Note Title 2/11/2019

```python
epoch = 5000 # how many epochs? (each epoch will pass all 4 data points through)
err = np.zeros((epoch,1)) # lets record error to plot (get a convergence plot)
inds = np.asarray([0,1,2,3]) # array of our 4 indices (data point index references)
for k in range(epoch):

    # init error
    err[k] = 0

    # random shuffle of data each epoch
    inds = np.random.permutation(inds)
    for i in range(4):

        # what index?
        inx = inds[i]

        # forward pass
        v = np.ones((3, 1))
        v[0] = np.dot(X[inx,:], n1_w) # neuron 1 fires (x as input)
        v[0] = sigmoid(v[0]) # neuron 1 sigmoid
        v[1] = np.dot(X[inx,:], n2_w) # neuron 2 fires (x as input)
        v[1] = sigmoid(v[1])
        oo = np.dot(np.transpose(v), n3_w) # neuron 3 fires, taking neuron 1 and 2 as input
        o = sigmoid(oo) # hey, result of our net!!!

        # error
        err[k] = err[k] + ((1.0/2.0) * np.power((o - y[inx]), 2.0))

        # backprop time!!!
```

layer 1

$\vec{w}_1^T \vec{x} = v_1$

$y_1 = \varphi(v_1)$

v_2

$y_2 = \varphi(\vec{w}_2^T \vec{x})$

output

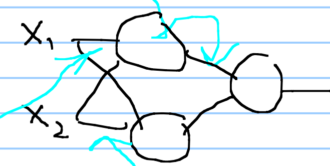
$y_3 = \varphi\left(\vec{w}_3^T \begin{bmatrix} y_1 \\ y_2 \end{bmatrix}\right)$

v_3

```python
# output layer
delta_1 = (-1.0)*(y[inx] - o)
delta_2 = sigmoid(o,derive=True)
# now, lets prop it back to the weights
delta_ow = np.ones((3, 1))
# format is
# delta_index = (input to final neuron) * (Err derivative * Sigmoid derivative)
delta_ow[0] = v[0]  *  (delta_1*delta_2)
delta_ow[1] = v[1]  *  (delta_1*delta_2)
delta_ow[2] = v[2]  *  (delta_1*delta_2)

# neuron n1
delta_3 = sigmoid(v[0],derive=True)
# same, need to prop back to weights
delta_hw1 = np.ones((3, 1))
# format
#              input      this Sig der     error from output     weight to output neuron
delta_hw1[0] = X[inx,0]  *  delta_3  *  ((delta_1*delta_2)   *n3_w[0])
delta_hw1[1] = X[inx,1]  *  delta_3  *  ((delta_1*delta_2)   *n3_w[0])
delta_hw1[2] = X[inx,2]  *  delta_3  *  ((delta_1*delta_2)   *n3_w[0])

# neuron n2
delta_4 = sigmoid(v[1],derive=True)
# same, need to prop back to weights
delta_hw2 = np.ones((3, 1))
delta_hw2[0] = X[inx,0]  *  delta_4  *  ((delta_1*delta_2)   *n3_w[1])
delta_hw2[1] = X[inx,1]  *  delta_4  *  ((delta_1*delta_2)   *n3_w[1])
delta_hw2[2] = X[inx,2]  *  delta_4  *  ((delta_1*delta_2)   *n3_w[1])

# update rule, so old value + eta weighted version of delta's above!!!
n1_w = n1_w + (-1.0) * eta * delta_hw1
n2_w = n2_w + (-1.0) * eta * delta_hw2
n3_w = n3_w + (-1.0) * eta * delta_ow
```

$$\frac{\partial E}{\partial y_3} = (-1)(d - y_3)$$

$$\frac{\partial y_3}{\partial v_3} = y_3(1-y_3)$$



$$\underbrace{\frac{\partial E}{\partial y_3}\frac{\partial y_3}{\partial v_3}}_{\delta_1 \delta_2}\frac{\partial v_3}{\partial w_3}$$

$\delta_1 \delta_2$

$$\frac{\partial v_3}{\partial y_1} \quad \frac{\partial y_1}{\partial v_1} \quad \frac{\partial v_1}{\partial w_1}$$

δ_1

ΔW

learn rate