DESCRIPTION OF ELEMENTS

- `X` Input dataset matrix where each row is a training example
- `y` Output dataset matrix where each row is a training example
- `l0` First Layer of the Network, specified by the input data
- `l1` Second Layer of the Network, otherwise known as the hidden layer
- `syn0` First layer of weights, Synapse 0, connecting l0 to l1.
- `*` Elementwise multiplication, so two vectors of equal size are multiplying corresponding values 1-to-1 to generate a final vector of identical size.
- `-` Elementwise subtraction, so two vectors of equal size are subtracting corresponding values 1-to-1 to generate a final vector of identical size. `x.dot(y)` If x and y are vectors, this is a dot product. If both are matrices, it's a matrix-matrix multiplication. If only one is a matrix, then it's vector matrix multiplication.

Line 04: This is our "nonlinearity". While it can be several kinds of functions, this nonlinearity maps a function called a "sigmoid". A sigmoid function maps any value to a value between 0 and 1. We use it to convert numbers to probabilities. It also has several other desirable properties for training neural networks.

Line 05: Notice that this function can also generate the derivative of a sigmoid (when deriv=True). One of the desirable properties of a sigmoid function is that its output can be used to create its derivative. If the sigmoid's output is a variable "out", then the derivative is simply out * (1-out). This is very efficient.

If you're unfamililar with derivatives, just think about it as the slope of the sigmoid function at a given point (as you can see above, different points have different slopes). For more on derivatives, check out this derivatives tutorial from Khan Academy.

```
import numpy as np

# sigmoid function
def nonlin(x,deriv=False):
    if(deriv==True):
        return x*(1-x)
    return 1/(1+np.exp(-x))
```

- Line 10: This initializes our input dataset as a numpy matrix. Each row is a single "training example". Each column corresponds to one of our input nodes. Thus, we have 3 input nodes to the network and 4 training examples.
- Line 16: This initializes our output dataset. In this case, I generated the dataset horizontally (with a single row and 4 columns) for space. ".T" is the transpose function. After the transpose, this y matrix has 4 rows with one column. Just like our input, each row is a training example, and each column (only one) is an output node. So, our network has 3 inputs and 1 output.

```
# input dataset
X = np.array([  [0,0,1],
                [0,1,1],
                [1,0,1],
                [1,1,1] ])

# output dataset
y = np.array([[0,0,1,1]]).T
```

- Line 20: It's good practice to seed your random numbers. Your numbers will still be randomly distributed, but they'll be randomly distributed in exactly the same way each time you train. This makes it easier to see how your changes affect the network.
- Line 23: This is our weight matrix for this neural network. It's called "syn0" to imply "synapse zero". Since we only have 2 layers (input and output), we only need one matrix of weights to connect them.

Its dimension is (3,1) because we have 3 inputs and 1 output. Another way of looking at it is that l0 is of size 3 and l1 is of size 1. Thus, we want to connect every node in l0 to every node in l1, which requires a matrix of dimensionality (3,1). :)

Also notice that it is initialized randomly with a mean of zero. There is quite a bit of theory that goes into weight initialization. For now, just take it as a best practice that it's a good idea to have a mean of zero in weight initialization.

Another note is that the "neural network" is really just this matrix. We have "layers" l0 and l1 but they are transient values based on the dataset. We don't save them. All of the learning is stored in the syn0 matrix.

```
# seed random numbers to make calculation
# deterministic (just a good practice)
np.random.seed(1)

# initialize weights randomly with mean 0
syn0 = 2*np.random.random((3,1)) - 1
```

- Line 25: This begins our actual network training code. This for loop "iterates" multiple times over the training code to optimize our network to the dataset.
- Line 28: Since our first layer, l0, is simply our data. We explicitly describe it as such at this point. Remember that X contains 4 training examples (rows). We're going to process all of them at the same time in this implementation. This is known as "full batch" training. Thus, we have 4 different l0 rows, but you can think of it as a single training example if you want. It makes no difference at this point. (We could load in 1000 or 10,000 if we wanted to without changing any of the code).
- Line 29: This is our prediction step. Basically, we first let the network "try" to predict the output given the input. We will then study how it performs so that we can adjust it to do a bit better for each iteration.

This line contains 2 steps. The first matrix multiplies l0 by syn0. The second passes our output through the sigmoid function. Consider the dimensions of each:

```
(4 x 3) dot (3 x 1) = (4 x 1)
```

Matrix multiplication is ordered, such the dimensions in the middle of the equation must be the same. The final matrix generated is thus the number of rows of the first matrix and the number of columns of the second matrix.

Since we loaded in 4 training examples, we ended up with 4 guesses for the correct answer, a (4 x 1) matrix. Each output corresponds with the network's guess for a given input. Perhaps it becomes intuitive why we could have "loaded in" an arbitrary number of training examples. The matrix multiplication would still work out. :)

- Line 32: So, given that l1 had a "guess" for each input. We can now compare how well it did by subtracting the true answer (y) from the guess (l1). l1_error is just a vector of positive and negative numbers reflecting how much the network missed.
- Line 36: Now we're getting to the good stuff! This is the secret sauce! There's a lot going on in this line, so let's further break it into two parts.

```
for iter in xrange(1000): # was 10000

    # forward propagation
    l0 = X
    l1 = nonlin(np.dot(l0,syn0))

    # how much did we miss?
    l1_error = y - l1
```

```
    # multiply how much we missed by the
    # slope of the sigmoid at the values in l1
    l1_delta = l1_error * nonlin(l1,True)

    # update weights
    syn0 += np.dot(l0.T,l1_delta)

print("Output After Training:")
print(l1)
```

If l1 represents these three dots, the code above generates the slopes of the lines below. Notice that very high values such as x=2.0 (green dot) and very low values such as x=-1.0 (purple dot) have rather shallow slopes. The highest slope you can have is at x=0 (blue dot). This plays an important role. Also notice that all derivatives are between 0 and 1.

Entire Statement: The Error Weighted Derivative

```
l1_delta = l1_error * nonlin(l1,True)
```

THE "JUST THE CODE" SECTION

```
### "just the code" section
import numpy as np
X = np.array([ [0,0,1],[0,1,1],[1,0,1],[1,1,1] ])
y = np.array([[0,1,1,0]]).T
syn0 = 2*np.random.random((3,4)) - 1
syn1 = 2*np.random.random((4,1)) - 1
for j in xrange(1000):  # was 60000
    l1 = 1/(1+np.exp(-(np.dot(X,syn0))))
    l2 = 1/(1+np.exp(-(np.dot(l1,syn1))))
    l2_delta = (y - l2)*(l2*(1-l2))
    l1_delta = l2_delta.dot(syn1.T) * (l1 * (1-l1))
    syn1 += l1.T.dot(l2_delta)
    syn0 += X.T.dot(l1_delta)
```

THE FULL THING IN ONE SHOT

```
import numpy as np

# sigmoid function
def nonlin(x,deriv=False):
    if(deriv==True):
        return x*(1-x)
    return 1/(1+np.exp(-x))


# input dataset
X = np.array([  [0,0,1],
                [0,1,1],
                [1,0,1],
                [1,1,1] ])

# output dataset
```

```
y = np.array([[0,0,1,1]]).T

# seed random numbers to make calculation
# deterministic (just a good practice)
np.random.seed(1)

# initialize weights randomly with mean 0
syn0 = 2*np.random.random((3,1)) - 1

for iter in xrange(1000): # was 10000

    # forward propagation
    l0 = X
    l1 = nonlin(np.dot(l0,syn0))

    # how much did we miss?
    l1_error = y - l1

    # multiply how much we missed by the
    # slope of the sigmoid at the values in l1
    l1_delta = l1_error * nonlin(l1,True)

    # update weights
    syn0 += np.dot(l0.T,l1_delta)

print("Output After Training:")
print(l1)

l1_delta = l1_error * nonlin(l1,True)
```