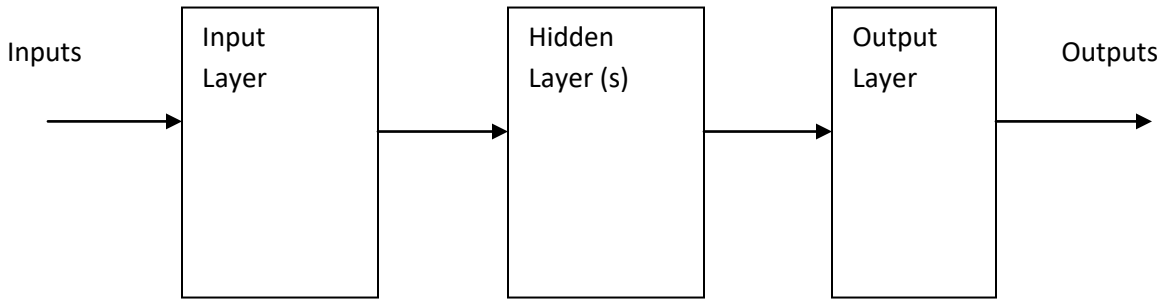
Lesson 21 Introduction To Neural Networks Last Update June 2, 2021

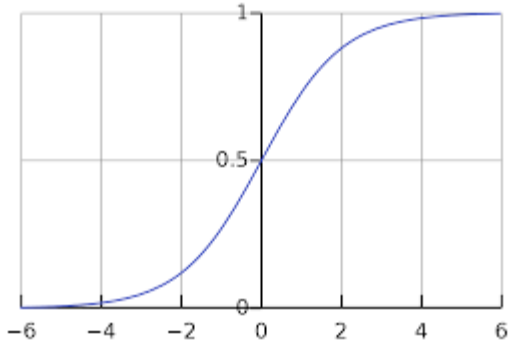
Neural Networks simulate how the human brain operates. Neural Networks are basically many mathematical operations. I doubt the human brain actually performs all these calculations. The human brain consists of billions of neurons for it thinking and decision process. Each neuron has an activation level that it fires on.

A neural network consists of inputs, hidden layers neurons and output layer neurons. A neural network may have 1 hidden layers or many hidden layers.

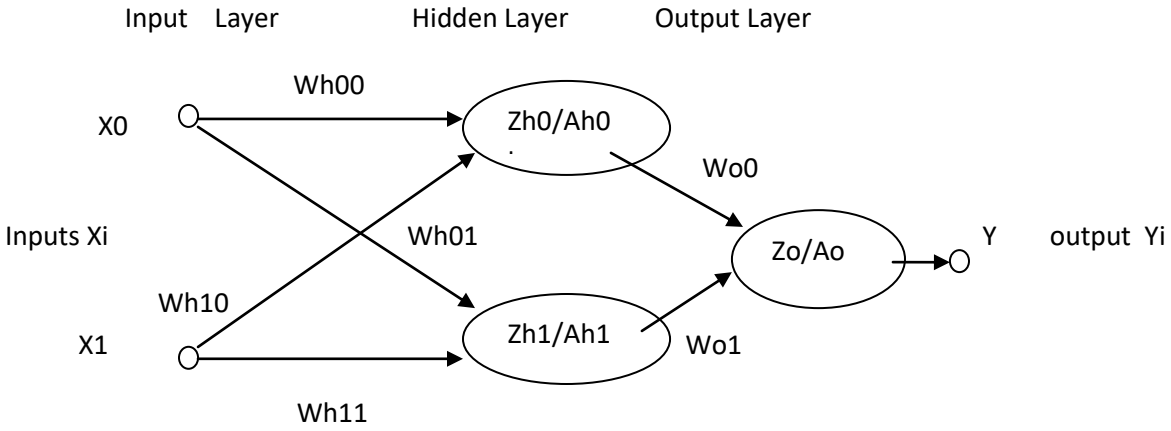


Each hidden layer has stored values called weights. The output of the hidden layer are the sums of the preceding inputs and stored weight. The output of the neuron is sent to an activation function. A neuron fires when it reaches a certain threshold determined by the activation function. The threshold is the calculation of the neuron weight and input values and the activation function mathematical formula. The activation function used is called **sigmoid**. The sigmoid function produces values between 0 and 1 for input values of –inf to + inf where input 0 has the value .5

$$\sigma(z) = \frac{1}{1 + e^{-z}}$$



A Simple two layer feed-forward neural network is as follows. Feed forward means in this network, the information moves in only one direction, forward from the input nodes, through the hidden nodes (if any) then to the output nodes.



Xi = inputs
Wh = weights hidden
Zh = internal hidden
Ah = activation hidden
Z o = internal output
Ao = activation output
Y = output

The output of each hidden layer neuron Zh is the sum of the inputs and hidden layer weights.

```
zh0 = x0 * wh00 + x1 * wh10
zh1 = x0 * wh01 + x1 * wh11
```

Each hidden layer is sent to a sigmoid activation function

```
Ah0 = sigmoid(Zh0)
Ah1 = sigmoid(Zh1)
```

The output of each output layer neuron Zo is the sum of the hidden layer outputs and output layer weights.

Zo = Ah0 * wo0 + Ah1 * wo1

Each output layer is sent to a sigmoid activation function

A0 = sigmoid(Zo)

Our simple neural network example will be the classic XOR Gate with 4 possible inputs X0 and X1 and a single output Y. The truth table for a XOR Gate XOR gate is as follows:

XOR Truth Table:

X0	**X1**	**Y**
0	0	0
0	1	1
1	0	1
1	1	0

When the inputs have the same value the output is a 0. When the inputs are different values the output produces a 1.

The inputs to our input layer is an array of test values **Xi**.

Xi = np.array([[0, 0],[0, 1], [1, 0],[1, 1]])

We use each one individually.

[X0 X1]

The hidden layers contain initial random weights **Wh**

Wh = np.random.random((2, 2))

[[Wh00 Wh01]
[Wh10 Wh11]]

The output layers contain initial random weights **Wo**.

Wo = np.random.random(2)

[[W00 W01]]

We also have an array of outputs Yi that is our desired result

Yi = np.array([0, 1, 1, 0])

again we use each one individually.

[[Y0]]

We use numpy arrays for all our calculations.

Our inputs are X0 and X1

[X0 X1]

Each hidden layer has an internal **Zh** and hidden calculated **Ah**.

The internal hidden calculation is the weights multiples by the transpose of the inputs where **transpose X** means changing matrix X from 1 by n matrix to n by 1 matrix

Zh = Wh X^T (dot product)

$$\begin{vmatrix} Zh0 \\ Zh1 \end{vmatrix} = \begin{vmatrix} Wh00 & Wh01 \\ Wh10 & Wh11 \end{vmatrix} \bullet \begin{vmatrix} X0 \\ X1 \end{vmatrix}$$

Zh0 = Wh00 * X0 + Wh01 * X1

Zh1 = Wh10 * X0 + Wh11 * X1

The external output is the result of applying the sigmoid σ function.

$$\sigma(z) = \frac{1}{1 + e^{-z}}$$

Each internal output needs an sigmoid activation function

The activation output equation would be:

Ah = σ(Zh)

The output layer uses the output layer as its input. Our output equation from the hidden layer to output would be

Zo = Wo * Zh

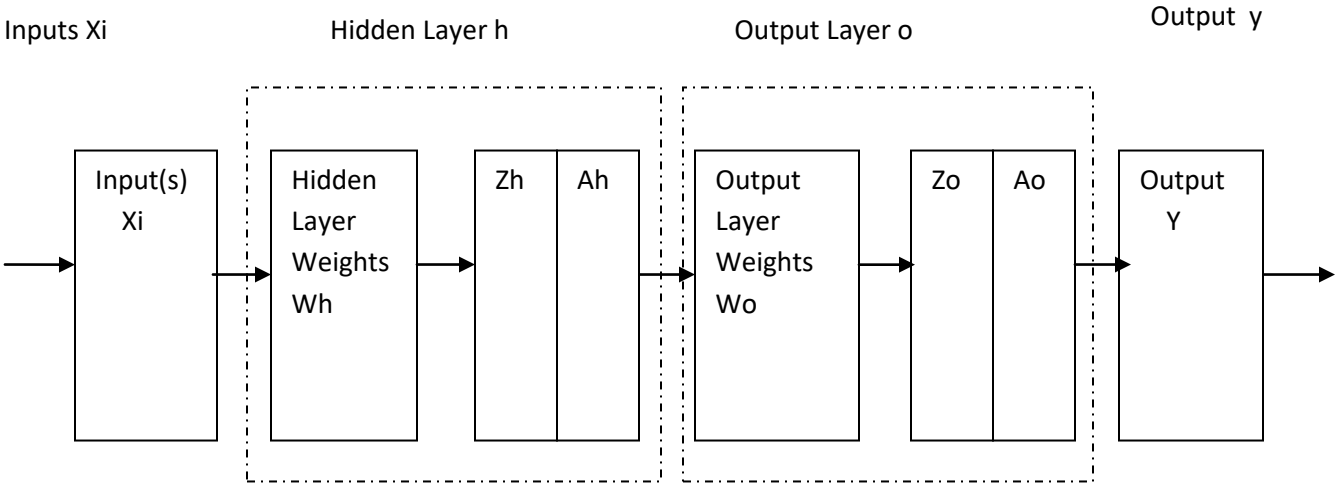
| Z0 | = | Wo0 Wo1 | * | Zh|

Z0 = Wo0 * Zh + Wo01* Zh

The activation equation would be:

Ao = σ(Zo)

Our neural network model is now:



Adding bias

Bias is needed to add a extra value to avoid results of zero.

Bias are initially random values that uses the random function from numpy that returns a random number between 0.0 and 1.0:

bh = [[random number], [random number]]

output layer bias also is a random number between 0 and 1

bo = [random number]

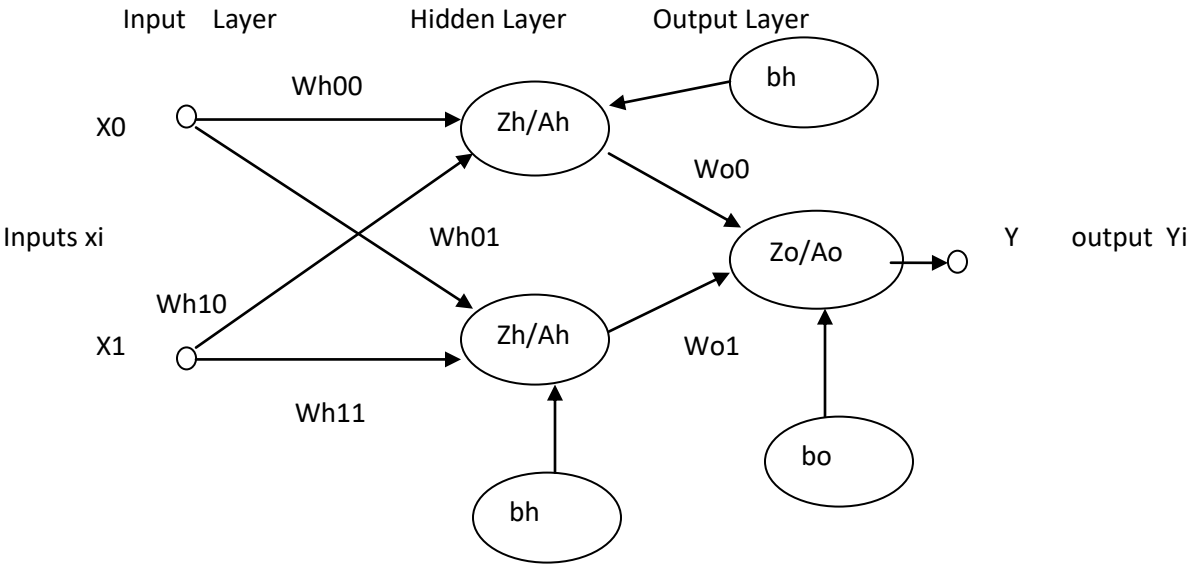
Our new feed forward equations with bias are now

$$\mathbf{Zh = Wh * X^T + bh}$$
$$\mathbf{a(Zh) = 1 / (1 + e^{-Zh})}$$

$$\mathbf{Zo = Wo * Zh + bo}$$

$$\mathbf{A(Zo) = 1/ (1 + e^{-Zo})}$$

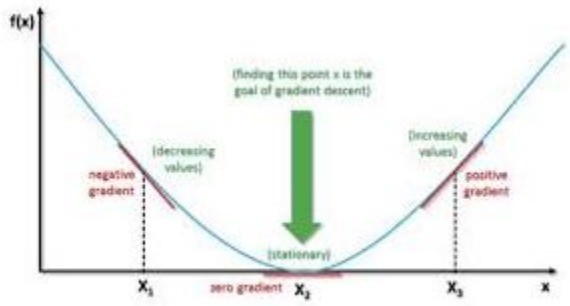
Our neural network with bias bh and bo is now:



Training Neural Network

We train the neural network by feeding it known inputs and measuring the error between the desired output and the actual output. We adjust the weights of the neural network to minimize the error. This is known as **back propagation** using **gradient descent**. We also adjust the bias to produce minimum error.

Gradient Descent uses mathematical equations to find the minimum point in a curve.



Output error is calculated from the difference of the desired output to the actual output.

Output Error = desired output – actual output

Y is the actual output where as Ao is the actual output from the neural network.

Error = Y – Ao

We use the **SigmoidDerivative** to adjust the weights using gradient descent. The **SigmoidDerivative** is the derivative of the Sigmoid function **f(z)**

$$f(z) = 1 / (1 + e^{-z})$$

$$f'(z) = f(z) (1 – f(z))$$

The sigmoid derivative is the value of f(x) for x of the gradient slope.

Cost function

A cost function is simply the function that finds the cost of the given predictions. We will use the mean squared error cost function. The mean squared error cost function can be mathematically represented as:

$$MSE = \frac{1}{\mathrm{n}} \sum_{i=1}^{n} (predicted - observed)^2$$

Here n is the number of observations.

In order to minimize the cost, we need to find the weight and bias values for which the cost function returns the smallest value possible. The smaller the cost, the more correct our predictions are.

$$repeat\ until\ convergence : \left\{ w_j := w_j - \alpha \frac{\partial}{\partial w_j} J(w_0, w_1 \ldots\ldots w_n) \right\}$$

In the above equation, **J** is the cost function.

Basically what the above equation says is: find the partial derivative of the cost function with respect to each weight and bias and subtract the result from the existing weight values to get the new weight values. A partial derivative is a way to find the slope in either the x or y direction, at the point indicated.

The derivative of a function gives us its slope at any given point. To find if the cost increases or decreases, given the weight value, we can find the derivative of the function at that particular weight value. If the cost increases with the increase in weight, the derivative will return a positive value which will then be subtracted from the existing value. On the other hand, if the cost is decreasing with an increase in weight, a negative value will be returned, which will be added to the existing weight value since negative into negative is positive.

There is an alpha α symbol, which is multiplied by the gradient. This is called the **learning rate.** The learning rate defines how fast our algorithm learns.

We need to repeat the above equation for all the weights and bias until the cost is minimized to the desirable level until we get such values for bias and weights, for which the cost function returns a value close to zero.

Chain rule

We need to differentiate this function with respect to each weight. We will use the chain rule of differentiation for this purpose

The chain rule is a rule for differentiating compositions of functions.

$$D\{f(g(x))\} = f'(g(x))\ g'(x)\,.$$

Let's suppose "d_cost" is the derivate of our cost function with respect to weight "w", we can use chain rule to find this derivative, as shown below:

$$\frac{d\_cost}{dw} = \frac{d\_cost}{d\_pred}\frac{d\_pred}{dz}, \frac{dz}{dw}$$

Where

$$\frac{d\_cost}{d\_pred} = 2\ (\text{predicted} - \text{observed}) = (\text{predicted} - \text{observed})$$

Next we have to find:

$$\frac{d\_pred}{dz}$$

d_pred is simply the sigmoid function and we have differentiated it with respect to input dot product "z".

This is defined as: sigmoid_derivative(z)

Lastly we need to find:

$$\frac{d\_z}{dw}$$

We know that:

z = x0 w0 + x1 w1 + b0

Therefore, derivative with respect to any weight is simply the corresponding input.

$$\frac{d\_z}{dw} = \text{input} = X$$

Hence, our final derivative of the cost function with respect to any weight is:

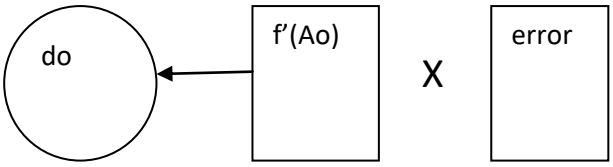
slope = input x dcost/dpred x dpred/dz

= input x (predicted – observed) x sigmoid_derivative(z)

For **back propagation** we start with the output layer and work backwards towards the hidden layer.

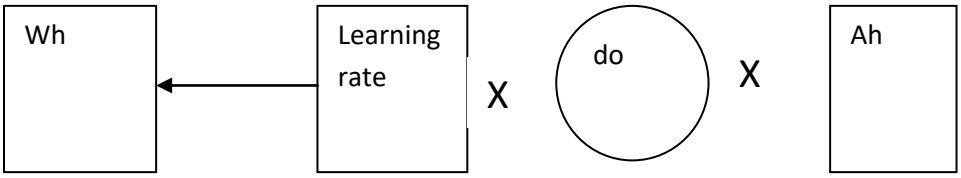
We multiply the sigmoid derivative of the output by the output error to get delta output **do**

do = sigmoid_derivative(A0) * error



We now update the output weight and output bias using the delta output and learning rate sets how fast the neural network learns. A rate to high will make it learn to fast. A rate to low will make it learn too slow.

Wo += learning_rate * do * Ah



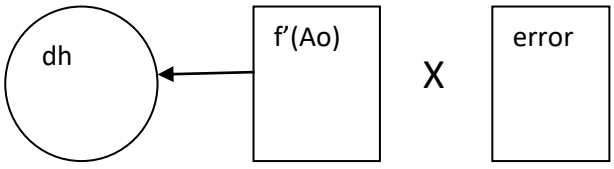
We also update the output bias

bo += learning_rate * do

We now work backwards to the hidden layers and do the same calculations.

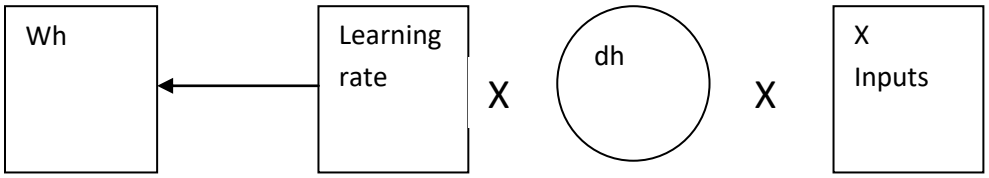
We multiply the sigmoid derivative of the hidden layer output by the delta output * weights of the output layers this give the delta of the hidden layers

dh = derivative(Ah) * do * Wo



We can now update the hidden weights by the learning rate the delta hidden and the X inputs

Wh += learning_rate * dh.reshape(-1,1) * X



We update the hidden layer bias as well.

bh += learning_rate * dh

When we update the weights we have to do element column multiply rather than element row multiply.

| a b | | e | _ | a * e b * e |
| c d | | f | ¯ | c * f d * f |

What **dh.reshape(-1,1)** is doing:

Converts a 1 dimensional array of 2 elements

dh (1 dimensional array of 2 elements)

[-1.486903334369486449e-02 -1.618694149027888329e-02]

To a 2 dimensional aray of 2 rows and 1 column

dh.reshape(-1,1) (changes to 2 rows and 1 column)

[[-1.486903334369486449e-02]

[-1.618694149027888329e-02]]

Here is the complete XOR python code. With explanation comments.

For easy array manipulation we use numpy arrays for our neural network

```
"""
neuralnetworkXOR.py
xor neural network

"""

import numpy as np

# program constants

num_iterations = 5000

learning_rate = .5

tolerence = .1


# print program title

print("XOR Neural Network")

# inputs

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

# outputs

Y = np.array([0, 1, 1, 0])

# hidden layer random weights

Wh = np.random.random((2, 2))


# hidden layer random bias

bh = np.random.random(2)

# output layer random weights

Wo = np.random.random(2)
```

```
# output layer random bias
bo = np.random.random(1)


# sigmoid function
def sigmoid(z):
   return 1 / (1 + np.exp(-z))


# sigmoid derivative function
def sigmoid_derivative(z):
   return z * (1-z)
# forward_propagation to calculate layer outputs
# returns outputs


def forward_propagation(X, Y, Wh,bh,Wo,bo):
   # calculate hidden layer output
   Zh = np.dot(Wh, X.T) + bh
   Ah = sigmoid(Zh)
   # calculate output layer output
   Zo = np.dot(Wo, Ah) + bo
   Ao = sigmoid(Zo)
   return Ah, Ao


# backward propagation to adjust weights using gradient descent
# returns output error
def backward_propagation(X, Y, Wh,Ah,bh,Ao,Wo,bo):
   # desired - actual
```

```
    error = Y - Ao;
    # calculate delta of output layer
    do = sigmoid_derivative(Ao) * error;
    # adjust weights of output layer
    Wo += learning_rate *  do * Ah
    # adjust output bias
    bo += learning_rate *  do
    # calculate delta of hidden layer
    dh = sigmoid_derivative(Ah) * do * Wo
    # adjust weights of hidden layer
    Wh += learning_rate *  dh.reshape(-1,1) * X

    # adjust hidden bias
    bh += learning_rate *  dh
    return error # return output error


# train neural network
print("training Neural Network")
for i in range(num_iterations+1):
    # for each input 00,01,10,11
    for j in range(4):
        # do forward propagation
        A1, A2 = forward_propagation(X[j], Y[j], Wh,bh,Wo,bo)
        # di backward propagation
        error = backward_propagation(X[j], Y[j], Wh,A1,bh,A2,Wo,bo)
```

```
 # early termination if in tolerence
    if np.abs(error) < tolerence:
        break;
# print out error and number of iterations
print("error: ",error,"number iterations: ",i)
# print out results
for j in range(4):
    A1,A2 = forward_propagation(X[j], Y[j], Wh,bh,Wo,bo)
    print(X[j],Y[j],":",A2)
```

Program Output

```
XOR Neural Network
training Neural Network
error:  [-0.09974407] number iterations:  849
[0 0] 0 : [0.08461562]
[0 1] 1 : [0.91374248]
[1 0] 1 : [0.91223311]
[1 1] 0 : [0.09516948]
```

Note: The neural network cannot always converge since the weights start at random values. To force quick convergence you may want to set the initial weights to predetermined values. If the neural network does not converge just rerun it again.

To do:

Change **num_iterations**, **learning_rate** and **tolerence** to reduce error and numberof iterations. Try different initial bias levels 1, -1 and 0. Use matplot to plot the above results cost vs epochs.

Homework Question 1

Try these other Logic gates

AND gate Truth Table

X0	X1	Y
0	0	0
0	1	0
1	0	0
1	1	1

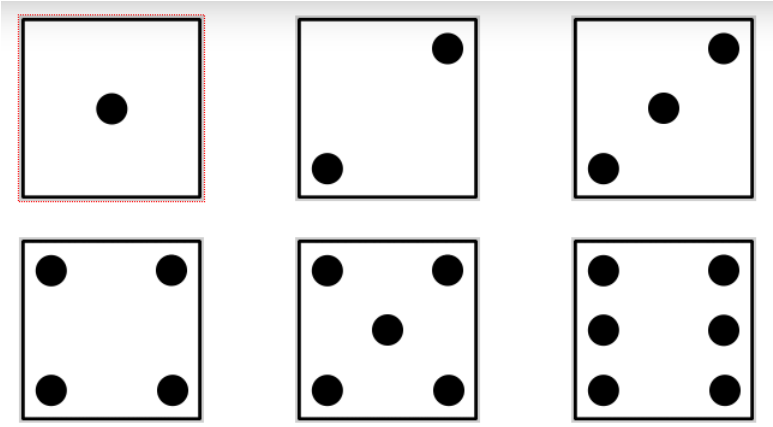
OR gate Truth Table

X0	X1	Y
0	0	0
0	1	1
1	0	1
1	1	1

Call your py file neurualnetworkhomework1.py

Classifying a dice using neural networks

Our next task is to classify a dice using neural networks.



Each dice will fill a 3*3 matrix where each dot is a 1.

0	0	0
0	1	0
0	0	0

For neural networks this will be represents by a 1 dimensional array

0	0	0	0	1	0	0	0	0

```
# dice patterns 1 to 6
X = np.array([[[0,0,0,0,1,0,0,0,0]],
              [[1,0,0,0,0,0,0,0,1]],
              [[1,0,0,0,1,0,0,0,1]],
              [[1,0,1,0,0,0,1,0,1]],
              [[1,0,1,0,1,0,1,0,1]],
              [[1,1,1,0,0,0,1,1,1]]])
```

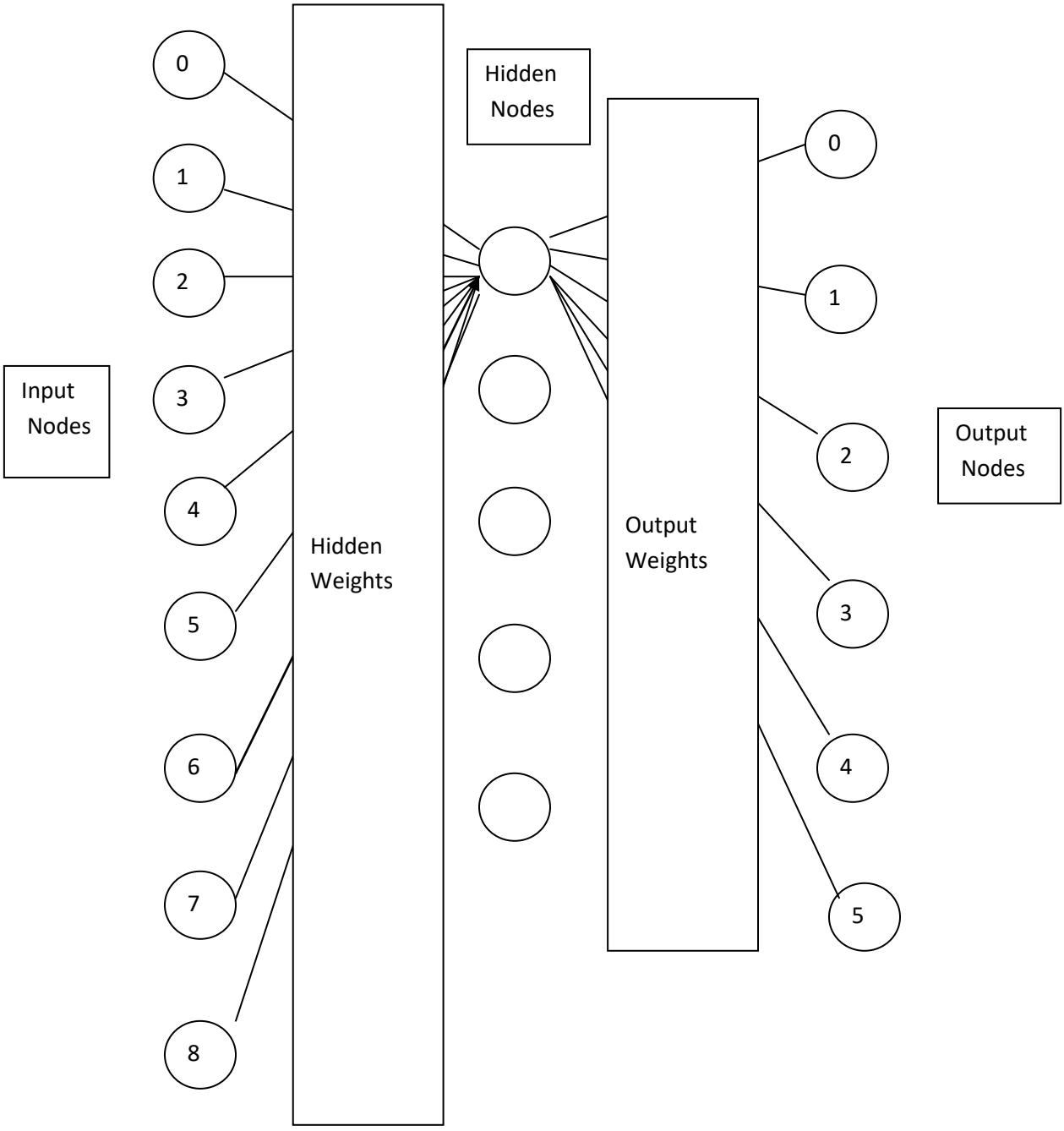
Where the inputs is represents by a 9 by 6 of a 2 dimensional array where each row will represent a dice dots.

```
# outputs
Y = np.array([[[0, 0, 0, 0, 0, 1]],
              [[0, 0, 0, 0, 1, 0]],
              [[0, 0, 0, 1, 0, 0]],
              [[0, 0, 1, 0, 0, 0]],
              [[0, 1, 0, 0, 0, 0]],
              [[1, 0, 0, 0, 0, 0]]])
```

Where the output is represents by a 6 by 6 of a 2 dimensional array where each row will represent a dice number where each column has a 1 for that number

We will have a 9 input neural network to input the dice dots and a 6 output where each output represents a dice number.

We must now decide how many hidden layers we need and how many nodes in each hidden layer. To make things simple we will have one hidden layer of five nodes.



The code for a multiple output neural network is similar the single output neural network. A multiple output neutral network is similar to the stochastic Logistic Regression classifiers that we studied previously

Here are our constants

```
#constants
num_iterations = 50000
learning_rate = .1
tolerence = .01
```

Here we set the number of inputs (3*3 grid) to the number of outputs (6) and we set the hidden layer to number of outputs – 1.

```
# sizes (row, middle, columns)
num_inputs =  X.shape[2] # 9
num_hiddens = Y.shape[2]-1 # 5
num_outputs = Y.shape[2] # 6
num_test_cases = X.shape[0] # 6
```

This is where we initialize out weights and bias with random numbers

```
# hidden layer random weights
Wh = np.random.random((num_inputs, num_hiddens))

# hidden layer random bias
Bh = np.random.random((num_hiddens))

# output layer random weights
Wo = np.random.random((num_hiddens,num_outputs))

# output layer random bias
Bo = np.random.random((num_outputs))
```

For the forward propagation hidden nodes we use the previous sigmoid function.

```
# sigmoid function
def sigmoid(z):
     return 1 / (1 + np.exp(-z))
```

For back propagation we also use the following derivative sigmoid function:

```
# sigmoid derivative function
def sigmoid_derivative(z):
      return sigmoid(z) * (1-sigmoid(z))
```

For the forward propagation output nodes we use the softmax(A) function

```
# softmax function
def softmax(A):
     expA = np.exp(A)
     return expA / expA.sum()
```

$$\sigma(\vec{z})_i = \frac{e^{z_i}}{\sum_{j=1}^{K} e^{z_j}}$$

σ = softmax

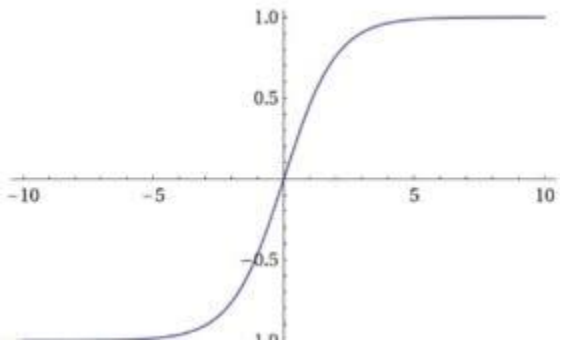
$\vec{z}$ = input vector

e^{z_i} = standard exponential function for input vector

K = number of classes in the multi-class classifier

e^{z_j} = standard exponential function for output vector

e^{z_j} = standard exponential function for output vector



The softmax function also called the normalized exponential function is used as the last activation function of a neural network to normalize the output of a network to a probability distribution over predicted output classes.

Feedforward propagation

Our forward propagation code is very similar to the previous forward propagation except we use the softmax function for the output layer neural network

```
# forward_propagation to calculate layer outputs
# returns outputs
def forward_propagation(x, y, wh,bh,wo,bo):

    # calculate hidden layer output

    zh = np.dot(x,wh) + bh
    ah = sigmoid(zh)

    # calculate output layer output
    zo = np.dot(ah,wo) + bo
    ao = softmax(zo)

    return zh, ah, zo, ao
```

Backpropagation

The back propagation is also similar to the previous back propagation , the only difference we must sum up the bias values since we have multiple output errors.

Since each output is sent to the appropriate weights we not need to sum the output errors.

```
# backward propagation to adjust weights using gradient descent
# returns output error
def backward_propagation(x, y, wh,zh,ah,bh,zo,ao,wo,bo):

    # calculate error = desired - actual
    error = y-ao;

    d_wo = np.dot(ah.T,error)
    d_bo = error

    dah = np.dot(error,wo.T)

    dzh = sigmoid_derivative(zh)
    d_wh = np.dot(x.T,dzh*dah)
    d_bh = dah * dzh;

    # adjust weights of output layer
    wo += learning_rate * d_wo

     # adjust hidden bias
    bo += learning_rate *  d_bo.sum()


    # adjust weights of hidden layer
    wh += learning_rate * d_wh

    # adjust hidden bias
    bh += learning_rate *  d_bh.sum()

    #print(error)

    return error # return output error
```

Training

We iterate until tolerance is met or max iterations are met.

```
# train neural network
print("training Neural Network")

total_error = 0;

for i in range(num_iterations+1):

   error = 0

   # for each input case
   for j in range(num_test_cases):

      # do forward propagation
      Zh, Ah, Zo, Ao = forward_propagation(X[j], Y[j], Wh,Bh,Wo,Bo)

      # do backward propagation
      error += backward_propagation(X[j], Y[j], Wh,Zh,Ah,Bh,Zo,Ao,Wo,Bo)

   # early termination if in tolerence
   sum_error = np.sum(abs(error))
   total_error += sum_error
   if sum_error/len(error) < tolerence:
      break


# print out error and num iterations
print("total error: ",total_error,"number iterations: ",i)

# print out results
for j in range(num_test_cases):
   Zh, Ah, Zo, Ao = forward_propagation(X[j], Y[j], Wh,Bh,Wo,Bo)
   print(X[j],Y[j],":",Ao)
```

Here is the output:

We are quite accurate

Dice Digits Neural Network

training Neural Network

total error: 250.42979215921847 number iterations: 3008

```
[[0 0 0 0 1 0 0 0 0]] [[0 0 0 0 0 1]]
: [[3.59066444e-04 4.01999490e-03 1.78054936e-07 1.28377046e-03 2.83535187e-07 9.94336707e-01]]
[[1 0 0 0 0 0 0 0 1]] [[0 0 0 0 1 0]]
: [[1.69306167e-05 7.09493672e-05 5.58381524e-03 2.09478447e-03 9.92233487e-01 3.28245838e-08]]
[[1 0 0 0 1 0 0 0 1]] [[0 0 0 1 0 0]]
: [[2.87120985e-07 2.49706002e-03 1.01609363e-05 9.94018059e-01 2.25612396e-03 1.21830855e-03]]
[[1 0 1 0 0 0 1 0 1]] [[0 0 1 0 0 0]]
: [[5.41794705e-04 4.93125152e-03 9.90590070e-01 1.00408944e-05  3.92683751e-03 5.79255927e-09]]
[[1 0 1 0 1 0 1 0 1]] [[0 1 0 0 0 0]]
: [[1.61207789e-03 9.90967425e-01 2.90066801e-03 2.11029126e-03  2.78896940e-05 2.38164849e-03]]
[[1 1 1 0 0 0 1 1 1]] [[1 0 0 0 0 0]]
: [[9.96553211e-01 1.10593518e-03 2.33968709e-03 1.14235776e-09 2.73522814e-07 8.92274441e-07]]
```

Here is the complete code:

```
# dice_neural_network.py
import numpy as np

# print program title
print("Dice Digits Neural Network")

# dice patterns 1 to 6
X = np.array([[[0,0,0,0,1,0,0,0,0]],
              [[1,0,0,0,0,0,0,0,1]],
              [[1,0,0,0,1,0,0,0,1]],
              [[1,0,1,0,0,0,1,0,1]],
              [[1,0,1,0,1,0,1,0,1]],
              [[1,1,1,0,0,0,1,1,1]]])

# outputs
Y = np.array([[[0, 0, 0, 0, 0, 1]],
              [[0, 0, 0, 0, 1, 0]],
              [[0, 0, 0, 1, 0, 0]],
              [[0, 0, 1, 0, 0, 0]],
              [[0, 1, 0, 0, 0, 0]],
              [[1, 0, 0, 0, 0, 0]]])
```

```
# constants
num_iterations = 50000
learning_rate = .1
tolerence = .01

# sizes (row, middle, columns)
num_inputs =  X.shape[2] # 9
num_hiddens = Y.shape[2]-1 # 5
num_outputs = Y.shape[2] # 6
num_test_cases = X.shape[0] # 6

# hidden layer random weights
Wh = np.random.random((num_inputs, num_hiddens))

# hidden layer random bias
Bh = np.random.random((num_hiddens))

# output layer random weights
Wo = np.random.random((num_hiddens,num_outputs))

# output layer random bias
Bo = np.random.random((num_outputs))

# sigmoid function
def sigmoid(z):
    return 1 / (1 + np.exp(-z))

# sigmoid derivative function
def sigmoid_derivative(z):
     return sigmoid(z) * (1-sigmoid(z))

# softmax function
def softmax(A):
    expA = np.exp(A)
    return expA / expA.sum()

# forward_propagation to calculate layer outputs
# returns outputs
def forward_propagation(x, y, wh,bh,wo,bo):

   # calculate hidden layer output
   zh = np.dot(x,wh) + bh
   ah = sigmoid(zh)
```

```
    # calculate output layer output
    zo = np.dot(ah,wo) + bo
    ao = softmax(zo)

    return zh, ah, zo, ao

# backward propagation to adjust weights using gradient descent
# returns output error

def backward_propagation(x, y, wh,zh,ah,bh,zo,ao,wo,bo):
    # calculate error = desired - actual
    error = y-ao;

    d_wo = np.dot(ah.T,error)
    d_bo = error

    dah = np.dot(error,wo.T)

    dzh = sigmoid_derivative(zh)
    d_wh = np.dot(x.T,dzh*dah)
    d_bh = dah * dzh;

    # adjust weights of output layer
    wo += learning_rate * d_wo

     # adjust hidden bias
    bo += learning_rate *  d_bo.sum()

    # adjust weights of hidden layer
    wh += learning_rate * d_wh

    # adjust hidden bias
    bh += learning_rate *  d_bh.sum()

    #print(error)
    return error # return output error

# train neural network
print("training Neural Network")

total_error = 0;
```

```
for i in range(num_iterations+1):

   error = 0

   # for each input case
   for j in range(num_test_cases):

      # do forward propagation
      Zh, Ah, Zo, Ao = forward_propagation(X[j], Y[j], Wh,Bh,Wo,Bo)

      # do backward propagation
      error += backward_propagation(X[j], Y[j], Wh,Zh,Ah,Bh,Zo,Ao,Wo,Bo)

   # early termination if in tolerence
   sum_error = np.sum(abs(error))
   total_error += sum_error
   if sum_error/len(error) < tolerence:
      break


# print out error and num iterations
print("total error: ",total_error,"number iterations: ",i)

# print out results
for j in range(num_test_cases):
   Zh, Ah, Zo, Ao = forward_propagation(X[j], Y[j], Wh,Bh,Wo,Bo)
   print(X[j],Y[j],":",Ao)
```

to do:

Type in or copy and paste the above code and get it running.

Homework Question 2

Make a neural network with 15 inputs (5*3) grid to represent digit shapes 0 to 9 and 10 outputs and to classify digits 0 to 9.

You can make a digit 4 like this.

1		1
1		1
1	1	1
		1
		1

The inputs would be represented by:

1	0	1	1	0	1	1	1	1	0	0	1	0	0	1

The output would be (indicating the digit 4))

0	1	2	3	4	5	6	7	8	9
0	0	0	0	1	0	0	0	0	0

Where width is 3 and height is 5

Call your py file neurualnetworkhomework2.py

END