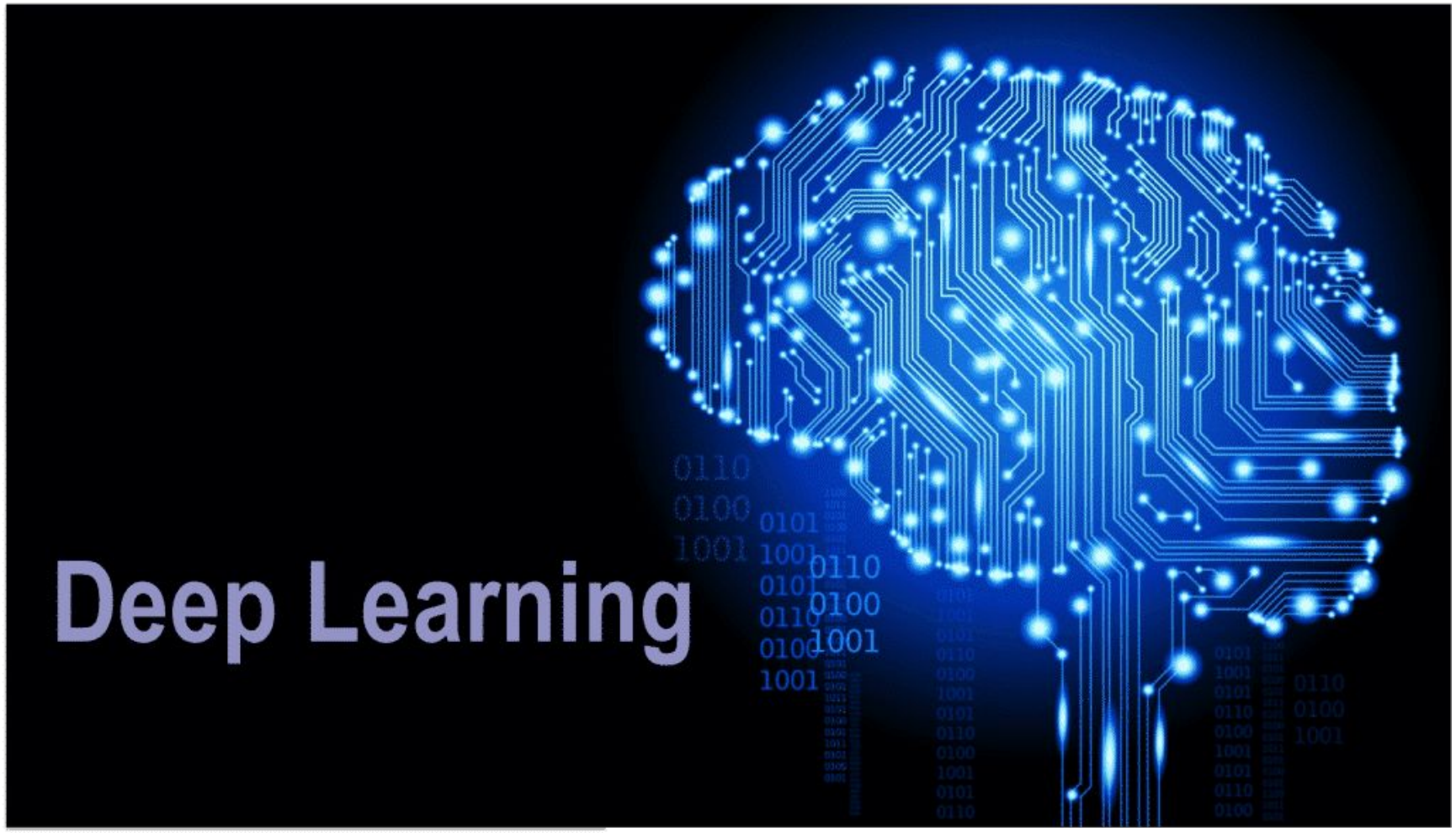

Deep Learning

Last lecture reminder

We learned about:

- Deep Learning - Introduction
- Artificial Neural Network (ANN) - Introduction
- Artificial Neural Network (ANN) - The Neuron
- Artificial Neural Network (ANN) - Activation Function
- Artificial Neural Network (ANN) - Backpropagation
- TensorFlow - Introduction
- Artificial Neural Network (ANN) - Python Example

Image Processing - Introduction

Image processing → This field involves the use of algorithms and techniques to perform operations on digital images to enhance them or extract useful information. It is a type of signal processing in which the input is an image and the output can either be an image or a set of characteristics or parameters related to that image.

Some techniques used in image processing include:

- **Image Enhancement** → This is used to improve the quality of an image, making it easier for humans or machines to interpret it. Techniques include brightness and contrast adjustment, noise removal, and sharpening.
- **Image Restoration** → This corrects distortions and degradations in an image, such as motion blur, camera misfocus, etc.
- **Image Segmentation** → This is the process of dividing an image into distinct regions that have similar attributes.
- **Image Recognition** → This technique identifies objects, features, or activities in an image.
- **Image Compression** → This used to reduce the volume of data required to store or transmit images.

Convolutional Neural Network - CNN

In this part we are going to focus on a main aspect of image processing called **Image Recognition**.

Convolutional Neural Network (CNN) → A Convolutional Neural Network (CNN) is a deep learning algorithm that takes in an input image, assigns importance (or weights) to various aspects of the image, and is able to differentiate one from the other.
This type of network is primarily used in the area of image and video recognition and processing. It's designed to automatically and adaptively learn spatial features through backpropagation.
Just like an ANN, a CNN is also a supervised learning model and it needs labeled data to learn.

The basic idea behind CNN is using specific features and focus on them instead of scanning the entire image each time. Once the model learn that existent of some features can be interpreted to recognition of the image it will use this knowledge in order to predict on future images.

Convolutional Neural Network - CNN

CNN build from 3 main layers:

- **Input layer** → This is where the image data is fed into the model. Each neuron in this layer corresponds to one pixel in the input image. Each pixel's value (0 or 1 for black and white photos or RGB value for colorful photos) are inputted as values into the corresponding neurons.

- **Convolutional Layer** → This layer is the key component of CNN. The purpose of this layer is to perform convolution operation in order to extract meaningful features from the input image. Those features will help the model to recognize properly the image given to him.

- **Fully Connected Layer (Output layer)** → The final layer in CNN is basically a regular ANN layer. It called fully connected because each input from the input layer of the ANN will be connected to each neuron in the hidden layer. This layer responsibility is to processing the extracted features and returning the final result.

CNN - Image Representation

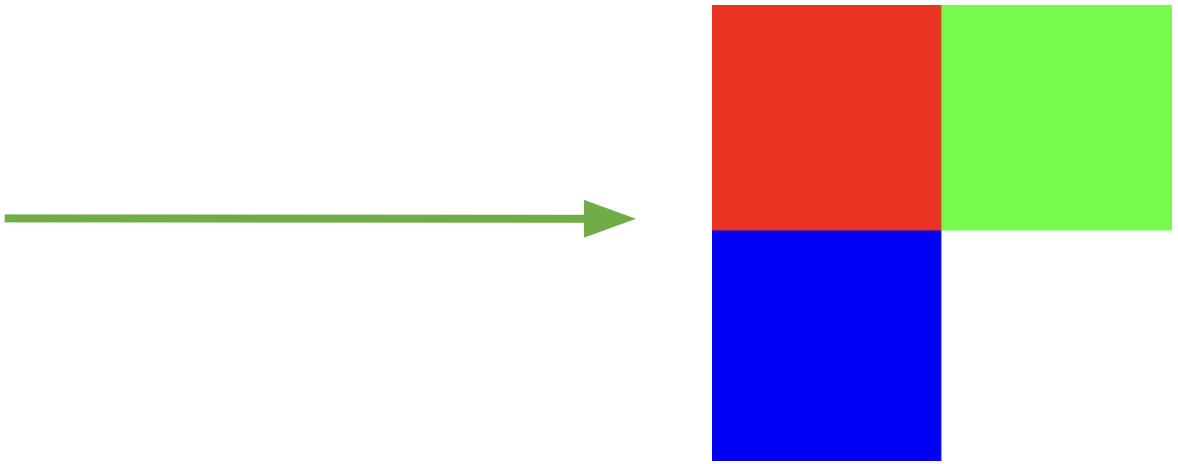
As we learned previously in the course, image can be represented as a 2D array of numbers.

Each cell in the array represent a pixel and each number (or set of numbers) represent the pixel color.

For example → Let's consider that we have an image of 2x2 pixels with red, green, blue and white colors.

The 3D array for this image could look like this:

```
image = [
        [
            [255, 0, 0], [0, 255, 0]
        ],
        [
            [0, 0, 255], [255, 255, 255]
        ]
    ]
```



CNN - Convolution Operation

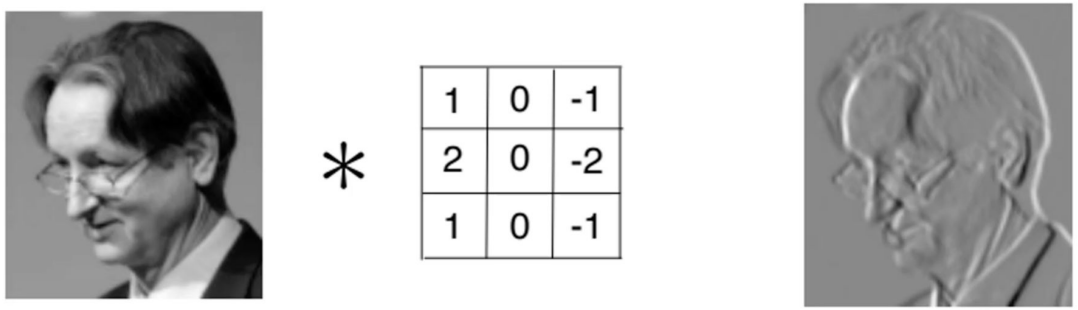
Convolution → Convolution is a mathematical operation perform on two functions, producing a third function that represents how the shape of one is modified by the other.
In the context of image processing in a Convolutional Neural Network, the convolution operation usually involves taking a small 2D filter (first function) and applying it onto the original 2D image (second function), producing a new 2D image (third function).

Convolution Filter → Convolution filters are matrices of weights that slide over an input image to extract certain features. The numbers in these matrices (weights) determine how much emphasis to place on each part of the input they cover.

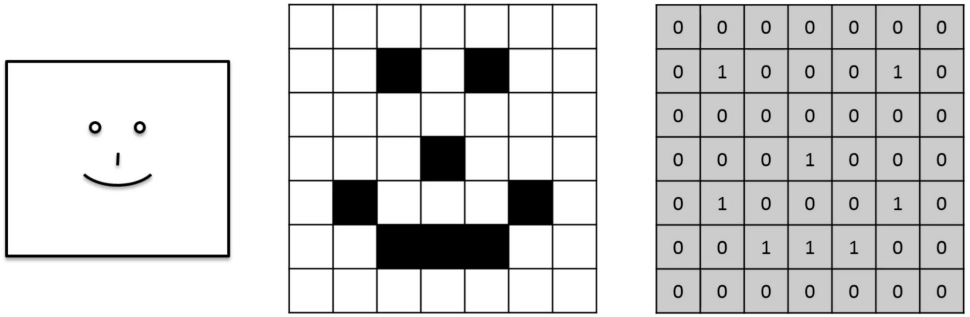
- **Positive Values** → These weights emphasize parts of the image.
- **Negative Values** → These weights de-emphasize or subtract from parts of the image.
- **Zero Values** → These weights ignore parts of the image.

CNN - Convolution Operation

For example → By applying the following convolution filter on the given photo input we will get the following output image.



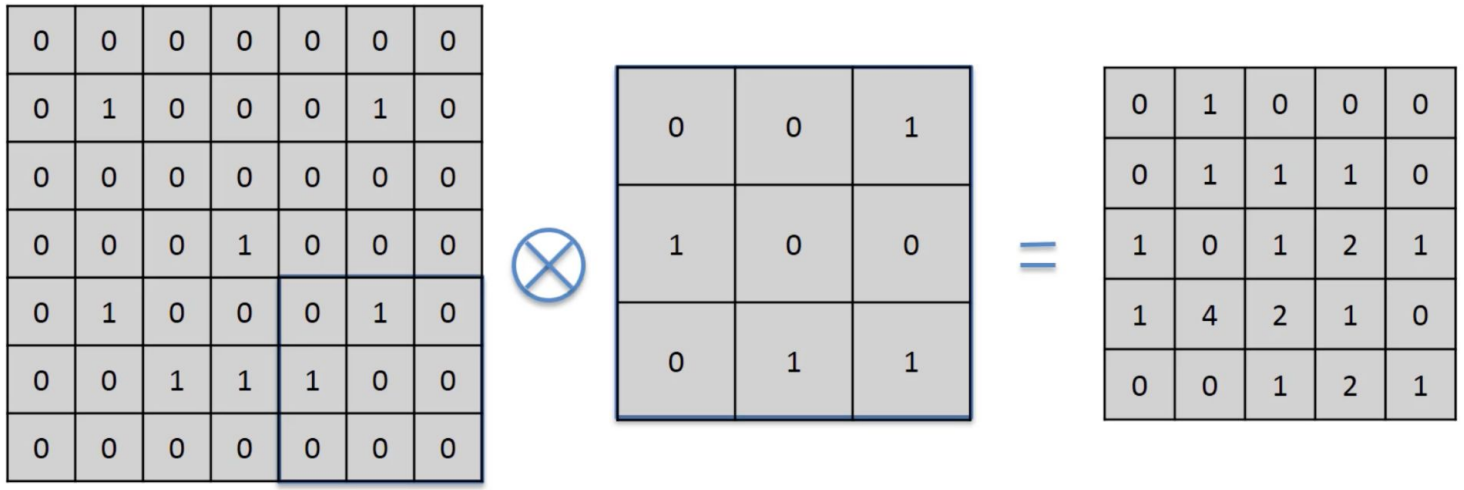
In order to better understand how the convolution operation is working, let's take this simple smiling face photo converted to 2D array.



CNN - Convolution Operation

Now, let's take a filter which is basically a subset of a photo and can described as the first function in the convolution operation. In our example the filter will be a 3X3 matrix.

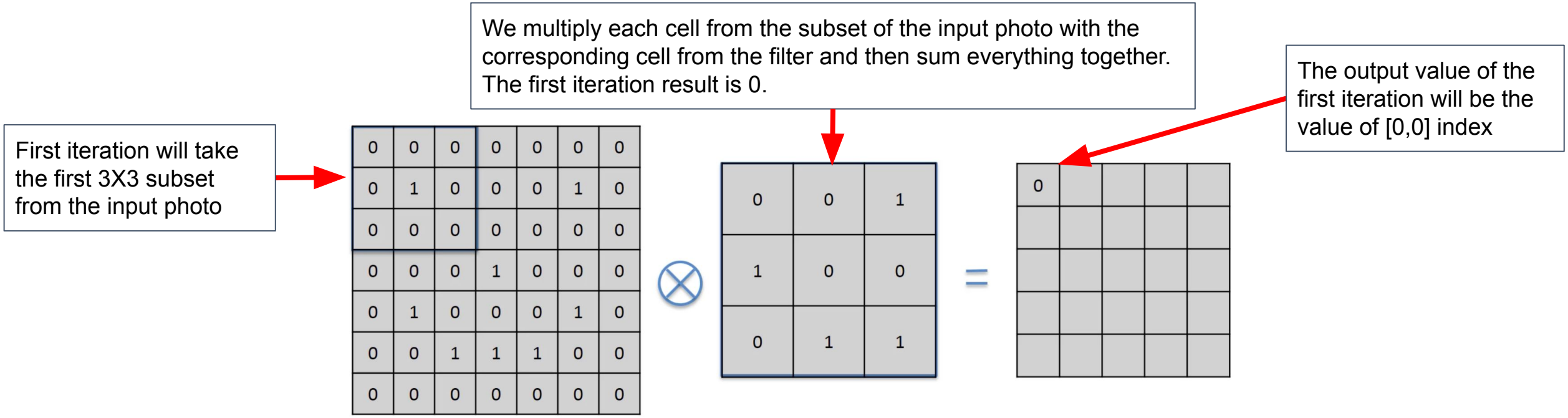
In addition to the filter we will take the input photo which can represent the second function and apply the convolution operation to create the output photo that can described as the third function.



CNN - Convolution Operation

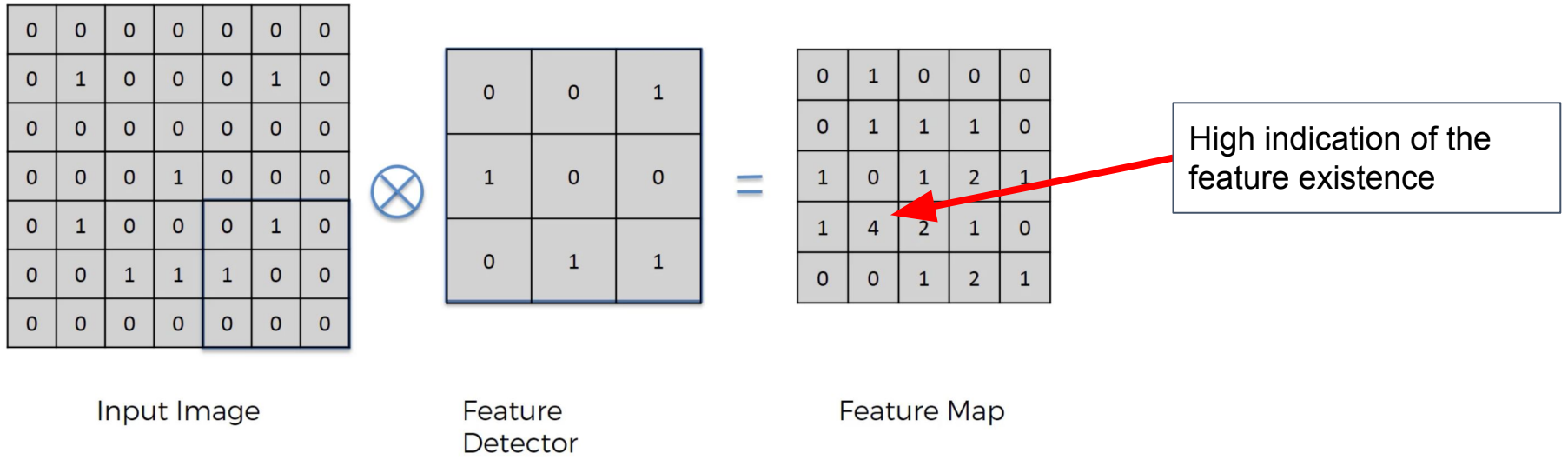
Let's understand how we managed to build the third image. Basically what the algorithm is doing is splitting the input image to 3X3 separated subsets and multiply each cell from the input subset with the corresponding cell in the filter subset. Then adding all the results together and put the outcome in the output subset.

For example → The first iteration will be as following:



CNN - Convolution Operation

The algorithm will keep perform those iterations until it will get the final result which also called the **Feature Map**.

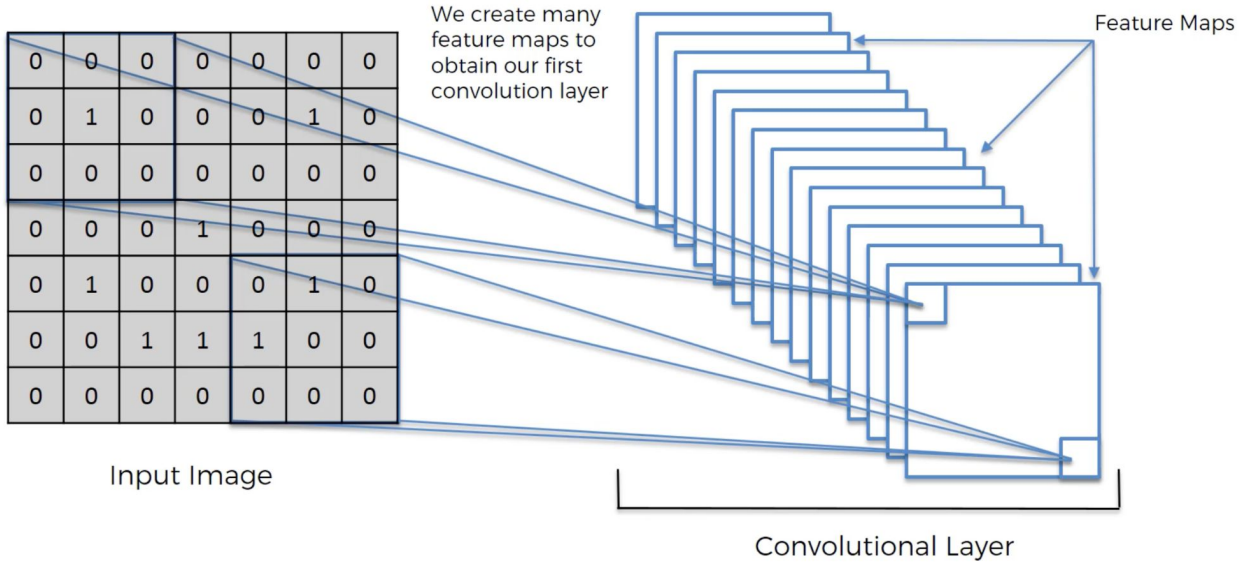


We can also see that we managed to reduce the size of the output photo from 7x7 to 5X5.

What is also important is that we were able to reduce the size but keep the features as part of the output image. The higher the number in the Feature Map the higher the indication of existent of feature in that place.

CNN - Convolutional Layer

In a common CNN it can be multiple filters resulting multiple Feature Maps. Those features maps creates the convolutional layer in our CNN.



The number of filters is a parameter that we need to configure before starting out CNN model.

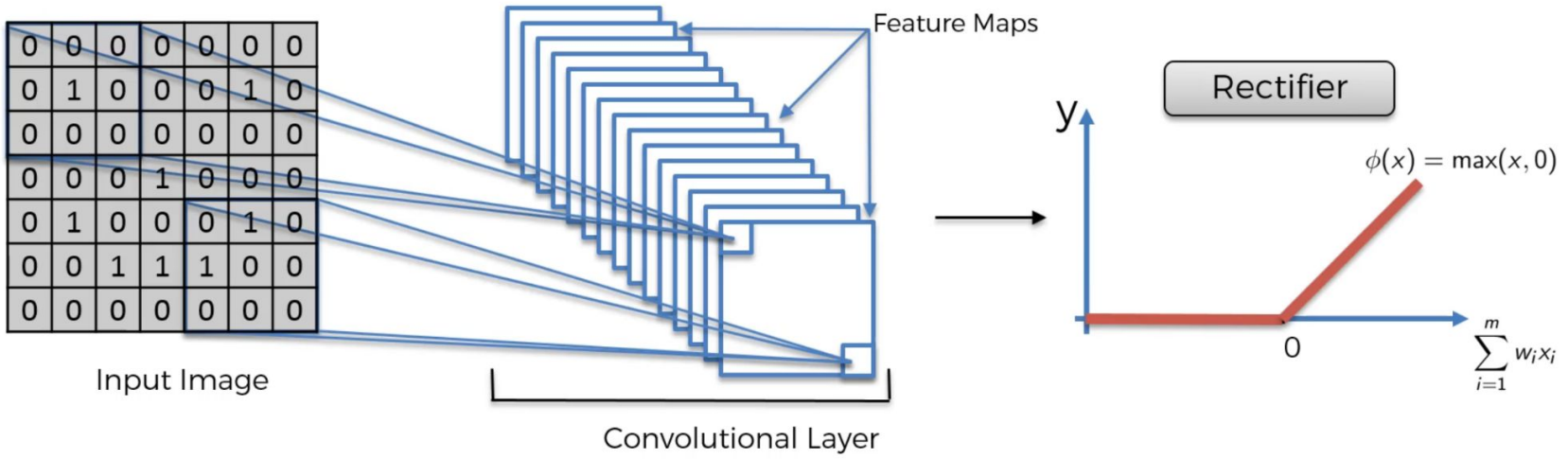
The more filters we added the more complex our model will become.

The filters themselves are first generated randomly and been optimized using the backpropagation process.

CNN - Appling ReLU Activation Function

Once we have our convolutional layer (built from multiple feature maps) the model will move to the next phase which is appling the ReLU activation function on each feature map.

The ReLU activation function keep positive values of the feature map and transform negative values to 0.



The main role of the ReLU activation function is to break the linearity between the different feature maps. Without non-linear activation functions, our CNN would only be able to model linear relationships, limiting their ability to capture complex patterns.

CNN - Pooling

The next phase is applying **Pooling** on our non-linear feature maps.

Pooling → Pooling is a crucial operation in CNNs that involves downsampling the feature maps produced by convolutional layers. It serves to reduce the spatial dimensions, enhance translation invariance, and mitigate overfitting.

There are 3 types of pooling:

- **Max pooling** → Selecting the maximum value within the pooling window.
- **Average pooling** → Calculating and select the average value of the entire cells within the pooling window.
- **Min pooling** → Selecting the minimum value within the pooling window.

Max pooling and average pooling are the most common types, with max pooling being more widely used.

CNN - Pooling

Why pooling is a crucial part in the CNN flow?

- **Reduces Complexity** → By shrinking the size of the data pooling reduces the amount of computation needed in the network. This speeds up the processing and lowers the computational cost.
- **Helps in Generalization** → Pooling makes the network less sensitive to small changes or variations in the input data (like slight shifts or distortions in an image). It helps the network generalize better to new, unseen data. This is very useful when the features we are trying to find are located in slightly different places in each picture (like head that turned around or the picture itself is slightly rotated).
- **Avoids Overfitting** → By focusing on the most important features and reducing the overall size, pooling helps in avoiding overfitting. Overfitting means the model performs well on training data but poorly on new data.

We can think of pooling like summarizing a book chapter, Max Pooling will pick out the most important sentence from each paragraph and Average Pooling will summarize the key point of each paragraph.
In both cases, we end up with a much shorter, still meaningful version of the chapter. Similarly, pooling reduces the size of data while preserving essential information.

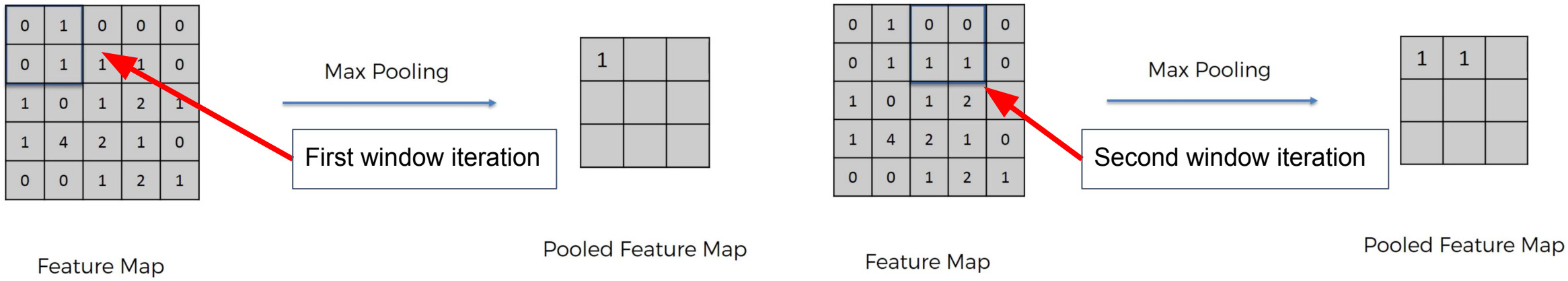
CNN - Pooling

How pooling works?

First we need to decide the pooling type and the size of the pooling window. Then the model will iterate on each feature map with a sliding window action (meaning it will move the pooling window on each iteration) and select the value according to the pooling type we selected.

The last step is contracting the Pooled Feature Map from those values.

For example → Let's apply max pooling with window of 2X2:

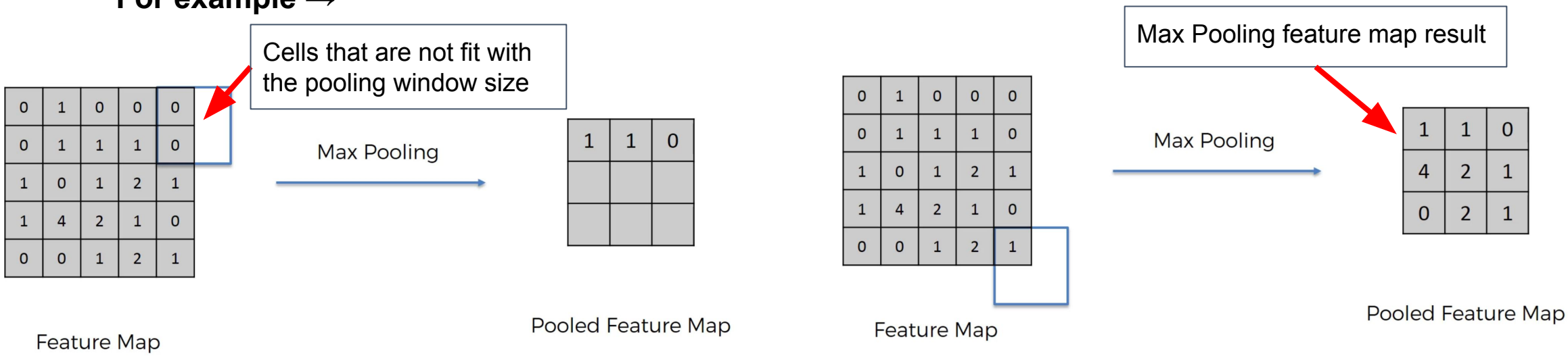


CNN - Pooling

Keep in mind that unlike with the filtering, in pooling there is no overlap between windows which means that every cell in the feature map will be part of exactly 1 pooling window.

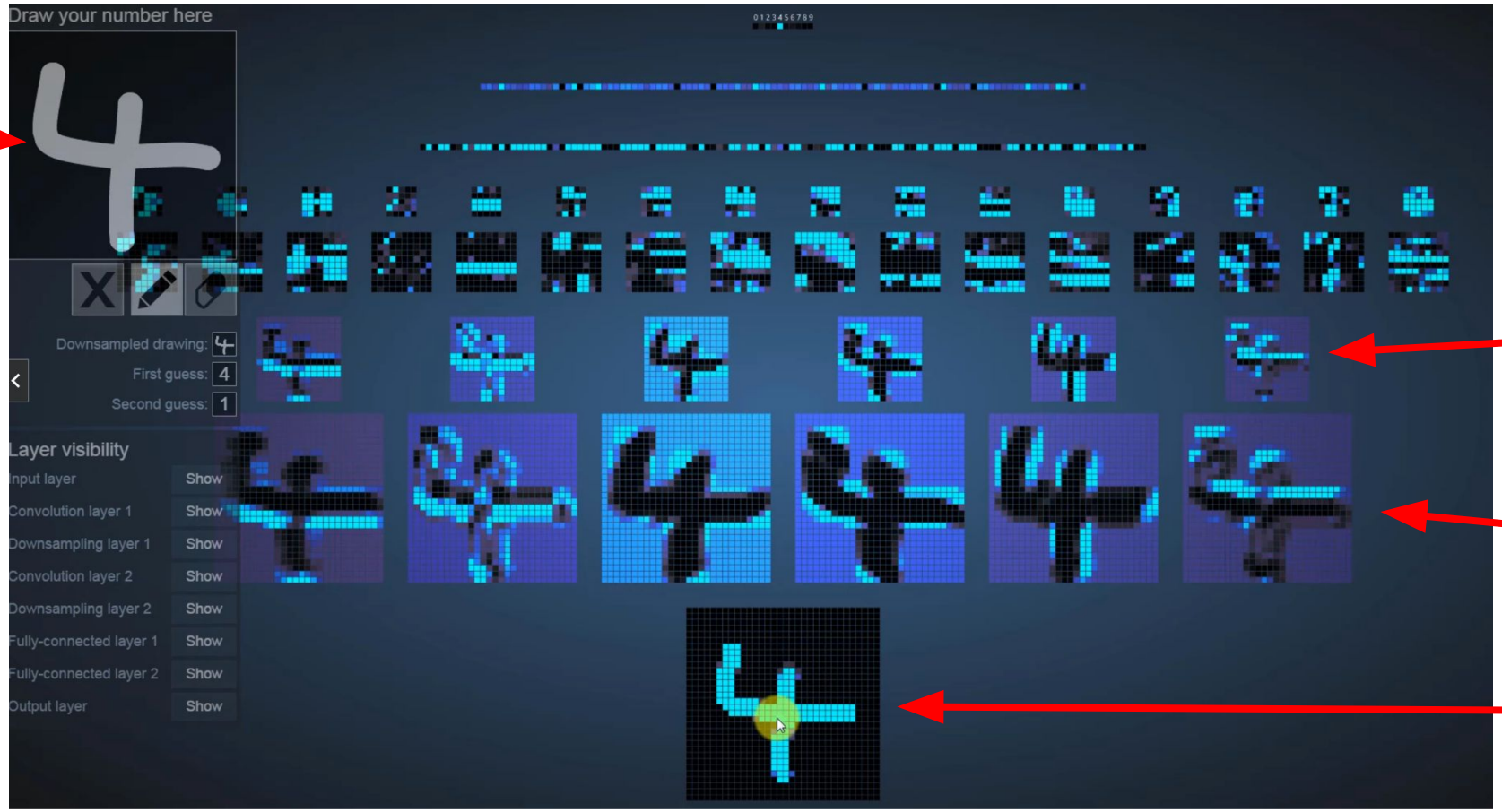
In case there are cells that can't fit to an entire pooling window (like in our example) the model will calculate the value according to the cells that fit to this window.

For example →



CNN - Pooling

In a real example (picture of the number 4) it will look like the following:



Input picture

Pooling layer results
The pooled feature maps

Convolutional layer result
with multiple filters
(the feature maps)

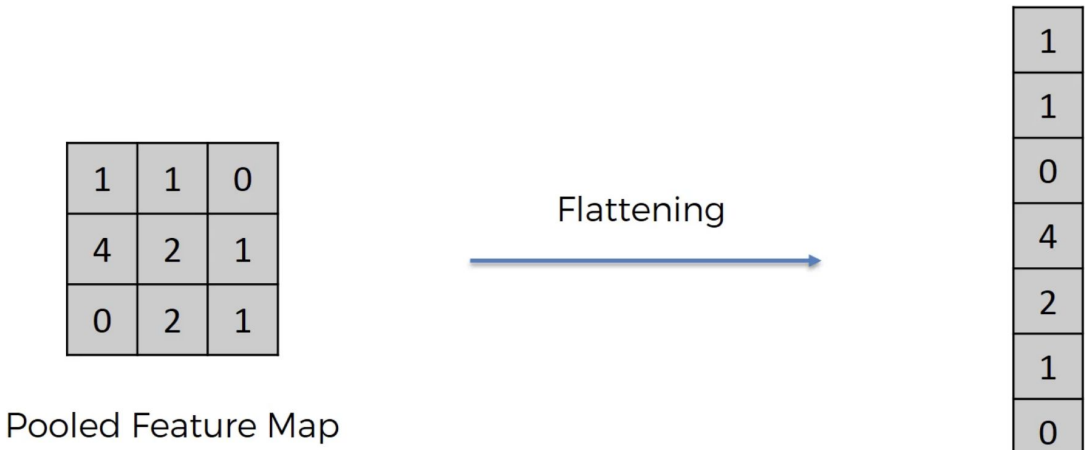
2D representation
of the input picture

CNN - Flattening

The final step in creating the output of the convolutional layer (that will be the input for the fully connected ANN layer) is flattening the pooled feature maps arrays.

This is a very simple step in which each pooled feature map array been flatten from nXn array to nX1 array according to their original order.

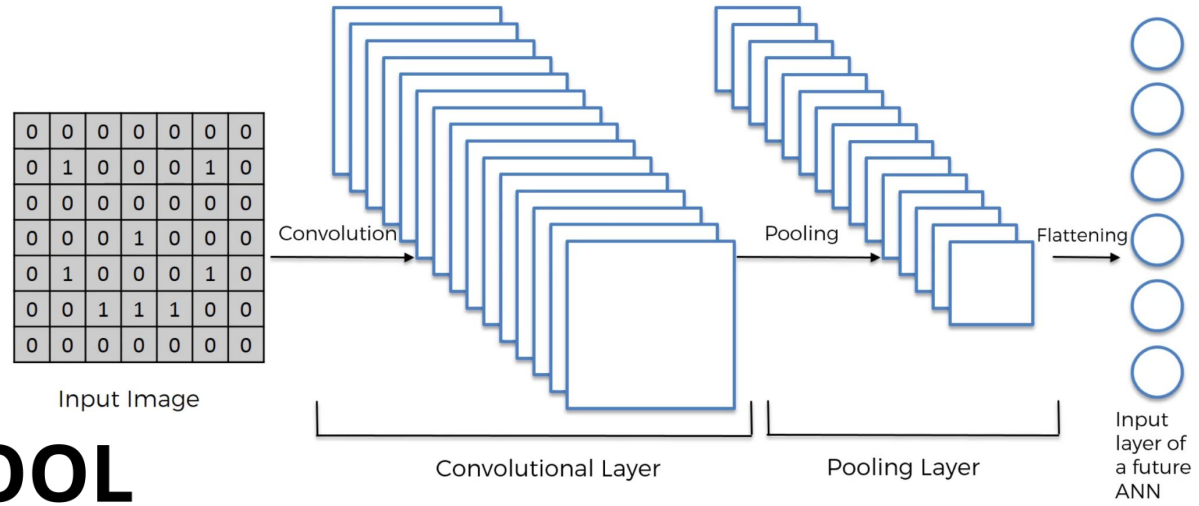
For example →



CNN - Convolutional Layer Conclusion

In conclusion, in order to produce the inputs for the fully connected ANN layer, the convolutional layer perform the following:

1. Takes any input image and produce feature maps according to the generated filters.
2. Execute LeRU activation function on each feature map to remove linearity.
3. Execute pooling on each feature map to handle picture variance and reduce size
4. Flattening each pooled feature maps to a single array. Each single array will be 1 input for the fully connected ANN layer.



CNN - Fully Connected Layer

Fully connected layer → In CNN, the fully connected layer is responsible for providing the final model output (the image recognition). The fully connected layer is essentially a traditional artificial neural network (ANN) layer where each neuron is connected to every neuron in the previous layer, which is why it is called fully connected.

Backpropagation in CNN → In CNN, the backpropagation process updates the weights of all layers, including the filters in convolutional layers and the weights in fully connected layers. This process ensures that the entire network is optimized to minimize the loss function. The filters in convolutional layers are adjusted during backpropagation, making them more meaningful as the network learns to better capture necessary features. Gradient descent or its variants are used to calculate these updates.

CNN - Fully Connected Layer

Activation function → In the CNN fully connected layer, the activation function of the hidden layer will often be LeRU (in order to introduce non-linearity to the model).
The activation function of the output layer will be determined according to the training data label.

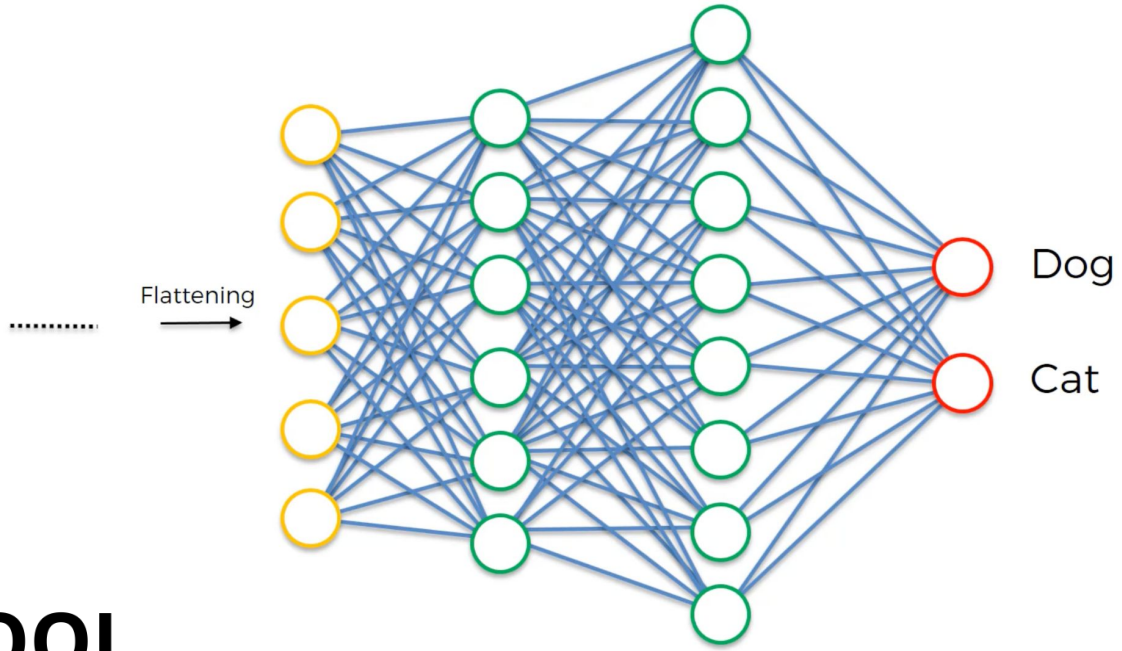
- **Binary classification** → In case of binary classification (For example, Cat / Dog) we will choose the **Sigmoid** activation function.
- **Multiple classification** → In case of multiple classification we can choose the **Softmax** activation function.

Final prediction → Once the model managed to optimized all it's layers (filters, input weights, etc…)
It can now move to performing the prediction. The number of output neurons is determined by the number of optional classes, in case of binary classification we can have 1 neuron and in case of multiple classification we should have an output neuron for each class. Once the training process has completed the weight values will no longer changes regardless the input.

CNN - Fully Connected Layer Example

In order to understand how the model manage to predict the image classification let's take a look on a simple example. In this example we trained our model on large dataset of Cat and Dog images. The model should now be able to predict if a given input image is a Cat or a Dog.

The model fully connected layer will look like this:



CNN - Fully Connected Layer Example

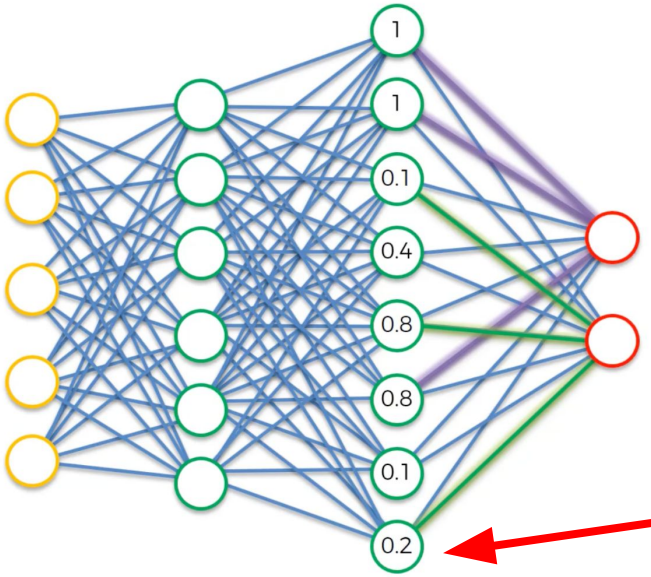
As we learned, once the training process has complete, each input has its own weight value.

For new image, the model will calculate those weight values for each input coming from the Convolutional layer. If the model training done correctly it should has managed to set high weights values on the input features that can indicate the most if a given image is a cat or a dog.

The Dog / Cat neuron will provide output incase its the neuron with the highest probability value.

For example → In case of a dog image:

Flattening

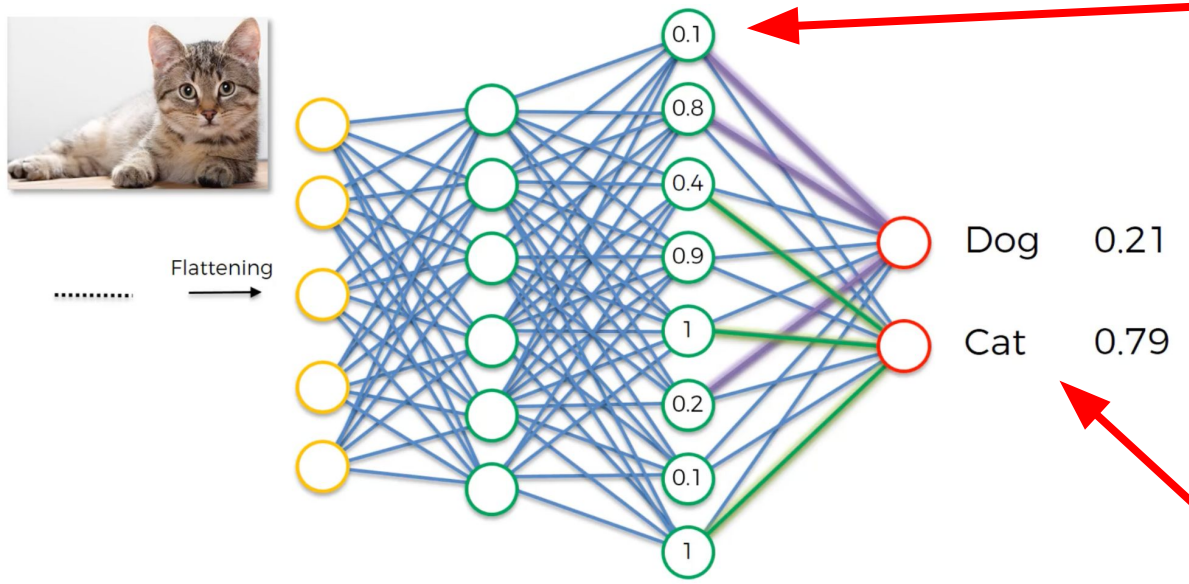


The activation value according the the input weights return 0.95 probability for the dog neuron so this neuron will trigger and provide the prediction.

Input values calculated with the fixed weights vales

CNN - Fully Connected Layer Example

For example → In case of a cat image:



Input values calculated with the fixed weights vales

The activation value according the the input weights return 0.79 probability for the cat neuron so this neuron will trigger and provide the prediction.

As we can see from the example, different input images yield different activations (values) due to the uniqueness of their features (even that the weights are the same) resulting different probability to get each class. The neuron with the highest probability will be triggered.

CNN - Python Example

For the full CNN Python example we will use the **‘CNN_dog_cat_dataset.zip’** file that contain 3 folders:

- **training_set** → Contain 8000 different images of cats and dogs.
- **testing_set** → Contain 2000 different images of cats and dogs.
- **single_prediction** → Contain 1 image of a cat and 1 image of a dog for the final model testing.

When applying this folder as the CNN input dataset its important to keep the folder directory structure as the following:

dataset/training_set/cats/

cat001.jpg

cat002.jpg

dataset/training_set/dogs/

dog001.jpg

dog002.jpg

CNN - Python Example

Proper folder directory and proper image naming is important because this is how TensorFlow manage to determine the correct label for each image. We don't providing to the model anything else except folders of images.

In our Python example we are going to build a full CNN model constructed from 2 convolutional layers, a flattening layer and a fully connected ANN layer.

Important Note: Because training CNN model is extremely complicated it required a lot of resources from our computer and it's not possible to perform this training officially without **Nvidia GPU** (graphics processing unit).
In order to solve this issue we will use **Google Colab** which allowing us to run Python code (like in jupyter notebook) and also configuring Nvidia GPU.

CNN - Python Example

First open your Drive in your Google Gmail account (if you don't have one, create one for free).

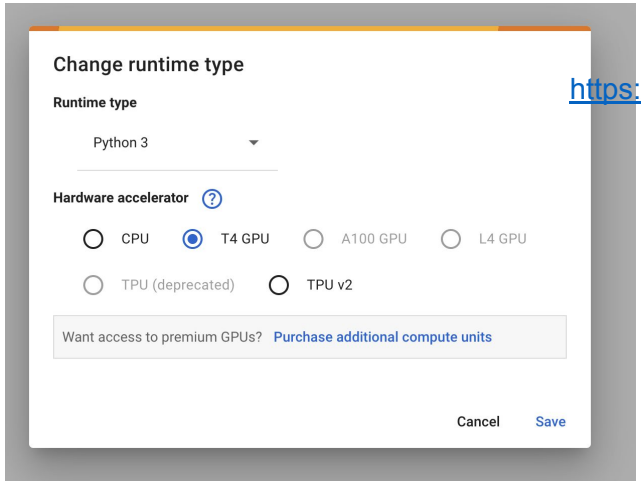
Load the **'CNN_dog_cat_dataset.zip'** file to a folder in your main Google Drive folder.

Next, open Google Colab on this link → https://colab.research.google.com/

Choose a New notebook and give it a name 'CNN_Python_Example.ipynb'.

Select Runtime → Change runtime type → T4 GPU and Save.

(This configure the Notebook to use Nvidia T4 GPU)



https://colab.research.google.com/

CNN - Python Example

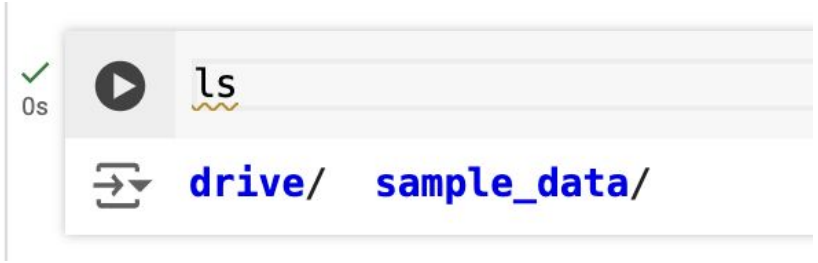
Now we need command Google Colab to extract the zip folder from our Google Drive and unzip it.

This process will create a new unzipped folder inside the Google Colab machine.

Keep in mind that you will need to provide Google Colab permissions to your Google Drive.

In order to perform it we will run the following code:

We can check that it works by run ls and see the **drive/** folder inside the Google Colab machine.



CNN - Python Example

Navigate to your zip file and perform the following code:

In order to perform it run the following code:

```
[4]  import zipfile
     zip_file = 'drive/MyDrive/full_stack_ecom/AI/Course_Excel_Files/Deep_Learning/CNN_dog_cat_dataset.zip'

     with zipfile.ZipFile(zip_file, 'r') as zip_ref:
       zip_ref.extractall('/content/dataset')
```

We can check that it works by run ls and see the **dataset/** folder inside the Google Colab machine.

```
[6]  ls

     dataset/  drive/  sample_data/
```

Now that we have the unzipped folder with all the images dataset inside our Google Colab machine we are ready to start the CNN training process.

CNN - Python Example

First let's import all the relevant libraries.

```
[7]  import numpy as np
     import pandas as pd
     import matplotlib.pyplot as plt
     import seaborn as sns
     import tensorflow as tf
```

Then, We need to configure our TensorFlow to use the Nvidia GPU by running the following command.

'Num GPUs Available: 1' will mean that TensorFlow is recognizing the Nvidia GPU and using it.

```
[8]  tf.config.set_visible_devices([], 'GPU')
     print("Num GPUs Available: ", len(tf.config.experimental.list_physical_devices('GPU')))

     Num GPUs Available:  1
```

CNN - Python Example

Next, we will perform image preprocessing on our training and testing datasets.

This preprocessing will include the following:

- **rescale** → Rescales the pixel values of the images from the range `[0, 255]` to the range `[0, 1]` (Normalization feature scaling).
- **shear_range** → Shearing parameter is a geometric transformation that tilts the shape of the image.
- **zoom_range** → Zooming parameter applies random zooming on the images.
  This helps the model to be invariant to different scales of objects in the images.
- **horizontal_flip** → This parameter randomly flips half of the images horizontally during each epoch. This is particularly useful for datasets where horizontal flipping is a valid operation (flipping an image of a cat still results in a valid image of a cat).
  This helps the model to be invariant to horizontal orientations of the objects in the images.

CNN - Python Example

Let’s apply the image preprocessing code itself:

```
[10] from tensorflow.keras.preprocessing.image import ImageDataGenerator

     train_datagen = ImageDataGenerator(
         rescale=1./255,
         shear_range=0.2,
         zoom_range=0.2,
         horizontal_flip=True
     )
```

Dividing each pixel value by 255 resulting pixel values between [0, 1]

Applying image tilting and image zooming with 20% magnitude

When applying `ImageDataGenerator` on the testing dataset, we will not perform shearing, zooming, and horizontal flip because those are considered data augmentation techniques that are typically used only during the training phase. The purpose of data augmentation is to artificially increase the variety of the training dataset, helping the model generalize better by seeing a wider range of transformations. However, in the testing phase, we want to evaluate the model's performance on the original, unaltered data to evaluate its true accuracy and robustness.

CNN - Python Example

The image preprocessing code for the test dataset will contain only the feature scaling.

```
[11] test_datagen = ImageDataGenerator(rescale=1./255)
```

0s

In addition, we will set the following parameters when applying the DataGenerator on the training and testing image datasets:

- **target_zise** → This parameter resizes all images to the specified target size
- **batch_size** → Determine how many images will be in every batch, this will help the model improve performance when processing the image dataset.
- **class_mode** → Determine the type of classification problem. 'binary' mode is used for binary classification problems where there are only two classes (in this case, cats and dogs). The labels will be either 0 or 1. ‘categorical’ mode is used for multi-classification problems.

CNN - Python Example

Let’s apply the ImageDataGenerator on the training and testing datasets.

```
train_set = train_datagen.flow_from_directory(
    'dataset/CNN_dog_cat_dataset/dataset/training_set',
    target_size=(64, 64),
    batch_size=32,
    class_mode='binary'
)

test_set = test_datagen.flow_from_directory(
    'dataset/CNN_dog_cat_dataset/dataset/test_set',
    target_size=(64, 64),
    batch_size=32,
    class_mode='binary'
)
```

```
Found 8000 images belonging to 2 classes.
Found 2000 images belonging to 2 classes.
```

Providing the training set folder path

Applying image resizing to 64X64 pixels

Applying batch size of 32 images in a single batch

Choosing ‘binary’ classification mode

```
print(train_set.class_indices)
```

```
{'cats': 0, 'dogs': 1}
```

This shows that the model classified 1 as Dog and 0 as Cat

CNN - Python Example

Now, let's build the CNN convolutional layers. We want to connect 2 separate convolutional layers. The first layer will get the original images as input and the second layer will get the first layer images outputs as the input.

We will use the **Conv2D** instance from TensorFlow which is extensively used in tasks dealing with image data, such as computer vision.

The parameters we will provide for each convolutional layer will be the following:

- **filters** → Determine the number of filters that will be initialized
- **kernel_size** → Determine the dimensions of the filter (also known as the kernel) that slides over the input feature map to produce the output feature map (in our case because those are colorful images it will be 3 dimensions).
- **activation** → Determine the activation function that will be executed by the layer.

CNN - Python Example

In addition to the **Conv2D** layer we will add a pooling layer using the matched **MaxPool2D** instance.

The parameters we will provide for each pooling layer will be the following:

- **pool_zise** → Determine the dimensions of the pooling window (in our case 2X2 window)
- **strides** → Determines how much the window moves after pooling each segment.
  If `strides=(2, 2)`, then the window will move 2 pixels in both the horizontal and vertical directions after each pooling operation.
- **padding** → Determines whether or not to add padding around the input feature map.
  'valid' means that no padding is added. The pooling window will only slide within the bounds of the input feature map (‘valid’ is also the default padding value).
  'same' means that paddings will be added such that the output feature map has the same dimensions as the input feature map when strides are 1.

CNN - Python Example

Now that we understand the convolutional layer and the different parameters let's actually add those layer to our code.

```
[13] cnn = tf.keras.models.Sequential()

     ## First convolutional layer
     cnn.add(tf.keras.layers.Conv2D(filters=32, kernel_size=3, activation='relu', input_shape=[64, 64, 3]))

     cnn.add(tf.keras.layers.MaxPool2D(pool_size=2, strides=2, padding='valid'))

     ## Second convolutional layer
     cnn.add(tf.keras.layers.Conv2D(filters=32, kernel_size=3, activation='relu'))

     cnn.add(tf.keras.layers.MaxPool2D(pool_size=2, strides=2, padding='valid'))
```

We initialize the CNN model instance

The first convolutional layer has the original images as input so we need to provide the input shape which is a 3 dimensional array with 64 X 64 X3 size

The second layer not getting original input so we don't need to pass the input_shape parameter

Now we can move to the flattening layer and add it to our CNN as well:

```
[14] ## Flatten layer
     cnn.add(tf.keras.layers.Flatten())
```

CNN - Python Example

Finally, we can add the fully connected ANN layer which will have a standard of 128 neurons in it's hidden layer and LeRU activation function.

In the output layer we will have 1 neuron (binary classification problem) with 'sigmoid' activation function.

```
[15] ## Fully conection layer
     cnn.add(tf.keras.layers.Dense(units=128, activation='relu'))

     cnn.add(tf.keras.layers.Dense(units=1, activation='sigmoid'))
```

Now our CNN model is completed and we can move to the training process.

Here we need to determine the optimizer, the loss function and metrics that we want to see during the training. In addition we need to specify the number of epochs (iterations).

Remember more epochs means more iteration on the entire dataset and more backpropagation process.

CNN - Python Example

Let's apply the code for the training process and activate the model training.

```
cnn.compile(optimizer='adam', loss='binary_crossentropy', metrics=['accuracy'])
try:
    cnn.fit(train_set, validation_data=test_set, epochs=25)

except Exception as e:
    print(e)
```

```
Epoch 1/25
250/250 [==============================] - 24s 89ms/step - loss: 0.6732 - accuracy: 0.5732 - val_loss: 0.6193 - val_
Epoch 2/25
250/250 [==============================] - 26s 103ms/step - loss: 0.6157 - accuracy: 0.6630 - val_loss: 0.6328 - val
Epoch 3/25
250/250 [==============================] - 23s 94ms/step - loss: 0.5718 - accuracy: 0.6998 - val_loss: 0.6062 - val_
Epoch 4/25
250/250 [==============================] - 22s 86ms/step - loss: 0.5411 - accuracy: 0.7251 - val_loss: 0.5441 - val_
Epoch 5/25
```

The 250 represent the number of batches each iteration will run. Because we chose 32 images in a batch and in our training dataset we have 8000 images thats mean that each iteration will run on
8000 / 32 = 250 batches

The model is running total of 25 iterations (according to the epochs parameter)

During each iteration the model report the accuracy because this is the metric we specify in the metrics parameter

CNN - Python Example

We can see that our model is able to increase its accuracy during each iteration and the final accuracy in the 25 iteration is a little above 90% which consider as a very good accuracy result in image recognition.

```
250/250 [==============================] - 23s 92ms/step - loss: 0.2898 - accuracy: 0.8733 - val_loss: 0.4949 - val_
Epoch 22/25
250/250 [==============================] - 24s 96ms/step - loss: 0.2779 - accuracy: 0.8840 - val_loss: 0.4908 - val_
Epoch 23/25
250/250 [==============================] - 24s 96ms/step - loss: 0.2579 - accuracy: 0.8914 - val_loss: 0.5424 - val_
Epoch 24/25
250/250 [==============================] - 23s 93ms/step - loss: 0.2434 - accuracy: 0.8994 - val_loss: 0.5277 - val_
Epoch 25/25
250/250 [==============================] - 22s 87ms/step - loss: 0.2355 - accuracy: 0.9025 - val_loss: 0.5399 - val_
```

The last phase is to test our model on truly unknown images and see if it's prediction is correct.

Because the model is a binary model it will provide as output the probability of the input image to be classified as 1 (in our case - Dog).

CNN - Python Example

When applying a new image we first need to resize it to the size as our model trained (64 X 64).

Next we need to extract the image as 3D array, normalized it and insert it as a dataset (array of 1 image).

```
from keras.preprocessing import image

test_image = image.load_img(
    'dataset/CNN_dog_cat_dataset/dataset/single_prediction/cat_or_dog_1.jpg',
    target_size=(64, 64)
)
test_image = image.img_to_array(test_image)
test_image = test_image/255.0
test_image = np.expand_dims(test_image, axis=0)

result = cnn.predict(test_image)
print(result)

if result[0][0] > 0.5:
    prediction = 'dog'
else:
    prediction = 'cat'

print(prediction)
```

```
1/1 [==============================] - 0s 58ms/step
[[0.9999994]]
dog
```

Extract the image as an array

Normalized the image array

Convert the image array to an array of 1 image by adding additional dimension to the start of the image array

The model predicted that the first image is 1 (a dog) with 0.99 probability