University of Minho
School of Engineering

ISLab

Deep Learning with TensorFlow™

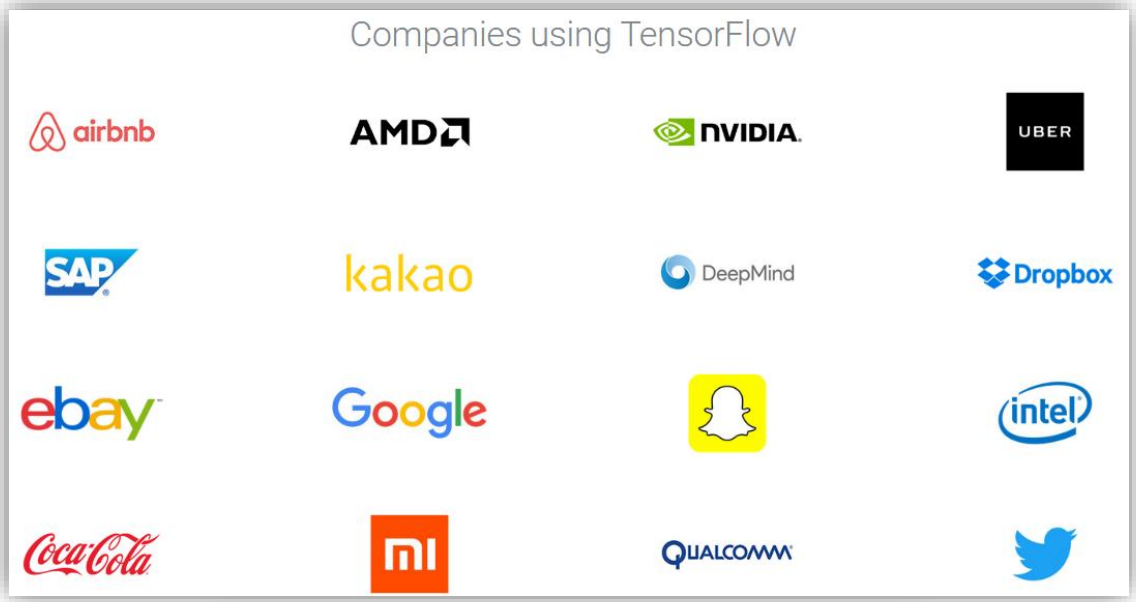
Connective Systems and Classifiers

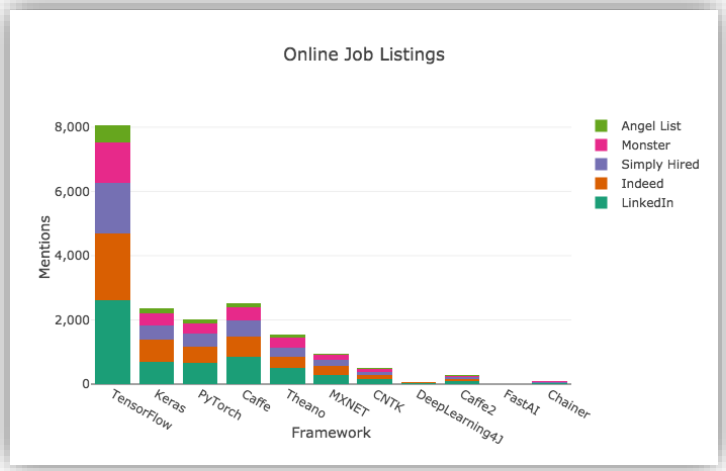
Perfil ML:FA @ MiEI/4º ano - 2º Semestre
Bruno Fernandes, Victor Alves

20/02/2020

Contents

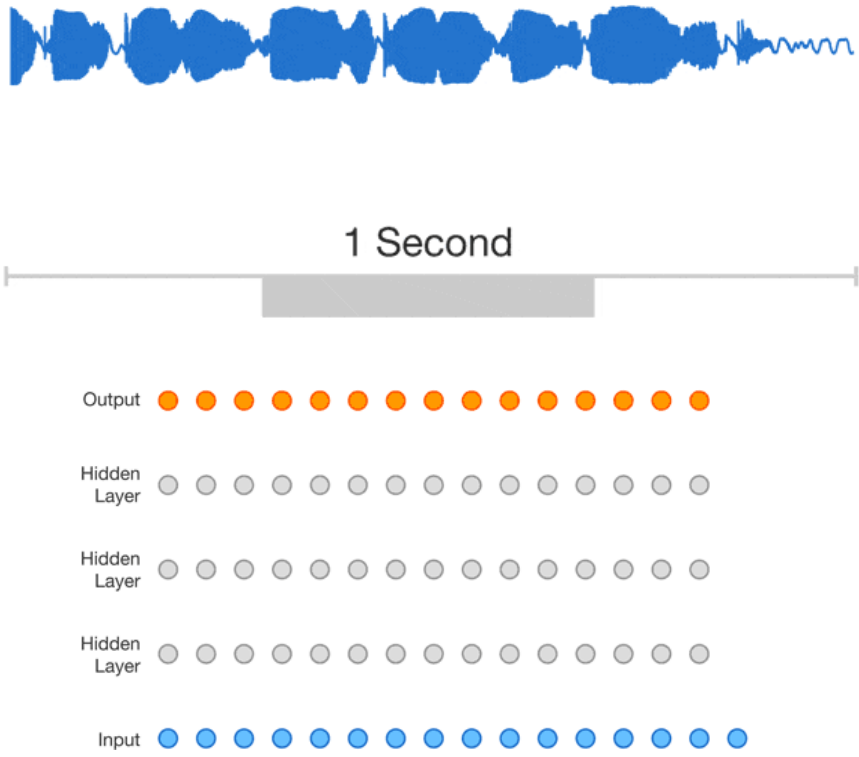
A brief re-introduction





Some cool projects!

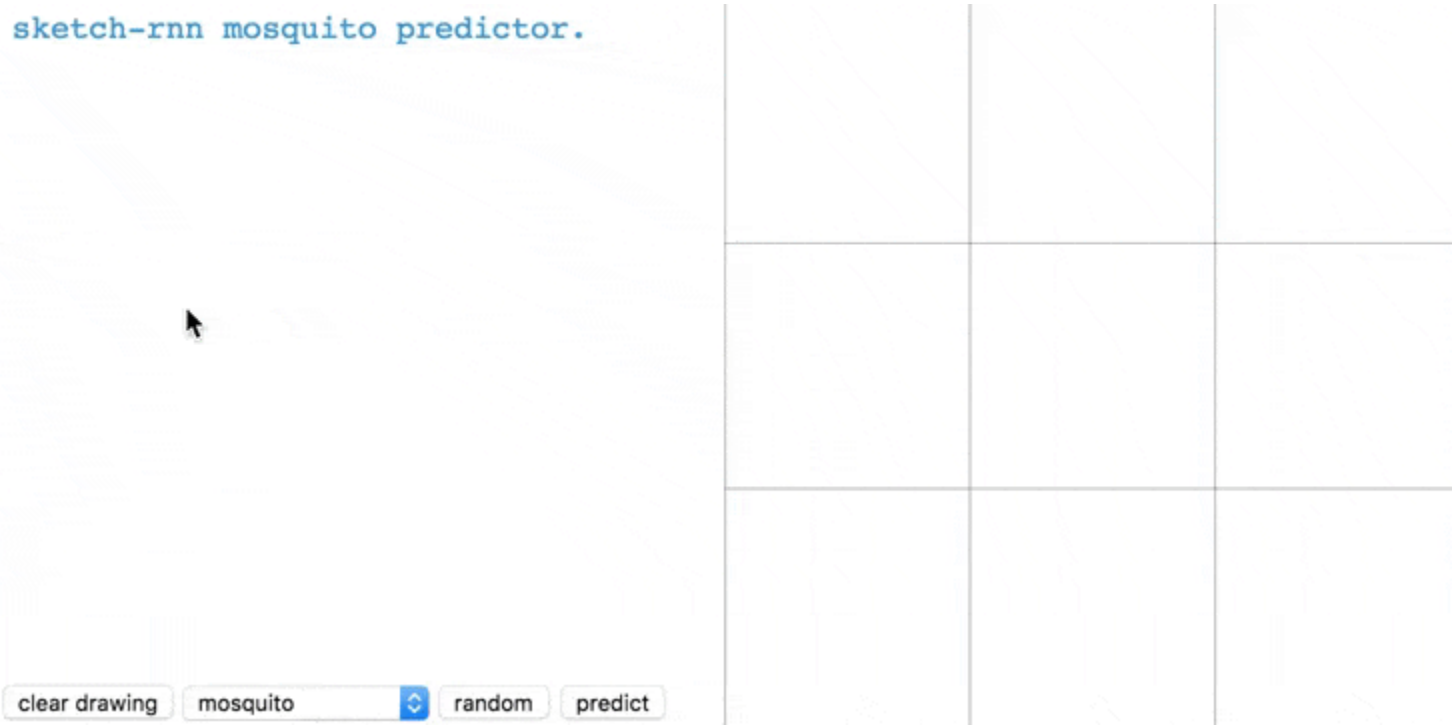
Text to Speech!



(https://deepmind.com/blog/wavenet-generative-model-raw-audio/)

Some cool projects!

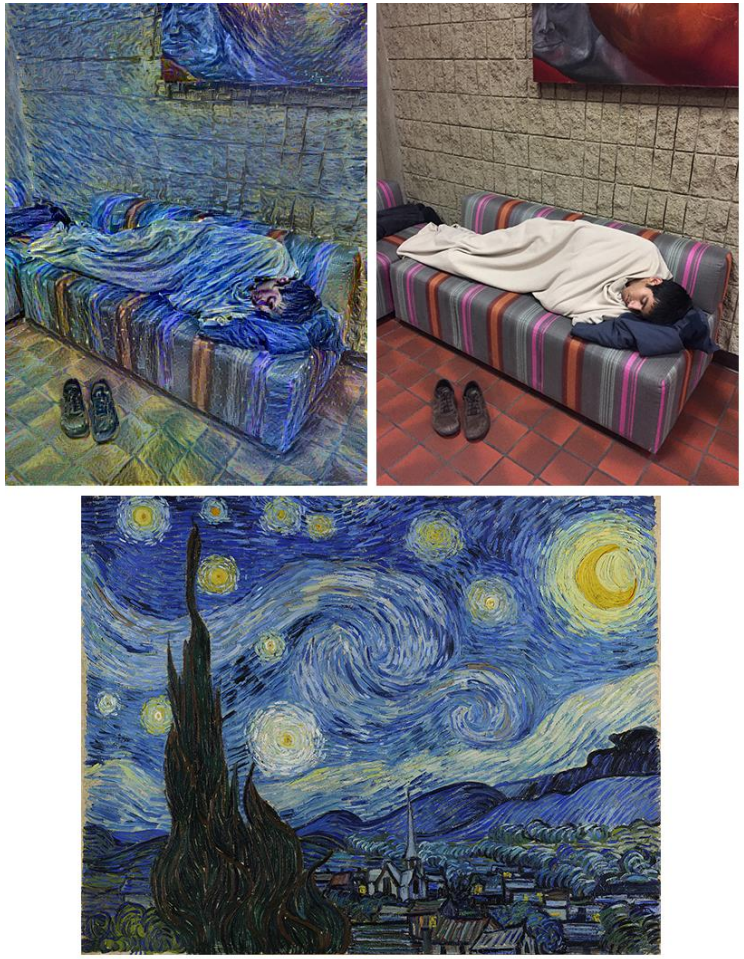
Drawing together with a Neural Network!



(https://magenta.tensorflow.org/sketch-rnn-demo)

Some cool projects!

Neural Style Translation!



(https://github.com/anishathalye/neural-style)

Some cool projects!

Let there be Color!

(http://iizuka.cs.tsukuba.ac.jp/projects/colorization/en/)

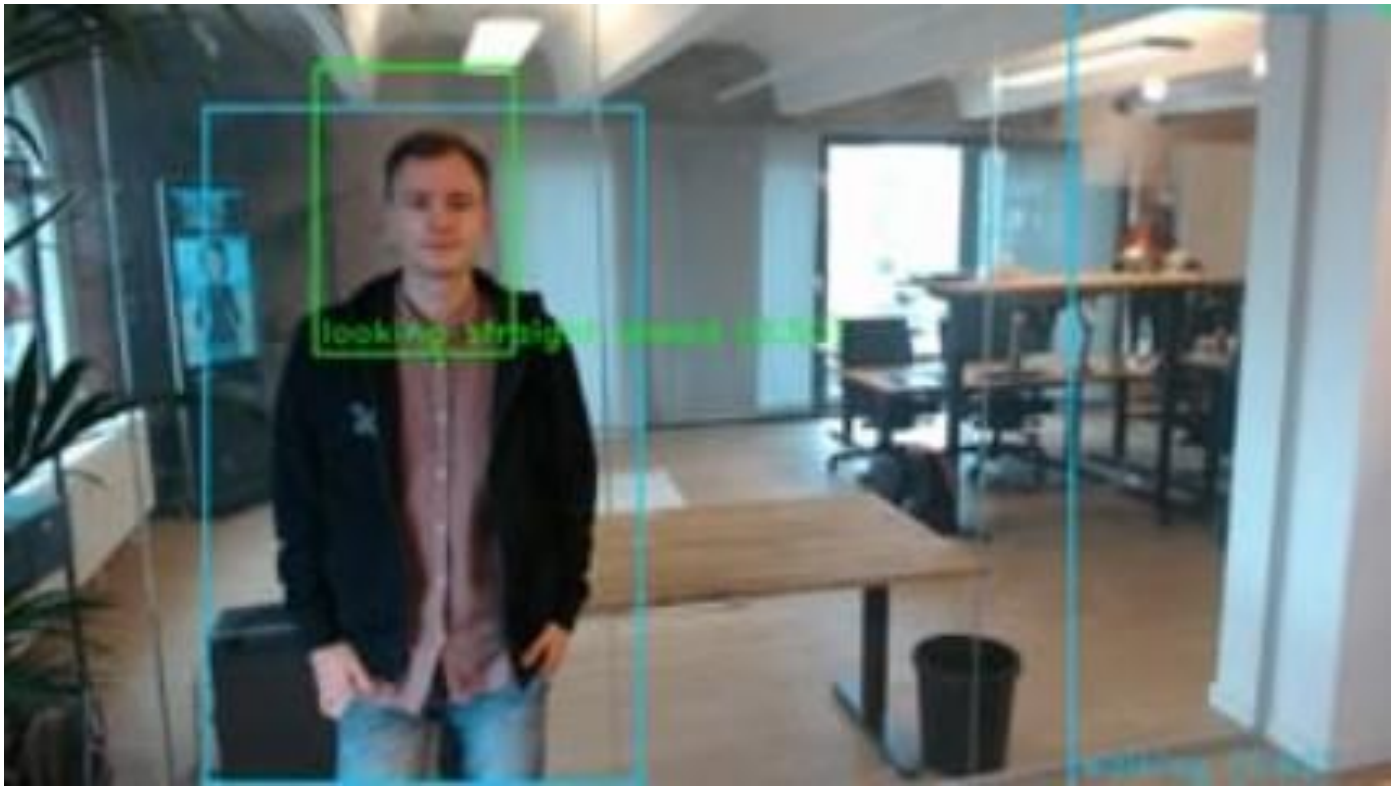
Some cool projects!

Let there be Color!

(http://iizuka.cs.tsukuba.ac.jp/projects/colorization/en/)

Some cool projects!

Human behavior Understanding!



(https://20bn.com/products/datasets)

Some cool projects!

None of this people … exists!!



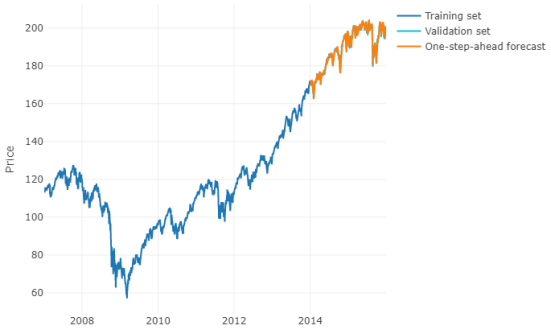


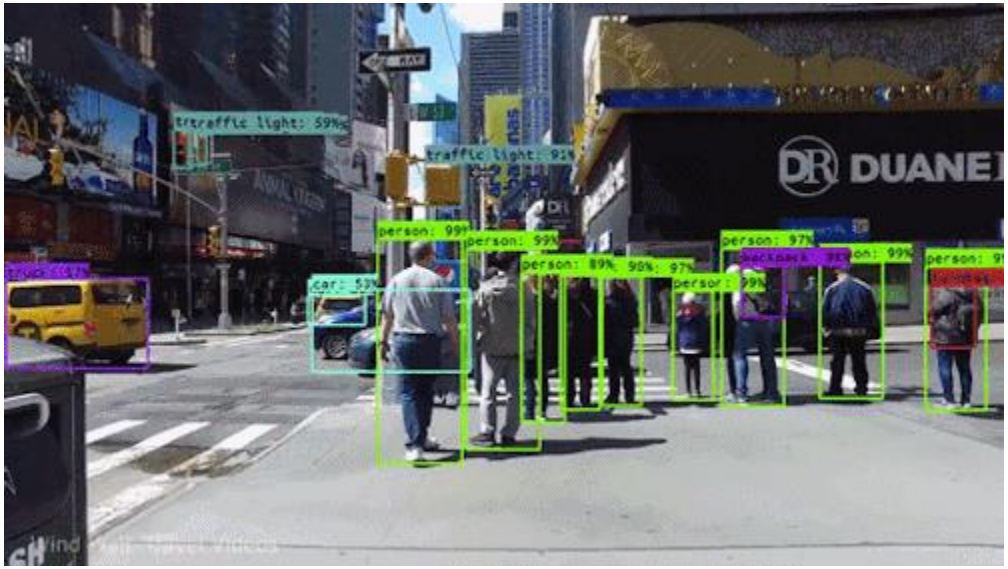

(https://www.thispersondoesnotexist.com/)

Some cool projects!

ANNs are also being used for:

- Fraud Detection
- Audio recognition
- Text-to-speech
- Text translation
- Image Classification
- Object Detection
- Time Series
- …





Our main goals

- Understand TensorFlow's dataflow graph to represent computation
- Explore TensorFlow's built-in classes and functions
- Learn how to conceive and build models best suited for connective systems

With a strong practical background!

But first … Install it!

With conda:

- conda install tensorflow

With pip:

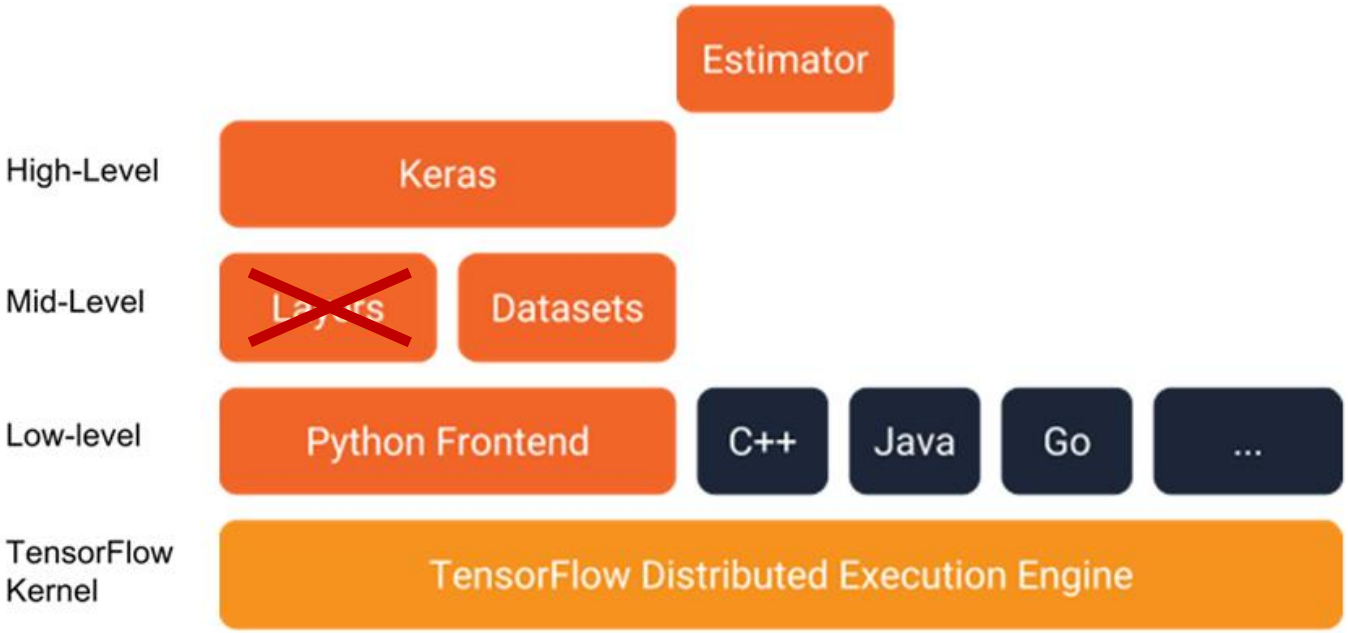
- Check it here - https://www.tensorflow.org/install/pip

~~While we wait for version 2.0 … we will be using TF v1.12.0 (released on 2 Nov 2018)!~~
TensorFlow v2.0 is here! Let's use it!

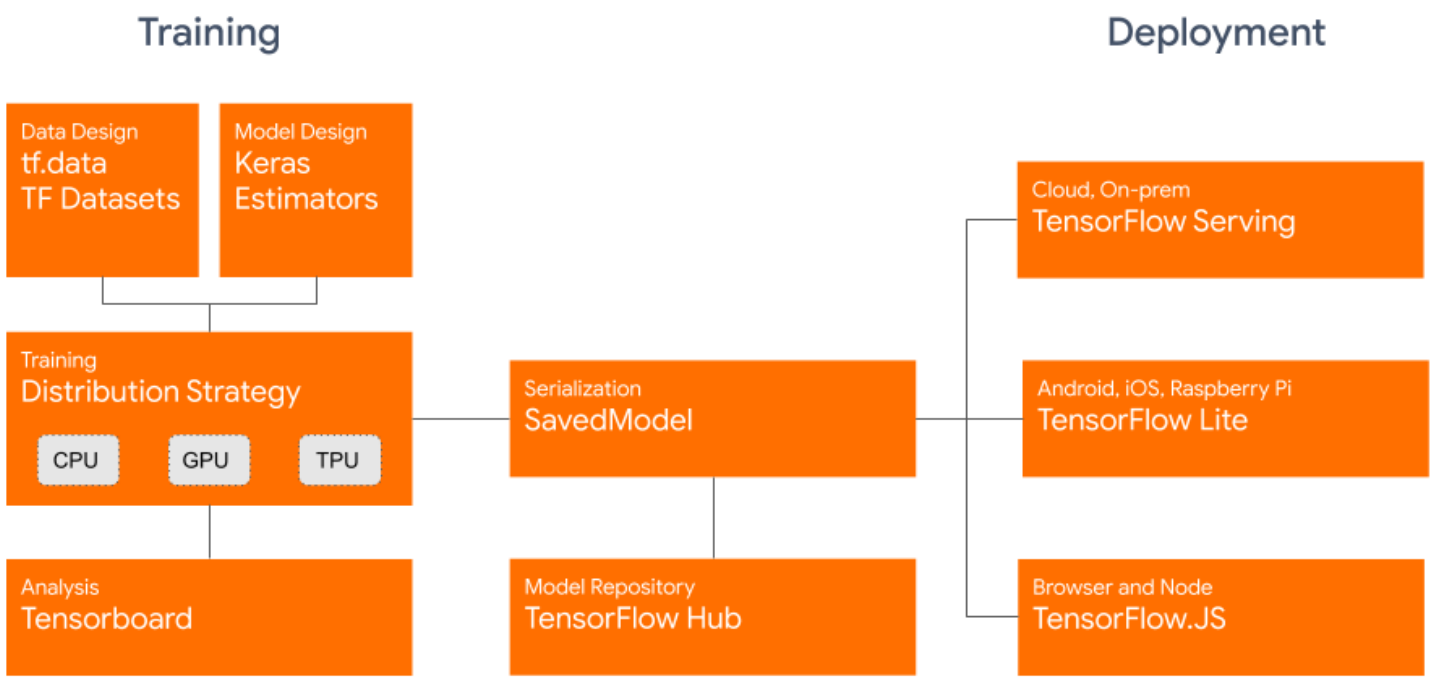
As for Python itself, I would stick with v3.7!

DO NOT work on your root environment!! Create a custom one!

The Programming Stack



The Programming Stack



Graphs and Sessions

TensorFlow separates the definition of computation from its execution!

The famous Graphs and Sessions!

As simple as it gets:

1. Assemble a Graph!
2. Use a Session to execute it!

Note: TensorFlow's eager execution model brought an alternative!

Note 2: The alternative is now the default runtime environment (for TensorFlow v2.x)!!

Graphs

TensorFlow uses a Dataflow Graph to represent computation in regard to the dependencies between individual operations.

Why? You may ask…

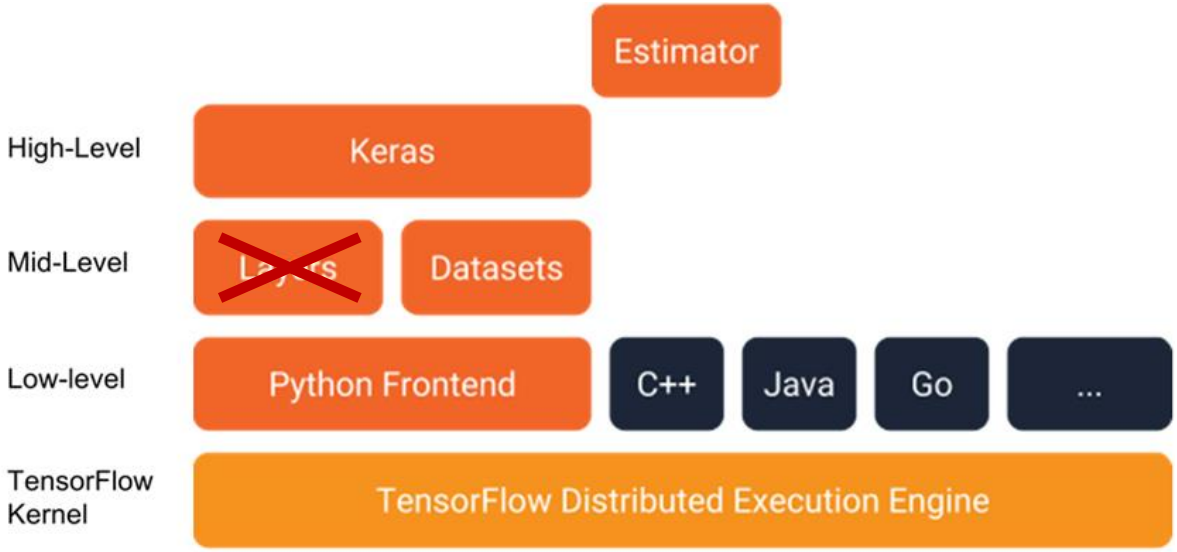
- Save computation - run subgraphs
- Break computation - into small, differential pieces
- Parallelism - by using explicit edges to represent dependencies between operations, it is easy to identify operations that can be executed in parallel
- Facilitate distributed computation - over CPUs, GPUs and TPUs
- Many common machine learning models are taught and visualized as graphs (Neural Networks!!!)

Indeed, Dataflow is a common programming paradigm for parallel computing

Graphs

This leads to a low-level programming model where you must first define the dataflow graph of computations and then create a TensorFlow session to run the graph

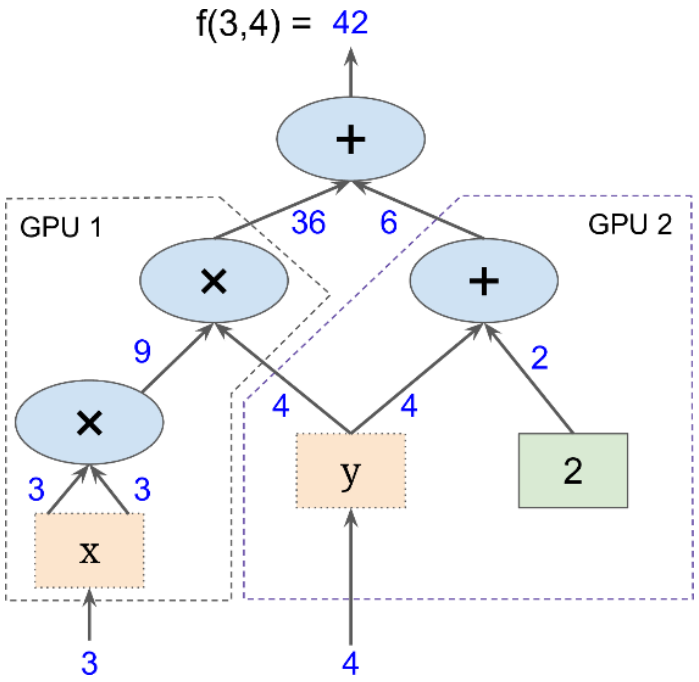
On the other hand, higher-level APIs hide the details of graphs (and sessions) from the user.



Graphs

Hence:

- Nodes represent units of computation
- Edges represent the data consumed or produced by a computation
- Data (Tensors!) flows through the graph



tf.Graph

What is a tf.Graph?

- It is a TensorFlow computation, containing a set of tf.Operation (or ops) objects (units of computation - nodes) and tf.Tensor objects (units of data - edges)

- TensorFlow provides a default graph that is an implicit argument to all API functions in a same context
  - Most programs rely solely on the default graph (however, advanced use cases may require multiple graphs)

Sessions

We have seen that, first, we are required to define the dataflow graph of computations…

- Ok!

But, then, with all computations defined…

- How to execute it?

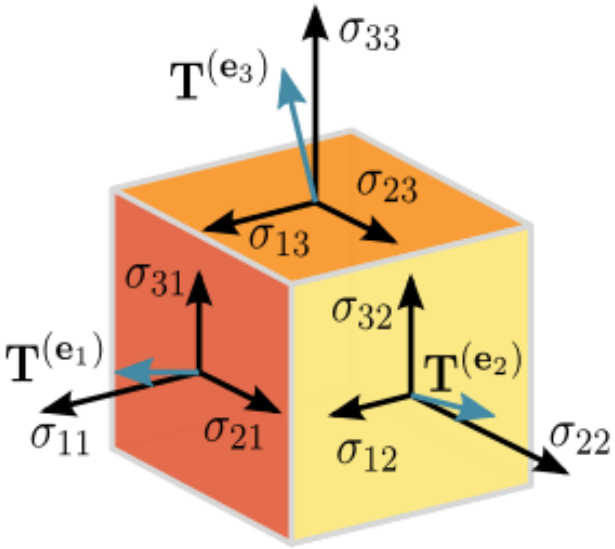
Well, you guessed it … until TF v2.x you needed to create a tf.session to run the graph!

tf.Session

What is a tf.Session?

- A Session object encapsulates the environment in which tf.Operation objects are executed and tf.Tensor objects are evaluated (a class for running TensorFlow operations)
- It will allocate memory to store the current values of variables
- It also caches information about your tf.Graph so that you can efficiently run the same computation multiple times
- tf.Session owns physical resources (such as GPUs and network connections). Hence it is typically used as a context manager (in a with block) that automatically closes the session. It is also possible to create a session with tf.Session() but you must explicitly call tf.Session.close() to free the resources!

Tensors


σ_{33}
$\mathbf{T}^{(\mathbf{e}_3)}$
σ_{23}
σ_{13}
σ_{31}
σ_{32}
$\mathbf{T}^{(\mathbf{e}_1)}$
$\mathbf{T}^{(\mathbf{e}_2)}$
σ_{11}
σ_{21}
σ_{12}
σ_{22}

Tensors

Tensors… What is that?

Just any n-dimensional array!!

0-d tensor: scalar
1-d tensor: vector
2-d tensor: matrix
and so on…

Tensor is just a fancy name for an array!

Tensors

Indeed, Tensors are data

…

Data that is flowing through the graph.

…

data + flow = tensor + flow = TensorFlow

Tensors

- More rigorously, it is the central unit of data in TensorFlow
- It consists of a set of primitive values shaped into an array of any number of dimensions
- TensorFlow uses three notational conventions to describe tensors' dimensionality:
  - Rank
  - Shape
  - Dimension number

Tensors Rank

A tensor's rank is its number of dimensions. Synonyms for rank include order or degree or n-dimension. Note that rank in TensorFlow is not the same as matrix rank in mathematics.

	Rank	Math entity
36	0	Scalar (magnitude only)
[36, 37, 38]	1	Vector (magnitude and direction)
[[36, 37], [18, 29]]	2	Matrix (table of numbers)
...	3	3-Tensor (cube of numbers)
...	n	n-Tensor (you get the idea)

Note:
tf.rank to get the rank of the Tensor

Tensors Shape

A tensor's shape is a tuple of integers specifying the array's length along each dimension. TensorFlow automatically infers shapes during graph construction. These inferred shapes might have known or unknown rank. If the rank is known, the sizes of each dimension might be known or unknown.

Rank	Shape	Dimension number	Example
0	[]	0-D	A 0-D tensor. A scalar.
1	[D0]	1-D	A 1-D tensor with shape [5].
2	[D0, D1]	2-D	A 2-D tensor with shape [3, 4].
3	[D0, D1, D2]	3-D	A 3-D tensor with shape [1, 4, 3].
n	[D0, D1, ... Dn-1]	n-D	A tensor with shape [D0, D1, ... Dn-1].

Note:
tf.shape to get the known shape of the Tensor

Tensors Shape

It is possible, and many times necessary, to change the shape of a tf.Tensor. Bear in mind that the number of elements of a tensor is the product of the sizes of all its shapes. This can be done with tf.reshape.

```
rank_three_tensor = tf.ones([3, 4, 5])                #3x4x5 (60 elements)
matrixA = tf.reshape(rank_three_tensor, [6, 10])      #Reshape into a 6x10 matrix (60 ...)
matrixB = tf.reshape(matrixA, [3, -1])                #Reshape into a 3x20 matrix (60 ...)
                                                      #-1 tells reshape to calculate the size
                                                      #of the dimension

#Note that the number of elements of the reshaped Tensors has to match the original
#number of elements. The following example generates an error
bad_matrix = tf.reshape(matrixB, [13, 2, -1])         #ERROR
```

tf.Tensor

- Tensors have a data type
- Each element in the Tensor has the same data type and the data type is always known
  - It is possible, however, to serialize data structures as strings and store them in tf.Tensors
- It is possible to cast tf.Tensors from one datatype to another using tf.cast
- The shape might be partially known
- With the exception of tf.Variable, the value of a tensor is immutable

Hello TensorFlow

Let's, now, play a bit…
But first…

TensorFlow 1.x vs 2.x

There are multiple changes in TensorFlow 2.x:

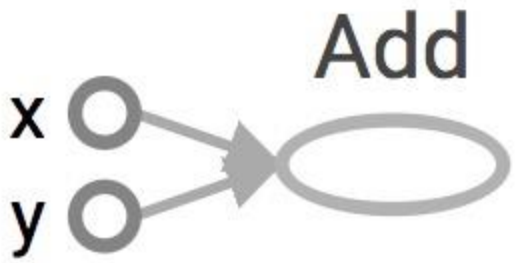
1. API Cleanup
2. Redundant APIs removed
3. APIs more consistent (Unified RNNs, Unified Optimizers)
4. Functions, not sessions (tf.function decorator)
5. Easy model building with Keras and eager execution
6. …

In fact, before v2.0 you were required to build the graph and then execute it using sessions! However, in v2.0 the default runtime mode is now Eager Execution!

But … How does Eager Execution differ from graph/session execution?

Graph/Session Execution


```
import tensorflow as tf

a = tf.add(1, 2)
```



Why x and y?

TensorFlow automatically names the nodes when they are not explicitly named

Graph/Session Execution


```
import tensorflow as tf

a = tf.add(1, 2)
print(a)
```

```
Tensor("Add:0", shape=(), dtype=int32)
```

(Not 3!)

We must first create a session and, within the session, evaluate the graph to fetch the value of a!

Graph/Session Execution


```
import tensorflow as tf

a = tf.add(1, 2)

sess = tf.Session()
print(sess.run(a))
sess.close()
```

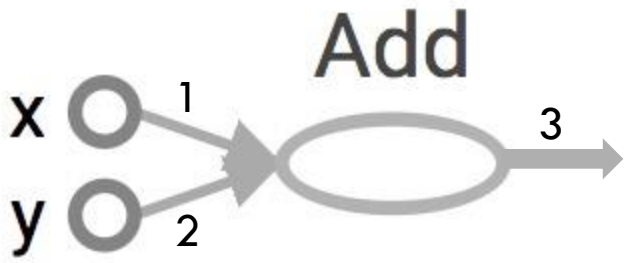
```
import tensorflow as tf

a = tf.add(1, 2)

with tf.Session() as sess:
        print(sess.run(a))
```

```
3
```

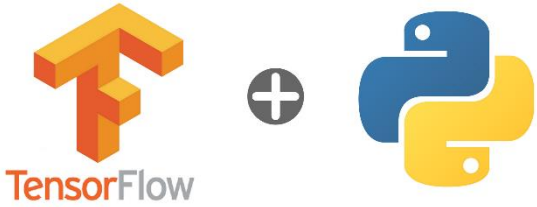
Graph/Session Execution


```
import tensorflow as tf

a = tf.add(1, 2)

sess = tf.Session()
print(sess.run(a))
sess.close()
```

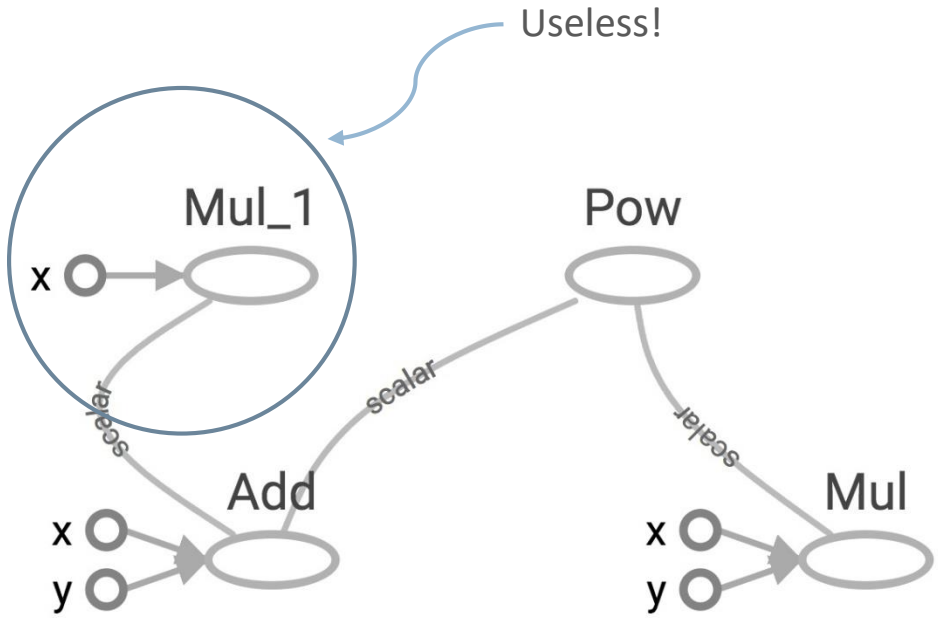
```
import tensorflow as tf

a = tf.add(1, 2)

with tf.Session() as sess:
        print(sess.run(a))
```



Graph/Session Execution




```
import tensorflow as tf

x = 2
y = 3

add_op = tf.add(x, y)
mul_op = tf.multiply(x, y)
useless = tf.multiply(x, add_op)
p_op = tf.pow(add_op, mul_op)

with tf.Session() as sess:
        z = sess.run(p_op)
```



Because we only want the value of p_op, and since p_op does not depend on useless, session won't compute the value of useless → save computation!

Graph/Session Execution

Error prone!!!

What about if we want to build more than one graph?

In fact, we can… But, as we saw previously, we usually don't need more than one graph as session runs the default graph!

But what if we really want to??

```
#create a graph
g = tf.Graph()

#to add operators, set it as default
with g.as_default():
        x = tf.add(2, 3)

sess = tf.Session(graph=g)
with tf.Session() as sess:
        sess.run(x)
```

```
#handle the default graph
g_def = tf.get_default_graph()
g_user = tf.Graph()

#add ops to the default one
with g_def.as_default():
        a = tf.Constant(2)
#add ops to user created graph
with g_user.as_default():
        b = tf.Constant(3)
```

Graph/Session Execution

- Multiple graphs are usually not useful as we cannot merge them and pass tensors between them

- Moreover, as we saw previously, the default graph can contain several models as disconnected sub-graphs. When we ask for a tensor in tf.Session.run(), TensorFlow will only run the necessary to compute the tensor we asked for. So, other sub-graphs will not run
  - However, if our program creates a large number of unconnected subgraphs, it then may become more efficient to use a different tf.Graph to build each subgraph, so that unrelated state can be garbage collected

Eager execution

Eager execution

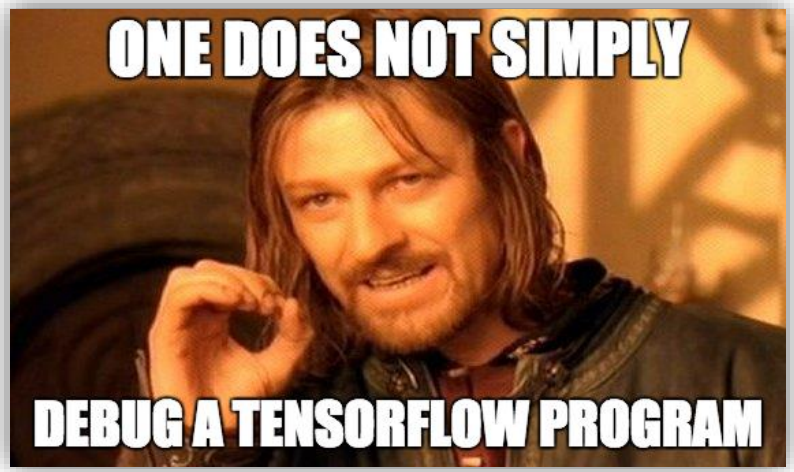
TensorFlow now presents us with an imperative programming environment that evaluates operations immediately, without building graphs! Operations return concrete values instead of constructing a computational graph to run later.

Indeed, graphs are:

- Optimizable
- Parallelizable
- Deployable
- Rewritable

However, they also are:

- Difficult to debug
- Un-Pythonic



Eager execution

With Eager execution:

- Operations return concrete values
- Debugging models is simpler
- It helps reducing boilerplate as well

Enabling eager execution changes the way TensorFlow operations behave, i.e., now they immediately evaluate and return their values to Python!

Eager execution

Graph/Session Execution:

```
import tensorflow as tf

a = tf.add(1, 2)
print(a)
```

```
Tensor("Add:0", shape=(), dtype=int32)
```

Eager Execution:

```
import tensorflow as tf

a = tf.add(1, 2)
print(a)
```

```
tf.Tensor(3, shape=(), dtype=int32)
```

Eager execution

Is it executing eagerly?

```
import tensorflow as tf

print('TF version: ', tf.__version__)
print(tf.executing_eagerly())

#ops on the fly
i = tf.constant(0)
while i < 20:
        i = tf.add(i, 1)
        print('Feels good %d' %i, 'times')
        print('Feels good ', i, 'times')
```

```
TF version: 2.0.0
True

Feels good 1 times
Feels good  tf.Tensor(1, shape=(), dtype=int32) times
Feels good 2 times
Feels good  tf.Tensor(2, shape=(), dtype=int32) times
…
```

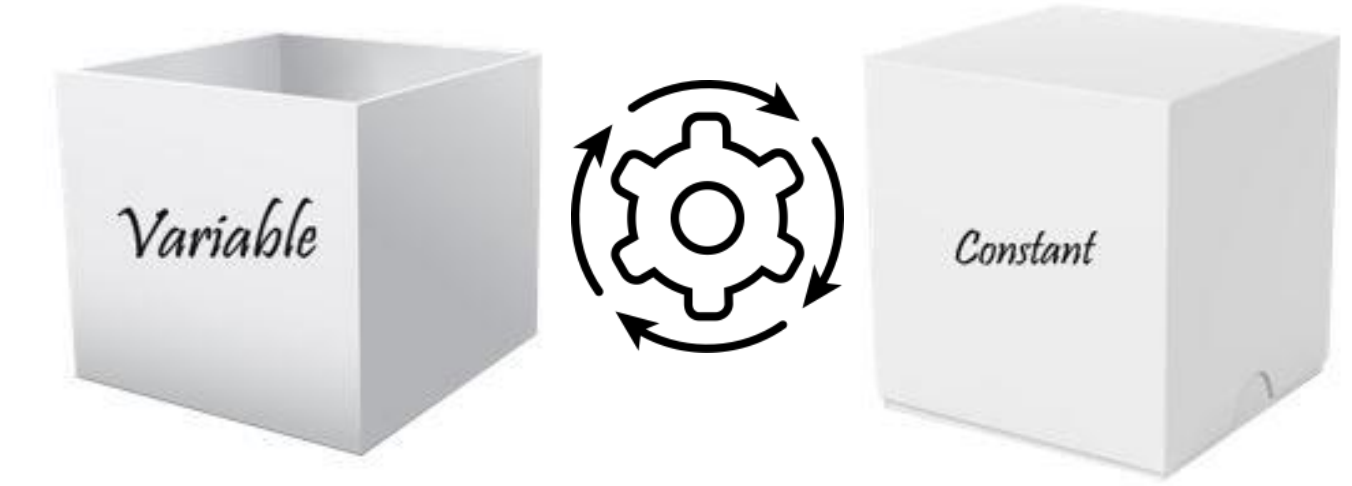
Eager execution

There is more to Eager Execution than this… We will get to it sooner.

For now just remember the difference between these two modes of execution.

Since we will be using TensorFlow 2.0, we will now focus only on Eager Execution.

Operations (ops), Constants and Variables



Constants


```
import tensorflow as tf

a = tf.constant([6, 7], name='a')
b = tf.constant([[1,2], [3,4]], name='b')
x = tf.multiply(a, b, name='mult')

print(x)
```

```
tf.constant(
        value,
        dtype=None,
        shape=None,
        name='Const'
)
```

»

```
tf.Tensor(
[[ 6 14]
 [18 28]], shape=(2, 2), dtype=int32)
```

Note: Broadcasting similar to numpy (i.e., arrays of different shapes - the smaller array is "broadcasted" across the larger array so that they have compatible shapes)

Constants as sequences


```
#creates a tensor of shape with all elements as 0
#tf.zeros(shape, dtype=tf.float32, name=None)
#similar to numpy.zeros
print(tf.zeros([3, 4], tf.int32))

#creates a tensor of shape and type as the
#input_tensor but all elements are zeros
#similar to numpy.zeros_like
tensor = tf.constant([[1, 2, 3], [4, 5, 6]])
print(tf.zeros_like(tensor))

#if instead of 0 you want 1, replace function zeros
#by ones. Similar to numpy.ones and numpy.ones_like
#tf.ones(...) and tf.ones_like(...)
print(tf.ones_like(tensor))
```

```
tf.Tensor(
[[0 0 0 0]
 [0 0 0 0]
 [0 0 0 0]], shape=(3, 4), dtype=int32)

tf.Tensor(
[[0 0 0]
 [0 0 0]], shape=(2, 3), dtype=int32)

tf.Tensor(
[[1 1 1]
 [1 1 1]], shape=(2, 3), dtype=int32)
```

Constants as sequences


```
#creates a tensor filled with a scalar value
#tf.fill(dims, value, name=None)
#similar to numpy.full
print(tf.fill([2, 3], 8))

#create a sequence of num evenly-spaced vals
#tf.linspace(start, stop, num, name=None)
print(tf.linspace(10.0, 13.0, 4))

#create a sequence that begins at start going
#delta up to but not including limit
print(tf.range(3, 18, 3) )
print(tf.range(5))
```

```
tf.Tensor(
[[8 8 8]
 [8 8 8]], shape=(2, 3), dtype=int32)

tf.Tensor([10. 11. 12. 13.],
    shape=(4,), dtype=float32)

tf.Tensor([3  6  9 12 15],
    shape=(5,), dtype=int32)

tf.Tensor([0 1 2 3 4],
    shape=(5,), dtype=int32)
```

Randomly Generated


```
#Randomly Generated Constants
#tf.random.normal
#tf.random.uniform
#tf.random.shuffle
#tf.random.gamma
#tf.random.set_seed <- important

print((tf.random.normal([2, 2]))
```

```
tf.Tensor(
[[-0.23003818 -0.9997286 ]
 [ 1.0731778   0.79688025]],
shape=(2, 2), dtype=float32)
```

Constants

Q.: What is the problem with constants?
A.: Constants are stored in the graph definition!

When constants are memory intensive, such as a matrixes with millions of entries, loading the graph becomes expensive!

Tip:

- Use constants for primitive types
- Use variables or readers for data that requires more memory

The graph definition is stored in a protobuf (protocol buffers, i.e., Google's language/platform-neutral mechanism for serializing structured data - think of XML, but smaller, faster and simpler)

TensorFlow Operations

Basic TF operations include:

```
#Ops
tf.unique
tf.where
tf.gather
tf.sort
tf.slice
tf.is_tensor
…
```

Note:

- Read the docs if doubts about an operation
- https://www.tensorflow.org/api_docs/python/tf

TensorFlow Math Ops

Quite standard… Similar to numpy.

```
#Arithmetic Ops
tf.math.abs
tf.math.negative
tf.math.sign
tf.math.reciprocal
tf.math.square
tf.math.equal
tf.math.greater
tf.math.sqrt
tf.math.pow
tf.math.exp
```

Note:

- Read the docs if doubts about an operation
- https://www.tensorflow.org/api_docs/python/tf/math

TensorFlow Div Ops


```
a = tf.constant([2, 2], name='a')
b = tf.constant([[0, 1], [2, 3]], name='b')

print(tf.math.divide(b, a))
print(tf.math.truediv(b, a))
print(tf.math.floordiv(b, a))
```

```
tf.Tensor(
[[0.  0.5]
 [1.  1.5]], shape=(2, 2), dtype=float64)

tf.Tensor(
[[0.  0.5]
 [1.  1.5]], shape=(2, 2), dtype=float64)

tf.Tensor(
[[0 0]
 [1 1]], shape=(2, 2), dtype=int32)
```

Note:

- Read the docs if doubts about an operation
- https://www.tensorflow.org/api_docs/python/tf/math

Dot product

To do the dot product in TensorFlow, we use tf.tensordot

```
#Dot product
a = tf.constant([10, 20], name='a')
b = tf.constant([2, 3], name='b')

print(tf.multiply(a, b))
print(tf.tensordot(a, b, 1))
```

```
tf.Tensor([20 60], shape=(2,), dtype=int32)

tf.Tensor(80, shape=(), dtype=int32)
```

TensorFlow Data Types


```
t_0 = 19
print(tf.zeros_like(t_0))
print(tf.ones_like(t_0))

t_1 = [1, 2, 3]
print(tf.zeros_like(t_1))
print(tf.ones_like(t_1))

t_2 = [[True, False],
       [False, False]]
print(tf.zeros_like(t_2))
print(tf.ones_like(t_2))
```

```
#scalars are treated as 0d tensors
tf.Tensor(0, shape=(), dtype=int32)
tf.Tensor(1, shape=(), dtype=int32)

#1d arrays are treated as 1d tensors
tf.Tensor([0. 0. 0.], shape=(3,), dtype=float32)
tf.Tensor([1. 1. 1.], shape=(3,), dtype=float32)

#2d arrays are treated as 2d tensors
tf.Tensor(
[[False False]
 [False False]], shape=(2, 2), dtype=bool)
tf.Tensor(
[[ True  True]
 [ True  True]], shape=(2, 2), dtype=bool)
```

Note:

- Scalars are converted to 0-d tensors, lists to 1d tensors (vectors), and so on
- https://www.tensorflow.org/api_docs/python/tf/dtypes/DType

TensorFlow Data Types

Use TensorFlow DType whenever possible!

- TensorFlow has to infer Python native types
  - Python lacks the ability to explicitly state the data type. For example, all integers are the same type, but TensorFlow has 8-bit, 16-bit, 32-bit and 64-bit integers available
- NumPy arrays issue
  - NumPy is not GPU compatible
- You may create hand-defined Tensor objects as NumPy arrays

NumPy Integration

- NumPy operations accept tf.Tensor arguments
- TensorFlow math operations convert Python objects and NumPy arrays to tf.Tensor objects

```
#a constant
a = tf.constant([[1, 2], [3, 4]])
print(a)

#broadcasting support
b = tf.add(a, 1)
print(b)

#operator overloading is supported
print(a * b)
```

```
tf.Tensor(
      [[1 2]
       [3 4]], shape=(2, 2), dtype=int32)

tf.Tensor(
      [[2 3]
       [4 5]], shape=(2, 2), dtype=int32)

tf.Tensor(
      [[2 6]
       [12 20]], shape=(2, 2), dtype=int32)
```

NumPy Integration

- NumPy operations accept tf.Tensor arguments
- TensorFlow math operations convert Python objects and NumPy arrays to tf.Tensor objects
- tf.Tensor.numpy returns the object's value as a NumPy ndarray
- Tensors are iterable (use tf.equal to compare Tensors)

```
#using numpy
import numpy as np

#a and b from previous slide
c = np.multiply(a, b)
print(c)

#obtain numpy value from a tensor
print(a.numpy())

#iterable
for i in tf.constant([1, 2, 3]) :
    print(i)
```

```
[[ 2  6]
 [12 20]]


[[1 2]
 [3 4]]


tf.Tensor(1, shape=(), dtype=int32)
tf.Tensor(2, shape=(), dtype=int32)
tf.Tensor(3, shape=(), dtype=int32)
```

Variables

A TensorFlow variable is the best way to represent shared, persistent state manipulated by your program.

Internally, a tf.Variable stores a persistent tensor.

Q.: But why tf.constant but tf.Variable? (lowercase c and uppercase V)

Because tf.constant is an op.
tf.Variable is a class with many ops!

```
x.assign()          #write op
x.assign_add()      #outputs tensor after the update is done
x.assign_sub()      #outputs tensor after the update is done
x.read_value()      #outputs tensor after the update is done
#and more ...
x.initializer       #only required if not in eager execution!
```

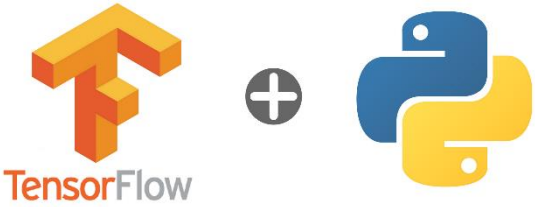
Variables


```
#create variables with tf.Variable
s = tf.Variable(2, dtype=tf.int32, name="a_scal_var")
print(s)

m = tf.Variable([[1, 2], [3, 4]], name="my_matrix")
print(m)

#multiply m by s
print(tf.multiply(m,s))

w = tf.Variable(tf.zeros([2, 3]))
print(w)

non_trainable = tf.Variable(2., trainable=False)
```

```
<tf.Variable 'a_scal_var:0' shape=()
dtype=int32, numpy=2>

<tf.Variable 'my_matrix:0' shape=(2, 2)
dtype=int32, numpy=
array([[1, 2],
       [3, 4]])>

tf.Tensor(
    [[2 4]
     [6 8]], shape=(2, 2), dtype=int32)

<tf.Variable 'Variable:0' shape=(2, 3)
dtype=float32, numpy=
array([[0., 0., 0.],
          [0., 0., 0.]], dtype=float32)>
```

Note: The Variable() constructor requires an initial value for the variable, which can be a Tensor of any type and shape. This initial value defines the type and shape of the variable. After construction, the type and shape of the variable are fixed. The value can be changed using one of the assign methods.

Variables




```
s = tf.Variable(2, dtype=tf.int32, name="a_scal_var")

#assign a new value
s.assign(10)
print(s)
print(s.value())
print(s.numpy())
print(s.device)
print(s.dtype)

#add first and then assign a new value
s.assign_add(10)
print(s)
```

»

```
#print(s) gives the Variable
<tf.Variable 'a_scal_var:0' shape=()
dtype=int32, numpy=10>

#print(s.value()) gives the Tensor
tf.Tensor(10, shape=(), dtype=int32)

#print(s.numpy()) gives the value
10

#print(s.device) gives the device hosting the var
/job:localhost/replica:0/task:0/device:CPU:0

#print(s.dtype) gives the dtype of the var
<dtype: 'int32'>

#add first, then assign
<tf.Variable 'a_scal_var:0' shape=()
dtype=int32, numpy=20>
```

An exercise

The FizzBuzz example! Run it eagerly!

```
import tensorflow as tf
import numpy as np

#tf version & is eager execution enabled
print('TF version: ', tf.__version__)
print(tf.executing_eagerly())

def fizzbuzz(limit):
    print('Is limit a tensor? %s' %tf.is_tensor(limit))
    if(not tf.is_tensor(limit)):
        limit = tf.convert_to_tensor(limit)
        print('Is it a tensor now? %s' %tf.is_tensor(limit))

    for i in tf.range(1, limit+1):
        #TODO
        pass

fizzbuzz(tf.constant(15))
```

HANDS ON

```
1
2
Fizz
4
Buzz
Fizz
7
8
Fizz
Buzz
11
Fizz
13
14
FizzBuzz
```

Controlling randomness

Seed random variables

Controlling randomness is crucial to get stable and replicable results!

Operations that rely on a random seed actually derive it from two seeds:

1. The global seed
2. The operation-level seed

```
#Global Seed
tf.random.set_seed(1234)

rd1 = tf.random.uniform([], -1, 1)
rd2 = tf.random.uniform([], -1, 1)
```

```
#Operation-level seed


rd3 = tf.random.uniform([], -1, 1, seed=1)
rd4 = tf.random.uniform([], -1, 1, seed=1)
```

Without seeds


```
#Without seeds

print(tf.random.uniform([1]).numpy())
print(tf.random.uniform([1]).numpy())
```

```
[0.2962978]
[0.5523262]

#closing the program and then running it again
[0.17070568]
[0.0832268]

#You get it…

#We get different results for every call to the op
#and every re-run of the program.
```

Global seed


```
#Global Seed but no operation-level seed
tf.random.set_seed(1234)

print(tf.random.uniform([1]).numpy())
print(tf.random.uniform([1]).numpy())
```

```
[0.5380393]
[0.3253647]

#closing the program and then running it again
[0.5380393]
[0.3253647]

#We get different results for every call to the op,
#but the same sequence for every re-run of the
#program.
```

Note: The reason we get 0.32 instead of 0.53 on the second call of *tf.random.uniform* above is because the second call uses a different operation seed.

Operation-level seed


```
#Operation-level Seed but no global seed

print(tf.random.uniform([1], seed=12).numpy())
print(tf.random.uniform([1], seed=12).numpy())
```

```
[0.6361525]
[0.13028836]

#closing the program and then running it again
[0.6361525]
[0.13028836]

#We get different results for every call to the op,
#but the same sequence for every re-run of the
#program.
```

Note: The reason we get 0.13 instead of 0.63 on the second call is because the same *tf.random.uniform* kernel is used by TF for all calls of it with the same arguments, and the kernel maintains an internal counter which is incremented every time it is executed, generating different results.

Operation and Global seed


```
#Operation-level Seed and global seed

tf.random.set_seed(1234)
print(tf.random.uniform([1], seed=12).numpy())
print(tf.random.uniform([1], seed=12).numpy())

print('Reset Kernel Counters')

tf.random.set_seed(1234)
print(tf.random.uniform([1], seed=12).numpy())
print(tf.random.uniform([1], seed=12).numpy())
```

```
[0.01017106]
[0.7369207]
Reset Kernel Counters
[0.01017106]
[0.7369207]

#closing the program and then running it again
[0.01017106]
[0.7369207]
Reset Kernel Counters
[0.01017106]
[0.7369207]
```

Note: Calling *tf.random.set_seed* will reset any such counters.

Operation-level seed


```
@tf.function
def foo():
  a = tf.random.uniform([1], seed=1)
  b = tf.random.uniform([1], seed=1)
  return a, b

print(foo())
print(foo())
```

```
(0.2390374, 0.2390374)
(0.22267115, 0.22267115)

#closing the program and then running it again
(0.2390374, 0.2390374)
(0.22267115, 0.22267115)
```

Note: However, when multiple identical random ops are wrapped in a *tf.function*, their behaviors change because the ops no longer share the same counter.

Seed random variables

Controlling randomness is crucial to get stable and replicable results!

Take note of how the two seeds interact:

1. If neither the global seed nor the operation seed is set:
   - A randomly picked seed is used for the op

2. If the global seed is set but the operation seed is not set:
   - The system picks an operation seed from a stream of seeds determined by the global seed

3. If the operation seed is set, but the global seed is not set:
   - A default global seed and the specified operation seed are used to determine the sequence

4. If both the global and the operation seed are set:
   - Both seeds are used in conjunction to determine the sequence

A Structured Model

An example on how to structure your TensorFlow model!

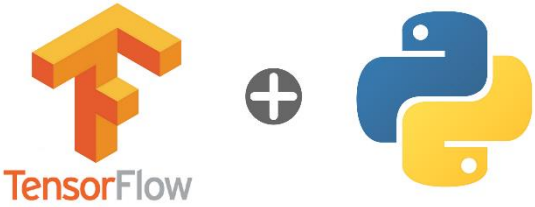
A Structured Model

Q.: How to make our model easily reusable?

A.: Let's be object oriented!

```
class OurModel:
    '''Constructor - setting the model variables'''
    def __init__(self, args):
        pass

    '''Load the data'''
    def import_data(self):
        pass

    '''Define the model, the weights and the bias'''
    def model_def(self):
        pass

    '''Train, compute loss and optimize'''
    def fit(self):
        pass

    '''Predict obs value'''
    def predict(self, obs):
        pass
```

A Structured Model



Then:

First, assemble the graph!

1. Import data/placeholders
2. Define the weights and bias
3. Define functions
4. Define optimizer

Second, compute it!

1. Initialize the variables
2. Feed training data
3. Calculate the output of each training input
4. Calculate the cost
5. Adjust the model parameters

Now:

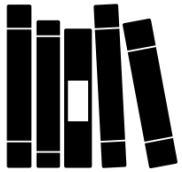
Just build and run it eagerly!!!

Glossary

- Eager Execution: a TF programming environment in which operations run immediately. By contrast, operations called in graph execution don't run until they are explicitly evaluated. Eager execution programs are generally far easier to debug than graph execution ones.
- Global and Operation-level Seeds: Controlling randomness to get stable and replicable results.
- Graph: a computation specification. Nodes in the graph represent operations. Edges are directed and represent passing the result of an operation (a Tensor) as an operand to another operation.
- Rank: number of dimensions in a Tensor. For instance, a scalar has rank 0, a vector has rank 1, and a matrix has rank 2.
- Shape: number of elements in each dimension of a tensor. The shape is represented as a list of integers. In a 2-d TF Tensor, the shape is [*number of rows*, *number of columns*].
- Tensor: the primary data structure in TF programs. Tensors are N-dimensional data structures, most commonly scalars, vectors, or matrices. The elements of a Tensor can hold integer, floating-point or string values.

(From: https://developers.google.com/machine-learning/glossary)

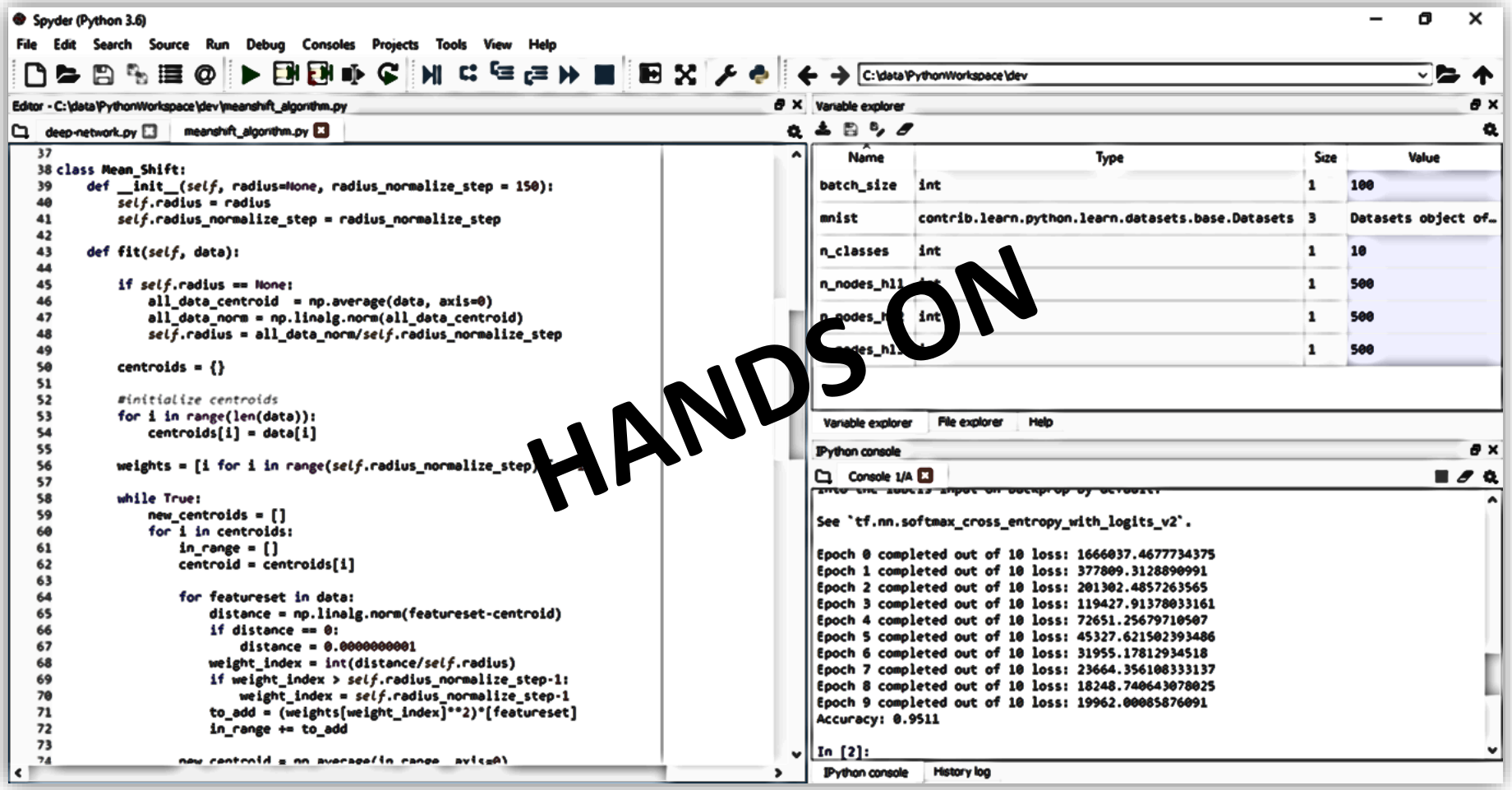
Resources



- Official Documentation
  - https://www.tensorflow.org/api_docs/
  - https://www.tensorflow.org/tutorials/quickstart/beginner
  - https://www.tensorflow.org/guide/tensor
  - …

- Papers, Books, online courses, tutorials…
  - TensorFlow for Deep Learning by Ramsundar, B. & Zadeh, R. (2017)
  - TensorFlow: Getting Started by Kurata, J. (2017)
  - TensorFlow for Deep Learning Research by Huyen, C. et al. (2017)

Hands On


Spyder (Python 3.6)
File Edit Search Source Run Debug Consoles Projects Tools View Help
C:\data\PythonWorkspace\dev
Editor - C:\data\PythonWorkspace\dev\meanshift_algorithm.py
deep-network.py
meanshift_algorithm.py
class Mean_Shift:
def __init__(self, radius=None, radius_normalize_step = 150):
self.radius = radius
self.radius_normalize_step = radius_normalize_step
def fit(self, data):
if self.radius == None:
all_data_centroid = np.average(data, axis=0)
all_data_norm = np.linalg.norm(all_data_centroid)
self.radius = all_data_norm/self.radius_normalize_step
centroids = {}
#initialize centroids
for i in range(len(data)):
centroids[i] = data[i]
weights = [i for i in range(self.radius_normalize_step)]
while True:
new_centroids = []
for i in centroids:
in_range = []
centroid = centroids[i]
for featureset in data:
distance = np.linalg.norm(featureset-centroid)
if distance == 0:
distance = 0.0000000001
weight_index = int(distance/self.radius)
if weight_index > self.radius_normalize_step-1:
weight_index = self.radius_normalize_step-1
to_add = (weights[weight_index]**2)*[featureset]
in_range += to_add
Variable explorer
Name Type Size Value
batch_size int 1 100
mnist contrib.learn.python.learn.datasets.base.Datasets 3 Datasets object of…
n_classes int 1 10
n_nodes_hl1 1 500
n_nodes_hl2 int 1 500
n_nodes_hl3 1 500
Variable explorer File explorer Help
IPython console
Console 1/A
See `tf.nn.softmax_cross_entropy_with_logits_v2`.
Epoch 0 completed out of 10 loss: 1666037.4677734375
Epoch 1 completed out of 10 loss: 377809.3128890991
Epoch 2 completed out of 10 loss: 201302.4857263565
Epoch 3 completed out of 10 loss: 119427.91378033161
Epoch 4 completed out of 10 loss: 72651.25679710507
Epoch 5 completed out of 10 loss: 45327.621502393486
Epoch 6 completed out of 10 loss: 31955.17812934518
Epoch 7 completed out of 10 loss: 23664.356108333137
Epoch 8 completed out of 10 loss: 18248.740643078025
Epoch 9 completed out of 10 loss: 19962.00085876091
Accuracy: 0.9511
In [2]:
IPython console History log
HANDS ON