Deep Learning

In An Afternoon

John Urbanic
Parallel Computing Scientist
Pittsburgh Supercomputing Center

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:

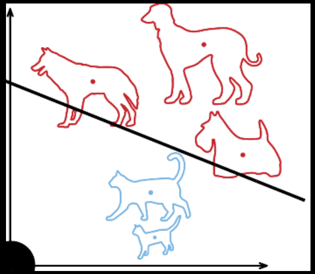
- Widespread parallel computing (GPUs)
- Big data training sets

Two Perspectives

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

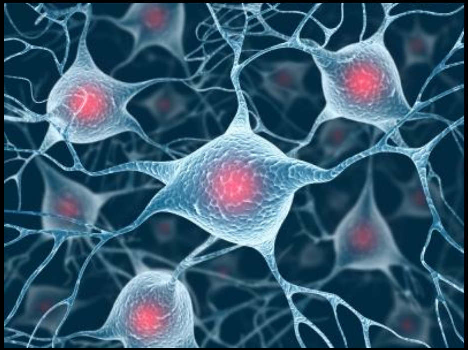
- Inspired by biological models.

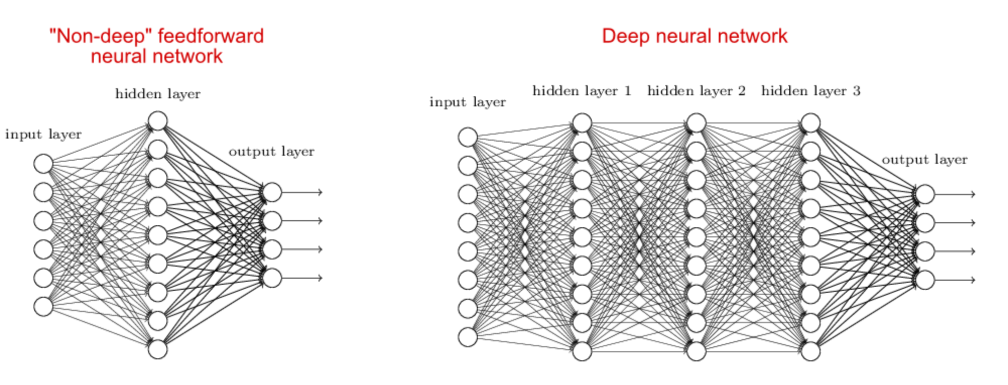
- An evolution of classic ML techniques (the perceptron).



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

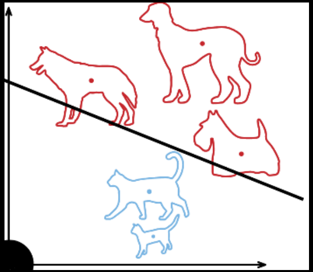




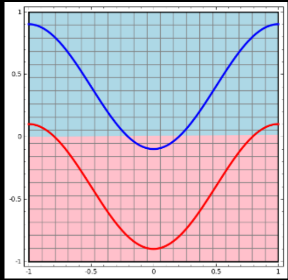
$$M = \begin{bmatrix} 0 & 1 & 0 & 0 & 0 & 0 & 1 & 0 & 0 & 1 & 0 & 0 & 0 \\ 1 & 0 & 1 & 0 & 0 & 0 & 1 & 1 & 0 & 1 & 0 & 1 & 0 \\ 0 & 1 & 0 & 1 & 0 & 0 & 0 & 1 & 1 & 1 & 1 & 0 & 1 \\ 0 & 0 & 1 & 0 & 0 & 0 & 0 & 0 & 1 & 1 & 0 & 0 & 0 \\ 0 & 0 & 0 & 0 & 0 & 1 & 0 & 0 & 0 & 0 & 0 & 1 & 0 \\ 0 & 0 & 1 & 1 & 0 & 0 & 0 & 0 & 0 & 1 & 0 & 0 & 0 \\ 1 & 1 & 0 & 0 & 0 & 0 & 0 & 0 & 0 & 1 & 0 & 0 & 0 \\ 0 & 1 & 1 & 0 & 0 & 0 & 0 & 0 & 0 & 1 & 1 & 1 & 1 \\ 0 & 0 & 0 & 0 & 1 & 0 & 0 & 0 & 0 & 0 & 0 & 1 & 0 \\ 1 & 1 & 1 & 0 & 0 & 0 & 1 & 1 & 0 & 0 & 0 & 1 & 0 \\ 0 & 0 & 1 & 0 & 0 & 0 & 0 & 1 & 0 & 0 & 0 & 0 & 1 \\ 0 & 1 & 1 & 0 & 1 & 0 & 0 & 1 & 1 & 1 & 0 & 0 & 0 \\ 0 & 0 & 1 & 0 & 0 & 0 & 0 & 1 & 0 & 0 & 1 & 0 & 0 \end{bmatrix}$$

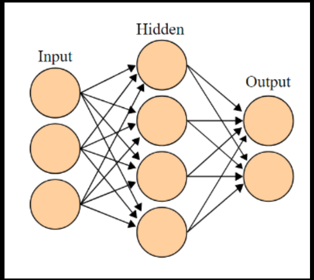
As a Highly Dimensional Non-linear Classifier

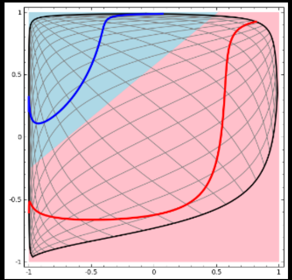
Perceptron



Network







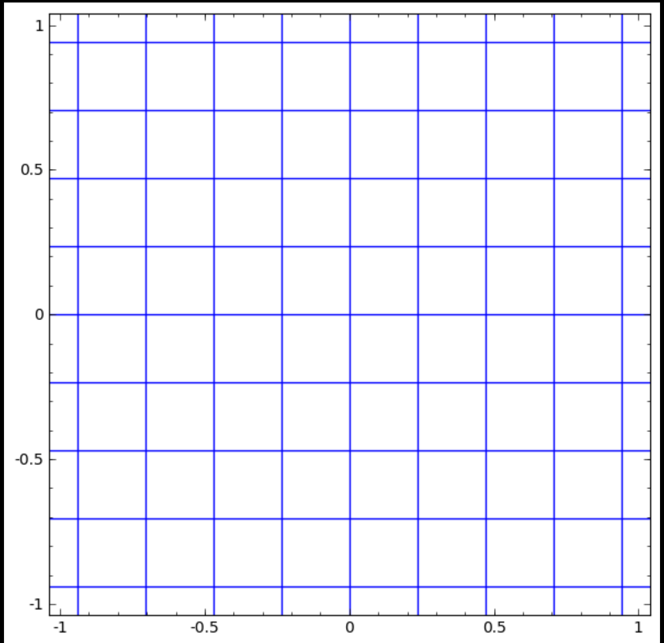
No Hidden Layer
Linear

>1 Hidden Layers
Nonlinear

Courtesy: Chris Olah

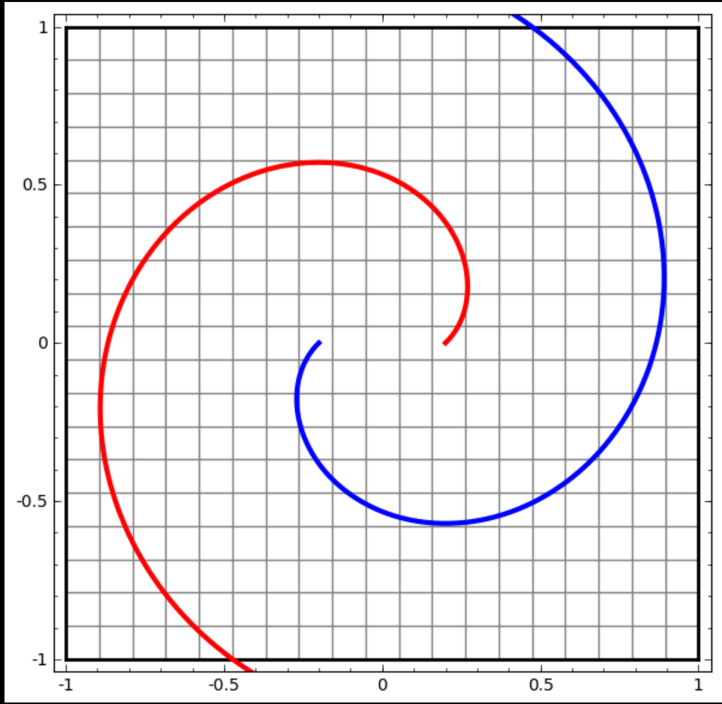
Linear + Nonlinear

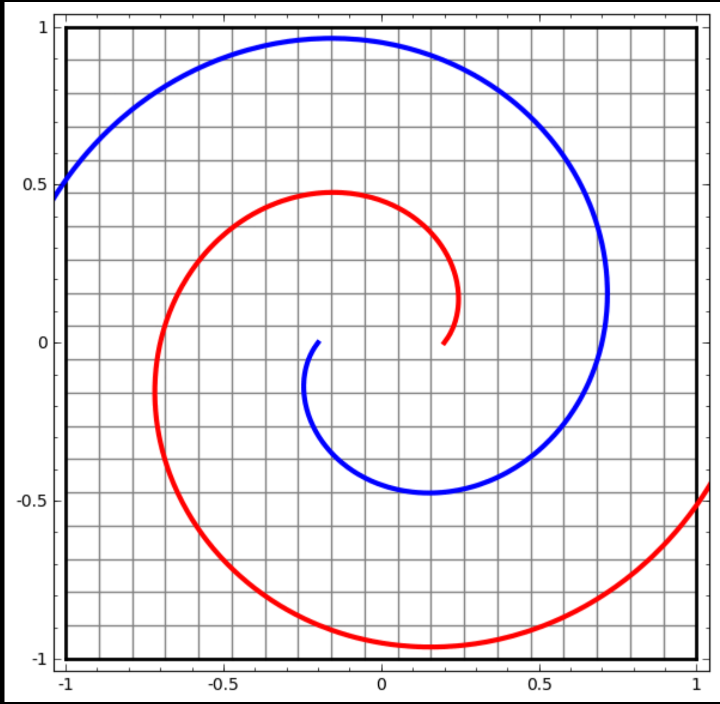
The magic formula for a neural net is that, at each layer, we apply linear operations (which look naturally like linear algebra matrix operations) and then pipe the final result through some kind of final nonlinear activation function. The combination of the two allows us to do very general transforms.



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.

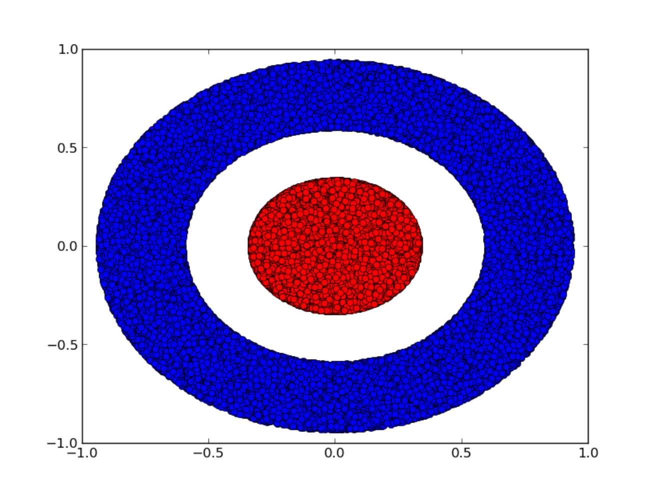




Courtesy: Chris Olah

Width of Network

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:

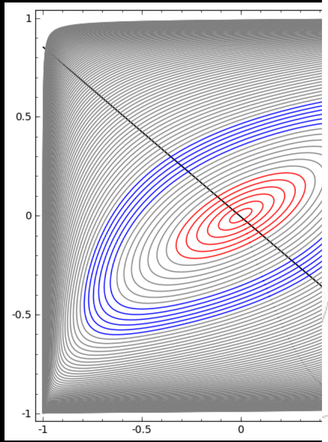


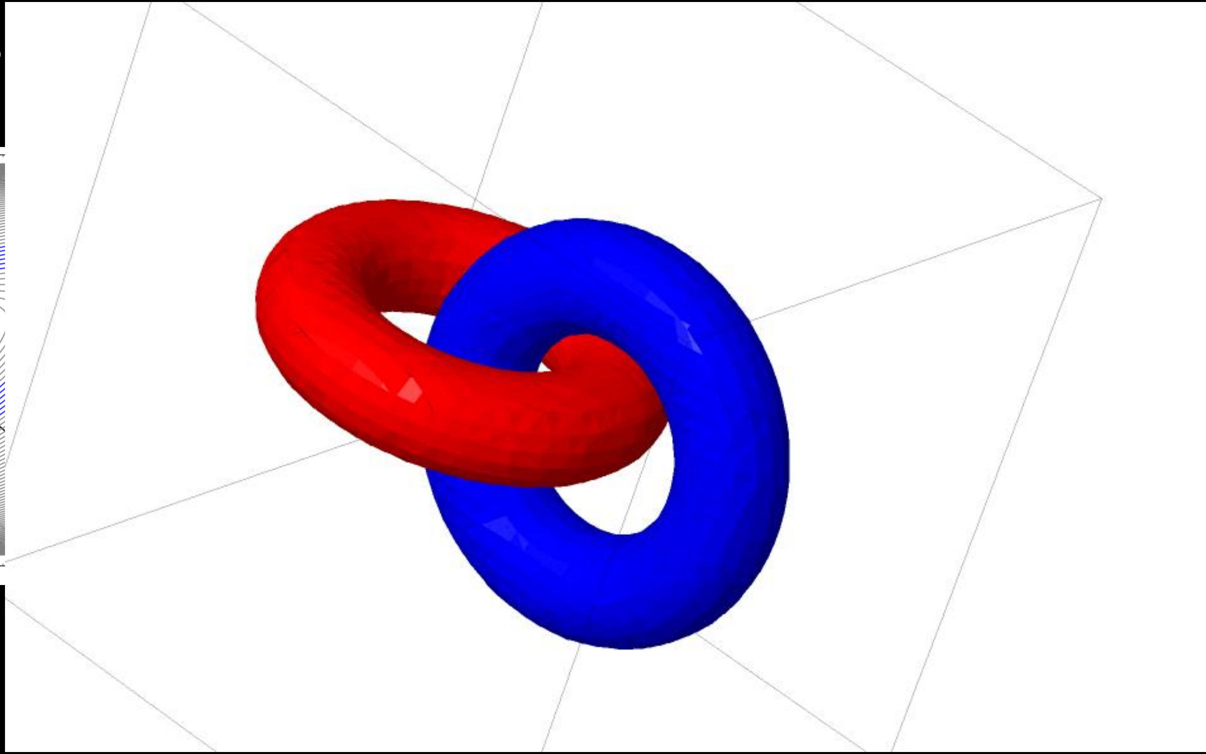
Can a neural net do this with twisting and deforming? What good does it do to have more than two dimensions with a 2D dataset?

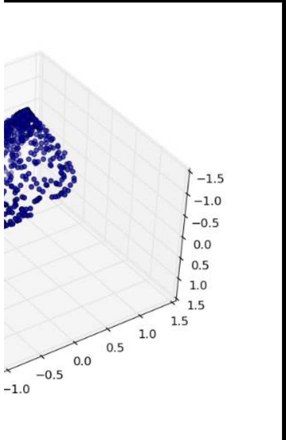
Courtesy: Chris Olah

Working In Higher Dimensions

It takes at least 3







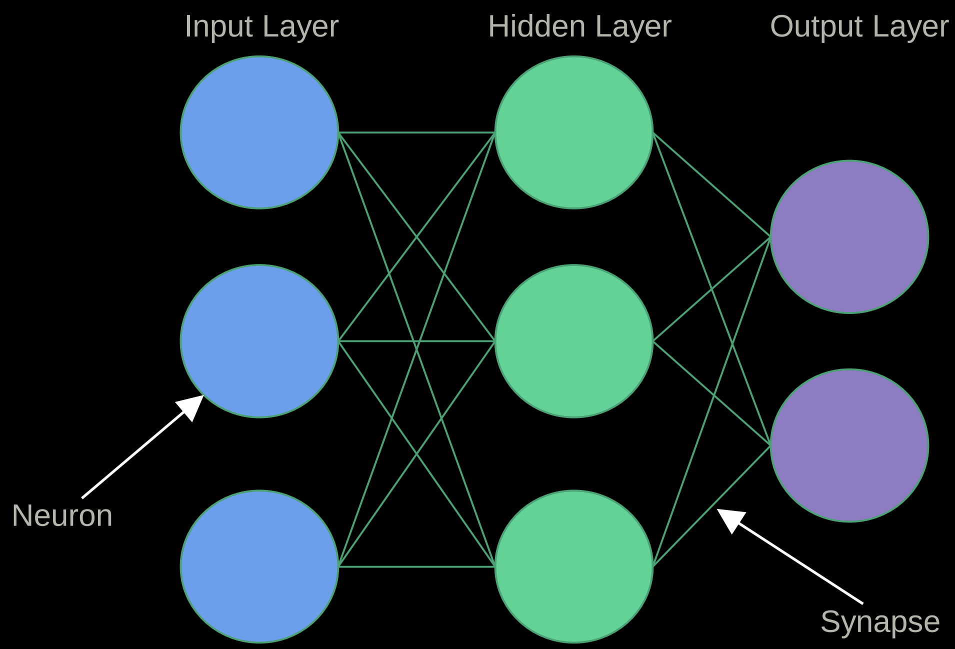
Trying

s in 3D

Greater depth allows us to stack these operations, and can be very effective. The gains from depth are harder to characterize.

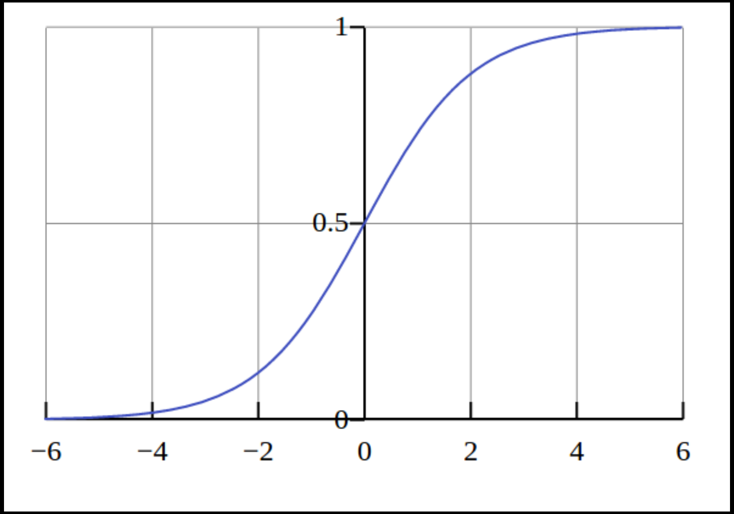
Courtesy: Chris Olah

Basic NN Architecture



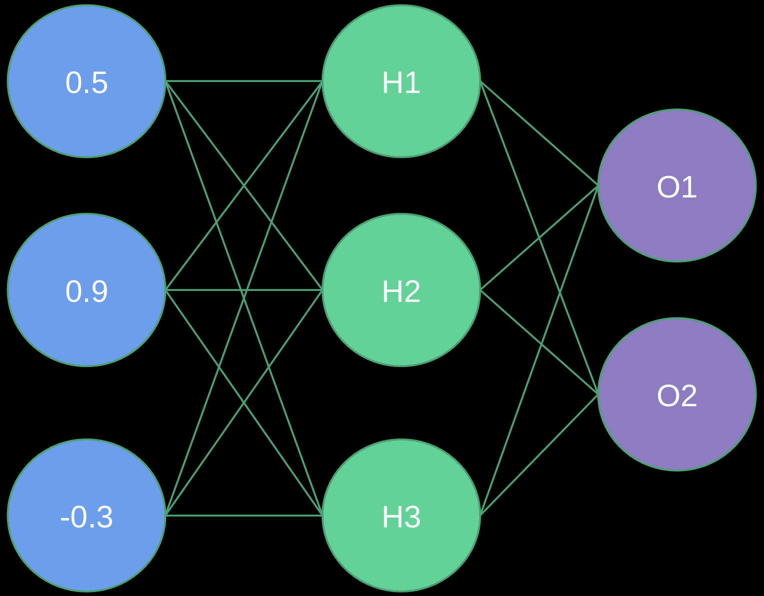
Activation Function

- Neurons apply activation function to their inputs.
- Activation functions are typically non-linear.
- The sigmoid function produces a value between 0 and 1 (so it is often used when a probability is desired)
- 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



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

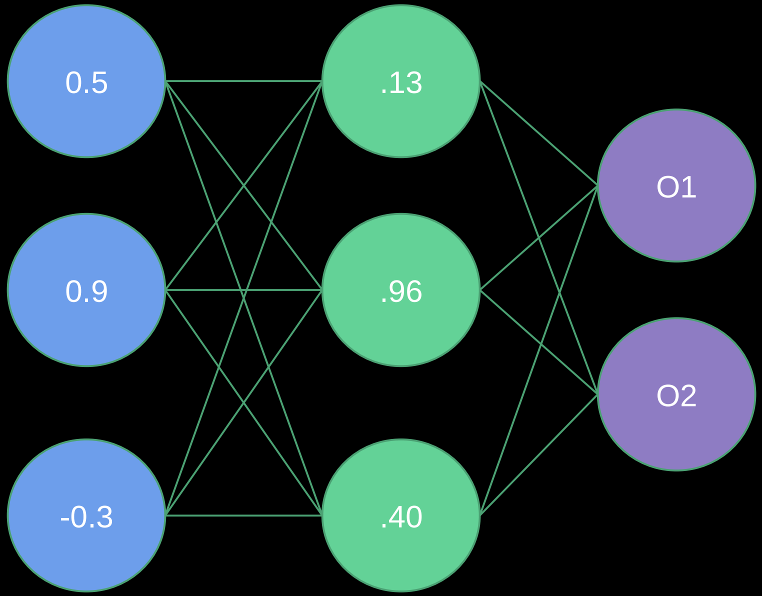
Inference



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)

Inference

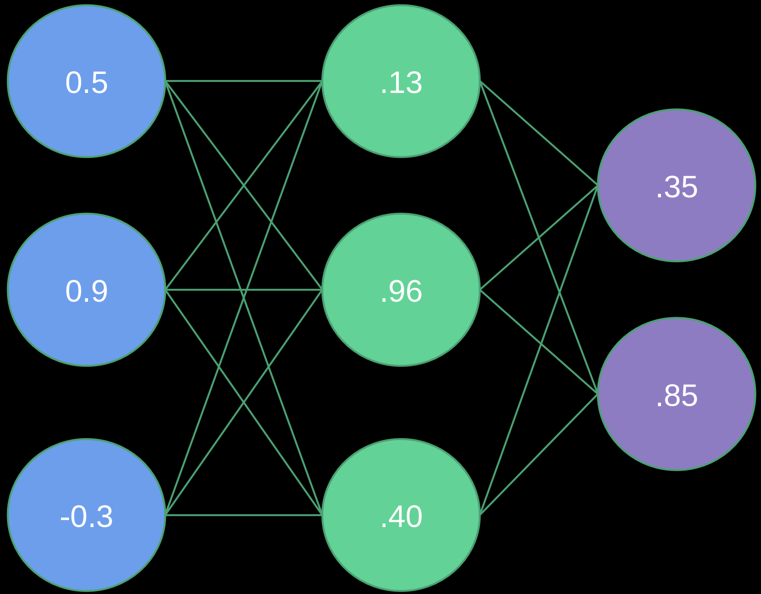


H1 Weights = (1.0, -2.0, 2.0)
H2 Weights = (2.0, 1.0, -4.0)
H3 Weights = (1.0, -1.0, 0.0)

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

H1 = S(0.5 * 1.0 + 0.9 * -2.0 + -0.3 * 2.0) = S(-1.9) = .13
H2 = S(0.5 * 2.0 + 0.9 * 1.0 + -0.3 * -4.0) = S(3.1) = .96
H3 = S(0.5 * 1.0 + 0.9 * -1.0 + -0.3 * 0.0) = S(-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 = S(.13 * -3.0 + .96 * 1.0 + .40 * -3.0) = S(-.63) = .35
O1 = S(.13 * 0.0 + .96 * 1.0 + .40 * 2.0) = S(1.76) = .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)

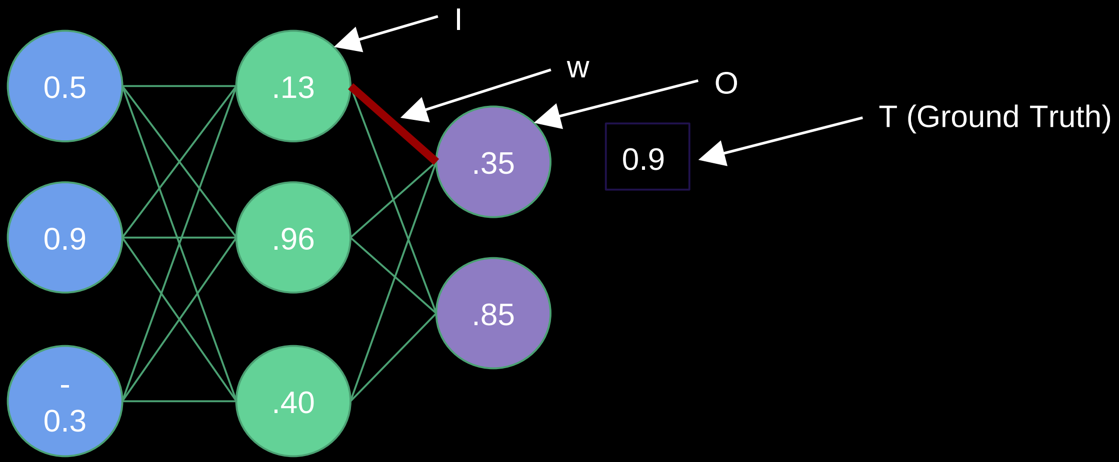
$$S\left(\underbrace{\begin{bmatrix} 1.0 & -2.0 & 2.0 \\ 2.0 & 1.0 & -4.0 \\ 1.0 & -1.0 & 0.0 \end{bmatrix}}_{\text{Hidden Layer Weights}} * \underbrace{\begin{bmatrix} 0.5 \\ 0.9 \\ -0.3 \end{bmatrix}}_{\text{Inputs}}\right) = S\left(\begin{bmatrix} -1.9 & 3.1 & -0.4 \end{bmatrix}\right) = \underbrace{\begin{bmatrix} .13 & .96 & 0.4 \end{bmatrix}}_{\text{Hidden Layer Outputs}}$$

Now this looks like something that we can pump through a GPU.

Training Neural Networks (Backpropogation)

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

Finding the Derivative



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

$$\frac{\partial E}{\partial w} = .13 \cdot (.35 - .9) \cdot .35 \cdot (1 - .35)$$

For Sigmoid

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

MNIST

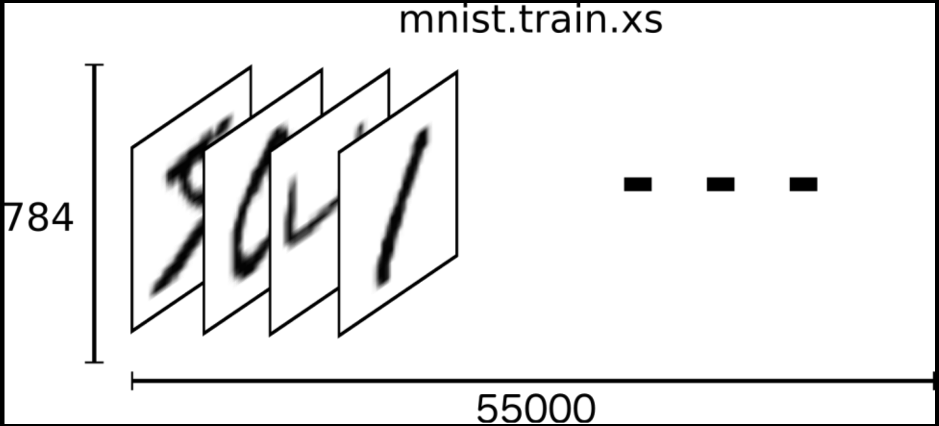
We now know enough to attempt a problem. Only because the Tensorflow framework fills in a lot of the details that we have glossed over. That is one of its functions.

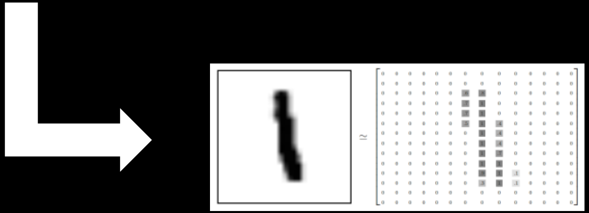
Our problem will be character recognition. We will learn to read handwritten digits by training on a large set of 28x28 greyscale samples.

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

MNIST Data

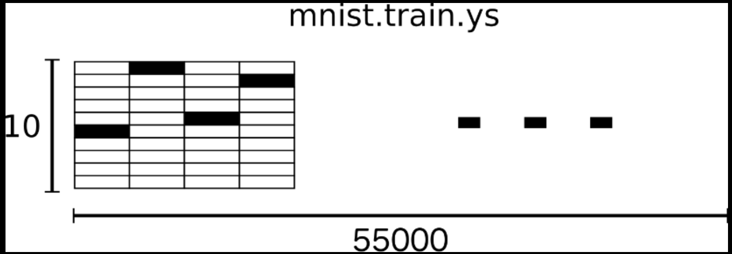
Specifically we will have a file with 55,000 of these numbers.





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
```

Simple MNIST

```
$ 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................
```

Only “mystery” code in whole workshop!

Just reads in files as we just discussed, in batches. Easy to do but a slight digression.

```
>>>>>> x , y = mnist.train.next_batch(2)
>>> y[0]
array([ 0.,  0.,  0.,  0.,  1.,  0.,  0.,  0.,  0.,  0.])
>>> x[0]
array([ 0.        ,  0.        ,  0.        ,  0.        ,  0.        ,
 0.        ,  0.        ,  0.        ,  0.        ,  0.        ,
 0.        ,  0.        ,  0.        ,  0.        ,  0.        ,
 0.        ,  0.        ,  0.        ,  0.02352941,  0.76470596,
 0.99607849,  1.        ,  0.93725497,  0.1137255 ,  0.        ,
 0.        ,  0.        ,  0.        ,  0.        ,  0.        ,
...
...
...
```

tf.nn
Overview
all_candidate_sampler
atrous_conv2d
atrous_conv2d_transpose
avg_pool
avg_pool3d
batch_normalization
batch_norm_with_global_norma...
bias_add
bidirectional_dynamic_rnn
compute_accidental_hits
conv1d
conv2d
conv2d_backprop_filter
conv2d_backprop_input
conv2d_transpose
conv3d
conv3d_backprop_filter_v2
conv3d_transpose
convolution
crelu
ctc_beam_search_decoder
ctc_greedy_decoder
ctc_loss
depthwise_conv2d
depthwise_conv2d_native
depthwise_conv2d_native_back...
depthwise_conv2d_native_back...
dilation2d
dropout
dynamic_rnn
elu
embedding_lookup
embedding_lookup_sparse
erosion2d
fixed_unigram_candidate_samp...
fractional_avg_pool
fractional_max_pool
fused_batch_norm
in_top_k
l2_loss
l2_normalize
learned_unigram_candidate_sa...
local_response_normalization

tf.nn.conv2d

Contents
tf.nn.conv2d

tf.nn.conv2d

```
conv2d(
    input,
    filter,
    strides,
    padding,
    use_cudnn_on_gpu=None,
    data_format=None,
    name=None
)
```

Defined in `tensorflow/python/ops/gen_nn_ops.py`.

See the guide: Neural Network > Convolution

Computes a 2-D convolution given 4-D `input` and `filter` tensors.

Given an input tensor of shape `[batch, in_height, in_width, in_channels]` and a filter / kernel tensor of shape `[filter_height, filter_width, in_channels, out_channels]`, this op performs the following:

1. Flattens the filter to a 2-D matrix with shape `[filter_height * filter_width * in_channels, output_channels]`.
2. Extracts image patches from the input tensor to form a *virtual* tensor of shape `[batch, out_height, out_width, filter_height * filter_width * in_channels]`.
3. For each patch, right-multiplies the filter matrix and the image patch vector.

In detail, with the default NHWC format,

```
output[b, i, j, k] =
    sum_{di, dj, q} input[b, strides[1] * i + di, strides[2] * j + dj, q] *
                    filter[di, dj, q, k]
```

Must have `strides[0] = strides[3] = 1`. For the most common case of the same horizontal and vertices strides, `strides = [1, stride, stride, 1]`.

Args:

- `input`: A `Tensor`. Must be one of the following types: `half`, `float32`, `float64`. A 4-D tensor. The dimension order is interpreted according to the value of `data_format`, see below for details.
- `filter`: A `Tensor`. Must have the same type as `input`. A 4-D tensor of shape `[filter_height, filter_width, in_channels, out_channels]`
- `strides`: A list of `ints`. 1-D tensor of length 4. The stride of the sliding window for each dimension of `input`. The dimension order is determined by the value of `data_format`, see below for details.

The API is well documented.

That is terribly unusual.

Softmax Regression MNIST

```
$ 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])
>>>
```

Placeholder
We will use TF placeholders for inputs and outputs. We will use TF Variables for persistent data that we can calculate. NONE means this dimension can be any length.

Image is 784 vector
We have flattened our 28x28 image to a 1-D 784 vector. You will encounter this simplification frequently.

b (Bias)
A bias is often added across all inputs to eliminate some independent “background”.

Softmax Regression MNIST

```
>>> 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}))
```

Error

We need to define our error so that our solver can minimize it. We will use "cross entropy", which works well comparing these kind of 1-hot vector labels.

We also use a "softmax" activation. It has some nice properties when paired with cross entropy and assures that the total outputs sum to 1.

At this point we have defined our model.

Solver

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

Softmax Regression MNIST

```
>>> 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}))
```

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.

Softmax Regression MNIST

```
>>> 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}))
0.9183
```

Results

- 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%.

92%

You may be impressed. Or not. This was just a simple walkthrough of constructing a graph with Tensorflow.

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.

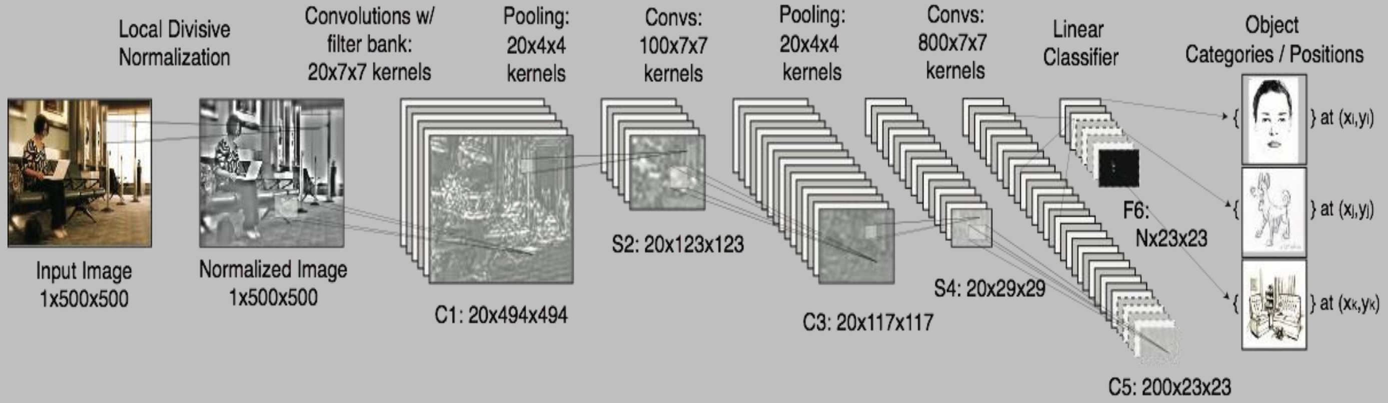
It will also take advantage of the actual 2D structure of the image that we ditched so cavalierly earlier.

It will include dropout! A surprising optimization to many.

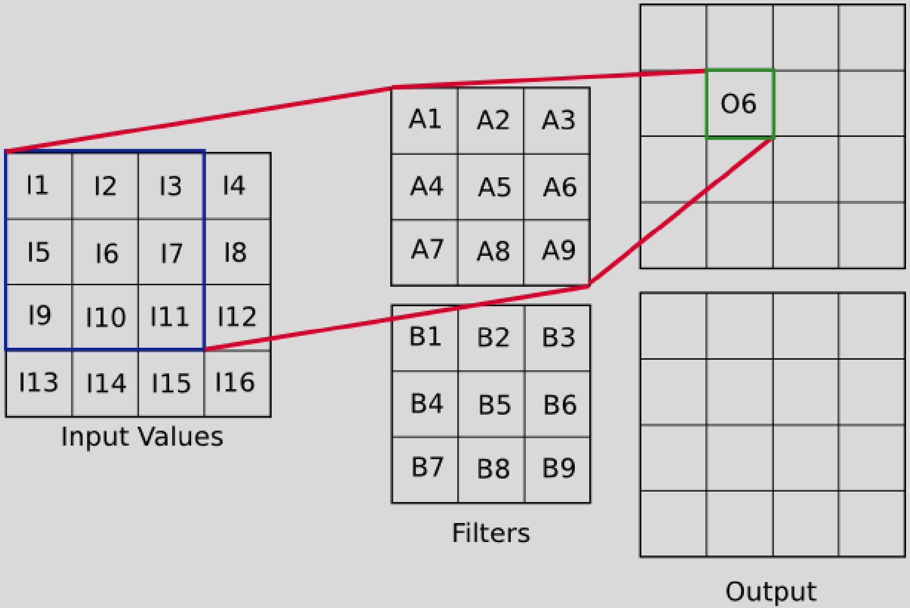
It will also be cleaner in many ways than the example we just did. So if I didn't tell you not to dwell too much on that intro example, unless you already really understand softmax regression:

Don't dwell too much on that intro example!

Convolutional Net



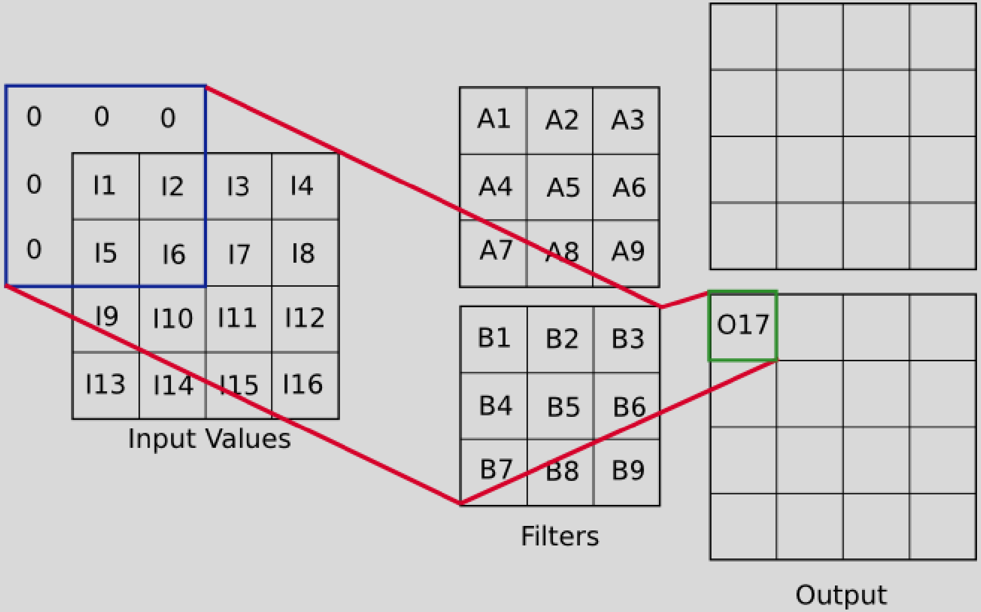
Convolution



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

Convolution

Boundary and Index Accounting!



$$O_{17} = B_5 \cdot I_1 + B_6 \cdot I_2 + B_8 \cdot I_5 + B_9 \cdot I_6$$

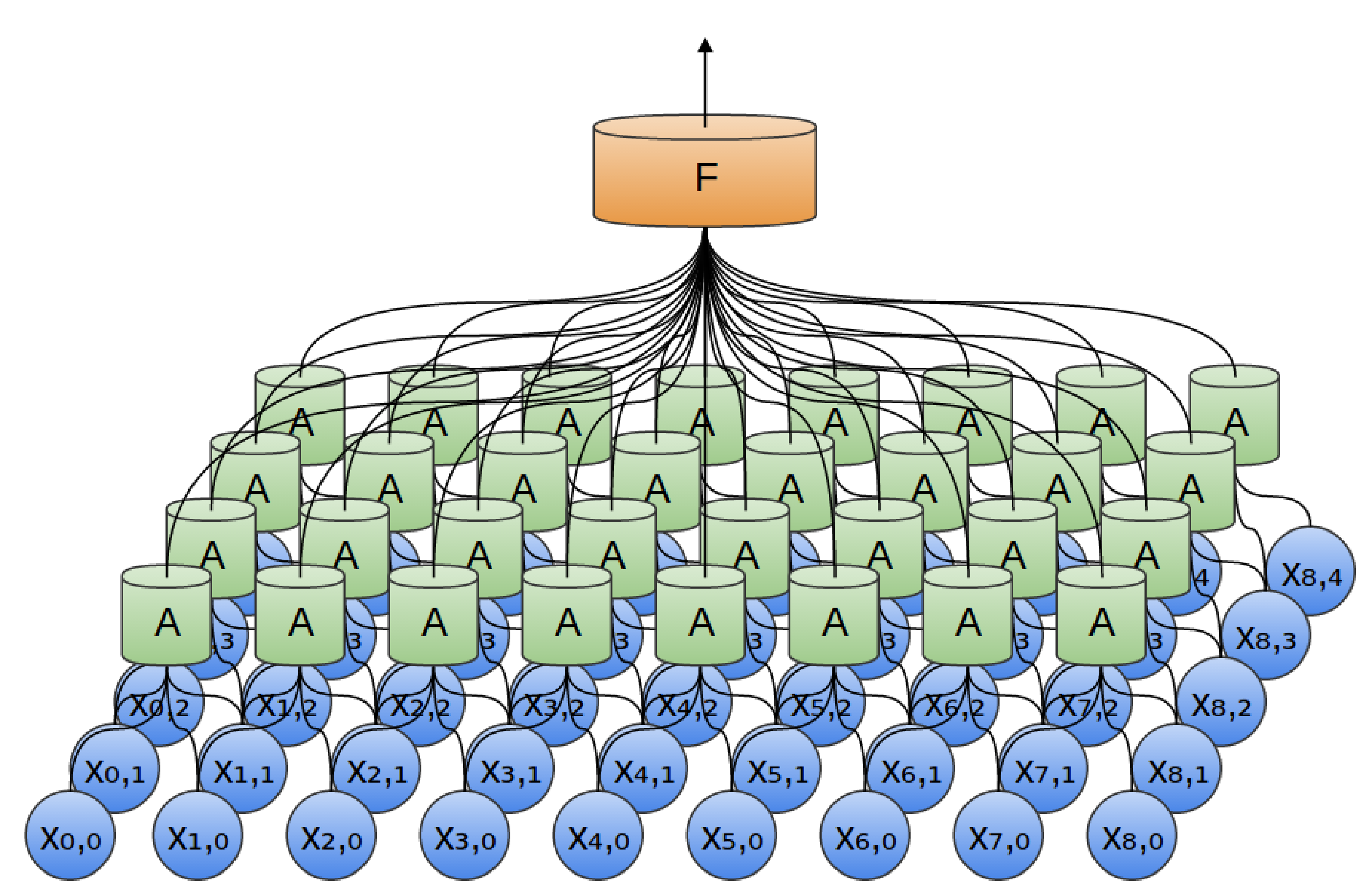
Straight Convolution

$$+ \begin{bmatrix} -1 & -1 & -1 \\ -1 & 8 & -1 \\ -1 & -1 & -1 \end{bmatrix} =$$

Edge Detector

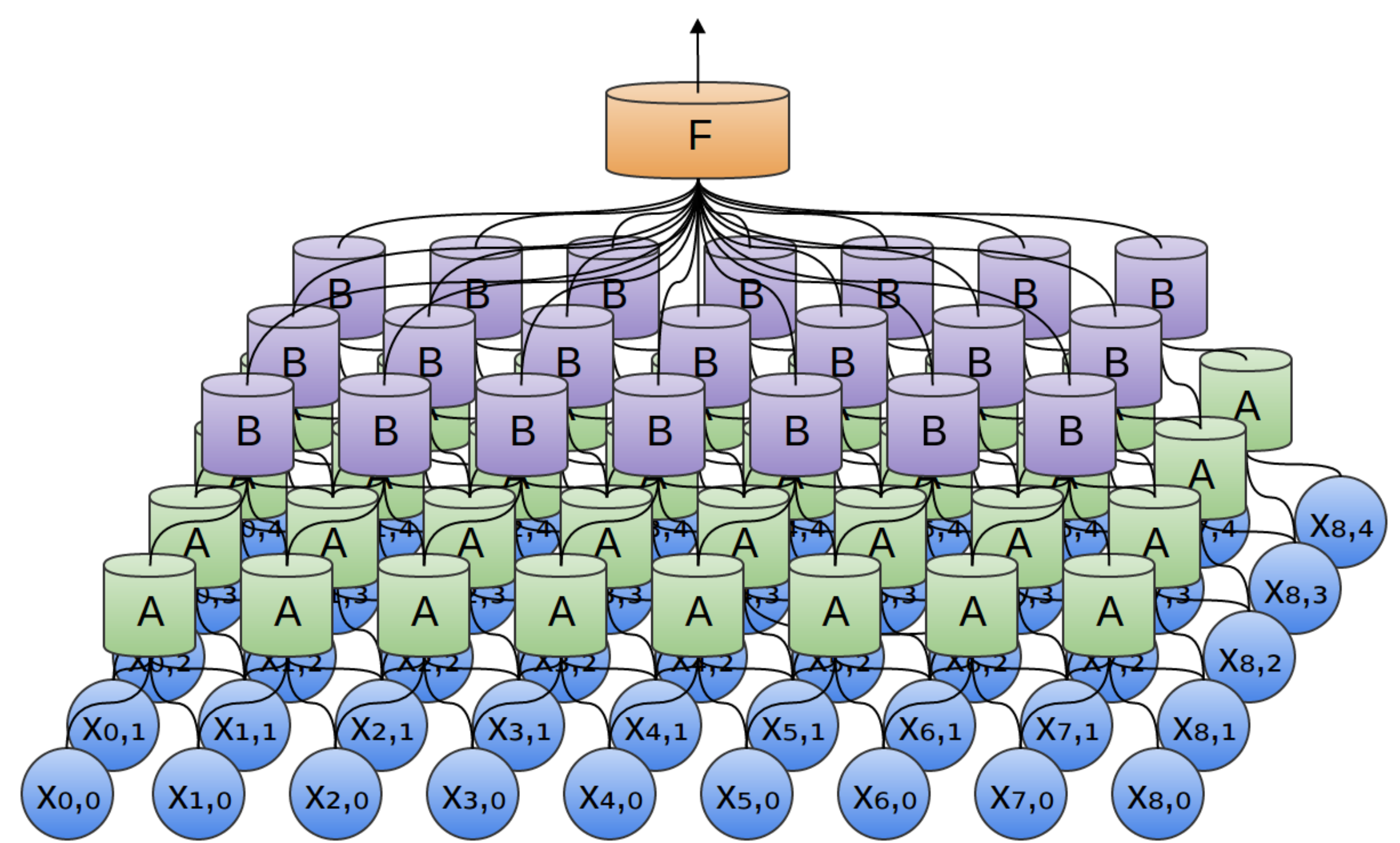
Images: Wikipedia

Simplest Convolution Net



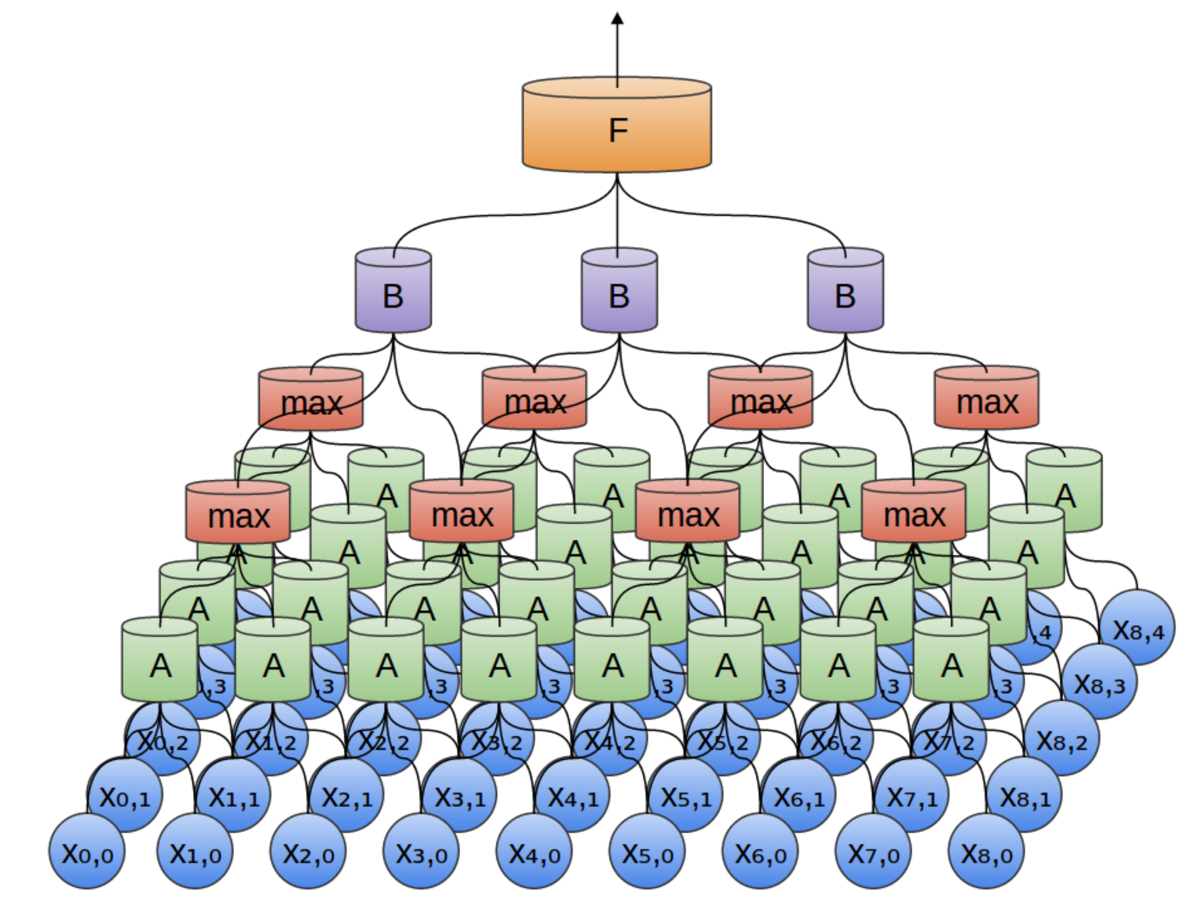
Courtesy: Chris Olah

Stacking Convolutions



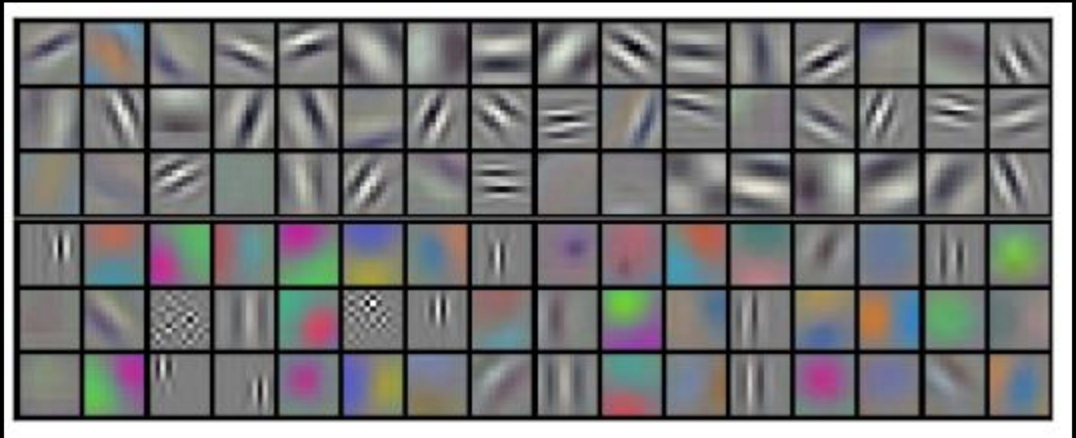
Courtesy: Chris Olah

Pooling



Courtesy: Chris Olah

Multiple Filters



These are the filters from one convolution on one layer (from Krizehvsky *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 $W_O \times H_O \times D_O$

$W_O = (W_I - N + 2P) / S+1$
$H_O = (H_I - F + 2P) / S+1$
$D_O = K$

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.

Convolutional MNIST

Complete Code

```
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})

print("test accuracy %g"%accuracy.eval(feed_dict={ x: mnist.test.images, y_: mnist.test.labels, keep_prob: 1.0}))
```

Convolutional MNIST

Loading 2D Images

```
>>> 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”

Convolutional MNIST

The First 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')
>>>
```

Convolutional MNIST

The First 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]))
```

We will have 32 5x5 filers in this layer

What values to initialize?

- Small random positive for weights
- Small constant for bias

Convolutional MNIST

The First 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)
```

TF will handle padding
- More explicit in cuDNN and Caffe

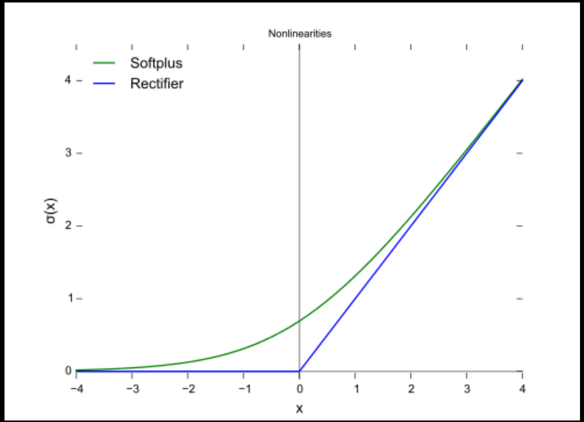
Stride of 1x1
- Must be same dims as X (just set depth/batch=1)

Convolutional MNIST

The First 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)
```

Add bias and apply our ReLU



Widely adopted around 2010!

Convolutional MNIST

The First 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')
```

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

The image we will pass to the next layer is now 14x14.

Convolutional MNIST

The First 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')
>>>
```

Convolutional MNIST

Second 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')
```

Now we have 32 features coming in, and we will use 64 on this layer.

The next layer will be getting a 7x7 image.

Convolutional MNIST

Fully Connected 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)
```

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

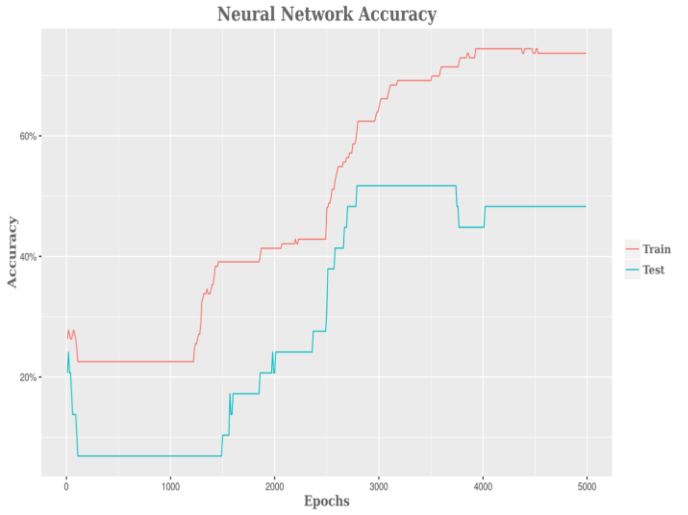
We will choose 1024 neurons for this layer.

Convolutional MNIST

Dropout

```
>>> 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
```

We will have a final FC layer that gets us from 1024 neurons down to our 10 possible outputs.



Convolutional MNIST

Last Steps Before Training

```
>>> 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))
```

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.

Convolutional MNIST

Training

```
>>> 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})
>>>
>>> print("test accuracy %g"%accuracy.eval(feed_dict={ x: mnist.test.images, y_: mnist.test.labels, keep_prob: 1.0}))
test accuracy 0.9915
```

Train away for 20,000 steps in batches of 50. Notice how we turn the dropout off when we periodically check our accuracy.

Convolutional MNIST

Testing

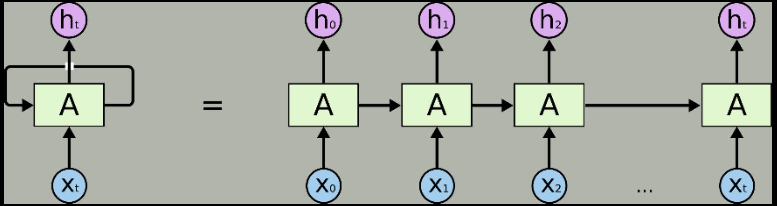
```
>>> from tensorflow.examples.tutorials.mnist import input_data
>>>
>>> import tensorflow as tf
>>>
>>> mnist = input_data.read_data_sets(".", one_hot=True)
>>>
>>> x = tf.placeholder(tf.float32, [None, 784])
>>> y_ = tf.placeholder(tf.float32, [None, 10])
>>>
>>> x_image = tf.reshape(x, [-1,28,28,1])
>>>
>>> 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})
>>>
>>> print("test accuracy %g"%accuracy.eval(feed_dict={ x: mnist.test.images, y_: mnist.test.labels, keep_prob: 1.0}))
test accuracy 0.9915
```

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!

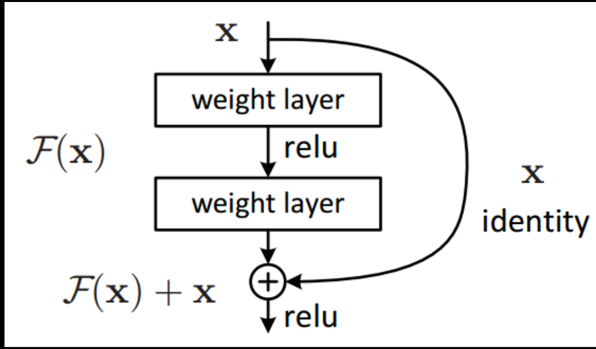
Other Significant Architectures

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



Courtesy: Chris Olah

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



Very Deep Neural Net
100s of layers, Pushing 1000

“Theoretician’s Nightmare”

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.

You now have a Toolbox

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 you 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.

Other Toolboxes

You have a plethora of alternatives available as well. You are now in a position to appreciate some comparisons.

Package	Applications	Language	Strengths
Tensorflow	Neural Nets	Python, C++	Very popular.
Caffe	Neural Nets	Python, C++	Many research projects and publications.
Spark MLLIB	Classification, Regression, Clustering, etc.	Python, Scala, Java, R	Very scalable. Widely used in serious applications.
Scikit-Learn	Classification, Regression, Clustering	Python	
cuDNN	Neural Nets	C++, GPU-based	Used in many other frameworks: TF, Caffe, etc.
Theano	Neural Nets	Python	Lower level numerical routines. NumPy-esque.
Torch	Neural Nets	Lua (PyTorch=Python)	Dynamic graphs (variable length input/output) good for RNN.
Keras	Neural Nets	Python (on top of TF, Theano)	Higher level approach.
Digits	Neural Nets	"Caffe", GPU-based	Used with other frameworks (only Caffe at moment).

Applications

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). https://www.kaggle.com/datasets
There are always multiple currently running competitions you can enter. *Competitions*: https://www.kaggle.com/competitions

Credits

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.