TensorFlow vs PyTorch – A Detailed Comparison

machinelearningplus.com/deep-learning/tensorflow1-vs-tensorflow2-vs-pytorch

Shruti Dash

Compare the popular deep learning frameworks: Tensorflow vs Pytorch. We will go into the details behind how TensorFlow 1.x, TensorFlow 2.0 and PyTorch compare against eachother. And how does keras fit in here.

Table of Contents:

1. Introduction

If you have ever come across the terms Deep learning or Neural Network, chances are you must also have heard about **TensorFlow** and **PyTorch**.

For the uninitiated, **Deep learning** is a branch of machine learning that can learn complex relationship in data and be used to solve a lot of complex problems, primarily based on artificial neural networks.

Unlike traditional machine learning, you don't have to engineer new feature variables, interaction etc that are to be considered and can scale to learn more complex relationships.

Coming to TensorFlow and PyTorch, these are two of the most popular frameworks today that are used to build and optimize a neural network. While Tensorflow is backed by Google, PyTorch is backed by Facebook. Both are actively developed and maintained.

TensorFlow now has come out with a newer TF2.0 version. What changed from the older version? and how does it compare with respect to PyTorch?

Now, Which one is better?

This is what this tutorial is all about. You will get all your your doubts resolved about the features of 2 of the most popular neural network frameworks and then can make a decision for yourself about what you would prefer!

2. TensorFlow : 1.x vs 2

Tensorflow has been developed by Google and was first launched in November 2015.

Later, an updated version, or what we call as TensorFlow2.0, was launched in September 2019. This led to the older version being classified as TF1.x and the newer version as TF2.0.

TensorFlow was one of the early framework to be developed for builidng neural networks. It is backed by Google, which is one of the main reasons it enjoys a large user base. Because Google continues to integrate AI into every one of their product offerings.

It is in most cases the first framework that you will hear about when starting out with a deep learning or artificial intelligence course.

The question that arises is what changed in the updated version of TensorFlow?

The most important change in TF2.0 over TF1.x is the support for **dynamic computation graphs**.

But what is a computation graph?

A computation graph is the series of operations and mathematical transformations that our input data is subjected to, to arrive at the final output.

Ok, then what is a **static computation graph**?

3. Difference between Static and Dynamic Computation Graph

> A static computation graph basically means that you can't change the parameters of the neural network on the fly, that is, while you are training the neural network.
>
> In a dynamic computation graph on the other hand, you can change the parameters of your neural network on the go, during execution, just like regular python code.

Dynamic computation graphs are often more preferred by developer and engineers.

While there are certain advantages with respect to speed with static computation graphs, especially when it comes to deployment, it can be an overkill for simple neural networks and regular development.

It also doesn't allow a lot of flexibility while experimenting with models.

This is why the update to dynamic computation graphs was a major development in TF2.0.

- It **increased the ease of use for developers** and **made it simpler to experiment with models.**
- **There was no more need for initialising separate sessions** in TensorFlow for separating it from Python.
  `tf.Session` was mandatory in TF1.x but TF2.0 doesnt use sessions at all. It only uses functions.

Static Computation Graphs	Dynamic Computation Graphs
First operations are defined and then executed	Execution is performed as operations are defined
Harder to program and debug	Writing and debugging is easier
Less flexible so it is harder to experiment with models	More flexible which makes experimenting easier
It's more restricted in terms of results; only the final output is available	It is less restricted; you can also see the results of intermediate steps
Easier to optimize; more efficient for deployment	Harder to optimize; not suitable for deployment

4. Keras integration or rather centralization

When TensorFlow 1.x was released, Keras got popular amongst developers to build any TF code. Because Keras simplified the model building process by providing a simpler model building API. Besides, it supported other deep learning frameworks like **Theano** and **CNTK**.

This made it possible to write your deep learning code in Keras, while, allowing the developer to choose whichever backend framework he or she wants with just one line of code.

However, with the release of TF, Keras has become more TensorFlow focused. Rather to say, it has become the center piece around which most code development happens in TF2.0.

Now, keras is a module that you can import from within TF2.0. With this integration, you can build deep learning models in TF2.0 using the original keras approach, namely, Sequential models and Functional APIs.

Plus, with TF2.0, you can also use **Model Subclassing**, which is more like how PyTorch does model building. More on that when we discuss PyTorch in coming section.

For a more summarised view of what's different between static and computation graphs, look at the table below.

To sum up, the differences between TF1.x and TF2.0 can also be summarised as below.

TensorFlow1.x	TensorFlow2.0
Static computation graphs only	Both static and dynamic computation graphs supported
The cycle followed is build-then-run	There are two workflows, eager execution (dynamic graphs) and lazy execution (static graphs)
Low level APIs used though support for high level APIs available	Tightly integrated with Keras API, which is a high level API
`tf.Session` is used for hard separation from Python	No sessions are required; only functions are used
tf.placeholder is required for variables that need estimation	No need for placeholders anymore
Keras was a standalone library that implements TF1.x in backend	Tightly integrates and makes Keras the center piece for development
No Model Subclassing	Allows model subclassing just like PyTorch
Debug with tfdbg	Debug the pythonic way with pdb as well

We have discussed about TF2.0 supports both static and synamic computation graphs. But how is it accomplished?

TF2.0 uses something called as **eager** and **lazy** execution.

5. What is Eager vs Lazy Execution

- Eager execution uses *imperative programming* which is basically the same concept as dynamic computation graphs. Code is executed and run on the go just like how Python works usually.
- Lazy execution uses *symbolic programming* which is same as static computation graphs. Functions are defined abstractly and no actual computation takes place until the function is explicitly compiled and run. That is, the values are not created at the time of defining the functions. In order to materialize the variables, the static computation graph is first created, compiled and then run.

In Tensorflow2.0, you can easily switch between eager execution which are better for development, and lazy mode which are better for deployment.

For a better understanding let’s look at a code snippet below.

```
# In tf2.0, eager execution is default
import tensorflow as tf
tf.executing_eagerly()
```

```
True
```

By default, you can see that Tensorflow2.0 uses eager execution. Let's perform a mathematical operation to check how computation graph is executed as variables and operations are defined.

```
# Define objects
a = tf.constant(5, name = "a")
b = tf.constant(7, name = "b")
c = tf.add(a, b, name = "sum")

c
```

```
<tf.Tensor: shape=(), dtype=int32, numpy=12>
```

In eager execution, the variable `c` 's value can be seen right after declaring variables `a` , `b` , and function `tf.add` .

Let's see the same thing in lazy execution model. It requires the declaration of `tf.Session` for the same operation.

For doing this in Tensorflow2.0, we enable the features of Tensorflow1.x by using `tf.v1.compat` library. All the coding paradigms related to earlier version (Tensorflow 1.x) are bundled up in `tf.compat` module.

```
# Disable eager execution
tf.compat.v1.disable_v2_behavior()

# or do: tf.compat.v1.disable_eager_execution()

tf.compat.v1.executing_eagerly()
```

```
WARNING:tensorflow:From /usr/local/lib/python3.6/dist-
packages/tensorflow/python/compat/v2_compat.py:96: disable_resource_variables (from
tensorflow.python.ops.variable_scope) is deprecated and will be removed in a future
version.
Instructions for updating:
non-resource variables are not supported in the long term




False
```

After running `disable_v2_behavior` you can see that eager execution is no more enabled by default.

Let’s perform another mathematical operation to check if we can still run the computation graph without initialising sessions.

```
# Define objects again
x = tf.constant(10, name = "x")
y = tf.constant(3, name = "y")
z = tf.add(x, y, name = "sum")

z

<tf.Tensor 'sum:0' shape=() dtype=int32>
```

`z` has a value of 0, because in lazy execution, or a static graph, the computation doesn’t take place without first defining all operations and then running it.

Let’s do the same thing as above using a session.

```
# Init Session and run
sess = tf.compat.v1.Session()
sess.run(z)

13
```

Now you can see that the correct value of `z` is visible.

Therefore TF2.0 supports both eager and lazy execution as seen above.

Well clearly TensorFlow2.0 solved the problems we faced with TensorFlow1.x. Where does PyTorch come into the picture then? Keep Reading further.

6. TensorFlow vs PyTorch

PyTorch was has been developed by Facebook and it was launched by in October 2016. At the time of its launch, the only other major/popular framework for deep learning was TensorFlow1.x which supported only static computation graphs.

PyTorch started being widely adopted for 2 main reasons:

- It used **dynamic computation graphs** for building NNs.
- It was **tightly integrated with Python** which made it easier to for Python developers to switch to this framework.

It was fundamentaly different from TensorFlow version available at the time.

Below you can see a summary of the differences between the early versions of TensorFlow and PyTorch.

TensorFlow1.x	PyTorch

TensorFlow1.x	PyTorch
Only static computation graphs supported	Only dynamic computation graphs supported
Debugging is done using TensorFlow specific libaray tfdbg	Debugging can be done using the standard Python library pdb or PyCharm
TensorBoard is used for visualisations of output	Standard python libraries like Matplotlib and Seaborn can be used for visualisation
`tf.Session` is used for separation from Python	PyTorch is tightly integrated with Python so no separation is needed
Data parallelization is difficult; use of `tf.Device` and `tf.DeviceSpec` is required	Data parallelization is easier; `torch.nn.DataParallel` is used

The main differences that changed with the new version of TensorFlow is that we don’t need `tf.Session` anymore and TF2.0 also supports dynamic graphs.

7. Model Subclassing

Another major change in TF2.0 is it allows ‘Model Subclassing’, which is a commonly followed practice to build neural network models in PyTorch. This method allows you to build complex model architectures, highly suited for experimentations.

So, in a sense, TF2.0 has adopted some of the key development practices already followed in PyTorch.

Below is an example of how similar the model subclassing code looks in TF2.0 and PyTorch

```
# Representative Code in PyTorch
import torch.nn as nn
import torch.nn.functional as F

class Model(nn.Module):
    def __init__(self):
        super(Model, self).__init__()
        self.conv1 = nn.Conv2d()
        self.conv2 = nn.Conv2d()

    def forward(self, x):
        x = F.relu(self.conv1(x))
        return F.relu(self.conv2(x))
```

A similar model built in TF2.0 looks something like below. Very similar.

```
# Equivalent Representative Code in TensorFlow2.0
import tensorflow as tf
from tensorflow import keras
from tensorflow.keras import layers, Model

class TFModel(Model):
  def __init__(self):
    super(TFModel, self).__init__()
    self.conv1 = layers.Conv2D()
    self.conv1 = layers.Conv2D()

  def call(self, x):
    x = layers.ReLU(self.conv1(x))
    return layers.ReLU(self.conv2(x))
```

8. Comparison between TensorFlow1.x, TensorFlow2.0 and PyTorch

Now that we know the differences between different versions of TensorFlow and between TensorFlow and PyTorch, let's look at a comaprison between all three, so that next time you decide to build to a deep learning network, you know exactly what framework to use!

TensorFlow1.x	PyTorch	TensorFlow2.0
Only static computation graphs supported	Only dynamic computation graphs supported	Both static and dynamic computation graphs supported
There is a need to use `tf.session` for separation from Python	PyTorch is tightly integrated with python	No requirement of initialising sessions as only functions are used
Low level APIs are used but support for high level APIs is available	REST API is used along with Flask for deployment	Keras API, which is also a high level API, is used for deployment