



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



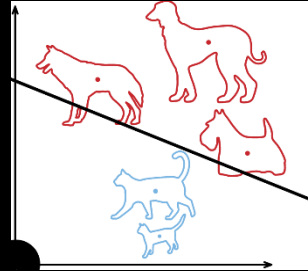
Two Perspectives

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

- Inspired by biological models.

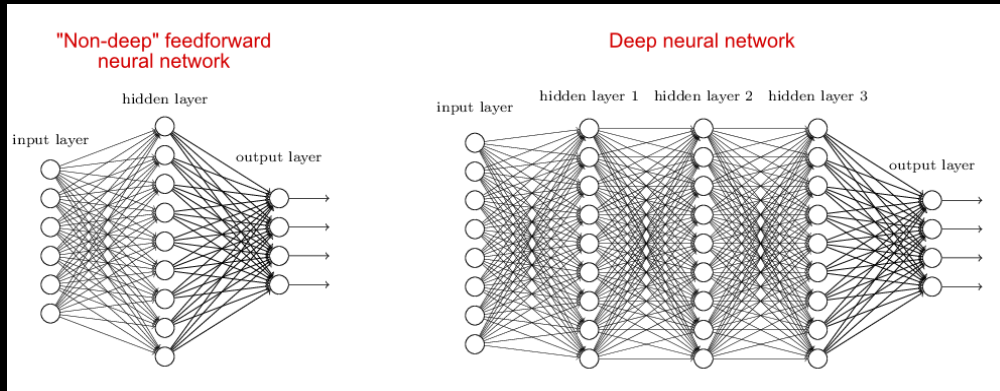


- An evolution of classic ML techniques (the perceptron).



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.

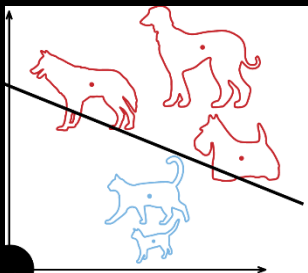
Modeled After The Brain



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

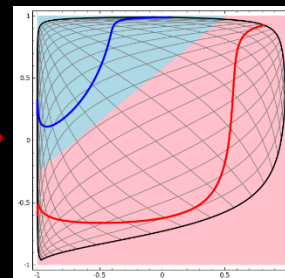
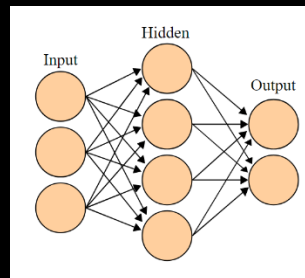
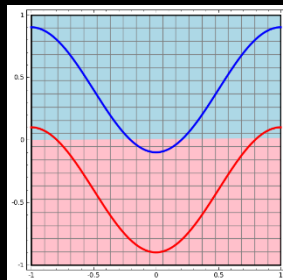
As a Highly Dimensional Non-linear Classifier

Perceptron



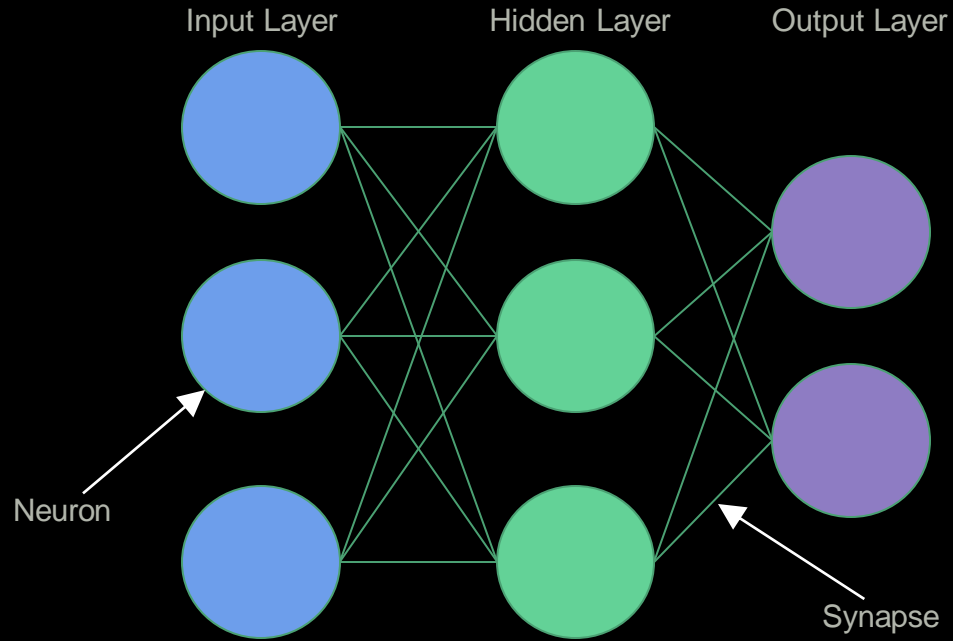
No Hidden Layer
Linear

Network



Hidden Layers
Nonlinear

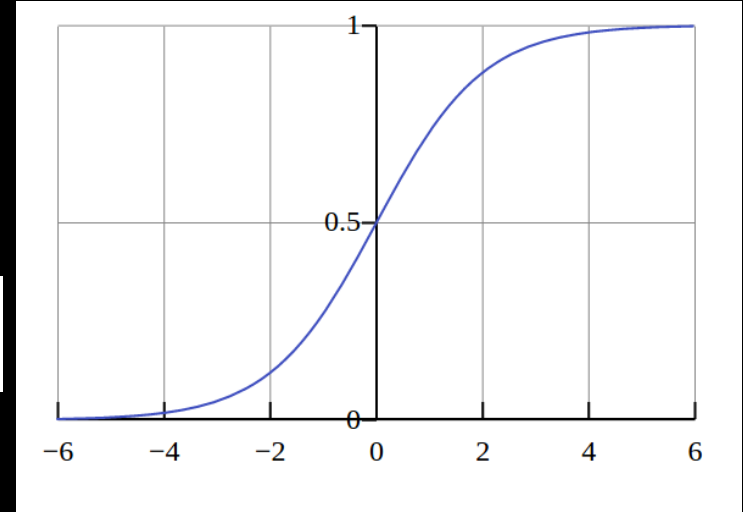
Basic NN Architecture



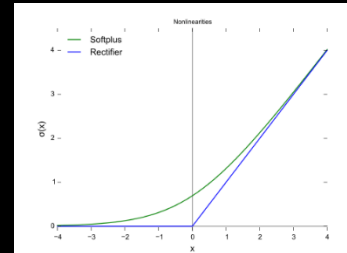
Activation Function

- 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.

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

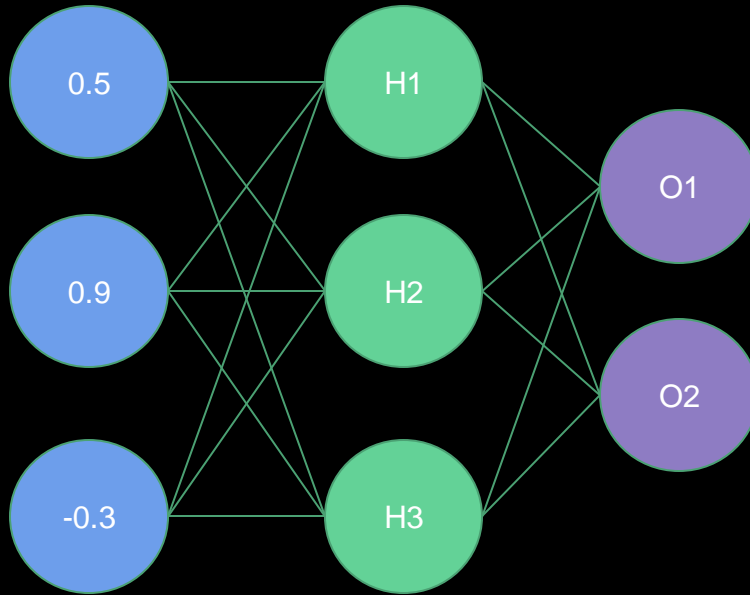


- The Rectified Linear activation function is zero when the input is negative and is equal to the input when the input is positive.
- Rectified Linear activation functions have become more popular because they are faster to compute than the sigmoid or hyperbolic tangent.
- We will use these later.



Inference

Using a NN



H1 Weights = (1.0, -2.0, 2.0)

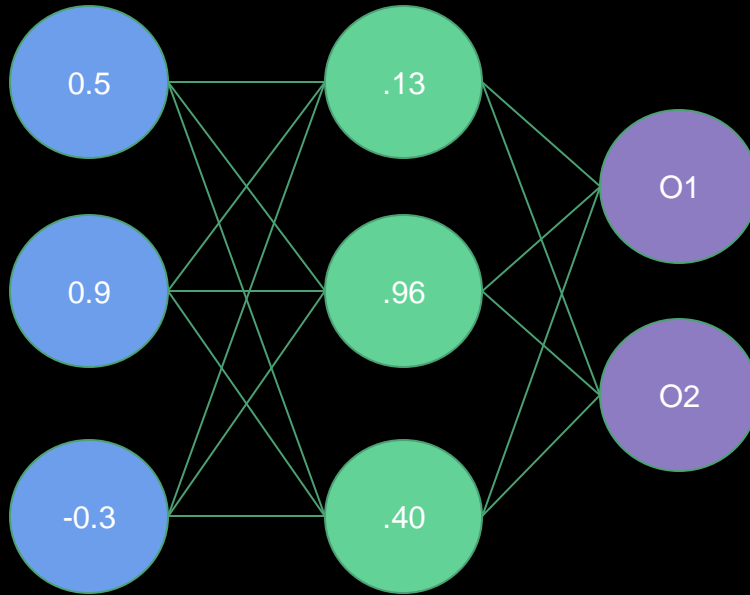
H2 Weights = (2.0, 1.0, -4.0)

H3 Weights = (1.0, -1.0, 0.0)

O1 Weights = (-3.0, 1.0, -3.0)

O2 Weights = (0.0, 1.0, 2.0)

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)

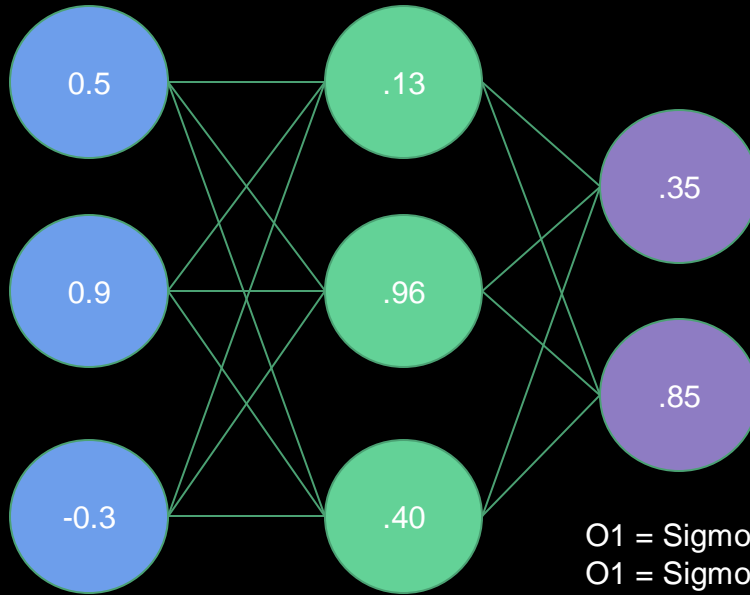
O2 Weights = (0.0, 1.0, 2.0)

$$H1 = \text{Sigmoid}(0.5 * 1.0 + 0.9 * -2.0 + -0.3 * 2.0) = \text{Sigmoid}(-1.9) = .13$$

$$H2 = \text{Sigmoid}(0.5 * 2.0 + 0.9 * 1.0 + -0.3 * -4.0) = \text{Sigmoid}(3.1) = .96$$

$$H3 = \text{Sigmoid}(0.5 * 1.0 + 0.9 * -1.0 + -0.3 * 0.0) = \text{Sigmoid}(-0.4) = .40$$

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)

$$O1 = \text{Sigmoid}(.13 * -3.0 + .96 * 1.0 + .40 * -3.0) = \text{Sigmoid}(-.63) = .35$$

$$O2 = \text{Sigmoid}(.13 * 0.0 + .96 * 1.0 + .40 * 2.0) = \text{Sigmoid}(1.76) = .85$$

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)

$$\text{Sig}\left(\begin{array}{|c|c|c|} \hline & \text{Hidden Layer Weights} & \\ \hline 1.0 & -2.0 & 2.0 \\ \hline 2.0 & 1.0 & -4.0 \\ \hline 1.0 & -1.0 & 0.0 \\ \hline \end{array} * \begin{array}{|c|} \hline \text{Inputs} \\ \hline 0.5 \\ \hline 0.9 \\ \hline -0.3 \\ \hline \end{array} \right) = \text{Sig}\left(\begin{array}{|c|c|c|} \hline -1.9 & 3.1 & -0.4 \\ \hline \end{array} \right) = \begin{array}{|c|c|c|} \hline \text{Hidden Layer Outputs} \\ \hline .13 & .96 & 0.4 \\ \hline \end{array}$$

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

Biases

It is also very useful to be able to offset our inputs by some constant. You can think of this as centering the activation function, or translating the solution (next slide). We will call this constant the *bias*, and it there will often be one value per layer.

Our math for the previously calculated layer now looks like this with $b=0.1$:

$$\text{Sig}\left(\begin{array}{|c|c|c|} \hline \text{Hidden Layer Weights} & & \\ \hline 1.0 & -2.0 & 2.0 \\ \hline 2.0 & 1.0 & -4.0 \\ \hline 1.0 & -1.0 & 0.0 \\ \hline \end{array} * \begin{array}{|c|} \hline \text{Inputs} \\ \hline 0.5 \\ \hline 0.9 \\ \hline -0.3 \\ \hline \end{array} + \begin{array}{|c|} \hline 0.1 \\ \hline 0.1 \\ \hline 0.1 \\ \hline \end{array} \right) = \text{Sig}\left(\begin{array}{|c|c|c|} \hline -1.8 & 3.2 & -0.3 \\ \hline \end{array} \right) = \begin{array}{|c|c|c|} \hline \text{Hidden Layer Outputs} \\ \hline .14 & .96 & 0.4 \\ \hline \end{array}$$

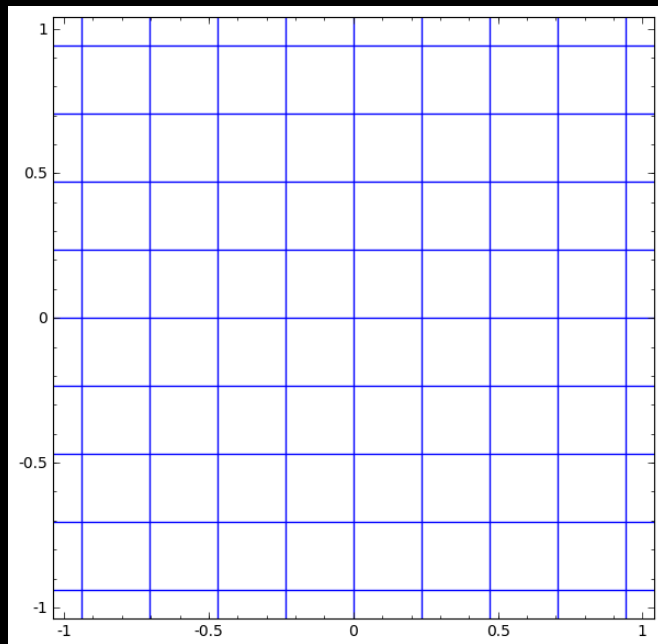
Linear + Nonlinear

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.

The matrix multiply provides the *skew* and *scale*.

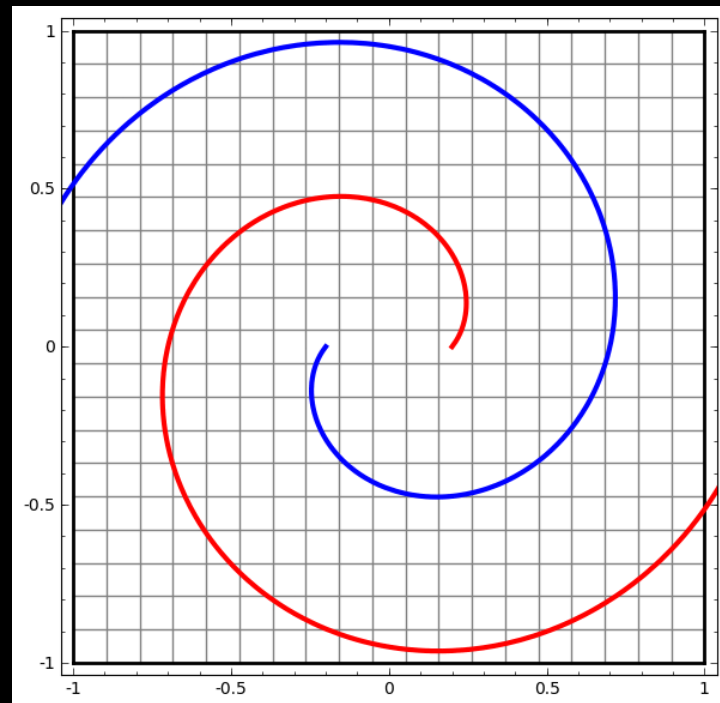
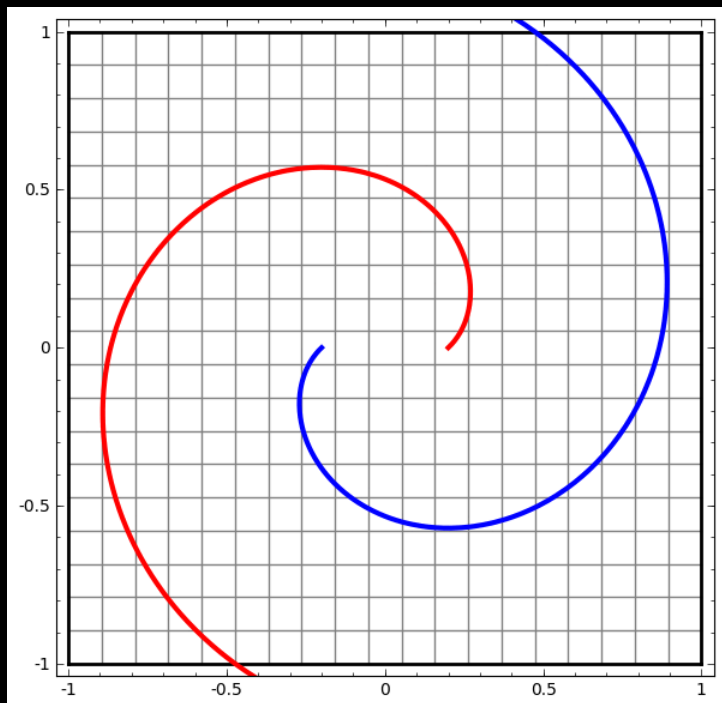
The bias provides the *translation*.

The activation function provides the *warp*.



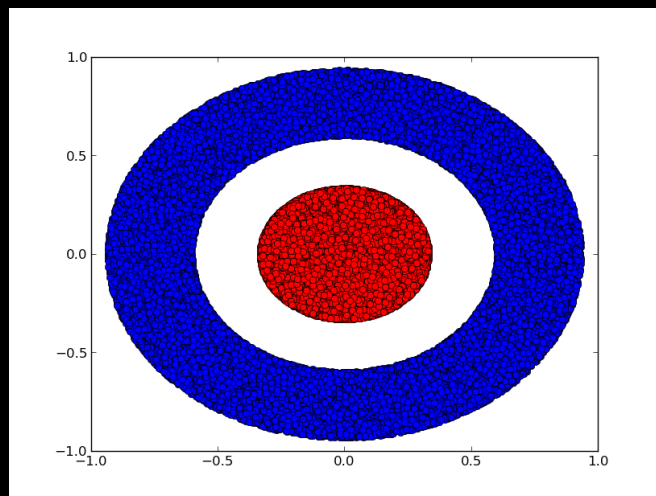
Linear + Nonlinear

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



Width of Network

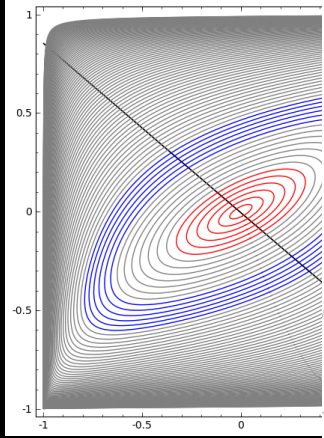
A very underappreciated fact about networks is that the width of any layer determines how many dimensions it can work in. This is valuable even for lower dimension problems. How about trying to classify (separate) this dataset:



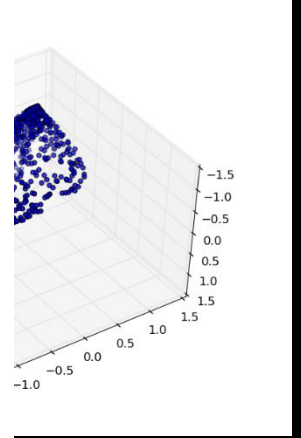
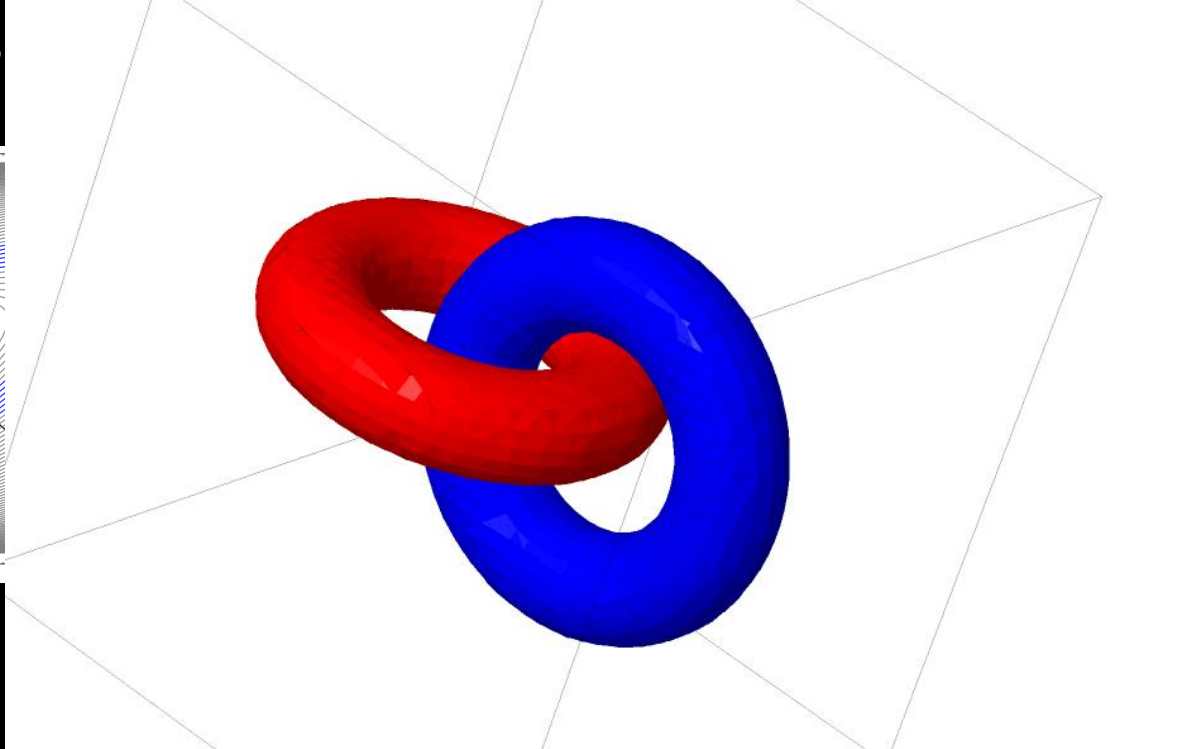
Can a neural net do this with twisting and deforming? What good does it do to have more than two dimensions with a 2D dataset?

Working In Higher Dimensions

It takes at least 3



Trying



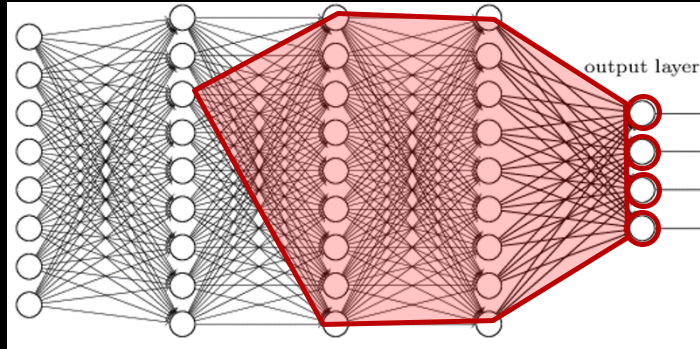
s in 3D

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

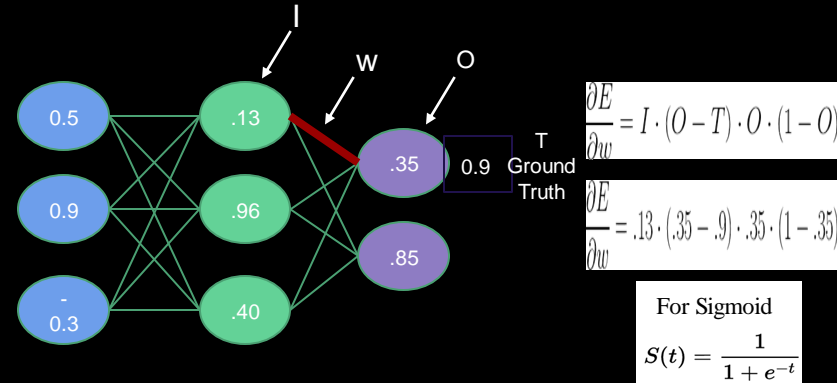
Training Neural Networks

So how do we find these magic weights? We want to minimize the error on our training data. Given labeled inputs, select weights that generate the smallest average error on the outputs.

We know that the output is a function of the weights: $E(w_1, w_2, w_3, \dots, i_1, \dots, t_1, \dots)$. So to figure out which way, and how much, to push any particular weight, say w_3 , we want to calculate $\frac{\partial E}{\partial w_3}$



There are a lot of dependencies going on here. It isn't obvious that there is a viable way to do this in very large networks.



If we take one small piece, it doesn't look so bad.

Backpropagation

If we use the chain rule repeatedly across layers we can work our way backwards from the output error through the weights, adjusting them as we go. Note that this is where the requirement that activation functions must have nicely behaved derivatives comes from.

This technique makes the weight inter-dependencies much more tractable. An elegant perspective on this can be found from Chris Olah at

<http://colah.github.io/posts/2015-08-Backprop> .

With basic calculus you can readily work through the details. You can find an excellent explanation from the renowned *3Blue1Brown* at

<https://www.youtube.com/watch?v=Ilg3gGewQ5U> .

You don't need to know the details, and this is all we have time to say, but you certainly can understand this fully if your freshman calculus isn't too rusty and you have some spare time.

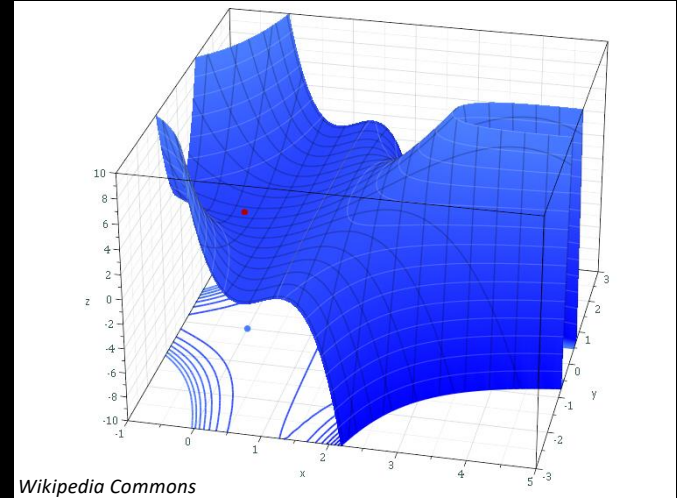
Solvers

However, even this efficient process leaves us with potentially many millions of simultaneous equations to solve (real nets have a lot of weights). They are non-linear to boot. Fortunately, this isn't a new problem created by deep learning, so we have options from the world of numerical methods.

The standard has been *gradient descent*. Methods, often similar, have arisen that perform better for deep learning applications. TensorFlow will allow us to use these interchangeably - and we will.

Most interesting recent methods incorporate *momentum* to help get over a local minimum. Momentum and *step size* are the two *hyperparameters* we will encounter later.

Nevertheless, we don't expect to ever find the actual global minimum.



We could/should find the error for all the training data before updating the weights (an *epoch*). However it is usually much more efficient to use a *stochastic* approach, sampling a random subset of the data, updating the weights, and then repeating with another *mini-batch*.

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.

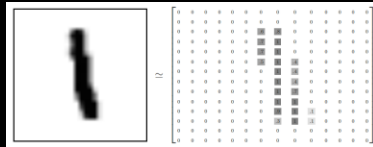
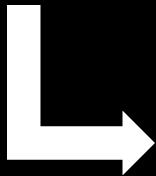
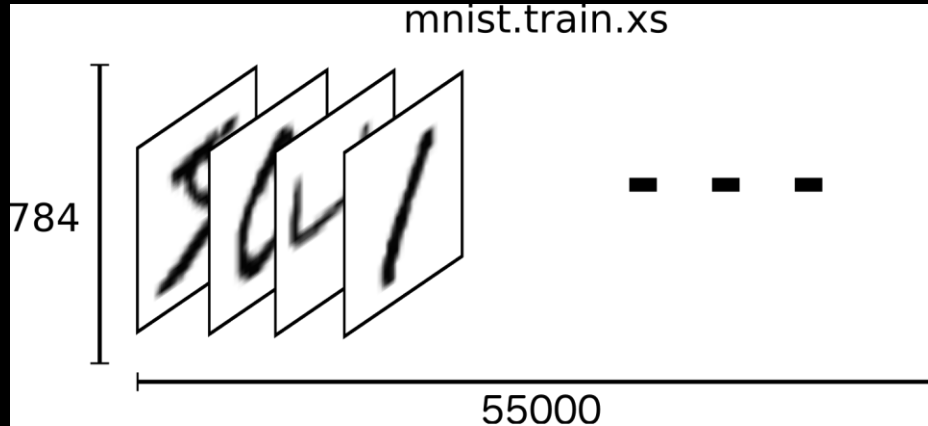
Our problem will be character recognition. We will learn to read handwritten digits by training on a large set of 28x28 greyscale samples.



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

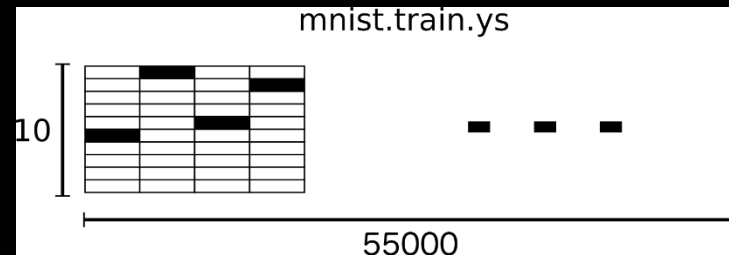
MNIST Data

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
```

To start TensorFlow:

```
gpu42% module load tensorflow/1.5_gpu  
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.....
```

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.          ,  0.          ],
       [ 0.          ,  0.          ,  0.          ,  0.          ,  0.          ,
         0.          ,  0.          ,  0.          ,  0.          ,  0.          ],
       [ 0.99607849,  1.          ,  0.93725497,  0.1137255 ,  0.          ,
         0.          ,  0.          ,  0.          ,  0.          ,  0.          ],
       [ 0.          ,  0.          ,  0.          ,  0.          ,  0.          ,
         0.          ,  0.          ,  0.          ,  0.          ,  0.          ]],
      dtype=float32)
```

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, out_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 vertical 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.

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_v2(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(50)
>>>     sess.run(train_step, feed_dict={x: batch_xs, y_: batch_ys})
>>>
>>> correct_prediction = tf.nn.argmax(y, axis=1) == tf.nn.argmax(y_, axis=1)
>>> accuracy = tf.nn.mean(correct_prediction)
>>> print(sess.run(accuracy, feed_dict={x: batch_xs, y_: batch_ys}))
```

GD Solver

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:

| Digit | Output | Exponential | Normalized |
|-------|--------|-------------|------------|
| 0 | 4.8 | 121 | .87 |
| 1 | -2.6 | 0.07 | .00 |
| 2 | 2.9 | 18 | .13 |

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

```
>>> 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_v2(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.

Testing 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_v2(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. *This is a linear matrix that knows how to read numbers by multiplying an image vector!* Or not. Consider this the most basic walkthrough of constructing a graph with TensorFlow.

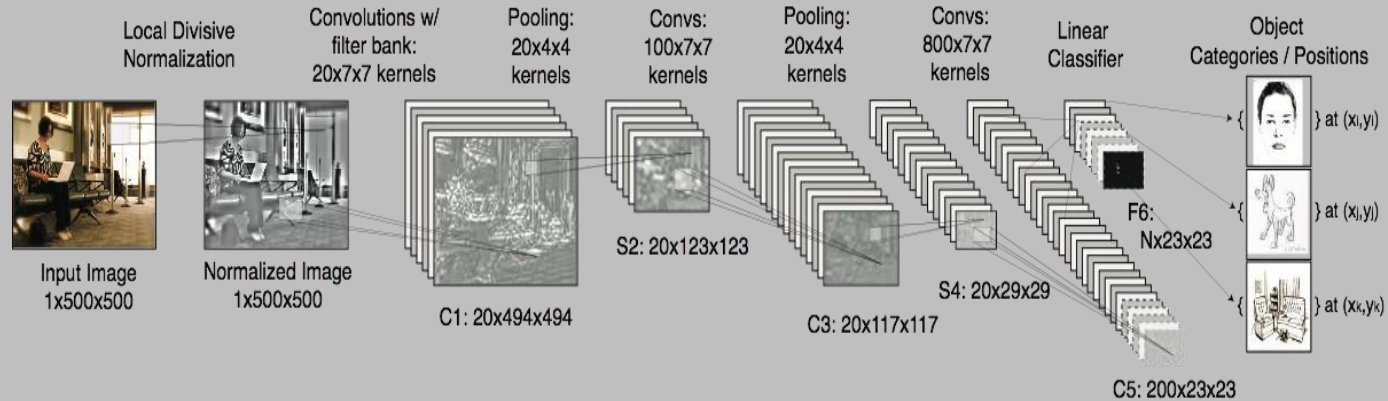
We can do much better using a real NN. We will even jump quite close to the state-of-the-art and use a Convolutional Neural Net.

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

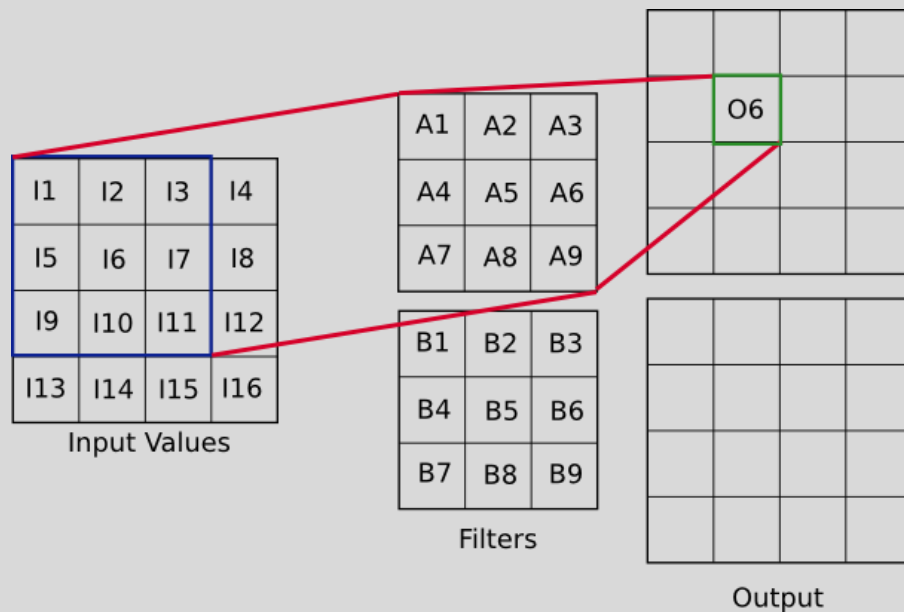
It will also take advantage of the actual 2D structure of the image that we ditched so cavalierly earlier.

It will include dropout! A surprising optimization to many.

Convolutional Net



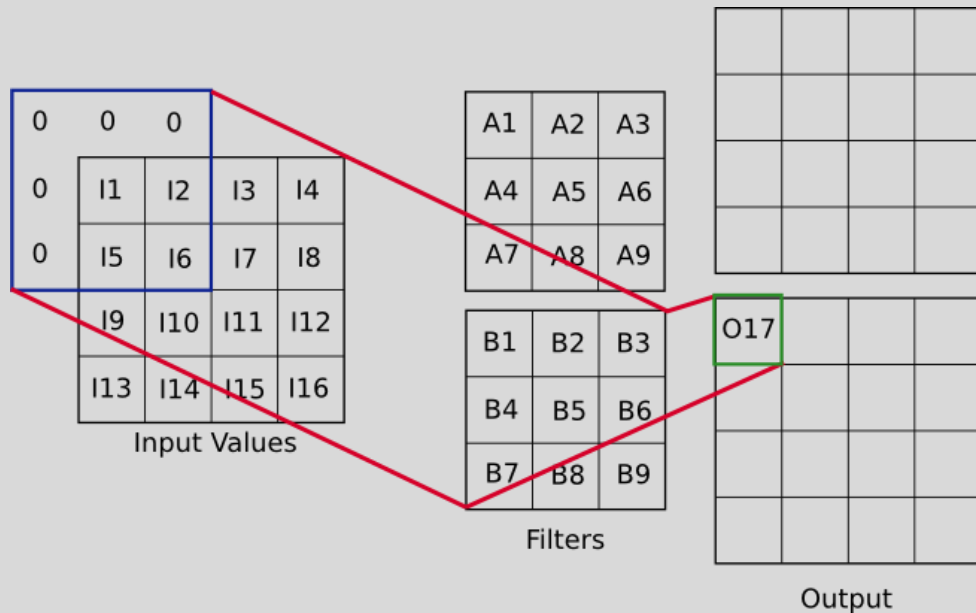
Convolution



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

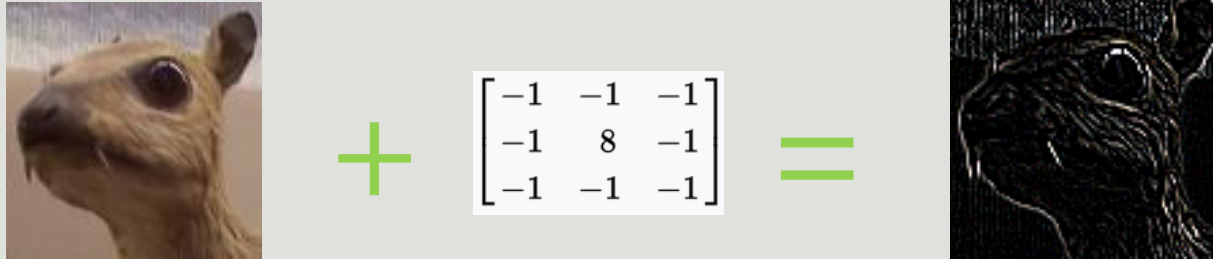
Convolution

Boundary and Index Accounting!



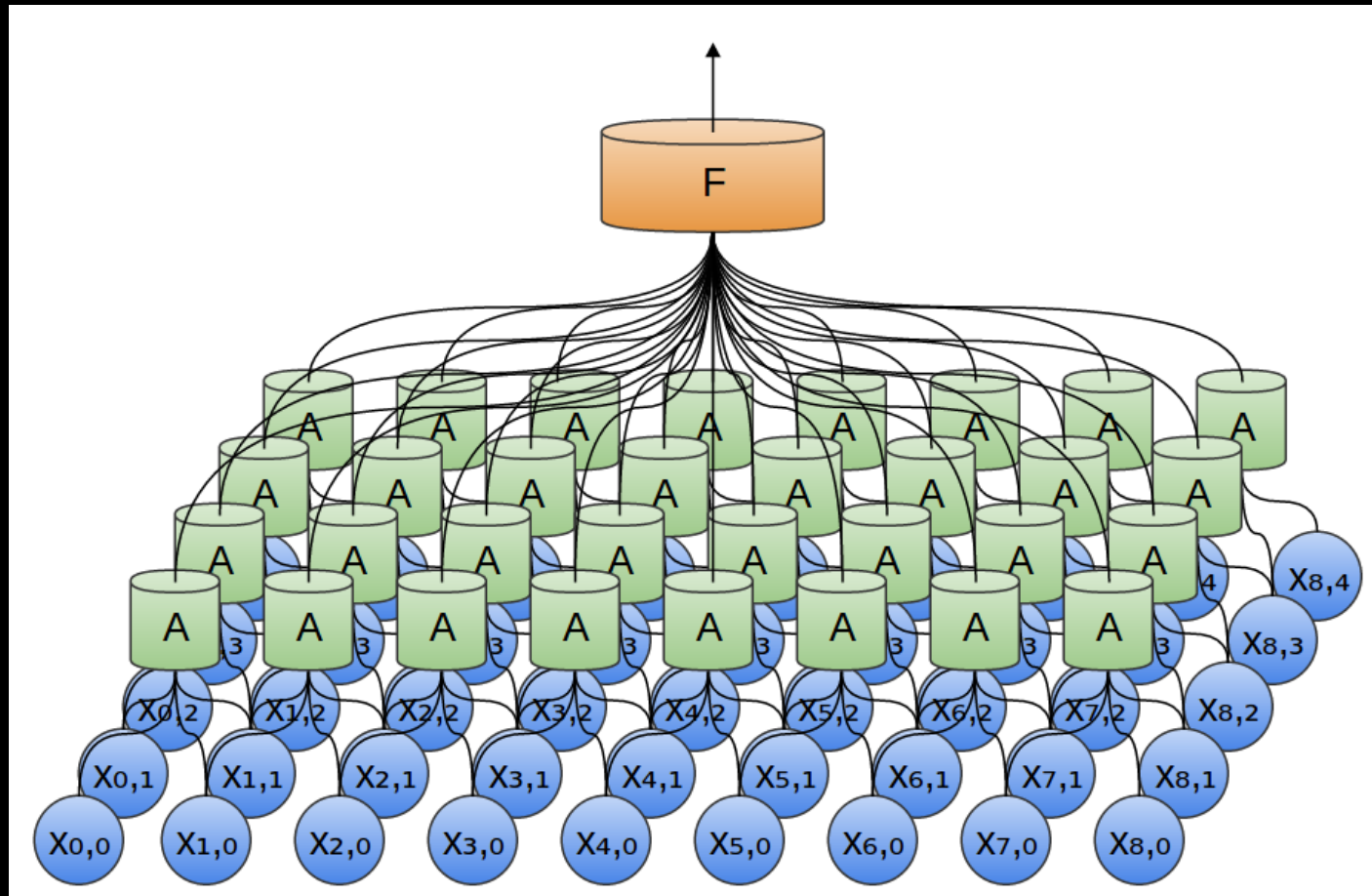
$$O_{17} = B_5 \cdot I_1 + B_6 \cdot I_2 + B_8 \cdot I_5 + B_9 \cdot I_6$$

Straight Convolution

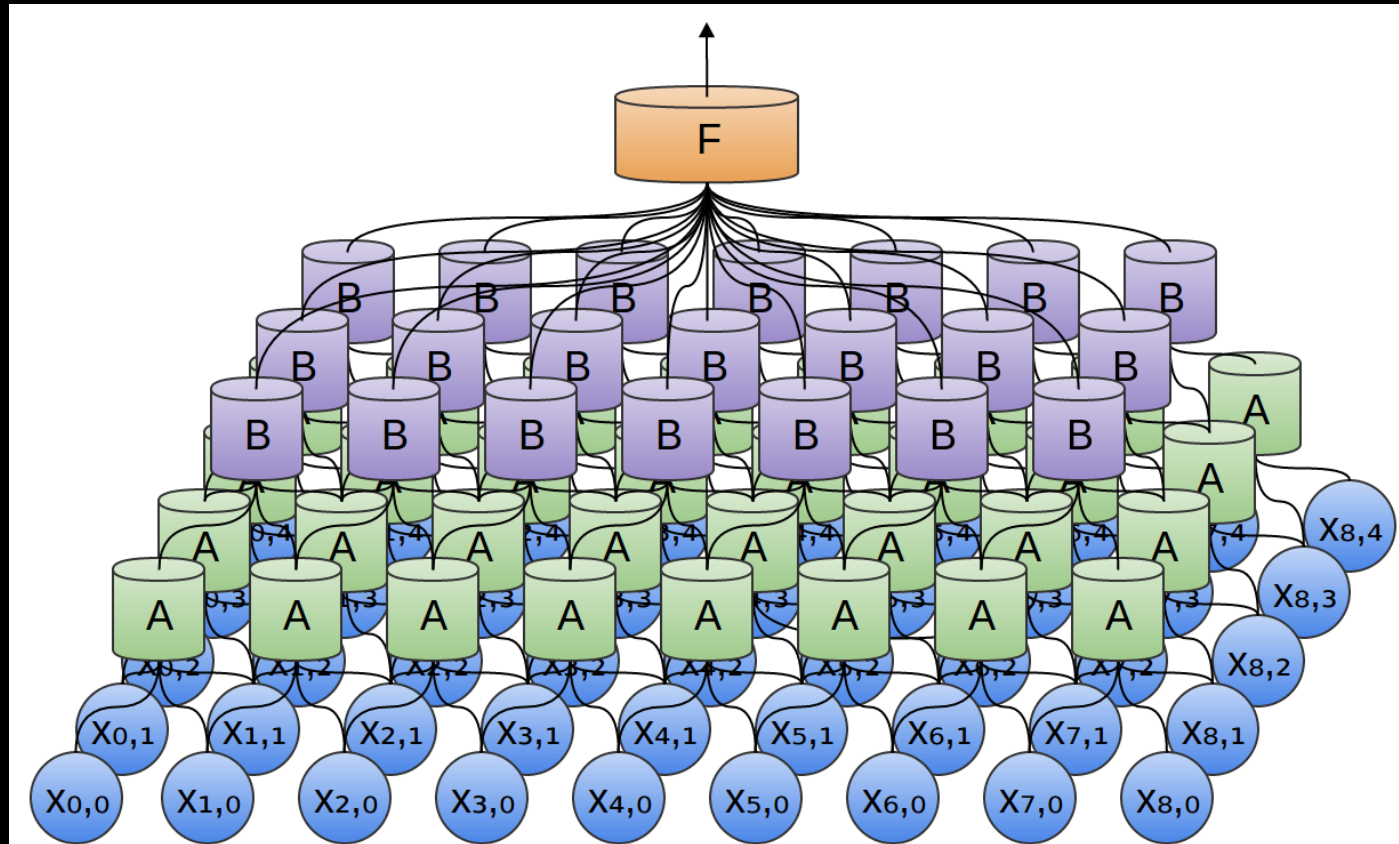

$$\begin{bmatrix} -1 & -1 & -1 \\ -1 & 8 & -1 \\ -1 & -1 & -1 \end{bmatrix}$$

Edge Detector

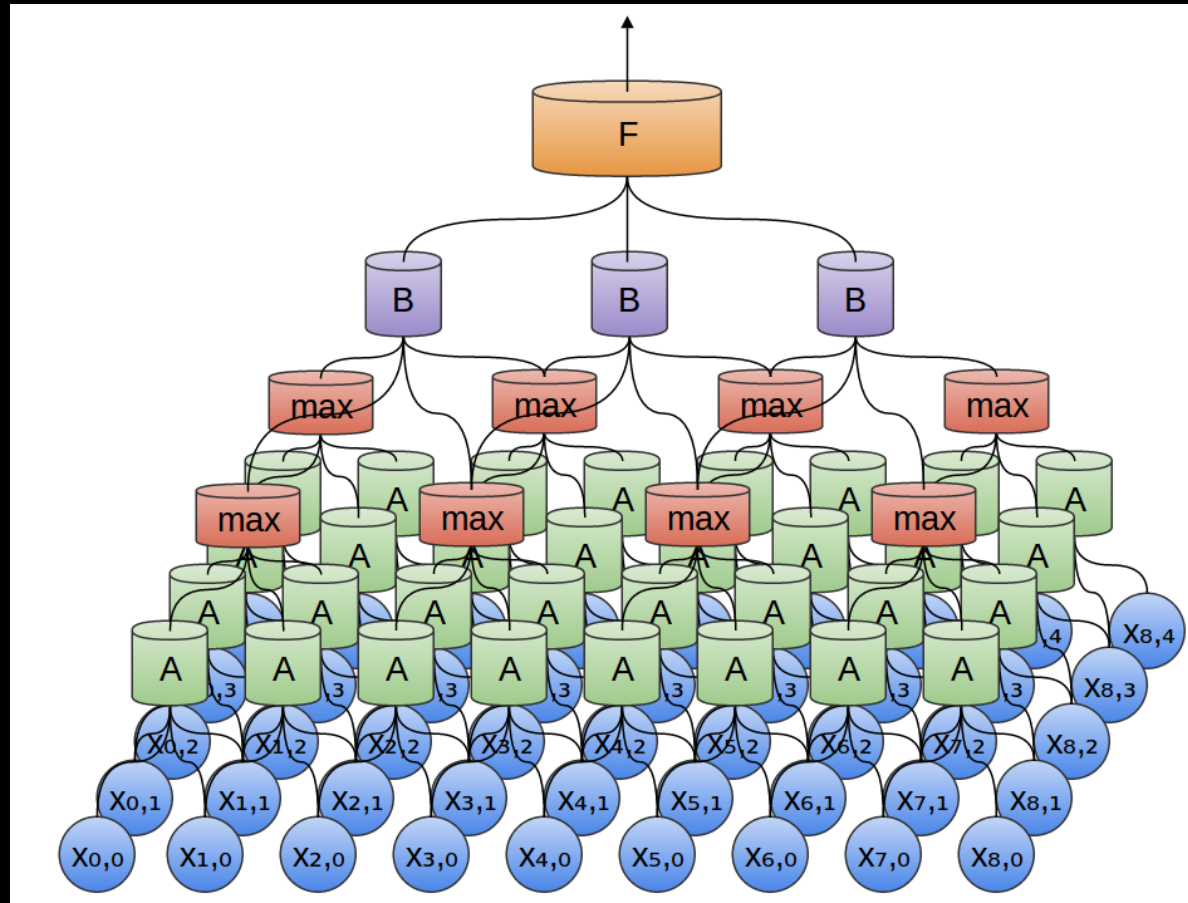
Simplest Convolution Net



Stacking Convolutions

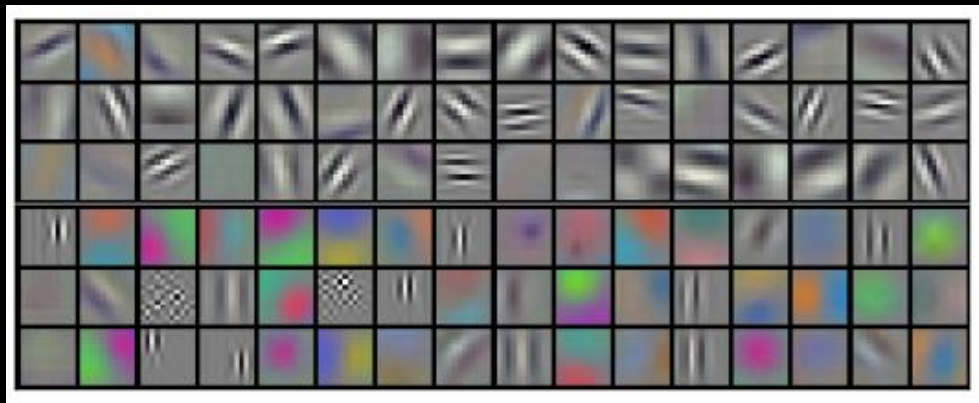


Pooling

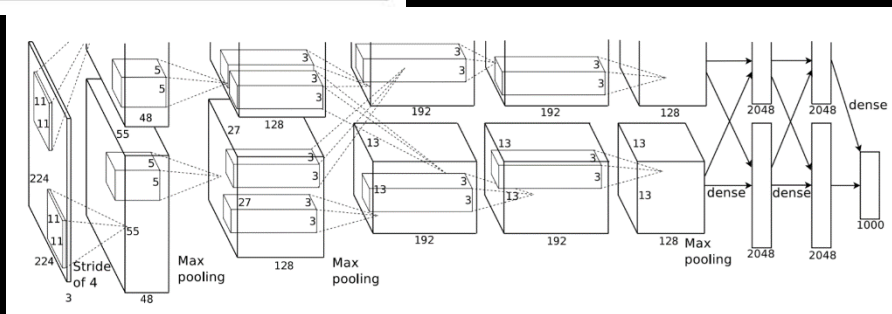


A Sophisticated Example

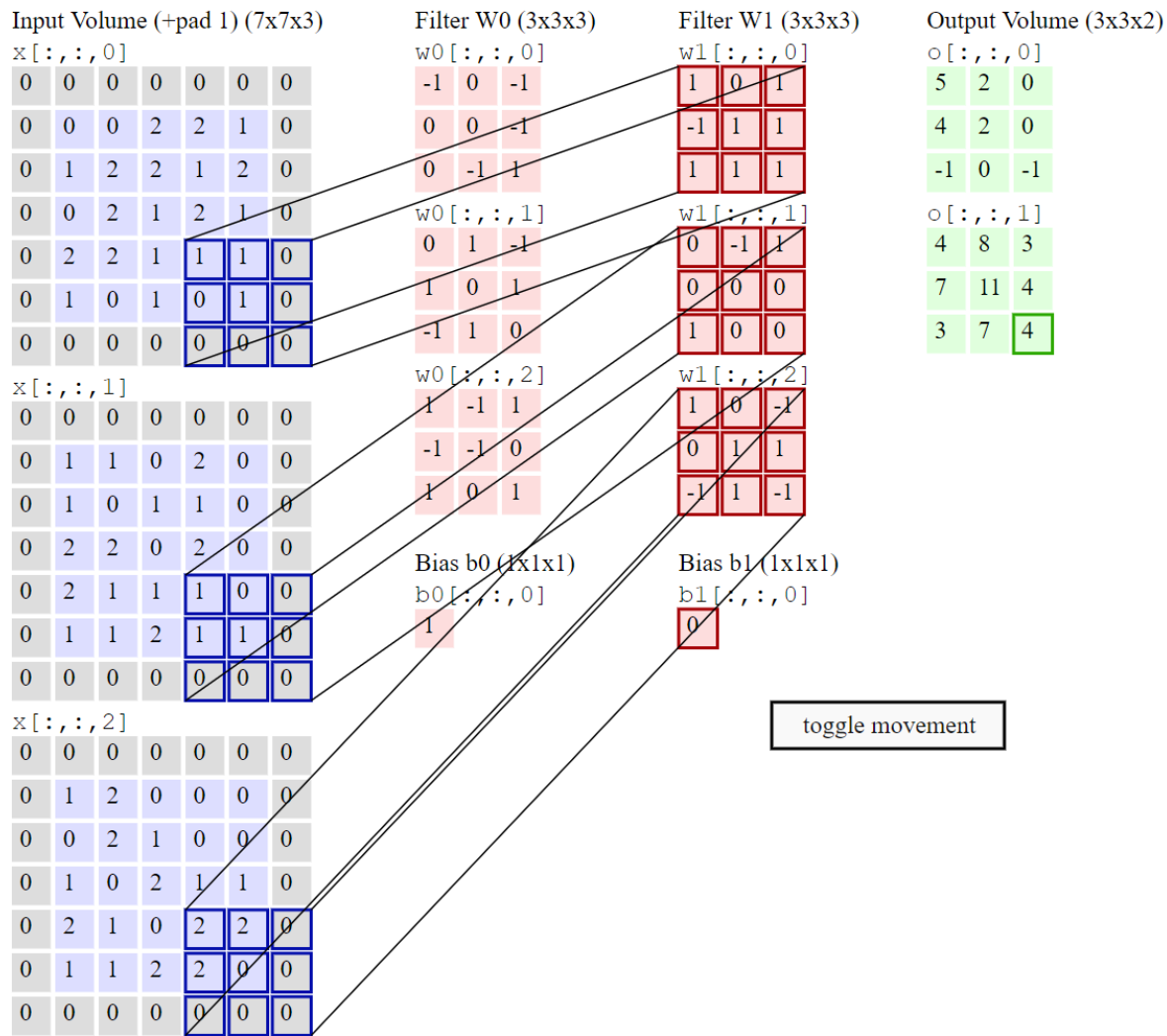
These are the 96 first layer 11x11 (x3, RGB, stacked here) filters from Krizhevsky *et al.* (2012), a landmark advance in ImageNet classification.



Among the several novel techniques combined in this work (such as early use of ReLU), they used dual GPUs, with different flows for each, communicating only at certain layers. A result is that the bottom GPU consistently specialized on color information, and the top did not.



Convolution



From the very nice
Stanford CS231n
course at
<http://cs231n.github.io/convolutional-networks/>

Stride = 2

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_v2(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 filters 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

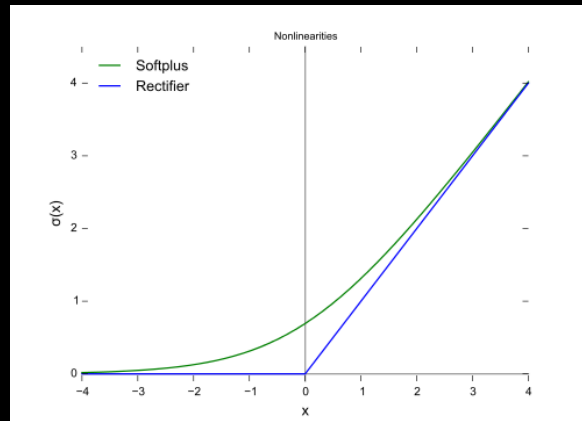
Must be same dims as X (just set depth,batch=1)

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.

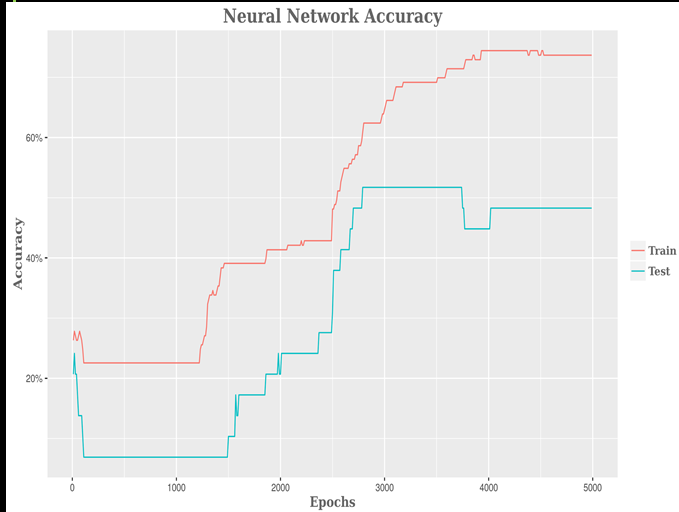
We will choose 1024 neurons for this layer.

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_v2(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_v2(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_v2(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!

Real Time Demo

This *amazing, stunning, beautiful* demo from Adam Harley (now just across campus) is very similar to what we just did, but different enough to be interesting.

<http://scs.ryerson.ca/~aharley/vis/conv/flat.html>

It is worth experiment with. Note that this is an excellent demonstration of how efficient the forward network is. You are getting very real-time analysis from a lightweight web program. Training it took some time.

Draw your number here



Downsampled drawing: 2

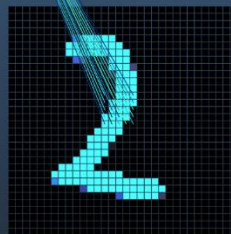
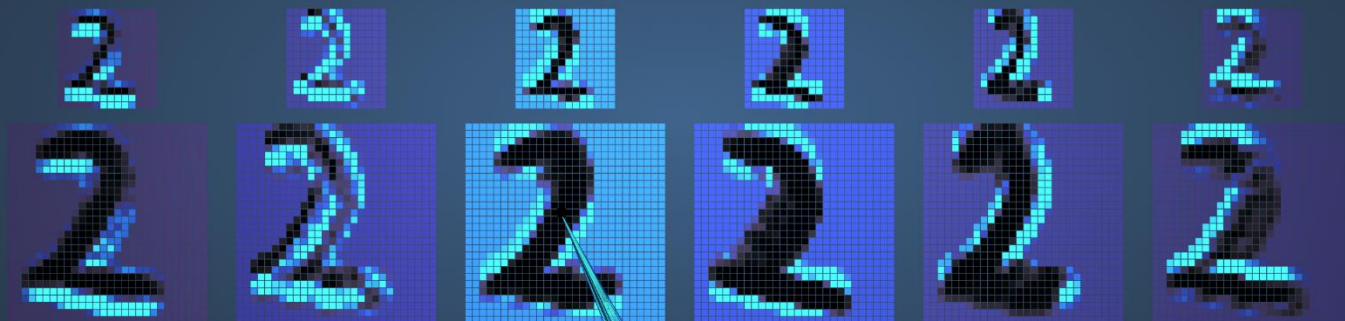
First guess: 2

Second guess: 0

Layer visibility

| | |
|-------------------------|------|
| Input layer | Show |
| Convolution layer 1 | Show |
| Downsampling layer 1 | Show |
| Convolution layer 2 | Show |
| Downsampling layer 2 | Show |
| Fully-connected layer 1 | Show |
| Fully-connected layer 2 | Show |
| Output layer | Show |

0123456789



Style vs. Content

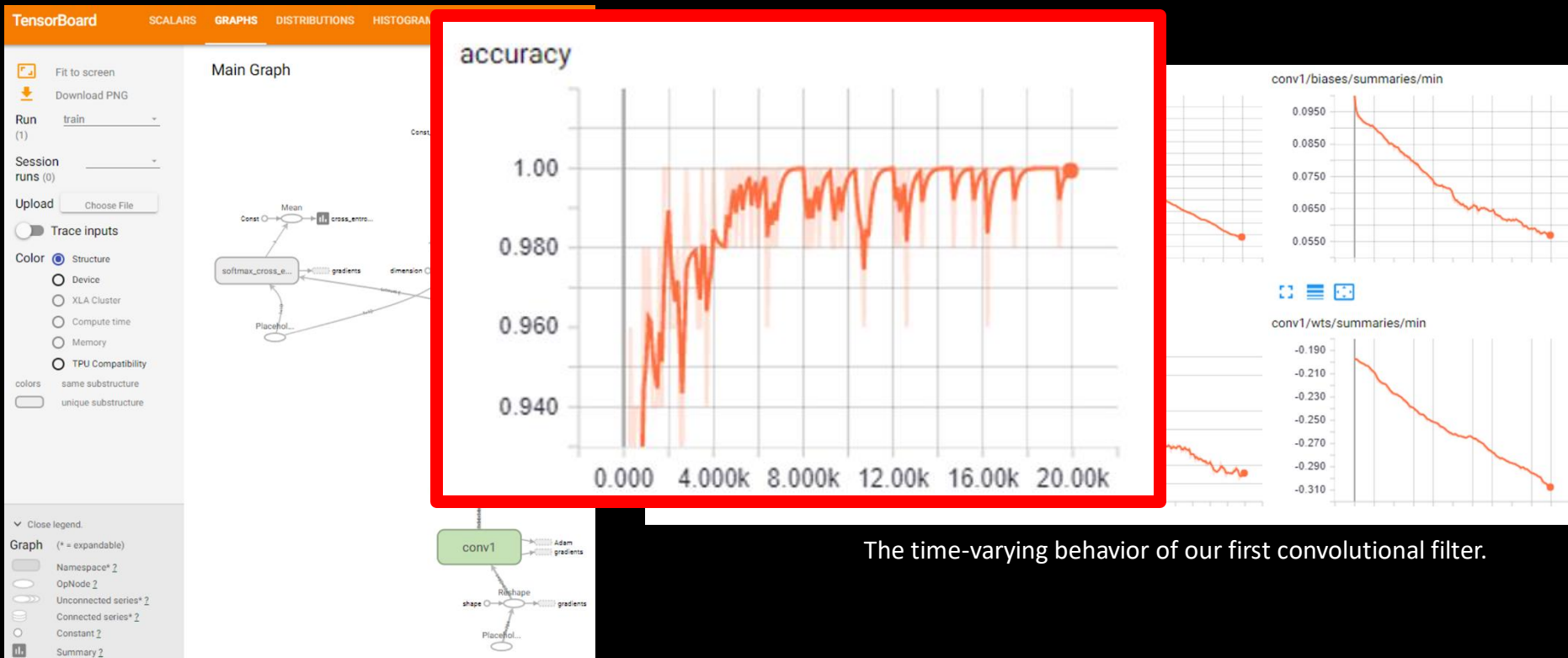
Deep Dream Generator



<https://deepdreamgenerator.com/feed>

TensorBoard

There is a tool that allows us visualize our graph and data more easily. It does require a *surprising* amount of instrumentation, but you may find it worthwhile.



The time-varying behavior of our first convolutional filter.

Our CNN graph.

Other Significant Architectures

Very Deep Neural Net

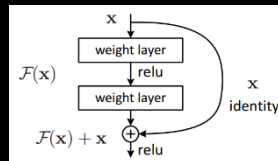
100s of layers, Pushing 1000

Residual Neural Net

Helps preserve reasonable gradients for very deep networks

Very effective at imagery

Used by AlphaGo Zero (40 residual CNN layers) in place of previous complex dual network



Recurrent Neural Net

Cycles back previous inputs (feedback)

Like short term memory

Adds context: language processing, time dependence for video, ...

Current advancement is Long Short Term Memory (LSTM)

Bit more complex

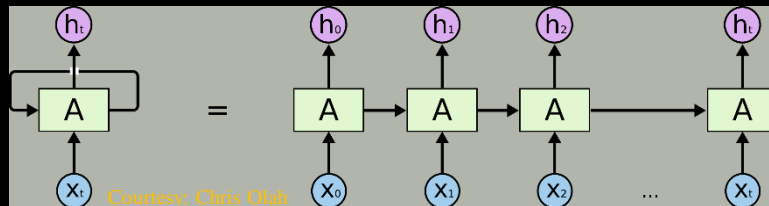
Very effective for certain tasks

This is where the action is now (winning all the comps!)

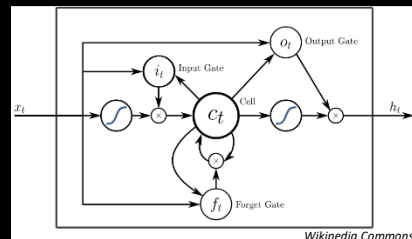
Often uses CNN as foundation

TensorFlow friendly

This is where Day 3 of this workshop would be spent!



RNN



LSTM

Learning Approaches

Supervised Learning

How you learned colors.

What we have been doing just now.

Used for: image recognition, tumor identification, segmentation.

Requires labeled data.

Lots of it. Augmenting helps.

Reinforcement Learning

How you learned to walk.

Requires goals (maybe long term, i.e. arbitrary delays between action and reward).

Used for: Go (AlphaGo Zero), robot motion, video games.

Unsupervised Learning

(Maybe) how you learned to see.

What we did earlier with clustering and our recommender.

Find patterns in data, compress data into model, find reducible representation of data.

Used for: Learning from unlabeled data.

All of these have been done with and without deep learning. DL has moved to the forefront of all of these.

“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. 2.0 more TF-like. |
| 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). |

Keras

Highest Level Approach

```
from __future__ import print_function
import keras
from keras.datasets import mnist
from keras.models import Sequential
from keras.layers import Dense, Dropout, Flatten
from keras.layers import Conv2D, MaxPooling2D
from keras import backend as K

batch_size = 128
num_classes = 10
epochs = 12

# input image dimensions
img_rows, img_cols = 28, 28

# the data, split between train and test sets
(x_train, y_train), (x_test, y_test) = mnist.load_data()

if K.image_data_format() == 'channels_first':
    x_train = x_train.reshape(x_train.shape[0], 1, img_rows, img_cols)
    x_test = x_test.reshape(x_test.shape[0], 1, img_rows, img_cols)
    input_shape = (1, img_rows, img_cols)
else:
    x_train = x_train.reshape(x_train.shape[0], img_rows, img_cols, 1)
    x_test = x_test.reshape(x_test.shape[0], img_rows, img_cols, 1)
    input_shape = (img_rows, img_cols, 1)

x_train = x_train.astype('float32')
x_test = x_test.astype('float32')
x_train /= 255
x_test /= 255
print('x_train shape:', x_train.shape)
print(x_train.shape[0], 'train samples')
print(x_test.shape[0], 'test samples')

# convert class vectors to binary class matrices
y_train = keras.utils.to_categorical(y_train, num_classes)
y_test = keras.utils.to_categorical(y_test, num_classes)
```

```
model = Sequential()
model.add(Conv2D(32, kernel_size=(3, 3),
                 activation='relu',
                 input_shape=input_shape))
model.add(Conv2D(64, (3, 3), activation='relu'))
model.add(MaxPooling2D(pool_size=(2, 2)))
model.add(Dropout(0.25))
model.add(Flatten())
model.add(Dense(128, activation='relu'))
model.add(Dropout(0.5))
model.add(Dense(num_classes, activation='softmax'))

model.compile(loss=keras.losses.categorical_crossentropy,
              optimizer=keras.optimizers.Adadelta(),
              metrics=['accuracy'])

model.fit(x_train, y_train,
          batch_size=batch_size,
          epochs=epochs,
          verbose=1,
          validation_data=(x_test, y_test))
score = model.evaluate(x_test, y_test, verbose=0)
print('Test loss:', score[0])
print('Test accuracy:', score[1])
```

Slightly smaller than our network, but same idea.

From https://github.com/keras-team/keras/blob/master/examples/mnist_cnn.py

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 color 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>

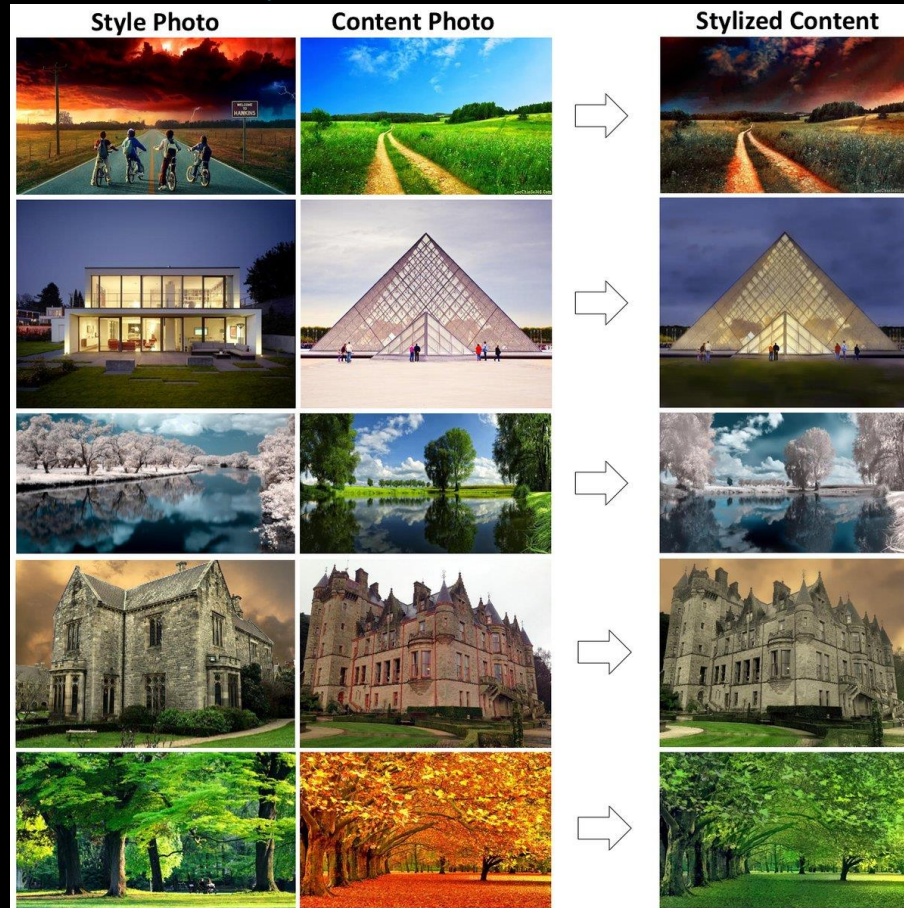
Demos

Ray-traced videogames soon? Recurrent CNN.

http://research.nvidia.com/sites/default/files/publications/dnn_denoise_author.pdf

Demos

Style vs. Content: A little more subtle



Grab it at <https://github.com/NVIDIA/FastPhotoStyle>

Demos & Discussion

A wise man once (not that long ago) told me "John, I don't need a neural net to rediscover conservation of energy."

Model-Free Prediction of Large Spatiotemporally Chaotic Systems from Data: A Reservoir Computing Approach

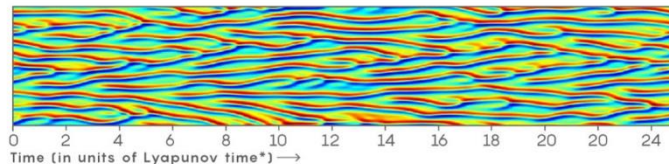
Jaideep Pathak, Brian Hunt, Michelle Girvan, Zhixin Lu, and Edward Ott
Phys. Rev. Lett. 120, 024102 – Published 12 January 2018

Training Computers to Tame Chaos

A machine-learning algorithm has been shown to accurately predict a chaotic system far further into the future than previously possible.

A Chaos Model

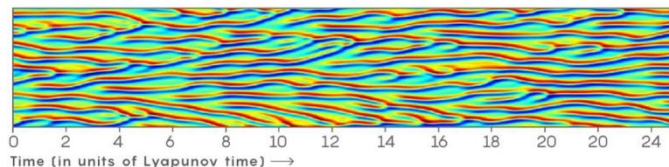
Researchers started with the evolving solution to the Kuramoto-Sivashinsky equation, which models propagating flames:



* Lyapunov time = Length of time before a small difference in the system's initial state begins to diverge exponentially. It typically sets the horizon of predictability, which varies from system to system.

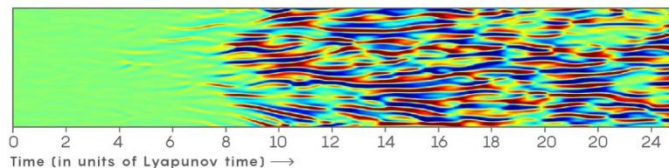
B Machine Learning

After training itself on data from the past evolution of the Kuramoto-Sivashinsky system, the "reservoir computing" algorithm predicts its future evolution:



A - B Do They Match?

Subtracting B from A shows that the algorithm accurately predicts the model out to an impressive 8 Lyapunov times, before chaos ultimately prevails:



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.

Anything not otherwise mentioned follows Apache License 2.0.