TensorFlow

Open source symbolic math library
Popular in ML, especially for neural networks

Developed by GoogleBrain
Google’s AI/Deep learning division
You may recall their major computer vision triumph circa 2014:

TensorFlow is not new, and not very special:
Many other symbolic math programs predate it!
**TensorFlow is unique in how quickly it gained so much marketshare**
Open-sourced only in 2015…
...and almost immediately became the dominant framework for NNs
TensorFlow: Installation

Easiest: `pip install tensorflow`

Also easy: install in anaconda

More information: [https://www.tensorflow.org/install/](https://www.tensorflow.org/install/)

**Note:** if you want to do fancier things (e.g., run on GPU instead of CPU), installation and setup gets a lot harder. For this course, we’re not going to worry about it. In general, for running on a GPU, if you don’t have access to a cluster with existing TF installation, you should consider paying for Amazon/GoogleCloud instances.
Fundamental concepts of TensorFlow

Tensor
Recall that a tensor is really just an array of numbers
“Rank” of a tensor is the number of dimensions it has
So, a matrix is a rank-2 tensor, vector is rank 1, scalar rank 0
A cube of numbers is a 3-tensor, and so on

Computational graph
Directed graph that captures the “flow” of data through the program
Nodes are operations (i.e., computations)
Edges represent data sent between operations
Note: most things you read will call this dimension the *rank* of the tensor, but you should know that some mathematicians use *rank* to mean the tensor generalization of linear algebraic rank. These people will usually use the term *order* instead.
Tensors: `tf.Tensor` objects

Tensors are represented in TensorFlow as `tf.Tensor` objects

Every `tf.Tensor` object has:
- data type (e.g., int, float, string, …)
- shape (e.g., 2-by-3-by-5, 5-by-5, 1-by-1, etc)

Shape encodes both rank and ‘length’ of each dimension

`tf.Tensor` objects are, in general, immutable

with slight exceptions, which we’ll talk about soon
**Special `tf.Tensor()` objects**

`tf.Constant`: will not change its value during your program.
   Like an immutable tensor

`tf.Placeholder`: gets its value from elsewhere in your program
   E.g., from training data or from results of other Tensor computations

`tf.Variable`: represents a tensor whose value may change during execution
   Unlike above `tf.Tensor` types, `tf.Variables` are mutable
   Useful for ML, because we want to update parameters during training

`tf.SparseTensor`: most entries of a SparseTensor will be zero
   TF stores this differently; saves on memory
   Useful for applications where data is sparse, such as networks
Special `tf.Tensor()` objects

**tf.constant:** will not change its value during your program.
- Like an immutable tensor

**tf.placeholder:** gets its value from elsewhere in your program
- E.g., from training data or from results of other Tensor computations

**tf.variable:** represents a tensor whose value may change during execution
- Unlike above `tf.Tensor` types, `tf.Variable` are **mutable**
- Useful for ML, because we want to update parameters during training

**tf.sparse_tensor:** most entries of a `SparseTensor` will be zero
- TF stores this differently; saves on memory
- Useful for applications where data is sparse, such as networks

For now, these three are the important ones.
Computational Graph

From the “Getting Started” guide: “A computational graph is a series of TensorFlow operations arranged into a graph of nodes.”

Every node takes zero or more tensors as input and outputs a tensor.

A TensorFlow program consists, essentially, of two sections:

1) Building the computational graph
2) Running the computational graph

An example of a computational graph that represents the computation \( z = a \times x + b \).
TF as Dataflow

Dataflow is a term for frameworks in which computation is concerned with the pipeline by which the data is processed. Data transformed and combined via a series of operations. This view makes it clear when parallelization is possible… ...because dependence between operations can be read off the graph. 

https://en.wikipedia.org/wiki/Dataflow
https://en.wikipedia.org/wiki/Stream_processing

This should sound familiar from PySpark!
Building the Computational Graph

Here's a snippet of a TF program, in which we define a computational graph.

```python
# Now we define some variables
da = tf.constant(2, dtype=tf.float32)
b = tf.constant(1, dtype=tf.float32)
x = tf.placeholder(tf.float32)
z = a * x + b
```

Equivalent computational graph:
Building the Computational Graph

Here’s a snippet of a TF program, in which we define a computational graph.

```python
# Now we define some variables
a = tf.constant(2, dtype=tf.float32)
b = tf.constant(1, dtype=tf.float32)
x = tf.placeholder(tf.float32)
z = a*x + b
```

**tf.constant** is a TF tensor whose value will not change. This is the TF analogue of an immutable type.

**tf.placeholder** is a TF tensor whose value will be assigned at runtime, after building the graph.

Equivalent computational graph:
Building the Computational Graph

```python
1 sess = tf.Session()
2 a = tf.constant(2, dtype=tf.float32)
3 x = tf.placeholder(tf.float32)
4 adder_node1 = a + x
5 adder_node2 = tf.add(a, x)
6 print(adder_node1)
7 print(adder_node2)

Tensor("add_22:0", dtype=float32)
Tensor("Add_4:0", dtype=float32)
```

+ Is just short for the `tf.add()` function.

Similarly, * is short for the `tf.multiply()` function.
Building the Computational Graph

```python
sess = tf.Session()
a = tf.constant(2, dtype=tf.float32)
b = tf.constant(1, dtype=tf.float32)
x = tf.placeholder(tf.float32)
z = a*x + b

print(a)
print(x)
print(z)
```

Tensor("Const_34:0", shape=(), dtype=float32)
Tensor("Placeholder_19:0", dtype=float32)
Tensor("add_17:0", dtype=float32)

```python
print(sess.run(z, {x: 4}))
print(sess.run(z, {x: 5}))
```

9.0
11.0

These are all `tf.Tensor` objects.

Variables don’t have values until you run the graph!
Running TensorFlow

```python
import tensorflow as tf

# Before we can actually do anything, # we have to start a session.
sess = tf.Session()

# Now we define some variables
a = tf.constant(2, dtype=tf.float32)
b = tf.constant(1, dtype=tf.float32)
x = tf.placeholder(tf.float32)

z = a*x + b

# Run the code, print the result.
print(sess.run(z, {x: 4}))

# Close the session
sess.close()
```

Operations are defined here, but we still haven’t actually computed anything, yet...

Computation only carried out once we give a value to `x` and ask TF to run the graph.
Data types in TensorFlow

Every `tf.Tensor()` object has a data type, accessed through the `dtype` attribute.

```python
helloworld = tf.constant('hello world!')
print(helloworld.dtype)
ramanujan = tf.constant(1729, dtype=tf.int16)
print(ramanujan.dtype)
approxpi = tf.constant(3.14159, dtype=tf.float32)
print(approxpi.dtype)
imaginary = tf.constant((0.0, 1.0), dtype=tf.complex64)
print(imaginary.dtype)
```

**Four basic data types:**
- Strings
- Integers
- Floats
- Complex numbers

Some flexibility in specifying precision

**Note:** if no `dtype` is specified, TF will do its best to figure it out from context, but this doesn’t always go as expected, such as when you want a vector of complex numbers. When in doubt, specify!
Creating Tensors

These are all rank-0 tensors. Yes, `tf.string` is a single item, and so is `tf.complex`.

To create a 1-tensor (i.e., a vector), just pass a list of scalars.

```
1 helloworld = tf.constant('hello world!', dtype=tf.string)
2 ramanujan = tf.constant(1729, dtype=tf.int16)
3 approxpi = tf.constant(3.14159, dtype=tf.float32)
4 imaginary = tf.constant((0.0, 1.0), dtype=tf.complex64)
```

```
1 fibonacci = tf.constant([0, 1, 1, 2, 3, 5, 8, 13, 21], dtype=tf.int8)
2 animals = tf.constant(['dog', 'cat', 'bird', 'goat'], dtype=tf.string)
3 print animals
```

```
Tensor("Const_143:0", shape=(4,), dtype=string)
```

**Note:** all elements of a `tf.Tensor` must be of the same datatype. The one sneaky way around this is to serialize objects to strings and store them in a tensor with `dtype=tf.string`. 
Creating Tensors

We can create a 1-by-1 matrix, which is **different** from a 1-vector, which is different from a scalar.

```python
onebyonemx = tf.constant([[3.1415]], dtype=tf.float32)
print(onebyonemx)
Tensor("Const_170:0", shape=(1, 1), dtype=float32)

onevec = tf.constant([3.1415], dtype=tf.float32)
print(onevec)
Tensor("Const_171:0", shape=(1,), dtype=float32)

scalar = tf.constant(3.1415, dtype=tf.float32)
print(scalar)
Tensor("Const_172:0", shape=(), dtype=float32)
```
Creating Tensors

To create a matrix, we can pass a list of its rows.

Matrix populated in row-major order.
Creating Tensors

To create a matrix, we can pass a list of its rows.

```python
1 identity = tf.constant([[1, 0, 0], [0, 1, 0], [0, 0, 1]], dtype=tf.float32)
2 print identity
Tensor("Const_144:0", shape=(3, 3), dtype=float32)
```

```python
with sess.as_default():
    print oneThruNine.eval()
[[ 1.  2.  3.]
 [ 4.  5.  6.]
 [ 7.  8.  9.]]
```

The `eval()` method actually computes a tensor’s value and returns it as a numpy array. `eval()` has to be run within a given session designated as “default”, so we specify `sess` as the default. More on this in a few slides.
Creating Tensors

Create a 10-by-10 matrix of all ones

Create a 4-tensor, which we could use to represent one second of 720p color video (27 frames per second, 1280x720 resolution, 3 colors)
Tensor shape

**Rank:** number of dimensions

**Shape:** sizes of the dimensions

```python
video = tf.zeros([27, 1280, 720, 3])
print(video)
```

```
Tensor("zeros_3:0", shape=(27, 1280, 720, 3), dtype=float32)
```

```python
print(video.shape)
```

```
(27, 1280, 720, 3)
```

**Note:** This looks like a tuple, but it is actually its own special type, `tf.TensorShape`
More about tensor rank

```python
video = tf.zeros([27, 1280, 720, 3])
print(video)
```

Tensor("zeros_4:0", shape=(27, 1280, 720, 3), dtype=\textit{float32})

```python
r = tf.rank(video)
len(video.shape)
```

$r$ will only get a value once we run the computational graph. There are good design reasons behind this: if the rank of our tensor depends on other inputs or variables, then we can’t know the rank of the tensor until runtime!

If the tensor is already populated (so that its rank is already established), one can simply look at the length of its shape object to get the rank.
It is often natural to refer to certain subsets of the entries of a tensor. A “subtensor” of a tensor is often called a slice, and the operation of picking out a slice is called slicing the tensor.
Tensor slices

One index is enough to specify a number in a vector (i.e., a 1-tensor)

Need two indices to pick out an entry of a matrix (i.e., a 2-tensor)
Tensor slices

Use `:` to pick out all entries along a row or column.

Create a vector from the second (zero-indexing!) column of the matrix.

Create a vector from the third row of the matrix.

Note: result is a “column vector” regardless of whether we slice a row or a column!
Tensor Slices

More complicated example: video processing

Four dimensions:
- Pixels (height-by-width)
- Three colors (RGB)
- Time index (multiple frames)
More complicated example: video processing

Four dimensions:
- Pixels (height-by-width)
- Three colors (RGB)
- Time index (multiple frames)

Test your understanding: What is the rank of the “video” tensor below?
Tensor Slices

More complicated example: video processing

Four dimensions:
- Pixels (height-by-width)
- Three colors (RGB)
- Time index (multiple frames)

Test your understanding:
What is the rank of the “video” tensor below?

Answer: 4, since there are four dimensions; height, width, color and time.
Tensor slices

```python
video = tf.zeros([27, 1280, 720, 3])
print(video)
Tensor("zeros_5:0", shape=(27, 1280, 720, 3), dtype=float32)

firstframe = video[0,:,:,:]
print(firstframe)
Tensor("strided_slice_19:0", shape=(1280, 720, 3), dtype=float32)

bluevideo = video[:,:,:,:2]
print(bluevideo)
Tensor("strided_slice_20:0", shape=(27, 1280, 720), dtype=float32)

redvideo = video[:,:,:,:0]
print(redvideo)
Tensor("strided_slice_21:0", shape=(27, 1280, 720), dtype=float32)
```

Use `:` to pick out all entries along a row or column.

Pick out the 3-color 1280-by-720 image that is the first frame of the video.

Pick out only the blue channel of the video (see RGB on wikipedia).

Pick out only the red channel of the video.
Reshaping tensors

Test your understanding:

Q: I have an x-by-y-by-z tensor. What is its rank?
Reshaping tensors

Test your understanding:

Q: I have an x-by-y-by-z tensor. What is its rank?

A: 3

Q: How many elements are in this x-by-y-by-z 3-tensor?
Reshaping tensors

Test your understanding:

Q: I have an x-by-y-by-z tensor. What is its rank?
A: 3

Q: How many elements are in this x-by-y-by-z 3-tensor?
A: \( x \cdot y \cdot z \)
Reshaping tensors

```python
mytensor = tf.zeros([10, 20, 30])
print(mytensor)

Tensor("zeros_7:0", shape=(10, 20, 30), dtype=float32)
```

```python
newtensor = tf.reshape(mytensor, [125, 3, 2, 8])
print(newtensor)

Tensor("Reshape_2:0", shape=(125, 3, 2, 8), dtype=float32)
```

```python
badtensor = tf.reshape(mytensor, [10, 20, 40])

---
ValueError
```

Reshape a 3-tensor into a 4-tensor. Note that the shapes are consistent with one another.

Reshaping to an inconsistent shape results in an error.
Evaluating Tensors

Evaluating the tensor lets us finally see the tensor’s contents rather than only its shape and dtype.

Evaluation requires running a computational graph, so we have to give TF a session to run.
Building a Simple Model: Linear Regression

```python
W = tf.Variable([.3], dtype=tf.float32)
b = tf.Variable([-3], dtype=tf.float32)
x = tf.placeholder(tf.float32)
linear_model = W*x + b
linear_model
<tf.Tensor 'add_27:0' shape=<unknown> dtype=float32>
```
Building a Simple Model: Linear Regression

Model: \( y = Wx + b \)

\( W \) and \( b \) are both rank-1 tensors, with values 0.3 and -0.3, respectively.

```
W = tf.Variable([.3], dtype=tf.float32)
b = tf.Variable([-0.3], dtype=tf.float32)
x = tf.placeholder(tf.float32)
linear_model = W*x + b
linear_model
```
Building a Simple Model: Linear Regression

From the documentation: The Variable() constructor requires an initial value for the variable, which can be a Tensor of any type and shape. The 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.

You don’t know what these are, yet, but we’ll see them soon.
Test your understanding: why is the shape unknown, here?
Building a Simple Model: Linear Regression

```python
W = tf.Variable([.3], dtype=tf.float32)
b = tf.Variable([-3.], dtype=tf.float32)
x = tf.placeholder(tf.float32)
linear_model = W*x + b

Model: y = Wx + b
```

```python
init = tf.global_variables_initializer()
sess.run(init)

print(sess.run(linear_model, {x: [1, 2, 3, 4]}))
```

```
[0. 0.3 0.6 0.9]
```

tf.Constant tensors are initialized immediately when we create them. On the other hand, tf.Variable Tensors need to be initialized before we can run the computational graph. init here becomes a pointer to a TF subgraph.

More information: https://www.tensorflow.org/api_docs/python/tf/global_variables_initializer
Building a Simple Model: Linear Regression

```python
W = tf.Variable([0.3], dtype=tf.float32)
b = tf.Variable([-0.3], dtype=tf.float32)
x = tf.placeholder(tf.float32)
linear_model = W*x + b

evaluate the computational graph with x taking the values 1, 2, 3 and 4.
```
Building a Simple Model: Linear Regression

So far, we have a circuit that computes a linear regression estimate.

To train our model, we need:

1) A loss function
2) A placeholder $y$ for the training data dependent values
Building a Simple Model: Linear Regression

```
1 y = tf.placeholder(tf.float32)
2 sq_err = tf.square(linear_model - y)
3 loss = tf.reduce_sum(sq_err)
4 print(sess.run(loss, {x: [1, 2, 3, 4], y: [0, -1, -2, -3]}))
```

23.66
Building a Simple Model: Linear Regression

```python
1 y = tf.placeholder(tf.float32)
2 sq_err = tf.square(linear_model - y)
3 loss = tf.reduce_sum(sq_err)
4 print(sess.run(loss, {x: [1, 2, 3, 4], y: [0, -1, -2, -3]})
```

Test your understanding: is `sq_err` a vector or a scalar?
Building a Simple Model: Linear Regression

```python
1 y = tf.placeholder(tf.float32)
2 sq_err = tf.square(linear_model - y)
3 loss = tf.reduce_sum(sq_err)
4 print(sess.run(loss, {x: [1, 2, 3, 4], y: [0, -1, -2, -3]}))
```

Note: `tf.reduce_sum` does just what you think it does!
Option 1: set $w$ and $b$ manually.
   We know $W=-1$, $b=1$ is the correct answer
   To change values of `tf.Variables`, use `tf.assign`

```python
1 fixedW = tf.assign(W, [-1])
2 fixedb = tf.assign(b, [1])
3 sess.run([fixedW, fixedb])
4 print(sess.run(loss, {x: [1, 2, 3, 4], y: [0, -1, -2, -3]}))
```

```
0.0
```
Building a Simple Model: Linear Regression

How can we improve (i.e., decrease) this loss?

Option 1: set $w$ and $b$ manually.
   We know $W=-1, b=1$ is the correct answer
   To change values of `tf.Variables`, use `tf.assign`

Option 2: use closed-form solution for loss-minimizing $\hat{W}$ and $b$.
   ...but then what happens when we have a model with no closed-form solution?
Building a Simple Model: Linear Regression

**Option 1:** set $w$ and $b$ manually.
We know $W=-1$, $b=1$ is the correct answer
To change values of `tf.Variables`, use `tf.assign`

**Option 2:** use closed-form solution for loss-minimizing $W$ and $b$.
...but then what happens when we have a model with no closed-form solution?

**Option 3:** use the built-in `tf.train` optimizer
Takes advantage of **symbolic differentiation**
Allows easy implementation of **gradient descent** and related techniques
Building a Simple Model: Linear Regression

[Code snippet]

```
1 y = tf.placeholder(tf.float32)
2 sq_err = tf.square(linear_model - y)
3 loss = tf.reduce_sum(sq_err)
4 print(sess.run(loss, {x: [1, 2, 3, 4], y: [0, -1, -2, -3]}))
```

23.66

**How can we improve (i.e., decrease) this loss?**

**Option 1:** set \( w \) and \( b \) manually.
- We know \( W=-1, \ b=1 \) is the correct answer
- To change values of \( tf.Variables \), use `tf.assign`

**Option 2:** use closed-form solution for loss-minimizing \( W \) and \( b \).
- What happens when we have a model with no closed form solution?

**Option 3:** use the built-in `tf.train` optimizer
- Takes advantage of **symbolic differentiation**
- Allows easy implementation of **gradient descent** and related techniques

**This is why we use TensorFlow!**
Training a Simple Model: Linear Regression

```
1 y = tf.placeholder(tf.float32)
2 sq_err = tf.square(linear_model - y)
3 loss = tf.reduce_sum(sq_err)
4 print(sess.run(loss, {x: [1, 2, 3, 4], y: [0, -1, -2, -3]}))
```

Reminder: this is what our model looks like, as a computational graph.
Gradient Descent: Crash Course

Iterative optimization method for minimizing a function
At location $x$, take gradient of loss function
Take a **gradient step** in the direction of the gradient
Size of step changes over time
  according to **learning rate**
Gradient Descent: Crash Course

Iterative optimization method for minimizing a function
At location $x$, take gradient of loss function
Take a **gradient step** in the direction of the gradient
Size of step changes over time according to **learning rate**

In short, gradient descent is a method for minimizing a function, provided we can compute its gradient (i.e., derivative). It’s enough for this course to treat this as a black box.

For more information:
Training a Simple Model: Linear Regression

```python
optimizer = tf.train.GradientDescentOptimizer(0.01)
train = optimizer.minimize(loss)
sess.run(init)  # reset values to incorrect defaults.
for i in range(1000):
    sess.run(train, {x: [1, 2, 3, 4], y: [0, -1, -2, -3]})
print(sess.run([W, b]))
```

[array([-0.9999969], dtype=float32), array([ 0.99999082], dtype=float32)]
Training a Simple Model: Linear Regression

```python
optimizer = tf.train.GradientDescentOptimizer(0.01)
train = optimizer.minimize(loss)
sess.run(init)  # reset values to incorrect defaults.
for i in range(1000):
    sess.run(train, {x: [1, 2, 3, 4], y: [0, -1, -2, -3]})
print(sess.run([W, b]))
```

[array([-0.999969], dtype=float32), array([ 0.9999082], dtype=float32)]

Each iteration of this loop computes one gradient step and updates the variables accordingly.
Training a Simple Model: Linear Regression

**Note:** As you can see below, the computational graph can get very complicated very quickly. TensorFlow has a set of built-in tools, collectively called **TensorBoard**, for handling some of this complexity: [https://www.tensorflow.org/get_started/graph_viz](https://www.tensorflow.org/get_started/graph_viz)

Each iteration of this loop computes one gradient step and updates the variables accordingly.
Training a Simple Model: Linear Regression

```python
optimizer = tf.train.GradientDescentOptimizer(0.01)
train = optimizer.minimize(loss)

sess.run(init)  # reset values to incorrect defaults.
losses = range(1000)
for i in range(1000):
    sess.run(train, {x: [1, 2, 3, 4], y: [0, -1, -2, -3]})
    losses[i] = sess.run(loss, {x: [1, 2, 3, 4], y: [0, -1, -2, -3]})
plt.xlabel('step'); plt.ylabel('loss'); _ = plt.plot(losses);
```

**Note:** TensorBoard includes a set of tools for visualization, including for tracking loss, but the approach here is quicker and easier for our purposes.

[https://www.tensorflow.org/get_started/summaries_and_tensorboard](https://www.tensorflow.org/get_started/summaries_and_tensorboard)
TensorFlow Estimators API: \texttt{tf.estimators}

\texttt{tf.estimators} is a TF module that simplifies model training and evaluation. Module allows one to run models on CPU or GPU, local or on cloud, etc. Simplifies much of the work of building the graph and estimating parameters.

More information:

https://www.tensorflow.org/get_started/get_started#tfestimator
https://www.tensorflow.org/programmers_guide/estimators
https://www.tensorflow.org/api_docs/python/tf/estimator
Readings

Required:

TensorFlow tutorial: Getting Started with TensorFlow
https://www.tensorflow.org/get_started/get_started
This is the whitepaper that originally introduced TensorFlow.

Recommended:

Assorted tutorials on statistical and neural models in TensorFlow:
https://www.tensorflow.org/tutorials/