Short Keras Introduction

Dr. U. Michelucci (TOELT)
umberto.Michelucci@toelt.ai

TensorFlow

TensorFlow 1.x/2.x

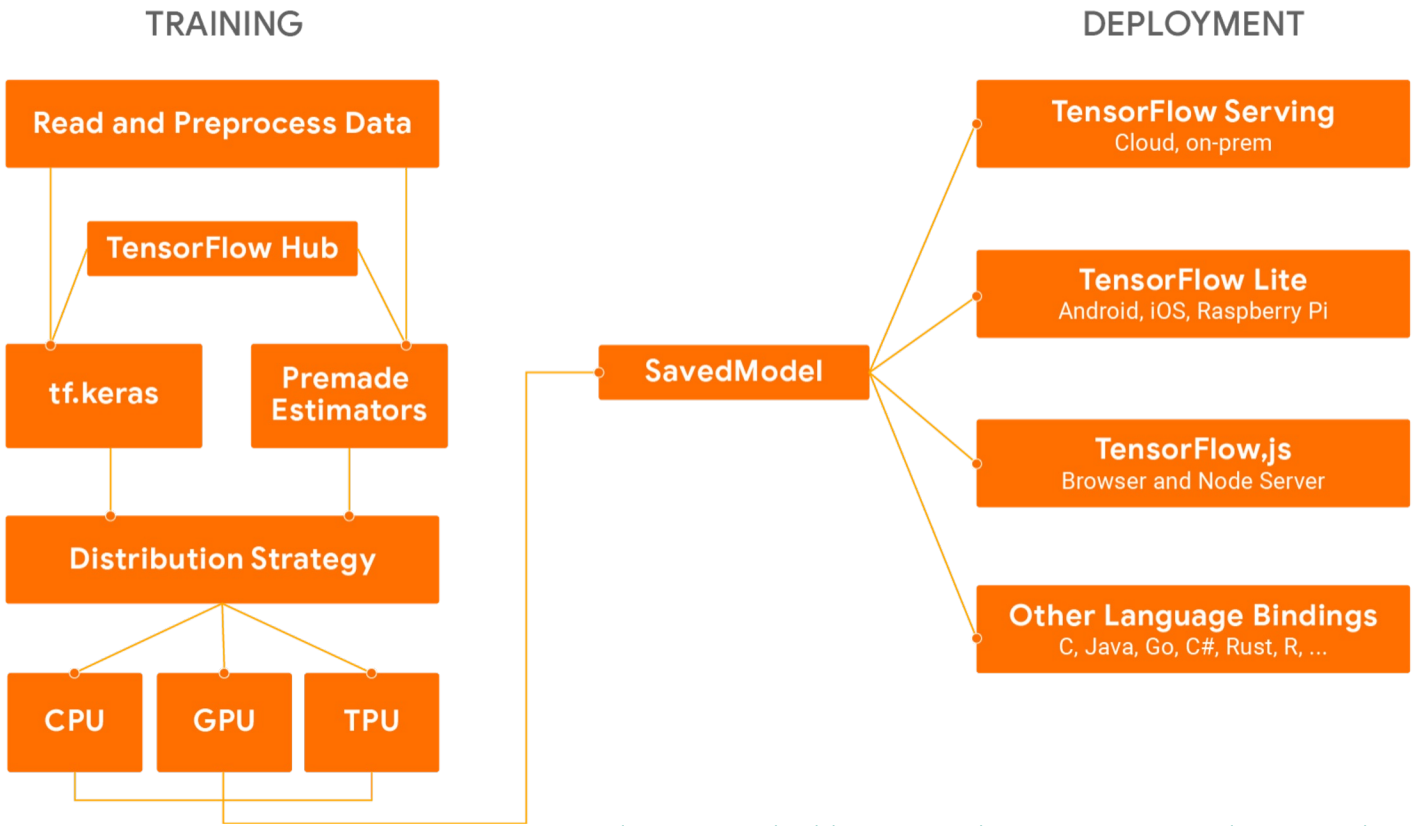
An open source Deep Learning library

- **>1,800 contributors worldwide**
- Apache 2.0 license
- Released by Google in 2015

TensorFlow 2.0

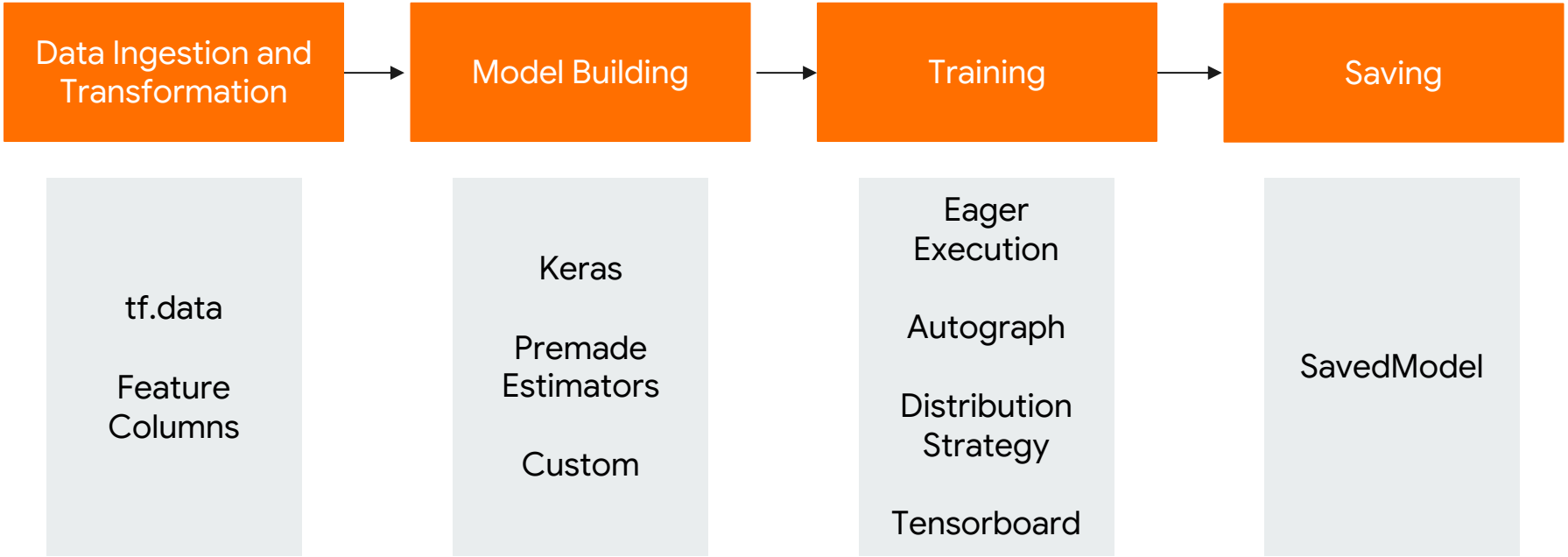
- Easier to learn and use
- For **beginners** and **experts**
- Available today

TensorFlow architecture



Source: Getting Started with TensorFlow 2.0 presentation Google I/O 2019

Training Workflow



Source: Getting Started with TensorFlow 2.0 presentation Google I/O 2019

TensorFlow

2.x

What is new in TF2.0

- Easy model building with **Keras** and **eager execution** (activated by default in TF2.0).
- Robust model **deployment** in production on any platform.
- Powerful experimentation for research.
- Simplifying the API by **cleaning up deprecated APIs** and reducing duplication (relevant in case you have code developed in TensorFlow 1.X and you need to convert it)
- Load your data using **tf.data**. Training data is read using input pipelines which are created using tf.data.
- Build, train and validate your model with **tf.keras**, or use **Premade Estimators**.
- **TensorFlow Hub**.
- **Run and debug with eager execution**, then use **tf.function** for the benefits of graphs.
- Use Distribution Strategies for distributed training.
- hardware accelerators like CPUs, GPUs, and TPUs; you can enable training workloads to be distributed to single-node/multi-accelerator as well as multi-node/multi-accelerator configurations, including TPU Pods.
- Export to **SavedModel**. TensorFlow will standardize on SavedModel as an interchange format for TensorFlow Serving, TensorFlow Lite, TensorFlow.js, TensorFlow Hub, and more.
- **Tensorflow Datasets**

`tensorflow-gpu` version dependencies

Software requirements

The following NVIDIA® software must be installed on your system:

- NVIDIA® GPU drivers —CUDA 10.0 requires 410.x or higher.
- CUDA® Toolkit —TensorFlow supports CUDA 10.0 (TensorFlow >= 1.13.0)
- CUPTI ships with the CUDA Toolkit.
- cuDNN SDK (>= 7.4.1)
- *(Optional)* TensorRT 5.0 to improve latency and throughput for inference on some models.

Source: TF documentation

Check the installed version

```
import tensorflow as tf
print(tf.__version__)
```

☆When you change TF version you need to restart the runtime in Google Colab

TensorFlow 2.0

Usability

- `tf.keras` as the recommended high-level API.
- Eager execution by default.

```
>>> tf.add(2, 3)
<tf.Tensor: id=2, shape=(), dtype=int32, numpy=5>
```

Source: Getting Started with TensorFlow 2.0 presentation Google I/O 2019

TensorFlow 2.0

Clarity

- Remove duplicate functionality
- Consistent, intuitive syntax across APIs
- Compatibility throughout the TensorFlow ecosystem

Source: Getting Started with TensorFlow 2.0 presentation Google I/O 2019

TensorFlow 2.0

Flexibility

- Full lower-level API.
- Internal ops accessible in `tf.raw_ops`
- Inheritable interfaces for variables, checkpoints, layers.

Source: Getting Started with TensorFlow 2.0 presentation Google I/O 2019

How to study TF2.0

https://github.com/michelucci/TensorFlow20-Notes/blob/master/howto_study_TF20.md

Basics

1. Tensors
2. Computational graphs basics (this is relevant when you are going to study how `@tf.function` works and how it creates a graph in the background. Unless you understand how Computational graphs are working, you will not really understand how `@tf.function` is working)
3. Eager Execution (basics and subtelties)
4. `tf.Variable` (aka no need to initialize them anymore in eager mode)
5. Keras basics (`Sequential()` models, `.compile()` and `.fit()` calls)
6. Keras functional APIs
7. `tf.data.Dataset` and data processing pipelines

How to study TF2.0

https://github.com/michelucci/TensorFlow20-Notes/blob/master/howto_study_TF20.md

Advanced

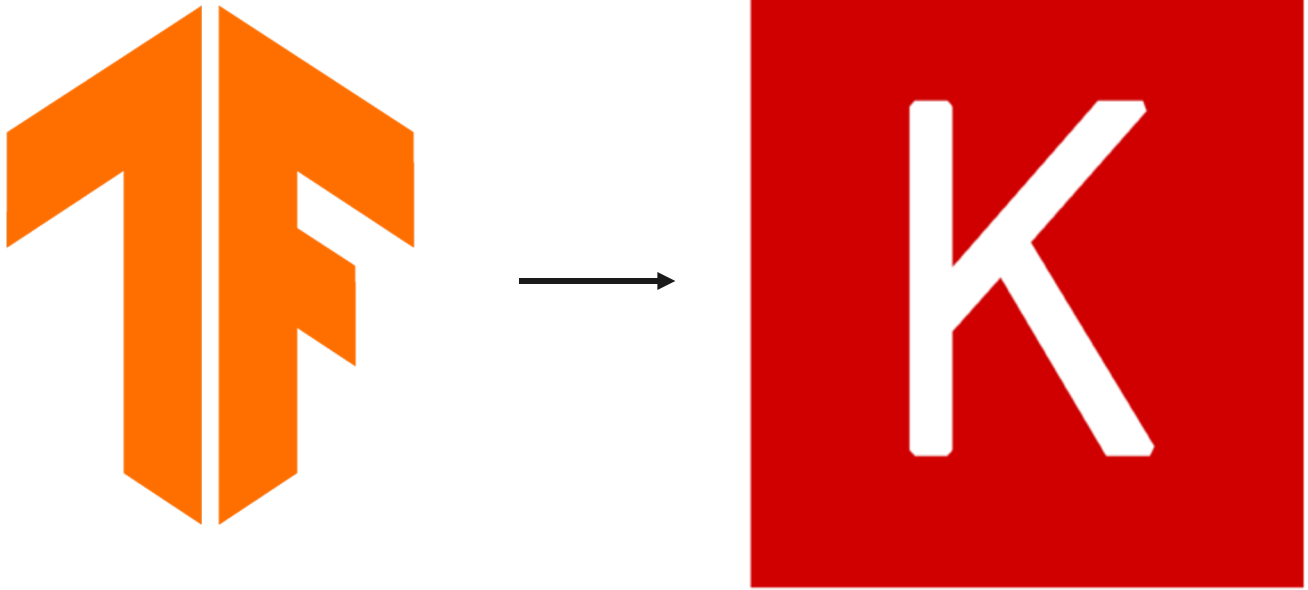
1. Model subclassing
2. Custom layers in Keras
3. Custom callbacks in Keras
4. `@tf.function`
5. Keras metrics
6. Autodifferentiation (`tf.GradientTape()`) and custom training loops
7. Custom layers and models
8. Save and load models
9. Using CPUs, GPUs and TPUs (hardware acceleration)

Use a built-in training loop...

```
model.fit(x_train, y_train, epochs=5)
```

Keras

KERAS: high level API



Keras.io (Reference implementation)

```
import keras
```

TensorFlow implementation

```
from tensorflow import keras
```

☆TensorFlow's implementation (a superset, built-in to TF)

For beginners

Model definition

```
model = tf.keras.models.Sequential(...)
```

Model compilation

```
model.compile(...)
```

Model fitting

```
model.fit(...)
```

Sequential() model

```
model = tf.keras.models.Sequential([
  tf.keras.layers.Flatten(),
  tf.keras.layers.Dense(512, activation='relu'),
  tf.keras.layers.Dropout(0.2),
  tf.keras.layers.Dense(10, activation='softmax')
])
```

A sequence of layers stacked one after the other

Sequential() model - beginners

```
model = tf.keras.models.Sequential([
  tf.keras.layers.Flatten(),
  tf.keras.layers.Dense(512, activation='relu'),
  tf.keras.layers.Dense(10, activation='softmax')
])
model.compile(optimizer='adam',
              loss='sparse_categorical_crossentropy',
              metrics=['accuracy'])

model.fit(x_train, y_train, epochs=5)
model.evaluate(x_test, y_test)
```

Sequential() model - experts

```
class MyModel(tf.keras.Model):
  def __init__(self, num_classes=10):
    super(MyModel, self).__init__(name='my_model')
    self.dense_1 = layers.Dense(32, activation='relu')
    self.dense_2 = layers.Dense(num_classes,activation='sigmoid')

  def call(self, inputs):
    x = self.dense_1(inputs)
    return self.dense_2(x)
```

`Sequential()` model

Sequential() model

The Sequential model is a linear stack of layers.
You can create a Sequential model by passing a list of layer instances to the constructor:

```
from keras.models import Sequential
from keras.layers import Dense, Activation

model = Sequential([
    Dense(32, input_shape=(784,)),
    Activation('relu'),
    Dense(10),
    Activation('softmax'),
])
```

Sequential() model

You can also simply add layers via the .add() method

```
model = Sequential()
model.add(Dense(32, input_dim=784))
model.add(Activation('relu'))
```

Compilation

Before training a model, you need to configure the learning process

```
model.compile(optimizer='rmsprop',
              loss='categorical_crossentropy',
              metrics=['accuracy'])
```

Training

Keras models are trained on Numpy arrays of input data and labels.

```
model.fit(x_train, y_train, epochs=5)
```

Layers in Keras

keras.layers

Most used

```
from keras.layers import Dense, Conv2D,
MaxPooling2D, Dropout, Flatten

keras.layers.Dense
keras.layers.Conv2D
keras.layers.MaxPooling2D
keras.layers.Dropout
keras.layers.Flatten
```

Example of custom layer

```
class Linear(layers.Layer):
  def __init__(self, units=32, input_dim=32):
    super(Linear, self).__init__()
    w_init = tf.random_normal_initializer()
    self.w = tf.Variable(initial_value=w_init(shape=(input_dim, units),
                                              dtype='float32'),
                         trainable=True)
    b_init = tf.zeros_initializer()
    self.b = tf.Variable(initial_value=b_init(shape=(units,),
                                              dtype='float32'),
                         trainable=True)

  def call(self, inputs):
    return tf.matmul(inputs, self.w) + self.b

x = tf.ones((2, 2))
linear_layer = Linear(4, 2)
y = linear_layer(x)
```

Custom callback classes

Introduction

A custom callback is a powerful tool to customize the behavior of a Keras model during training, evaluation, or inference, including reading/changing the Keras model.

> “In Keras, **`Callback`** is a python class meant to be subclassed to provide specific functionality, with a set of methods called at various stages of training (including batch/epoch start and ends), testing, and predicting. “

Source: https://www.tensorflow.org/guide/keras/custom_callback

Custom `Callback` methods

- `on_train_begin`: Called at the beginning of training
- `on_train_end`: Called at the end of training
- `on_epoch_begin`: Called at the start of an epoch
- `on_epoch_end`: Called at the end of an epoch
- `on_batch_begin`: Called right before processing a batch
- `on_batch_end`: Called at the end of a batch

Example of Callback

```
class CustomCallback1(keras.callbacks.Callback):
    def on_epoch_end(self, epoch, logs={}):
        print ("Just finished epoch", epoch)
        print (logs)
        return
```

Callback during training

```
CC1 = CustomCallback1()
model.fit(train_images, train_labels, epochs = 2,validation_data =
   (test_images,test_labels), callbacks = [CC1])

>>> {'val_loss': 0.2545496598124504, 'val_acc': 0.9244, 'loss':
0.05098680723309517, 'acc': 0.9878}
```

Example of `Callback`

```
class CustomCallback3(keras.callbacks.Callback):
     def on_epoch_end(self, epoch, logs={}):
          print ("Just finished epoch", epoch)
          print ('Loss evaluated on the validation dataset
     =',logs.get('val_loss'))
          print ('Accuracy reached is', logs.get('acc')) return
```

```
>>> Just finished epoch 0
>>> Loss evaluated on the validation dataset = 0.2546206972360611
```

Frequency of logging

```
if (epoch % 10 == 0):
```

Pre-ready callbacks

Classes

`class BaseLogger` : Callback that accumulates epoch averages of metrics.

`class CSVLogger` : Callback that streams epoch results to a csv file.

`class Callback` : Abstract base class used to build new callbacks.

`class EarlyStopping` : Stop training when a monitored quantity has stopped improving.

`class History` : Callback that records events into a `History` object.

`class LambdaCallback` : Callback for creating simple, custom callbacks on-the-fly.

`class LearningRateScheduler` : Learning rate scheduler.

`class ModelCheckpoint` : Save the model after every epoch.

`class ProgbarLogger` : Callback that prints metrics to stdout.

`class ReduceLROnPlateau` : Reduce learning rate when a metric has stopped improving.

`class RemoteMonitor` : Callback used to stream events to a server.

`class TensorBoard` : Enable visualizations for TensorBoard.

`class TerminateOnNaN` : Callback that terminates training when a NaN loss is encountered.

https://www.tensorflow.org/versions/r2.0/api_docs/python/tf/keras/callbacks

Multiple Callbacks

Multiple callbacks can be used as a simple list

```
model.fit(x_train, y_train,
          batch_size=64,
          steps_per_epoch=5,
          epochs=15,
          verbose=0,
          callbacks=[LossAndErrorPrintingCallback(),
                     LearningRateScheduler(lr_schedule)])
```

Keras functional APIs

Keras Functional APIs - an introduction

The Keras functional API is the way to go for defining complex models, such as multi-output models, directed acyclic graphs, models with shared layers or for example residual neural networks.

Example of Keras Functional APIs

Model definition

```
from keras.layers import Input, Dense
from keras.models import Model

inputs = Input(shape=(784,))
output_1 = Dense(64, activation='relu')(inputs)
output_2 = Dense(64, activation='relu')(output_1)
predictions = Dense(10, activation='softmax')(output_2)
model = Model(inputs=inputs, outputs=predictions)
```

Example of Keras Functional APIs

Compiling and training remain the same

```
model.compile(optimizer='rmsprop',
              loss='categorical_crossentropy',
              metrics=['accuracy'])
model.fit(data, labels)
```

Keras Functional APIs - everything is a layer

With the functional API, it is easy to reuse trained models: you can treat any model as if it were a layer, by calling it on a tensor. Note that by calling a model you aren't just reusing the architecture of the model, you are also reusing its weights.

Keras Functional APIs - everything is a layer

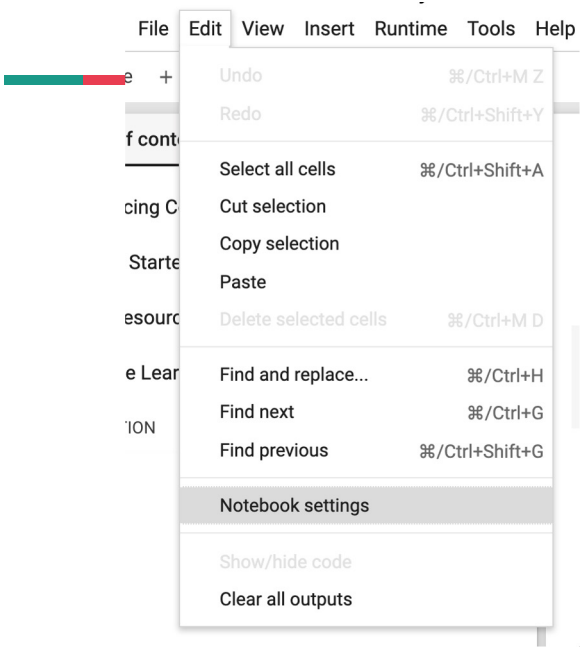
This can allow, for instance, to quickly create models that can process sequences of inputs. You could turn an image classification model into a video classification model, in just one line.

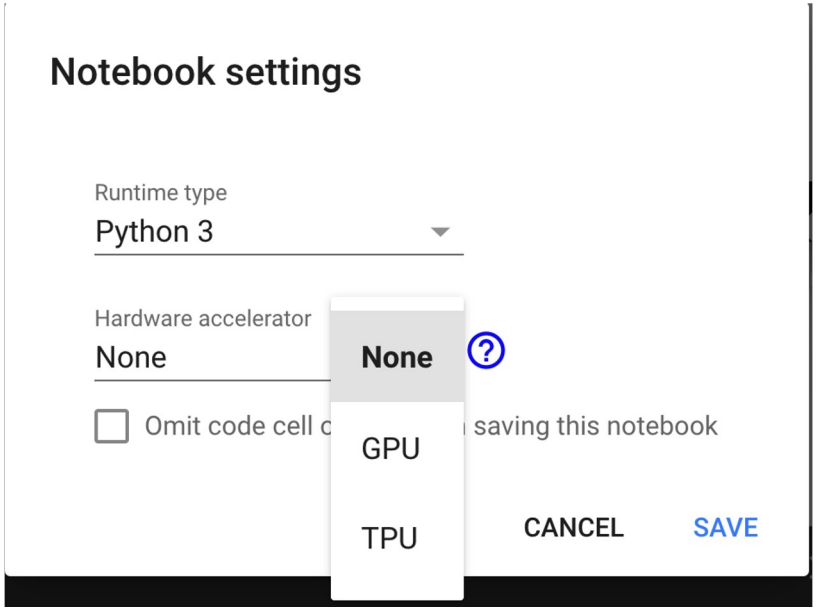
```
from keras.layers import TimeDistributed

input_sequences = Input(shape=(20, 784))
processed_sequences = TimeDistributed(model)(input_sequences)
```

Hardware acceleration

Using hardware acceleration in Google Colab





Changing the Hardware accelerator setting will make restarting the runtime necessary

Testing presence of a GPU

```
print(tf.test.is_gpu_available())

device_name = tf.test.gpu_device_name()
if device_name != '/device:GPU:0':
    raise SystemError('GPU device not found.')
print('Found GPU at: {}'.format(device_name))
```

Putting operations on a device

```
with tf.device('/gpu:0'):
   # Your code here

with tf.device('/cpu:0'):
   # Your code here
```