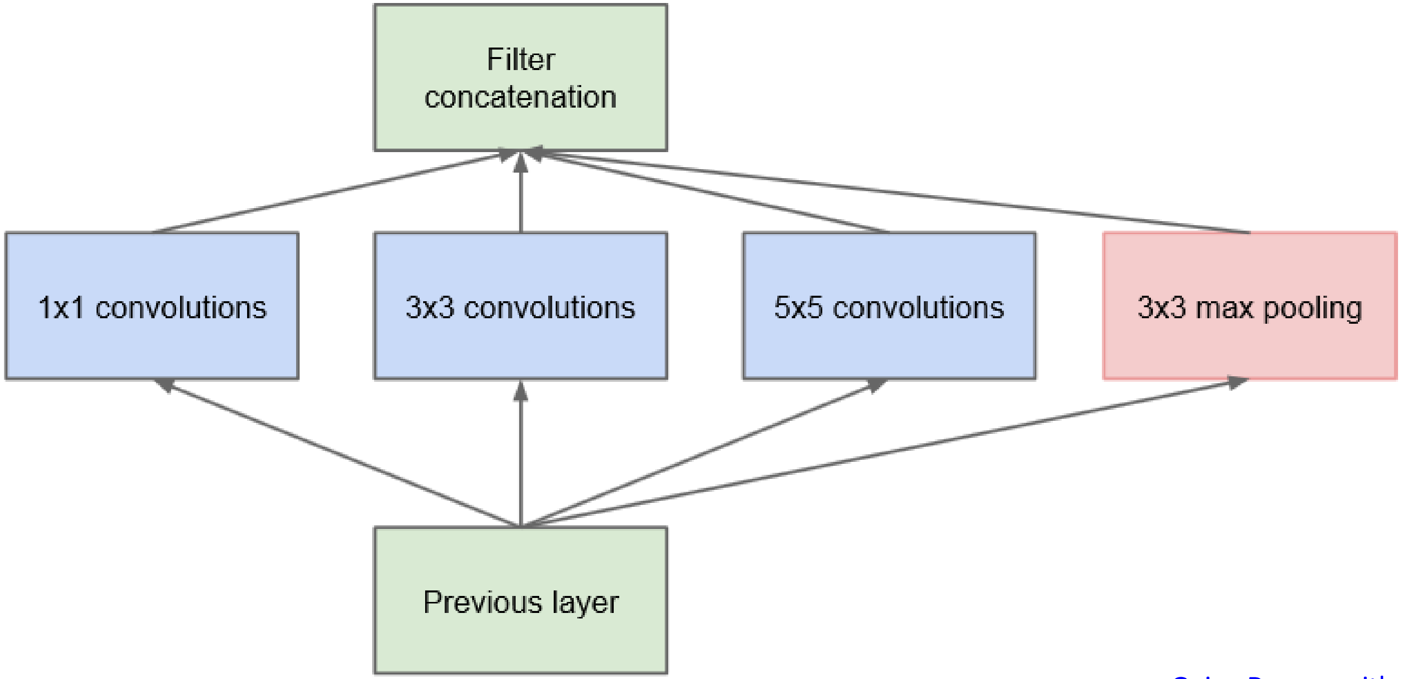
This repetitive set of layers is referred to as the **inception module**.
We take a closer look at this over the next few slides.



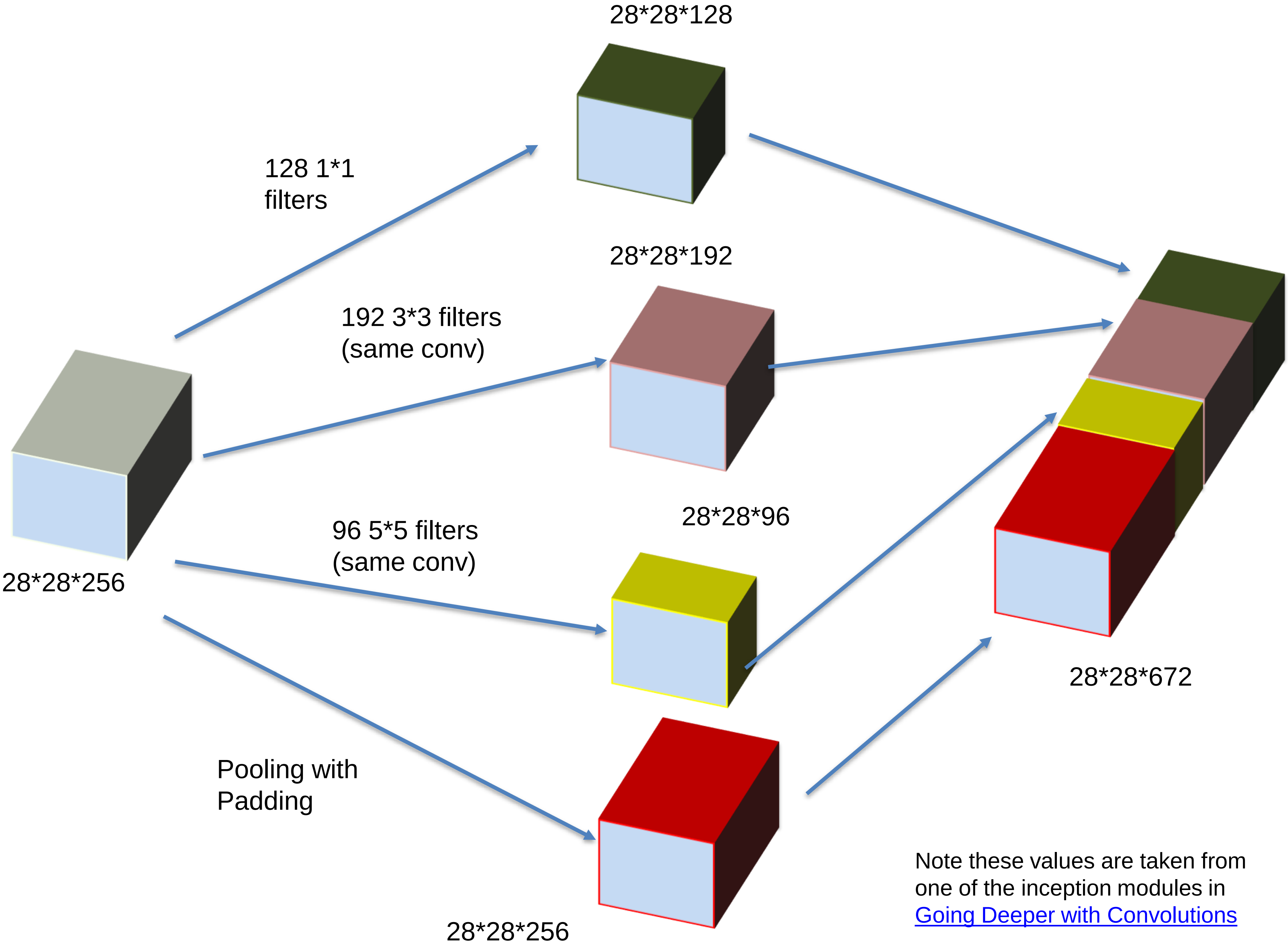
Inception (GoogleNet) CNNs

- One of the challenges when building a CNN is deciding if we should use a **pooling layer** or a **convolutional layer** at any particular stage of the network. And if using a convolutional layer should we use a **1*1**, **3*3** convolution or **5*5** convolution.

- The inception network attempts to use all four options and merge the result. The following is a basic view of the inception module.



Going Deeper with Convolutions



28*28*128

128 1*1
filters

28*28*192

192 3*3 filters
(same conv)

28*28*96

96 5*5 filters
(same conv)

28*28*256

Pooling with
Padding

28*28*256

28*28*672

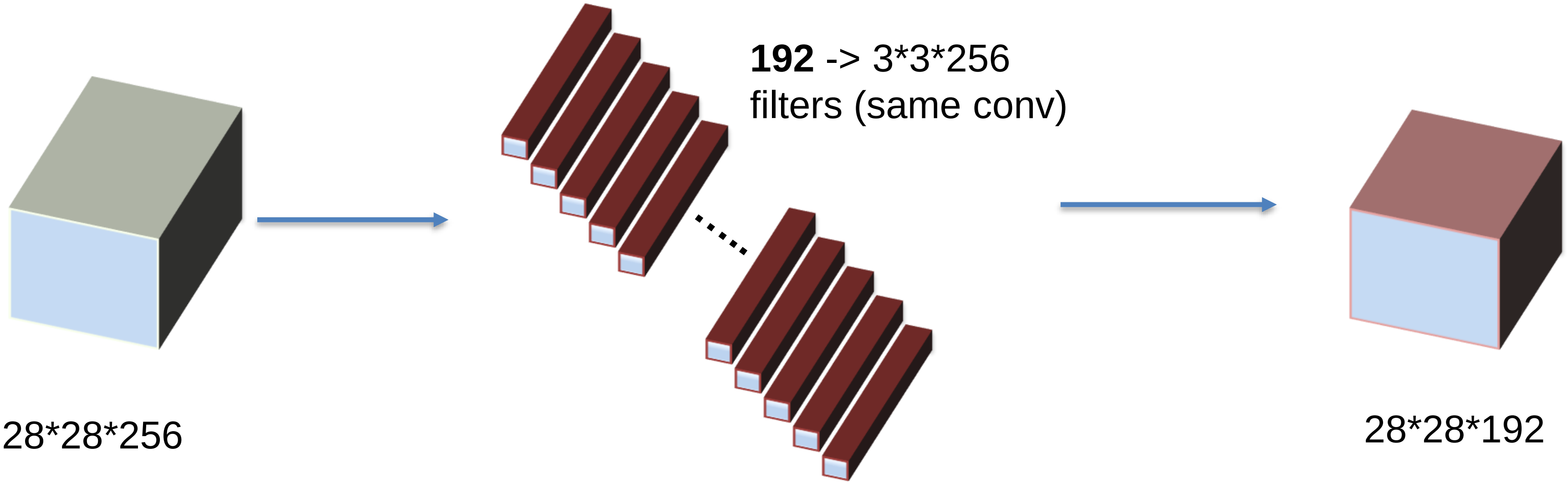
Note these values are taken from one of the inception modules in [Going Deeper with Convolutions]

High Computational Cost

- The problem with the example that we just described on the previous slide is that it has a very **high computation cost**. Let’s just take the number of operations involved in the 3*3 convolution we described on the previous slide.

- How many values are in the feature map on the right?

- How many calculations did we need to perform for each one of these?



192 -> 3*3*256
filters (same conv)

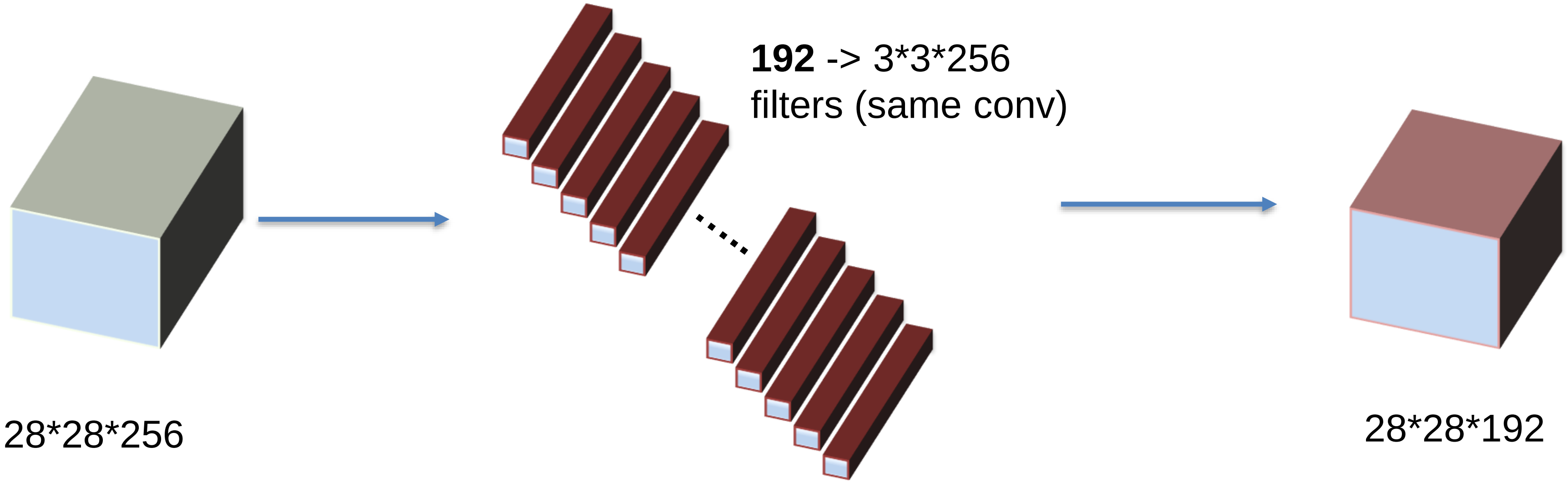
28*28*256

28*28*192

High Computational Cost

- The problem with the example that we just described on the previous slide is that it has a very **high computation cost**. Let’s just take the number of operations involved in the 3*3 convolution we described on the previous slide.

- The feature map that results from the convolution contains **28*28*192 (a total of 150, 528)** values.
- To calculate each one of these numbers we perform a convolution with the **3*3** filter across each of the **256 channels**, which is **3*3*256 (2304)** multiplication operations.
- Therefore, the total number of operations is **150,528* 2304 = 346,816,512** (approx **347M** operations).
- This is a very significant computational cost and a serious impediment to the practical application of this type of model given that this is just one of the elements in an inception module.



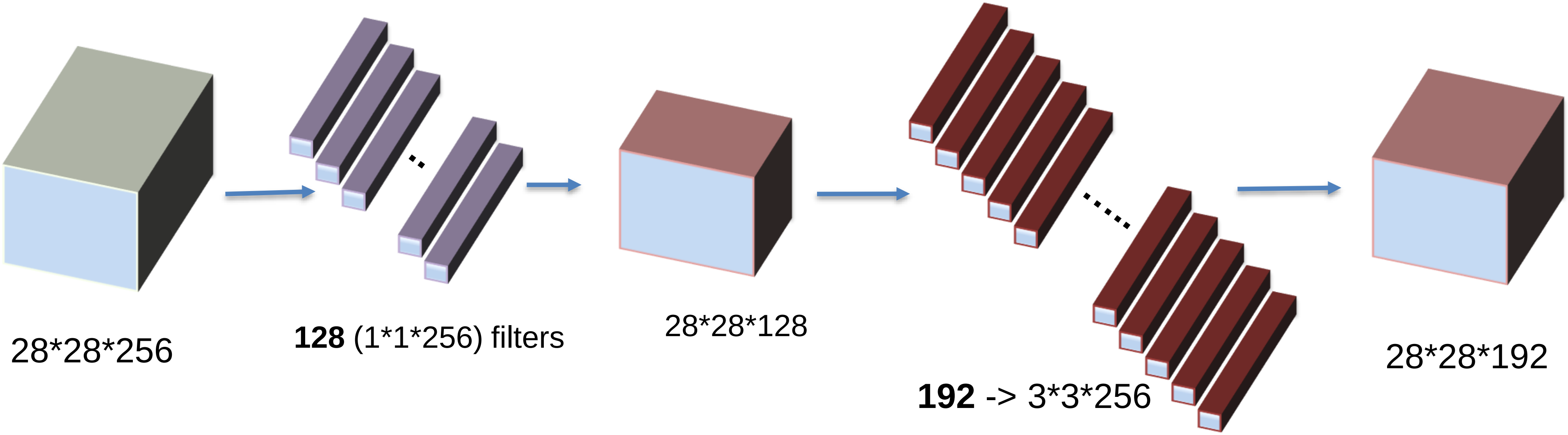
192 -> 3*3*256
filters (same conv)

28*28*256

28*28*192

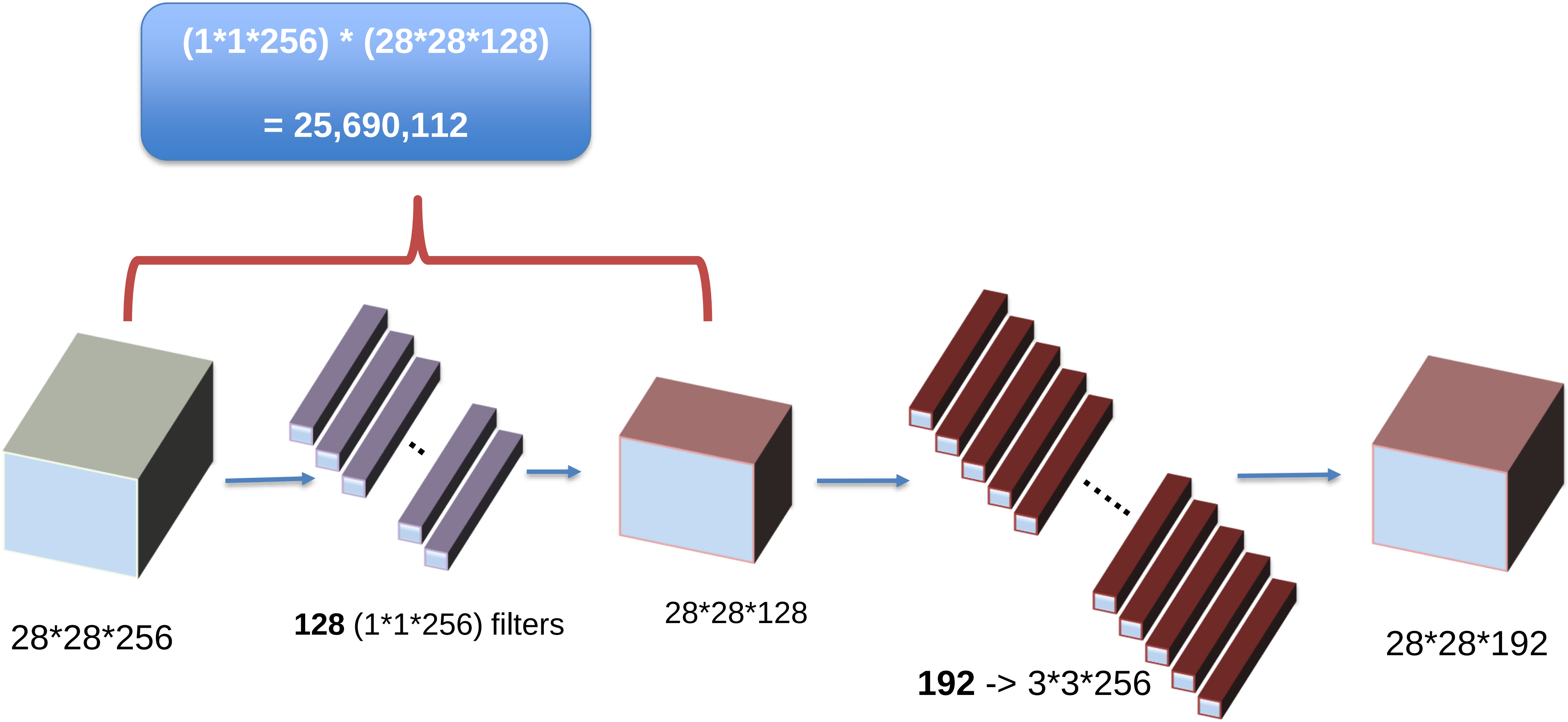
Inception (GoogleNet) CNNs

- The inception network uses **1*1 convolutions** to **significantly reduce the computation cost** of the inception network.
- Notice both input and output matrices below still have the same dimensions as they had in the previous slide but we inject this intermediate step that shrinks the number of channels in the original matrix from **256 to 128**. So let's work out the total number of operations.



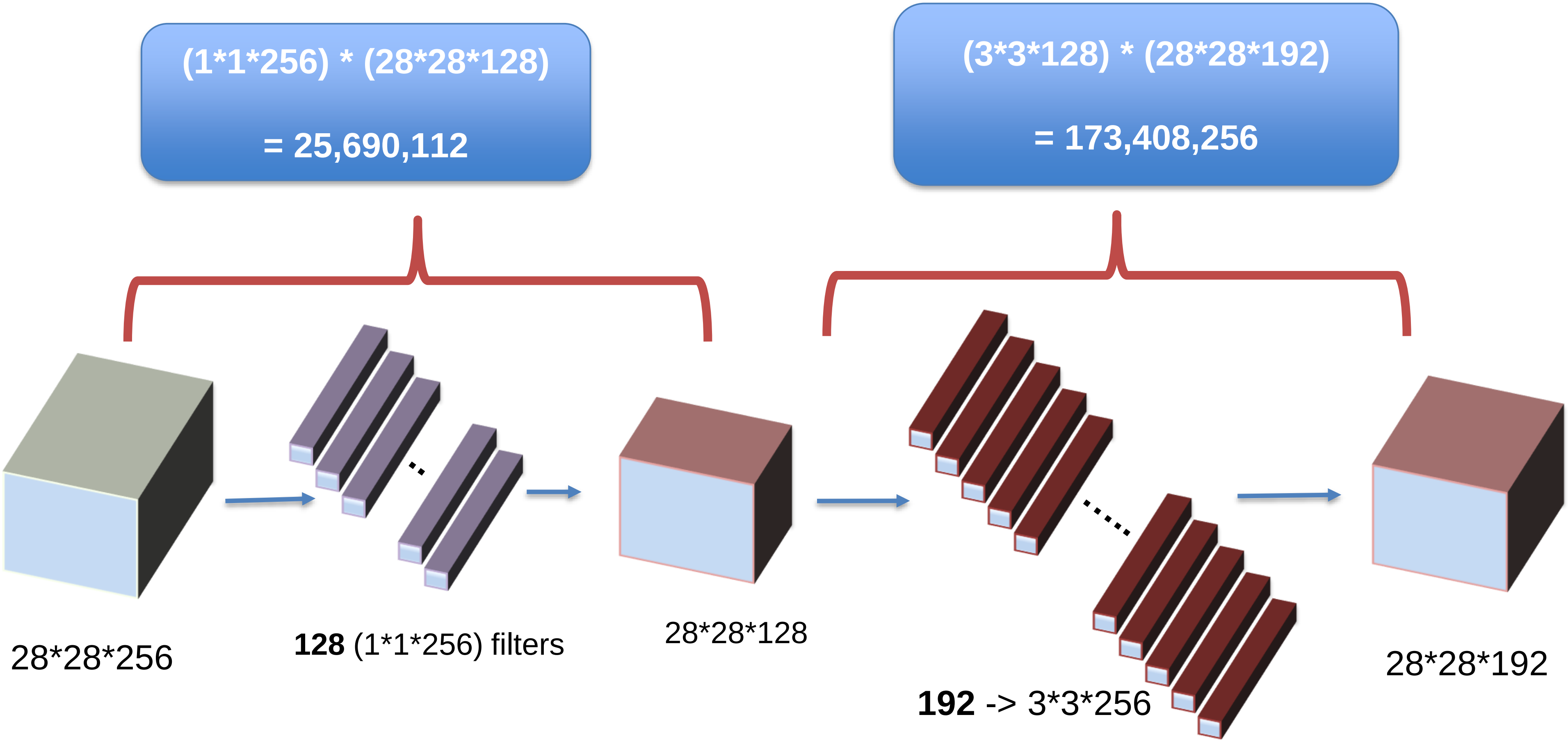
Inception (GoogleNet) CNNs

- The inception network uses **1*1 convolutions** to **significantly reduce the computation cost** of the inception network.
- Notice both input and output matrices below still have the same dimensions as they had in the previous slide but we inject this intermediate step that shrinks the number of channels in the original matrix from **256 to 128**. So let's work out the total number of operations.



Inception (GoogleNet) CNNs

- The inception network uses **1*1 convolutions** to **significantly reduce the computation cost** of the inception network.
- Notice both input and output matrices below still have the same dimensions as they had in the previous slide but we inject this intermediate step that shrinks the number of channels in the original matrix from **256 to 128**. So let’s work out the total number of operations.



(1*1*256) * (28*28*128)

= 25,690,112

(3*3*128) * (28*28*192)

= 173,408,256

28*28*256

128 (1*1*256) filters

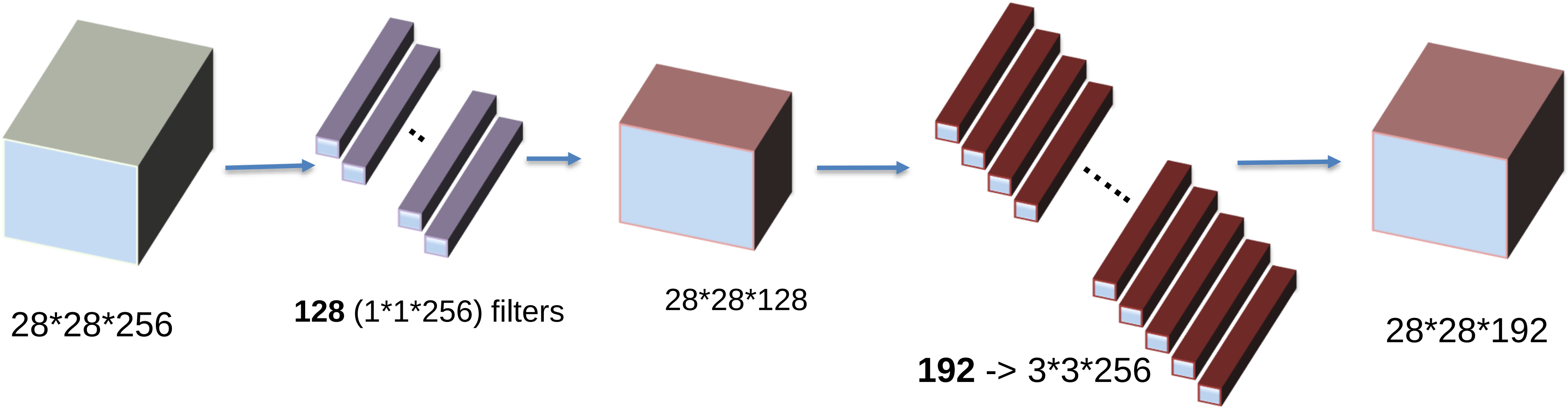
28*28*128

192 -> 3*3*256

28*28*192

Inception (GoogleNet) CNNs

- The inception network uses **1*1 convolutions** to **significantly reduce the computation cost** of the inception network.
- Notice both input and output matrices below still have the same dimensions as they had in the previous slide but we inject this intermediate step that shrinks the number of channels in the original matrix from **256 to 128**.
- Now the total number of operations are **25,690,112+173,408,256** = **199,098,368** (approx. 199M). Remember for the unaltered variant on the previous slide the number of operations was 347M (a saving of approximately 148M operations) .



28*28*256

128 (1*1*256) filters

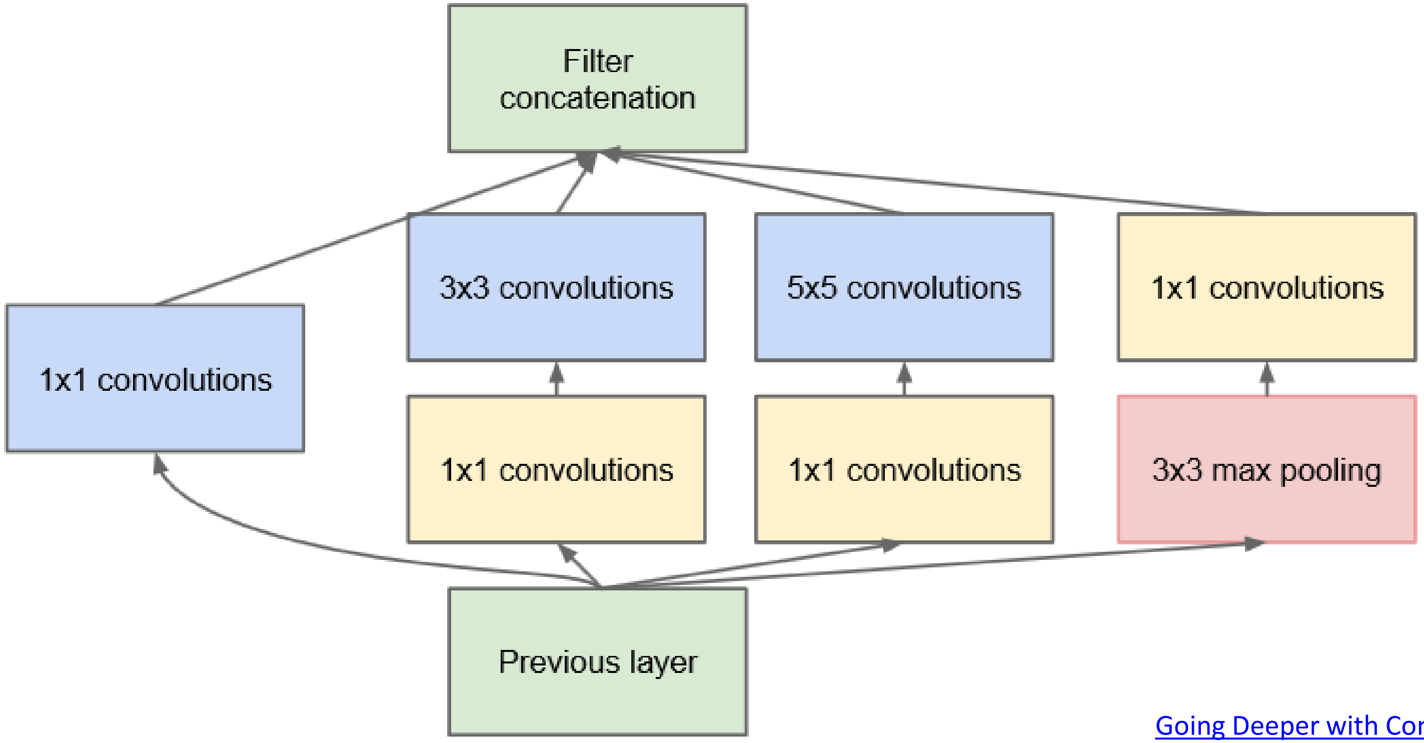
28*28*128

192 -> 3*3*256

28*28*192

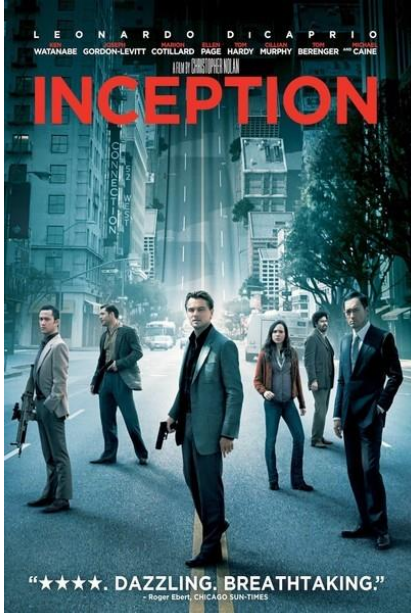
Inception (GoogleNet) CNN

- Therefore, the full inception module introduces 1*1 convolutions for the 3*3 and 5*5 convolution steps, which will significantly reduce the number of computations that must be performed.
- Notice a 1*1 convolution is also introduced after the pool layer. This is primarily to reduce the number of channels as the pooling layer produces the same number of channels as are the original layer.



Going Deeper with Convolutions

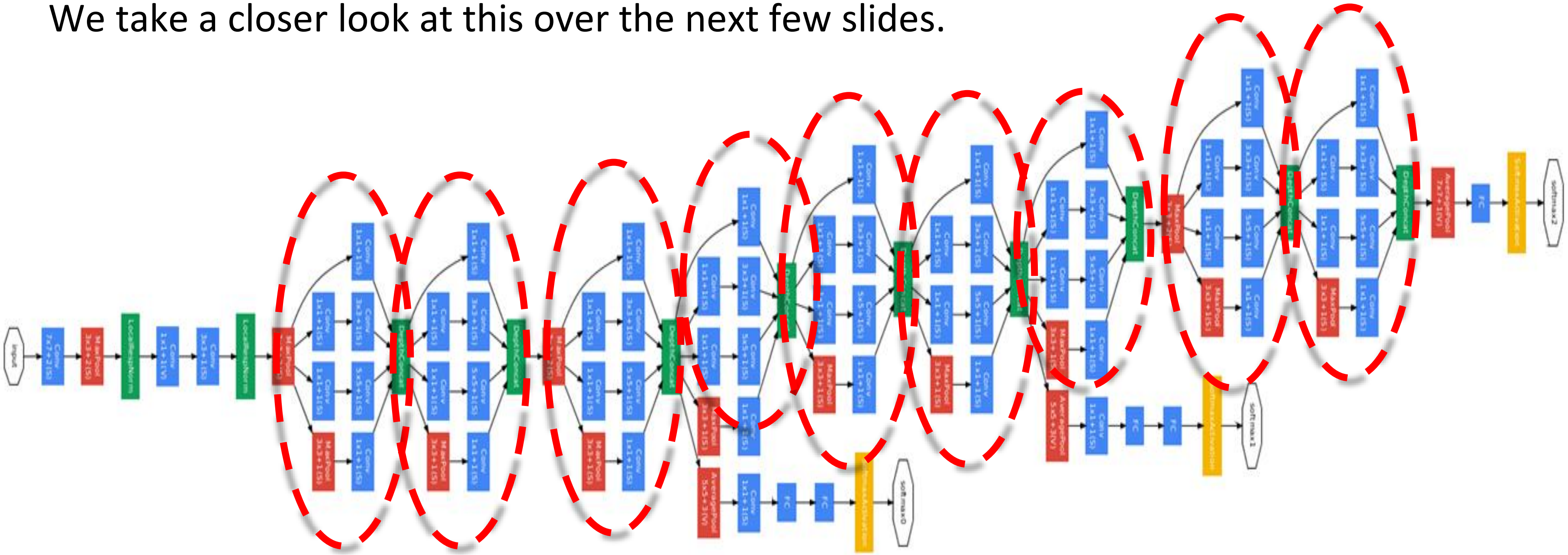
Inception Network


LEONARDO DICAPRIO
KEN WATANABE JOSEPH GORDON-LEVITT MARION COTILLARD ELLEN PAGE TOM HARDY CILLIAN MURPHY TOM BERENGER MICHAEL CAINE
A FILM BY CHRISTOPHER NOLAN
INCEPTION
"★★★★. DAZZLING. BREATHTAKING."
- Roger Ebert, CHICAGO SUN-TIMES

The model below is taken from the original paper on inception networks (Going Deeper with Convolutions 2014).
This is the original inception network (also called GoogLeNet)

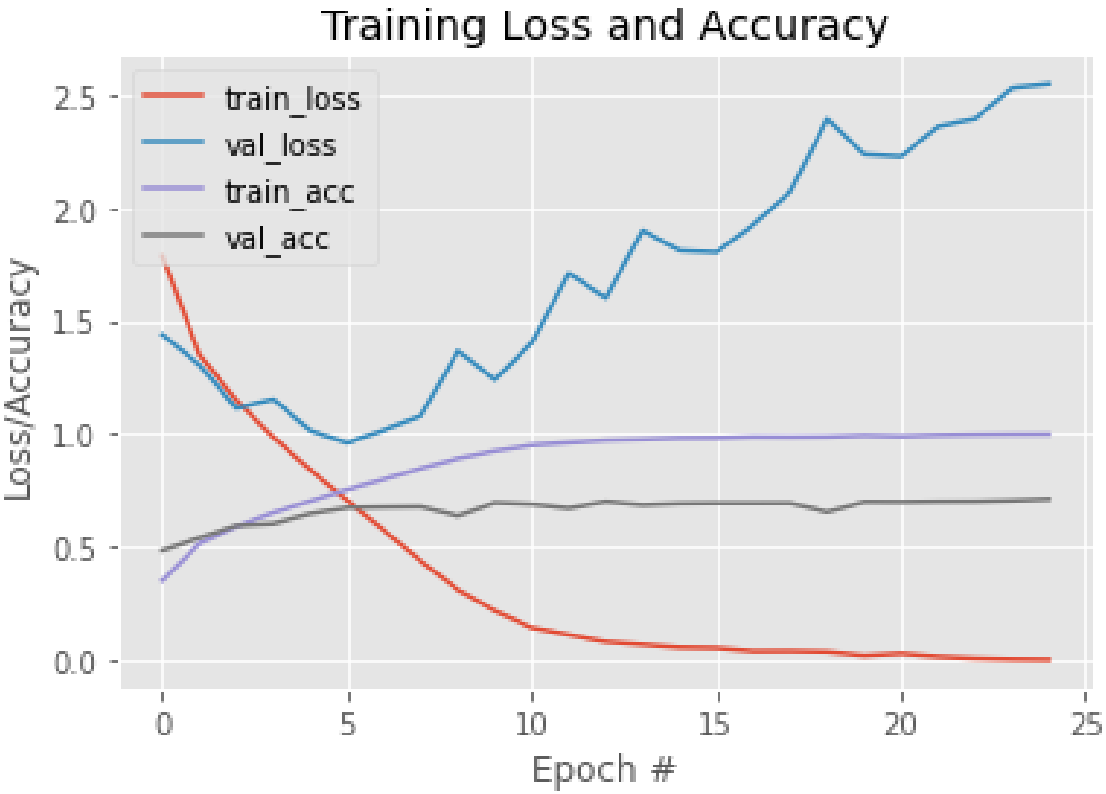
You may notice that there is a repetitive pattern in the model. module. We take a closer look at this over the next few slides

This repetitive set of layers is referred to as the **inception module**. We take a closer look at this over the next few slides.



Checkpointing Models

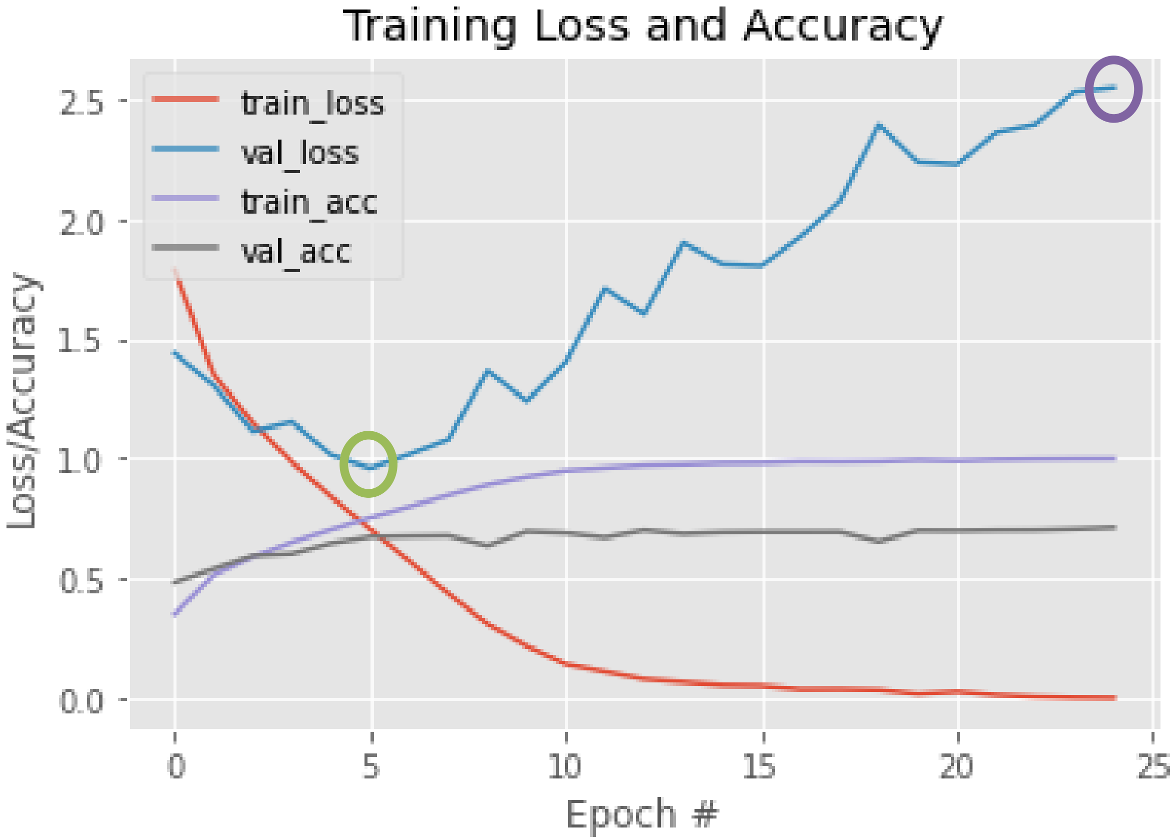
- So far we have left models train for a significant number of iterations.
- This is useful because we can easily identify the point where the model begins to overfit.
- For example below we can see the model is **overfitting after approx. 5 epochs**.
- However, one significant problem of course is that the model we end up with is the model after the final iteration of training. In the case below this model is seriously overfitting on the training data.



This the example where we previously looked at a shallowVGG applied to CIFAR10

Checkpointing Models

- Clearly it would be preferable to capture the model that has the lowest validation loss (which is circled in green below) rather than the model that will be returned at the end of the training process (highlighted in purple).

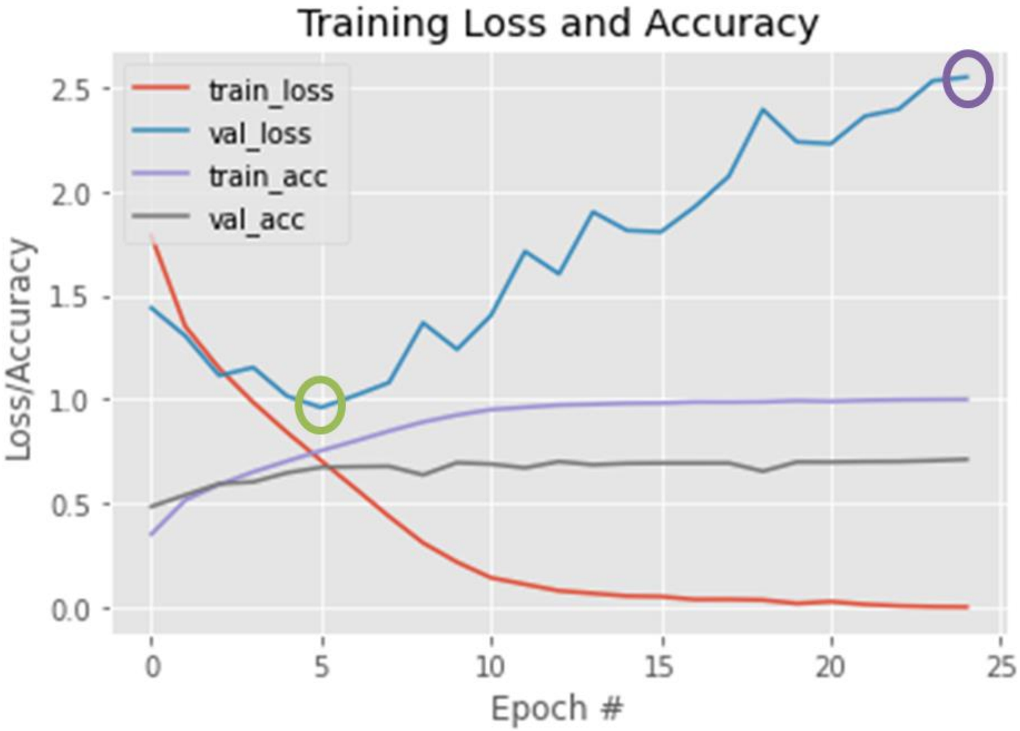


Checkpointing Models

- Checkpointing is a process that allows us to **save a model to disk each time there is an improvement during training**.

- An improvement above means either: (i) an increase in accuracy or (ii) a decrease in loss (typically validation as opposed to training).
- In the example below, as we train the model we could serialize it if the new model (for the current epoch) achieves a lower validation loss than any model we have seen in the training process so far.

  - Checkpointing is also useful if you have a **large model** that takes a **very long time to train**. For example, if something goes wrong during the training process then you don't have to restart from scratch.



Checkpointing Models – What are Callbacks

- A callback is basically a **set of functions** that can be provided to the **fit method**. These functions can then be applied at **given stages of the training procedure**.

- For example one callback that is automatically called is the **history callback**. After each epoch the history callback is called and it updates the history object with the loss and accuracy for the current epoch.

- Another alternative is an **early stopping callback** that will terminate the training process if a monitored metric such as validation loss has stopped improving.

- The **LearningRateScheduler** callback allows to update the learning rate used after a fixed epoch or a certain number of of epochs.

- You can pass a **list** of callbacks (as the **keyword argument callbacks**) to the .fit() method of the Sequential or Model classes.

Checkpointing Models – What are Callbacks

- The is also a **ModelCheckpoint callback** that allows you to checkpoint your model during the training process.

- The following are the main arguments for this class.

- **filepath**: A string that can contain formatting options such as the epoch number. For example the following is a common filepath (**weights.{epoch:02d}-{val_loss:.2f}.hdf5**)
- **monitor**: (typically ‘val_loss’or ‘val_accuracy’)
- **mode**: Should be minimizing or maximizing the monitor value (typically either ‘min’ or ‘max’)
- **save_best_only**: If this is set to true then it will only save the model for the current epoch if it’s metric values is better than what has gone before. However, if you set save_best_only to false it will save every model after each epoch (regardless of whether that model was better then previous models or not).

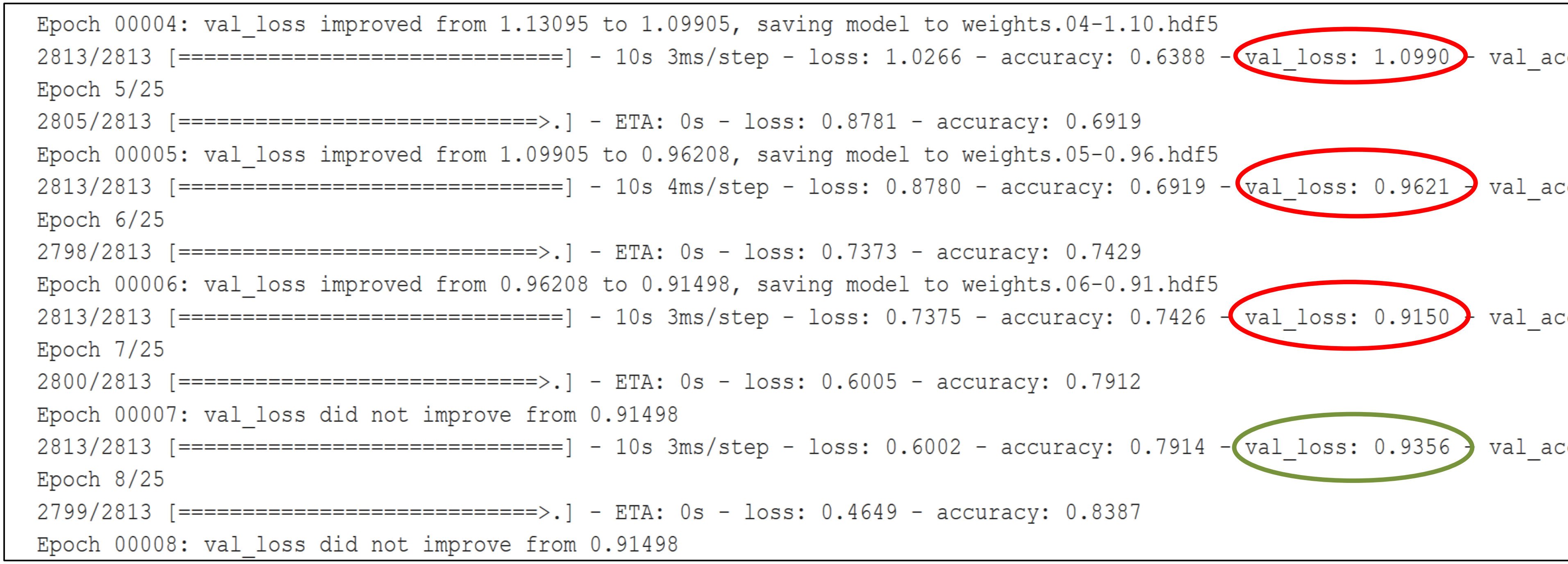
```
import tensorflow as tf
from tensorflow import keras

class ShallowVGGNet:

    @staticmethod
    def build(width, height, depth, classes):

        model = keras.models.Sequential()

        inputShape = (height, width, depth)


        # first CONV => CONV => POOL layer set
        model.add(keras.layers.Conv2D(32, (3, 3), padding="same",
                                              input_shape=inputShape, activation='relu'))
        model.add(keras.layers.Conv2D(32, (3, 3), padding="same",activation='relu'))
        model.add(keras.layers.MaxPooling2D(pool_size=(2, 2)))

        # second CONV => CONV => POOL layer set
        model.add(keras.layers.Conv2D(64, (3, 3), padding="same",activation='relu'))
        model.add(keras.layers.Conv2D(64, (3, 3), padding="same",activation='relu'))
        model.add(keras.layers.MaxPooling2D(pool_size=(2, 2)))

        # first (and only) set of FC => RELU layers
        model.add(keras.layers.Flatten())
        model.add(keras.layers.Dense(512,activation='relu'))

        # softmax classifier
        model.add(keras.layers.Dense(classes, activation='softmax'))

        # return the constructed network architecture
        return model
```

Associated Colab Notebook

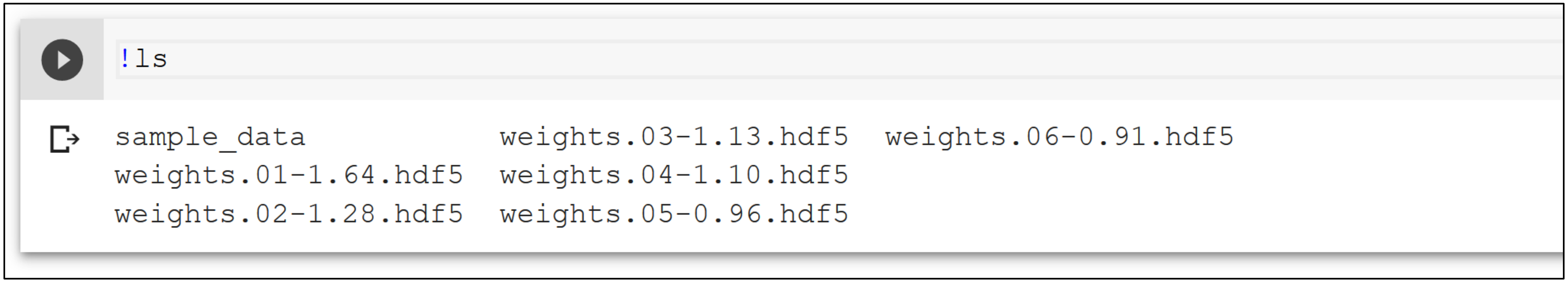
Notice we create an instance of ModelCheckpoint and set it to save a model for the current epoch if the validation loss for that model is less than any model we have seen so far in the training process.

Associated Colab Notebook

```
 NUM_EPOCHS = 25
 ((trainX, trainY), (testX, testY)) = tf.keras.datasets.cifar10.load_data()
 trainX = trainX.astype("float") / 255.0
 testX = testX.astype("float") / 255.0



 # initialize the optimizer and model
 print("Compiling model...")
 opt = keras.optimizers.SGD(lr=0.01)

 model = shallowVGGNet(width=32, height=32, depth=3, classes=10)

 model.compile(loss="sparse_categorical_crossentropy", optimizer=opt,
metrics=["accuracy"])

 fname = "weights.{epoch:02d}-{val_loss:.2f}.hdf5"
 checkpoint = tf.keras.callbacks.ModelCheckpoint(fname, monitor="val_loss",
          mode="min", save_best_only=True, verbose=1)

 print("Training network...")
 H = model.fit(trainX, trainY, batch_size=16,
        epochs=NUM_EPOCHS, validation_split=0.1,
          callbacks=[checkpoint])
```

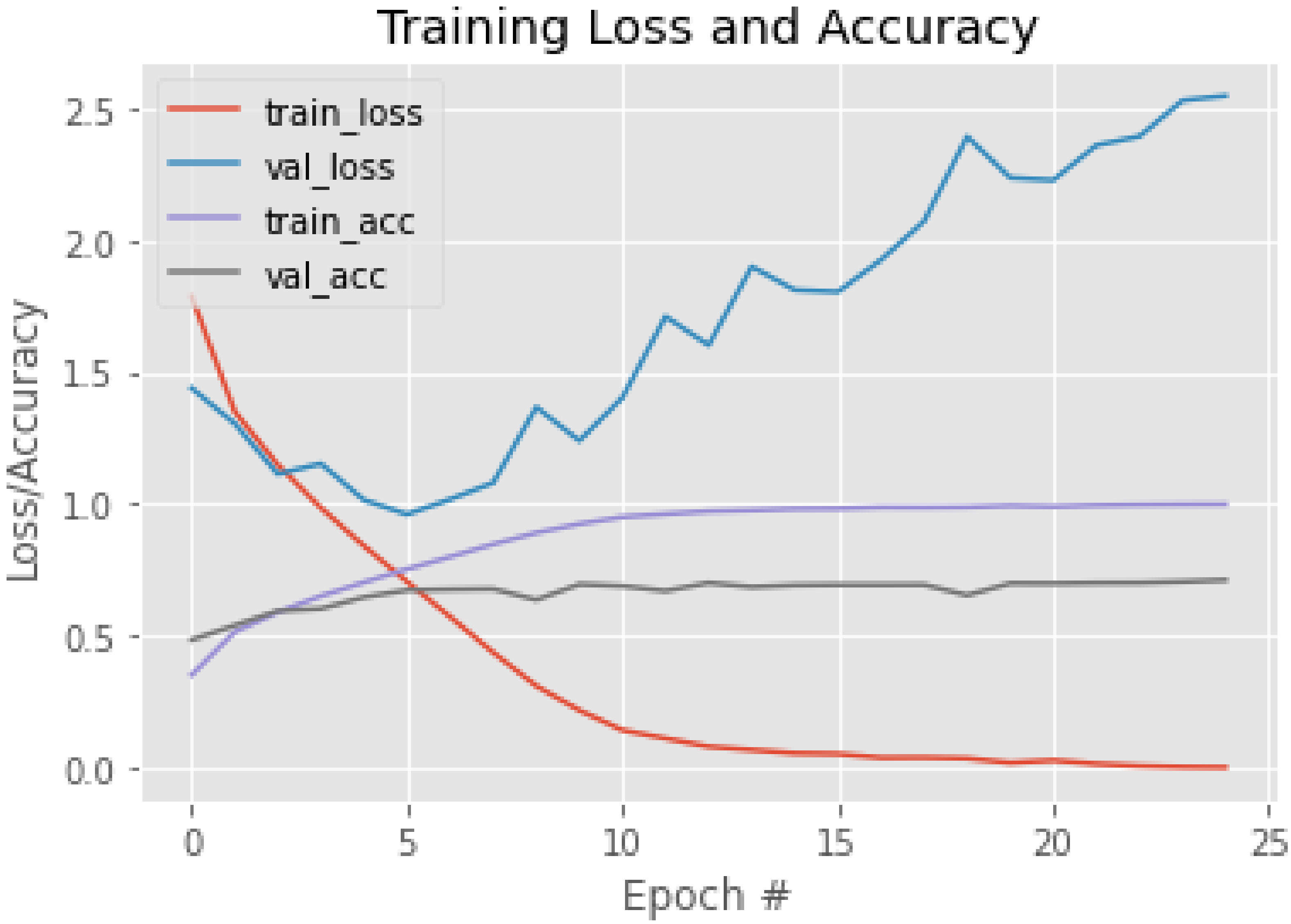


```
Epoch 00004: val_loss improved from 1.13095 to 1.09905, saving model to weights.04-1.10.hdf5
2813/2813 [==============================] - 10s 3ms/step - loss: 1.0266 - accuracy: 0.6388 - val_loss: 1.0990 - val_ac
Epoch 5/25
2805/2813 [============================>.] - ETA: 0s - loss: 0.8781 - accuracy: 0.6919
Epoch 00005: val_loss improved from 1.09905 to 0.96208, saving model to weights.05-0.96.hdf5
2813/2813 [==============================] - 10s 4ms/step - loss: 0.8780 - accuracy: 0.6919 - val_loss: 0.9621 - val_ac
Epoch 6/25
2798/2813 [============================>.] - ETA: 0s - loss: 0.7373 - accuracy: 0.7429
Epoch 00006: val_loss improved from 0.96208 to 0.91498, saving model to weights.06-0.91.hdf5
2813/2813 [==============================] - 10s 3ms/step - loss: 0.7375 - accuracy: 0.7426 - val_loss: 0.9150 - val_ac
Epoch 7/25
2800/2813 [============================>.] - ETA: 0s - loss: 0.6005 - accuracy: 0.7912
Epoch 00007: val_loss did not improve from 0.91498
2813/2813 [==============================] - 10s 3ms/step - loss: 0.6002 - accuracy: 0.7914 - val_loss: 0.9356 - val_ac
Epoch 8/25
2799/2813 [============================>.] - ETA: 0s - loss: 0.4649 - accuracy: 0.8387
Epoch 00008: val_loss did not improve from 0.91498
```



```
!ls
```

```
sample_data          weights.03-1.13.hdf5  weights.06-0.91.hdf5
weights.01-1.64.hdf5  weights.04-1.10.hdf5
weights.02-1.28.hdf5  weights.05-0.96.hdf5
```


Training Loss and Accuracy
train_loss
val_loss
train_acc
val_acc
Loss/Accuracy
2.5
2.0
1.5
1.0
0.5
0.0
0
5
10
15
20
25
Epoch #

Loading the model we obtained after epoch 5 is very easy.

We can create a new instance of our shallowVGGNet and then call load_weights.

Remember you still have to compile the model as normal (expect we obviously don’t need to call fit as we already have the desired weights for the model)

```
model = shallowVGGNet(width=32, height=32, depth=3, classes=10)
model.load_weights("weights.06-0.91.hdf5")

opt = keras.optimizers.SGD(lr=0.01)

model.compile(loss="sparse_categorical_crossentropy", optimizer=opt,
metrics=["accuracy"])


((trainX, trainY), (testX, testY)) = tf.keras.datasets.cifar10.load_data()
trainX = trainX.astype("float") / 255.0
testX = testX.astype("float") / 255.0
scores = model.evaluate(testX, testY, verbose=0)

print (scores[0])
```

Associated Colab Notebook

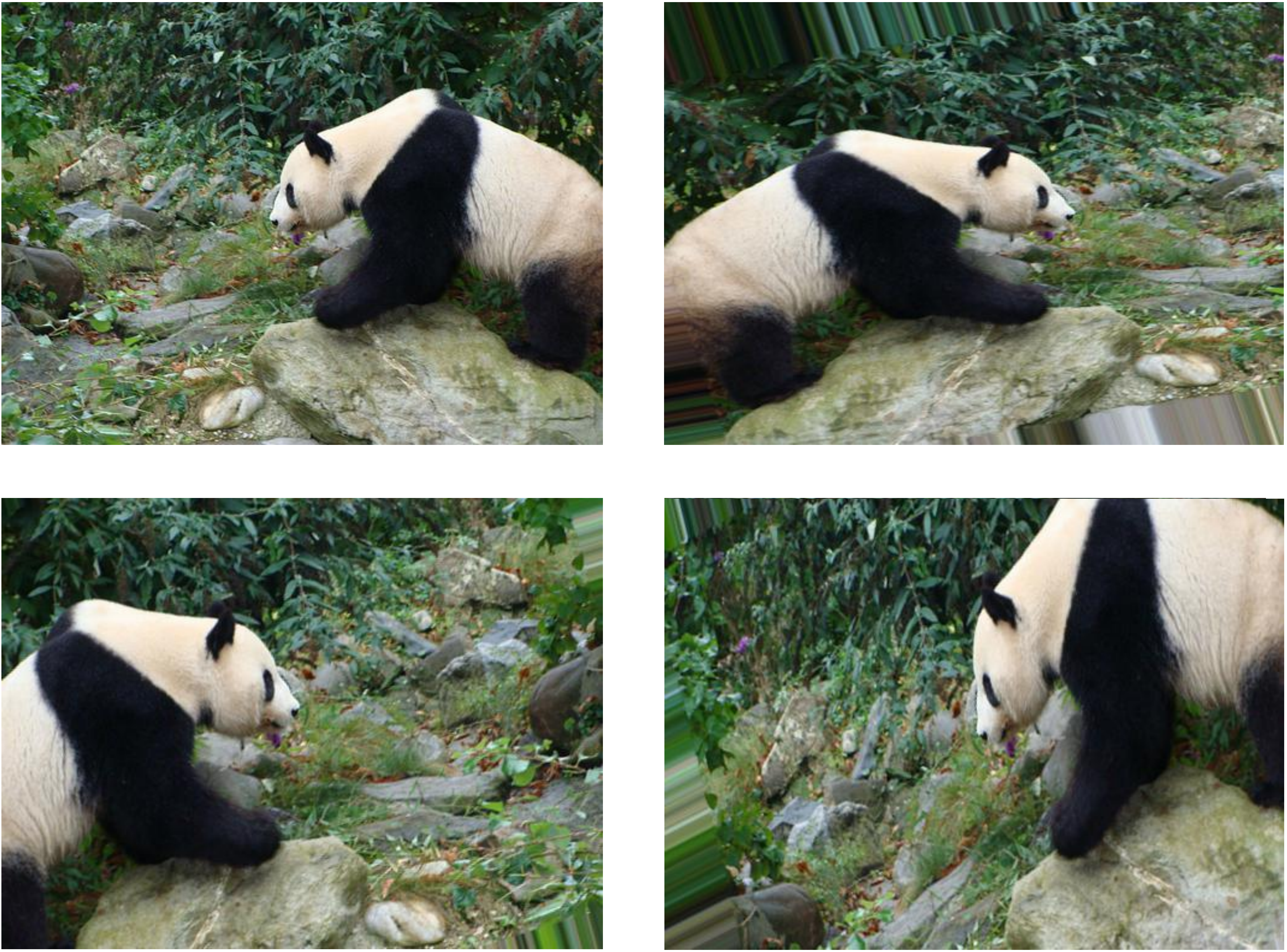
Data Augmentation

- Computer vision problems are **data hungry**.
- Typically convolutional neural networks will perform better when given more data.

- Data augmentation is a useful method that can allow us to **artificially generate more training data** for our model.

- The end result is that a **network consistently sees “new” training data points** generated from the original training data.

- This has two benefits
  - For an image dataset with a **small number of images** it can help generate additional data and improve overall performance.
  - Because the model is continually training on “different” images it **helps to alleviate overfitting** (therefore, data augmentation can be viewed as a regularization technique).



ImageDataGenerator

- Keras allows us to easily provide data augmentation using the Image Data Generator class.

    - The ImageDataGenerator class facilitates the **generation of batches of images** with real time data augmentation.

- The following are a selection of the arguments for the ImageDataGenerator.
    - **rotation_range**: degrees (0 to 180).
    - **width_shift_range** and **height_shift_range** are used for horizontal and vertical shifts, respectively.
    - **zoom_range**: amount of zoom. If scalar z, zoom will be randomly picked in the range [1-z, 1+z].
    - **horizontal_flip**: whether to randomly flip images horizontally.
    - **vertical_flip**: whether to randomly flip images vertically.
    - **rescale**: rescaling factor. This multiplies the image by the rescaling factor. This can be a useful way of normalising the data by specifying a rescaling factor of 1.0/255.

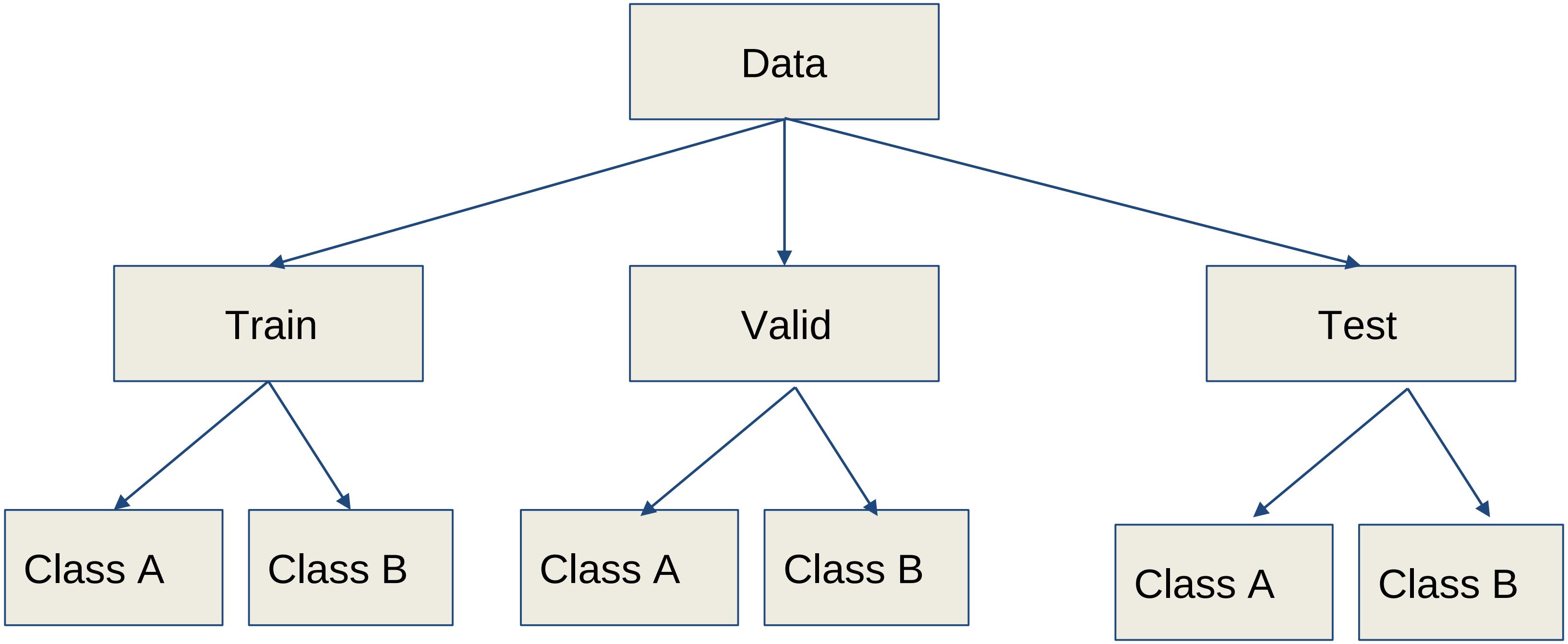
(A full list of the broad range of parameters can be accessed here)

ImageDataGenerator – flow()

- The ImageDataGenerator has a number of methods. One very useful method is:
    - **flow**: Takes feature data & label arrays and return an iterator that generates **batches of augmented data**.
        - **x**: Input data. NumPy array of rank 4 (Expects to receive a set of images)
        - **y**: Class labels
        - **batch_size**: Int (default: 32).
        - **save_to_dir**: Allows you to optionally specify a directory to which to save the augmented pictures being generated
        - **save_prefix**: Str. Prefix to use for filenames of saved pictures
        - **save_format**: one of "png", "jpeg" (only relevant if save_to_dir is set). Default: "png".
    - The flow function **returns an Iterator yielding tuples of (x, y)** where x is a list of NumPy arrays (in the case with additional inputs) and y is a NumPy array of corresponding labels.

ImageDataGenerator - flow_from_directory()

- Another useful method is **flow_from_directory**, which takes a **path to a directory** and generates batches of augmented data.
- This method expects the data to be stored using a specific type of directory structure. Each class must have it’s own folder inside the training or validation folder.



ImageDataGenerator - flow_from_directory()

- The following are the main arguments for the **flow_from_directory** method.

  - **directory**: Path to the target directory. It should contain one subdirectory per class. Can support PNG, JPG, BMP, PPM or TIF.

  - **target_size**: Tuple of integers (**height, width**), default: (256, 256). This is very useful as it allows you to resize the all images to a standard size.

  - **class_mode**: One of "categorical", "binary", "sparse", "input", or None. Default: "categorical". Determines the type of label arrays that are returned:
    - "categorical" will be 2D one-hot encoded labels,
    - "sparse" will be 1D integer labels
    - "binary" will be 1D binary labels

  - **batch_size**: Size of the batches of data (default: 32)

- The flow_from_directory method **returns an Iterator that yields tuples of (x, y)** where x is a NumPy array containing a batch of images and y is a NumPy array of corresponding labels.

ImageDataGenerator - flow_from_directory()

- Please be aware the flow_from_directory can **slow down your model training process** as each time you generate a new batch you are reading directly from disk before making the data augmentation modifications and then passing the batch into the network.

Building a model using Generators

- The **fit method** also allows you to train your current model on data generated **batch-by-batch by a Python generator**.

- Rather than passing **training and validation data** in the normal way we can just pass **the generator object**.

- Therefore, rather then providing feature training data and associated class labels we can pass the fit method the generator object to generate new training data.

- New parameters we need to set are
- **steps_per_epoch**: Integer. Total number of steps (batches of samples) to yield from generator before one epoch is complete. Typically set to (num_samples / batch_size)

- **validation_data** (can be a generator or a tuple as normal)
- **validation_steps**: If validation_data is a generator. Plays a similar role to the steps_per_epoch function. Total number of steps (batches of samples) to yield from validation_data generator before stopping at the end of every epoch. Normally equal to the size of validation dataset divided by the batch size

This example is only going to take in a **single image** and **augment** that image. The image is called panda.jpg and is stored in the same directory as the python code file.

The first two lines of code load the image as a NumPy array.

The shape of the image is **(375, 500, 3).**

When we pass the image to the flow function later it will expect to receive a batch of images. Therefore, it expects to receive a 4D data structure. Therefore, we add one extra dimension to the existing data structure to make its shape **(1, 375, 500, 3)**

```
import numpy as np
import tensorflow as tf


# load the input image, convert it to a NumPy array, and then
# reshape it to have an extra dimension
image = tf.keras.preprocessing.image.load_img("./panda.jpg")
image = tf.keras.preprocessing.image.img_to_array(image)
print ((image.shape))

image = np.expand_dims(image, axis=0)
print ((image.shape))



# construct the image generator for data augmentation
aug = tf.keras.preprocessing.image.ImageDataGenerator(
rotation_range=20, width_shift_range=0.1, shear_range=0.2,
zoom_range=0.4, horizontal_flip=True)

imageGen = aug.flow(image, batch_size=1,
                    save_to_dir='output', save_prefix='panda',
                                     save_format='jpeg')
i = 0
for batch in imageGen:
    i += 1
    if i > 20:
        break
```

First we create an instance of the **ImageDataGenerator**. Notice we have given a range of augmentation arguments.

The **flow function** returns a generator, which is a Python **iterator** object that is used to construct our augmented images.

We'll pass in our input image, a batch_size of 1 (since we are only augmenting one image), along with a few additional parameters to specify the output image file paths, the prefix for each file path,

and the image file format.

Finally we start looping over each image in the imageGen generator.

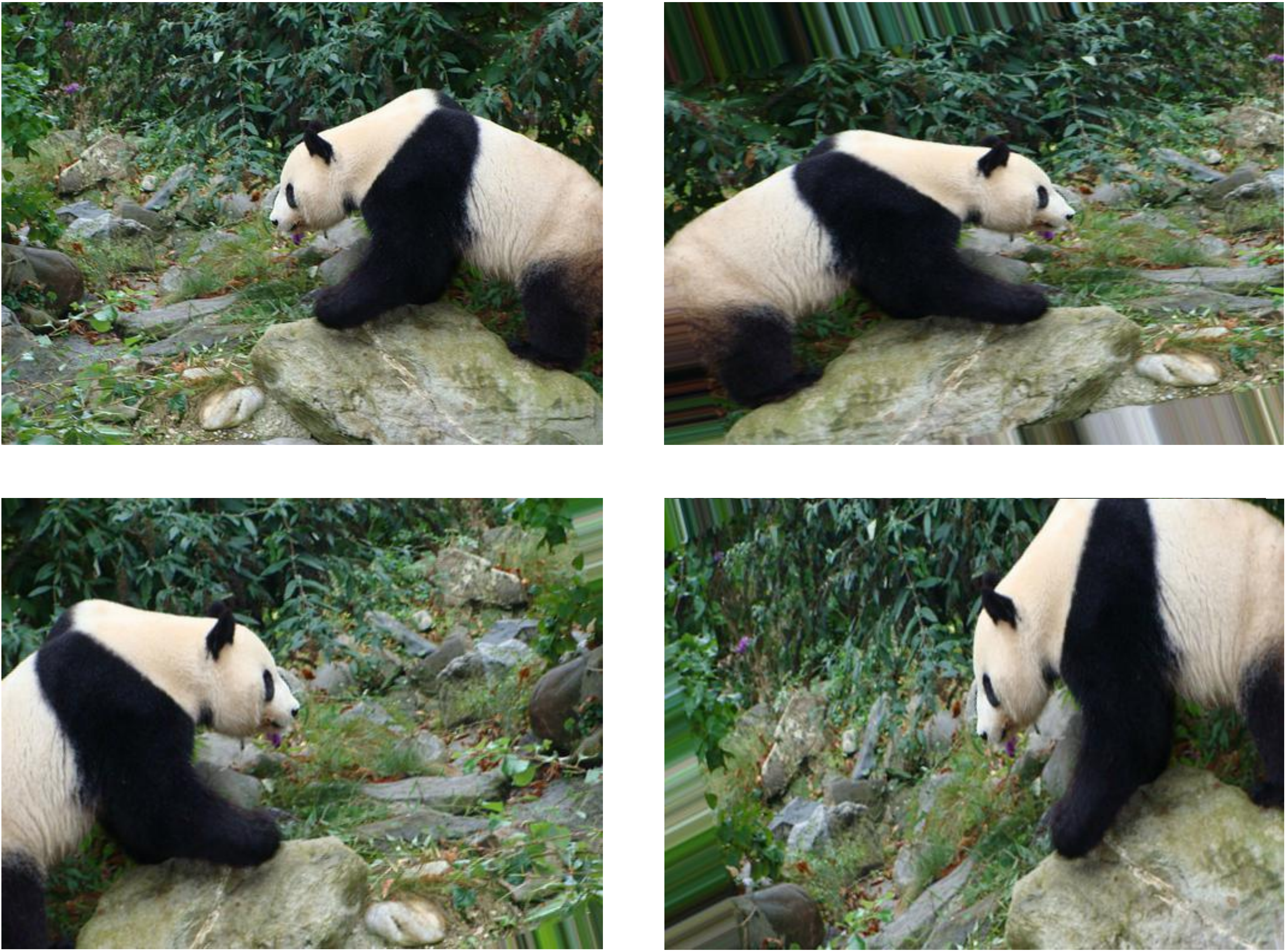
Internally, imageGen is automatically generating a new training sample each time one is requested via the loop.

```
import numpy as np
import tensorflow as tf


# load the input image, convert it to a NumPy array, and then
# reshape it to have an extra dimension
image = tf.keras.preprocessing.image.load_img("./panda.jpg")
image = tf.keras.preprocessing.image.img_to_array(image)
print ((image.shape))

image = np.expand_dims(image, axis=0)
print ((image.shape))



# construct the image generator for data augmentation then
# initialize the total number of images generated thus far
aug = tf.keras.preprocessing.image.ImageDataGenerator(
rotation_range=20, width_shift_range=0.1, shear_range=0.2,
zoom_range=0.4, horizontal_flip=True)

imageGen = aug.flow(image, batch_size=1,
                    save_to_dir='output', save_prefix='panda',
                                          save_format='jpeg')


i = 0
for batch in imageGen:
    i += 1
    if i > 20:
        break
```

```
import numpy as np
import tensorflow as tf


# load the input image, convert it to a NumPy array, and then
# reshape it to have an extra dimension
image = tf.keras.preprocessing.image.load_img("./panda.jpg")
image = tf.keras.preprocessing.image.img_to_array(image)
print ((image.shape))

image = np.expand_dims(image, axis=0)
print ((image.shape))



# construct the image generator for data augmentation then
# initialize the total number of images generated thus far
aug = tf.keras.preprocessing.image.ImageDataGenerator(
rotation_range=20, width_shift_range=0.1, shear_range=0.2,
zoom_range=0.4, horizontal_flip=True)

imageGen = aug.flow(image, batch_size=1,
                    save_to_dir='output', save_prefix='panda',
                                          save_format='jpeg')


i = 0
for batch in imageGen:
    i += 1
    if i > 20:
        break
```

Each time we iterate with the iterator object imageGen is automatically generating an augmented version of the training image(s) in the current batch.

In this case we iterate 20 times. We have a batch size of 1. Therefore, it just creates 20 augmented images.



1. To help illustrate the impact of data augmentation we will look at the cats and dogs dataset in Kaggle.

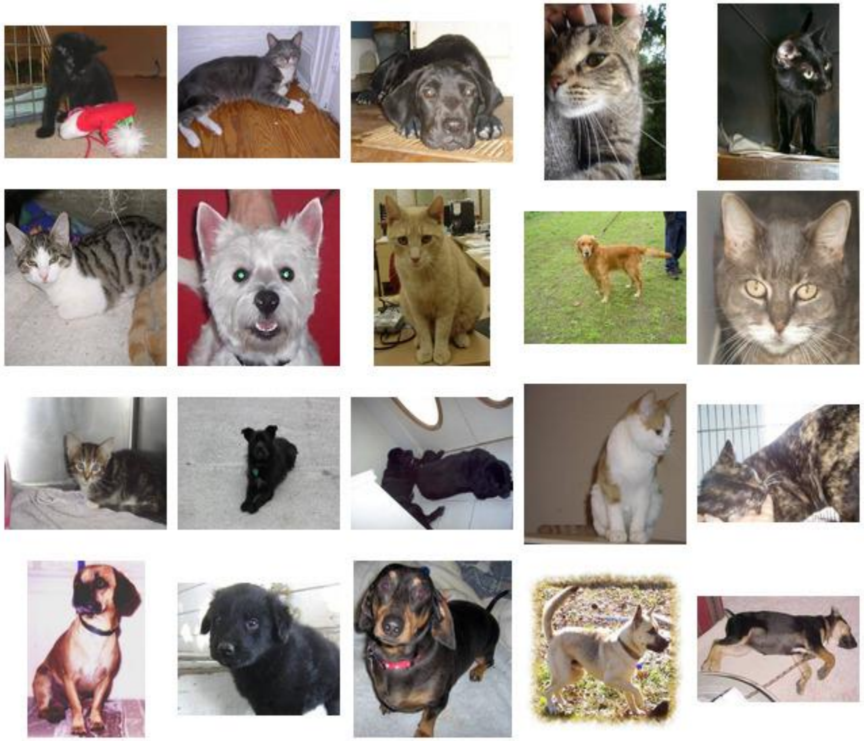
2. The training set contains **25,000 images** of dogs and cats (12500 each).

3. As data augmentation is typically very useful when we have a small amount of data we will use a fraction of the data to illustrate the impact of the data augmentation.

4. We are going to use **2000** images of cats and dogs for training and then we will **validate on 800 images** of each class.

5. On the next slide we first look at the results when we build a CNN **without any augmentation** (aside from scaling).

6. This example is based on Francois Chollet’s book on Deep Learning with Keras.



Over the next few slides we will look at the application of **flow_from_directory** to this problem.

For this example, we are going to use the following network architecture. This class is stored in a package called conv.

This is a standard architecture that consists of 3 consecutive **conv+pool layers** followed by a fully connected dense layer.

Notice because the problem is one of binary classification the final node is a **sigmoid** unit.

```
import tensorflow as tf


class convModel:

    @staticmethod
    def build(width, height, depth):

        inputShape = (width, height, depth)
        model = tf.keras.models.Sequential()
        model.add(tf.keras.layers.Conv2D(32, (3, 3), input_shape=inputShape,
activation='relu'))
        model.add(tf.keras.layers.MaxPooling2D(pool_size=(2, 2)))

        model.add(tf.keras.layers.Conv2D(32, (3, 3), activation='relu'))
        model.add(tf.keras.layers.MaxPooling2D(pool_size=(2, 2)))

        model.add(tf.keras.layers.Conv2D(64, (3, 3), activation='relu'))
        model.add(tf.keras.layers.MaxPooling2D(pool_size=(2, 2)))

        model.add(tf.keras.layers.Flatten())
        model.add(tf.keras.layers.Dense(64, activation='relu'))
        model.add(tf.keras.layers.Dropout(0.5))
        model.add(tf.keras.layers.Dense(1, activation='sigmoid'))

        # return the constructed network architecture
        return model
```

Associated Colab Notebook

This is the main code for our data augmentation example.

The images in the dataset are of different sizes so we want them all resized to **150*150 pixels**.

We call our build function which builds our conv neural network and we pass it the width height and depth of each image.

Next we compile our model as usual.

```
import tensorflow as tf
from conv.convModel import convModel
import matplotlib.pyplot as plt
import numpy as np


# resize all images to thie width and height
width, height= 150, 150

trainDataDir = '/home/local/CIT/ted.scully/Datasets/dogs_cats/data/train'
validationDataDir= '/home/local/CIT/ted.scully/Datasets/dogs_cats/data/validation'

numTrainingSamples = 2000
numValidationSamples = 800

NUM_EPOCHS = 50
batchSize = 64

model = convModel.build(width, height, 3)

model.compile(loss='binary_crossentropy', optimizer='rmsprop',
          metrics=['accuracy'])
```

In the first example we will only apply rescaling as part of the data augmentation.

Associated Colab Notebook

Notice we create a ImageDataGenerator for both the training and validation data. The only purpose it will play for the validation data is to allow us to rescale the validation data.

Remember the flow_from_directory method allows us to **specify a path to a directory** and it will then generate batches of augmented data.

However, notice in this example, that the only modification made by either the train or test generator is that they rescale the data. (In the next example we will add the data augmentation options to the train generator object!)

Associated Colab Notebook

```
# In this example we create an ImageDataGenerator for both training
and set images

# This ImageDataSetGenerator will  normalize the images for us.
trainDataGenerator =
tf.keras.preprocessing.image.ImageDataGenerator(rescale= 1.0/255)

# The ImageDataSetGenerator for testing
valDataGenerator =
tf.keras.preprocessing.image.ImageDataGenerator(rescale=1.0/255)

train_generator = trainDataGenerator.flow_from_directory(
    trainDataDir,
    target_size=(width, height),
    batch_size=batchSize,
    class_mode='binary')

validation_generator = valDataGenerator.flow_from_directory(
    validationDataDir,
    target_size=(width, height),
    batch_size=batchSize,
    class_mode='binary')

H = model.fit(
    train_generator,
    steps_per_epoch= numTrainingSamples // batchSize,
    epochs=NUM_EPOCHS,
    validation_data=validation_generator,
    validation_steps=numValidationSamples // batchSize)
```

To train the model we now call **fit_generator**. We pass it the train_generator (instead of passing training data).

Because the generator will generate augmented data endlessly the Keras model needs to know how many samples to draw from the generator before declaring an epoch over.

This is the role of the **steps_per_epoch** argument: after having drawn steps_per_epoch batches from the generator—that is, after having run for steps_per_epoch gradient descent steps—the fitting process will go to the next epoch.

The **validation_steps** argument tells the process how many batches to draw from the validation generator for evaluation.

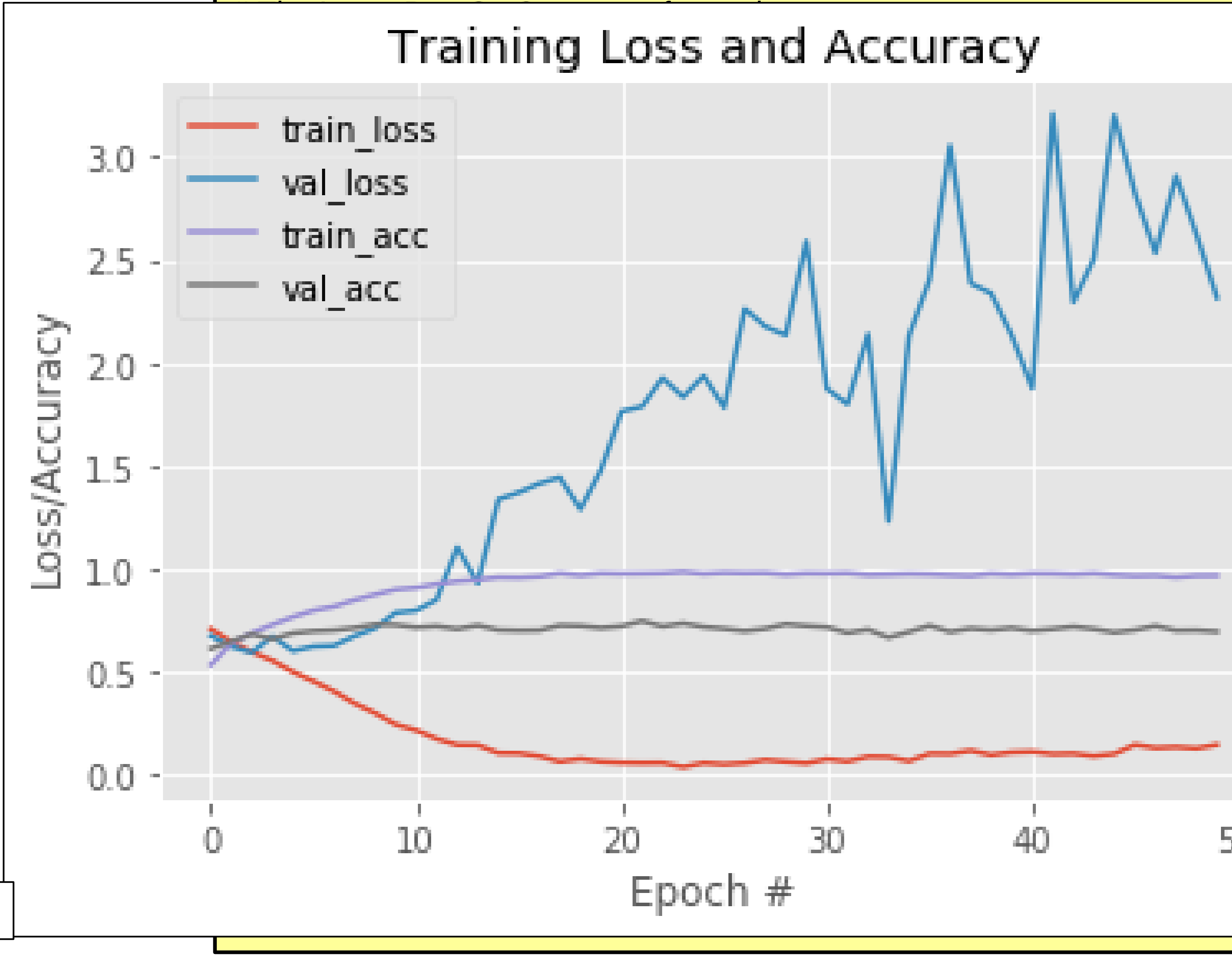
Associated Colab Notebook

```
# In this example we create an ImageDataGenerator for both training
and set images

# This ImageDataSetGenerator will  normalize the images for us.
trainDataGenerator =
tf.keras.preprocessing.image.ImageDataGenerator(rescale= 1.0/255)

# The ImageDataSetGenerator for testing
valDataGenerator =
tf.keras.preprocessing.image.ImageDataGenerator(rescale=1.0/255)

train_generator = trainDataGenerator.flow_from_directory(
    trainDataDir,
    target_size=(width, height),
    batch_size=batchSize,
    class_mode='binary')

validation_generator = valDataGenerator.flow_from_directory(
    validationDataDir,
    target_size=(width, height),
    batch_size=batchSize,
    class_mode='binary')

H = model.fit(
    train_generator,
    steps_per_epoch = numTrainingSamples // batchSize,
    epochs=NUM_EPOCHS,
    validation_data=validation_generator,
    validation_steps=numValidationSamples // batchSize)
```

The validation accuracy of the model is **71**%. Notice the model begins to overfit very early .

It is certainly **overfitting** before we reach 6 or 7 epochs. Training accuracy goes to 100% very quickly

```
# In this example we create an ImageDataGenerator for both training
and set images

# This ImageDataSetGenerator will  normalize the images for us.
trainDataGenerator =
tf.keras.preprocessing.image.ImageDataGenerator(rescale= 1.0/255)
```



Associated Colab Notebook

Now we add some code for performing data augmentation.

Notice we are now generating slightly modified images by applying zoom, shear, rotation and enable horizontal flips.

Also notice that we only add these augmentation options to the training generator so only the training data will ever be changed according to these option.

```
# In this example we create an ImageDataGenerator for both training
and set images

trainDataGenerator =
tf.keras.preprocessing.image.ImageDataGenerator(
    rescale=1.0/255,
    shear_range=0.2,
    zoom_range=0.2,
    rotation_range=30,
    horizontal_flip=True)


testDataGenerator =
tf.keras.preprocessing.image.ImageDataGenerator(rescale=1.0/255)

train_generator = trainDataGenerator.flow_from_directory(
    trainDataDir,
    target_size=(width, height),
    batch_size=batchSize,
    class_mode='binary')

validation_generator = testDataGenerator.flow_from_directory(
    validationDataDir,
    target_size=(width, height),
    batch_size=batchSize,
    class_mode='binary')

H = model.fit(
    train_generator,
    steps_per_epoch= numTrainingSamples // batchSize,
    epochs=NUM_EPOCHS,  validation_data=validation_generator,
    validation_steps=numValidationSamples // batchSize)
```

Associated Colab Notebook

These changes result in obtaining a best model accuracy of **80% accuracy**. Notice we no longer have the same level of **dramatic overfitting**.

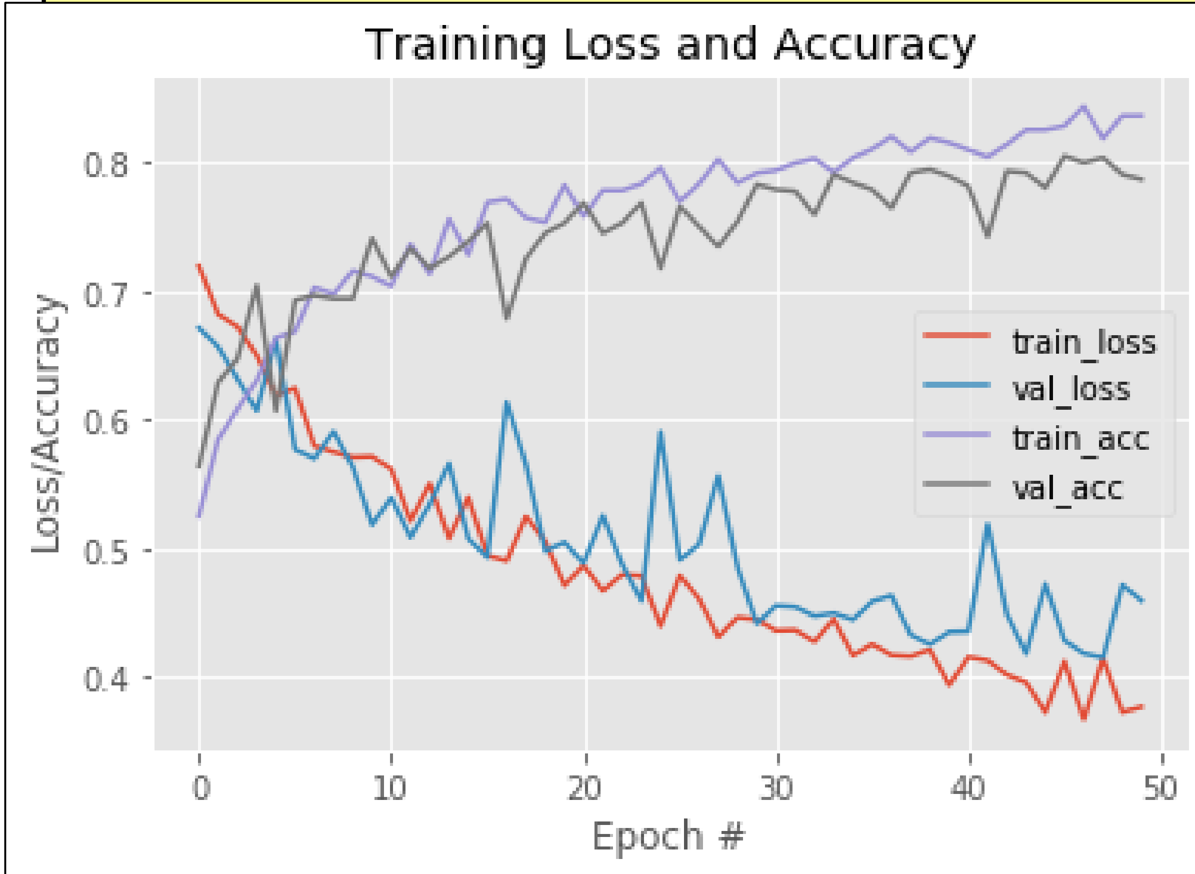
The training accuracy no longer hits 100% percent

The val_loss and train_loss are also much closer. There is no significant deviation between the two.

Please note that you may not always obtain results like this when using data augmentation. Typically this technique is most effective when you have a small amount of image training data to work with.

Associated Colab Notebook

```
# In this example we create an ImageDataGenerator for both training
and set images

trainDataGenerator =
tf.keras.preprocessing.image.ImageDataGenerator(
    rescale=1.0/255,
    shear_range=0.2,
    zoom_range=0.2,
    rotation_range=30,
    horizontal_flip=True)


testDataGenerator =
```


Training Loss and Accuracy
Loss/Accuracy
Epoch #
train_loss
val_loss
train_acc
val_acc