Deep Learning / Neural Nets

Without question the biggest thing in ML and computer science right now. Is the hype real? Can you learn anything meaningful in an afternoon? How did we get to this point?

The ideas have been around for decades. Two components came together in the past decade to enable astounding progress:

- Widespread parallel computing (GPUs)
- Big data training sets
Two Perspectives

There are really two common ways to view the fundamentals of deep learning.

- Inspired by biological models.
- An evolution of classic ML techniques (the perceptron).

They are both fair and useful. We’ll give each a thin slice of our attention before we move on to the actual implementation. You can decide which perspective works for you.
Modeled After The Brain
As a Highly Dimensional Non-linear Classifier

Perceptron

No Hidden Layer
Linear

Network

Hidden Layers
Nonlinear

Courtesy: Chris Olah
Basic NN Architecture

Input Layer → Hidden Layer → Output Layer

Neuron → Synapse
Activation Function

• Neurons apply activation functions at these summed inputs.
• Activation functions are typically non-linear.
• The sigmoid function produces a value between 0 and 1, so it is intuitive when a probability is desired, and was almost standard for many years.

\[ S(t) = \frac{1}{1 + e^{-t}} \]

• The Rectified Linear activation function is zero when the input is negative and is equal to the input when the input is positive.
• Rectified Linear activation functions have become more popular because they are faster to compute than the sigmoid or hyperbolic tangent.
• We will use these later.
Inference
Using a NN

H1 Weights = (1.0, -2.0, 2.0)
H2 Weights = (2.0, 1.0, -4.0)
H3 Weights = (1.0, -1.0, 0.0)

O1 Weights = (-3.0, 1.0, -3.0)
O2 Weights = (0.0, 1.0, 2.0)
H1 Weights = (1.0, -2.0, 2.0)
H2 Weights = (2.0, 1.0, -4.0)
H3 Weights = (1.0, -1.0, 0.0)

O1 Weights = (-3.0, 1.0, -3.0)
O2 Weights = (0.0, 1.0, 2.0)

H1 = \( \text{Sigmoid}(0.5 * 1.0 + 0.9 * -2.0 + -0.3 * 2.0) = \text{Sigmoid}(-1.9) = .13 \)
H2 = \( \text{Sigmoid}(0.5 * 2.0 + 0.9 * 1.0 + -0.3 * -4.0) = \text{Sigmoid}(3.1) = .96 \)
H3 = \( \text{Sigmoid}(0.5 * 1.0 + 0.9 * -1.0 + -0.3 * 0.0) = \text{Sigmoid}(-0.4) = .40 \)
Inference

H1 Weights = (1.0, -2.0, 2.0)
H2 Weights = (2.0, 1.0, -4.0)
H3 Weights = (1.0, -1.0, 0.0)

O1 Weights = (-3.0, 1.0, -3.0)
O2 Weights = (0.0, 1.0, 2.0)

O1 = Sigmoid(0.13 * -3.0 + 0.96 * 1.0 + 0.40 * -3.0) = Sigmoid(-0.63) = 0.35
O1 = Sigmoid(0.13 * 0.0 + 0.96 * 1.0 + 0.40 * 2.0) = Sigmoid(1.76) = 0.85
As A Matrix Operation

H1 Weights = (1.0, -2.0, 2.0)
H2 Weights = (2.0, 1.0, -4.0)
H3 Weights = (1.0, -1.0, 0.0)

\[
\begin{align*}
\text{Hidden Layer Weights} & = \begin{bmatrix}
1.0 & -2.0 & 2.0 \\
2.0 & 1.0 & -4.0 \\
1.0 & -1.0 & 0.0 
\end{bmatrix} \\
\text{Inputs} & = \begin{bmatrix}
0.5 \\
0.9 \\
-0.3 
\end{bmatrix} \\
\text{Hidden Layer Outputs} & = \begin{bmatrix}
0.13 \\
0.96 \\
0.4 
\end{bmatrix}
\end{align*}
\]

Now this looks like something that we can pump through a GPU.
The magic formula for a neural net is that, at each layer, we apply linear operations (which look naturally like linear algebra matrix operations) and then pipe the final result through some kind of final nonlinear activation function. The combination of the two allows us to do very general transforms.
Linear + Nonlinear

These are two very simple networks untangling spirals. Note that the second does not succeed. With more substantial networks these would both be trivial.

Courtesy: Chris Olah
A very underappreciated fact about networks is that the width of any layer determines how many dimensions it can work in. This is valuable even for lower dimension problems. How about trying to classify (separate) this dataset:

Can a neural net do this with twisting and deforming? What good does it do to have more than two dimensions with a 2D dataset?
Working In Higher Dimensions

It takes at least 3 units wide to pull this off, regardless of depth. Greater depth allows us to stack these operations, and can be very effective. The gains from depth are harder to characterize.

Courtesy: Chris Olah
Training Neural Networks
How do we find these magic weights?

Backpropagation

1. Originally, the weights of a neural network are assigned randomly
2. The neural network then predicts the labels for the examples in the training set using inference
3. The error between the prediction and the label is used to determine how the weights should be updated
4. The weights are slowly changed to minimize the error
5. Error minimization is achieved with Gradient Descent (or some variant)
   • This routine needs to know the derivative of the error with respect to the weights
   • Stochastic Gradient Descent (SGD) is a variation of Gradient Descent that uses a subset of the training data at each time step to approximate the overall derivative to update the weights
Using the Derivative and Chain Rule

\[ \frac{\partial E}{\partial w} = I \cdot (O - T) \cdot O \cdot (1 - O) \]

For Sigmoid
\[ S(t) = \frac{1}{1 + e^{-t}} \]
We now know enough to attempt a problem. Only because the Tensorflow framework fills in a lot of the details that we have glossed over. That is one of its functions.

Our problem will be character recognition. We will learn to read handwritten digits by training on a large set of 28x28 greyscale samples.

First we’ll do this with the simplest possible model just to show how the Tensorflow framework functions. Then we will implement a quite sophisticated and accurate convolutional neural network for this same problem.
MNIST Data

Specifically we will have a file with 55,000 of these numbers.

The labels will be “one-hot vectors”, which means a 1 in the numbered slot:

\[ 6 = [0,0,0,0,0,0,1,0,0,0] \]
Tensorflow Startup

Make sure you are on a GPU node:

```
br006% interact -gpu
gpu42%
```

These examples assume you have the MNIST data sitting around in your current directory:

```
gpu42% ls
-rw-r--r-- 1 urbanic pscstaff 1648877 May  4 02:13 t10k-images-idx3-ubyte.gz
-rw-r--r-- 1 urbanic pscstaff  4542 May  4 02:13 t10k-labels-idx1-ubyte.gz
-rw-r--r-- 1 urbanic pscstaff  9912422 May  4 02:13 train-images-idx3-ubyte.gz
-rw-r--r-- 1 urbanic pscstaff   28881 May  4 02:13 train-labels-idx1-ubyte.gz
```

As of this week Tensorflow startup has one extra step:

```
gpu42% module load tensorflow/1.1.0
gpu42% source $TENSORFLOW_ENV/bin/activate
gpu42% python
```
MNIST With Regression

$ python
Python 3.6.1 |Continuum Analytics, Inc.| (default, Mar 22 2017, 19:54:23)
[GCC 4.4.7 20120313 (Red Hat 4.4.7-1)] on linux
Type "help", "copyright", "credits" or "license" for more information.
>>> from tensorflow.examples.tutorials.mnist import input_data
>>> import tensorflow as tf
>>> mnist = input_data.read_data_sets(".", one_hot=True)

....You may get some congratulatory noise here...
...........Pay it no heed...............
The API is well documented.

That is terribly unusual.
Regression MNIST

```python
$ python
Python 3.6.1 |Continuum Analytics, Inc.| (default, Mar 22 2017, 19:54:23)
[GCC 4.4.7 20120313 (Red Hat 4.4.7-1)] on linux
Type "help", "copyright", "credits" or "license" for more information.
>>> from tensorflow.examples.tutorials.mnist import input_data
>>> import tensorflow as tf
>>> mnist = input_data.read_data_sets(".", one_hot=True)
>>> x = tf.placeholder(tf.float32, [None, 784])
>>> W = tf.Variable(tf.zeros([784, 10]))
>>> b = tf.Variable(tf.zeros([10]))
>>> y = tf.matmul(x, W) + b
>>> y_ = tf.placeholder(tf.float32, [None, 10])
```
Softmax Regression MNIST

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

>>> x = tf.placeholder(tf.float32, [None, 784])
>>> W = tf.Variable(tf.zeros([784, 10]))
>>> b = tf.Variable(tf.zeros([10]))
>>> y = tf.matmul(x, W) + b

>>> y_ = tf.placeholder(tf.float32, [None, 10])

>>> cross_entropy = tf.reduce_mean(tf.nn.softmax_cross_entropy_with_logits(labels=y_, logits=y))
>>> train_step = tf.train.GradientDescentOptimizer(0.5).minimize(cross_entropy)

>>> sess = tf.InteractiveSession()
>>> tf.global_variables_initializer().run()

>>> for _ in range(1000):
    batch_xs, batch_ys = mnist.train.next_batch(100)
    sess.run(train_step, feed_dict={x: batch_xs, y_: batch_ys})

>>> correct_prediction = tf.equal(tf.argmax(y, 1), tf.argmax(y_, 1))
>>> accuracy = tf.reduce_mean(tf.cast(correct_prediction, tf.float32))

>>> print(sess.run(accuracy, feed_dict={x: mnist.test.images, y_: mnist.test.labels}))
```

Here we define the solver and details like step size to minimize our error.

The values coming out of our matrix operations can have large, and negative values. We would like our solution vector to be conventional probabilities that sum to 1.0. An effective way to normalize our outputs is to use the popular **Softmax** function. Let's look at an example with just three possible digits:

<table>
<thead>
<tr>
<th>Digit</th>
<th>Output</th>
<th>Exponential</th>
<th>Normalized</th>
</tr>
</thead>
<tbody>
<tr>
<td>0</td>
<td>7.8</td>
<td>21.2</td>
<td>.73</td>
</tr>
<tr>
<td>1</td>
<td>-3.6</td>
<td>0.0</td>
<td>.00</td>
</tr>
<tr>
<td>2</td>
<td>2.9</td>
<td>7.8</td>
<td>.27</td>
</tr>
</tbody>
</table>

Given the sensible way we have constructed these outputs, the **Cross Entropy Loss** function is a very good way to define the error across all possibilities. Better than squared error, which we have been using until now.
Training Regression MNIST

Launch
Launch the model and initialize the variables.

Train
Do 1000 iterations with batches of 100 images, labels instead of whole dataset. This is stochastic.
Testing Regression MNIST

- **Argmax** selects index of highest value. We end up with a list of booleans showing matches.
- Reduce that list of 0s,1s and take the mean.
- Run the graph on the test dataset to determine accuracy. No solving involved.

Result is 92%.
You may be impressed. Or not. This was just a simple walkthrough of constructing a graph with Tensorflow and involved just one matrix of weights.

We can do much better using a real NN. We will even jump quite close to the state-of-the-art and use a Convolutional Neural Net.

This will have a multi-layer structure like the deep networks we considered earlier.

It will also take advantage of the actual 2D structure of the image that we ditched so cavalierly earlier.

It will include dropout! A surprising optimization to many.

It will also be cleaner in many ways than the example we just did. So if I didn’t tell you not to dwell too much on that intro example, unless you already really understand softmax regression:
Convolutional Net
Convolution

\[ O_6 = A_1 \cdot I_1 + A_2 \cdot I_2 + A_3 \cdot I_3 + A_4 \cdot I_5 + A_5 \cdot I_6 + A_6 \cdot I_7 + A_7 \cdot I_9 + A_8 \cdot I_{10} + A_9 \cdot I_{11} \]
Convolution

Boundary and Index Accounting!

\[ O_{17} = B_5 \cdot I_1 + B_6 \cdot I_2 + B_8 \cdot I_5 + B_9 \cdot I_6 \]
Straight Convolution

\[ \begin{bmatrix} -1 & -1 & -1 \\ -1 & 8 & -1 \\ -1 & -1 & -1 \end{bmatrix} \]

Edge Detector

Simplest Convolution Net

Courtesy: Chris Olah
Stacking Convolutions

Courtesy: Chris Olah
These are the filters from one convolution on one layer (from Krizhevsky et al. (2012)). Each filter has learned to detect a different type of feature.
Convolution Math

Each Convolutional Layer:

Inputs a volume of size $W_I \times H_I \times D_I$ (D is depth)

Requires four hyperparameters:
- Number of filters $K$
- their spatial extent $N$
- the stride $S$
- the amount of padding $P$

Produces a volume of size $W_O \times H_O \times D_O$

$$W_O = \frac{(W_I - N + 2P)}{S+1}\quad H_O = \frac{(H_I - F + 2P)}{S+1}\quad D_O = K$$

This requires $N \cdot N \cdot D_I$ weights per filter, for a total of $N \cdot N \cdot D_I \cdot K$ weights and $K$ biases

In the output volume, the $d$-th depth slice (of size $W_O \times H_O$) is the result of performing a convolution of the $d$-th filter over the input volume with a stride of $S$, and then offset by $d$-th bias.
from tensorflow.examples.tutorials.mnist import input_data

import tensorflow as tf

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

x = tf.placeholder(tf.float32, [None, 784])
y_ = tf.placeholder(tf.float32, [None, 10])

x_image = tf.reshape(x, [-1,28,28,1])

W_conv1 = tf.Variable(tf.truncated_normal([5, 5, 1, 32], stddev=0.1))
b_conv1 = tf.Variable(tf.constant(0.1, shape=[32]))
h_conv1 = tf.nn.relu(tf.nn.conv2d(x_image, W_conv1, strides=[1, 1, 1, 1], padding='SAME') + b_conv1)
h_pool1 = tf.nn.max_pool(h_conv1, ksize=[1, 2, 2, 1], strides=[1, 2, 2, 1], padding='SAME')

W_conv2 = tf.Variable(tf.truncated_normal([5, 5, 32, 64], stddev=0.1))
b_conv2 = tf.Variable(tf.constant(0.1, shape=[64]))
h_conv2 = tf.nn.relu(tf.nn.conv2d(h_pool1, W_conv2, strides=[1, 1, 1, 1], padding='SAME') + b_conv2)
h_pool2 = tf.nn.max_pool(h_conv2, ksize=[1, 2, 2, 1], strides=[1, 2, 2, 1], padding='SAME')

W_fc1 = tf.Variable(tf.truncated_normal([7 * 7 * 64, 1024], stddev=0.1))
b_fc1 = tf.Variable(tf.constant(0.1, shape=[1024]))
h_pool2_flat = tf.reshape(h_pool2, [-1, 7*7*64])
h_fc1 = tf.nn.relu(tf.matmul(h_pool2_flat, W_fc1) + b_fc1)

W_fc2 = tf.Variable(tf.truncated_normal([1024, 10], stddev=0.1))
b_fc2 = tf.Variable(tf.constant(0.1, shape=[10]))
keep_prob = tf.placeholder(tf.float32)

h_fc1_drop = tf.nn.dropout(h_fc1, keep_prob)
y_conv = tf.matmul(h_fc1_drop, W_fc2) + b_fc2

cross_entropy = tf.reduce_mean(tf.nn.softmax_cross_entropy_with_logits(labels=y_, logits=y_conv))
train_step = tf.train.AdamOptimizer(1e-4).minimize(cross_entropy)
correct_prediction = tf.equal(tf.argmax(y_conv,1), tf.argmax(y_,1))
accuracy = tf.reduce_mean(tf.cast(correct_prediction, tf.float32))

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

for i in range(20000):
  batch = mnist.train.next_batch(50)
  if i%100 == 0:
    train_accuracy = accuracy.eval(feed_dict={x:batch[0], y_: batch[1], keep_prob: 1.0})
    print('step %d, training accuracy %g' % (i, train_accuracy))
  train_step.run(feed_dict={x: batch[0], y_: batch[1], keep_prob: 0.5})

print('test accuracy %g' % accuracy.eval(feed_dict={x: mnist.test.images, y_: mnist.test.labels, keep_prob: 1.0}))
>>> from tensorflow.examples.tutorials.mnist import input_data
>>> import tensorflow as tf
>>> mnist = input_data.read_data_sets(".", one_hot=True)

>>> x = tf.placeholder(tf.float32, [None, 784])
>>> y_ = tf.placeholder(tf.float32, [None, 10])

>>> x_image = tf.reshape(x, [-1,28,28,1])

[batch, height, width, channels] -1 is TF for “unknown”
>>> from tensorflow.examples.tutorials.mnist import input_data
>>> import tensorflow as tf

>>> mnist = input_data.read_data_sets(".", one_hot=True)

>>> x = tf.placeholder(tf.float32, [None, 784])
>>> y_ = tf.placeholder(tf.float32, [None, 10])

>>> x_image = tf.reshape(x, [-1, 28, 28, 1])

>>> W_conv1 = tf.Variable(tf.truncated_normal([5, 5, 1, 32], stddev=0.1))
>>> b_conv1 = tf.Variable(tf.constant(0.1, shape=[32]))

>>> h_conv1 = tf.nn.relu(tf.nn.conv2d(x_image, W_conv1, strides=[1, 1, 1, 1], padding='SAME') + b_conv1)

>>> h_pool1 = tf.nn.max_pool(h_conv1, ksize=[1, 2, 2, 1], strides=[1, 2, 2, 1], padding='SAME')

Convolutional MNIST
The First Layer
We will have 32 5x5 filters in this layer
What values to initialize?
  Small random positive for weights
  Small constant for bias
>>> from tensorflow.examples.tutorials.mnist import input_data
>>> import tensorflow as tf

>>> mnist = input_data.read_data_sets(".", one_hot=True)

>>> x = tf.placeholder(tf.float32, [None, 784])
>>> y_ = tf.placeholder(tf.float32, [None, 10])

>>> x_image = tf.reshape(x, [-1,28,28,1])

>>> W_conv1 = tf.Variable(tf.truncated_normal([5, 5, 1, 32],
                                              stddev=0.1))
>>> b_conv1 = tf.Variable(tf.constant(0.1, shape=[32]))

>>> h_conv1 = tf.nn.relu(tf.nn.conv2d(x_image, W_conv1, strides=[1, 1, 1, 1], padding='SAME') + b_conv1)

The First Layer

TF will handle padding
More explicit in cuDNN and Caffe
Stride of 1x1
Must be same dims as X (just set depth,batch=1)
>>> from tensorflow.examples.tutorials.mnist import input_data
>>> import tensorflow as tf
>>> mnist = input_data.read_data_sets(".", one_hot=True)

>>> x = tf.placeholder(tf.float32, [None, 784])
>>> y_ = tf.placeholder(tf.float32, [None, 10])

>>> x_image = tf.reshape(x, [-1,28,28,1])

>>> W_conv1 = tf.Variable(tf.truncated_normal([5, 5, 1, 32], stddev=0.1))
>>> b_conv1 = tf.Variable(tf.constant(0.1, shape=[32]))

>>> h_conv1 = tf.nn.relu(tf.nn.conv2d(x_image, W_conv1, strides=[1, 1, 1, 1], padding='SAME') + b_conv1)

Convolutional MNIST
The First Layer

Add bias and apply our ReLU

Widely adopted around 2010!
>>> from tensorflow.examples.tutorials.mnist import input_data
>>> import tensorflow as tf

>>>
>>> mnist = input_data.read_data_sets('.', one_hot=True)

>>> x = tf.placeholder(tf.float32, [None, 784])
>>> y_ = tf.placeholder(tf.float32, [None, 10])

>>>
>>> x_image = tf.reshape(x, [-1,28,28,1])

>>> W_conv1 = tf.Variable(tf.truncated_normal([5, 5, 1, 32], stddev=0.1))
>>> b_conv1 = tf.Variable(tf.constant(0.1, shape=[32]))

>>> h_conv1 = tf.nn.relu(tf.nn.conv2d(x_image, W_conv1,strides=[1,1,1,1], padding='SAME') + b_conv1)

>>> h_pool1 = tf.nn.max_pool(h_conv1, ksize=[1, 2, 2, 1], strides=[1, 2, 2, 1], padding='SAME')

[batch, height, width, channels]
For window size and stride.

The image we will pass to the next layer is now 14x14.
>>> from tensorflow.examples.tutorials.mnist import input_data
>>> import tensorflow as tf
>>> mnist = input_data.read_data_sets(".", one_hot=True)
>>> x = tf.placeholder(tf.float32, [None, 784])
>>> y_ = tf.placeholder(tf.float32, [None, 10])

>>> x_image = tf.reshape(x, [-1, 28, 28, 1])

>>> W_conv1 = tf.Variable(tf.truncated_normal([5, 5, 1, 32], stddev=0.1))
>>> b_conv1 = tf.Variable(tf.constant(0.1, shape=[32]))

>>> h_conv1 = tf.nn.relu(tf.nn.conv2d(x_image, W_conv1, strides=[1, 1, 1, 1], padding='SAME') + b_conv1)
>>> h_pool1 = tf.nn.max_pool(h_conv1, ksize=[1, 2, 2, 1], strides=[1, 2, 2, 1], padding='SAME')
Now we have 32 features coming in, and we will use 64 on this layer.

The next layer will be getting a 7x7 image.
>>> from tensorflow.examples.tutorials.mnist import input_data
>>> import tensorflow as tf

>>> mnist = input_data.read_data_sets(".", one_hot=True)

>>> x = tf.placeholder(tf.float32, [None, 784])
>>> y_ = tf.placeholder(tf.float32, [None, 10])

>>> x_image = tf.reshape(x, [-1,28,28,1])

>>> W_conv1 = tf.Variable(tf.truncated_normal([5, 5, 1, 32], stddev=0.1))
>>> b_conv1 = tf.Variable(tf.constant(0.1, shape=[32]))
>>> h_conv1 = tf.nn.relu(tf.nn.conv2d(x_image, W_conv1, strides=[1, 1, 1, 1], padding='SAME') + b_conv1)

>>> h_pool1 = tf.nn.max_pool(h_conv1, ksize=[1, 2, 2, 1], strides=[1, 2, 2, 1], padding='SAME')

>>> W_conv2 = tf.Variable(tf.truncated_normal([5, 5, 32, 64], stddev=0.1))
>>> b_conv2 = tf.Variable(tf.constant(0.1, shape=[64]))
>>> h_conv2 = tf.nn.relu(tf.nn.conv2d(h_pool1, W_conv2, strides=[1, 1, 1, 1], padding='SAME') + b_conv2)

>>> h_pool2 = tf.nn.max_pool(h_conv2, ksize=[1, 2, 2, 1], strides=[1, 2, 2, 1], padding='SAME')

>>> W_fc1 = tf.Variable(tf.truncated_normal([7 * 7 * 64, 1024], stddev=0.1))
>>> b_fc1 = tf.Variable(tf.constant(0.1, shape=[1024]))

>>> h_pool2_flat = tf.reshape(h_pool2, [-1, 7 * 7 * 64])

>>> h_fc1 = tf.nn.relu(tf.matmul(h_pool2_flat, W_fc1) + b_fc1)

Convolutional MNIST

Fully Connected Layer

Now we can just flatten our 64 7x7 images into one big vector for the FC layer to analyze.

We will choose 1024 neurons for this layer.
>>> from tensorflow.examples.tutorials.mnist import input_data
>>> import tensorflow as tf

>>> mnist = input_data.read_data_sets(".", one_hot=True)

>>> x = tf.placeholder(tf.float32, [None, 784])
>>> y_ = tf.placeholder(tf.float32, [None, 10])

>>> x_image = tf.reshape(x, [-1,28,28,1])

>>> W_conv1 = tf.Variable(tf.truncated_normal([5, 5, 1, 32], stddev=0.1))
>>> b_conv1 = tf.Variable(tf.constant(0.1, shape=[32]))

>>> h_conv1 = tf.nn.relu(tf.nn.conv2d(x_image, W_conv1, strides=[1, 1, 1, 1], padding='SAME') + b_conv1)

>>> h_pool1 = tf.nn.max_pool(h_conv1, ksize=[1, 2, 2, 1], strides=[1, 2, 2, 1], padding='SAME')

>>> W_conv2 = tf.Variable(tf.truncated_normal([5, 5, 32, 64], stddev=0.1))
>>> b_conv2 = tf.Variable(tf.constant(0.1, shape=[64]))

>>> h_conv2 = tf.nn.relu(tf.nn.conv2d(h_pool1, W_conv2, strides=[1, 1, 1, 1], padding='SAME') + b_conv2)

>>> h_pool2 = tf.nn.max_pool(h_conv2, ksize=[1, 2, 2, 1], strides=[1, 2, 2, 1], padding='SAME')

>>> W_fc1 = tf.Variable(tf.truncated_normal([7 * 7 * 64, 1024], stddev=0.1))
>>> b_fc1 = tf.Variable(tf.constant(0.1, shape=[1024]))

>>> h_pool2_flat = tf.reshape(h_pool2, [-1, 7*7*64])

>>> h_fc1 = tf.nn.relu(tf.matmul(h_pool2_flat, W_fc1) + b_fc1)

>>> W_fc2 = tf.Variable(tf.truncated_normal([1024, 10], stddev=0.1))
>>> b_fc2 = tf.Variable(tf.constant(0.1, shape=[10]))

>>> keep_prob = tf.placeholder(tf.float32)

>>> h_fc1_drop = tf.nn.dropout(h_fc1, keep_prob)

>>> y_conv = tf.matmul(h_fc1_drop, W_fc2) + b_fc2

Convolutional MNIST

Dropout

We will have a final FC layer that gets us from 1024 neurons down to our 10 possible outputs.
Just like the regression model, we will define error as cross entropy and count our correct predictions. However this time we will use a sophisticated newer (2015) optimizer called ADAM. It is as simple as dropping it in.
>>> from tensorflow.examples.tutorials.mnist import input_data
>>> import tensorflow as tf

>>> mnist = input_data.read_data_sets(".", one_hot=True)

>>> x = tf.placeholder(tf.float32, [None, 784])

>>> y_ = tf.placeholder(tf.float32, [None, 10])

>>> x_image = tf.reshape(x, [-1, 28, 28, 1])

>>> W_conv1 = tf.Variable(tf.truncated_normal([5, 5, 1, 32], stddev=0.1))

>>> b_conv1 = tf.Variable(tf.constant(0.1, shape=[32]))

>>> h_conv1 = tf.nn.relu(tf.nn.conv2d(x_image, W_conv1, strides=[1, 1, 1, 1], padding='SAME') + b_conv1)

>>> h_pool1 = tf.nn.max_pool(h_conv1, ksize=[1, 2, 2, 1], strides=[1, 2, 2, 1], padding='SAME')

>>> W_conv2 = tf.Variable(tf.truncated_normal([5, 5, 32, 64], stddev=0.1))

>>> b_conv2 = tf.Variable(tf.constant(0.1, shape=[64]))

>>> h_conv2 = tf.nn.relu(tf.nn.conv2d(h_pool1, W_conv2, strides=[1, 1, 1, 1], padding='SAME') + b_conv2)

>>> h_pool2 = tf.nn.max_pool(h_conv2, ksize=[1, 2, 2, 1], strides=[1, 2, 2, 1], padding='SAME')

>>> W_fc1 = tf.Variable(tf.truncated_normal([7 * 7 * 64, 1024], stddev=0.1))

>>> b_fc1 = tf.Variable(tf.constant(0.1, shape=[1024]))

>>> h_pool2_flat = tf.reshape(h_pool2, [-1, 7*7*64])

>>> h_fc1 = tf.nn.relu(tf.matmul(h_pool2_flat, W_fc1) + b_fc1)

>>> W_fc2 = tf.Variable(tf.truncated_normal([1024, 10], stddev=0.1))

>>> b_fc2 = tf.Variable(tf.constant(0.1, shape=[10]))

>>> keep_prob = tf.placeholder(tf.float32)

>>> h_fc1_drop = tf.nn.dropout(h_fc1, keep_prob)

>>> y_conv = tf.matmul(h_fc1_drop, W_fc2) + b_fc2

>>> cross_entropy = tf.reduce_mean(tf.nn.softmax_cross_entropy_with_logits(labels=y_, logits=y_conv))

>>> train_step = tf.train.AdamOptimizer(1e-4).minimize(cross_entropy)

>>> correct_prediction = tf.equal(tf.argmax(y_conv,1), tf.argmax(y_,1))

>>> accuracy = tf.reduce_mean(tf.cast(correct_prediction, tf.float32))

>>> sess = tf.InteractiveSession()

>>> sess.run(tf.global_variables_initializer())

>>> for i in range(20000):

>>>   batch = mnist.train.next_batch(50)

>>>   if i%100 == 0:

>>>     train_accuracy = accuracy.eval(feed_dict={x:batch[0], y_: batch[1], keep_prob: 1.0})

>>>     print("step %d, training accuracy %g"%(i, train_accuracy))

>>>   train_step.run(feed_dict={x: batch[0], y_: batch[1], keep_prob: 0.5})

>>> test accuracy 0.9915
>>> from tensorflow.examples.tutorials.mnist import input_data
>>> import tensorflow as tf
>>> mnist = input_data.read_data_sets(".", one_hot=True)
>>> x = tf.placeholder(tf.float32, [None, 784])
>>> y_ = tf.placeholder(tf.float32, [None, 10])

>>> x_image = tf.reshape(x, [-1, 28, 28, 1])

>>> W_conv1 = tf.Variable(tf.truncated_normal([5, 5, 1, 32], stddev=0.1))
>>> b_conv1 = tf.Variable(tf.constant(0.1, shape=[32]))
>>> h_conv1 = tf.nn.relu(tf.nn.conv2d(x_image, W_conv1, strides=[1, 1, 1, 1], padding='SAME') + b_conv1)
>>> h_pool1 = tf.nn.max_pool(h_conv1, ksize=[1, 2, 2, 1], strides=[1, 2, 2, 1], padding='SAME')

>>> W_conv2 = tf.Variable(tf.truncated_normal([5, 5, 32, 64], stddev=0.1))
>>> b_conv2 = tf.Variable(tf.constant(0.1, shape=[64]))
>>> h_conv2 = tf.nn.relu(tf.nn.conv2d(h_pool1, W_conv2, strides=[1, 1, 1, 1], padding='SAME') + b_conv2)
>>> h_pool2 = tf.nn.max_pool(h_conv2, ksize=[1, 2, 2, 1], strides=[1, 2, 2, 1], padding='SAME')

>>> W_fc1 = tf.Variable(tf.truncated_normal([7 * 7 * 64, 1024], stddev=0.1))
>>> b_fc1 = tf.Variable(tf.constant(0.1, shape=[1024]))
>>> h_pool2_flat = tf.reshape(h_pool2, [-1, 7*7*64])
>>> h_fc1 = tf.nn.relu(tf.matmul(h_pool2_flat, W_fc1) + b_fc1)

>>> W_fc2 = tf.Variable(tf.truncated_normal([1024, 10], stddev=0.1))
>>> b_fc2 = tf.Variable(tf.constant(0.1, shape=[10]))
>>> keep_prob = tf.placeholder(tf.float32)
>>> h_fc1_drop = tf.nn.dropout(h_fc1, keep_prob)
>>> y_conv = tf.matmul(h_fc1_drop, W_fc2) + b_fc2

>>> cross_entropy = tf.reduce_mean(tf.nn.softmax_cross_entropy_with_logits(labels=y_, logits=y_conv))
>>> train_step = tf.train.AdamOptimizer(1e-4).minimize(cross_entropy)
>>> correct_prediction = tf.equal(tf.argmax(y_conv, 1), tf.argmax(y_, 1))
>>> accuracy = tf.reduce_mean(tf.cast(correct_prediction, tf.float32))

>>> sess = tf.InteractiveSession()
>>> sess.run(tf.global_variables_initializer())
>>> for i in range(20000):
...    batch = mnist.train.next_batch(50)
...    if i%100 == 0:
...        train_accuracy = accuracy.eval(feed_dict={x: batch[0], y_: batch[1], keep_prob: 1.0})
...        print("step %d, training accuracy %g\n" % (i, train_accuracy))
...        train_step.run(feed_dict={x: batch[0], y_: batch[1], keep_prob: 0.5})
...    print("test accuracy %g\n" % accuracy.eval(feed_dict={x: mnist.test.images, y_: mnist.test.labels, keep_prob: 1.0}))
>>> test accuracy 0.9915

Convolutional MNIST

Testing

We finally test against a whole difference set of test data (that is what mnist.test returns) and find that we are:

99.15% Accurate!
Other Significant Architectures

Recurrent Neural Net
- Cycles back previous inputs (feedback)
- Like short term memory
- Adds context (like for language processing)
- Current advancement is Long Short Term Memory
  - bit more complex
  - very effective for certain tasks

Residual Neural Net
- Helps preserve reasonable gradients for very deep networks
- Very effective at imagery

Very Deep Neural Net
- 100s of layers, Pushing 1000
That is paraphrasing Yann LeCun, the godfather of Deep Learning.

If it feels like this is an oddly empirical branch of computer science, you are spot on.

Many of these techniques were developed through experimentation, and many of them are not amenable to classical analysis. A theoretician would suggest that non-convex loss functions are at the heart of the matter, and that situation isn’t getting better as many of the latest techniques have made this much worse.

You may also have noticed that many of the techniques we have used today have very recent provenance. This is true throughout the field. Rarely is the undergraduate researcher so reliant upon results groundbreaking papers of a few years ago.
The reason that we have attempted this ridiculously ambitious workshop is that the field has reached a level of maturity where the tools can encapsulate much of the complexity in black boxes.

One should not be ashamed to use a well-designed black box. Indeed it would be foolish for you to write your own FFT or eigensolver math routines. Besides wasting time, you won’t reach the efficiency of a professionally tuned tool.

On the other hand, most programmers using those tools have been exposed to the basics of the theory, and could dig out their old textbook explanation of how to cook up an FFT. This provides some baseline level of judgement in using tools provided by others.

You are treading on newer ground. However this means there are still major discoveries to be made using these tools in fresh applications.

Any one particularly exciting dimension to this whole situation is that exploring hyperparameters has been very fruitful. The toolbox allows you to do just that.
You have a plethora of alternatives available as well. You are now in a position to appreciate some comparisons.

<table>
<thead>
<tr>
<th>Package</th>
<th>Applications</th>
<th>Language</th>
<th>Strengths</th>
</tr>
</thead>
<tbody>
<tr>
<td>Tensorflow</td>
<td>Neural Nets</td>
<td>Python, C++</td>
<td>Very popular.</td>
</tr>
<tr>
<td>Caffe</td>
<td>Neural Nets</td>
<td>Python, C++</td>
<td>Many research projects and publications.</td>
</tr>
<tr>
<td>Spark MLLIB</td>
<td>Classification, Regression, Clustering, etc.</td>
<td>Python, Scala, Java, R</td>
<td>Very scalable. Widely used in serious applications.</td>
</tr>
<tr>
<td>Scikit-Learn</td>
<td>Classification, Regression, Clustering</td>
<td>Python</td>
<td></td>
</tr>
<tr>
<td>cuDNN</td>
<td>Neural Nets</td>
<td>C++, GPU-based</td>
<td>Used in many other frameworks: TF, Caffe, etc.</td>
</tr>
<tr>
<td>Theano</td>
<td>Neural Nets</td>
<td>Python</td>
<td>Lower level numerical routines. NumPy-esque.</td>
</tr>
<tr>
<td>Torch</td>
<td>Neural Nets</td>
<td>Lua (PyTorch=Python)</td>
<td>Dynamic graphs (variable length input/output) good for RNN.</td>
</tr>
<tr>
<td>Keras</td>
<td>Neural Nets</td>
<td>Python (on top of TF, Theano)</td>
<td>Higher level approach.</td>
</tr>
<tr>
<td>Digits</td>
<td>Neural Nets</td>
<td>“Caffe”, GPU-based</td>
<td>Used with other frameworks (only Caffe at moment).</td>
</tr>
</tbody>
</table>
Deep Learning has had so many recent successes that this is more a discussion starter than a comprehensive list. Open up a newspaper for new and exciting applications. Here are some commercially significant applications:

- Handwriting Recognition
- Language Translation
- Speech Recognition
- Image Classification
- Medical Diagnosis
  - Classification: which pixel tumor, which is not?
- Autonomous Driving
  - Classification: which pixel is road, which is pedestrian?
Exercises

We are going to leave you with a few substantial problems that you are now equipped to tackle. Feel free to use your extended workshop access to work on these, and remember that additional time is an easy Startup Allocation away. Of course everything we have done is standard and you can work on these problems in any reasonable environment.

CIFAR
The CIFAR-10 dataset consists of 60,000 32x32 colour images in 10 classes (airplane, auto, bird, cat, dog, ship, etc.) with 6,000 images per class. There are 50,000 training images and 10000 test images.

ImageNet
150,000 photographs, collected from flickr and other search engines, hand labeled with the presence or absence of 1000 object categories. Competition: http://image-net.org/challenges/LSVRC/2017/

Kaggle Challenge
Many datasets of great diversity (crime, plants, sports, stocks, etc). Competitions: https://www.kaggle.com/competitions
There are always multiple currently running competitions you can enter. Competitions: https://www.kaggle.com/competitions
This talk has benefited from the generous use of materials from NVIDIA and Christopher Olah in particular.

The NVIDIA materials were drawn from their excellent Deep Learning Institute

https://developer.nvidia.com/teaching-kits

Christopher Olah’s blog is insightful and not to be missed if you are interested in this field.

http://colah.github.io/

Other materials used as credited.

Any code examples used were substantially modified from the original.

Anything not otherwise mentioned follows Apache License 2.0.