TensorFlow Tutorial

slides adapted from Bharath Ramsundar
(CS224d - Stanford)

Deep-Learning Package Zoo

Torch

Caffe

Theano (Keras, Lasagne)

CuDNN

Tensorflow

Mxnet

Etc.

Deep-Learning Package Design Choices

Model specification: **Configuration file** (e.g. Caffe, DistBelief, CNTK) versus **programmatic generation** (e.g. Torch, Theano, Tensorflow)

For programmatic models, choice of high-level language: Lua (Torch) vs. Python (Theano, Tensorflow) vs others.

We chose to work with **python** because of rich community and library infrastructure.

TensorFlow vs. Theano

Theano is another deep-learning library with python-wrapper (was inspiration for Tensorflow)

Theano and TensorFlow are very similar systems. TensorFlow has better support for distributed systems though, and has development funded by Google, while Theano is an academic project.

tensorflow / tensorflow   Watch 2,056   Star 21,175   Fork 7,618

What is TensorFlow?

TensorFlow is a deep learning library recently open-sourced by Google.

But what does it actually do?

TensorFlow provides primitives for defining functions on tensors and automatically computing their derivatives.

But what's a Tensor?

Formally, tensors are multilinear maps from vector spaces to the real numbers (V vector space, and V^* dual space)

$$f : \underbrace{V^* \times \cdots V^*}_{p \text{ copies}} \times \underbrace{V \times \cdots V}_{q \text{ copies}} \to \mathbb{R}$$

A scalar is a tensor ($f : \mathbb{R} \to \mathbb{R},\ f(e_1) = c$)

A vector is a tensor ($f : \mathbb{R}^n \to \mathbb{R},\ f(e_i) = v_i$)

A matrix is a tensor ($f : \mathbb{R}^n \times \mathbb{R}^m \to \mathbb{R},\ f(e_i, e_j) = A_{ij}$)

Common to have fixed basis, **so a tensor can be represented as a multidimensional array of numbers.**

TensorFlow vs. Numpy

Few people make this comparison, but TensorFlow and Numpy are quite similar. (Both are N-d array libraries!)

Numpy has Ndarray support, but doesn't offer methods to create tensor functions and automatically compute derivatives (+ no GPU support).

NumPy *VS* TensorFlow

Simple Numpy Recap

```
In [23]: import numpy as np

In [24]: a = np.zeros((2,2)); b =
np.ones((2,2))

In [25]: np.sum(b, axis=1)
Out[25]: array([ 2.,  2.])

In [26]: a.shape
Out[26]: (2, 2)

In [27]: np.reshape(a, (1,4))
Out[27]: array([[ 0.,  0.,  0.,  0.]])
```

Repeat in TensorFlow

More on `Session` *soon*

More on `.eval()` *in a few slides*

```
In [31]: import tensorflow as tf

In [32]: tf.InteractiveSession()

In [33]: a = tf.zeros((2,2)); b = tf.ones((2,2))

In [34]: tf.reduce_sum(b, reduction_indices=1).eval()
Out[34]: array([ 2.,  2.], dtype=float32)

In [35]: a.get_shape()
Out[35]: TensorShape([Dimension(2), Dimension(2)])

In [36]: tf.reshape(a, (1, 4)).eval()
Out[36]: array([[ 0.,  0.,  0.,  0.]], dtype=float32)
```

`TensorShape` *behaves like a python tuple.*

Numpy to TensorFlow Dictionary

Numpy	TensorFlow
`a = np.zeros((2,2)); b = np.ones((2,2))`	`a = tf.zeros((2,2)), b = tf.ones((2,2))`
`np.sum(b, axis=1)`	`tf.reduce_sum(a,reduction_indices=[1])`
`a.shape`	`a.get_shape()`
`np.reshape(a, (1,4))`	`tf.reshape(a, (1,4))`
`b * 5 + 1`	`b * 5 + 1`
`np.dot(a,b)`	`tf.matmul(a, b)`
`a[0,0], a[:,0], a[0,:]`	`a[0,0], a[:,0], a[0,:]`

TensorFlow requires explicit evaluation!

```
In [37]: a = np.zeros((2,2))

In [38]: ta = tf.zeros((2,2))

In [39]: print(a)
[[ 0.  0.]
 [ 0.  0.]]

In [40]: print(ta)
Tensor("zeros_1:0", shape=(2, 2), dtype=float32)

In [41]: print(ta.eval())
[[ 0.  0.]
 [ 0.  0.]]
```

TensorFlow computations define a ***computation graph*** *that has no numerical value until evaluated!*

TensorFlow Session Object (1)

"A Session object encapsulates the environment in which Tensor objects are evaluated" - [TensorFlow Docs]

```
In [20]: a = tf.constant(5.0)

In [21]: b = tf.constant(6.0)

In [22]: c = a * b

In [23]: with tf.Session() as sess:
   ....:     print(sess.run(c))
   ....:     print(c.eval())
   ....:
30.0
30.0
```

`c.eval()` is just syntactic sugar for `sess.run(c)` in the currently active session!

TensorFlow Session Object (2)

`tf.InteractiveSession()` is just convenient syntactic sugar for keeping a default session open in ipython.

`sess.run(c)` is an example of a TensorFlow *Fetch*. Will say more on this soon.

Tensorflow Computation Graph

"TensorFlow programs are usually structured into a construction phase, that assembles a graph, and an execution phase that uses a session to execute ops in the graph." - TensorFlow docs

All computations add nodes to global default graph (docs)

TensorFlow Variables (1)

“When you train a model you use variables to hold and update parameters. Variables are in-memory buffers containing tensors” - TensorFlow Docs.

All tensors we’ve used previously have been *constant* tensors, not variables.

TensorFlow Variables (2)

```
In [32]: W1 = tf.ones((2,2))

In [33]: W2 = tf.Variable(tf.zeros((2,2)), name="weights")

In [34]: with tf.Session() as sess:
             print(sess.run(W1))
             sess.run(tf.initialize_all_variables())
             print(sess.run(W2))
   ....:
[[ 1.  1.]
 [ 1.  1.]]
[[ 0.  0.]
 [ 0.  0.]]
```

Note the initialization step `tf.initialize_all_variables()`

TensorFlow Variables (3)

TensorFlow variables must be initialized before they have values! Contrast with constant tensors.

```
In [38]: W = tf.Variable(tf.zeros((2,2)), name="weights")

In [39]: R = tf.Variable(tf.random_normal((2,2)), name="random_weights")

In [40]: with tf.Session() as sess:
   ....:     sess.run(tf.initialize_all_variables())
   ....:     print(sess.run(W))
   ....:     print(sess.run(R))
   ....:
```

`Variable` *objects can be initialized from constants or random values*

Initializes all variables with specified values.

Updating Variable State

```
In [63]: state = tf.Variable(0, name="counter")

In [64]: new_value = tf.add(state, tf.constant(1))

In [65]: update = tf.assign(state, new_value)

In [66]: with tf.Session() as sess:
   ....:     sess.run(tf.initialize_all_variables())
   ....:     print(sess.run(state))
   ....:     for _ in range(3):
   ....:         sess.run(update)
   ....:         print(sess.run(state))
   ....:
0
1
2
3
```

Roughly `new_value = state + 1`

Roughly `state = new_value`

Roughly

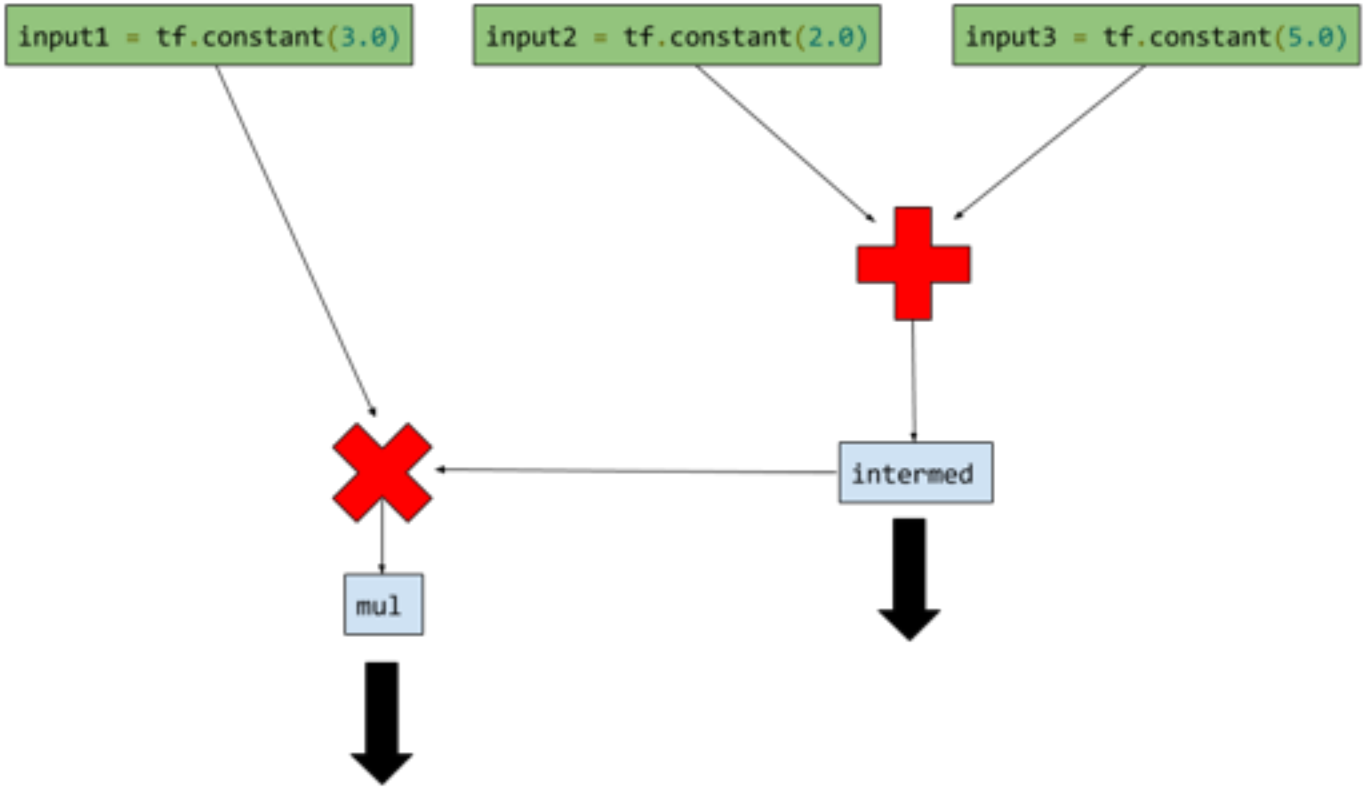
```
state = 0
print(state)
for _ in range(3):
  state = state + 1
  print(state)
```

Fetching Variable State (1)

```
In [82]: input1 = tf.constant(3.0)
In [83]: input2 = tf.constant(2.0)
In [84]: input3 = tf.constant(5.0)
In [85]: intermed = tf.add(input2, input3)
In [86]: mul = tf.mul(input1, intermed)
In [87]: with tf.Session() as sess:
   ....:       result = sess.run([mul,
intermed])
   ....:       print(result)
   ....:
[21.0, 7.0]
```

Calling `sess.run(var)` on a `tf.Session()` object retrieves its value. Can retrieve multiple variables simultaneously with `sess.run([var1, var2])` (See *Fetches* in TF docs)

Fetching Variable State (2)



Inputting Data

All previous examples have manually defined tensors. How can we input external data into TensorFlow?

Simple solution: Import from Numpy:

```
In [93]: a = np.zeros((3,3))
In [94]: ta = tf.convert_to_tensor(a)
In [95]: with tf.Session() as sess:
   ....:     print(sess.run(ta))
   ....:
[[ 0.  0.  0.]
 [ 0.  0.  0.]
 [ 0.  0.  0.]]
```

Placeholders and Feed Dictionaries (1)

Inputting data with `tf.convert_to_tensor()` is convenient, but doesn't scale.

Use `tf.placeholder` variables (dummy nodes that provide entry points for data to computational graph).

A `feed_dict` is a python dictionary mapping from `tf.placeholder` vars (or their names) to data (numpy arrays, lists, etc.).

Placeholders and Feed Dictionaries (2)

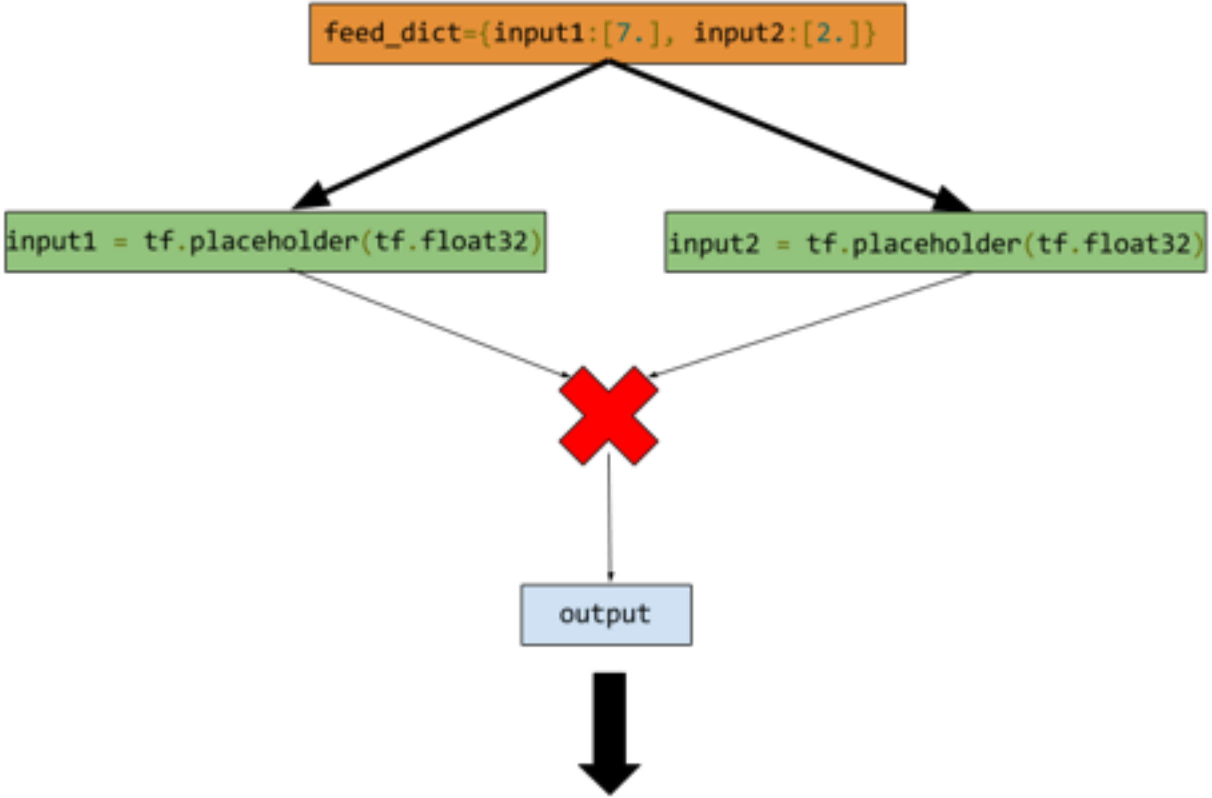
```
In [96]: input1 = tf.placeholder(tf.float32)

In [97]: input2 = tf.placeholder(tf.float32)

In [98]: output = tf.mul(input1, input2)

In [99]: with tf.Session() as sess:
   ....:         print(sess.run([output], feed_dict={input1:[7.], input2:[2.]}))
   ....:

[array([ 14.], dtype=float32)]
```

Define `tf.placeholder` *objects for data entry.*

Fetch value of output from computation graph.

Feed data into computation graph.

Placeholders and Feed Dictionaries (3)



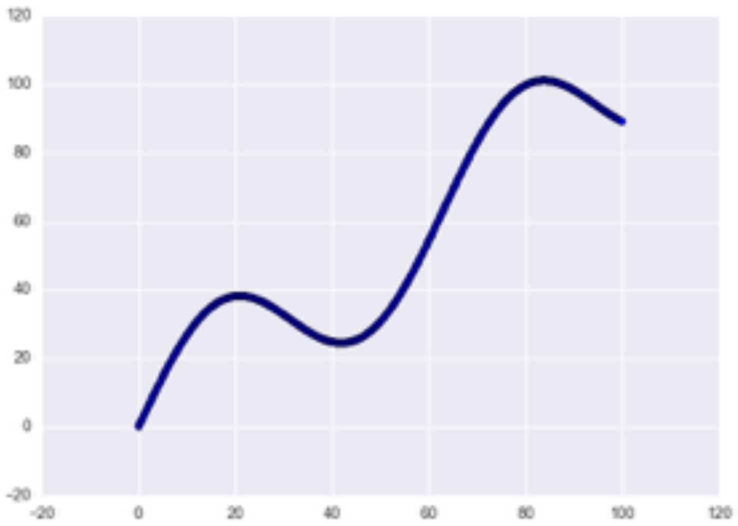
Variable Scope

Complicated TensorFlow models can have hundreds of variables.

- `tf.variable_scope()` provides simple name-spacing to avoid clashes.
- `tf.get_variable()` creates/accesses variables from within a variable scope.

Ex: Linear Regression in TensorFlow (1)

```
import numpy as np
import seaborn

# Define input data
X_data = np.arange(100, step=.1)
y_data = X_data + 20 * np.sin(X_data/
10)

# Plot input data
plt.scatter(X_data, y_data)
```



Ex: Linear Regression in TensorFlow (2)

```
# Define data size and batch size
n_samples = 1000
batch_size = 100

# Tensorflow is finicky about shapes, so resize
X_data = np.reshape(X_data, (n_samples,1))
y_data = np.reshape(y_data, (n_samples,1))

# Define placeholders for input
X = tf.placeholder(tf.float32, shape=(batch_size, 1))
y = tf.placeholder(tf.float32, shape=(batch_size, 1))
```

Ex: Linear Regression in TensorFlow (3)

```
# Define variables to be learned
with tf.variable_scope("linear-regression"):
  W = tf.get_variable("weights", (1, 1),)
                      initializer=tf.random_normal_initializer())
  b = tf.get_variable("bias", (1,),
                      initializer=tf.constant_initializer(0.0))
  y_pred = tf.matmul(X, W) + b
  loss = tf.reduce_sum((y - y_pred)**2/n_samples)
```

$$J(W, b) = \frac{1}{N} \sum_{i=1}^{N} (y_i - (Wx_i + b))^2$$

Ex: Linear Regression in TensorFlow (4)

```
# Sample code to run one step of gradient descent
In [136]: opt = tf.train.AdamOptimizer()

In [137]: opt_operation = opt.minimize(loss)

In [138]: with tf.Session() as sess:
   .....:     sess.run(tf.initialize_all_variables())
   .....:     sess.run([opt_operation], feed_dict={X: X_data, y: y_data})
   .....:
```

Note TensorFlow scope is not python scope! Python variable `loss` is still visible.

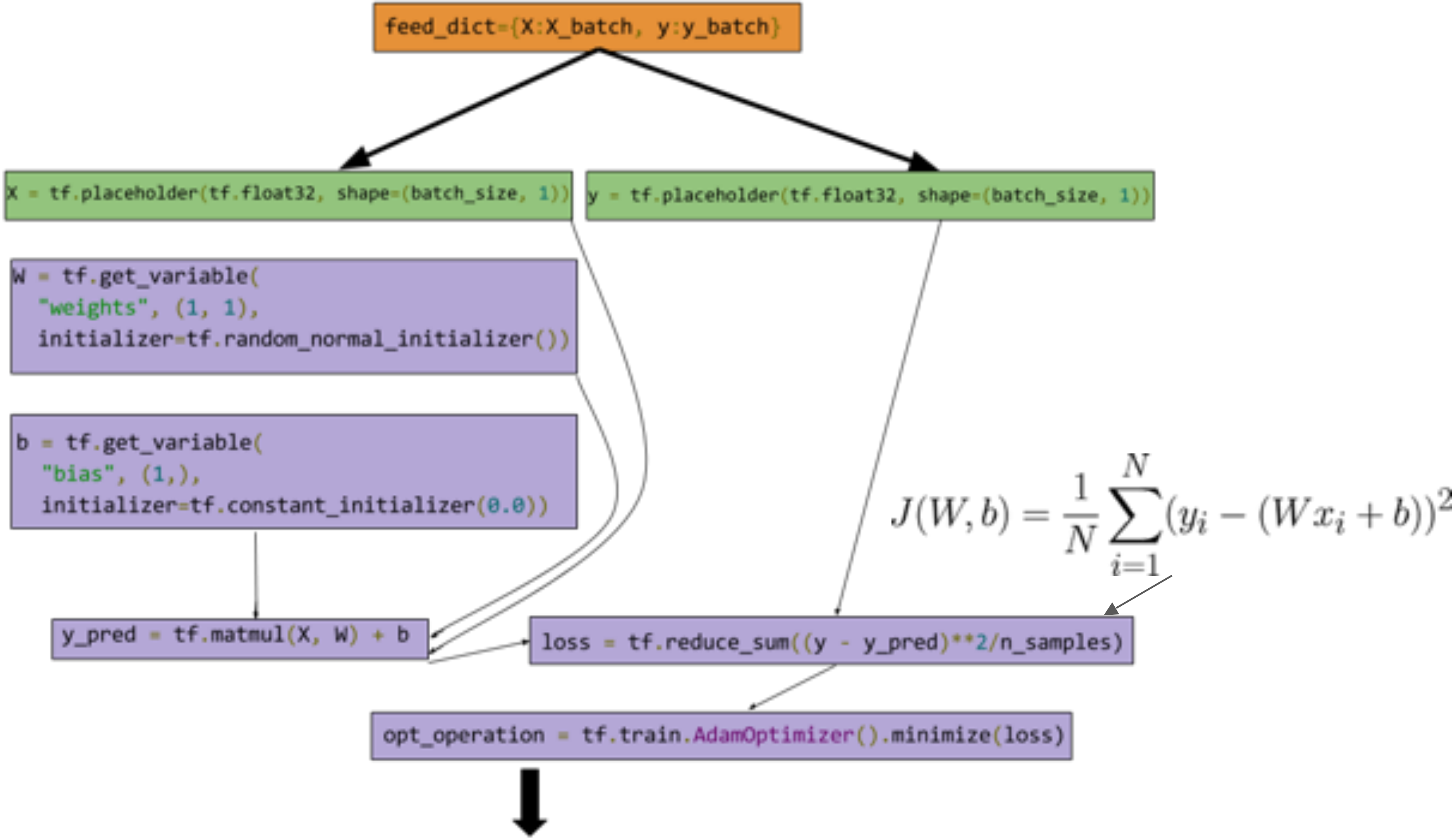
But how does this actually work under the hood? Will return to TensorFlow computation graphs and explain.

Ex: Linear Regression in TensorFlow (4)

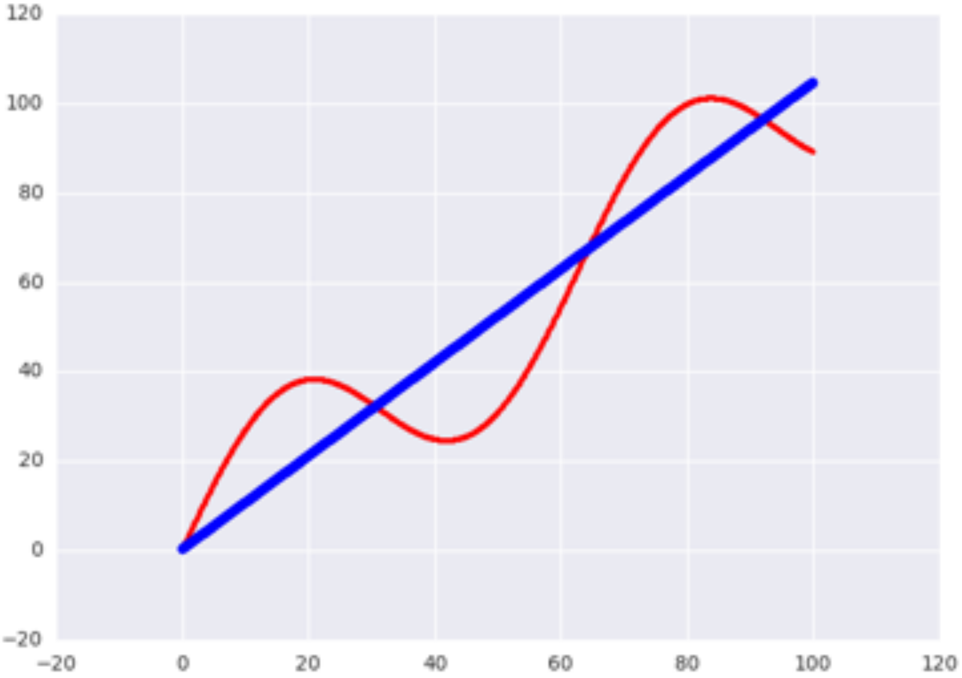
```
# Sample code to run full gradient descent:
# Define optimizer operation
opt_operation = tf.train.AdamOptimizer().minimize(loss)

with tf.Session() as sess:
  # Initialize Variables in graph
  sess.run(tf.initialize_all_variables())
  # Gradient descent loop for 500 steps
  for _ in range(500):
    # Select random minibatch
    indices = np.random.choice(n_samples, batch_size)
    X_batch, y_batch = X_data[indices], y_data[indices]
    # Do gradient descent step
    _, loss_val = sess.run([opt_operation, loss], feed_dict={X: X_batch, y: y_batch})
```

Let's do a deeper. graphical dive into this operation

Ex: Linear Regression in TensorFlow (5)



Ex: Linear Regression in TensorFlow (6)



Learned model offers nice fit to data.

Concept: Auto-Differentiation

Linear regression example computed L2 loss for a linear regression system. How can we fit model to data?

`tf.train.Optimizer` creates an optimizer.

`tf.train.Optimizer.minimize(loss, var_list)` adds optimization operation to computation graph.

Automatic differentiation computes gradients without user input!

TensorFlow Gradient Computation

TensorFlow nodes in computation graph have attached gradient operations.

Use backpropagation (using node-specific gradient ops) to compute required gradients for all variables in graph.

TensorFlow Gotchas/Debugging (1)

Convert tensors to numpy array and print.

TensorFlow is fastidious about types and shapes. Check that types/shapes of all tensors match.

TensorFlow API is less mature than Numpy API. Many advanced Numpy operations (e.g. complicated array slicing) not supported yet!

TensorFlow Gotchas/Debugging (2)

If you're stuck, try making a pure Numpy implementation of forward computation.

Then look for analog of each Numpy function in TensorFlow API

Use `tf.InteractiveSession()` to experiment in shell. Trial and error works!

TensorBoard

TensorFlow has some neat built-in visualization tools (TensorBoard).

Takeaway message

To use TensorFlow you should understand how TensorFlow:

- Represents computations as graphs

- Executes graphs in the context of Sessions

- Represents data as tensors

- Maintains state with Variables

- Uses feeds and fetches to get data into and out of arbitrary operations.