**What is a CNN?**

A Convolutional Neural Network (CNN or ConvNet) is a deep learning algorithm specifically designed for any task where object recognition is crucial such as image classification, detection, and segmentation.  Many real-life applications, such as self-driving cars, surveillance cameras, and more, use CNNs.

**The importance of CNNs**

These are several reasons why CNNs are important, as highlighted below:

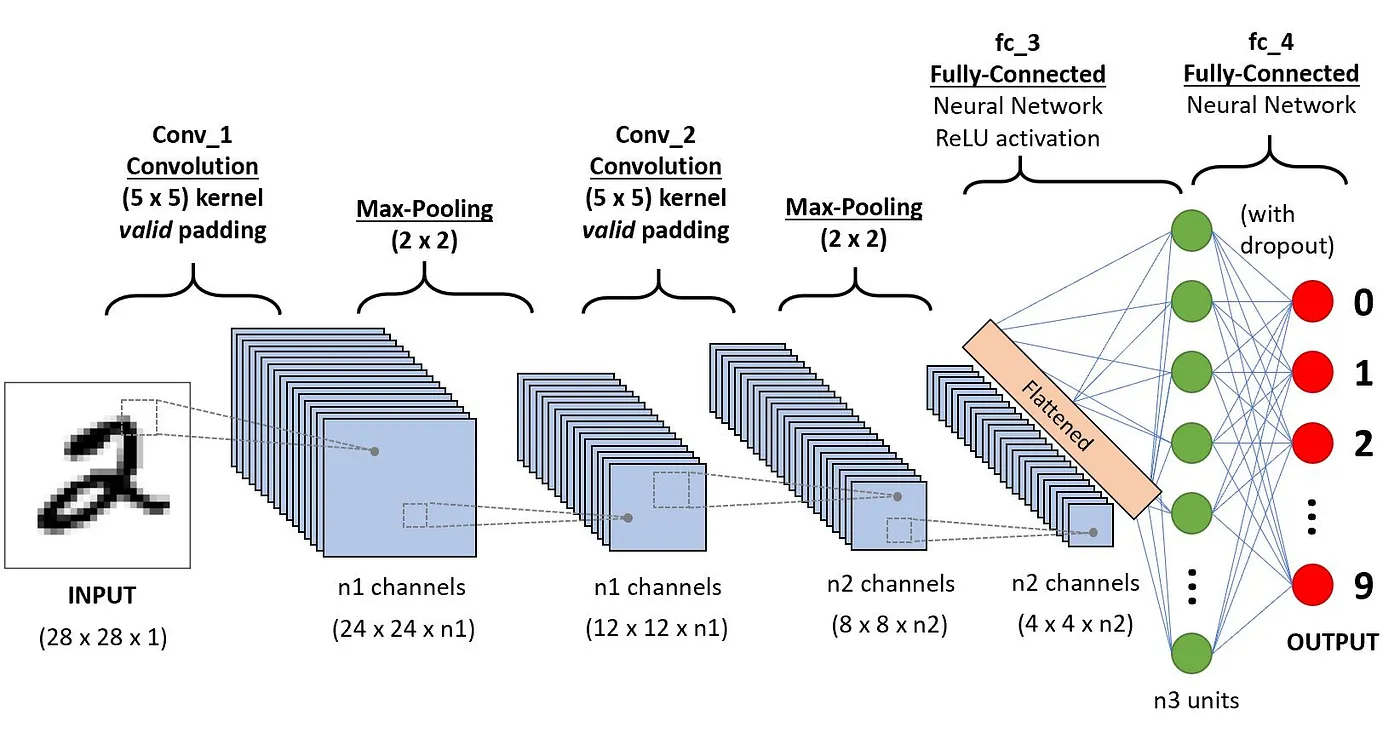
* Unlike traditional machine learning models like SVM and decision trees that require manual feature extractions, CNNs can perform automatic feature extraction at scale, making them efficient.
* The convolutions layers make CNNs translation invariant, meaning they can recognize patterns from data and extract features regardless of their position, whether the image is rotated, scaled, or shifted.
* Multiple pre-trained CNN models such as VGG-16, ResNet50, Inceptionv3, and EfficientNet are proved to have reached state-of-the-art results and can be fine-tuned on news tasks using a relatively small amount of data.
* CNNs can also be used for non-image classification problems and are not limited to natural language processing, time series analysis, and speech recognition.

**Architecture of a CNN**

CNNs’ architecture tries to mimic the structure of neurons in the human visual system composed of multiple layers, where each one is responsible for detecting a specific feature in the data.  As illustrated in the image below, the typical CNN is made of a combination of four main layers:

* Convolutional layers
* Rectified Linear Unit (ReLU for short)
* Pooling layers
* Fully connected layers

Let’s understand how each of these layers works using the following example of classification of the handwritten digit.

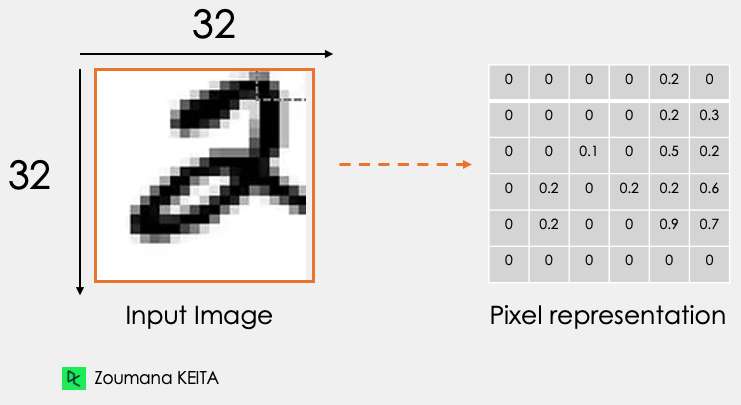


**Convolution layers**

This is the first building block of a CNN. As the name suggests, the main mathematical task performed is called convolution, which is the application of a sliding window function to a matrix of pixels representing an image. The sliding function applied to the matrix is called kernel or filter, and both can be used interchangeably.

In the convolution layer, several filters of equal size are applied, and each filter is used to recognize a specific pattern from the image, such as the curving of the digits, the edges, the whole shape of the digits, and more.

Let’s consider this 32x32 grayscale image of a handwritten digit. The values in the matrix are given for illustration purposes.



Also, let’s consider the kernel used for the convolution. It is a matrix with a dimension of 3x3. The weights of each element of the kernel is represented in the grid. Zero weights are represented in the black grids and ones in the white grid.

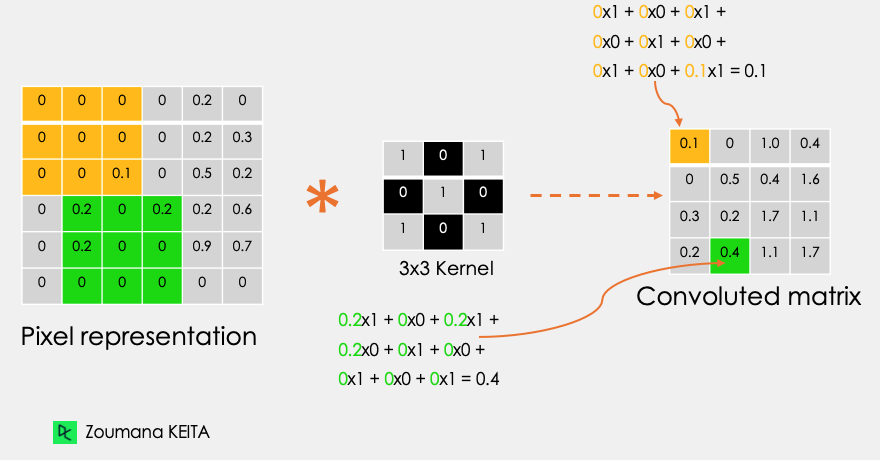
**Do we have to manually find these weights?**

In real life, the weights of the kernels are determined during the training process of the neural network.

Using these two matrices, we can perform the convolution operation by taking applying the dot product, and work as follows:

1. Apply the kernel matrix from the top-left corner to the right.
2. Perform element-wise multiplication.
3. Sum the values of the products.
4. The resulting value corresponds to the first value (top-left corner) in the convoluted matrix.
5. Move the kernel down with respect to the size of the sliding window.
6. Repeat from step 1 to 5 until the image matrix is fully covered.

The dimension of the convoluted matrix depends on the size of the sliding window. The higher the sliding window, the smaller the dimension.



Another name associated with the kernel in the literature is feature detector because the weights can be fine-tuned to detect specific features in the input image.

For instance:

* Averaging neighboring pixels kernel can be used to blur the input image.
* Subtracting neighboring kernel is used to perform edge detection.

The more convolution layers the network has, the better the layer is at detecting more abstract features.

**Activation function**

A ReLU activation function is applied after each convolution operation. This function helps the network learn non-linear relationships between the features in the image, hence making the network more robust for identifying different patterns. It also helps to mitigate the vanishing gradient problems.

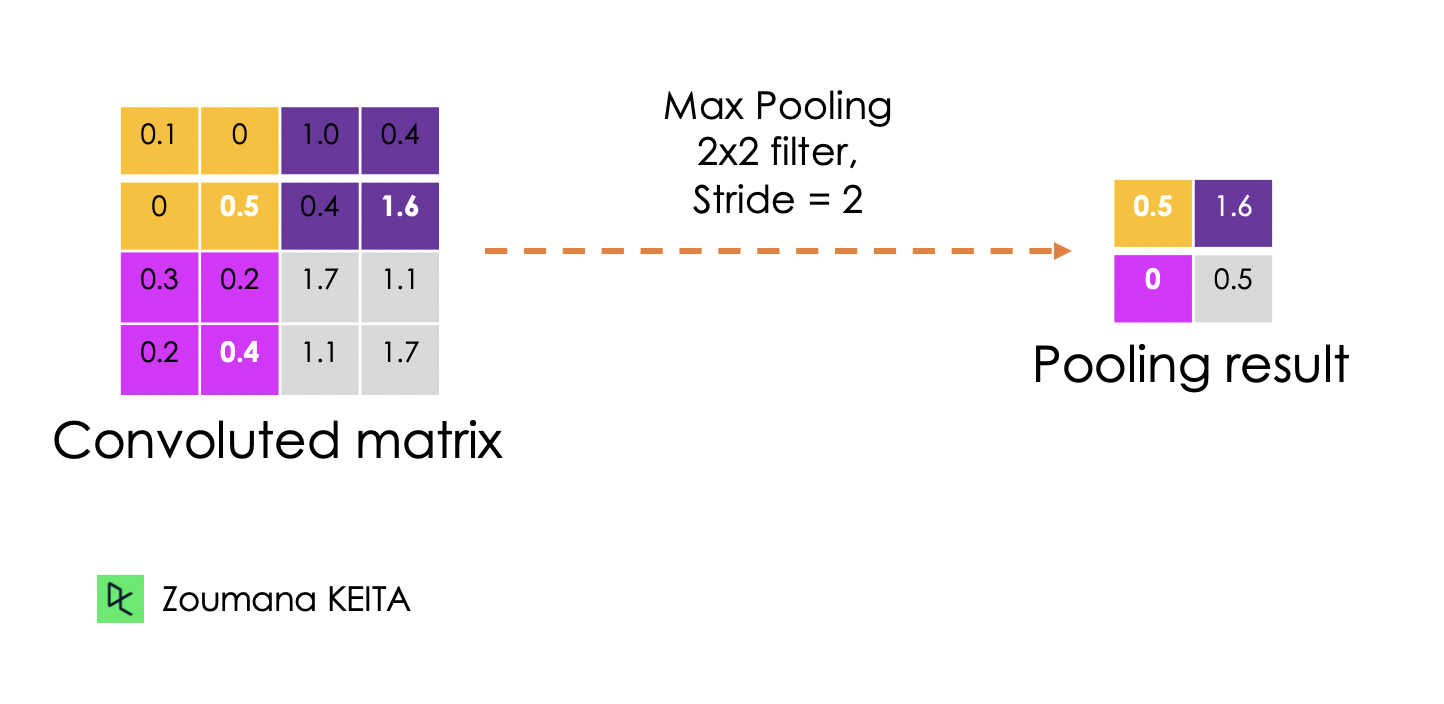
**Pooling layer**

The goal of the pooling layer is to pull the most significant features from the convoluted matrix. This is done by  applying some aggregation operations, which reduces the dimension of the feature map (convoluted matrix), hence reducing the memory used while training the network.  Pooling is also relevant for mitigating overfitting.

The most common aggregation functions that can be applied are:

* Max pooling which is the maximum value of the feature map
* Sum pooling corresponds to the sum of all the values of the feature map
* Average pooling is the average of all the values.

Below is an illustration of each of the previous example:



Also, the dimension of the feature map becomes smaller as the polling function is applied.

The last pooling layer flattens its feature map so that it can be processed by the fully connected layer.

**Fully connected layers**

These layers are in the last layer of the convolutional neural network,  and  their inputs correspond to the flattened one-dimensional matrix generated by the last pooling layer. ReLU activations functions are applied to them for non-linearity.

Finally, a softmax prediction layer is used to generate probability values for each of the possible output labels, and the final label predicted is the one with the highest probability score.

**Dropout**

Dropout is a regularization technic applied to improve the generalization capability of the neural networks with a large number of parameters. It consists of randomly dropping some neurons during the training process, which forces the remaining neurons to learn new features from the input data.

Since the technical implementation will be performed using TensorFlow 2, the next section aims to provide a complete overview of different components of this framework to efficiently build deep learning models.

**What is the TensorFlow Framework?**

Google developed TensorFlow in November 2015. They define it to be an open-source machine learning framework for everyone for several reasons.

* **Open-source**: released under the Apache 2.0 open-source license. This allows researchers, organizations, and developers to make their contribution to the library by building upon it without any restrictions.
* **Machine learning framework**: meaning that it has a set of libraries and tools that support the building process of machine learning models.
* **For everyone**: Using TensorFlow makes the implementation of machine learning models easier through common programming languages like Python. Furthermore, built-in libraries such as Keras make it even easier to create robust deep learning models.

All these functionalities make Tensorflow a good candidate for building neural networks.

Furthermore, installing Tensorflow 2 is straightforward and can be performed as follows using the Python package manager ***pip*** as explained in the [official documentation](https://www.tensorflow.org/install).

After the installation, we can see that the version being used is the 2.9.1

import tensorflow as tf

print("TensorFlow version:", tf.\_\_version\_\_)

OpenAI

This code imports the TensorFlow library in Python and prints the version of TensorFlow that is currently installed.

• The **import** statement brings in the TensorFlow library, which is a popular open-source machine learning framework.

• The **print** statement then outputs the version of TensorFlow that is installed on the system.

• The **tf.\_\_version\_\_** attribute retrieves the version number of the TensorFlow library.

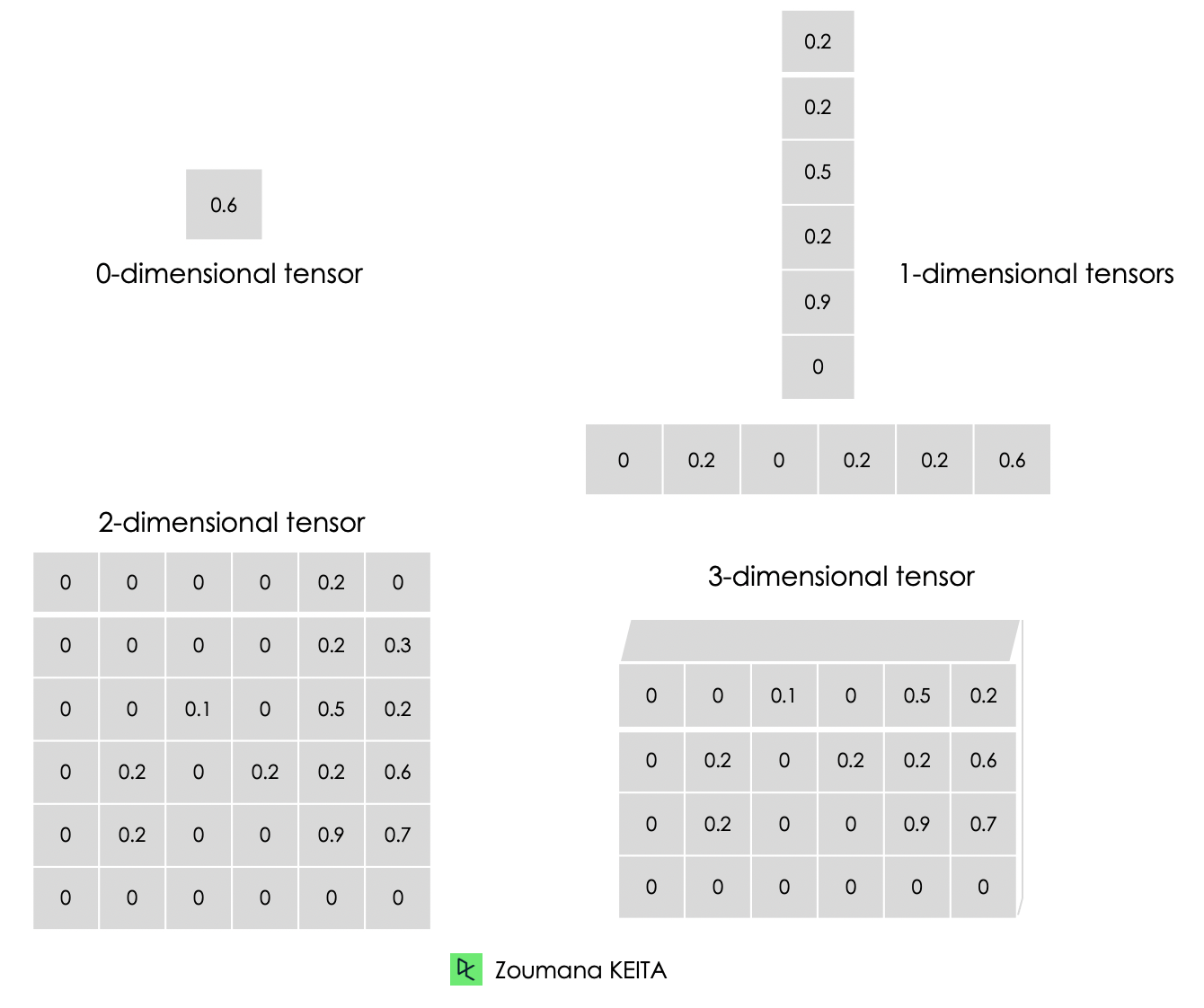
Was this helpful? Yes No

Now, let’s further explore the main components for creating those networks.

**What are Tensors?**

We mainly deal with high-dimensional data when building machine learning and deep learning models. Tensors are multi-dimensional arrays with a uniform type used to represent different features of the data.

Below is the graphical representation of the different types of dimensions of tensors.



* A 0-dimensional tensor contains a single value.
* A 1-dimensional tensor, also known as “rank-1” tensor is list of values.
* A 2-dimensional tensor is a “rank-2” tensor.
* Finally, we can have a N-dimensional tensor, where N represents the number of dimensions within the tensor. In the previous cases, N is respectively 0, 1 and 2.

Below is an illustration of a zero to a 3-dimensional tensor. Each tensor is created using the constant() function from TensorFlow.

# Zero dimensional tensor

zero\_dim\_tensor = tf.constant(20)

print(zero\_dim\_tensor)

# One dimensional tensor

one\_dim\_tensor = tf.constant([12, 20, 53, 26, 11, 56])

print(one\_dim\_tensor)

# Two dimensional tensor

two\_dim\_array = [[3, 6, 7, 5],

[9, 2, 3, 4],

[7, 1, 10,6],

[0, 8, 11,2]]

two\_dim\_tensor = tf.constant(two\_dim\_array)

print(two\_dim\_tensor)

OpenAI

This code uses the TensorFlow library to create and print tensors of different dimensions.

• First, a zero-dimensional tensor is created using the **tf.constant()** function and assigned to the variable **zero\_dim\_tensor**.

• The value of this tensor is set to 20.

• Next, a one-dimensional tensor is created using the same function and assigned to the variable **one\_dim\_tensor**.

• The values of this tensor are specified as a list of integers.

• Finally, a two-dimensional tensor is created using the **tf.constant()** function and assigned to the variable **two\_dim\_tensor**.

• The values of this tensor are specified as a nested list of integers.

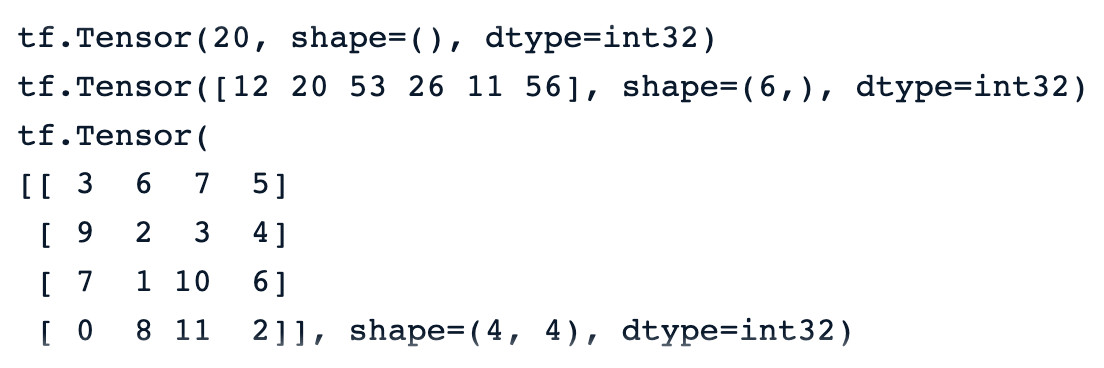
• Each tensor is then printed using the **print()** function.

• Overall, this code demonstrates how to create tensors of different dimensions using TensorFlow.

Was this helpful? Yes No

A successful execution of the previous code should generate the outputs below, and we can notice the keyword “tf.Tensor” to mean that the result is a tensor. It has three parameters:

* The actual value of the tensor.
* The **shape()** of the tensor, which is 0, 6 by 1, and 4 by 4, respectively for the first, second, and third tensors.
* The data type represented by the **dtype** attribute, and all the tensors are int32.



Our **[Tensorflow Tutorial for Beginners](https://www.datacamp.com/tutorial/tensorflow-tutorial" \t "_blank)** provides a complete overview of TensorFlow and teaches how to build and train models.

**Tensors vs Matrices: Differences**

Many people confuse tensors with matrices. Even though these two objects look similar, they have completely different properties. This section provides a better understanding of the difference between matrices and tensors.

* We can think of a matrice as a tensor with only two dimensions.
* Tensors, on the other hand, is a more general format that can have any number of dimensions.

As opposed to matrices, tensors are more suitable for deep learning problems for the following reasons:

* They can deal with any number of dimensions, which makes them a better fit for multi-dimensional data.
* Tensors’ ability to be compatible with a wide range of data types, shapes, and dimensions makes them more versatile than matrices.
* Tensorflow provides GPU and TPU support to speed up computations. Using tensors, machine learning engineers can automatically take advantage of these benefits.
* Tensors natively support broadcasting, which consists of making arithmetic operations between tensors of different shapes, which is not always possible when dealing with matrices.

**TensorFlow: Constants, Variables, and Placeholders**

Constants are not the only types of tensors. There are also variables and placeholders, which are all building blocks of a computational graph.

A computational graph is basically and a representation of a sequence of operations and the flow of data between them.

Now, let’s understand the difference between these types of tensors.

**Constants**

Constants are tensors whose values do not change during the execution of the computational graph. They are created using the **tf.constant()** function and are mainly used to store fixed parameters that do not require any change during the model training.

**Variables**

Variables are tensors whose value can be changed during the execution of the computational graph and they are created using the **tf.Variable()** function. For instance, in the case of neural networks, weights, and biases can be defined as variables since they need to be updated during the training process.

**Placeholders**

These were used in the first version of Tensorflow as empty containers that do not have specific values. They are just used to reverse a spot for data to be used in the future. This gives the users the freedom to use different datasets and batch sizes during model training and validation.

In Tensorflow version 2, placeholders have been replaced by the**tf.function()** function, which is a more Pythonic and dynamic approach to feeding data into the computational graph.