Machine Learning for IoT

Part-2: ML & DL in IoT

Organization

- This part of the course touches two main topics

1. <u>TensorFlow 2:</u>
   - Overview of the TF2 APIs in Python
   - Learn how to create, train and test models in TF/Keras, how to build data pipelines, etc.
   - Useful for what comes after

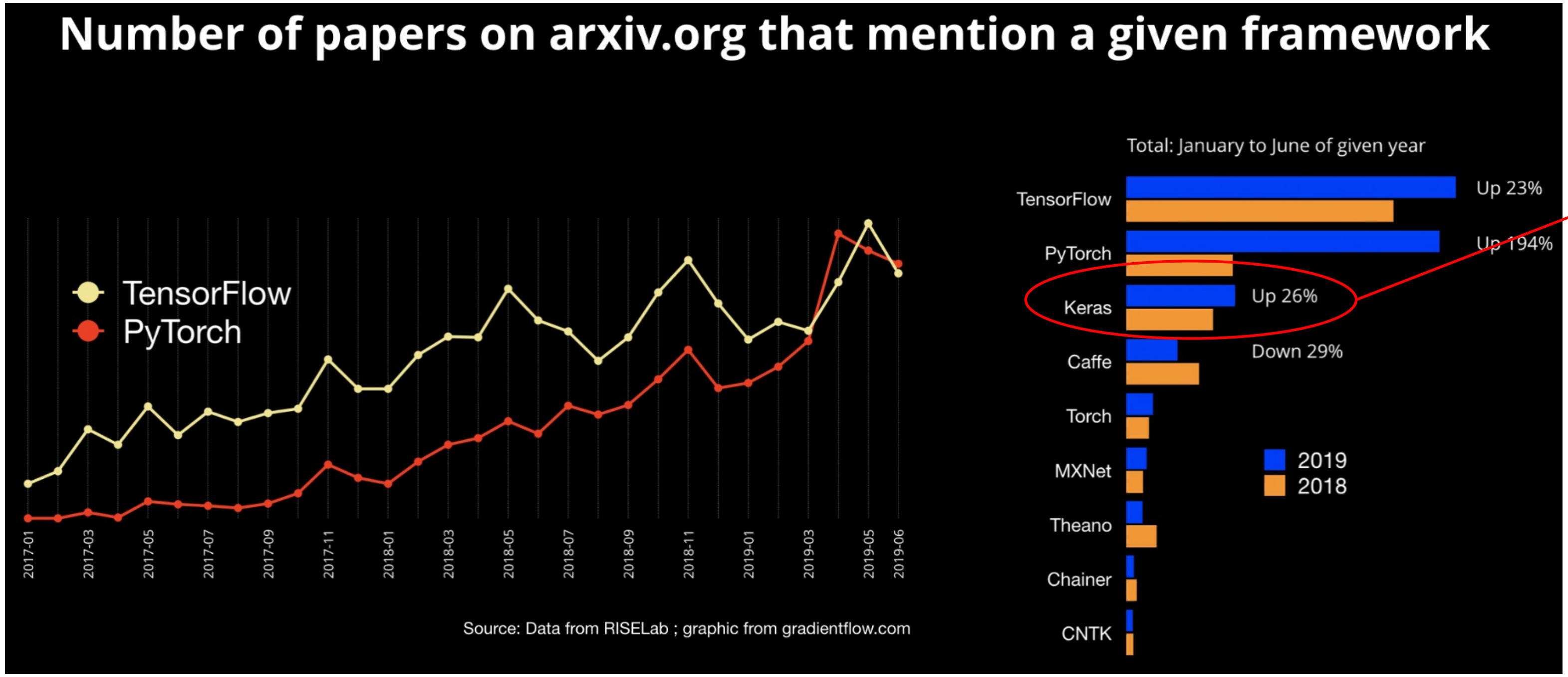
2. <u>Model Optimizations for Deployment on IoT Devices:</u>
   - Overview of the main optimizations that can be applied to your ML/DL model to make it smaller, more energy efficient and faster when deployed on an IoT device.
   - E.g. data quantization, pruning, distillation, etc.
   - Examples on how to apply these in TF2 (*when possible)

Tensorflow 2

Introduction

Why TF2

- Reason #1: It's good for you to know!



Keras is strongly coupled with TF 2

- PyTorch is quickly catching up, but TF is still the **top-1 ML/DL framework** in both academia and industry

Why TF2

- Reason #1: it's good for you to know!
  - You should be already familiar with PyTorch from Prof. Caputo's course Machine Learning and Deep Learning
  - After this course, you'll have at least a basic familiarity with the top-2 industry-backed frameworks for ML/DL

– **Disclaimer:** this is **not** a theoretical ML/DL course. We will:

  - Overview the basic APIs, assuming that you're familiar with the underlying ML/DL concepts
  - Focus on aspects related to model deployment for IoT.

Why TF2

- Reason #2: deployment features

  - Being the first DL-oriented, industry-backed framework to appear, TF is (or was…) more advanced from the point of view of deployment, especially for IoT targets.

  - First to introduce a mobile/edge-device oriented runtime (TensorFlow Lite)
    - ARM Cortex-A class devices, such a smartphones or the Raspberry Pi, supporting Android, iOS or Linux

  - First to introduce a runtime for microcontrollers (TFLite Micro)
    - ARM Cortex-M class devices

  - **PyTorch is catching up here too** (see PyTorch Mobile).

Why TF2

- Reason #2: deployment features

  - TF has better support in third-party deployment toolchains:
    - E.g. STMicroelectronics CUBE.AI framework for STM32 microcontrollers
    - PyTorch-based deployment supported through an intermediate ONNX conversion which limits the available features

  - TF supports a richer set of model optimization features for IoT
    - Post-training and training-aware quantization
    - Weights pruning, etc.

  - More on this later, don't worry…

Sources

- Some inspiration for the following material has been taken from these sources:
  - Tensorflow 2 official documentation
  - "*Introduction to Tensorflow 2.0*" by Google on Coursera
  - "*Tensorflow: Data and Deployment*" by deeplearning.ai on Coursera
  - "*Tensorflow 2 for Deep Learning Specialization*" by Imperial College London on Coursera
  - The Web…

What is Tensorflow

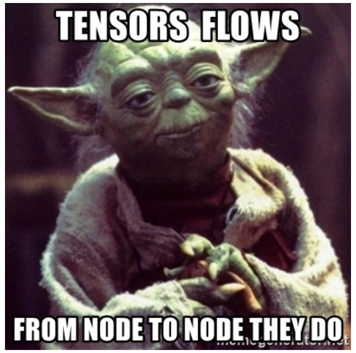
- Definition from Google's course on Coursera: *"TensorFlow is an open-source, high-performance library for numerical computation that uses directed graphs"*
    - Any numerical computation, not just machine learning/deep learning
    - Graph-based programming model
    - API to write code in a high-level language (Python) and have it executed in an extremely efficient way (C++)

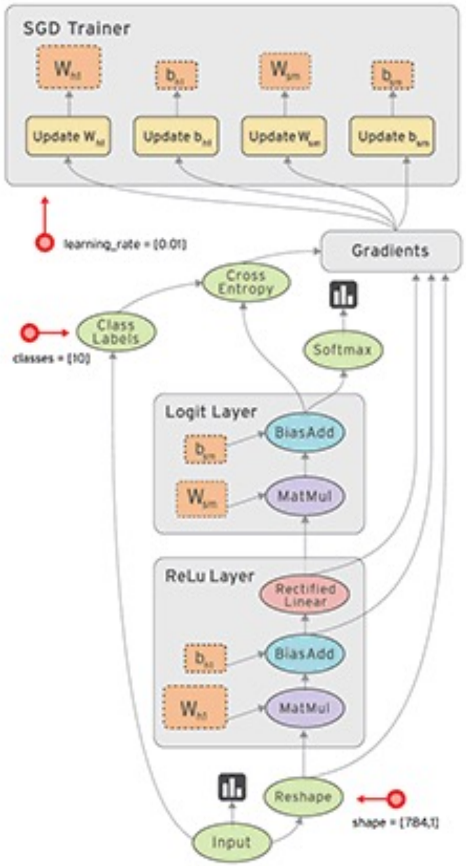
TensorFlow

What is Tensorflow

- Respresent your computation as a Directed (Acyclic) Graph, or DAG
- Nodes represent operations (MatMul, Add, ReLU, etc.)
- Edges represents data (tensors) flowing towards the output

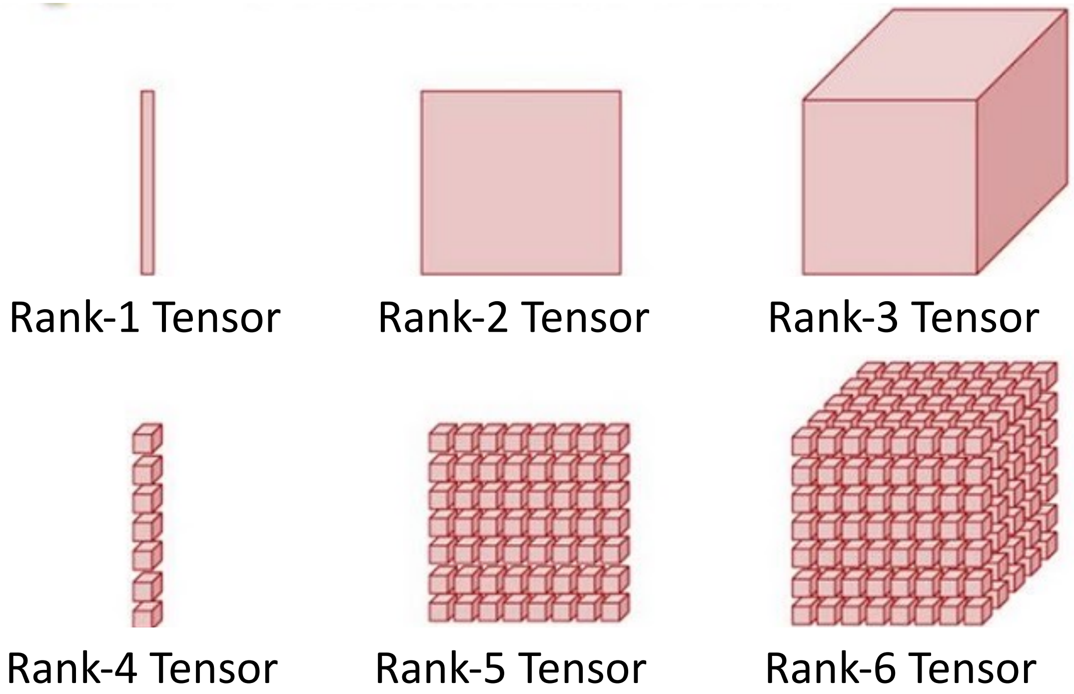
TensorFlow





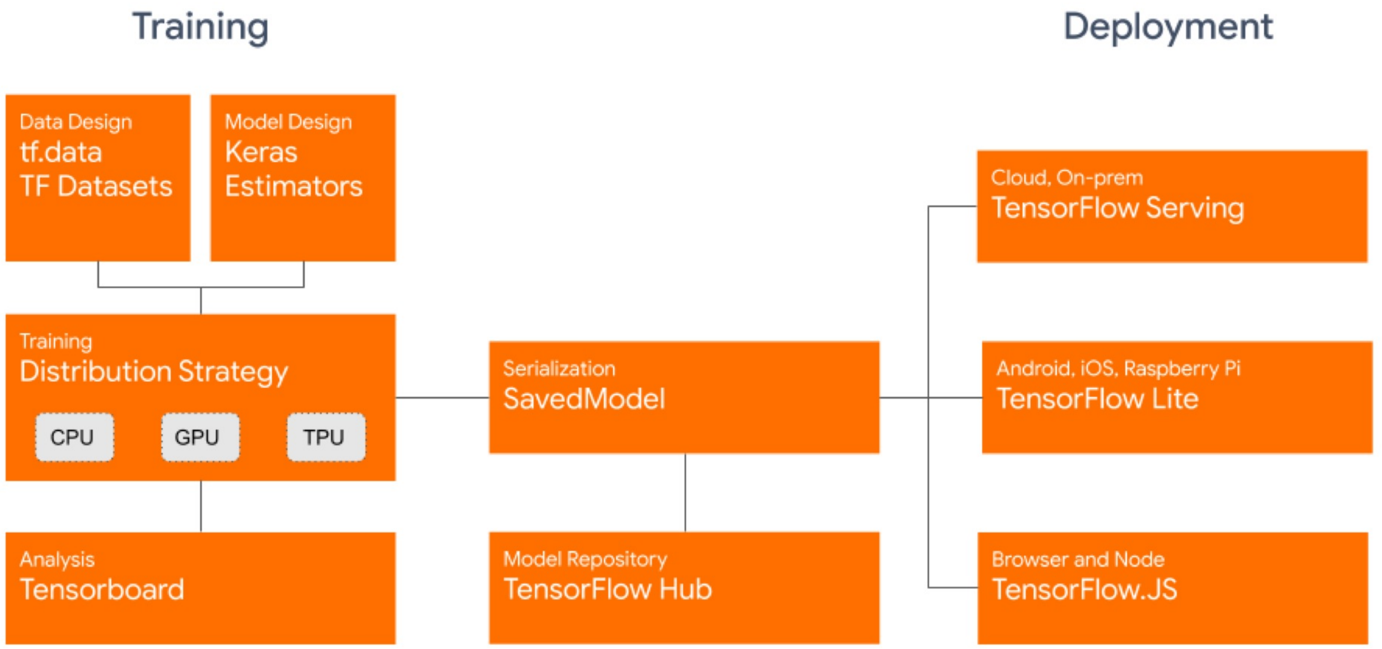
What is Tensorflow

- Tensor = multi-dimensional array of data
  - Tensor rank = number of dimensions (scalar = rank 0)



What is Tensorflow

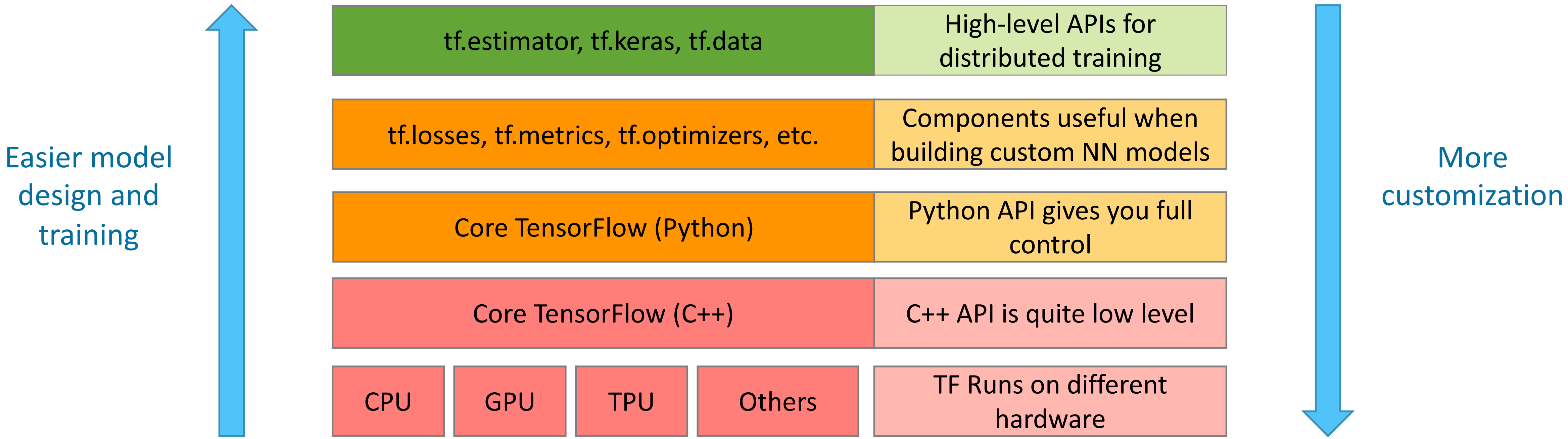
- Why a DAG model of computation? --> Portability
- Single DAG model, multiple target hardware (and languages)
  - TF execution engine (C++) extremely optimized for the target HW (CPU, GPU, etc.)
  - Model developer doesn't have to care about these optimizations



All relevant in the IoT world

Tensorflow API Hierarchy

- Tensorflow exposes APIs at multiple abstraction levels

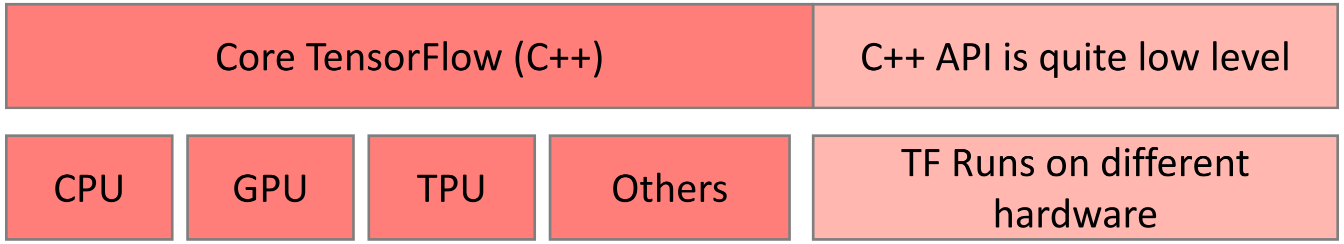


Tensorflow API Hierarchy

- Back-end code for different hardware platforms.
  - Low-level kernels in CUDA for NVIDIA GPUs, Math Kernel Library (MKL) for Intel CPUs, etc.
  - Almost never touched directly except by hardware developers

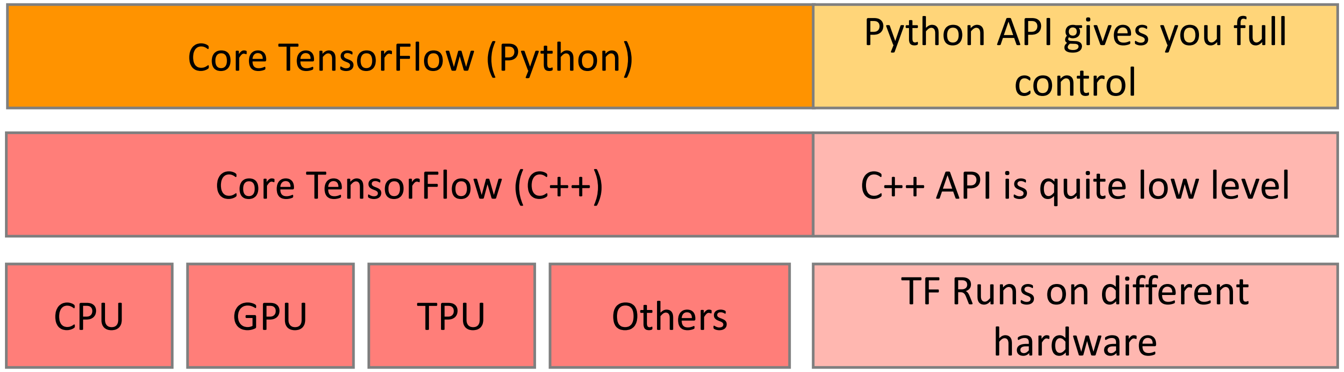
Tensorflow API Hierarchy

- The Core C++ API is used to write a custom TF Op. or extend an existing one
  - You can then export Python wrappers to use these ops within your model
  - Again, rarely touched directly except for advanced ML/DL research



Tensorflow API Hierarchy

- The Core Python API contains most of the numeric processing code:
  - Add, subtract, mul, etc.
  - Creation of variables and tensors
  - Not ML-specific



Tensorflow API Hierarchy

- A set of convenience modules containing pre-made Neural Network (NN) components
  - Entire layers (tf.layers), loss functions (tf.losses), metrics (tf.metrics), gradient-based optimizers (tf.optimizers), etc.
  - Useful to build custom NN models, training loops, etc.

tf.losses, tf.metrics, tf.optimizers, etc.	Components useful when building custom NN models
Core TensorFlow (Python)	Python API gives you full control
Core TensorFlow (C++)	C++ API is quite low level
CPU / GPU / TPU / Others	TF Runs on different hardware

Tensorflow API Hierarchy

- High-level APIs for standard train/evaluate/serve flows and models
  - Make model definition, data preprocessing, (distributed) training, etc. much easier

tf.estimator, tf.keras, tf.data	High-level APIs for distributed training
tf.losses, tf.metrics, tf.optimizers, etc.	Components useful when building custom NN models
Core TensorFlow (Python)	Python API gives you full control
Core TensorFlow (C++)	C++ API is quite low level
CPU, GPU, TPU, Others	TF Runs on different hardware

Tensorflow API Hierarchy

- Our focus in this course

tf.estimator, tf.keras, tf.data	High-level APIs for distributed training
tf.losses, tf.metrics, tf.optimizers, etc.	Components useful when building custom NN models
Core TensorFlow (Python)	Python API gives you full control
Core TensorFlow (C++)	C++ API is quite low level
CPU, GPU, TPU, Others	TF Runs on different hardware

Tensorflow 2

TF vs PyTorch

TF vs PyTorch - Release

• TF first released by Google in 2015, PyTorch released by Facebook in 2017

Third pic omitted…

TF vs PyTorch - API Complexity

- Core TF APIs were quite complex and innatural for Python developers, especially with the static graph paradigm (see next slide)

- PyTorch was much more pythonic from the beginning

- TF has gradually added higher-level and easier APIs. The tighter integration with Keras in TF2.0 made TF-based development much easier.

TF vs PyTorch – Computation Graph

- Both frameworks model computation as a graph
- TF **initially** adopted a **static graph** approach whereas PyTorch always used a **dynamic graph**.

- Static graph:
  - DAG defined beforehand with a placeholder for data.
  - Then, data was fed to the graph (during a so-called “session”) to run training/inference
  - Great for performance on multiple targets, but painful to debug and limited flexibility

- Dynamic graph:
  - Computations done line by line as code is interpreted
  - Easier to debug, and more flexible (e.g. variable-length inputs for RNNs)

TF vs PyTorch – Computation Graph (cont'd)

- TF later introduced a so-called "Eager execution" mode to support dynamic graphs. This became the default in TF2
- Both frameworks still allow building/exporting static graphs (e.g. for TFLite and TorchScript)
- Both "eager" and "graph" modes available in both frameworks

TF vs PyTorch – Distributed Computing

- In early days, training on multiple GPUs was not easy in TF

- Now it is almost effortless in both frameworks

- TF has better support for Google’s TPUs (of course)

TF vs PyTorch – Deployment

- The main area where TF is still slightly more mature

- As anticipated:
  - Better support for deployment to IoT devices (including microcontrollers)
  - Better support by third party toolchains

TF vs PyTorch - Summary

- In summary, in early days the two frameworks were based on quite different phylosophies.
- Nowadays, the similarities are many more than the differences.
- It is sometimes not easy to distinguish TF code from PyTorch code

TF vs PyTorch - Example

- NN Model definition with the "subclassing" API

PyTorch

```
class MyModel(nn.Module):
  def __init__(self):
    super(MyModel, self).__init__()
    self.conv1 = Conv2d(in_channels=1,
          out_channels=32, kernel_size=3)
    self.flatten = Flatten()
    self.d1 = Linear(21632, 128)
    self.d2 = Linear(128, 10)

  def forward(self, x):
    x = F.relu(self.conv1(x))
    x = self.flatten(x)
    x = F.relu(self.d1(x))
    x = self.d2(x)
    return output
```

Tensorflow (Keras)

```
class MyModel(Model):
  def __init__(self):
    super(MyModel, self).__init__()
    self.conv1 = Conv2D(filters=32,
      kernel_size=3, activation='relu')
    self.flatten = Flatten()
    self.d1 = Dense(128, activation='relu')
    self.d2 = Dense(10)

  def call(self, x):
    x = self.conv1(x)
    x = self.flatten(x)
    x = self.d1(x)
    output = self.d2(x)
    return output
```