Previous lecture: Introduction to TensorFlow

- `tf.Tensor` objects represent tensors
- Tensors are combined into a computational graph
  - Captures the computational operations to be carried out at runtime

This lecture: Advanced TF

- More detail on the computational graph and `tf.Tensor` objects

Lab: recognizing MNIST handwritten digits
Recall: TensorFlow as DataFlow

Computational graph: how data “flows” through program

In previous lecture:
  We were a bit fast and loose with nodes and edges

Strictly speaking:
  Nodes are operations (`tf.Operation`)
  Edges are tensors (`tf.Tensor`)
More on the Computational Graph

tf.Graph

Special class provided by TF to represent a computational graph
Contains tf.Operation objects and tf.Tensor objects
...and keeps track of how they interact (i.e., the graph structure itself)

When you define tensors in TF, a graph is built for you automatically
Called the default graph
At all times, some graph is the default graph
Call tf.get_default_graph() to access it

More information: https://www.tensorflow.org/api_docs/python/tf/Graph
More on the Computational Graph

`tf.Tensor`

(Already familiar to you)
Represents a tensor, i.e., data on which to perform computations

`tf.Operation`

TF class that represents a computation performed on zero or more tensors
Also a node in a computational graph
Tensor operations

Previous lecture: we saw different ways of creating tensors... 
...but not much in the way of how to do things with them.

Example functions available in TF:
  tf.abs(...) : computes absolute value of a tensor
  tf.add_n(...) : adds two or more tensors, element-wise
  tf.cholesky(...) : computes Cholesky decomposition
    https://en.wikipedia.org/wiki/Cholesky_decomposition
  tf.exp(...) : computes exponential, element-wise
  tf.less(...) : evaluates x < y, element-wise
  tf.sigmoid(...) : computes sigmoid function element-wise
    https://en.wikipedia.org/wiki/Sigmoid_function
Tensor operations: +,-,*,/

```python
import tensorflow as tf

sess = tf.Session()

a = tf.constant(5, dtype=tf.float32)
b = tf.constant(3.1415, dtype=tf.float32)
c = tf.constant(2, dtype=tf.float32)
x = tf.placeholder(tf.float32)
y = tf.placeholder(tf.float32)
z = tf.placeholder(tf.float32)

ans = x/a + b*y - c*z

print(sess.run(ans, {x: [4, 3, 2, 1], y: [2, 3, 4, 5], z: [1, 1, 2, 2]}))
sess.close()
```

```
[5.08300018 8.02449989 8.9659996 11.90750027]
```

```python
sess = tf.Session()
divbyzero = x/y

print(sess.run(divbyzero, {x:1, y:0}))
```

```
inf
```
Matrix multiplication in TF: `tf.matmul()`

```python
sess = tf.Session()
M = tf.constant([[1, 0, 1], [0, 1, 1], [1, 1, 0]], dtype=tf.float32)
oneThruNine = tf.constant([[1, 2, 3], [4, 5, 6], [7, 8, 9]], dtype=tf.float32)
c = tf.matmul(oneThruNine, M)
with sess.as_default():
    print(c.eval())
```

```
[[  4.  10.  16.]
 [  5.  11.  17.]
 [  3.   9.  15.]]
```

```python
M1 = tf.constant([[1, 0, 1], [0, 1, 1]], dtype=tf.float32)
M2 = tf.constant([[1, 0, 1], [0, 0, 1]], dtype=tf.float32)
R = tf.matmul(M1, M2)
```

```
ValueError: Dimensions must be equal, but are 3 and 2 for 'MatMul_13' (op: 'MatMul') with input shapes: [2,3], [2,4].
```

tf.matmul(A,B) multiplies tensors A and B, as matrices, provided their ranks and types agree.
Matrix multiplication in TF: `tf.matmul()`

```python
with sess.as_default():
    print(c.eval())
```

```
[[  4.  10.  16.]
 [  5.  11.  17.]
 [  3.  9.  15.]]
```

**tf.matmul(A, B)** multiplies tensors A and B, as matrices, provided their ranks and types agree.

**Note:** `tf.matmul()` can be used to multiply tensors of arbitrary rank. Using appropriate flags, we can transpose/adjoint the arguments as we please. Details: [https://www.tensorflow.org/api_docs/python/tf/matmul](https://www.tensorflow.org/api_docs/python/tf/matmul)
More matrix operations in TF

- `tf.matrix_diag`: picks out diagonal of a matrix (or other tensor)
- `tf.matrix_determinant`: computes determinant of a matrix
- `tf.matrix_inverse`: computes inverse of a matrix
- `tf.matrix_solve`: solves $Ax = b$
- `tf.matrix_transpose`: transposes a matrix
Element-wise operations in TF

TF element-wise operations are just like Numpy universal functions

Examples:

- `tf.abs()`: computes absolute value
- `tf.acos()`: computes arccosine
- `tf.cos()`: computes cosine
- `tf.exp()`: computes exponential
- `tf.log()`: computes logarithm
- `tf.sigmoid()`: computes sigmoid function

https://en.wikipedia.org/wiki/Sigmoid_function
Element-wise comparisons in TF

TF supports element-wise comparisons of tensors

- `tf.less()`, `tf.less_equal()`
- `tf.greater()`, `tf.greater_equal()`
- `tf.equal()`, `tf.not_equal()`

Logical (operate on tensors with `dtype=bool`)

- `tf.logical_and()`
- `tf.logical_or()`
- `tf.logical_xor()`

Also supported: `tf.logical_not()`, but this isn’t a comparison
So, TF has a lot of stuff going on!

“low-level” TF API makes lots of powerful tools available

...almost too many!

I just wanted to train a neural net! Why do I have to worry about all this stuff!
Rest of Lecture: Lab

1) We’ll use softmax regression to classify handwritten digits
   Using the low-level API that we discussed last lecture

2) We’ll build and train a simple NN on the same data
   Also using the low-level API
   So you can see why many people just use the `tf.estimator` API!
Workshop: Recognizing MNIST Digits

MNIST is a famous computer vision data set
28-by-28 greyscale images of hand-written digits
https://en.wikipedia.org/wiki/MNIST_database

Each image is labeled according to what digit it represents

2012: 0.23 percent error rate: https://arxiv.org/abs/1202.2745
(there has probably been improvement in this number since then…)

Follow along: https://www.tensorflow.org/get_started/mnist/beginners

Pared-down demo code:
http://www-personal.umich.edu/~klevin/teaching/Fall2017/STATS700-002/demo/softmax_mnist.ipynb

Recognizing MNIST Digits

**Goal:** given an image, classify what digit it represents.

In particular, we’ll build a model that outputs a vector of probabilities

\[ i \text{-th entry of vector will be model’s confidence that image is digit } i. \]
Softmax Regression

Generalizes logistic regression to categorical variables with >2 values

Softmax function: \( \sigma_j(z) = \frac{e^{z_j}}{\sum_i e^{z_i}} \)

Our model will assign probabilities to digits as

\[ P[Y = j] = \sigma_j(WX + b) \]

More information:
https://en.wikipedia.org/wiki/Multinomial_logistic_regression
https://en.wikipedia.org/wiki/Softmax_function
The Plan

Represent 28-by-28 images by flattened 784-dimensional vectors

Apply softmax regression to vectors
  Learn weights $\mathbf{w}$ and bias $b$
  Train on a training set of labeled images

Evaluate learned model on test set
Flattening the data

Images are most naturally represented as matrices...

...but softmax regression requires vector inputs.

**Solution:** “unroll” image into a vector. It doesn’t matter how we do this, so long as we’re consistent. That is, so long as every image is flattened to a vector in the same way.
Building the model

```python
import tensorflow as tf
x = tf.placeholder(tf.float32, [None, 784])
W = tf.Variable(tf.zeros([784, 10]))
b = tf.Variable(tf.zeros([10]))
y = tf.nn.softmax(tf.matmul(x, W) + b)
```

Image credit: TensorFlow tutorial
Building the model

$W$ is a matrix of weights. We are computing $xW$, so for matrix multiplication to make sense rows of $W$ must agree with columns of $x$. Our model outputs a 10-dimensional probability, so 10 columns is what we want.

Each row of $x$ is going to be a single observation, each of which is 784-dimensional vector (28-by-28 image has 784 pixels), but we don’t know how many rows $x$ will have, yet.

$W$ is a matrix of weights. We are computing $xW$, so for matrix multiplication to make sense rows of $W$ must agree with columns of $x$. Our model outputs a 10-dimensional probability, so 10 columns is what we want.

Bias term is same dimension as $Wx$.

Image credit: TensorFlow tutorial
Training the model

To train our model, we need to choose a loss function.

We’ll use cross-entropy: [https://en.wikipedia.org/wiki/Cross_entropy](https://en.wikipedia.org/wiki/Cross_entropy)

Related to the KL divergence

\[
H_{y'}(y) = \sum_i y'_i \log y_i
\]

- Sum over digits 0 to 9
- The true distribution
- Our model
Training the model

To train our model, we need to choose a loss function.

We’ll use cross-entropy: https://en.wikipedia.org/wiki/Cross_entropy

Related to the KL divergence

\[ H_{y'}(y) = \sum_i y'_i \log y_i \]

**Note:** the formula above is the sum for one observation. Our actual loss function will be a sum of these sums: for each training example, we need to sum of over the 10 digits.
Training the model

To train our model, we need to choose a loss function.

We’ll use cross-entropy: [https://en.wikipedia.org/wiki/Cross_entropy](https://en.wikipedia.org/wiki/Cross_entropy)

Related to the KL divergence

\[ H_{y'}(y) = \sum_i y'_i \log y_i \]

```python
1 cross_entropy = tf.reduce_mean(-tf.reduce_sum(ytrue * tf.log(y), reduction_indices=[1]))
```
Training the model

To train our model, we need to choose a loss function

We’ll use cross-entropy: [https://en.wikipedia.org/wiki/Cross_entropy](https://en.wikipedia.org/wiki/Cross_entropy)

Related to the KL divergence

\[ H_{y'}(y) = \sum_i y'_i \log y_i \]

```
1 cross_entropy = tf.reduce_mean(-tf.reduce_sum(ytrue * tf.log(y), reduction_indices=[1]))
```

“True” y  Our model

Tells TF to take the mean across the second axis.
Training the model

To train our model, we need to choose a loss function.

We’ll use cross-entropy: https://en.wikipedia.org/wiki/Cross_entropy

Related to the KL divergence

\[ H_{y'}(y) = \sum_i y'_i \log y_i \]

“True” y  
Our model

```
1 cross_entropy = tf.reduce_mean(-tf.reduce_sum(ytrue * tf.log(y), reduction_indices=[1]))
```

Note: it turns out that it’s more efficient and more numerically stable to use TF built-in function for cross-entropy, but this is how we would implement it if we had to.

Tells TF to take the mean across the second axis.
We'll read the truth into $y_{true}$. Again, we don’t know how many training instances there will be.

Specify the gradient descent step. This operation encodes a single gradient step in trying to minimize the cross-entropy.

Specify the **learning rate**, which controls the step size in our gradient descent algorithm.
Training the model: building more of the graph

We’ll read the truth into `ytrue`. Again, we don’t know how many training instances there will be.

```
1 ytrue = tf.placeholder(tf.float32, [None, 10])
2 cross_entropy = tf.reduce_mean(tf.nn.softmax_cross_entropy_with_logits(labels=ytrue, logits=y))
3 train_step = tf.train.GradientDescentOptimizer(0.5).minimize(cross_entropy)
```

**Note:** we are using what is called a **one-hot** encoding in the true labels `ytrue`. 
Aside: one-hot encodings

In ML, it is common to represent categorical variables by vectors. K possible values for the variable represent by a K-dimensional vector. Object of type k represented by vector with k-th entry 1, rest 0.
Aside: one-hot encodings

In ML, it is common to represent categorical variables by vectors
K possible values for the variable
represent by a K-dimensional vector
Object of type k represented by vector with k-th entry 1, rest 0

Note: this is a case where it’s good to use the tf.SparseTensor object. If K is really big, it’s expensive to store all those 0s! In our application, K=10, so it’s no big deal, but in, for example, NLP, K=1e6 is not uncommon.
Training the model: building more of the graph

We'll read the truth into $y_{true}$. Again, we don't know how many training instances there will be.

```python
1  ytrue = tf.placeholder(tf.float32, [None, 10])
2  cross_entropy = tf.reduce_mean(tf.nn.softmax_cross_entropy_with_logits(labels=ytrue, logits=y))
3  train_step = tf.train.GradientDescentOptimizer(0.5).minimize(cross_entropy)
```

**Note:** TensorFlow supplies a number of other optimization routines [https://www.tensorflow.org/api_guides/python/train#Optimizers](https://www.tensorflow.org/api_guides/python/train#Optimizers)
Running the Computational Graph

Here's the graph we've built, so far:

Note: this is a simplification of the graph that TF would build for you. You can view the actual graph using TensorBoard: https://www.tensorflow.org/get_started/graph_viz
Putting it all together

```python
from tensorflow.examples.tutorials.mnist import input_data

data_dir = "data_dir"

mnist = input_data.read_data_sets(data_dir, one_hot=True)

Extracting data_dir/train-images-idx3-ubyte.gz
Extracting data_dir/train-labels-idx1-ubyte.gz
Extracting data_dir/t10k-images-idx3-ubyte.gz
Extracting data_dir/t10k-labels-idx1-ubyte.gz

tf

ytrue = tf.placeholder(tf.float32, [None, 10])
cross_entropy = tf.reduce_mean(tf.nn.softmax_cross_entropy_with_logits(labels=ytrue, logits=y))
train_step = tf.train.GradientDescentOptimizer(0.5).minimize(cross_entropy)

with tf.Session() as sess:
    tf.global_variables_initializer().run()

for _ in range(5000):
    batch_xs, batch_ys = mnist.train.next_batch(100)
    sess.run(train_step, feed_dict={x: batch_xs, ytrue: batch_ys})
```
Putting it all together

```python
from tensorflow.examples.tutorials.mnist import input_data
data_dir = "data_dir"
mnist = input_data.read_data_sets(data_dir, one_hot=True)

# Extracting data_dir/train-images-idx3-ubyte.gz
# Extracting data_dir/train-labels-idx1-ubyte.gz
# Extracting data_dir/t10k-images-idx3-ubyte.gz
# Extracting data_dir/t10k-labels-idx1-ubyte.gz

y_true = tf.placeholder(tf.float32, [None, 10])
cross_entropy = tf.reduce_mean(tf.nn.softmax_cross_entropy_with_logits(labels=y_true, logits=y))
train_step = tf.train.GradientDescentOptimizer(0.5).minimize(cross_entropy)

sess = tf.InteractiveSession()
tf.global_variables_initializer().run()
for _ in range(5000):
    batch_xs, batch_ys = mnist.train.next_batch(100)
sess.run(train_step, feed_dict={x: batch_xs, y_true: batch_ys})
```

TF includes code for downloading MNIST data. We just need to tell it what directory to save it in.

Build the computational graph \((x, y, W, b)\) omitted for space.

Start session, take 5000 gradient steps. Use a batched approach. Each gradient step is based on a small subset of the training data.
Assessing the model: test data

Once we’ve trained a model, how do we tell if it’s good?

Use train/test split

Data set aside ahead of time, which the model hasn’t seen before
Train on one set of data (train data), evaluate on another (test data)

```python
1 correct_prediction = tf.equal(tf.argmax(y, 1), tf.argmax(ytrue, 1))
2 accuracy = tf.reduce_mean(tf.cast(correct_prediction, tf.float32))
3 print(sess.run(accuracy, feed_dict={x: mnist.test.images, ytrue: mnist.test.labels}))
```

0.9221

Now we’re using the **test data** instead of the training data.
Workshop II: Better Digit Recognition with NNs

Can we do better than 92% accuracy?

One obvious flaw:
   Our softmax regression doesn’t use structure of the image
   **How** we vectorized our image didn’t matter!

Two options:
   1) Write down a better model
   2) Use a neural net!
Crash Course: Neural Nets

Biologically-inspired computing model
Inputs processed by units (“neurons”)
  Each unit outputs a function of some inputs
Units apply linear functions to their inputs...
...followed by a nonlinear activation function

Goal: build a model that approximates some function
   Ex: input is an audio signal, output is a (prob. dist. over) word label
   Ex: input is English text, output is (prob. dist. over) French text
   Ex: input is an image, output is (prob. dist. over) label
Crash Course: Neural Nets

Note: multiple arrows from a unit denote broadcast, not different outputs.

Note: each unit has its own weight and bias. We will often collect the weights and biases from a single layer into a pair of vectors.
Crash Course: Neural Nets

Early NNs: perceptron (Rosenblatt, 1957)
  Single-layer of computation
  Can only learn linearly separable functions
  https://en.wikipedia.org/wiki/Perceptron

Multilayer perceptron (MLP)
  Multiple layers of units, can learn more complicated functions (e.g., XOR)
  https://en.wikipedia.org/wiki/Multilayer_perceptron

Feed-forward vs recurrent neural net (RNN)
  Feed-forward network is an acyclic graph
  RNN can have units whose outputs feed back to earlier units
Convolutional Neural Nets (CNNs)

Deep (many layers)

Feed-forward (NN connections are acyclic)

Three basic types of layers:
  Convolutional
  Pooling
  Fully connected

Dropout “layer” provides regularization
Convolution

(Based on) an operation from signal processing
Roughly speaking, convolution computes response of a system to an input
https://en.wikipedia.org/wiki/Convolution

Typical NNs: units apply matrix multiplication followed by nonlinearity

CNN: units apply convolution instead of matrix multiplication
Still a linear operation

In image processing, units apply convolution to their receptive fields
Biologically inspired: neurons in visual cortex respond selectively
Pooling

Typical setup: pass output of one unit to next layer

Pooling replaces this with a **summary statistic**
  - Input to next layer is a function of several units from previous layer
  - Example: pool adjacent pixels in an image

Common pooling operations:
  - Max pooling: report maximum value over the outputs
  - (weighted) average: take weighted average over the outputs
    - Weighted according to, e.g., distance from center of receptive field
Dropout

Common technique for regularization (avoiding overfitting)

At each training step, randomly choose some units to drop

These units do not contribute to the network computation
Forces other weights to “compensate”, introduces redundancy across units

This is the paper in which dropout was initially suggested.
Building the Neural Net

Four layers
  Two convolutional layers
  Two fully-connected layers
  Dropout between FC layers

Nonlinearity: We’ll use Rectified Linear Unit (RELU)

Pooling: max-pooling over 2-by-2 squares

Jupyter notebook:
  http://www-personal.umich.edu/~klevin/teaching/Fall2017/STATS700-002/demo/cnn_mnist.ipynb
Readings (this lecture)

Required:
   None

Recommended:
      Chapter 6: Deep Feedforward Neural Nets
      Chapter 9: Convolutional Networks
   http://www.deeplearningbook.org/