TensorFlow & Keras

Table of Contents

TensorFlow....1
Tensor....2
Computation Graph (in simple terms):....3
Eager execution....4
Key Features of Eager Execution:....5
Comparison with Graph Mode:....5
Example from the Image:....5
What is Keras?....8
Key Features of Keras:....8
Key Benefits of tf.keras:....8
Use Case Examples of Keras (tf.keras):....8
Summary:....9
1. Why 256 and 128 neurons in the hidden layers?....13
2. Why batch size of 32?....13
3. Why 30 epochs?....13
Summary:....14

TensorFlow

TensorFlow is an **open-source machine learning framework** developed by **Google** that is widely used for **deep learning** and other machine learning tasks. It helps developers build, train, and deploy machine learning models in a scalable and efficient way.

Here's why TensorFlow stands out:

1. **Deep Learning Powerhouse**: It is especially popular for deep learning, which involves training neural networks for tasks like image recognition, natural language processing, and speech recognition.
2. **Cross-Platform & Portability**: TensorFlow models can run on a wide range of devices:
    - **CPUs** (for general-purpose processing),
    - **GPUs** (for faster training using parallel processing),
    - **TPUs** (Tensor Processing Units, Google's custom hardware for machine learning).
3. **Flexibility**:
    - You can use it to create simple experiments or complex machine learning systems.
    - It supports low-level customizations (e.g., building a neural network from scratch) or high-level APIs like Keras for faster prototyping.
4. **Scalability**:
    - TensorFlow is designed to handle everything from small experiments on a laptop to large-scale production systems that process terabytes of data in real time.
5. **Community & Ecosystem**:
    - TensorFlow is **open-source**, meaning anyone can use and contribute to it, making it

widely adopted and well-documented.
- It has a rich ecosystem, including libraries like **TensorFlow Lite** (for mobile/embedded devices) and **TensorFlow.js** (for web apps).

6. **Broad Application Areas**:
    - TensorFlow is not just limited to AI researchers. Developers in fields like healthcare, finance, robotics, and more use it to solve real-world problems.

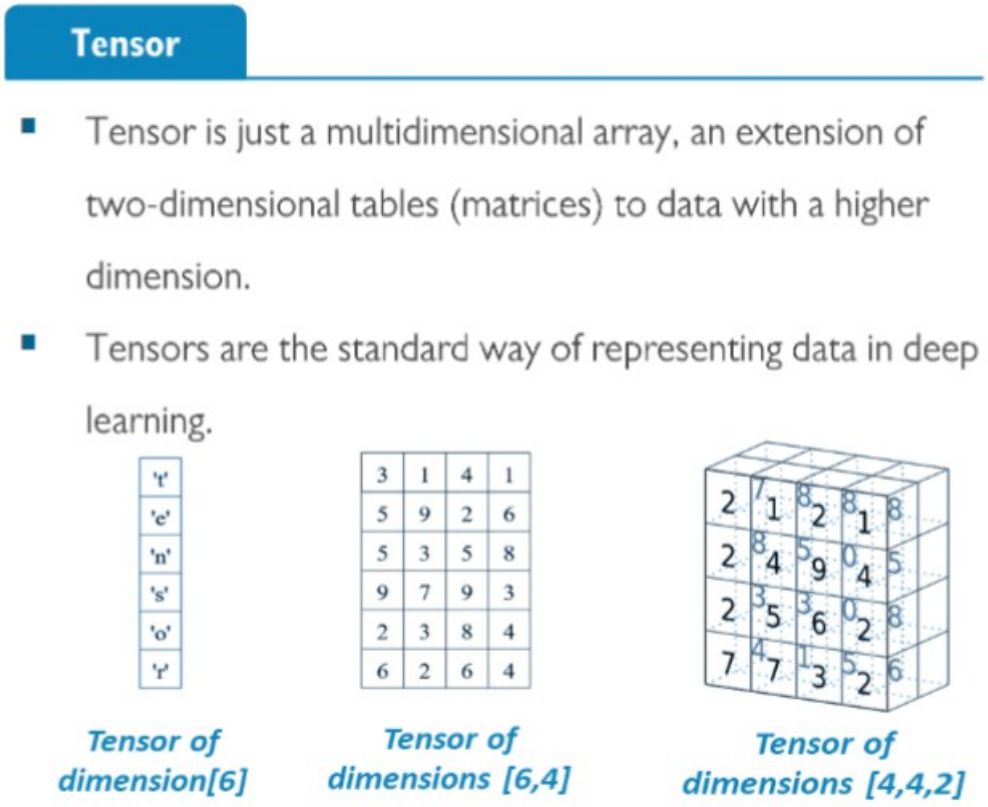
In essence, TensorFlow is a **versatile, portable, and scalable framework** that empowers developers to build intelligent systems leveraging the power of machine learning and deep learning. Whether you're training a model on your laptop or deploying it in a distributed cloud environment, TensorFlow has you covered!

Tensor

A **tensor** is just a fancy way of saying "data stored in multiple dimensions." It's like a container for numbers.

- A **scalar** (single number) is like a 0D tensor.
- A **vector** (list of numbers, like `[1, 2, 3]`) is a 1D tensor.
- A **matrix** (grid of numbers, like `[[1, 2], [3, 4]]`) is a 2D tensor.
- Higher-dimensional tensors (like a cube of numbers or beyond) go to 3D, 4D, and so on.

In machine learning, tensors hold data such as input images, features, weights, and outputs.



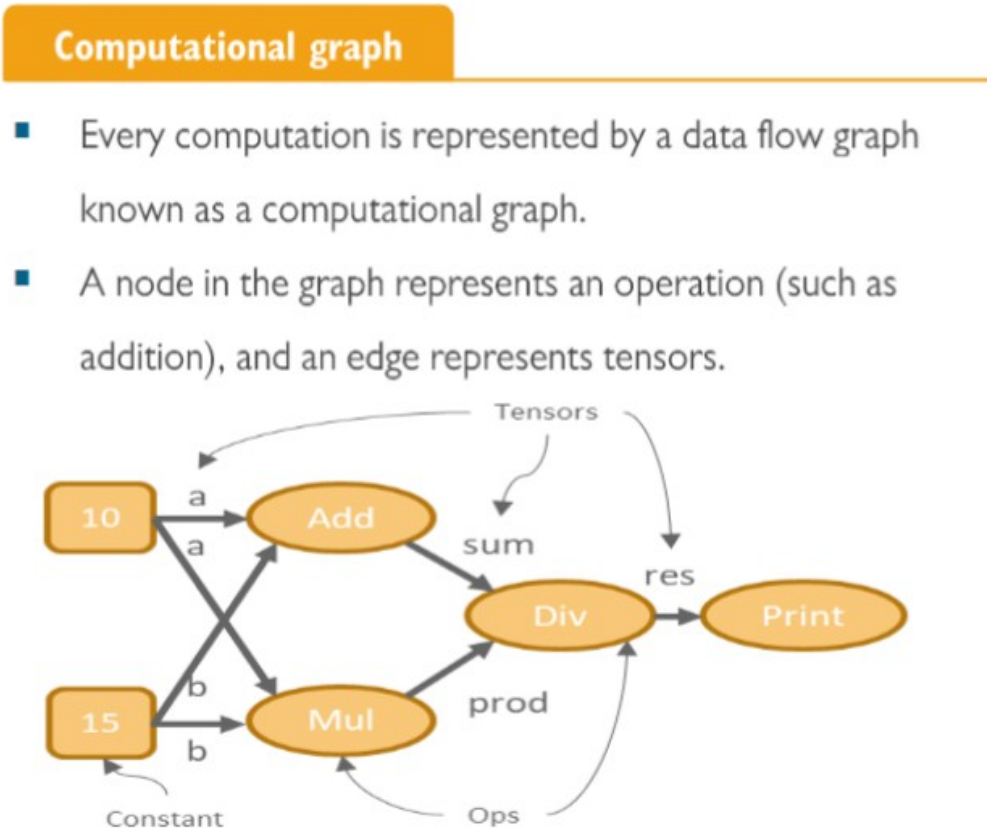
Computation Graph (in simple terms):

A **computation graph** is like a flowchart for math operations. It shows how data (tensors) move through a series of steps to produce a result.

For example:

1. Start with some inputs (like numbers or tensors).
2. Perform operations (like adding, multiplying, or applying functions).
3. Produce a final output.

Each "node" in the graph is an operation (e.g., addition, multiplication), and the "edges" show how tensors flow from one step to the next.



Eager execution

Eager execution in TensorFlow is a way of running code that is more intuitive and Pythonic. Unlike the traditional "graph mode," where operations are first built into a computational graph and then executed, **eager execution** runs operations **immediately** as they are called. This makes it easier to debug, experiment, and develop machine learning models interactively.

Key Features of Eager Execution:

1. **Immediate Execution**:
    - Each operation is executed as soon as it is called, without waiting to build a graph. This is similar to how Python code normally runs.
    - It's great for debugging and experimentation because you can see the results immediately.
2. **Intuitive Programming**:
    - Writing TensorFlow code with eager execution feels like writing regular Python code.
    - No need to think about "sessions" or building computation graphs explicitly.
3. **Default in TensorFlow 2.x**:
    - TensorFlow 2.x automatically uses eager execution by default, making it more beginner-friendly.
4. **Better Integration**:
    - Eager execution works well with Python debugging tools and integrates seamlessly with libraries like NumPy.

Comparison with Graph Mode:

- **Graph Mode**:
    - Operations are added to a computation graph first and then executed in a session.
    - Efficient for large-scale and production-grade tasks since the graph is optimized before execution.
- **Eager Execution**:
    - Direct execution of operations without pre-constructing a graph.
    - Simpler and more interactive for smaller tasks or when prototyping.

Example from the Image:

In the example:

- Eager execution allows TensorFlow operations like `tf.reduce_mean` and `tf.reduce_sum` to run directly, and the results are printed immediately (e.g., `7.24` and `11.0`).
- It's simple and requires no session or graph creation.

TensorFlow: Eager Execution

In graph mode, a graph needs to be constructed every time to perform any operations.

Eager execution mode follows a programming paradigm where any operation can be executed immediately, just like in Python.

Code example

```
#Examples of operations using Eager Execution mode
tf.enable_eager_execution()

x = tf.Variable([1.0,3.1,7.4,11.2,13.5])
print(tf.reduce_mean(input_tensor=x).numpy())

a = [1., 3., 2.]
b = [2., 1., 3.]
dot_product = tf.reduce_sum(tf.multiply(a,b))
print(dot_product.numpy())
```

Code output

```
7.2400002
11.0
```

TensorFlow 2.x sets the eager execution mode by default.

TensorFlow: Building and Running a Graph

Building a computational graph

Code example

```
import tensorflow as tf

node1 = tf.constant(3.0, tf.float32)
node2 = tf.constant(4.0)
```

Constant nodes

Build a computational graph with a series of TensorFlow operations.

Running a computational graph

Code example

```
sess = tf.Session()
print(sess.run([node1, node2]))
```

Code output

```
[3.0, 4.0]
```

To execute the graph, we must run it within a **session.**

The **session** encapsulates the control and state of the TensorFlow runtime.

TensorFlow Example

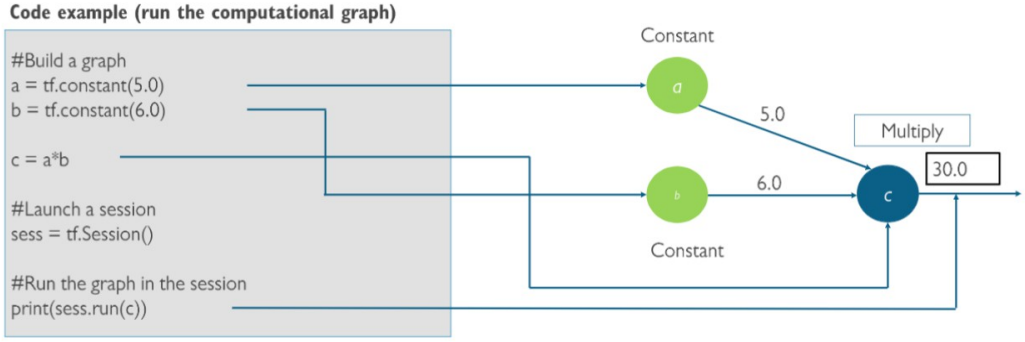
Code example (run the computational graph)

```
#Build a graph
a = tf.constant(5.0)
b = tf.constant(6.0)

c = a*b

#Launch a session
sess = tf.Session()

#Run the graph in the session
print(sess.run(c))
```



Code output

```
30.0
```

What is Keras?

Keras is a **high-level API** for building and working with **neural networks**. It simplifies the process of ***creating, training, and deploying machine learning models***, making it beginner-friendly while still offering flexibility for advanced users. Keras emphasizes **ease of use, modularity, and user-friendly design**.

Key Features of Keras:

1. **Ease of Use**:
    - Designed to make deep learning accessible, even for beginners.
    - Provides a simple and intuitive interface for defining models, layers, and training steps.
2. **Flexibility**:
    - Highly modular: You can combine and customize neural network layers and components.
    - Advanced users can still dive into the details and build complex architectures.
3. **Beautiful Design**:
    - Focused on simplicity and readability, Keras encourages clean, organized code.
    - The API is built with a "Pythonic" approach to enhance productivity.
4. **Backend Support**:
    - Keras supports multiple backend engines for performing computations. These include:
        - **TensorFlow**
        - **Apache MXNet**
        - **Microsoft Cognitive Toolkit (CNTK)** (now deprecated)
        - **Theano** (historically used but also deprecated)
    - This flexibility allows developers to choose the backend that best fits their project needs.

Key Benefits of `tf.keras`:

1. **Seamless TensorFlow Integration**:
    - Takes full advantage of TensorFlow's features (e.g., GPU/TPU support, distributed training, etc.).
2. **Single Environment**:
    - No confusion about managing backends—it directly works with TensorFlow as the backend.
3. **Standardized Development**:
    - **TensorFlow + Keras = A single ecosystem** for research, development, and deployment.

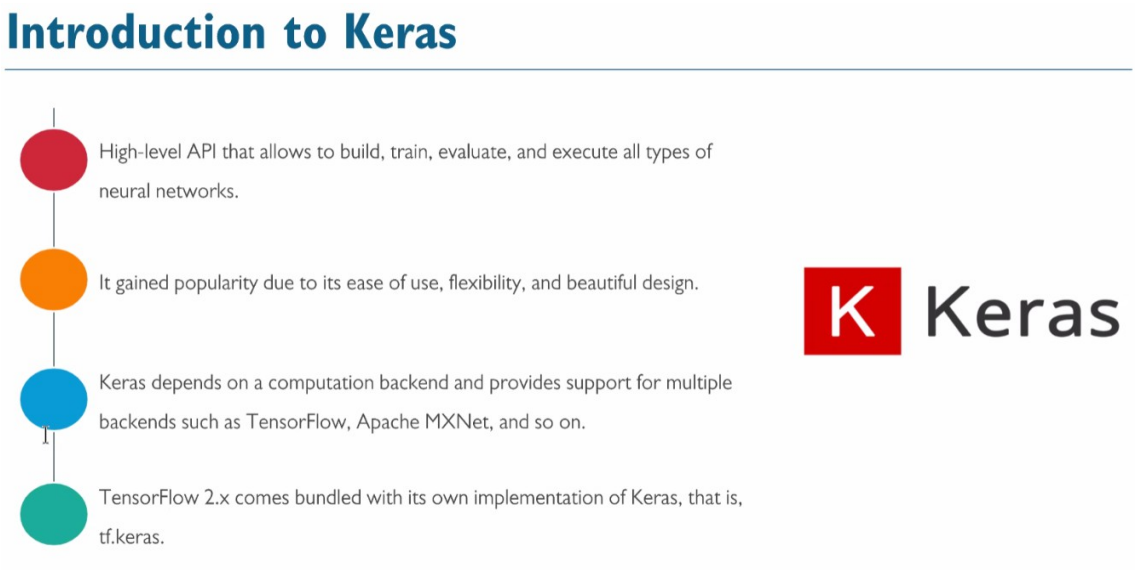
Use Case Examples of Keras (`tf.keras`):

- Quickly prototype models with high-level APIs like `Sequential` or `Functional API`.
- Build and fine-tune complex models (e.g., ResNet, LSTM, Transformers) with low-level

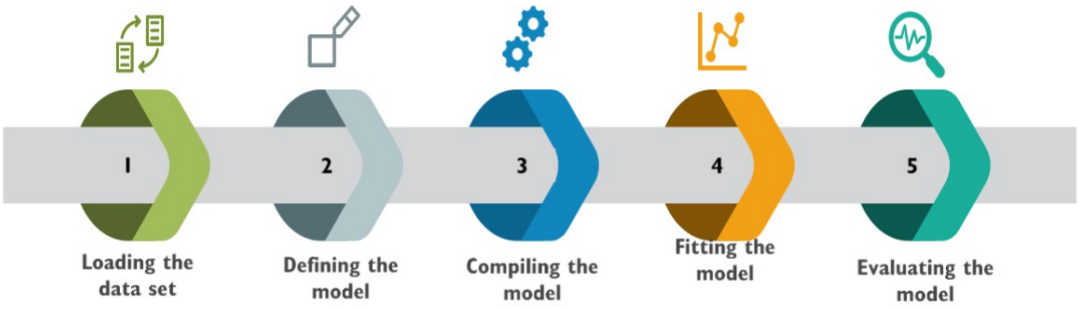
customizations.
- Deploy trained models to mobile (using TensorFlow Lite) or to production systems.

Summary:

- Keras is a **user-friendly high-level API** for deep learning.
- **Standalone Keras** supports multiple backends like TensorFlow, Apache MXNet, etc.
- **`tf.keras`** is TensorFlow's built-in implementation of Keras, making it the preferred choice for TensorFlow users in **TensorFlow 2.x and beyond**.
- Together, Keras and TensorFlow combine **simplicity**, **power**, and **scalability** in one ecosystem, enabling both beginners and advanced developers to work efficiently.



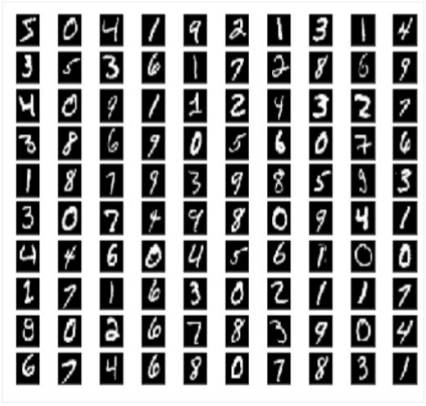
Building a Neural Network in Tensorflow 2.x



Step 1: Loading the Data Set (Overview of Data)

- Data set under consideration: MNIST — Modified National Institute of Standards and Technology data set
- Consists of 60,000 training images and 10,000 testing images of handwritten digits between 0 and 9
- Each digit represented using an array of 28 x 28 (784) greyscale pixels

Preview of the data set



Step 1: Loading the Data Set (Using Keras)

- Load the data set from sample data sets
- Create train and test sets
- Normalize the data by dividing *x* by 255.0 (maximum value that *x* can take)

Code example

```
# Import MNIST data
mnist = tf.keras.datasets.mnist

# Creating training and testing datasets
(x_train,y_train), (x_test,y_test) = mnist.load_data()

#Normalizing the datasets
x_train, x_test = tf.cast(x_train/255.0, tf.float32), tf.cast(x_test/255.0, tf.float32)
y_train, y_test = tf.cast(y_train, tf.int64),
tf.cast(y_test, tf.int64)
```

Step 2: Defining the Model

Sequential API

- Layers are stacked one above the other.
- Any number of layers can be stacked up.
- It creates the simplest type of Keras model.

Functional API

- It provides more flexibility as compared to sequential API.
- Any layer can be connected to any other layer.
- It is used for complex models.

We will use sequential API for our model.

Step 2: Defining the Model (Methods)

- Let us build a model with three layers as described in the below code:

Code example

```
# Defining a Sequential Model
model = tf.keras.models.Sequential()

#Defining layers of the model
model.add(tf.keras.layers.Flatten())
model.add(tf.keras.layers.Dense(256, activation='relu'))
model.add(tf.keras.layers.Dense(128, activation='relu'))
model.add(tf.keras.layers.Dense(10, activation='softmax'))
```

- The first layer is a flattened layer. It is used to convert each image into a 1D array.
- Define the next two layers as dense layers, which implies a fully-connected layer. The number of neurons and activation functions are specified for each layer.
- A dense output layer is added with 10 neurons, one per digit. Softmax activation is used to handle multiclass.

Step 3: Compiling the Model (Parameters)

Optimizer

- Defines the optimization algorithm to be used
- Stochastic Gradient Descent (SGD) used in this case

Loss

- For example: Cross-entropy loss or Mean Squared Error
- Function to be optimized

Metrics

- Parameter used to assess the mode
- Accuracy — used in this case

Code example

```
model.compile(optimizer='sgd', loss='sparse_categorical_crossentropy',
metrics=['accuracy'])
```

Step 4: Training the Model

Code example

```
#Training the Model
model.fit(x_train, y_train, batch_size=32, epochs=30)
```

Forward pass

Forward propagation from the input to the output layer

Backward pass

Backpropagation from the output to the input layer

Epoch

Number of times the model goes through the complete training data set

Batch size

Number of training samples used in one forward and one backward pass

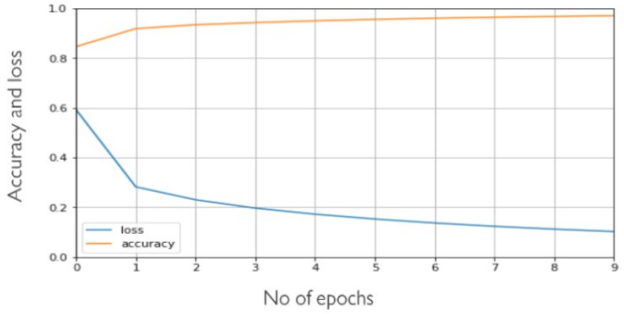
Step 5: Evaluating the Model

- After training the model, it can be evaluated on either testing or training data set.
- In this case, a testing set has been used.

Code example

```
#Evaluating the Model
model.evaluate(x_test, y_test)
```

A graph of loss and accuracy can be displayed as shown here. (Epoch is set to 10)



The choice of parameters such as the number of neurons in hidden layers, batch size, and number of epochs often involves a combination of intuition, domain knowledge, and experimentation.

1. Why 256 and 128 neurons in the hidden layers?

- **Role of hidden layers and neurons**:
  - Hidden layers capture patterns or features in the data. The number of neurons determines how much information the layer can learn.
  - More neurons allow the model to capture more complex patterns, but too many can lead to overfitting (memorizing instead of generalizing).
- **Why 256 and 128?**
  - These numbers are **heuristic choices**:
    - Powers of 2 (like 256, 128, etc.) are commonly used due to memory efficiency on hardware like GPUs.
    - 256 is large enough to capture patterns in the 784 features (pixels) from the flattened image but not excessively large to risk overfitting.
    - 128 neurons in the second hidden layer further refines the features extracted by the first layer.
  - These are **not fixed rules**—they are starting points. You could try other values (e.g., 512, 64, etc.) and tune them based on model performance.

2. Why batch size of 32?

- **What is batch size?**
  - The batch size determines how many samples are processed before updating the model's weights during training.
- **Why 32?**
  - **Trade-off between speed and accuracy**:
    - A small batch size (like 1) gives more precise updates but is computationally slow.
    - A large batch size (like 128 or more) is faster but less precise and requires more memory.
  - 32 is a **default and balanced choice**:
    - It uses less memory compared to larger sizes.
    - It often achieves good accuracy and performance for many datasets, including MNIST.
- **Experimentation**: You can try increasing or decreasing the batch size to see its effect on training speed and accuracy.

3. Why 30 epochs?

- **What is an epoch?**
  - An epoch is one full pass through the entire training dataset.

- **Why 30?**
    - **Empirical testing**:
        - The model's accuracy improves with more epochs as it learns better, but only up to a point.
        - Beyond a certain number of epochs, the model risks overfitting, where it performs well on training data but poorly on unseen data.
    - 30 epochs is a common starting point for simple datasets like MNIST.
    - You can use a tool like **early stopping** (stop training when performance stops improving) or evaluate the model's performance after each epoch to find the optimal number.

Summary:

- **Hidden layers (256, 128)**: Provide enough capacity to capture patterns in the data but are not overly complex.
- **Batch size (32)**: Balances computational efficiency and model precision.
- **Epochs (30)**: Allows sufficient learning but avoids overfitting.

These choices are not strict rules—they are **starting points** for experimentation. You can tune these parameters by:

- **Cross-validation**: Test different values to find the optimal ones.
- **Hyperparameter optimization**: Use automated tools like grid search or random search.