Classification & CNN with keras

Gloria Beraldo (gloria.beraldo@unipd.it)
Department of Information Engineering, University of Padova

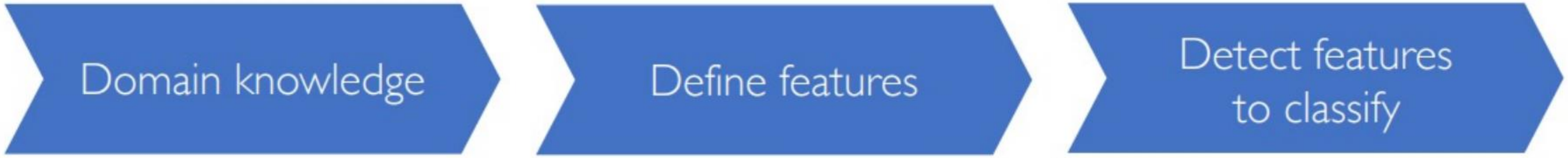
Topics:

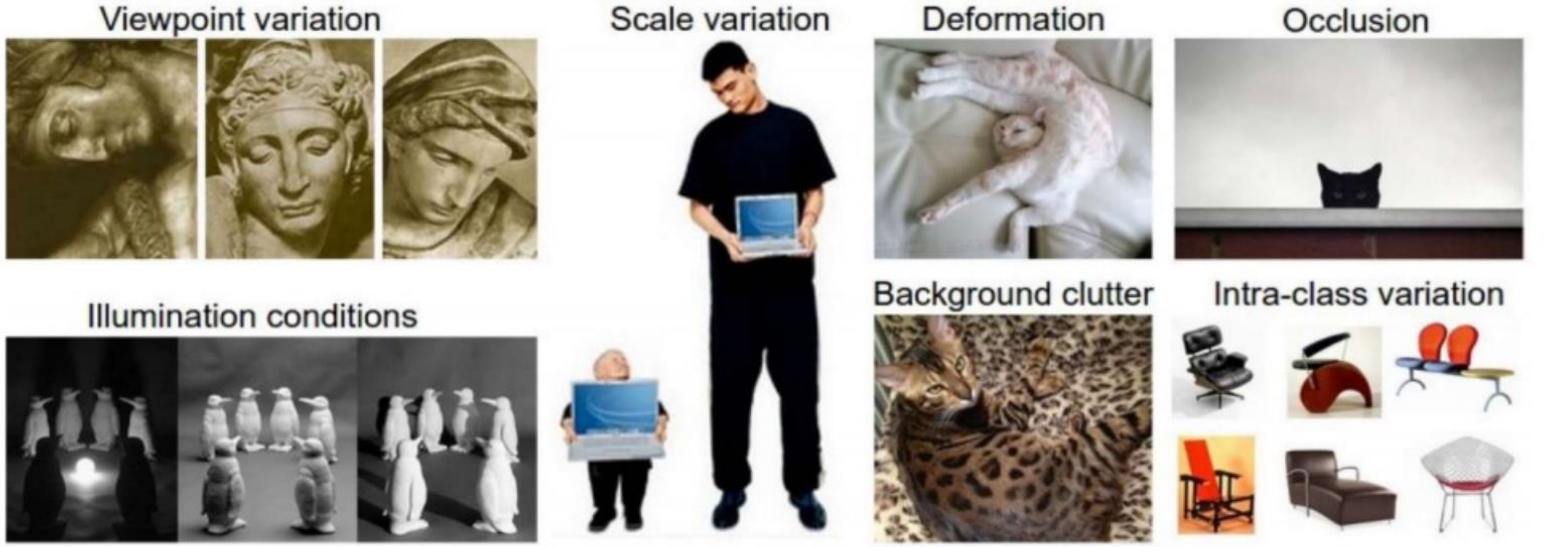
- Introduction
- Convolution Neural Network: Recap
- Keras for deep learning in Python
- Keras-Sequential API
- An example of CNN with keras
- Dog-Cat dataset
- Split the dataset and Check the data
- Using image data augmentation
- Standardizing the data
- Build the model
- Compile & train the model
- Evaluate the model
- Test on new samples
- Save the model

DIPARTIMENTO
DI INGEGNERIA
DELL'INFORMAZIONE

Introduction

Defining the good features for training a machine learning algorithm can be hard in Computer Vision





"Coming up with features is difficult, time-consuming, requires expert knowledge. 'Applied machine learning' is basically feature engineering."

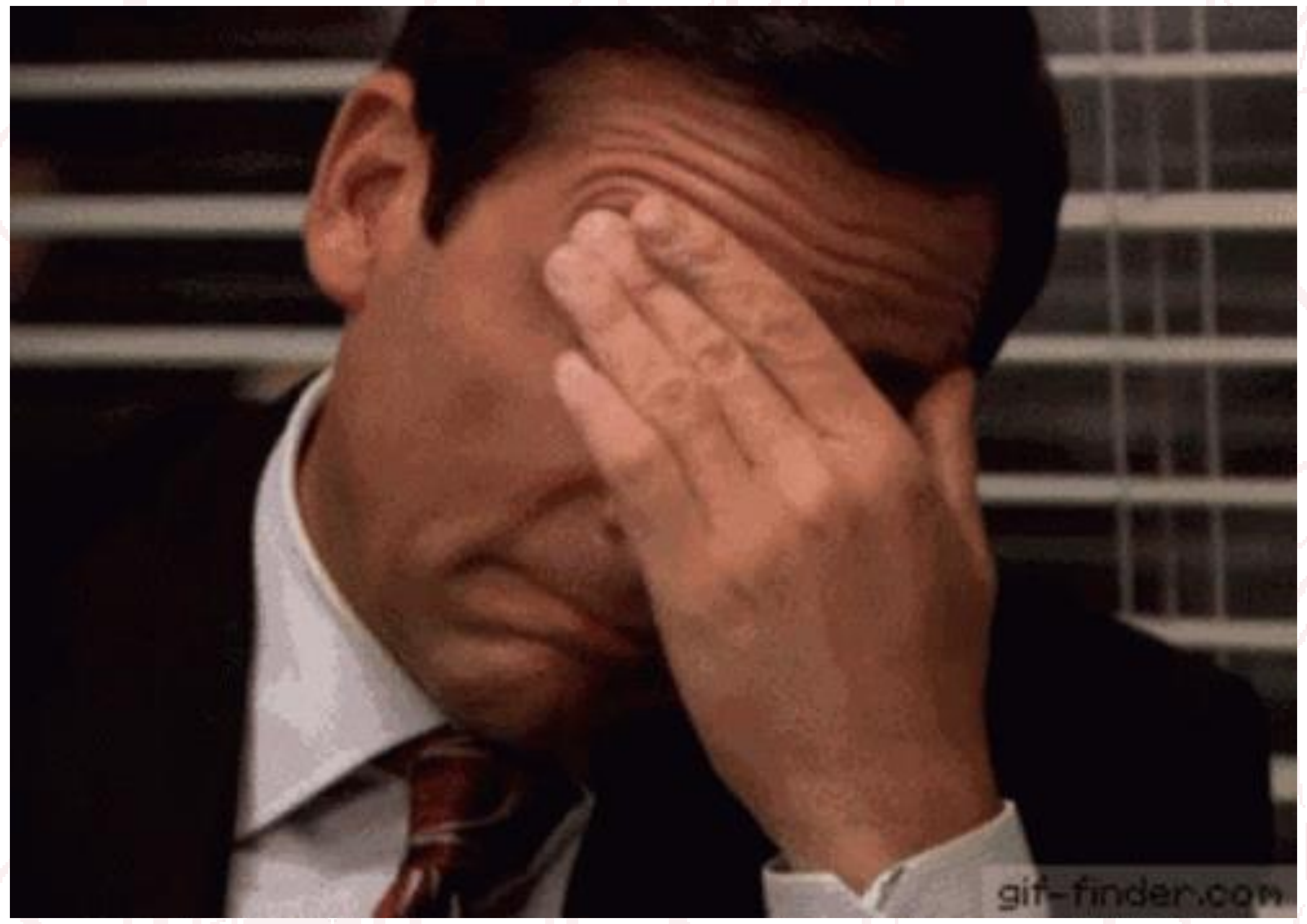
– Andrew Ng

-MIT Introduction to Deep Learning http://introtodeeplearning.com

Introduction

When you manually define the features (i.e., **handcrafted feature engineering**), there are some factors to consider:

- Tedious
- Possible interaction/relationship with other variables
- Limitation due to the human time constraint and imagination
- Introduction of bias due to the specific data domain and previous analyses

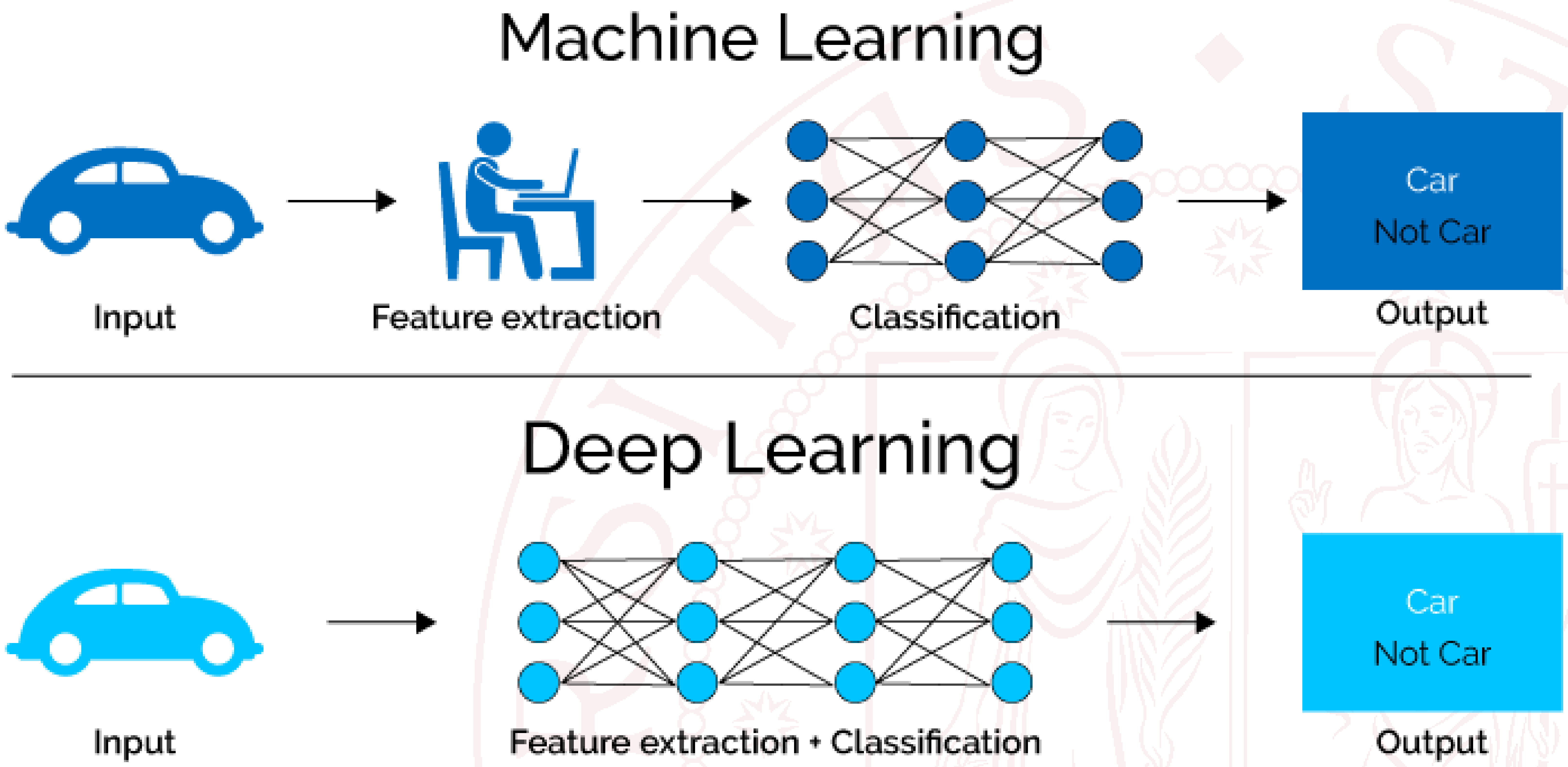


- MIT Introduction to Deep Learning http://introtodeeplearning.com

Introduction

The power of **automated feature engineering consists via deep learning** consists of:

- The number of features can be practically infinite
- No human bias
- It is possible to capture complex non-linear interactions among features
- We can apply dimension reduction/feature selection technique at any time to get rid of redundant/zero important features



- MIT Introduction to Deep Learning http://introtodeeplearning.com

Convolution Neural Network: Recap

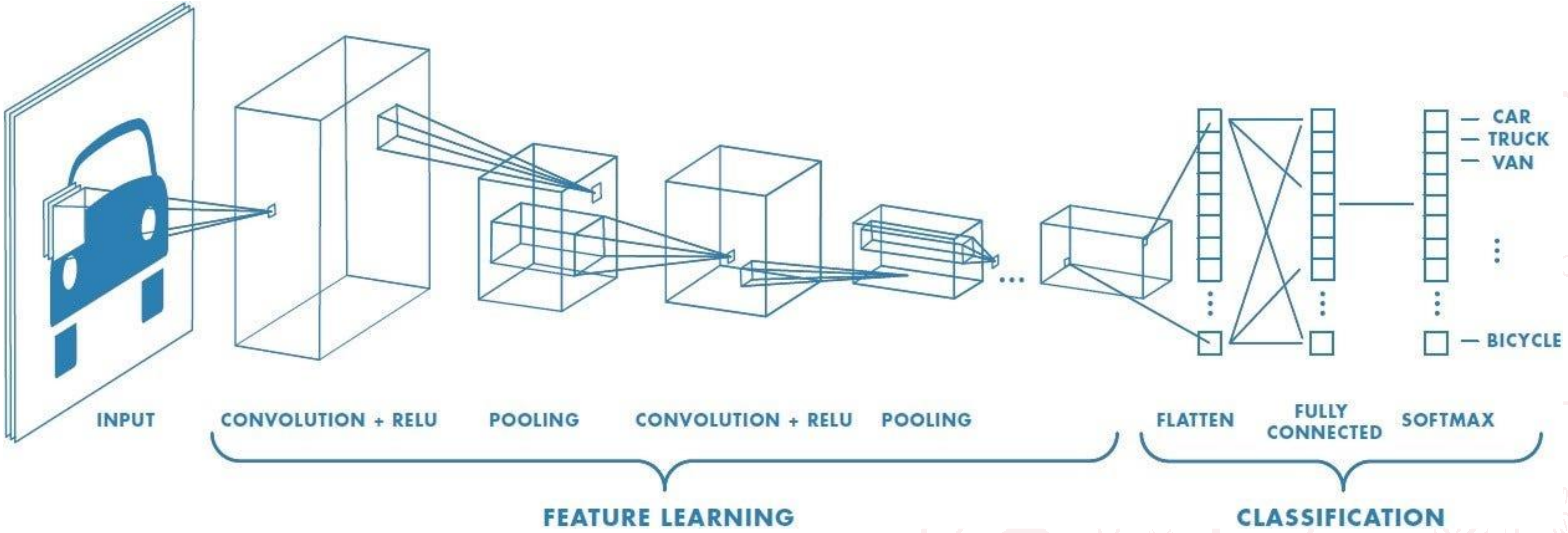
Convolutional Neural Networks (CNNs)

- **Convolutional neural network** (**CNN**) is one that contains spatially local connections
- **Kernel**: a pattern of weights that is replicated across multiple local regions
- **Convolution**: the process of applying the kernel to the pixels of the image
  - input vector $\mathbf{x}$ of size n, corresponding to n pixels in a one-dimensional image, and vector kernel $\mathbf{k}$ of size l
  - convolution operation, $\mathbf{z} = \mathbf{x} * \mathbf{k}$

$$z_i = \sum_{j=1}^{l} k_j x_{j+i-(l+1)/2}$$

  - kernels centers are separated distance called **stride**, s
  - convolution stops at the edges of the image, but **padding** the input with extra pixels is possible so that kernel is applied exactly $\lceil n/s \rceil$ times

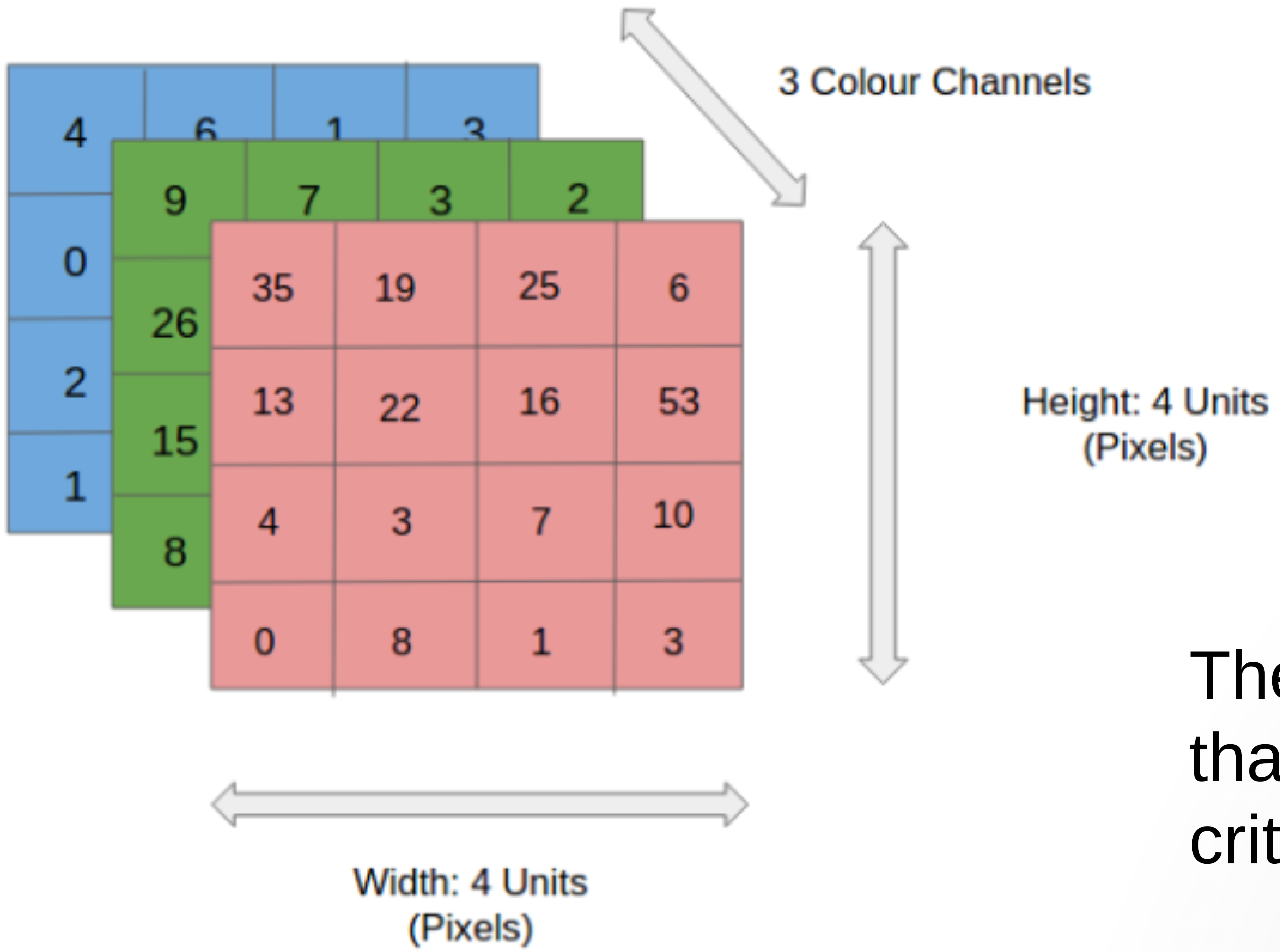
Convolution Neural Network: Recap



Thanks to the hidden layers, CNN **identifies patterns**.

When, we apply CNN, the **number of filters** and **their dimensions** should be specified.

Convolution Neural Network: Recap



Typically, the input for CNN are images.

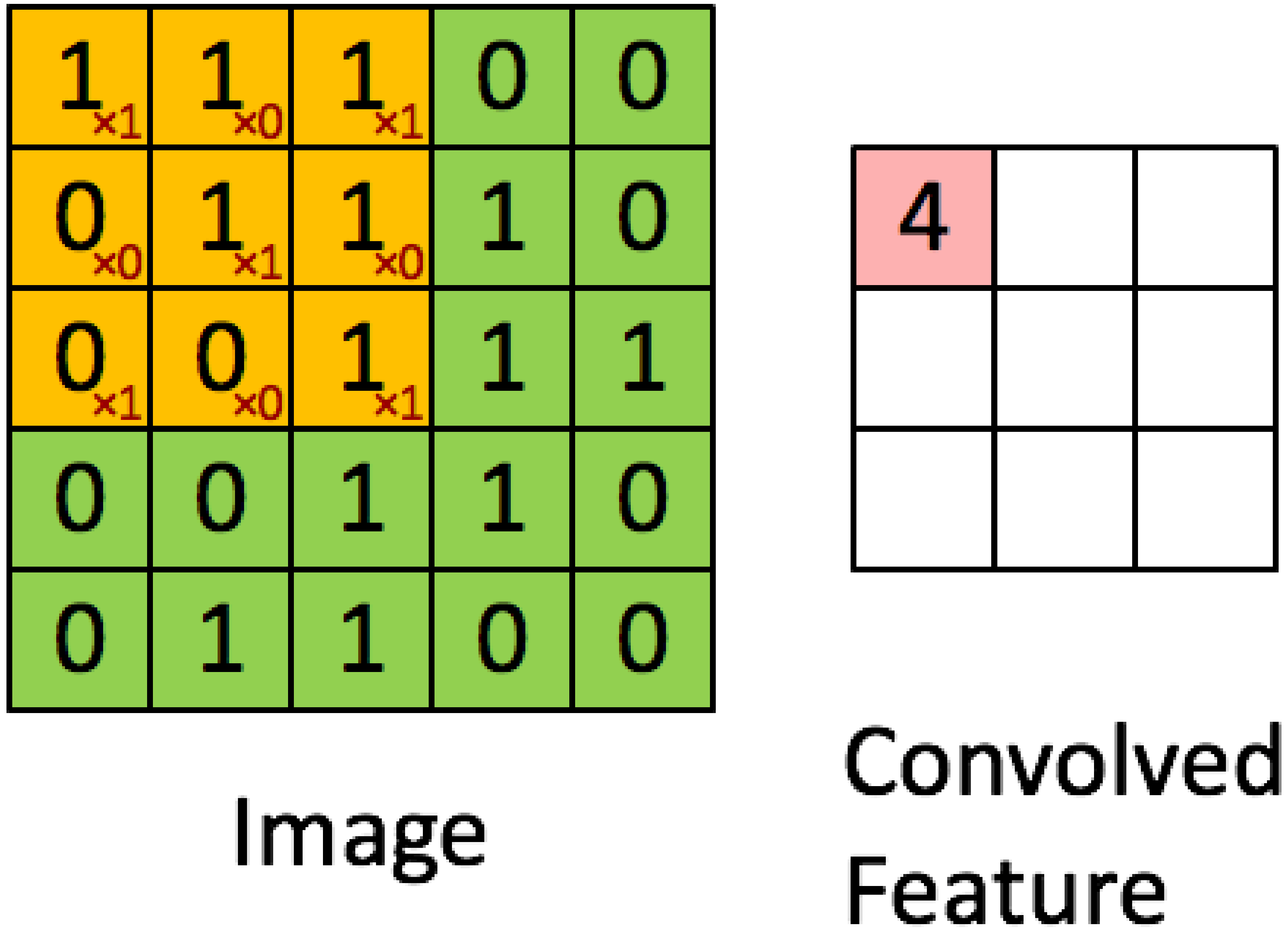
An image is characterized by the:

- Width
- Height
- Channels

The role of a CNN is to transform the images into a form that is easier to process, without losing features that are critical for getting a good prediction.

Conventionally, the first ConvLayer is responsible for capturing the Low-Level features such as edges, color, gradient orientation, etc. With added layers, the architecture adapts to the High-Level features as well, giving us a network that has a wholesome understanding of images in the dataset, similar to how we would.

Convolution Layer – The kernel



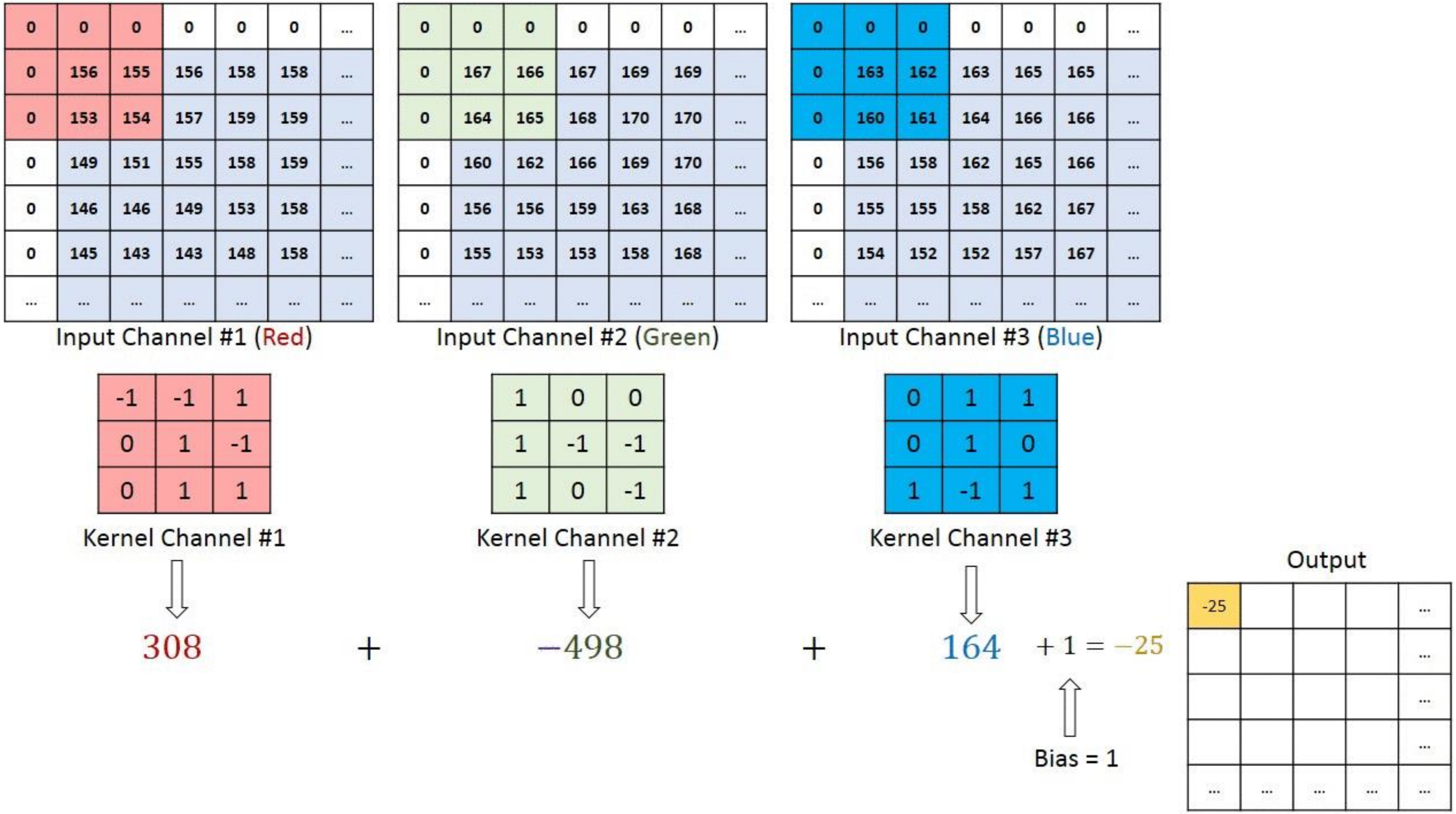
The green section resembles our 5x5x1 input image

The element involved in the convolution operation (Kernel filter) is represented in yellow.

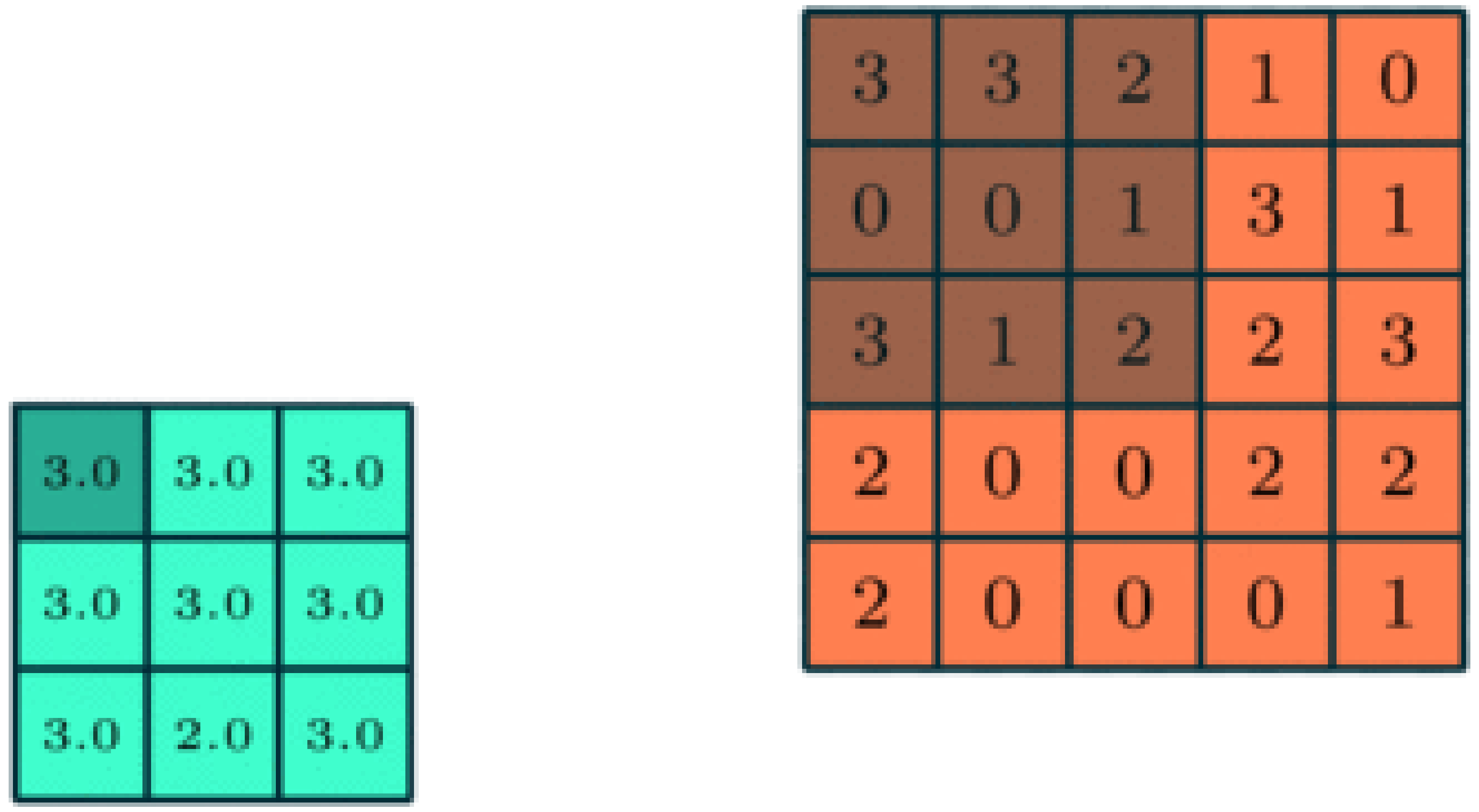
The filter moves to the right with a certain Stride Value till it parses the complete width. Moving on, it hops down to the beginning (left) of the image with the same Stride Value and repeats the process until the entire image is traversed.

The Kernel shifts 9 times because of Stride Length = 1, every time performing an elementwise multiplication operation (Hadamard Product) between K and the portion P of the image over which the kernel is hovering.

Convolution Layer – The kernel



Pooling Layer



Similar to the Convolutional Layer, the Pooling layer is responsible for reducing the spatial size of the Convolved Feature.

This is to decrease the computational power required to process the data through dimensionality reduction.

Furthermore, it is useful for extracting dominant features which are rotational and positional invariant, thus maintaining the process of effectively training the model.

There are two types of Pooling:

- **Max Pooling** returns the **maximum value** from the portion of the image covered by the Kernel;
- **Average Pooling** returns the **average** of all the values from the portion of the image covered by the Kernel

Convolutional Neural Network

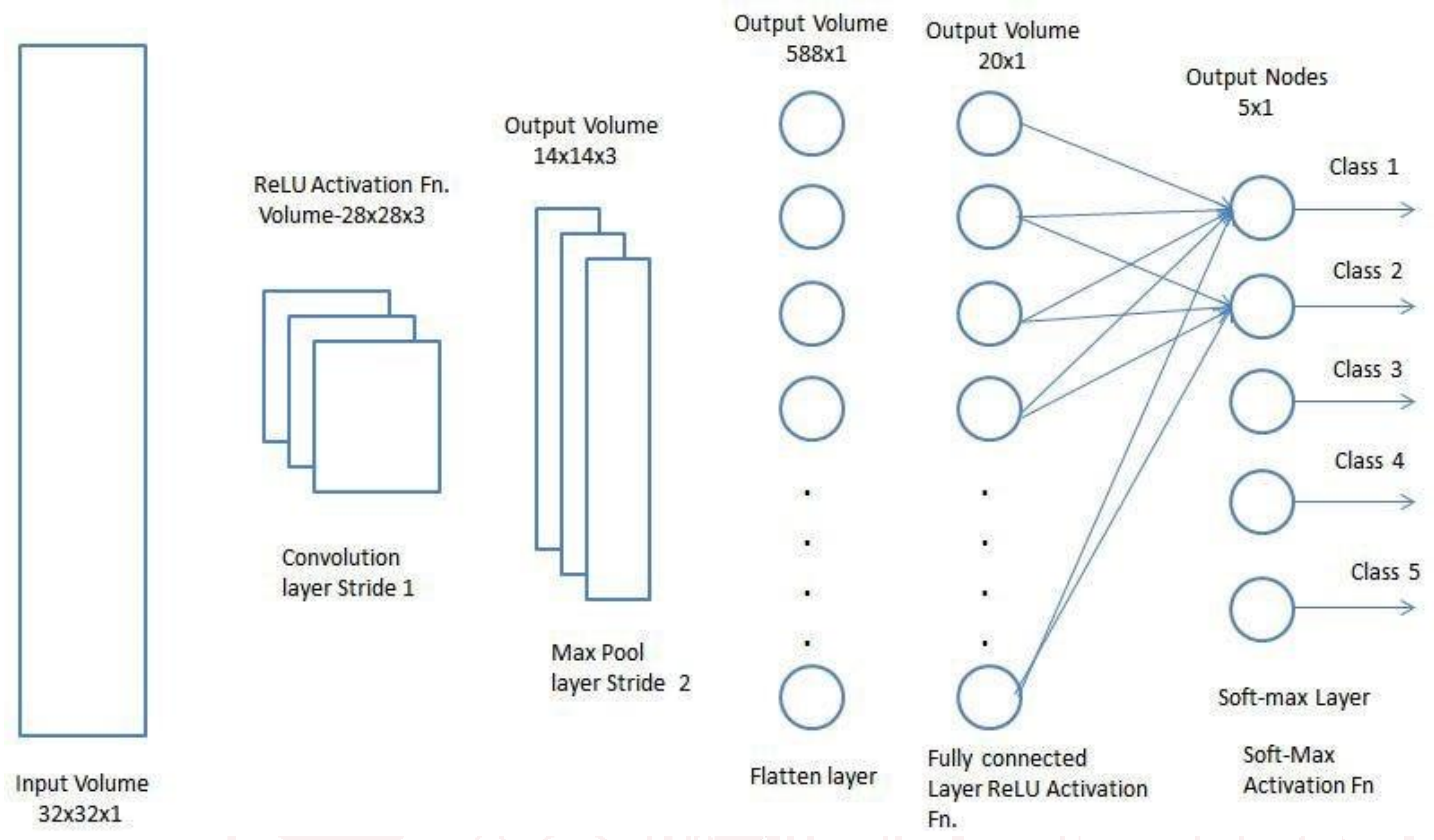
The Convolutional Layer and the Pooling Layer, together form the i-th layer of a Convolutional Neural Network.

Depending on the complexities in the images, the number of such layers may be increased for capturing low-level details even further, but at the cost of more computational power.

After this process, we have successfully enabled the model to understand the features.

Moving on, we are going to flatten the final output (i.e., the image is transformed into a column vector) and feed it to a regular Neural Network for classification purposes.

Over a series of epochs, the model is able to distinguish between dominating and certain low-level features in images and classify them using the Softmax Classification technique.



Keras for deep learning in Python

It is an approachable, highly-productive interface for solving machine learning problems, with a focus on modern deep learning.

It provides essential abstractions and building blocks for developing and shipping machine learning solutions with high iteration velocity.

Keras is a deep learning API written in Python, running on top of the machine learning platform TensorFlow.

Keras is:

- **Simple** -- Keras reduces developer cognitive load to free you to focus on the parts of the problem that really matter.
- **Flexible** -- Simple workflows should be quick and easy.
- **Powerful** -- Keras provides industry-strength performance and scalability: it is used by organizations and companies including NASA, YouTube, or Waymo.

```
import tensorflow as tf
from tensorflow import keras
from tensorflow.keras import layers
```

https://keras.io/about/

Keras-Sequential API

In this lecture, we will use Tensorflow and in particular the Keras Sequential API, that allows to easily create a model from few lines of code.

Indeed, since this lecture is focused on Classification, we use the Sequential model, that is appropriate for a plain stack of layers where **each layer has exactly one input tensor and one output tensor**.

For more complex architectures, you should use the Keras functional API, which allows to build arbitrary graphs of layers, or write models entirely from scratch via subclasssing.

```
from tensorflow.keras.models import Sequential

model = Sequential()
```

https://keras.io/about/

An example of CNN with keras: Dataset

In this lecture, we classify dogs vs. cats using a standard computer vision dataset that involves classifying photos as either containing a dog or cat.

The dataset was developed as a partnership between Petfinder.com and Microsoft.

NB: Some samples are corrupted, so we need to filter them.

```
!curl -O https://download.microsoft.com/download/3/E/1/3E1C3F21-ECDB-4869-8368-6DEBA77B919F/kagglecatsanddogs_5340.zip
!unzip -q kagglecatsanddogs_5340.zip
!ls

!ls PetImages
```

```
  % Total    % Received % Xferd  Average Speed   Time    Time     Time  Current
                                 Dload  Upload   Total   Spent    Left  Speed
100  786M  100  786M    0     0  69.1M      0  0:00:11  0:00:11 --:--:-- 82.8M
 CDLA-Permissive-2.0.pdf      PetImages         sample_data
 kagglecatsanddogs_5340.zip  'readme[1].txt'
Cat  Dog
```

Split the dataset

We use the **image_dataset_from_directory** utility in Keras to generate the training and validation sets from image files in a directory.

```
image_size = (180, 180)
batch_size = 128


train_ds, val_ds = tf.keras.utils.image_dataset_from_directory(
    "PetImages",
    validation_split=0.2,
    subset="both",
    seed=1337,
    image_size=image_size,
    batch_size=batch_size,
)

print("Training set: %d images" % len(train_ds))
print("Validation set: %d images" % len(val_ds))
```

```
Found 23410 files belonging to 2 classes.
Using 18728 files for training.
Using 4682 files for validation.
Training set: 147 images
Validation set: 37 images
```

Batch_size= the number of samples that are processed simultaneously

Args	
`directory`	Directory where the data is located. If `labels` is "inferred", it should contain subdirectories, each containing images for a class. Otherwise, the directory structure is ignored.
`labels`	Either "inferred" (labels are generated from the directory structure), None (no labels), or a list/tuple of integer labels of the same size as the number of image files found in the directory. Labels should be sorted according to the alphanumeric order of the image file paths (obtained via `os.walk(directory)` in Python).
`label_mode`	String describing the encoding of `labels`. Options are: • 'int': means that the labels are encoded as integers (e.g. for `sparse_categorical_crossentropy` loss). • 'categorical' means that the labels are encoded as a categorical vector (e.g. for `categorical_crossentropy` loss). • 'binary' means that the labels (there can be only 2) are encoded as `float32` scalars with values 0 or 1 (e.g. for `binary_crossentropy`). • None (no labels).
`class_names`	Only valid if "labels" is "inferred". This is the explicit list of class names (must match names of subdirectories). Used to control the order of the classes (otherwise alphanumerical order is used).
`color_mode`	One of "grayscale", "rgb", "rgba". Default: "rgb". Whether the images will be converted to have 1, 3, or 4 channels.
`batch_size`	Size of the batches of data. Default: 32. If `None`, the data will not be batched (the dataset will yield individual samples).
`image_size`	Size to resize images to after they are read from disk, specified as `(height, width)`. Defaults to `(256, 256)`. Since the pipeline processes batches of images that must all have the same size, this must be provided.
`shuffle`	Whether to shuffle the data. Default: True. If set to False, sorts the data in alphanumeric order.
`seed`	Optional random seed for shuffling and transformations.
`validation_split`	Optional float between 0 and 1, fraction of data to reserve for validation.
`subset`	Subset of the data to return. One of "training", "validation" or "both". Only used if `validation_split` is set. When `subset="both"`, the utility returns a tuple of two datasets (the training and validation datasets respectively).
`interpolation`	String, the interpolation method used when resizing images. Defaults to `bilinear`. Supports `bilinear`, `nearest`, `bicubic`, `area`, `lanczos3`, `lanczos5`, `gaussian`, `mitchellcubic`.

https://www.tensorflow.org/api_docs/python/tf/keras/utils/image_dataset_from_directory

Check the data

For instance, we can verify the data inside the training set, we have just achieved:

```python
import numpy as np
train_ds_iterator = train_ds.as_numpy_iterator() # it is an iterator on the training dataset
batch = train_ds_iterator.next()
print(batch[0].shape) #the number of images from batch, it should correspond to the ones specified in the keras.utilis.image_dataset_from_directory
print(batch[1]) # this corresponds to the label
```

```
(128, 180, 180, 3)
[0 0 1 0 0 0 1 1 1 0 1 0 0 0 0 1 0 0 1 0 0 0 0 0 0 1 0 1 0 0 1 1 1 1 1 0 1
 1 0 0 0 0 1 0 1 1 0 0 1 0 1 1 0 1 1 1 1 0 1 1 1 0 0 1 1 1 0 0 1 1 0 1 1 0
 1 0 1 1 1 1 0 1 0 0 1 0 1 0 1 0 0 0 1 0 0 0 1 0 1 0 0 1 0 0 0 1 0 1 1 0 0
 1 1 1 1 1 1 1 1 0 1 1 1 0 0 0 0 0]
```

128 images in the batch

180x180 corresponds to the image size

3 number of channels

Verify them comparing with the data in the image_dataset_from_directory

Check the data

For instance, we can verify the data inside the training set, we have just achieved:

```
import numpy as np
train_ds_iterator = train_ds.as_numpy_iterator() # it is an iterator on the training dataset
batch = train_ds_iterator.next()
print(batch[0].shape) #the number of images from batch, it should correspond to the ones specified in the keras.utilis.image_dataset_from_directory
print(batch[1]) # this corresponds to the label
```

```
(128, 180, 180, 3)
0 0 1 0 0 0 1 1 1 0 1 0 0 0 0 1 0 0 1 0 0 0 0 0 0 1 0 1 0 0 1 1 1 1 1 0 1
1 0 0 0 0 1 0 1 1 0 0 1 0 1 1 0 1 1 1 1 0 1 1 1 0 0 1 1 1 0 0 1 1 0 1 1 0
1 0 1 1 1 1 0 1 0 0 1 0 1 0 1 0 0 0 1 0 0 0 1 0 1 0 0 1 0 0 0 1 0 1 1 0 0
1 1 1 1 1 1 1 1 0 1 1 1 0 0 0 0 0]
```

Label: 0 CAT

1 DOG

Let’s visualize some samples

Visualize the data

To visualize the data on the dataset, we can use matplotlib. For instance, we plot the 9 images in the training dataset and the corresponding label. As you can see, label 1 is associated with "dog" and label 0 with "cat".

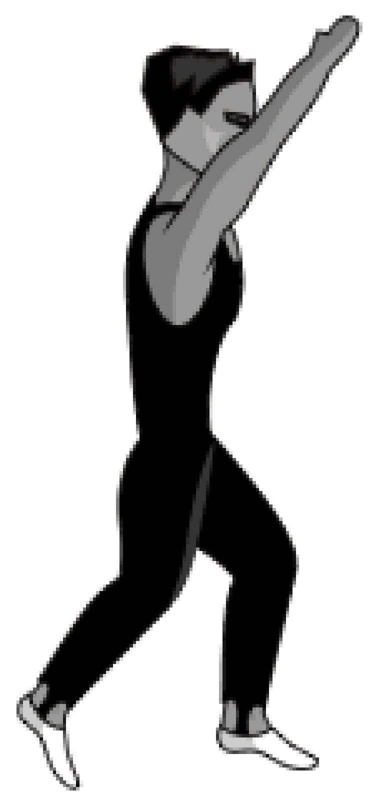
```
import matplotlib.pyplot as plt

plt.figure(figsize=(10, 10))
for images, labels in train_ds.take(1):
    for i in range(9):
        ax = plt.subplot(3, 3, i + 1)
        plt.imshow(images[i].numpy().astype("uint8"))
        plt.title(int(labels[i]))
        plt.axis("off")
```

Using image data augmentation

It's a good practice to artificially introduce sample diversity by applying random yet realistic transformations to the training images, such as random horizontal flipping or small random rotations for instance. This helps expose the model to different aspects of the training data while slowing down overfitting.

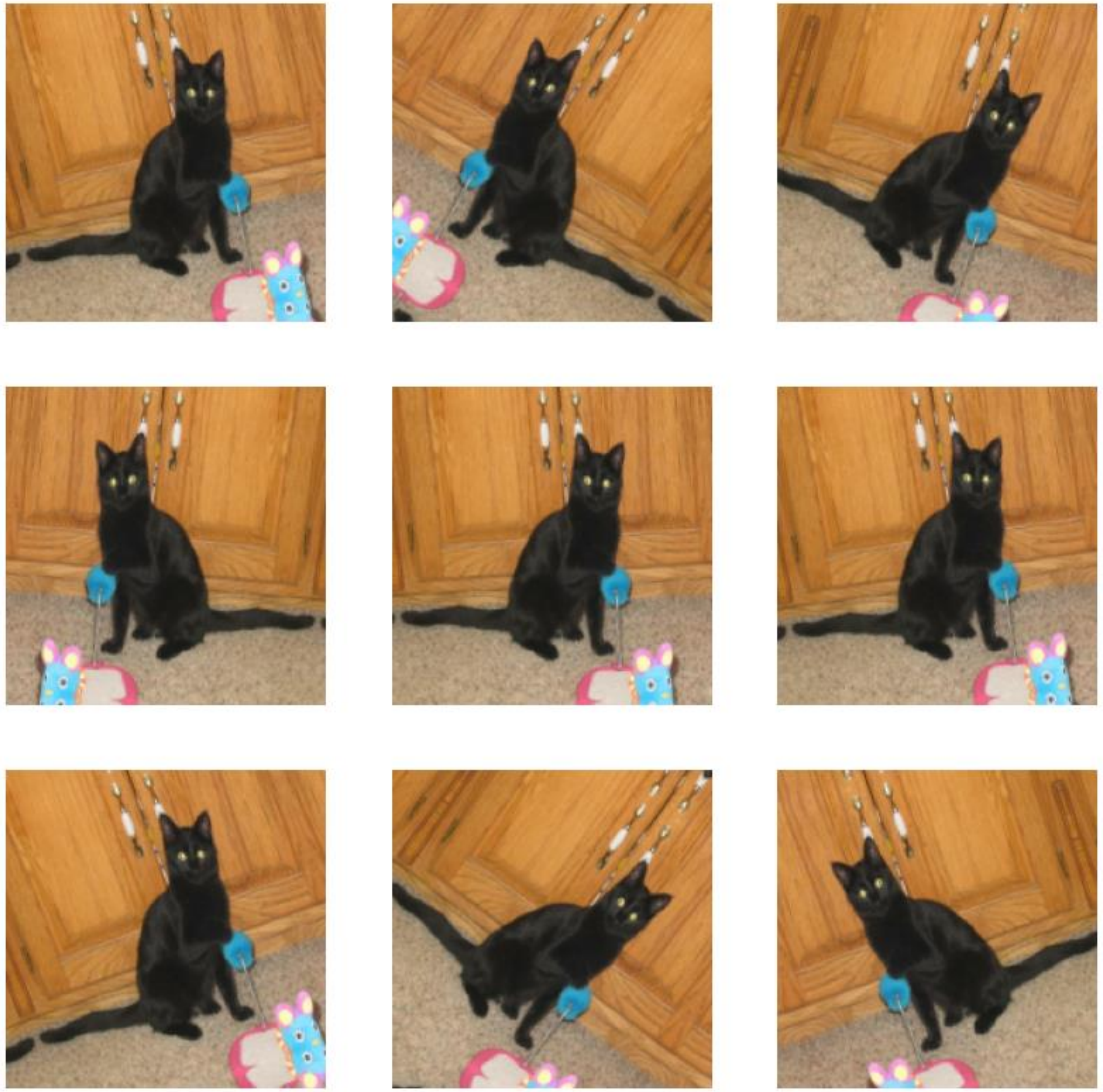
```
data_augmentation = keras.Sequential(
    [
        layers.RandomFlip("horizontal"),
        layers.RandomRotation(0.1),
    ]
)
```



Using image data augmentation

Let's visualize what the augmented samples look like, by applying **data_augmentation** repeatedly (defined before) to the first image in the dataset:

```
plt.figure(figsize=(10, 10))
for images, _ in train_ds.take(1):
    for i in range(9):
        augmented_images = data_augmentation(images)
        ax = plt.subplot(3, 3, i + 1)
        plt.imshow(augmented_images[0].numpy().astype("uint8"))
        plt.axis("off")
```



Standardizing the data

The images that we are using are already in a standard size (180x180), as they are being yielded as contiguous float32 batches by our dataset.

However, their RGB channel values are in the [0, 255] range.

This is not ideal for a neural network; in general you should seek to make your input values small. Here, we will standardize values to be in the [0, 1] by using a Rescaling layer at the start of our model.

```
train_ds = train_ds.map(lambda x,y: (x/255, y))
val_ds = val_ds.map(lambda x,y: (x/255, y))
print(train_ds.as_numpy_iterator().next()[0].min()) # we check for instance the min
print(train_ds.as_numpy_iterator().next()[0].max()) # we check for instance the max
```

Configure the dataset for performance

Let's apply data augmentation and standardization to our training dataset, and let's make sure to use buffered prefetching so we can yield data from disk without having I/O becoming blocking:

```
# Apply `data_augmentation` and standardization to the training images.
train_ds = train_ds.map(     # apply a sort of transformation, it is quick
    lambda img, label: (data_augmentation(img), label),
    num_parallel_calls=tf.data.AUTOTUNE,
)
print(train_ds.as_numpy_iterator().next()[0].max())
# Prefetching samples in GPU memory helps maximize GPU utilization.
train_ds = train_ds.prefetch(tf.data.AUTOTUNE)
val_ds = val_ds.prefetch(tf.data.AUTOTUNE)
```

```
1.0
```

Build the model

Create the convolutional base The lines of code below define the convolutional base using a common pattern: a stack of Conv2D and MaxPooling2D layers to find the features

As input, a CNN takes tensors of shape (image_height, image_width, color_channels), ignoring the batch size. color_channels refers to (R,G,B). In this example, you will configure your CNN to process inputs of shape (180, 180, 1). You can do this by passing the argument input_shape to your first layer.

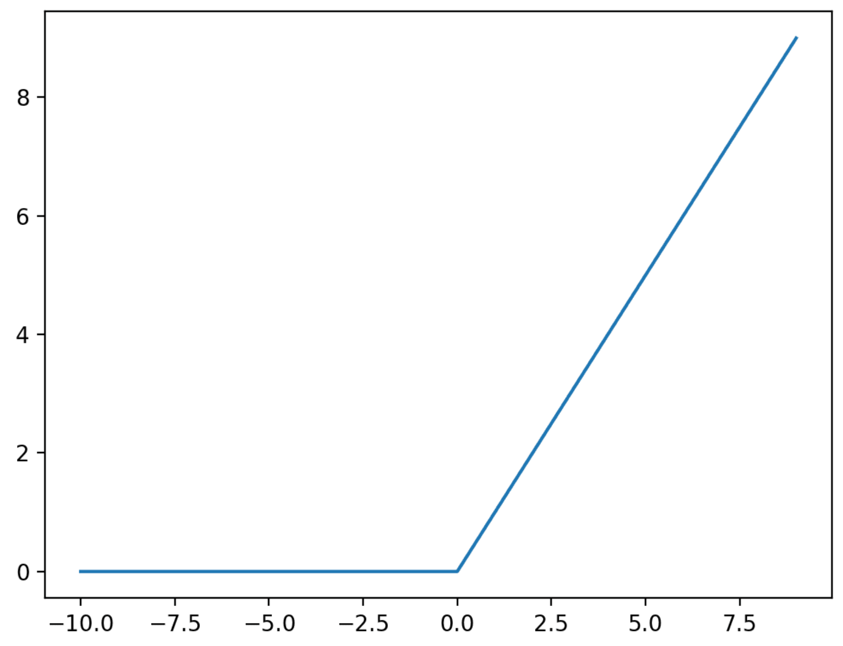
```
from tensorflow.keras import models
model = models.Sequential() # Sequential you have one inpunt and one output as in this case

# There are some hyperparameters in the architecture, according to these, the performance of the architecture can change

# We add then the layer sequentially
model.add(layers.Conv2D(16, (3, 3), 1, activation='relu', input_shape=(180, 180, 3)))

# The first parameter in Conv2D is the number of filters
# The dimention 3x3 in this case
# 1 pixel as stride
# the output pass through an activation function (relu in this case)
# input shape
model.add(layers.MaxPooling2D((2, 2))) ## go the max in the region and condense this information down, the region size is 2x2
model.add(layers.MaxPooling2D((2, 2)))
model.add(layers.Conv2D(16, (3, 3), activation='relu'))
model.add(layers.MaxPooling2D((2, 2)))
```

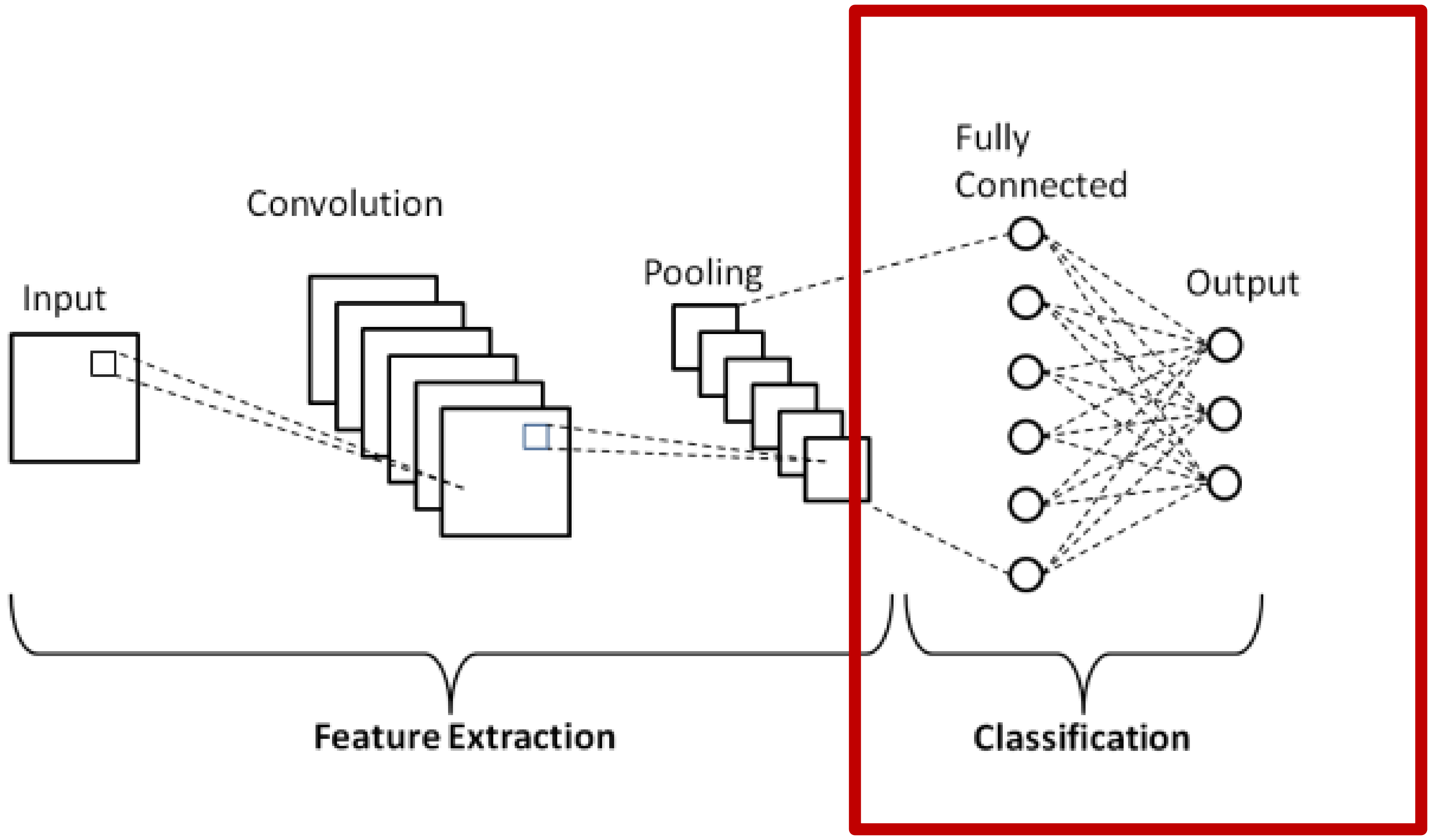
RELU ACTIVATION FUNCTION

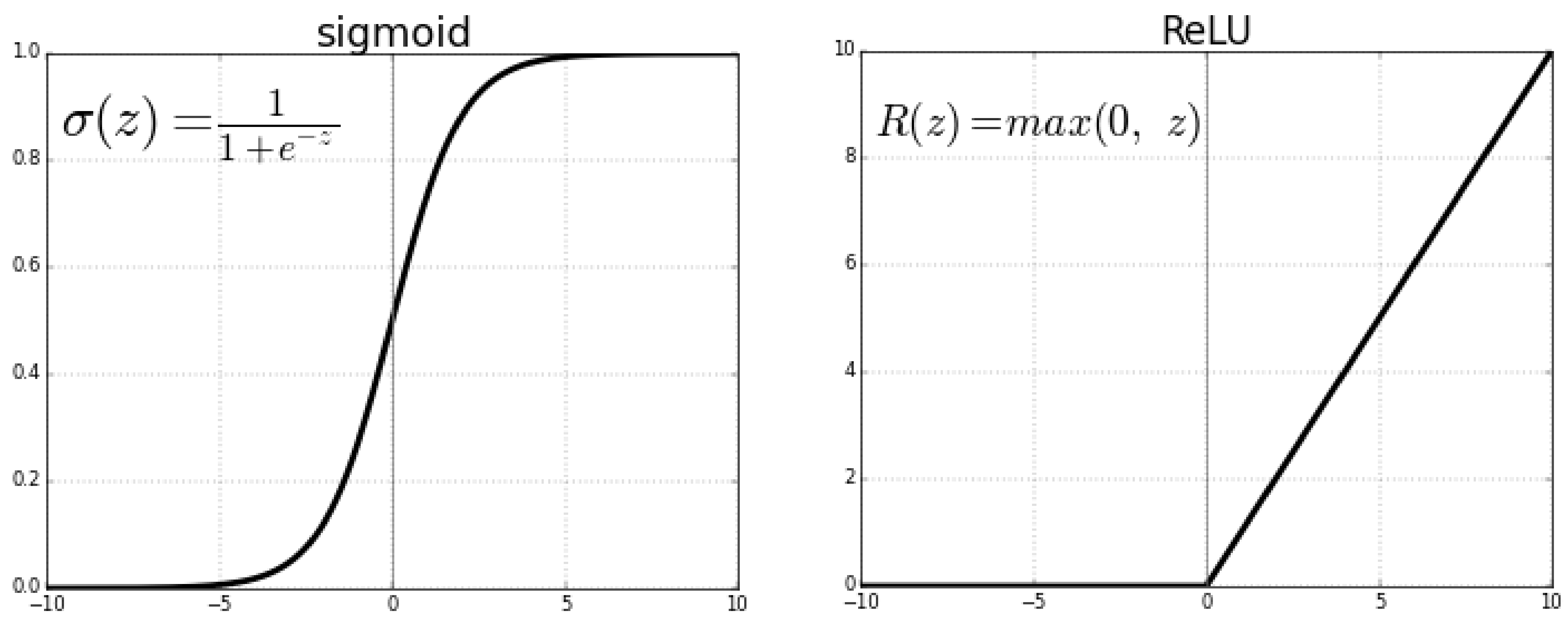


Build the model

Add Dense layers on top to complete the model, you will feed the last output tensor from the convolutional base into one or more Dense layers to perform classification. Dense layers take vectors as input (which are 1D), while the current output is a 3D tensor. First, you will flatten (or unroll) the 3D output to 1D, then add one or more Dense layers on top.

```
model.add(layers.Flatten()) #from 3D to 1D
model.add(layers.Dense(64, activation='relu')) # first parameter is the number of neurons
model.add(layers.Dense(1, activation='sigmoid')) # reduce the output into one --> just one output, if 0 corresponds to class 0, if 1 to class 1
```





Build the model

Let's display the architecture of your model so far:

```
model.summary()
keras.utils.plot_model(model, show_shapes=True)
```

```
Model: "sequential_1"
_________________________________________________________________
 Layer (type)                Output Shape              Param #
=================================================================
 conv2d (Conv2D)             (None, 178, 178, 16)      448

 max_pooling2d (MaxPooling2D  (None, 89, 89, 16)       0
 )

 max_pooling2d_1 (MaxPooling  (None, 44, 44, 16)       0
 2D)

 conv2d_1 (Conv2D)           (None, 42, 42, 16)        2320

 max_pooling2d_2 (MaxPooling  (None, 21, 21, 16)       0
 2D)

 flatten (Flatten)           (None, 7056)              0

 dense (Dense)               (None, 64)                451648

 dense_1 (Dense)             (None, 1)                 65
```

178/2 = 89 → given the region size 2x2

7056 = 21*21*16 see previous layer

64 (see previous layer) + 1

Compile & Train the model

To compile a model in Keras, you need to define the optimizer, loss function, and optional metrics.

```
epochs = 5 #for timing constraints, a common number for instance is 20-25
model.compile( # this is very important, define the optimizer, the loss and the metrics to track
    optimizer=keras.optimizers.Adam(1e-3), # for regularization
    loss="binary_crossentropy",
    metrics=["accuracy"],
)


logdir='logs' # 1.02
tensorboard_callback = tf.keras.callbacks.TensorBoard(log_dir=logdir) # it allows to create the checkpoint to check the log, to save a tmp model, to see how your


# fit the data
hist = model.fit(
    train_ds, # training data
    epochs=epochs, # how long to train
    validation_data=val_ds, # we pass then the validation, we can see how the model performs in real time
    callbacks=[tensorboard_callback] # pass the callback for the checkpoint

)


# we save the output in hist in order to retrieve the information about the training of the model
```

The compilation step prepares the model for training by specifying how it should be optimized and evaluated.

Compile & Train the model

To compile a model in Keras, you need to define the optimizer, loss function, and optional metrics.

```
epochs = 5 #for timing constraints, a common number for instance is 20-25
model.compile( # this is very important, define the optimizer, the loss and the metrics to track
    optimizer=keras.optimizers.Adam(1e-3), # for regularization
    loss="binary_crossentropy",
    metrics=["accuracy"],
)


logdir='logs' # 1.02
tensorboard_callback = tf.keras.callbacks.TensorBoard(log_dir=logdir) # it allows to create the checkpoint to check the log, to save a tmp model, to see how your


# fit the data
hist = model.fit(
    train_ds, # training data
    epochs=epochs, # how long to train
    validation_data=val_ds, # we pass then the validation, we can see how the model performs in real time
    callbacks=[tensorboard_callback] # pass the callback for the checkpoint

)


# we save the output in hist in order to retrieve the information about the training of the model
```

We train the model using the fit function

Compile & Train the model

To compile a model in Keras, you need to define the optimizer, loss function, and optional metrics.

```
epochs = 5 #for timing constraints, a common number for instance is 20-25
model.compile( # this is very important, define the optimizer, the loss and the metrics to track
    optimizer=keras.optimizers.Adam(1e-3), # for regularization
    loss="binary_crossentropy",
    metrics=["accuracy"],
)


logdir='logs' # 1.02
tensorboard_callback = tf.keras.callbacks.TensorBoard(log_dir=logdir) # it allows to create the checkpoint to check the log, to save a tmp model, to see how your


# fit the data
hist = model.fit(
    train_ds, # training data
    epochs=epochs, # how long to train
    validation_data=val_ds, # we pass then the validation, we can see how the model performs in real time
    callbacks=[tensorboard_callback] # pass the callback for the checkpoint

)


# we save the output in hist in order to retrieve the information about the training of the model
```

We train the model using the fit function

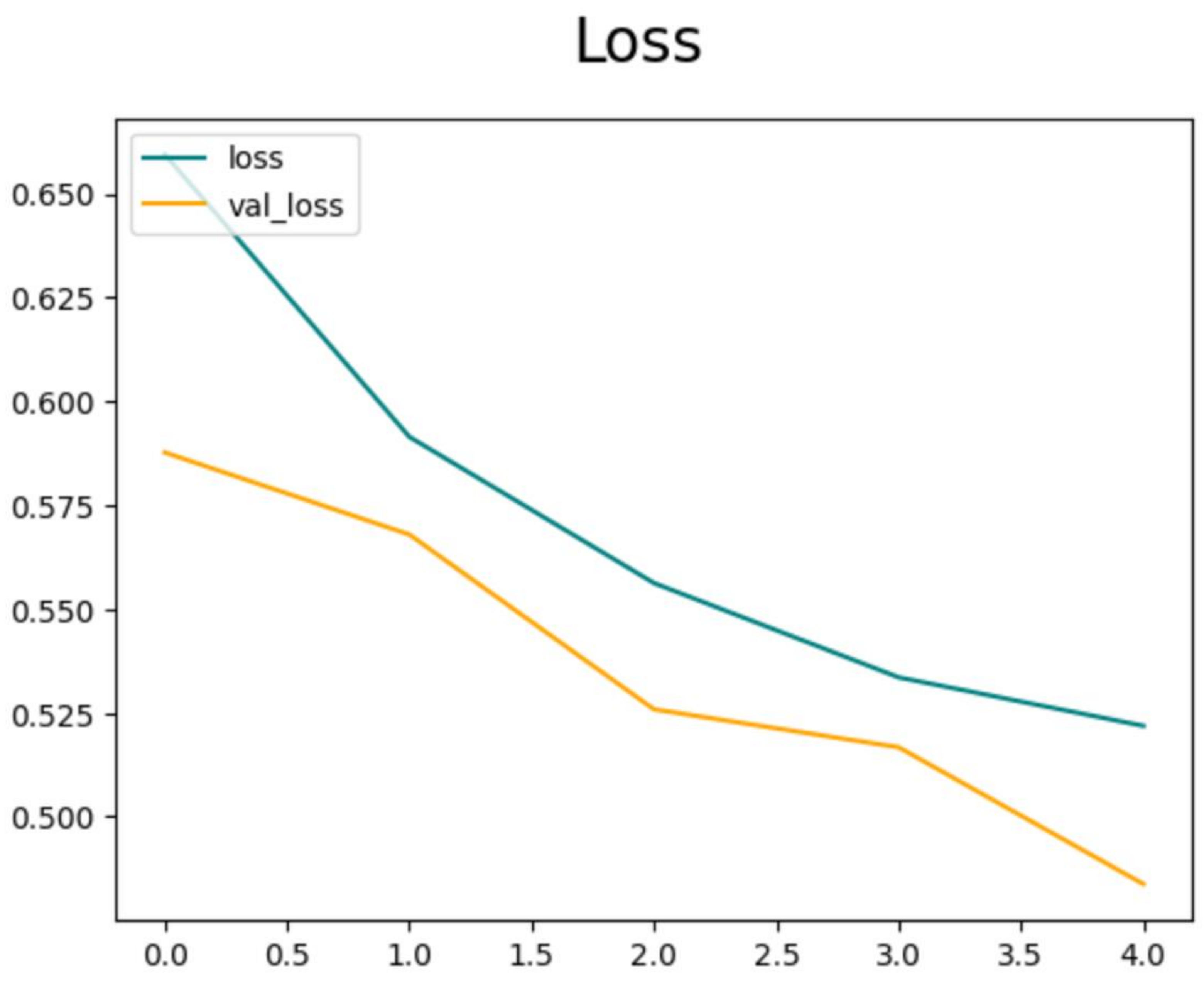
```
Epoch 1/5
147/147 [==============================] - 89s 546ms/step - loss: 0.6595 - accuracy: 0.6022 - val_loss: 0.5877 - val_accuracy: 0.6912
Epoch 2/5
147/147 [==============================] - 89s 598ms/step - loss: 0.5915 - accuracy: 0.6816 - val_loss: 0.5680 - val_accuracy: 0.7063
Epoch 3/5
147/147 [==============================] - 86s 568ms/step - loss: 0.5563 - accuracy: 0.7145 - val_loss: 0.5259 - val_accuracy: 0.7420
Epoch 4/5
147/147 [==============================] - 88s 587ms/step - loss: 0.5336 - accuracy: 0.7333 - val_loss: 0.5167 - val_accuracy: 0.7478
Epoch 5/5
147/147 [==============================] - 83s 552ms/step - loss: 0.5219 - accuracy: 0.7407 - val_loss: 0.4838 - val_accuracy: 0.7757
```

↓ loss ↑ accuracy

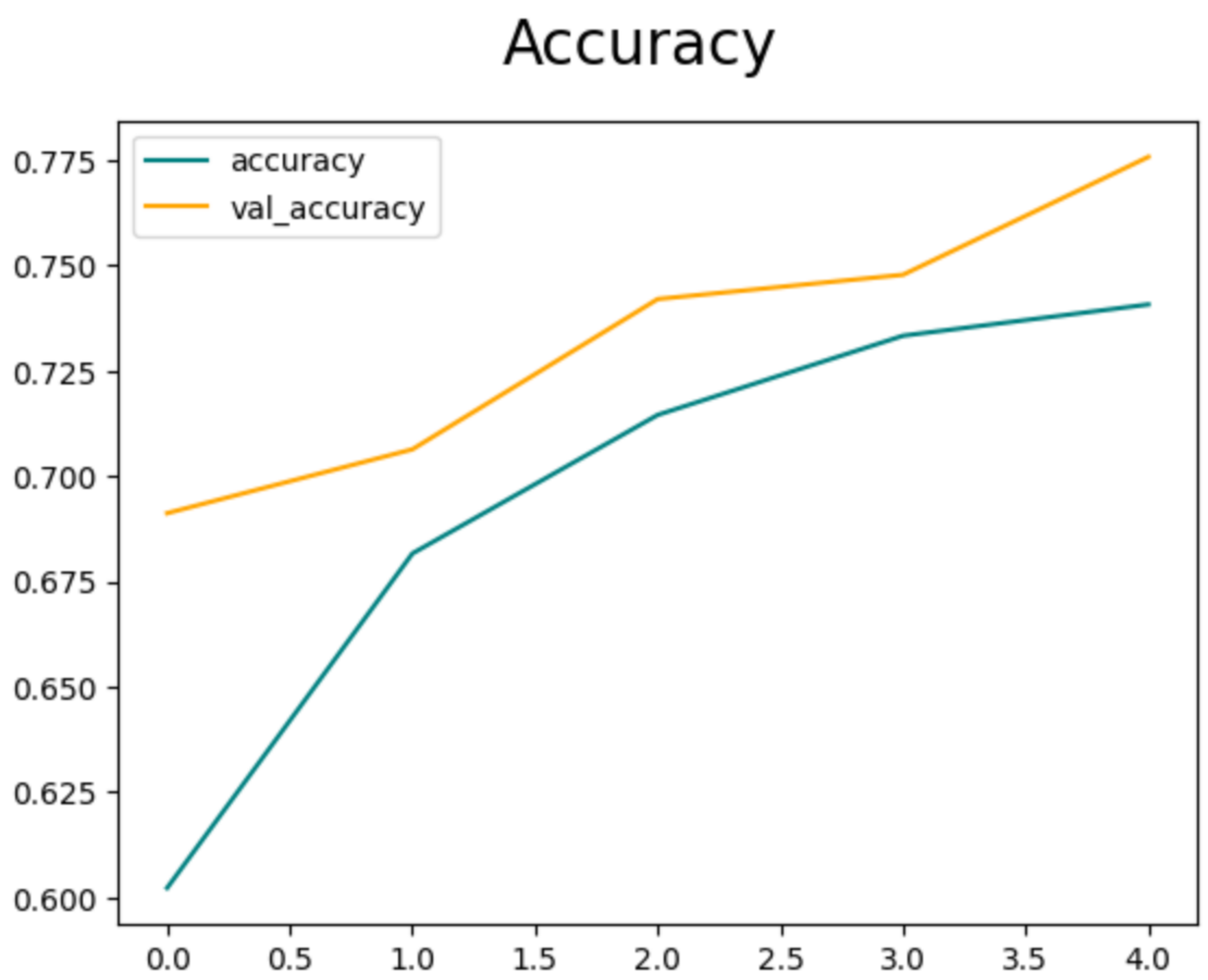
Evaluate the model

Let's display the trend of the loss and the accuracy per epoch:

```
fig = plt.figure()
plt.plot(hist.history['loss'], color='teal', label='loss')
plt.plot(hist.history['val_loss'], color='orange', label='val_loss')
fig.suptitle('Loss', fontsize=20)
plt.legend(loc="upper left")
plt.show()
```




```
fig = plt.figure()
plt.plot(hist.history['accuracy'], color='teal', label='accuracy')
plt.plot(hist.history['val_accuracy'], color='orange', label='val_accuracy')
fig.suptitle('Accuracy', fontsize=20)
plt.legend(loc="upper left")
plt.show()
```



Test on new samples

We test the model on new samples using predict function:

```
resize = tf.image.resize(rgb_img, image_size)
plt.imshow(resize.numpy().astype(int))
plt.show()
```

```
yhat = model.predict(np.expand_dims(resize/255, 0)) # the resize image has to be divided by 255
```

```
yhat
```

```
if yhat > 0.5:  # this happens in binary classification
    print(f'Predicted class is Dog')
else:
    print(f'Predicted class is Cat')
```


```
array([[0.38692367]], dtype=float32)
```

CAT

Test on new samples

```
resize = tf.image.resize(rgb_img, image_size)
plt.imshow(resize.numpy().astype(int))
plt.show()
```

```
yhat = model.predict(np.expand_dims(resize/255, 0)) # the resize image has to be divided by 255
```

```
yhat
```

```
if yhat > 0.5:  # this happens in binary classification
    print(f'Predicted class is Dog')
else:
    print(f'Predicted class is Cat')
```

```
array([[0.6478666]], dtype=float32)
```

DOG

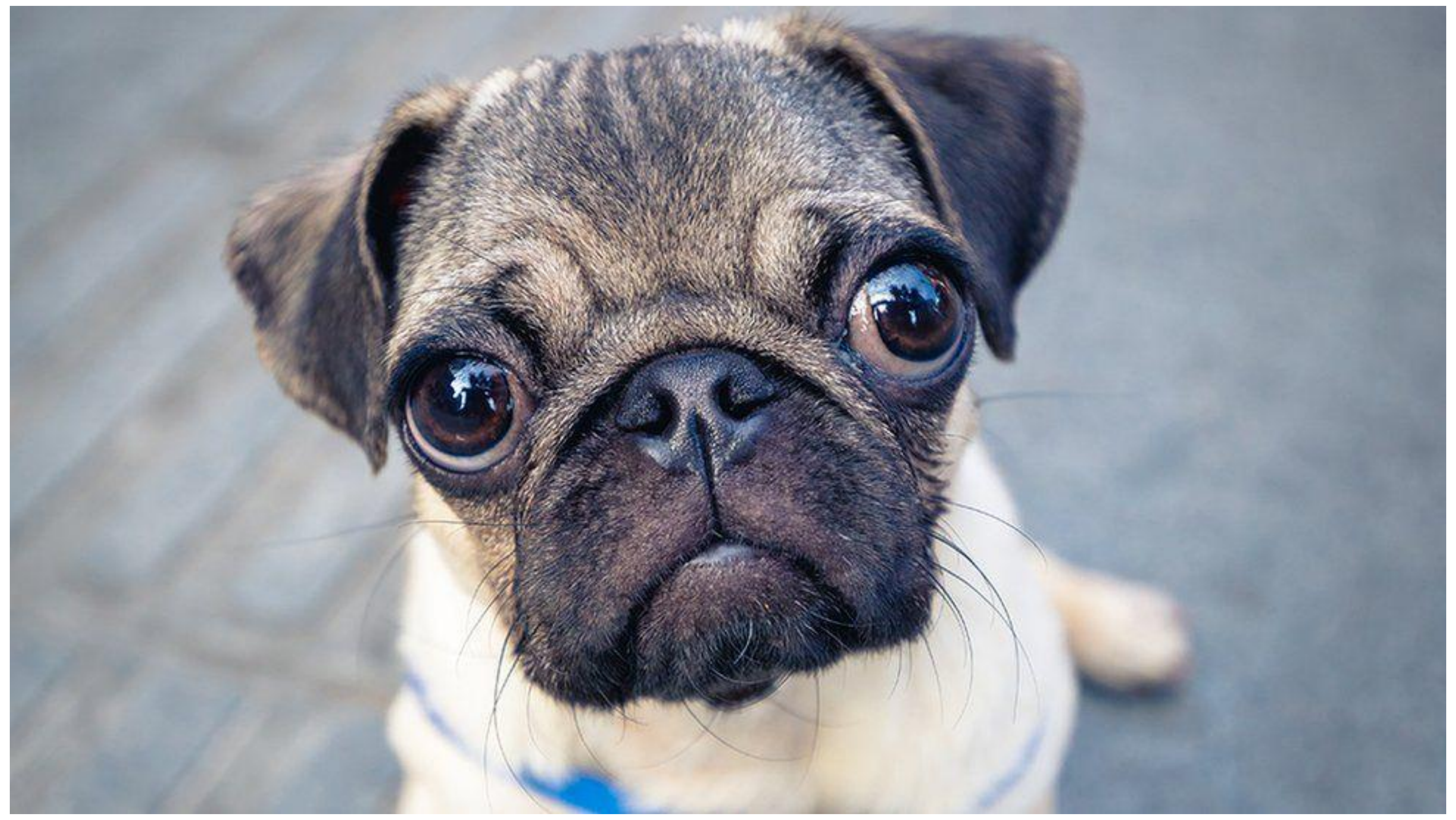
Test on new samples

```
resize = tf.image.resize(rgb_img, image_size)
plt.imshow(resize.numpy().astype(int))
plt.show()
```

```
yhat = model.predict(np.expand_dims(resize/255, 0)) # the resize image has to be divided by 255
```

```
yhat
```

```
if yhat > 0.5:  # this happens in binary classification
    print(f'Predicted class is Dog')
else:
    print(f'Predicted class is Cat')
```



```
array([[0.58503526]], dtype=float32)
```

DOG

Test on new samples

```
resize = tf.image.resize(rgb_img, image_size)
plt.imshow(resize.numpy().astype(int))
plt.show()
```

```
yhat = model.predict(np.expand_dims(resize/255, 0)) # the resize image has to be divided by 255
```

```
yhat
```

```
if yhat > 0.5:  # this happens in binary classification
    print(f'Predicted class is Dog')
else:
    print(f'Predicted class is Cat')
```

```
array([[0.9201538]], dtype=float32)
```

DOG

Save the Model

Keras provides the function to save and load the model in order to use it again:

```
from tensorflow.keras.models import load_model # to save and load then the model
```

```
model.save(os.path.join('models','imageclassifier.h5'))
```

```
new_model = load_model(os.path.join('models','imageclassifier.h5'))
```

```
new_model.predict(np.expand_dims(resize/255, 0))
```

```
1/1 [==============================] - 0s 126ms/step
array([[0.8562703]], dtype=float32)
```

An .h5 file, or Hierarchical Data Format 5 file, is a data file format commonly used in scientific computing and data analysis. It is designed to store and organize large amounts of numerical data and metadata in a hierarchical structure.

Questions?