Chapter 2

TensorFlow 2

TensorFlow is introduced as an end-to-end machine learning framework, widely used for deep learning but also supporting probabilistic modeling, computer graphics, and pre-trained models through TensorFlow Hub. It provides TensorBoard for monitoring and visualization, which makes it a complete ecosystem for research and deployment. Figures 1 summarizes these extended capabilities, showing that TensorFlow is not limited to neural networks alone.

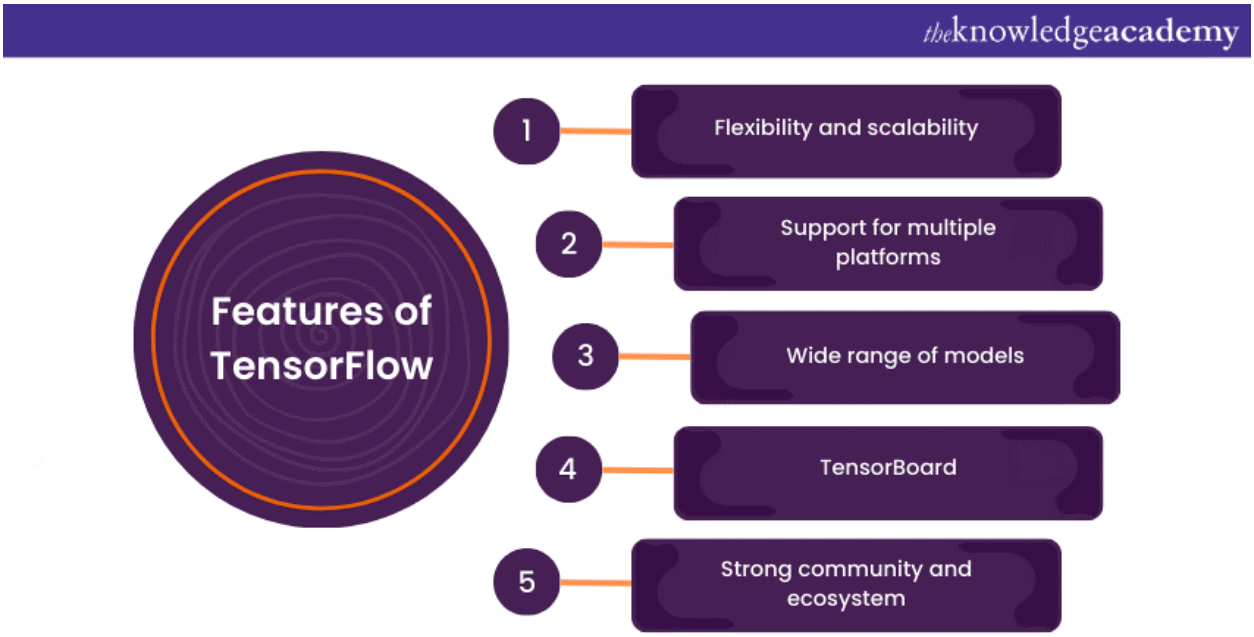

Figure 1 Different features of TensorFlow

TensorFlow 2 represents a major transformation from the earlier 1.x versions, addressing many of the difficulties that made the original framework powerful yet challenging to use. TensorFlow 1.x required developers to build static computation graphs before execution, which created a divide between defining a model and running it. While this approach offered efficiency, it also introduced confusion, boilerplate code, and a steep learning curve. TensorFlow 2 simplifies the process by adopting eager execution by default, integrating Keras as the official high-level API, and providing a more Pythonic, intuitive interface for model development.

To illustrate the improvements in TensorFlow 2, it begins with a multilayer perceptron (MLP), a network with an input layer, hidden layer, and output layer. Computation in the hidden layer uses a sigmoid activation, while the output is normalized using softmax to form probabilities. Figure 2 shows this structure, emphasizing weights, biases, and the transformation from inputs to outputs. In TensorFlow 2, these computations are made easier through eager execution, where declaration

and execution occur simultaneously. This contrasts sharply with TensorFlow 1, where the graph had to be explicitly defined before execution. Table 1 highlights these differences, noting that TensorFlow 2 is more user-friendly, debuggable, and better suited for object-oriented programming.

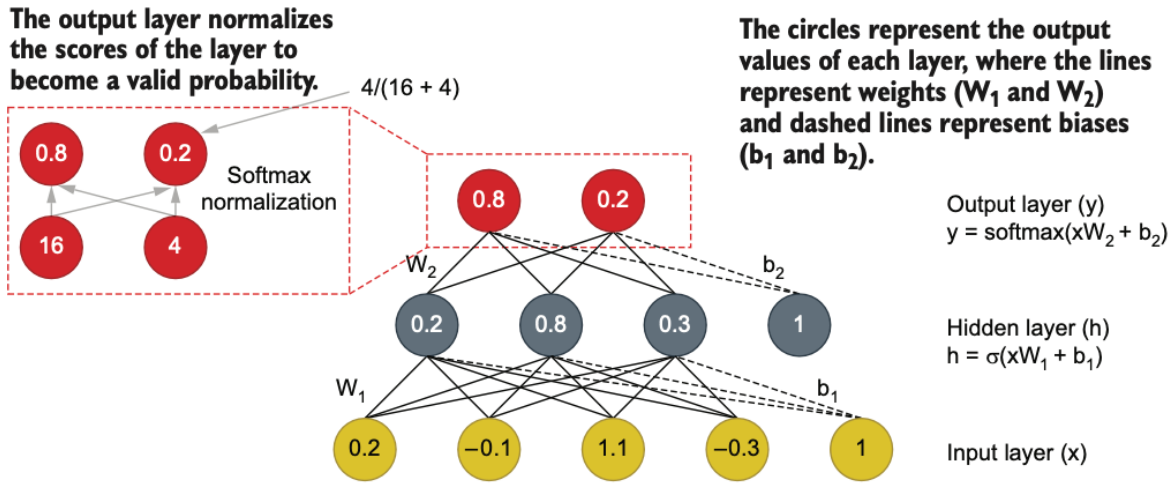

Figure 2 The structure of multilayer perceptron (MLP)

Table 1 Differences between TensorFlow 1 and TensorFlow 2

TensorFlow 1	TensorFlow 2
Does not use eager execution by default	Uses eager execution by default
Uses symbolic placeholders to represent inputs to the graph	Directly feeds actual data
Difficult to debug as results are not evaluated imperatively	Easy to debug as operations are evaluated imperatively
Needs to explicitly and manually create the data-flow graph	Has AutoGraph functionality, which traces Tensor-Flow operations and creates the graph automatically
Does not encourage object-oriented programming, as it forces you to define the computational graph in advance	Encourages object-oriented programming

Code reproduction of MLP:

```
import numpy as np
```

```
import tensorflow as tf
x = np.random.normal(size=[1, 4]).astype('float32')
init = tf.keras.initializers.RandomNormal()

w1 = tf.Variable(init(shape=[4, 3]))
b1 = tf.Variable(init(shape=[1, 3]))

w2 = tf.Variable(init(shape=[3, 2]))
b2 = tf.Variable(init(shape=[1, 2]))

@tf.function
def forward(x, W, b, act):
    return act(tf.matmul(x, W) + b)

h = forward(x, w1, b1, tf.nn.sigmoid)
y = forward(h, w2, b2, tf.nn.softmax)

print(y)      # e.g. tf.Tensor([[0.49  0.51]], shape=(1,2),
dtype=float32)
```

A tf.Variable is mutable and ideal for model parameters that change during training, while a tf.Tensor is immutable and represents static outputs. Operations (tf.Operation) such as tf.matmul or tf.add form the computational backbone. Table 2 connects these primitives to the earlier MLP example. A key distinction is that tf.Variable can be reassigned, while tf.Tensor cannot, making them complementary structures in model building.

Table 2 tf.Variable, tf.Tensor, and tf.Operation entities from the MLP example

Element	Example
Tf.Variable	W1, b1, w2, and b2
Tf.Tensor	H and y
Tf.Operation	Tf.matmul

Using a sample image Figure 3, the process of converting an RGB image to grayscale is shown via matrix multiplication, an operation central not just to computer vision but also to fully connected networks. This is followed by convolution, where kernels slide over the image to detect features such as edges. Figure 4 shows the result of applying a Laplacian filter for edge detection. Finally, pooling operations are introduced to reduce dimensionality and retain key features, with Figure 5 comparing results of average pooling and max pooling after edge detection.

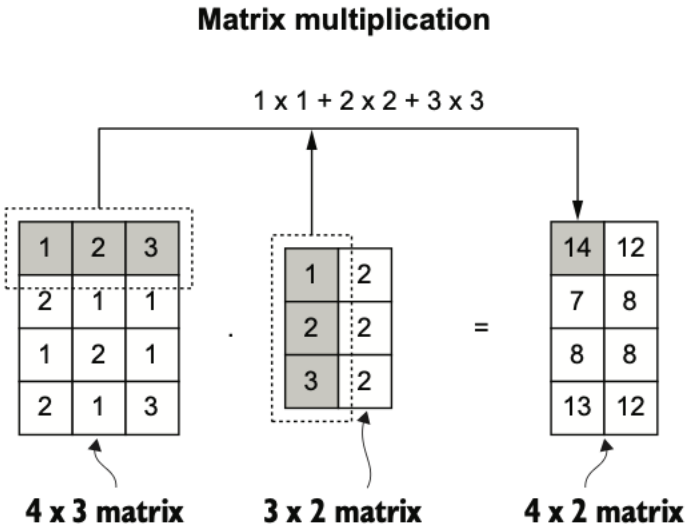

Figure 3 Sample of matrix multiplication

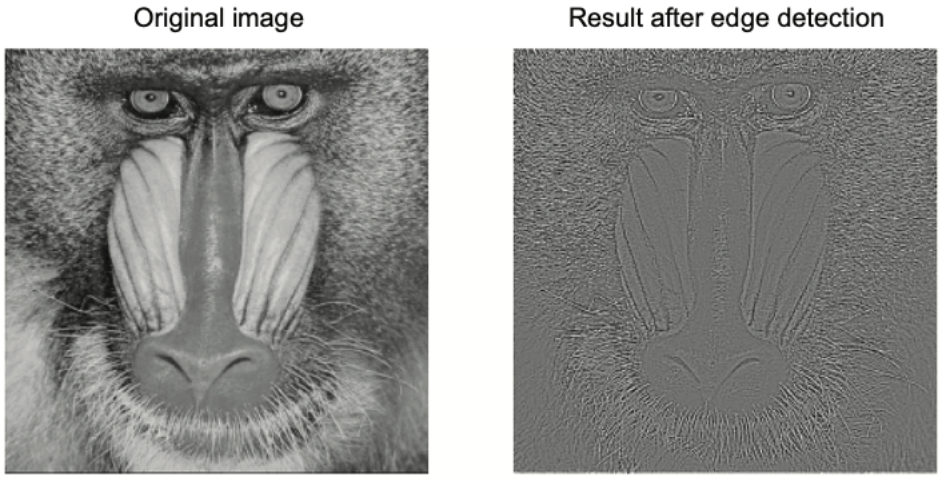

Figure 4 Original black-and-white image versus result of edge detection

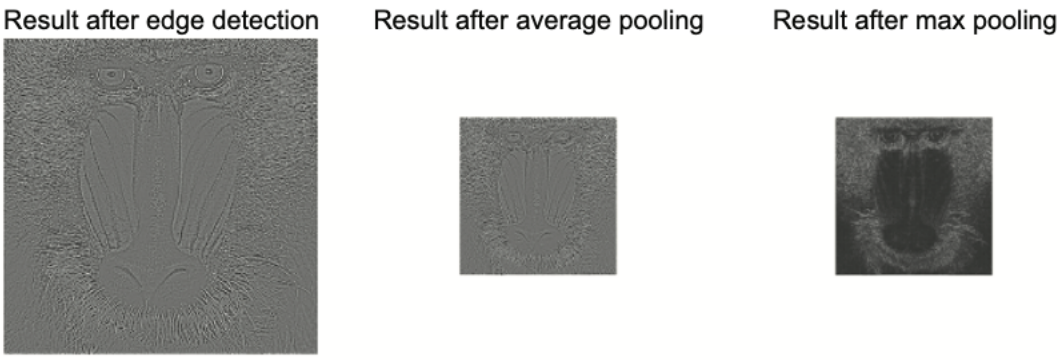

Figure 5 Result after edge detection versus result after average or max polling

In summary, TensorFlow 2 simplifies development by enabling eager execution, supporting AutoGraph for graph optimization, and providing clear primitives (Variable, Tensor, Operation) that underpin all computations. Its operations, such as matrix multiplication, convolution, and pooling, form the foundation of deep neural networks.