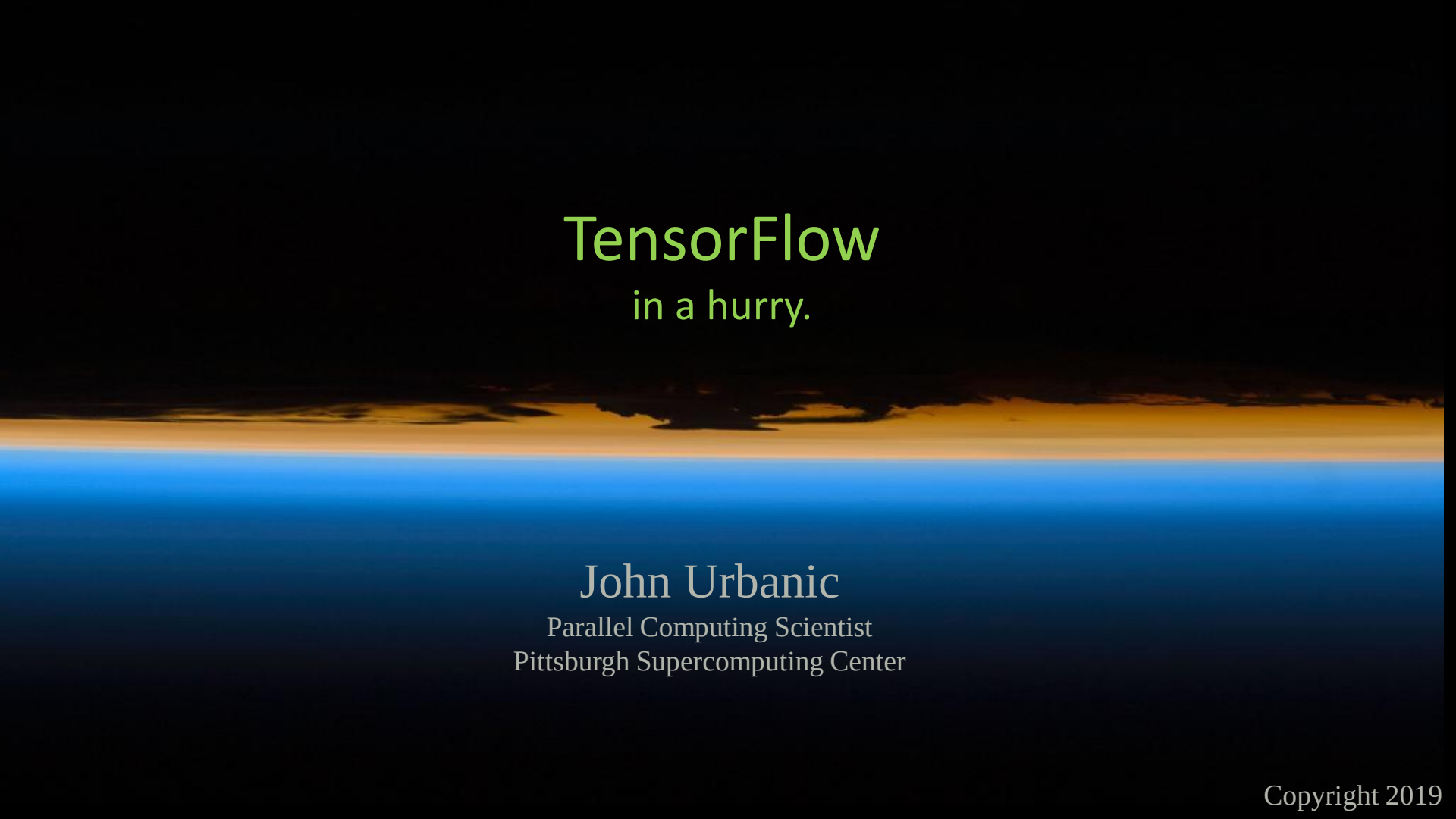


John Urbanic
Parallel Computing Scientist
Pittsburgh Supercomputing Center

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:

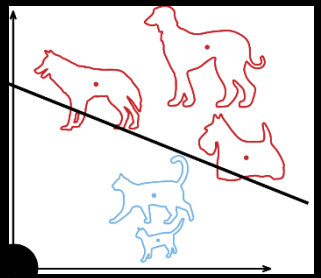
- Widespread parallel computing (GPUs)
- Big data training sets

Two Perspectives

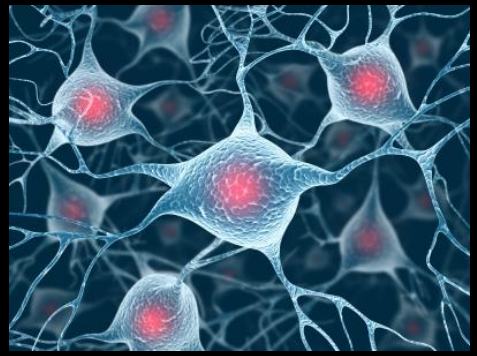
There are really two common ways to view the fundaments of deep learning.

- Inspired by biological models.

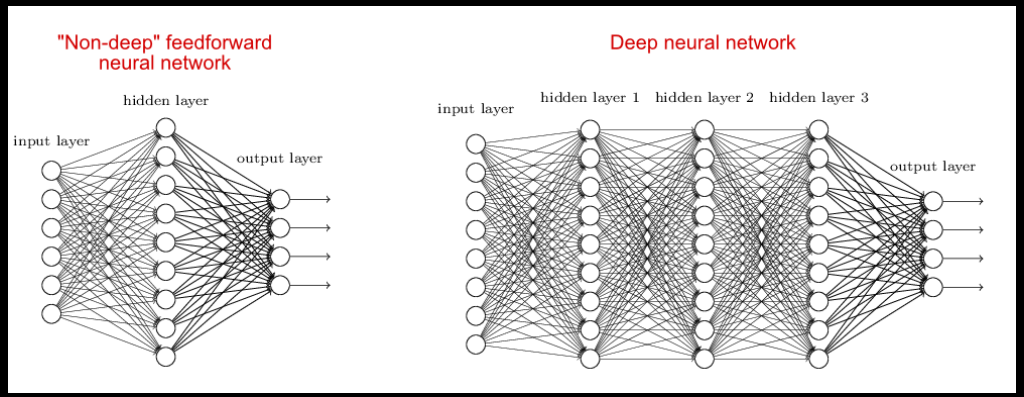
- An evolution of classic ML techniques (the perceptron).



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.



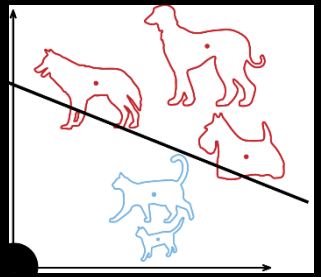
Modeled After The Brain



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

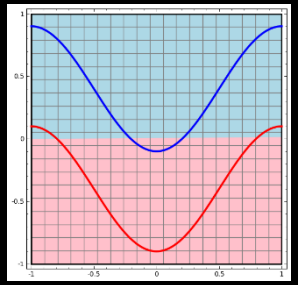
As a Highly Dimensional Non-linear Classifier

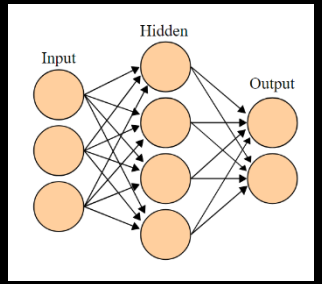
Perceptron

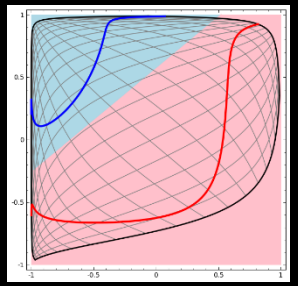


No Hidden Layer
Linear

Network



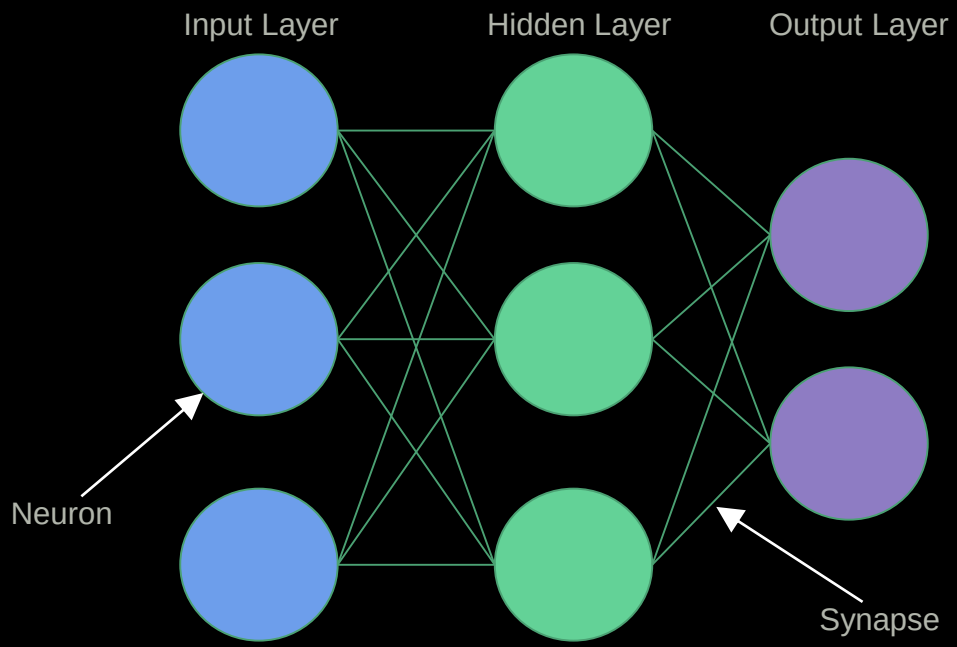




Hidden Layers
Nonlinear

Courtesy: Chris Olah

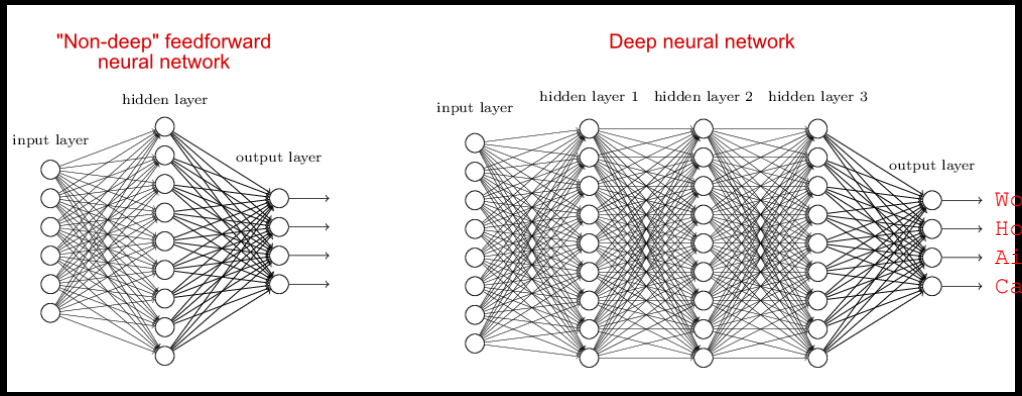
Basic NN Architecture



In Practice

How many inputs?

For an image it could be one (or 3) per pixel.



How deep?

100+ layers have become common.

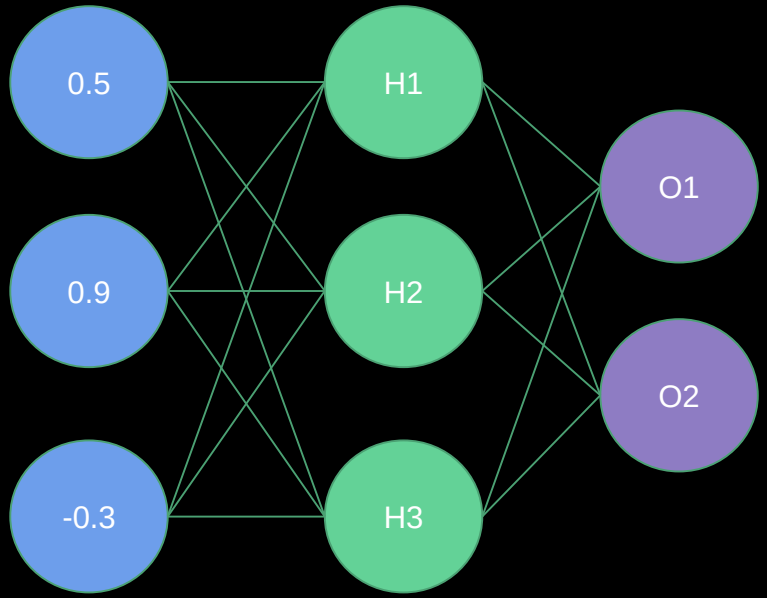
How many outputs?

Might be an entire image.

Or could be discreet set of classification possibilities.

Inference

The "forward" or thinking step

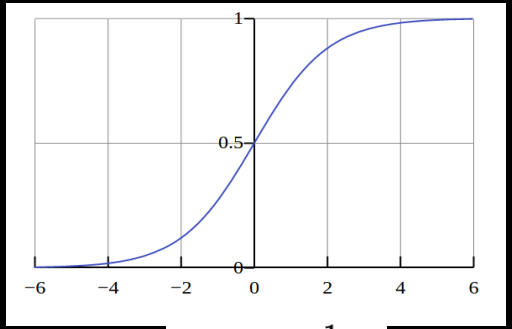


H1 Weights = (1.0, -2.0, 2.0)
H2 Weights = (2.0, 1.0, -4.0)
H3 Weights = (1.0, -1.0, 0.0)

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

Activation Function

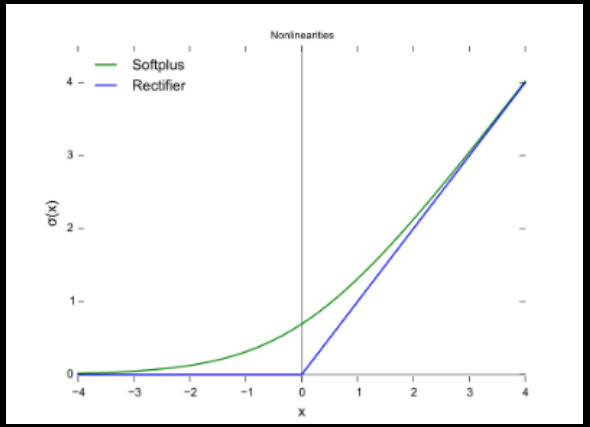
Neurons apply activation functions at these summed inputs. Activation functions are typically non-linear.



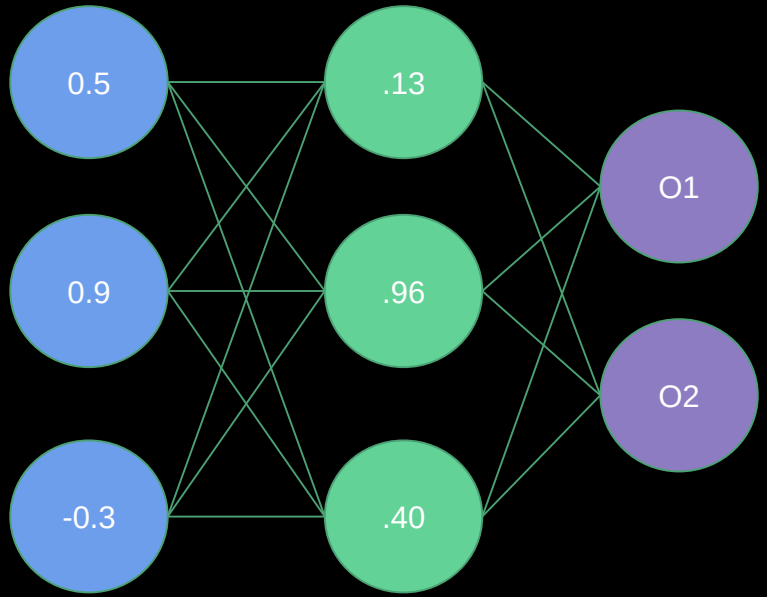
$$S(t) = \frac{1}{1 + e^{-t}}$$

- 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.

- 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 are currently the most popular activation function as they are more efficient than the sigmoid or hyperbolic tangent.
    - Sparse activation: In a randomly initialized network, only 50% of hidden units are active.
    - Better gradient propagation: Fewer vanishing gradient problems compared to sigmoidal activation functions that saturate in both directions.
    - Efficient computation: Only comparison, addition and multiplication.
    - There are Leaky and Noisy variants.



Inference

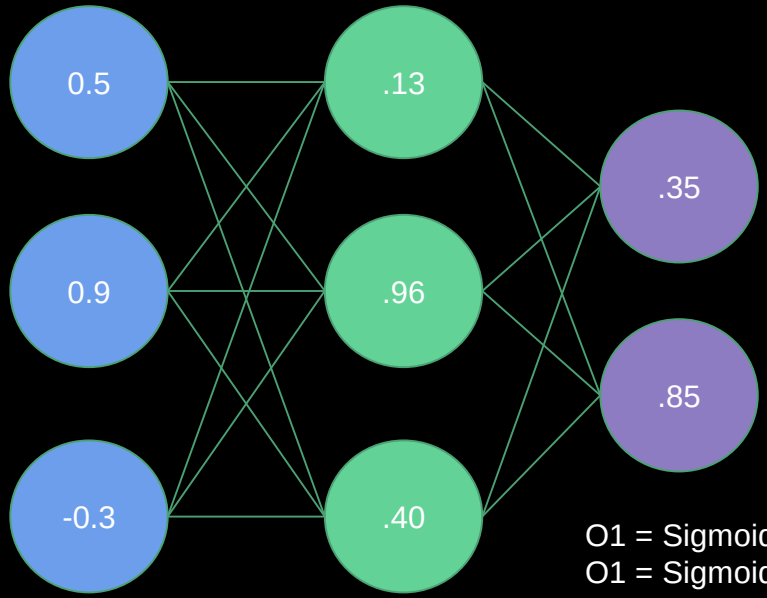


H1 Weights = (1.0, -2.0, 2.0)
H2 Weights = (2.0, 1.0, -4.0)
H3 Weights = (1.0, -1.0, 0.0)

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

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

Inference



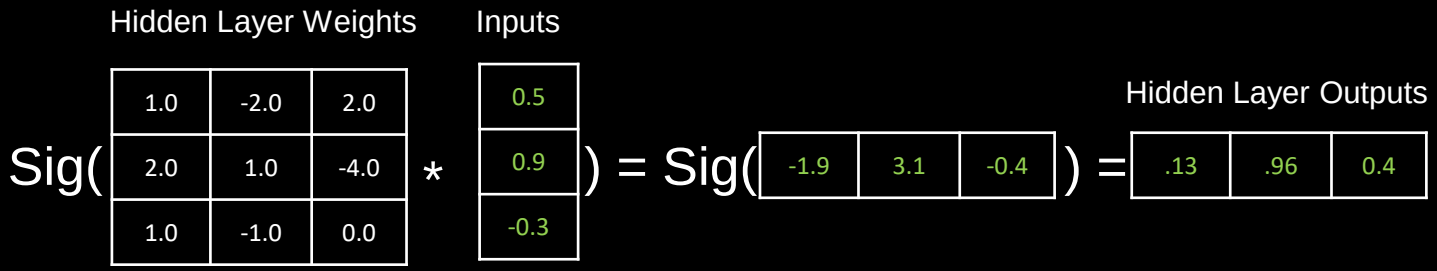
H1 Weights = (1.0, -2.0, 2.0)
H2 Weights = (2.0, 1.0, -4.0)
H3 Weights = (1.0, -1.0, 0.0)

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

O1 = Sigmoid(.13 * -3.0 + .96 * 1.0 + .40 * -3.0) = Sigmoid(-.63) = .35
O1 = Sigmoid(.13 * 0.0 + .96 * 1.0 + .40 * 2.0) = 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)



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:

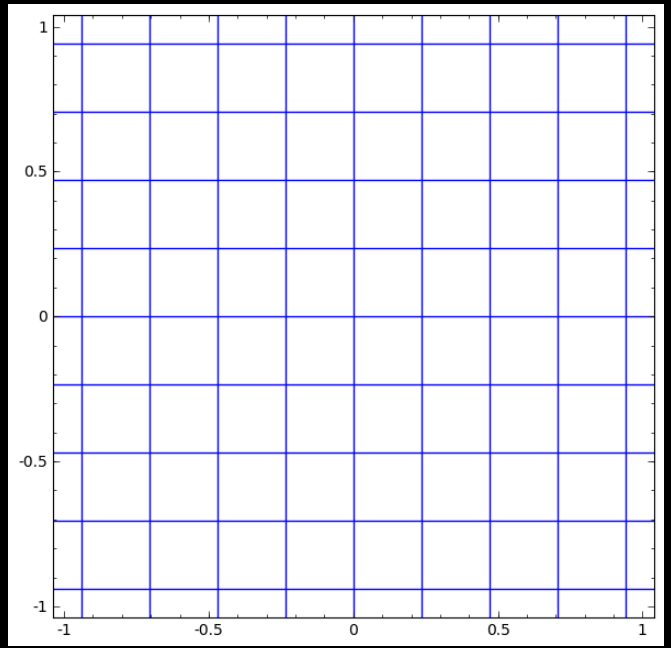
Linear + Nonlinear

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, rotation* and *scale*.

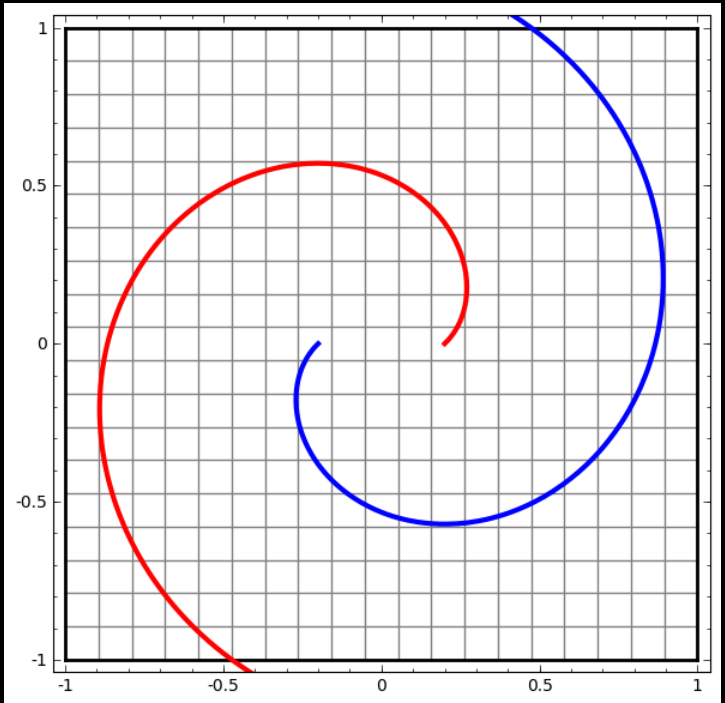
The bias provides the *translation*.

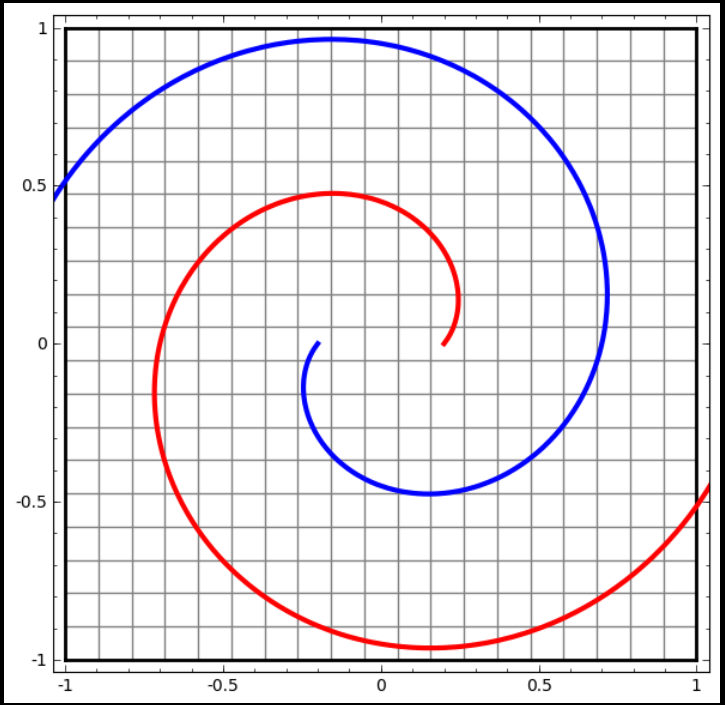
The activation function provides the *warp*.



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.

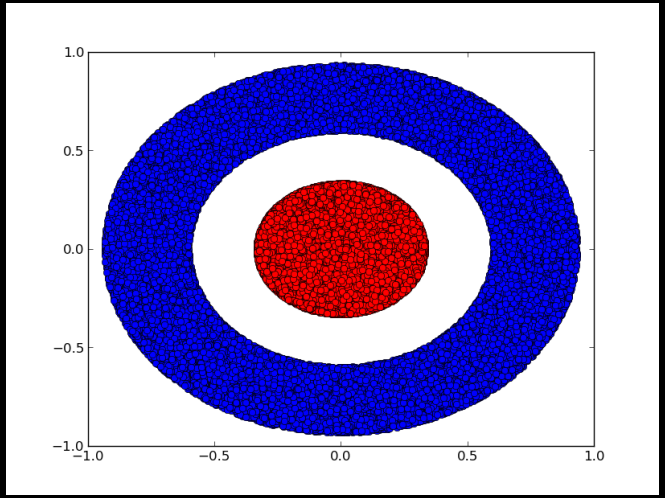




Courtesy: Chris Olah

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:

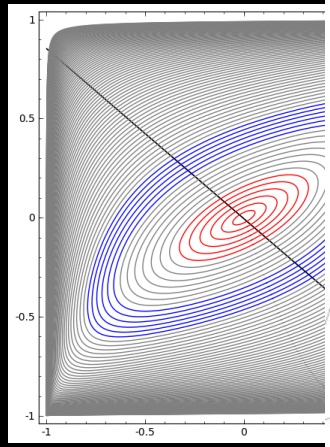


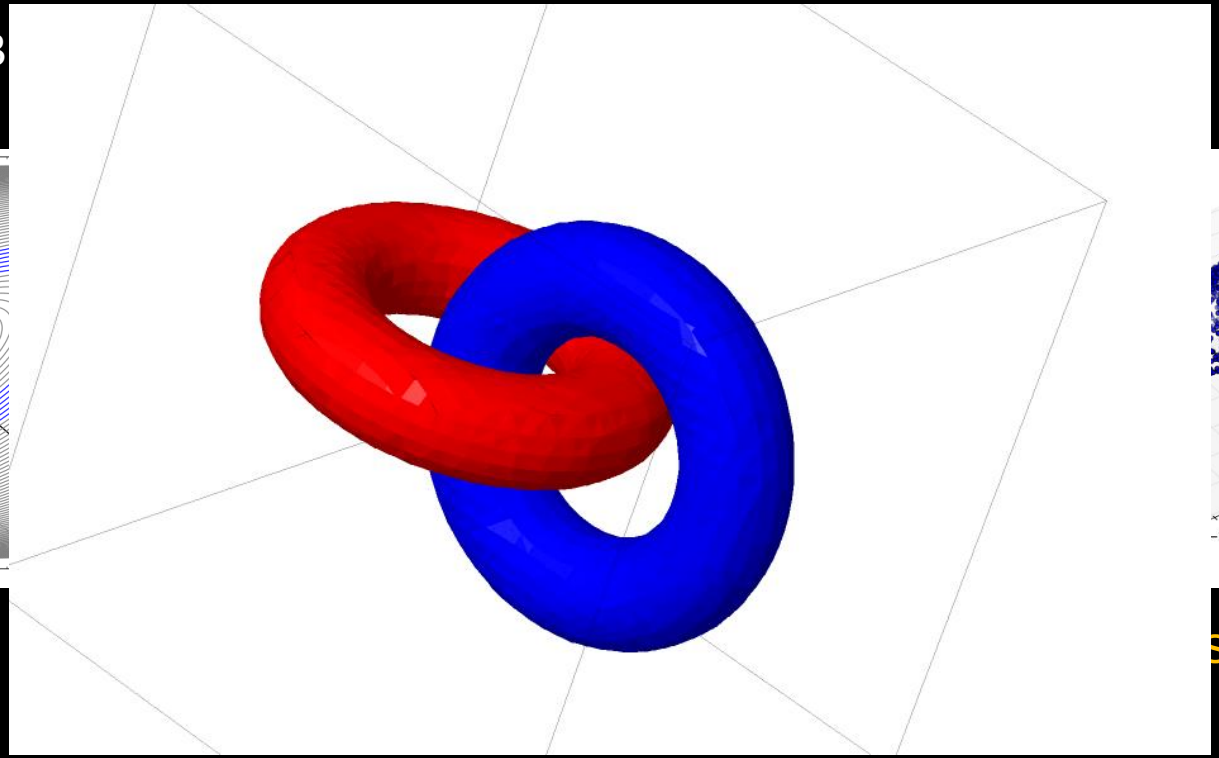
Can a neural net do this with twisting and deforming? What good does it do to have more than two dimensions with a 2D dataset?

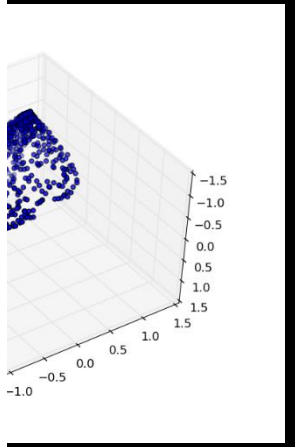
Courtesy: Chris Olah

Working In Higher Dimensions

It takes at least 3







Trying

s in 3D

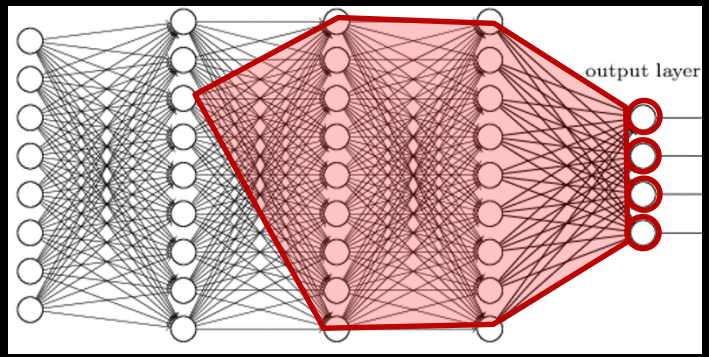
Greater depth allows us to stack these operations, and can be very effective. The gains from depth are harder to characterize.

Courtesy: Chris Olah

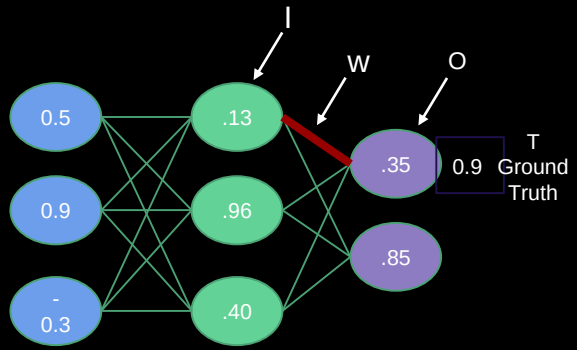
Training Neural Networks

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: $E(w_1,w_2,w_3,...i_1,...t_1,...)$. 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.



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

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

For Sigmoid

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

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

Note that the role of the gradient, $\frac{\partial E}{\partial w_3}$, here means that it becomes a problem if it vanishes. This is an issue for very deep networks.

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.

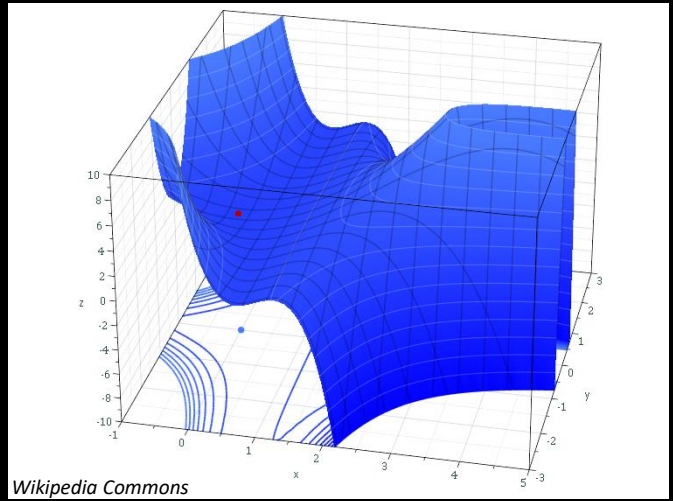
Solvers

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.



Wikipedia Commons

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

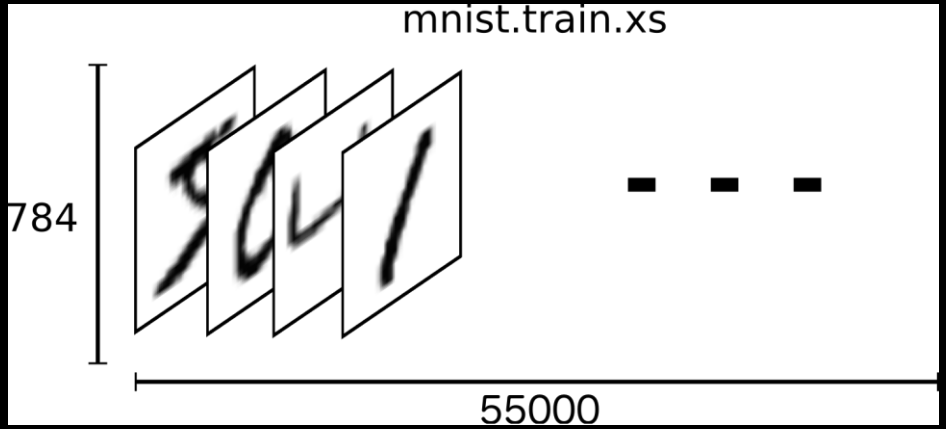
We now know enough to attempt a problem. Only because the TensorFlow framework fills in a lot of the details that we have glossed over. That is one of its functions.

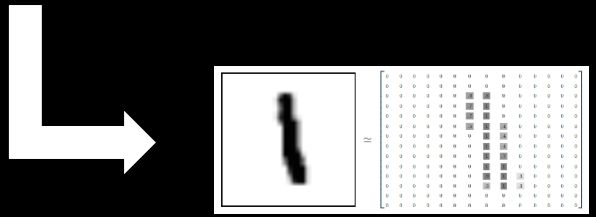
Our problem will be character recognition. We will learn to read handwritten digits by training on a large set of 28x28 greyscale samples.

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

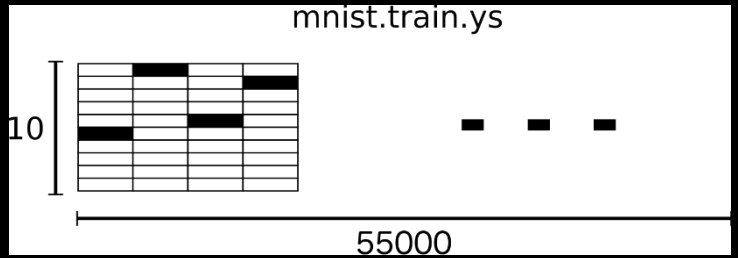
Specifically we will have a file with 55,000 of these numbers.





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
-rw-r--r-- 1 urbanic pscstaff 9912422 May
-rw-r--r-- 1 urbanic pscstaff   28881 May
```

To start TensorFlow:

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

Two Other Ways To Play Along

Run the python programs from the command line:

```
gpu42% python CNN.py
```

Invoke them from within the python shell:

```
>>> execfile('CNN.py')
```

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.        ,  0.02352941,  0.76470596,
        0.99607849,  1.        ,  0.93725497,  0.1137255 ,  0.        ,
        0.        ,  0.        ,  0.        ,  0.        ,  0.        ,
...
...
...
```

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.

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 Cross Entropy Loss

```
>>> 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 rang
>>>  batch_xs, ba
>>>  sess.run(tra
>>>
>>> correct_predi
>>> accuracy = tf
>>> print(sess.ru
```

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 good way to define the error across all possibilities. Better than squared error, which we have been using until now. It is defined as $-\Sigma\ y\_\ \log\ y$, or if this really is a 0, y_=(1,0,0), and

$$-1\log(0.87) - 0\log(0.0001) - 0\log(0.13) = -\log(0.87) = -0.13$$

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.

Next week!

Intro Exercise

As simple as our network is, it has several parameters that I haven't really justified. These *hyperparameters* may allow us to solve our problem more accurately, or more efficiently. The search for optimal hyperparameters is the core of machine learning, and certainly deep learning.

1) What are the hyperparameters in our current implementation? Hint: There are at least 3 very significant ones.

2) Can you find better values for any of them? Hint: Graphing is "state of the art" here.

3) If you can improve on the given implementation, send your results (urbanic@psc.edu) and we will discuss next week.