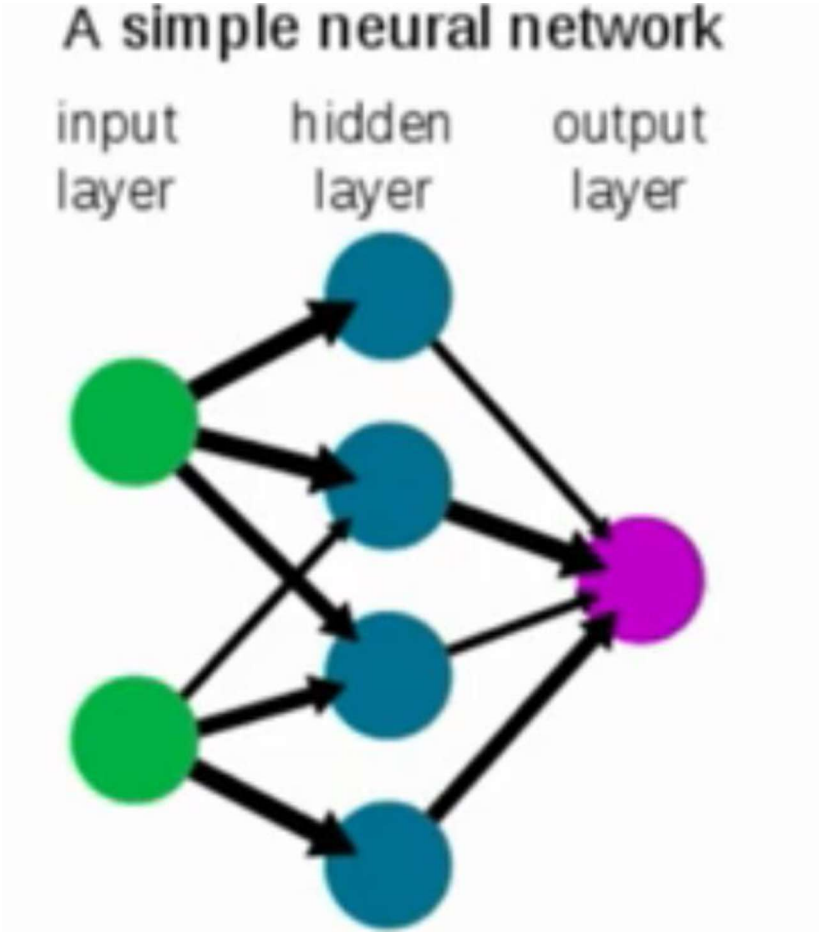
What is Backpropagation?

Backpropagation is a supervised learning algorithm, for training a neural network.



Backpropgation

The backpropagation algorithm consists of two parts:

- forward pass
- backward pass

In the forward pass, the expect output corresponding to the given inputs are evaluated

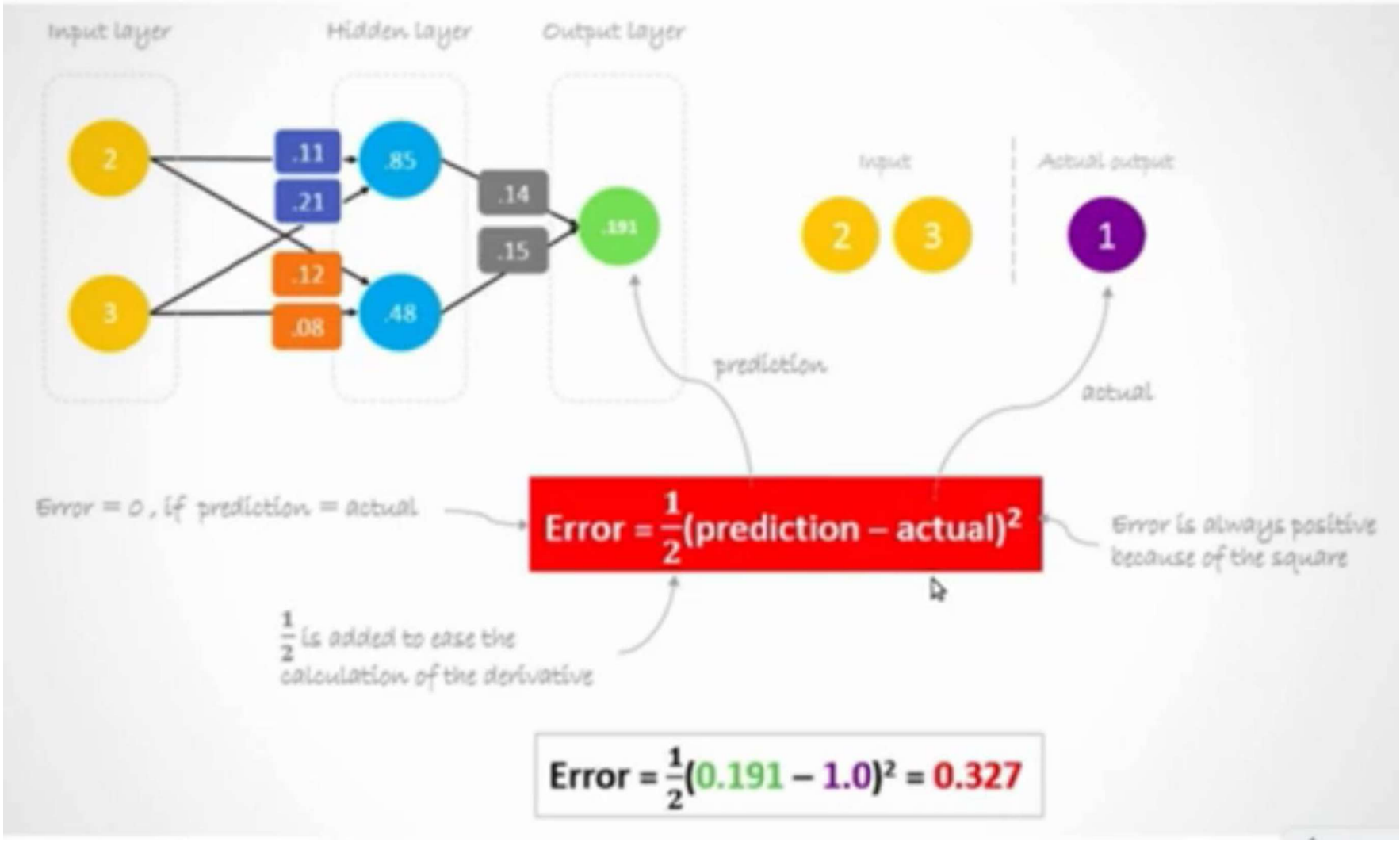
In the backward pass, partial derivatives of the cost function with respects to the different parameters are propagated back through the network.

The process continues until the **error is at the lowest value**.

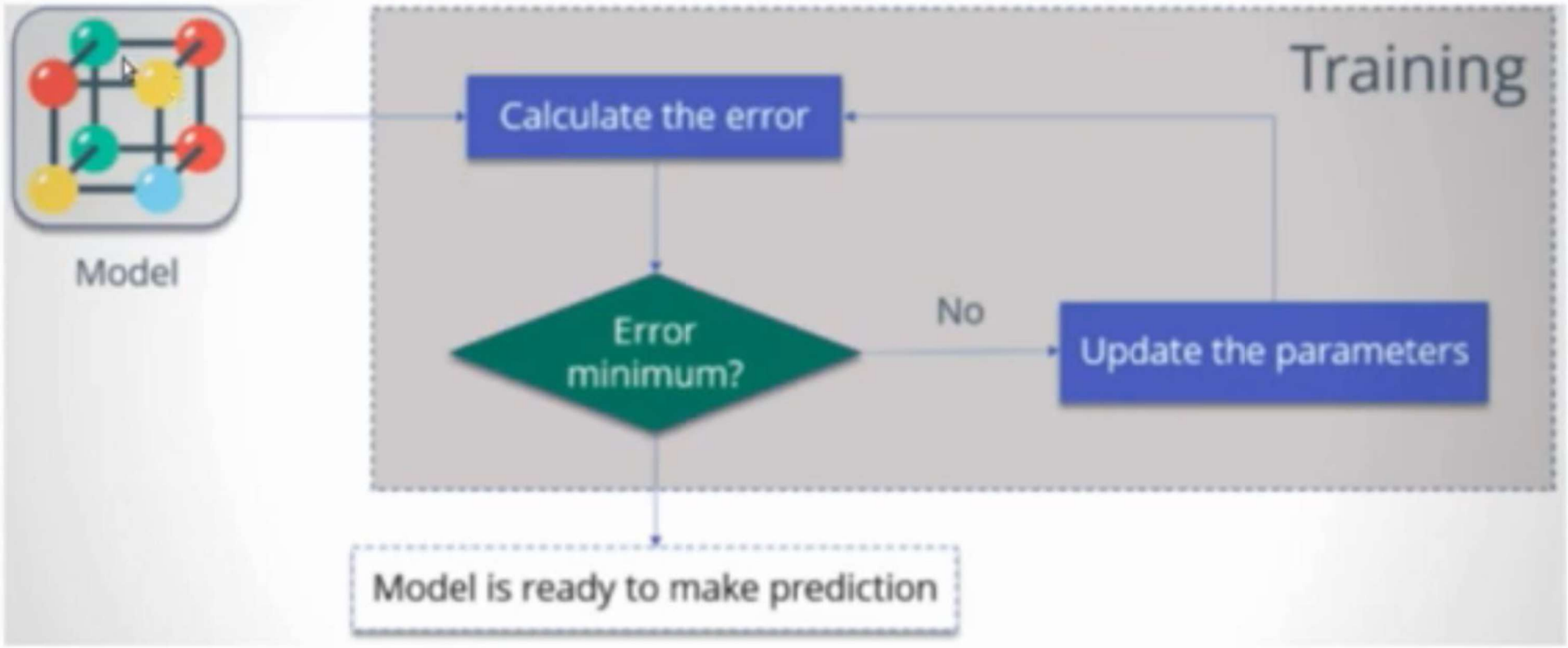
Backpropagation Steps

- Step 1: Determine the architecture
  - how many input and output neurons; what output encoding ?
  - hidden neurons and layers.
- Step 2: Initialize all weights and biases to small random values, typically $\in [-1,1]$, choose a learning rate η.
- Step 3: Repeat until termination criteria satisfied
  - Present a training example and propagate it through the network (forward pass)
  - Calculate the actual output
    - Inputs applied
    - Multiplied by weights
    - Summed
    - 'Squashed' by sigmoid activation function
    - Output passed to each neuron in next layer
  - Adapt weights starting from the output layer and working backwards (backward pass)

Why we need Backpropagation?



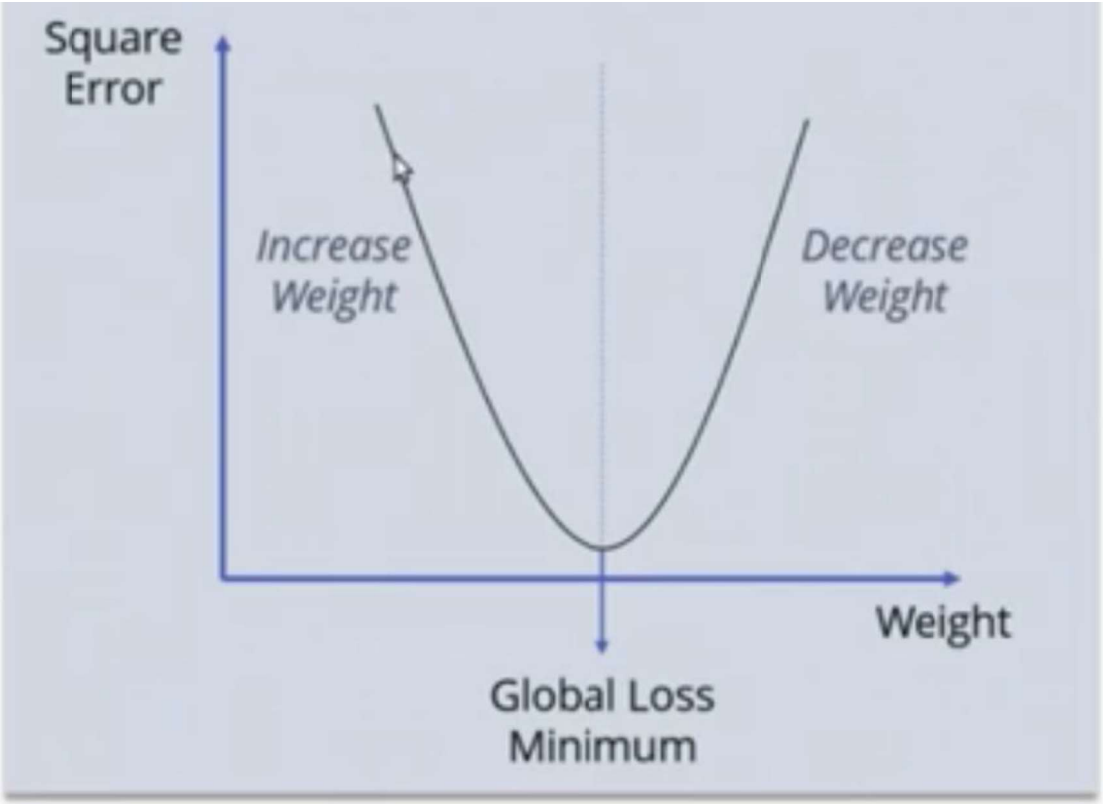
OVERVIEW:


Training
Calculate the error
Model
Error minimum?
No
Update the parameters
Model is ready to make prediction

Gradient Descent

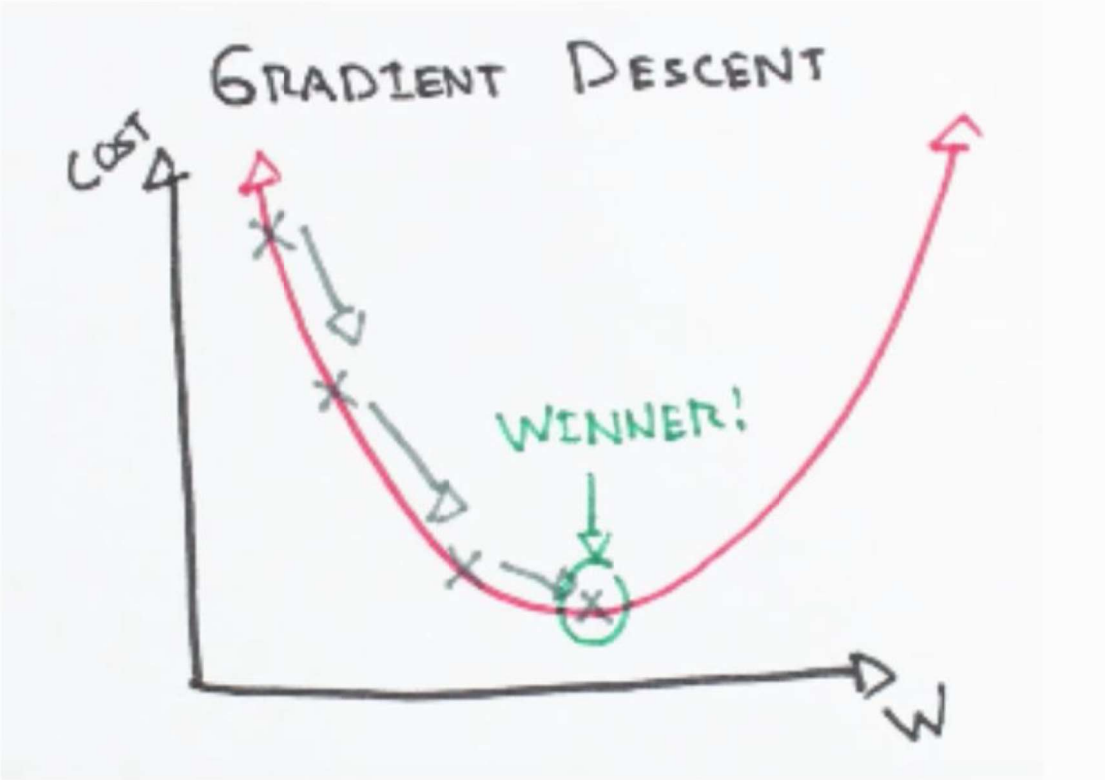
Gradient Descent is known as one of the most commonly used optimization algorithms to train machine learning models by means of minimizing errors between actual and expected results. Further, gradient descent is also used to train Neural Networks.

- Update the weights using gradient descent.
- Gradient descent is used for finding the minimum of a function.
- In our case we want to minimize the error function.



Gradient Descent

Gradient descent is an optimization algorithm used to minimize some function by iteratively moving in the direction of steepest descent as defined by the negative of the gradient. In machine learning, we use gradient descent to update the parameters of our model. Parameters refer to coefficients in Linear Regression and weights in neural networks.

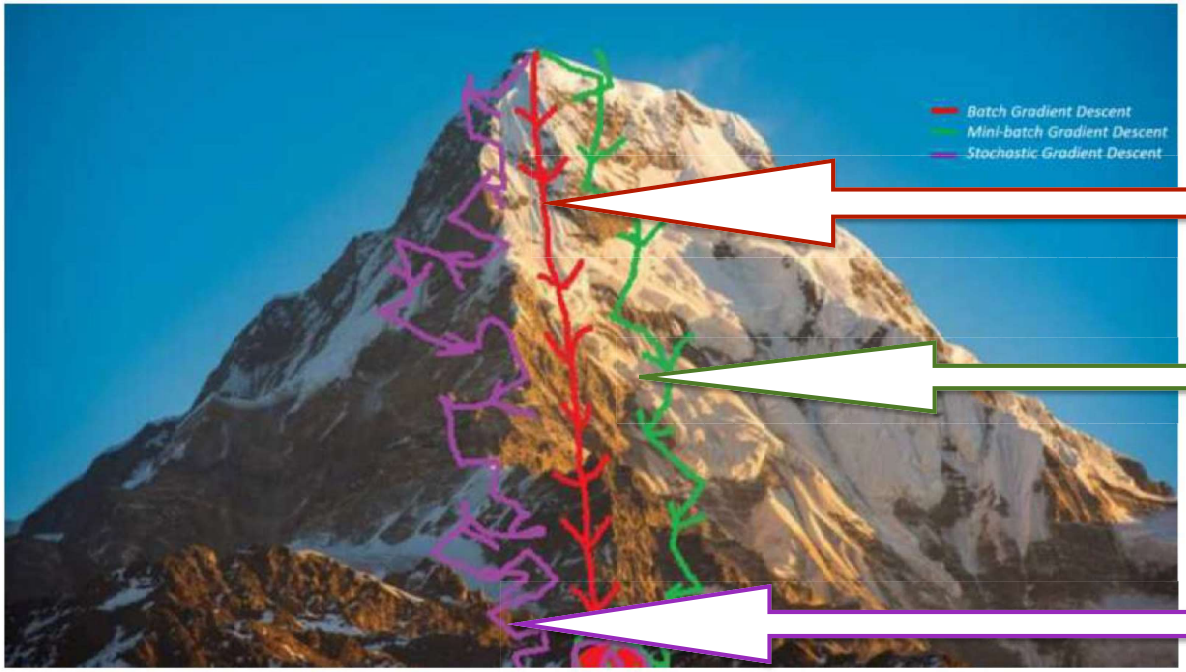


GD optimiser, not used in Deep Learning (for large dataset)

- GD means considering all the data points from the dataset.[weights get converge quickly].
- **Disadvantage:** Suppose data points is more than million, that point it will need more time to load it and computational power will be more, more number of resources required more. (because of million record).
- To overcome this, SGD used (more specifically mini batch SGD-taken k- data points). Here, try to select batch, batch may be 100/200 data points depends.
- In SGD, it tries to update weight for each and every data points.
- So, takes more time to reach global minima. To remove noise in SGD—momentum is used.

Gradient Descent

MACHINE LEARNING



n data point

Batch Gradient Descent

K data point

Mini Batch Gradient Descent

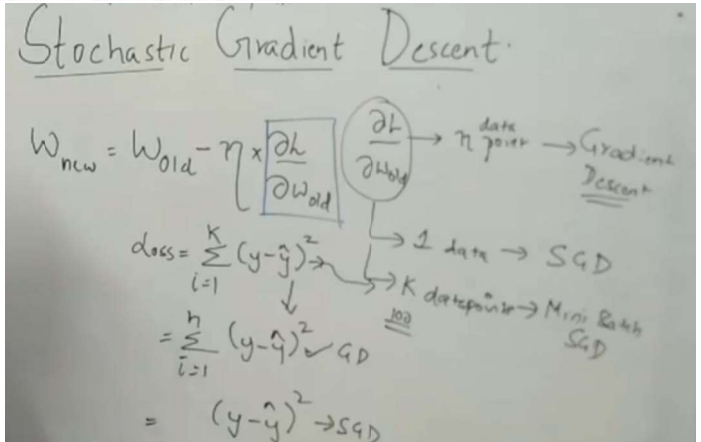
1 data point

Stochastic Gradient Descent

SGD: Update the weights for each and every datapoint.

Gradient descent is an iterative optimization algorithm for finding the minimum of a function; in our case we want to minimize the error function.

- Batch gradient descent refers to calculating the derivative from all training data before calculating an update.
- Stochastic gradient descent refers to calculating the derivative from each training data instance and calculating the update immediately.



derivative of loss w.r.t derivative of wt

if ∃ an finding derivative Slope for all data points.

$\frac{\partial L}{\partial w}$

- n Data Points → Gradient Descent ✓
- 1 D.P → S.G.D ✓
- K. Data Points (where $K<n$) → mini Batch SGD ✓

Here, Slope for Single data point

if only for K-points ($K<n$) then mini batch

$$\text{Loss} = \sum_{i=1}^{(K<n)} (y - \hat{y}_i)^2 \rightarrow \text{Mini Batch S.G.D}$$

$$= \left(\sum_{i=1}^{n}\right) (y_i - \hat{y}_i)^2 \rightarrow \text{G.D}$$

$$= (y_i - \hat{y}_i)^2 \rightarrow \text{S.G.D.}$$

Differentiation Rules

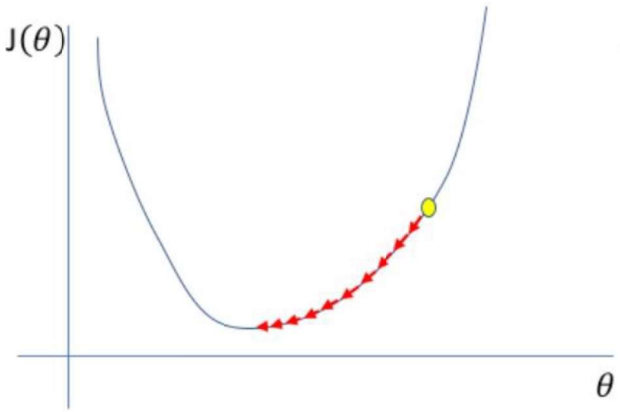
Constant Rule	$\frac{d}{dx}[c] = 0$
Power Rule	$\frac{d}{dx}x^n = nx^{n-1}$
Product Rule	$\frac{d}{dx}[f(x)g(x)] = f'(x)g(x) + f(x)g'(x)$
Quotient Rule	$\frac{d}{dx}\left[\frac{f(x)}{g(x)}\right] = \frac{g(x)f'(x) - f(x)g'(x)}{[g(x)]^2}$
Chain Rule	$\frac{d}{dx}[f(g(x))] = f'(g(x))g'(x)$

Learning Rate

- The amount that the weights are updated during training is referred to as the step size or the "*learning rate*."
- Specifically, the learning rate is a configurable hyperparameter used in the training of neural networks that has a small positive value, often in the range between 0.0 and 1.0.
- *… learning rate, a positive scalar determining the size of the step.*
- The learning rate is often represented using the notation of the lowercase Greek letter eta (*n*).
- During training, the backpropagation of error estimates the amount of error for which the weights of a node in the network are responsible. Instead of updating the weight with the full amount, it is scaled by the learning rate.
- This means that a learning rate of 0.1, a traditionally common default value, would mean that weights in the network are updated 0.1 * (estimated weight error) or 10% of the estimated weight error each time the weights are updated.

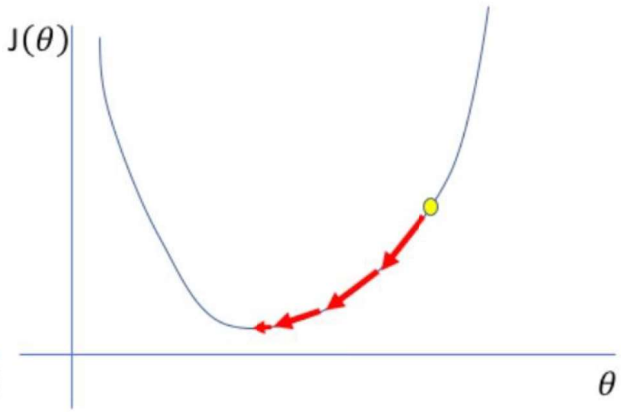
When the learning rate is too large, gradient descent can inadvertently increase rather than decrease the training error. […] When the learning rate is too small, training is not only slower, but may become permanently stuck with a high training error.

Too low



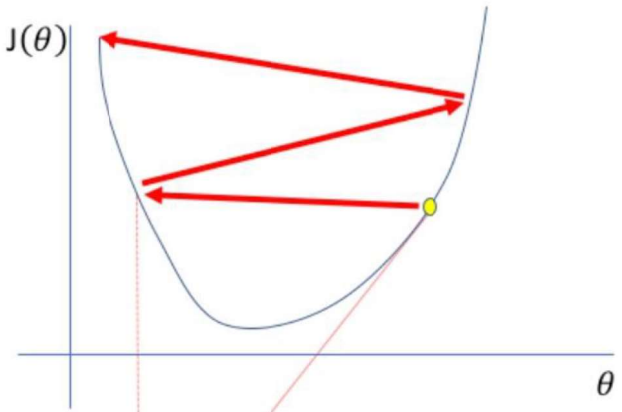
A small learning rate requires many updates before reaching the minimum point

Just right



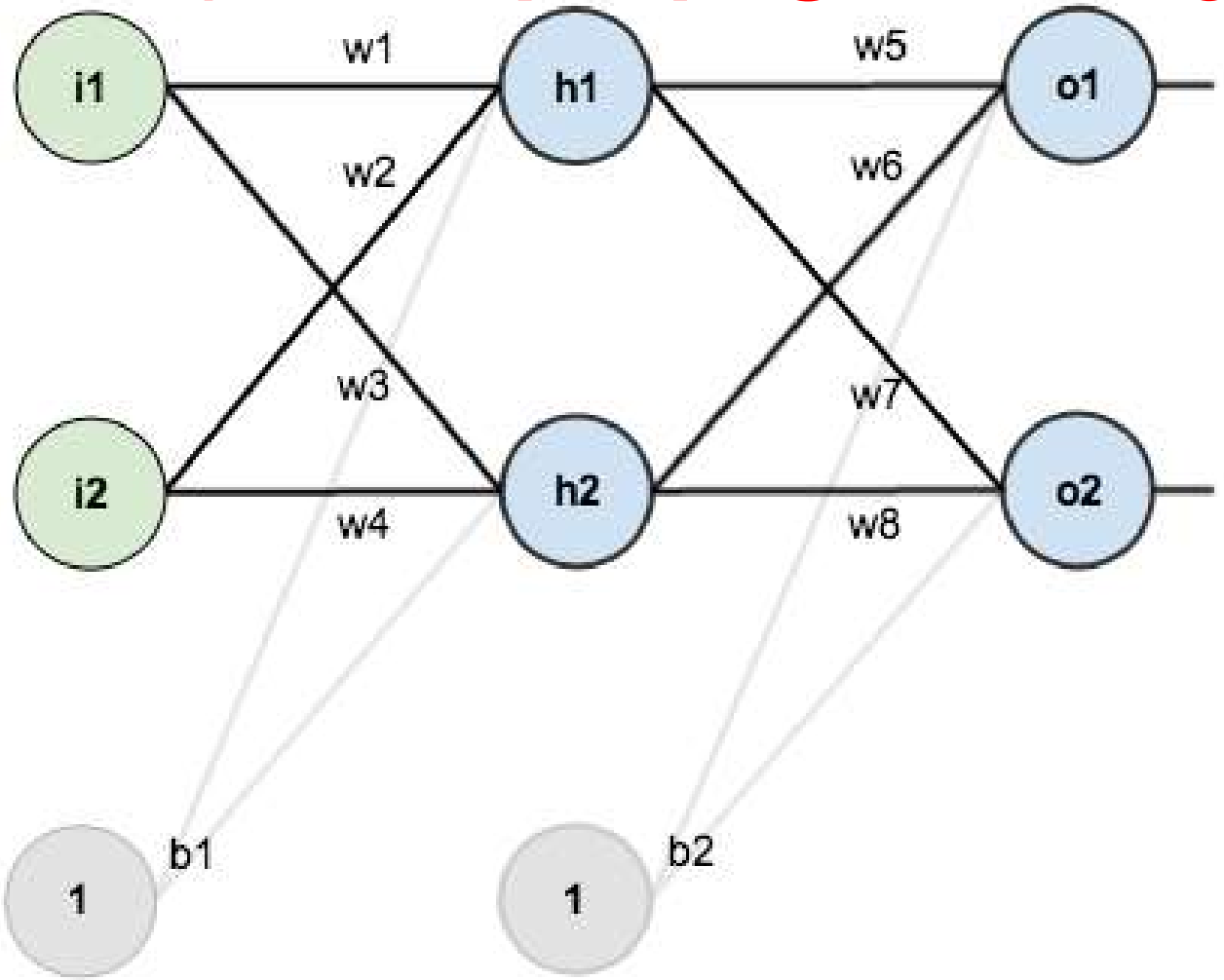
The optimal learning rate swiftly reaches the minimum point

Too high

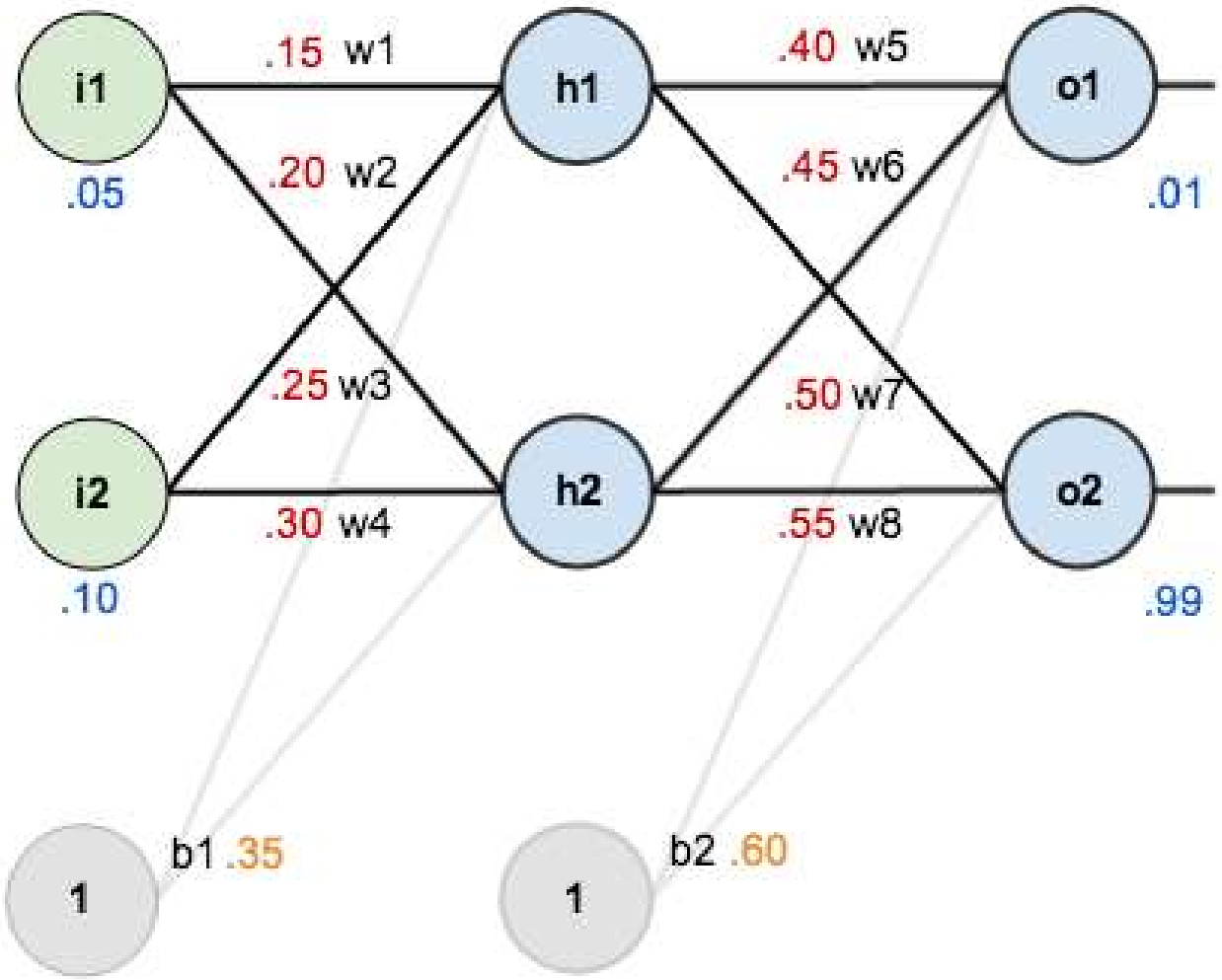


Too large of a learning rate causes drastic updates which lead to divergent behaviors

Step-by-Step **Backpropagation** Algorithm



The goal of backpropagation is to optimize the weights so that the neural network can learn how to correctly map arbitrary inputs to outputs.
For the rest of this tutorial we're going to work with a single training set:
given inputs 0.05 and 0.10,
we want the neural network to output 0.01 and 0.99.



i1=0.05, i2=0.10

w1=0.15, w2=0.20

w3=0.25, w4=0.30

b1=0.3

w5=0.4. w6=0.45

w7=0.5, w8=0.55

b2=0.6

o1=0.01, o2=0.99

Forward Pass

To begin, lets see what the neural network currently predicts given the weights and biases above and inputs of 0.05 and 0.10. To do this we'll feed those inputs forward though the network.

$$net_{h1} = w_1 * i_1 + w_2 * i_2 + b_1 * 1$$

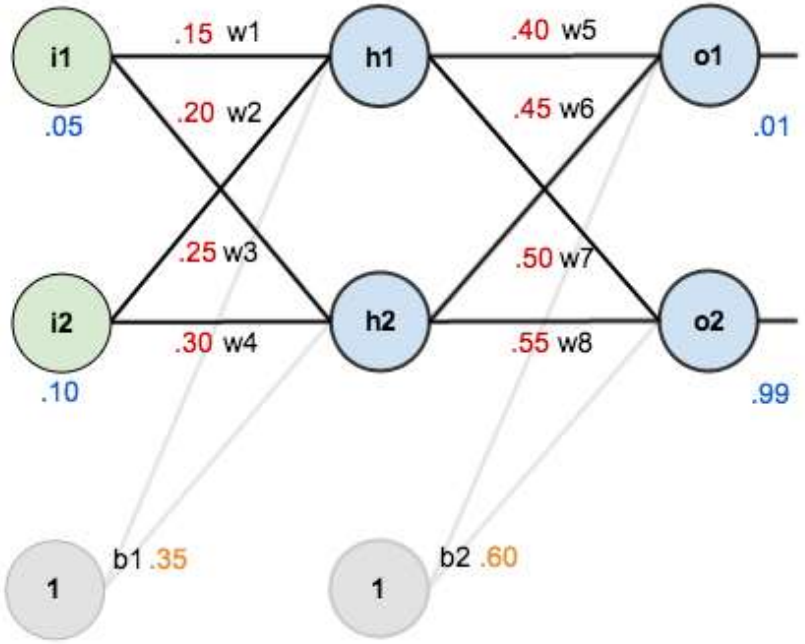
$$net_{h1} = 0.15 * 0.05 + 0.2 * 0.1 + 0.35 * 1 = 0.3775$$

Squash it using logistic function to get output

$$out_{h1} = \frac{1}{1+e^{-net_{h1}}} = \frac{1}{1+e^{-0.3775}} = 0.593269992$$

Same process for h_2

$$out_{h2} = 0.596884378$$



Forward Pass

Now calculating O_1

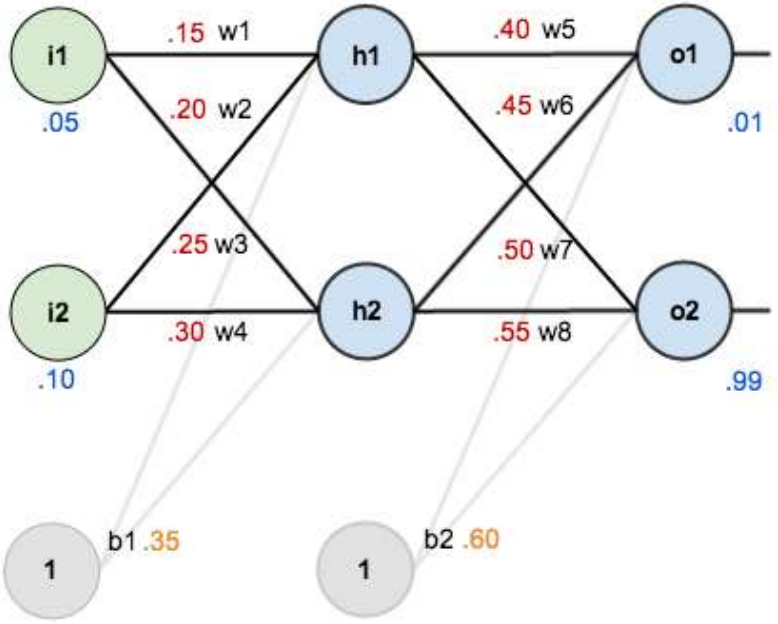
$$net_{o1} = w_5 * out_{h1} + w_6 * out_{h2} + b_2 * 1$$

$$net_{o1} = 0.4 * 0.593269992 + 0.45 * 0.596884378 + 0.6 * 1 = 1.105905967$$

$$out_{o1} = \frac{1}{1+e^{-net_{o1}}} = \frac{1}{1+e^{-1.105905967}} = 0.75136507$$

And carrying out the same process for o_2 we get:

$$out_{o2} = 0.772928465$$



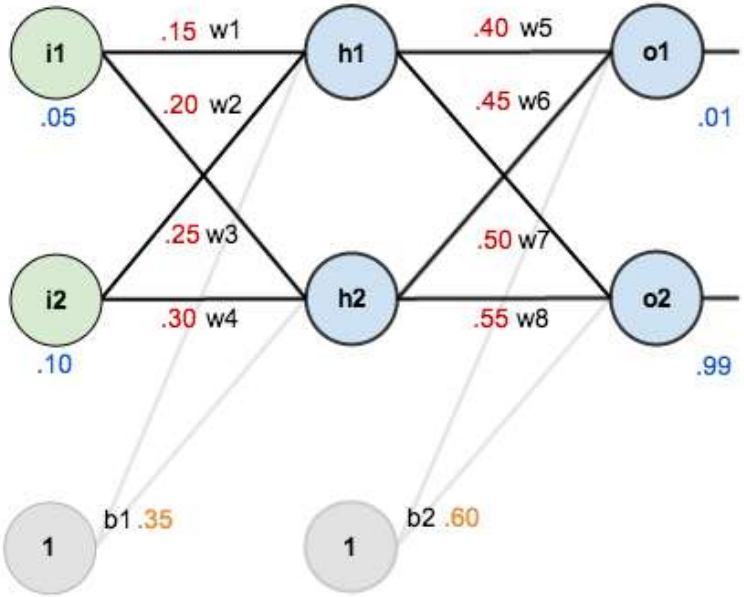
But, Target Values

T_1 0.01

T_2 0.99

Calculating Total Error

$$E_{total} = \sum \frac{1}{2}(target - output)^2$$

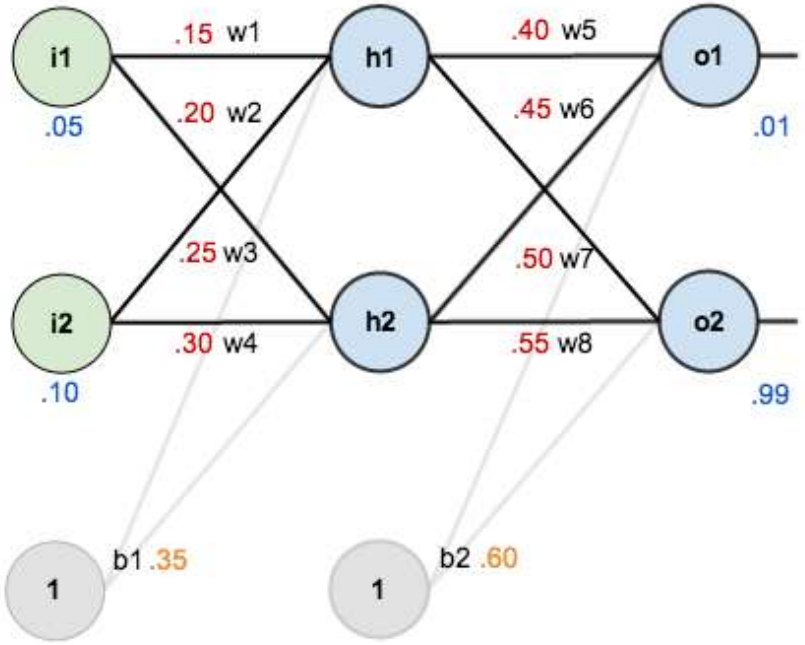


$$E_{o1} = \frac{1}{2}(target_{o1} - out_{o1})^2 = \frac{1}{2}(0.01 - 0.75136507)^2 = 0.274811083$$

$$E_{o2} = 0.023560026$$

$$E_{total} = E_{o1} + E_{o2} = 0.274811083 + 0.023560026 = 0.298371109$$

Backward Pass



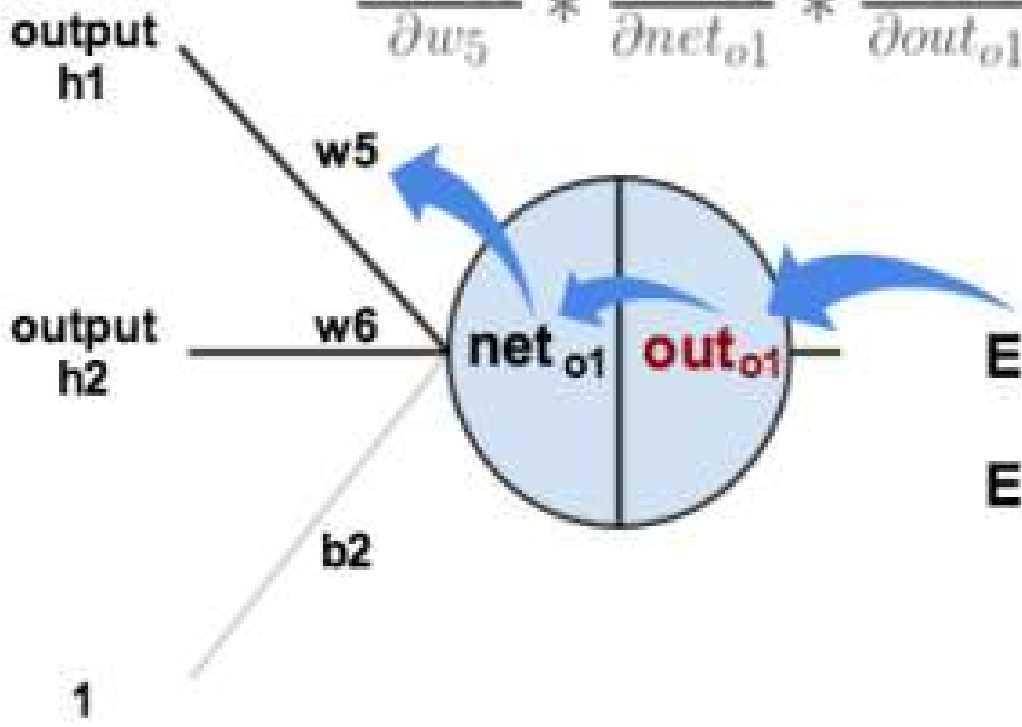
Our goal with backpropagation:

- **Is to update each of the weights in the network using gradient descent.**
- **So that they cause the actual output to be closer to the target output.**
- **Thereby minimizing the error for each output neuron and the network as a whole.**

Chain Rule

To calculate Error at w_5.

$$\frac{\partial net_{o1}}{\partial w_5} * \frac{\partial out_{o1}}{\partial net_{o1}} * \frac{\partial E_{total}}{\partial out_{o1}} = \frac{\partial E_{total}}{\partial w_5}$$



$$E_{o1} = \tfrac{1}{2}(target_{o1} - out_{o1})^2$$

$$E_{total} = E_{o1} + E_{o2}$$

Output Layer:

To update Weights:

OUTPUT LAYER:

Consider w5.

$\frac{\partial E_{total}}{\partial w_5}$ partial derivative of E_{total} wrt w5.

$$E_{total} = E_{o1} + E_{o2} \qquad E_{o1} = \frac{1}{2}(target_{o1} - out_{o1})^2$$

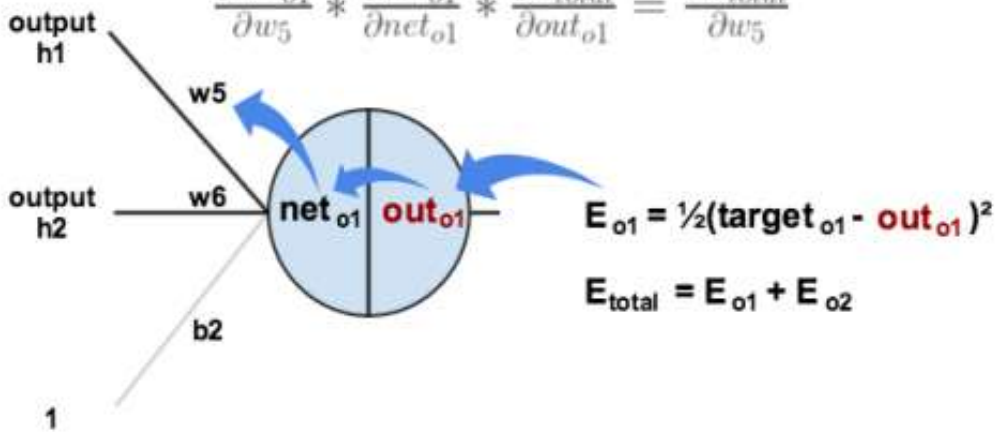
$$out_{o1} = \frac{1}{1+e^{-net_{o1}}}$$

$$net_{o1} = w_5 * out_{h1} + w_6 * out_{h2} + b_2 * 1$$

$$\frac{\partial E_{total}}{\partial w_5} = \frac{\partial E_{total}}{\partial out_{o1}} * \frac{\partial out_{o1}}{\partial net_{o1}} * \frac{\partial net_{o1}}{\partial w_5}$$

$$\delta_{o1} = \frac{\partial E_{total}}{\partial out_{o1}} * \frac{\partial out_{o1}}{\partial net_{o1}} = \frac{\partial E_{total}}{\partial net_{o1}}$$

$$\frac{\partial net_{o1}}{\partial w_5} * \frac{\partial out_{o1}}{\partial net_{o1}} * \frac{\partial E_{total}}{\partial out_{o1}} = \frac{\partial E_{total}}{\partial w_5}$$



$$\frac{\partial E_{total}}{\partial w_5} = \frac{\partial E_{total}}{\partial out_{o1}} * \frac{\partial out_{o1}}{\partial net_{o1}} * \frac{\partial net_{o1}}{\partial w_5}$$

Apply p.d on both side w.r.t w_5 / out_{o1}

$$E_{total} = \frac{1}{2}(target_{o1} - out_{o1})^2 + \frac{1}{2}(target_{o2} - out_{o2})^2$$

$$\frac{\partial E_{total}}{\partial out_{o1}} = 2 * \frac{1}{2}(target_{o1} - out_{o1})^{2-1} * -1 + 0$$

$$\frac{\partial E_{total}}{\partial out_{o1}} = -(target_{o1} - out_{o1}) = -(0.01 - 0.75136507) = 0.74136507$$

$$\frac{\partial E_{total}}{\partial w_5} = \frac{\partial E_{total}}{\partial out_{o1}} * \frac{\partial out_{o1}}{\partial net_{o1}} * \frac{\partial net_{o1}}{\partial w_5}$$

P.d

$$out_{o1} = \frac{1}{1+e^{-net_{o1}}} \Rightarrow \frac{\partial out_{o1}}{\partial net_{o1}} = out_{o1}(1 - out_{o1})$$

$$f(x) = \frac{1}{1+e^{-x}} = \frac{e^x}{1+e^x},$$

$$\frac{d}{dx} f(x) = \frac{e^x \cdot (1+e^x) - e^x \cdot e^x}{(1+e^x)^2} = \frac{e^x}{(1+e^x)^2} = f(x)(1 - f(x)).$$

$$\frac{\partial out_{o1}}{\partial net_{o1}} = 0.75136507(1 - 0.75136507) = 0.186815602$$

$$\frac{\partial E_{total}}{\partial w_5} = \frac{\partial E_{total}}{\partial out_{o1}} * \frac{\partial out_{o1}}{\partial net_{o1}} * \frac{\partial net_{o1}}{\partial w_5}$$

$$net_{o1} = w_5 * out_{h1} + w_6 * out_{h2} + b_2 * 1$$

$$\frac{\partial net_{o1}}{\partial w_5} = 1 * out_{h1} * w_5^{(1-1)} + 0 + 0 = out_{h1} = 0.593269992$$

putting it all together:

$$\frac{\partial E_{total}}{\partial w_5} = \frac{\partial E_{total}}{\partial out_{o1}} * \frac{\partial out_{o1}}{\partial net_{o1}} * \frac{\partial net_{o1}}{\partial w_5}$$

update w_5

$$\frac{\partial E_{total}}{\partial w_5} = 0.74136507 * 0.186815602 * 0.593269992 = 0.082167041$$

Change in w_5

- Update the weight.
- To decrease the error, we then subtract this value from the current weight.

$$w_5^+ = w_5 - \eta * \frac{\partial E_{total}}{\partial w_5} = 0.4 - 0.5 * 0.082167041 = 0.35891648$$

η = learning rate (eta)

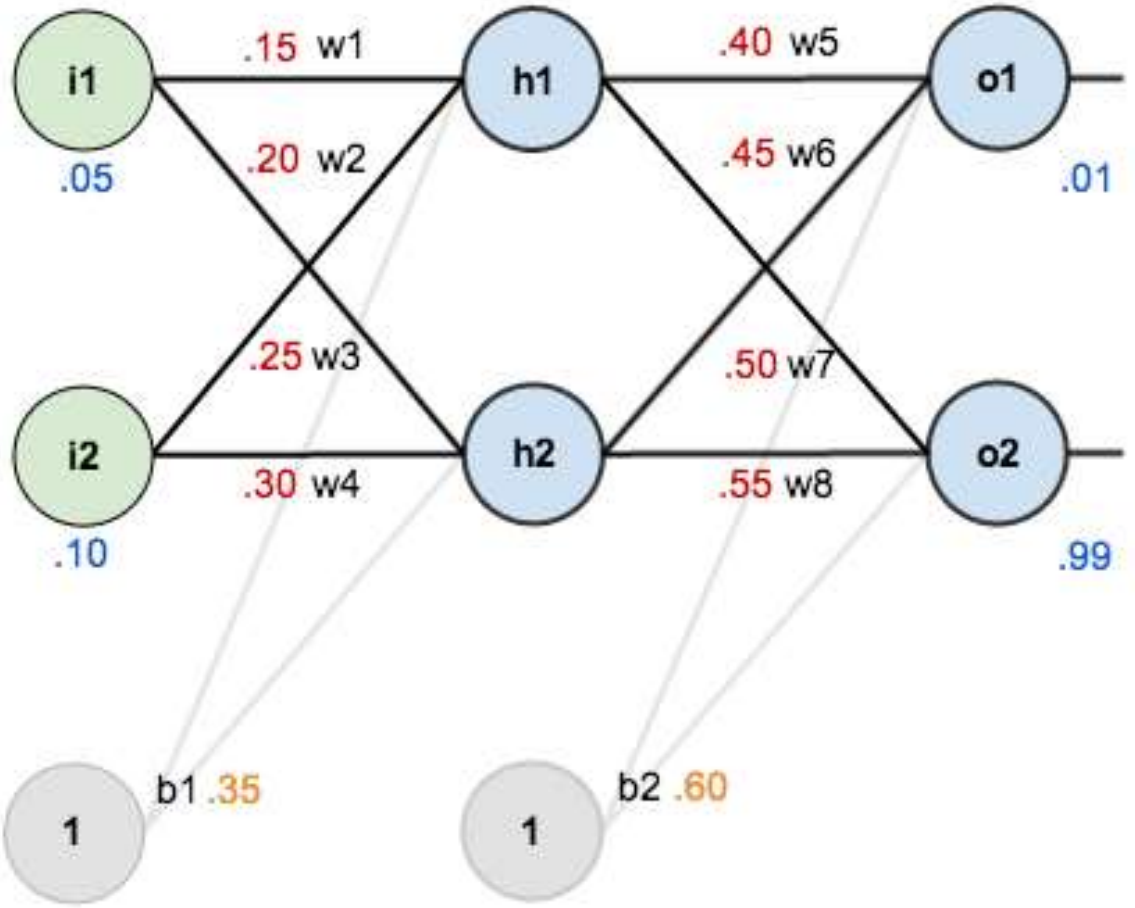
Same way :

$$w_6^+ = 0.408666186$$

$$w_7^+ = 0.511301270$$

$$w_8^+ = 0.561370121$$

We perform the actual updates in the neural network after we have the new weights leading into the hidden layer neurons (ie, we use the original weights, not the updated weights, when we continue the backpropagation algorithm below).



Hidden Layer:

Next, we'll continue the backwards pass by calculating new values for

w_1, w_2, w_3, w_4.

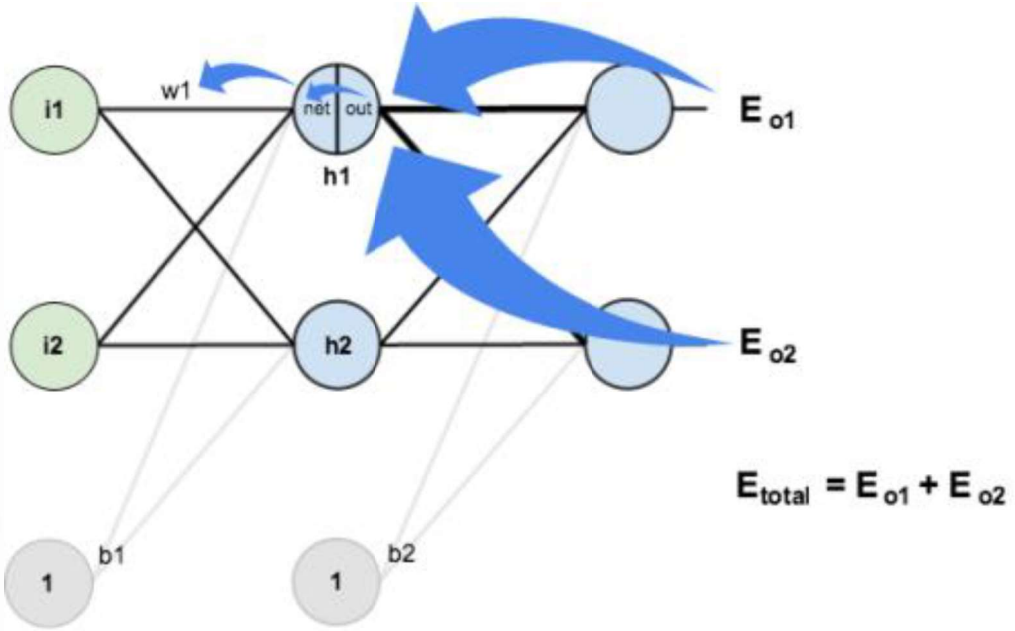
HIDDEN LAYER:

Consider w1.

updating wts

$$\frac{\partial E_{total}}{\partial w_1} = \frac{\partial E_{total}}{\partial out_{h1}} * \frac{\partial out_{h1}}{\partial net_{h1}} * \frac{\partial net_{h1}}{\partial w_1}$$

$$\frac{\partial E_{total}}{\partial out_{h1}} = \frac{\partial E_{o1}}{\partial out_{h1}} + \frac{\partial E_{o2}}{\partial out_{h1}}$$



W.K.T

$$E_{o1} = \frac{1}{2}(target_{o1} - out_{o1})^2$$

$$out_{o1} = \frac{1}{1+e^{-net_{o1}}}$$

$$net_{o1} = w_5 * out_{h1} + w_6 * out_{h2} + b_2 * 1$$

$$\frac{\partial E_{o1}}{\partial out_{h1}} = \frac{\partial E_{o1}}{\partial out_{o1}} * \frac{\partial out_{o1}}{\partial net_{o1}} * \frac{\partial net_{o1}}{\partial out_{h1}}$$

$$\frac{\partial net_{o1}}{\partial out_{h1}} = w_5$$

$$\frac{\partial E_{o1}}{\partial out_{h1}} = \frac{\partial E_{o1}}{\partial net_{o1}} * \frac{\partial net_{o1}}{\partial out_{h1}} = 0.138498562 * 0.40 = 0.055399425$$

$$\frac{\partial E_{o2}}{\partial out_{h1}} = -0.019049119$$

$$\frac{\partial E_{total}}{\partial out_{h1}} = \frac{\partial E_{o1}}{\partial out_{h1}} + \frac{\partial E_{o2}}{\partial out_{h1}} = 0.055399425 + -0.019049119 = 0.036350306$$

$$\frac{\partial E_{total}}{\partial w_1} = \frac{\partial E_{total}}{\partial out_{h1}} * \frac{\partial out_{h1}}{\partial net_{h1}} * \frac{\partial net_{h1}}{\partial w_1}$$

$$out_{h1} = \frac{1}{1+e^{-net_{h1}}}$$

$$\frac{\partial out_{h1}}{\partial net_{h1}} = out_{h1}(1 - out_{h1}) = 0.59326999(1 - 0.59326999) = 0.241300709$$

$$net_{h1} = w_1 * i_1 + w_3 * i_2 + b_1 * 1$$

$$\frac{\partial net_{h1}}{\partial w_1} = i_1 = 0.05$$

Putting it all together

$$\frac{\partial E_{total}}{\partial w_1} = \frac{\partial E_{total}}{\partial out_{h1}} * \frac{\partial out_{h1}}{\partial net_{h1}} * \frac{\partial net_{h1}}{\partial w_1}$$

$$\frac{\partial E_{total}}{\partial w_1} = 0.036350306 * 0.241300709 * 0.05 = 0.000438568$$

We can now update w_1:

$$w_1^+ = w_1 - \eta * \frac{\partial E_{total}}{\partial w_1} = 0.15 - 0.5 * 0.000438568 = 0.149780716$$

Repeating this for w_2, w_3, and w_4

$$w_2^+ = 0.19956143$$

$$w_3^+ = 0.24975114$$

$$w_4^+ = 0.29950229$$

- Finally, we've updated all of our weights!
- When we fed forward the 0.05 and 0.1 inputs originally, the error on the network was 0.298371109.
- After this first round of backpropagation, the total error is now down to 0.291027924.
- It might not seem like much, but after repeating this process 10,000 times, for example, the error plummets to 0.0000351085.

Finally, we've updated all of our weights! When we fed forward the 0.05 and 0.1 inputs originally, the error on the network was 0.298371109. After this first round of backpropagation, the total error is now down to 0.291027924. It might not seem like much, but after repeating this process 10,000 times, for example, the error plummets to 0.0000351085. At this point, when we feed forward 0.05 and 0.1, the two outputs neurons generate 0.015912196 (vs 0.01 target) and 0.984065734 (vs 0.99 target).

Backpropagation Algorithm

BACKPROPAGATION (training_example, η, n_{in}, n_{out}, n_{hidden})

- Each training example is a pair of the form (x, t), where (x) is the vector of network input values, and (t) is the vector of target network output values.
- η is the learning rate (e.g., 0.05).
- n_i, is the number of network inputs,
- n_{hidden} the number of units in the hidden layer, and
- n_{out} the number of output units.
- The input from unit i into unit j is denoted x_{ji}, and the weight from unit i to unit j is denoted w_{ji}

Backpropagation Algorithm

- Create a feed-forward network with n_i inputs, n_{hidden} hidden units, and n_{out} output units.
- Initialize all network weights to small random numbers
- Until the termination condition is met, Do
  - For each (x, t), in training examples, Do
    - Propagate the input forward through the network:

      1. Input the instance x, to the network and compute the output o_u of every unit u in the network.

    - Propagate the errors backward through the network

      2. For each network unit k, calculate its error term δ_k

$$\delta_k \leftarrow o_k(1 - o_k)(t_k - o_k)$$

      3. For each network unit h, calculate its error term δ_h

$$\delta_h \leftarrow o_h(1 - o_h) \sum_{k \in outputs} w_{h,k}\delta_k$$

4. Update each network weight w_{ji}

$$w_{ji} \leftarrow w_{ji} + \Delta w_{ji}$$

Where

$$\Delta w_{ji} = \eta \delta_j x_{ji}$$

BACKPROPOGATION Algorithm

- create a feed-forward network with n_{in} inputs. n_{hidden} hidden untis, and n_{out} output units
- Initialize all network weights to small random numbers
- Until the **termination condition** is met, Do
  - For each $< \vec{x}, \vec{t} >$ in $training\_examples$, Do

    Propagate the input forward through the network:

    1. Input $\vec{x}$ to the network and compute o_u of every unit u

    Propagate the errors back trough the network:

    2. For each network **output unit** k, calculate its error term δ_k

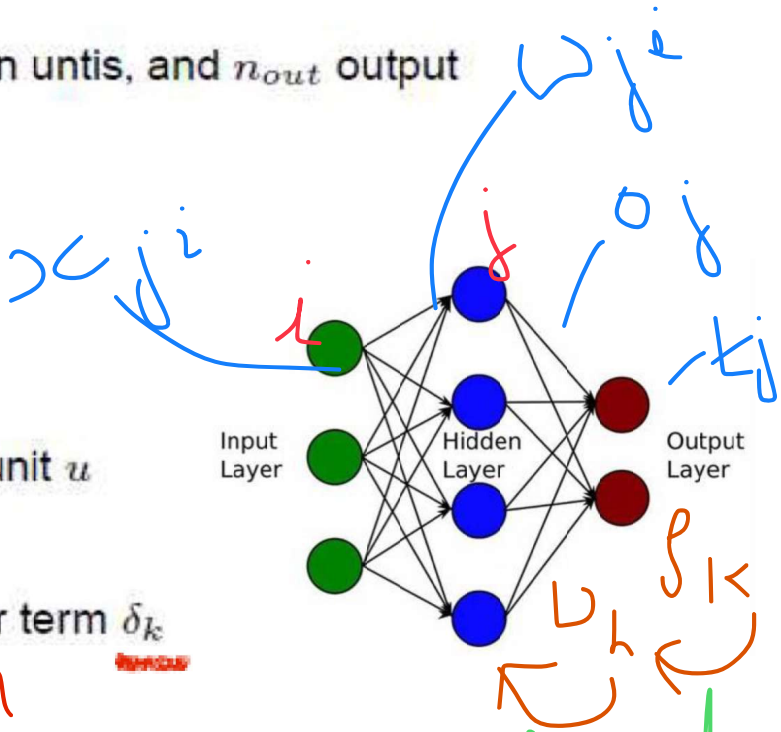
       $\delta_k \leftarrow o_k(1 - o_k)(t_k - o_k)$

    3. For each **hidden unit** h, calculate its error term δ_h

       $\delta_h \leftarrow o_h(1 - o_h) \sum_{k \in outputs} w_{kh} \delta_k$

    4. Update each weight w_{ji}

       $w_{ji} \leftarrow w_{ji} + \Delta w_{ji}$ where $\Delta w_{ji} = \eta \delta_j x_{ji}$



By Chain Rule

error

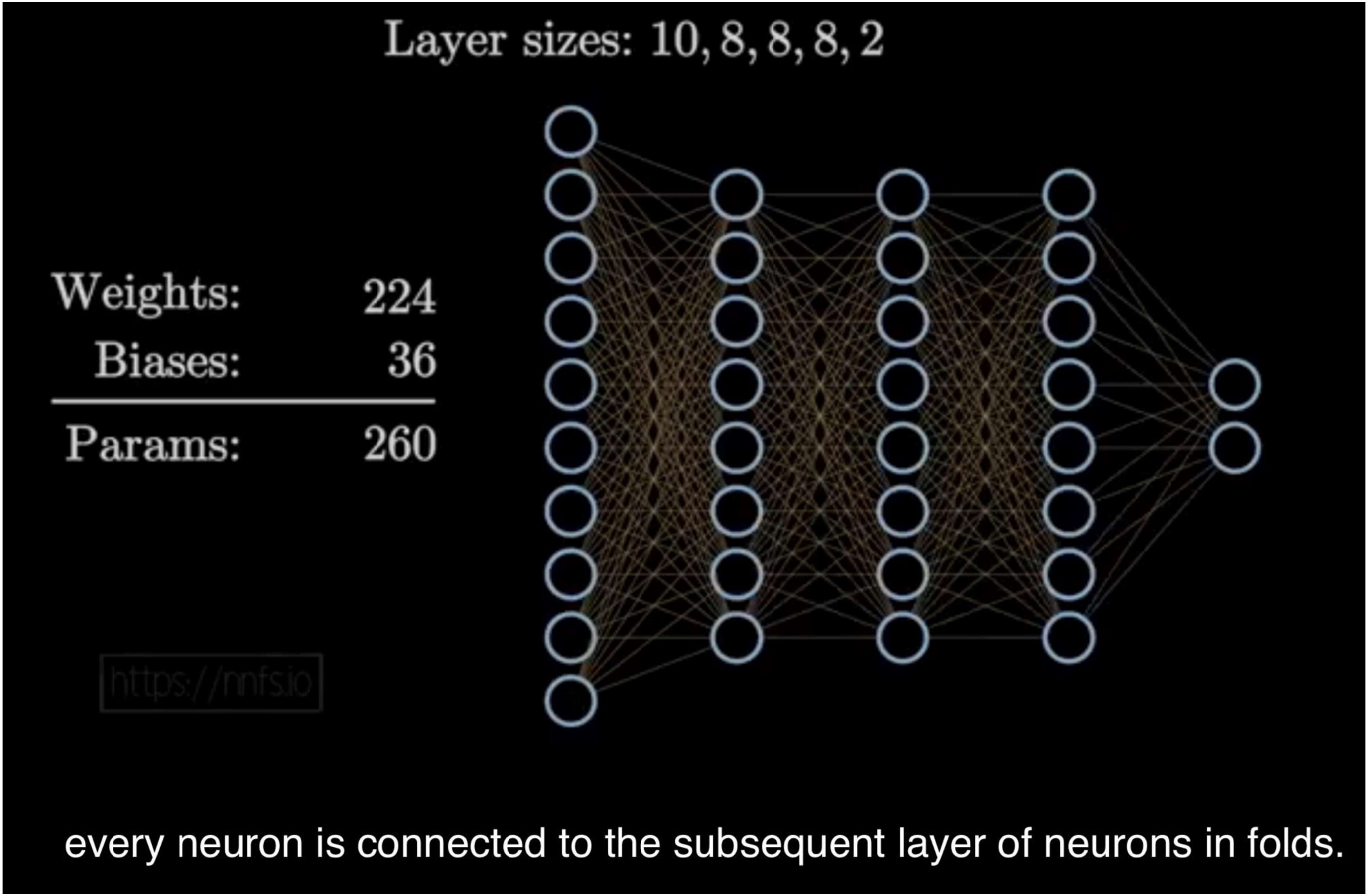
internal node

learning rate

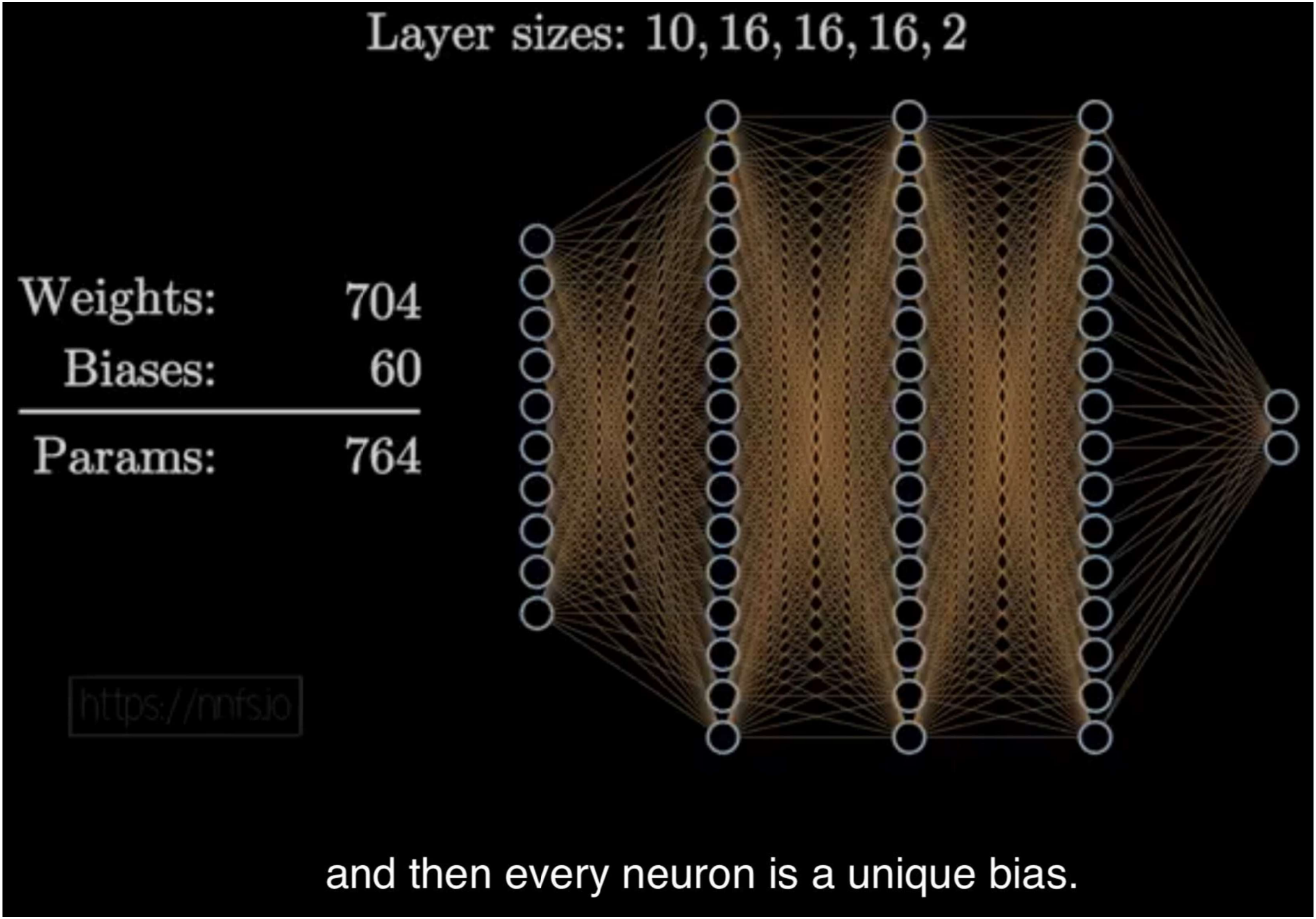
Input

From δ_K & δ_h

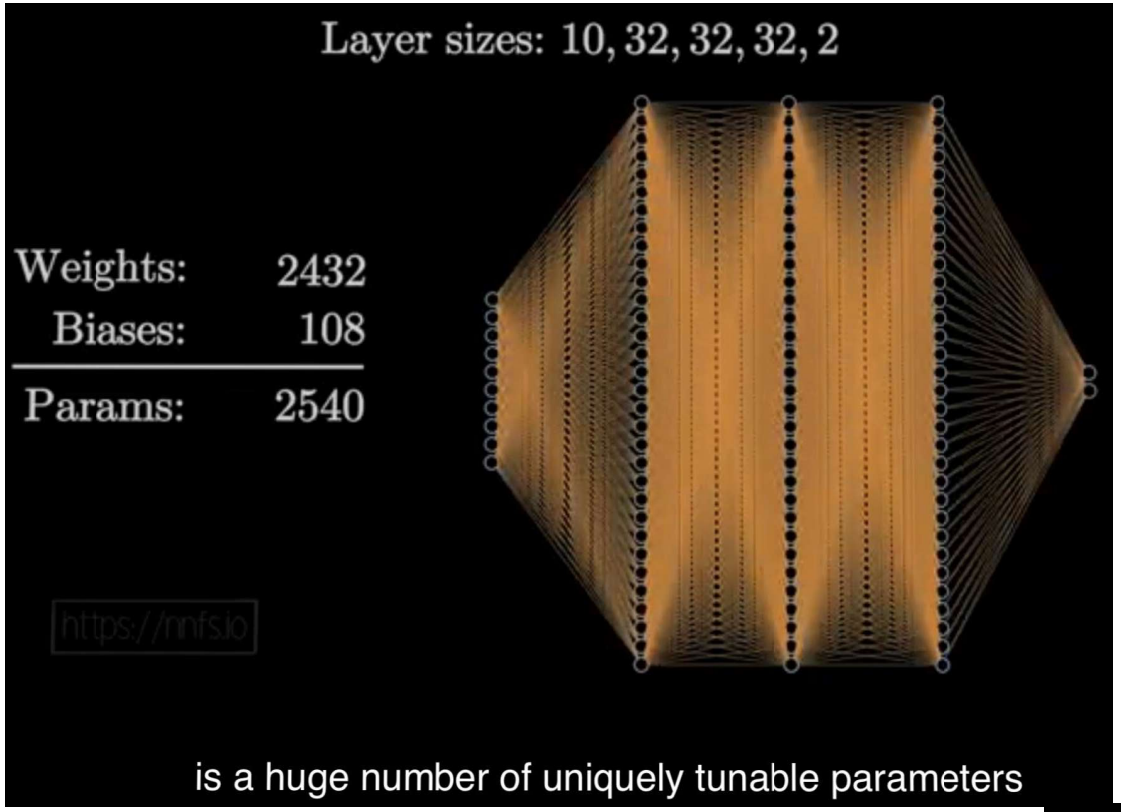
change in weight



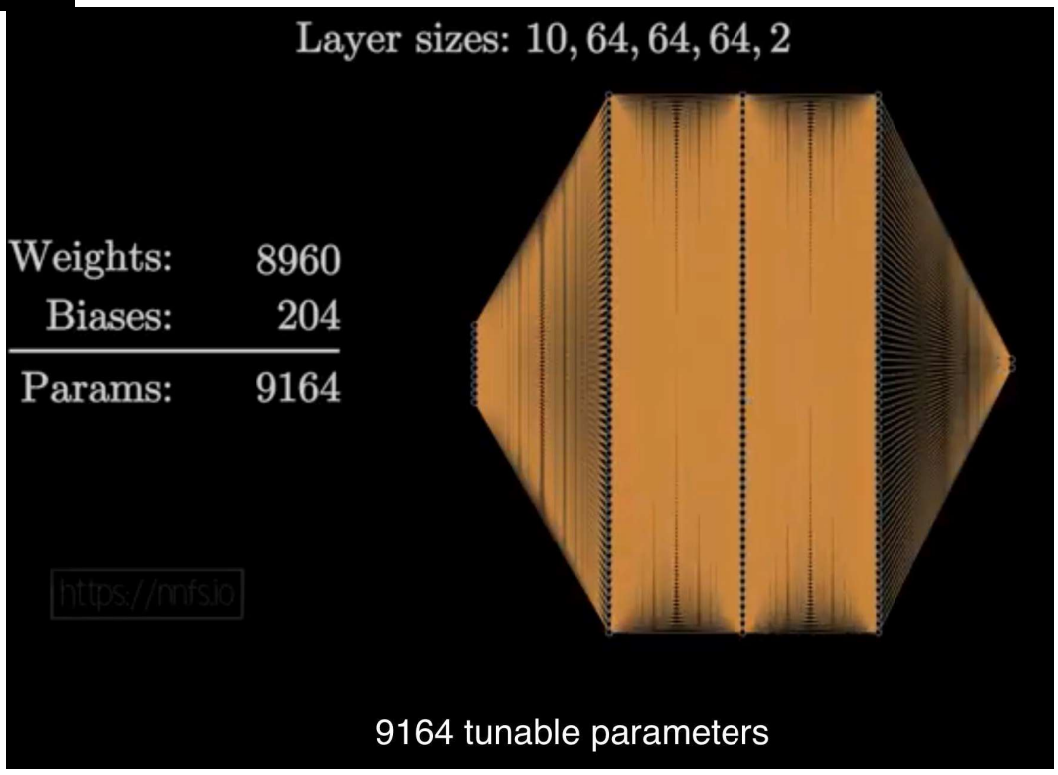
every neuron is connected to the subsequent layer of neurons in folds.



and then every neuron is a unique bias.



is a huge number of uniquely tunable parameters

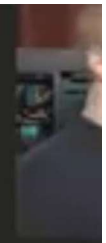


9164 tunable parameters

```python
import sys
import numpy as np
import matplotlib


print("Python:", sys.version)
print("Numpy:", np.__version__)
print("Matplotlib: ", matplotlib.__version__)
```

```
Python: 3.7.7 (default, Mar 10 2020, 15:16:38)
[GCC 7.5.0]
Numpy: 1.18.2
Matplotlib:  3.2.1
[Finished in 0.3s]
```

```python
inputs = [1.2, 5.1, 2.1]
weights = [3.1, 2.1, 8.7]
bias = 3



output = inputs[0]*weights[0] + inputs[1]*weights[1] + inputs[2]*weights[2] + bias
print(output)
```



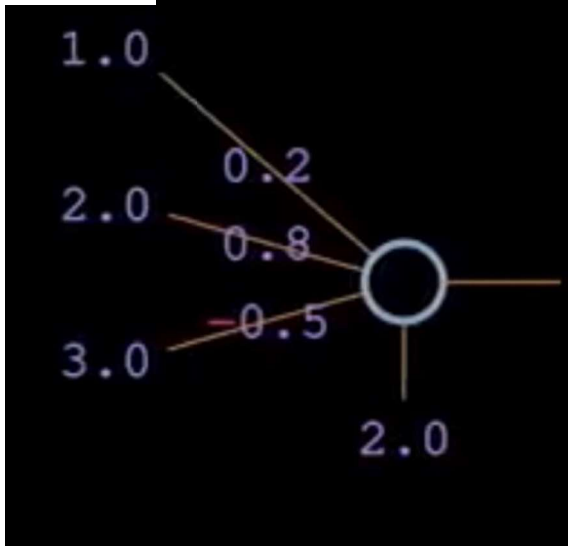
```
35.7
[Finished in 0.0s]
```

```python
inputs = [1, 2, 3]
weights = [0.2, 0.8, -0.5]
bias = 2



output = inputs[0]*weights[0] + inputs[1]*weights[1] + inputs[2]*weights[2] + bias
print(output)
```

```
2.3
[Finished in 0.0s]
```

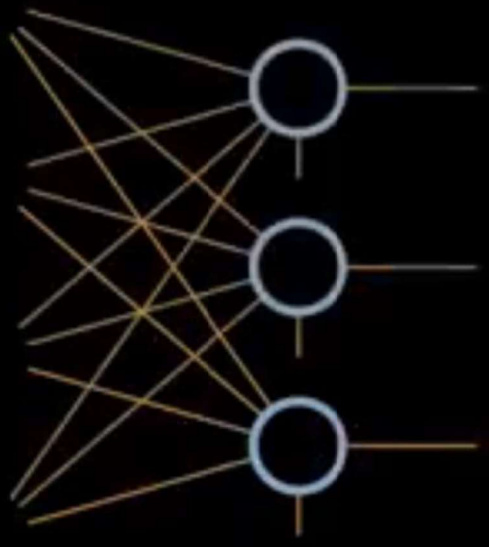
```python
inputs = [1.0, 2.0, 3.0]
weights = [0.2, 0.8, -0.5]
bias = 2.0

output = inputs[0]*weights[0] + inputs[1]*weights[1] + inputs[2]*weights[2] + bias
print(output)
```

```
>>> 2.3
```



$$1.0*0.2 + 2.0*0.8 + 3.0*-0.5 + 2.0 = 2.3$$

```
inputs = [1.0, 2.0, 3.0, 2.5]
weights1 = [0.2, 0.8, -0.5, 1.0]
weights2 = [0.5, -0.91, 0.26, -0.5]
weights3 = [-0.26, -0.27, 0.17, 0.87]
bias1 = 2.0
bias2 = 3.0
bias3 = 0.5

outputs = [
    inputs[0]*weights1[0] + inputs[1]*weights1[1] + inputs[2]*weights1[2] + inputs[3]*weights1[3] + bias1,
    inputs[0]*weights2[0] + inputs[1]*weights2[1] + inputs[2]*weights2[2] + inputs[3]*weights2[3] + bias2,
    inputs[0]*weights3[0] + inputs[1]*weights3[1] + inputs[2]*weights3[2] + inputs[3]*weights3[3] + bias3
]
print(outputs)

>>> [4.8, 1.21, 2.385]
```

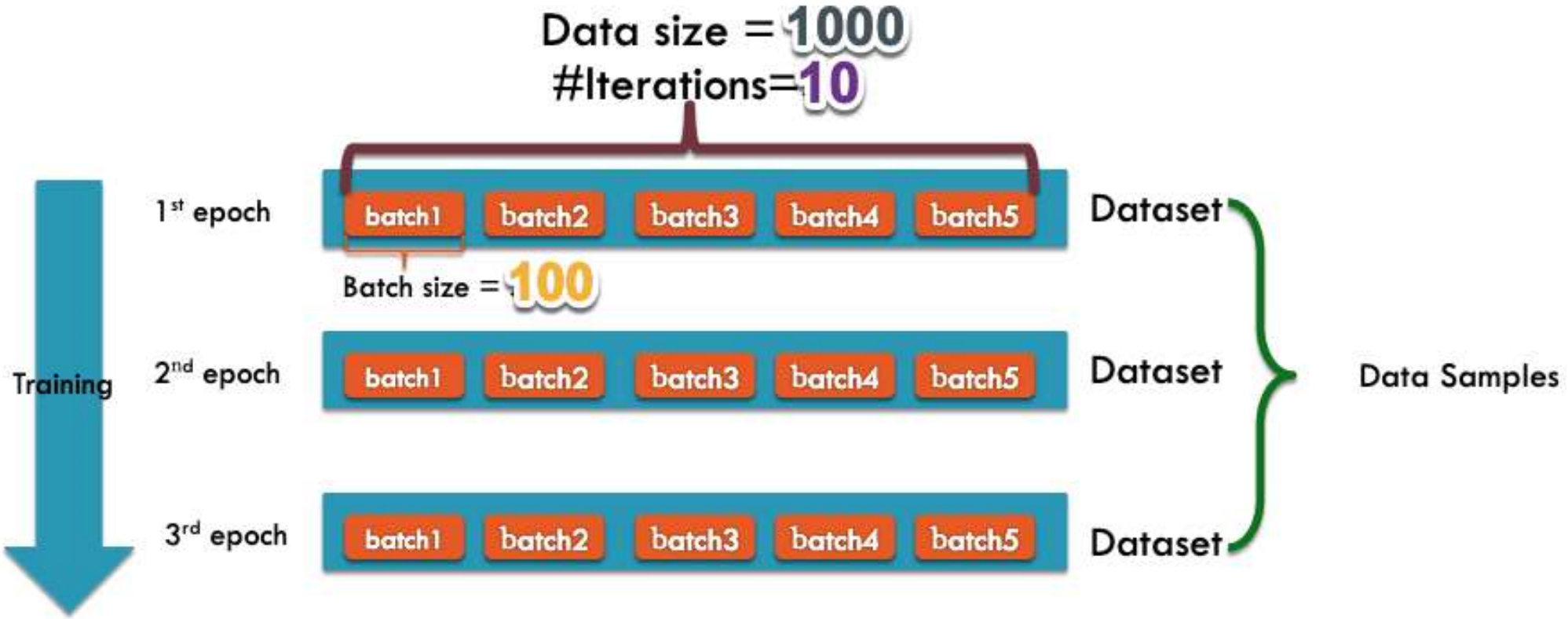
```
1.0*0.2 + 2.0*0.8 + 3.0*-0.5 + 2.5*1.0 + 2.0 = 4.8

1.0*0.5 + 2.0*-0.91 + 3.0*0.26 + 2.5*-0.5 + 3.0 = 1.21

1.0*-0.26 + 2.0*-0.27 + 3.0*0.17 + 2.5*0.87 + 0.5 = 2.385
```

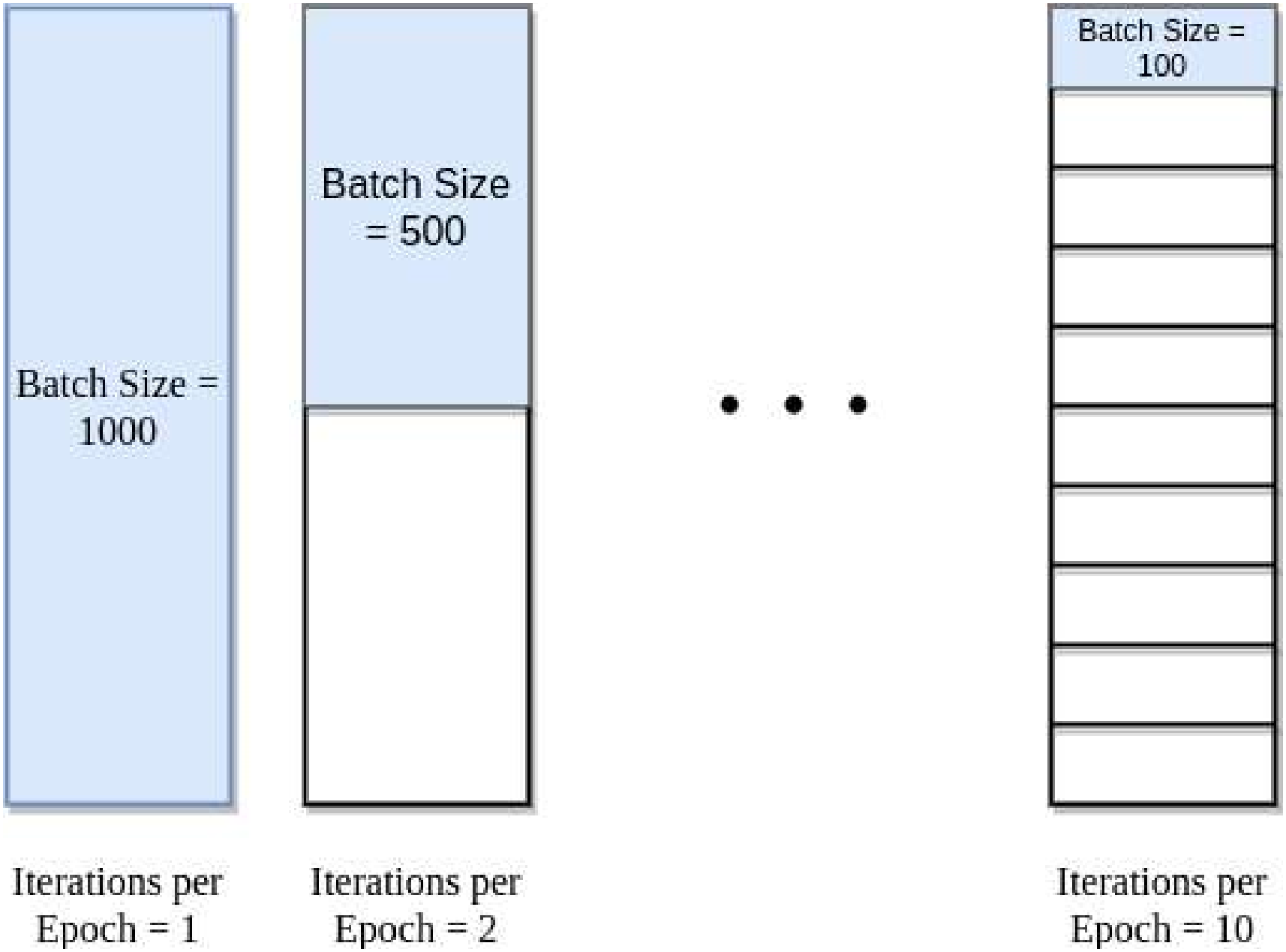
Epoch: One Epoch is when an ENTIRE dataset is passed forward and backward through the neural network only ONCE

Batch Size: Total number of training examples present in a single batch

.

Iteration: The number of passes to complete one epoch.




Batch Size = 1000
Batch Size = 500
Batch Size = 100
Iterations per Epoch = 1
Iterations per Epoch = 2
Iterations per Epoch = 10

Program 4: Build an artificial neural network by implementing the Backpropagation Algorithm and test the same using appropriate datasets.

```
import numpy as np

X = np.array(([2, 9], [1, 5], [3, 6]), dtype=float)

y = np.array(([92], [86], [89]), dtype=float)

X = X/np.amax(X,axis=0) # maximum of X array longitudinally

y = y/100

#Sigmoid Function

def sigmoid (x):

    return 1/(1 + np.exp(-x))

#Derivative of Sigmoid Function

def derivatives_sigmoid (x):

        return x*(1-x)
```

#Variable initialization

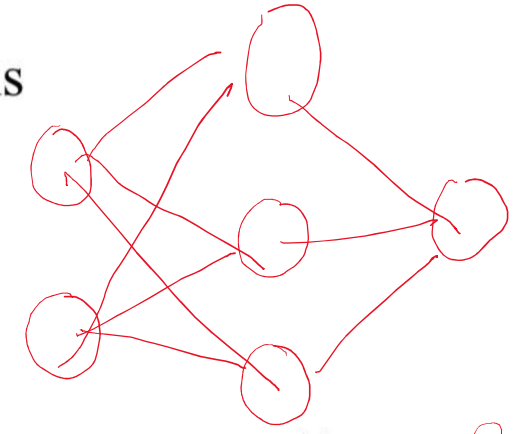
epoch=7000 #Setting training iterations

lr=0.1 #Setting learning rate

inputlayer_neurons = 2 #number of features in data set

hiddenlayer_neurons = 3 #number of hidden layers neurons

output_neurons = 1 #number of neurons at output layer



#weight and bias initialization

wh=np.random.uniform(size=(inputlayer_neurons,hiddenlayer_neurons)) 2 X 3

bh=np.random.uniform(size=(1,hiddenlayer_neurons)) 1 X 3

wout=np.random.uniform(size=(hiddenlayer_neurons,output_neurons)) 1 X 1

3 times

bout=np.random.uniform(size=(1,output_neurons)) 1 X 1

```
#draws a random range of numbers uniformly of dim x*y
for i in range(epoch):
#Forward Propogation
    hinp1=np.dot(X,wh)
    hinp=hinp1 + bh
    hlayer_act = sigmoid(hinp)
    outinp1=np.dot(hlayer_act,wout)
    outinp= outinp1+ bout
    output = sigmoid(outinp)
```

#Backpropagation

```
EO = y-output

outgrad = derivatives_sigmoid(output)

d_output = EO* outgrad

EH = d_output.dot(wout.T)

hiddengrad = derivatives_sigmoid(hlayer_act)#how much hidden layer wts contributed to error

d_hiddenlayer = EH * hiddengrad

wout += hlayer_act.T.dot(d_output) *lr# dotproduct of nextlayererror and currentlayerop


    wh += X.T.dot(d_hiddenlayer) *lr


print("Input: \n" + str(X))

print("Actual Output: \n" + str(y))

print("Predicted Output: \n" ,output)
```

//output

Input:

[[0.66666667 1.   ]

 [0.33333333 0.55555556]

 [1.   0.66666667]]

Actual Output:

[[0.92]

 [0.86]

 [0.89]]

Predicted Output:

 [[0.90224607]

 [0.87341151]

 [0.8925683]]

4) Build an Artificial Neural Network by implementing the Backpropagation algorithm and test the same using appropriate data sets.

```
import numpy as np
X = np.array(([2, 9], [1, 5], [3, 6]), dtype
y = np.array(([92], [86], [89]), dtype=flo
X = X/np.amax(X,axis=0) # maximum o
y = y/100

#Sigmoid Function
def sigmoid (x):
  return 1/(1 + np.exp(-x))

#Sigmoid Function
def derivatives_sigmoid (x):
  return x*(1-x)

#Variable initialization
epoch=7000 #Setting training iterations
lr=0.1 #Setting learning rate
inputlayer_neurons = 2 #number of featu
hiddenlayer_neurons = 3 #number of hid
output_neurons = 1 #number of neurons
```

```
#weight and bias initialization
wh=np.random.uniform(size=(inputlayer_neurons,hiddenlayer_neurons))
bh=np.random.uniform(size=(1,hiddenlayer_neurons))
wout=np.random.uniform(size=(hiddenlayer_neurons,output_neurons))
bout=np.random.uniform(size=(1,output_neurons))

#draws a random range of numbers uniformly of dim x*y
for i in range(epoch):
#Forward Propogation
  hinp1=np.dot(X,wh)
  hinp=hinp1 + bh
  hlayer_act = sigmoid(hinp)
  outinp1=np.dot(hlayer_act,wout)
  outinp= outinp1+ bout
  output = sigmoid(outinp)
#Backpropagation
  EO = y-output
  outgrad = derivatives_sigmoid(output)
  d_output = EO* outgrad
  EH = d_output.dot(wout.T)
  hiddengrad = derivatives_sigmoid(hlayer_act)#how much hidden layer wts contributed to error
  d_hiddenlayer = EH * hiddengrad
  wout += hlayer_act.T.dot(d_output) *lr# dotproduct of nextlayererror and currentlayerop
  # bout += np.sum(d_output, axis=0,keepdims=True) *lr
  wh += X.T.dot(d_hiddenlayer) *lr
  #bh += np.sum(d_hiddenlayer, axis=0,keepdims=True) *lr
print("Input: \n" + str(X))
print("Actual Output: \n" + str(y))
```

Back propagation steps

$$\delta_k \leftarrow o_k(1-o_k)(t_k-o_k)$$

calculate its error term δ_h

$$\delta_h \leftarrow o_h(1-o_h)\sum_{k \in outputs} w_{kh}\delta_k$$

$$w_{ji} \leftarrow w_{ji} + \Delta w_{ji}$$

$$\Delta w_{ji} = \eta\, \delta_j\, x_{ji}$$

Calculate $(t_k - o_k)$

Calculate $o_k(1-o_k)$

Calculate $\delta_k \leftarrow o_k(1-o_k)(t_k-o_k)$

Calculate $\sum_{k \in outputs} w_{kh}\delta_k$

Calculate $o_k(1-o_k)$

Calculate $\delta_h \leftarrow o_h(1-o_h)\sum_{k \in outputs} w_{kh}\delta_k$

$$w_{ji} \leftarrow w_{ji} + \Delta w_{ji}$$

$$\Delta w_{ji} = \eta\, \delta_j\, x_{ji}$$

T : means transpose because here weight vector used

Data Set : Play Hrs, Study Hrs, Sleep Hrs

Features (Hrs Slept , Hrs Studied)

Labels (marks obtained)

Chaitra.H

4) Build an Artificial Neural Network by implementing the Backpropagation algorithm and test the same using appropriate data sets.

```
import numpy as np
X = np.array(([2, 9], [1, 5], [3, 6]), dtype=float)
y = np.array(([92], [86], [89]), dtype=float)
X = X/np.amax(X,axis=0) # maximum of X array longitudinally
y = y/100
```

Normalizing within Scale 0 to 1

Normalize values between 0 to 1.

Normalize Output

Return max of an array (or) max along an axis.

```
#Sigmoid Function
def sigmoid (x):
    return 1/(1 + np.exp(-x))

#Sigmoid Function
def derivatives_sigmoid (x):
    return x*(1-x)
```

// derivative of Sigmoid func

like $= out_1(1-out_1)$ in ex.

```
#Variable initialization
epoch=7000 #Setting training iterations
lr=0.1 #Setting learning rate
inputlayer_neurons = 2 #number of features in data set
hiddenlayer_neurons = 3 #number of hidden layers neurons
output_neurons = 1 #number of neurons at output layer
```

(make changes in this & see results)

#amount that the NN are updated during training.

```
#weight and bias initialization
wh=np.random.uniform(size=(inputlayer_neurons,hiddenlayer_neurons))
bh=np.random.uniform(size=(1,hiddenlayer_neurons))
wout=np.random.uniform(size=(hiddenlayer_neurons,output_neurons))
bout=np.random.uniform(size=(1,output_neurons))
```

Random

(1 x 1) - 1 time

2x3

1 x 3

1 x 1 — 3 times

I/p layer

O/p layer

random.uniform(a, b)

↓

return random floating

print no. N

such that (a ≤ N ≤ b if a ≤ b, b ≤ N ≤ a if b ≤ a)

```
#draws a random range of numbers uniformly of dim x*y
for i in range(epoch):
#Forward Propogation
    hinp1=np.dot(X,wh)
    hinp=hinp1 + bh
    hlayer_act = sigmoid(hinp)
    outinp1=np.dot(hlayer_act,wout)
    outinp= outinp1+ bout
    output = sigmoid(outinp)
#Backpropagation
    EO = y-output
    outgrad = derivatives_sigmoid(output)
    d_output = EO* outgrad
    EH = d_output.dot(wout.T)
    hiddengrad = derivatives_sigmoid(hlayer_act)#how much hidden layer wts contributed to error
    d_hiddenlayer = EH * hiddengrad
    wout += hlayer_act.T.dot(d_output) *lr# dotproduct of nextlayererror and currentlayerop
    # bout += np.sum(d_output, axis=0,keepdims=True) *lr
    wh += X.T.dot(d_hiddenlayer) *lr
    #bh += np.sum(d_hiddenlayer, axis=0,keepdims=True) *lr
print("Input: \n" + str(X))
print("Actual Output: \n" + str(y))
```

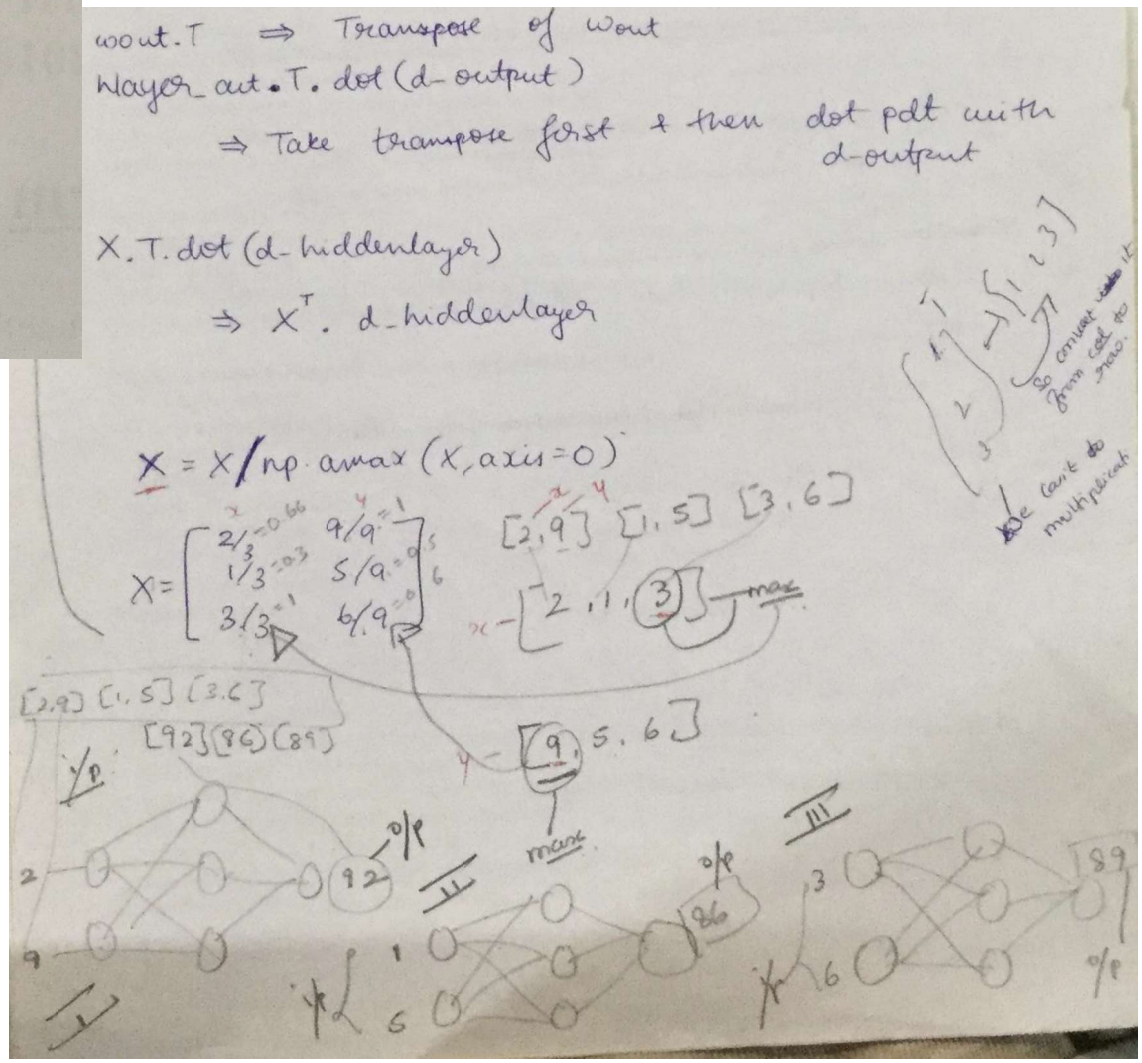
#Dot product + bias layer
hinp = np.dot(X, wh) + bh

Wout.T ⟹ Transpose of wout

hlayer_act.T.dot(d_output) ⟹ Take transpose first & then dot product with d_output.

X.T.dot(d_hiddenlayer) ⟹ X^T . d_hiddenlayer

```
print("Predicted Output: \n" ,output)


//output
Input:
[[ 0.66666667  1.        ]
 [ 0.33333333  0.55555556]
 [ 1.          0.66666667]]
Actual Output:
[[ 0.92]
 [ 0.86]
 [ 0.89]]
Predicted Output:
 [[ 0.90224607]
 [ 0.87341151]
 [ 0.8925683 ]]
```



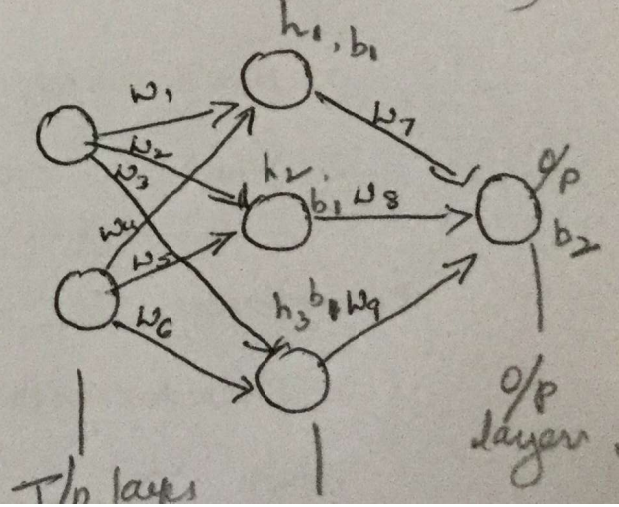
4) Build an Artificial Neural Network by implementing the Backpropagation algorithm and test the same using appropriate data sets.

```
import numpy as np
X = np.array(([2, 9], [1, 5], [3, 6]), dtype=float)
y = np.array(([92], [86], [89]), dtype=float)
X = X/np.amax(X,axis=0) # maximum of X array longitudinally
y = y/100

#Sigmoid Function
def sigmoid (x):
    return 1/(1 + np.exp(-x))

#Sigmoid Function
def derivatives_sigmoid (x):
    return x*(1-x)

#Variable initialization
epoch=7000 #Setting training iterations
lr=0.1 #Setting learning rate
inputlayer_neurons = 2 #number of features in data set
hiddenlayer_neurons = 3 #number of hidden layers neurons
output_neurons = 1 #number of neurons at output layer

#weight and bias initialization
wh=np.random.uniform(size=(inputlayer_neurons,hiddenlayer_neurons))
bh=np.random.uniform(size=(1,hiddenlayer_neurons))
wout=np.random.uniform(size=(hiddenlayer_neurons,output_neurons))
bout=np.random.uniform(size=(1,output_neurons))
```

// derivative of Sigmoid func

random uniform(a, b)
↓
return random floating
print no. N
such that (∃ 'N')
$a \leq N \leq b$ if $a \leq b$
$b \leq N \leq a$ if $b \leq a$

(make changes in this & see result)



```
#draws a random range of numbers uniformly of dim x*y
for i in range(epoch):
#Forward Propogation
    hinp1=np.dot(X,wh)
    hinp=hinp1 + bh
    hlayer_act = sigmoid(hinp)
    outinp1=np.dot(hlayer_act,wout)
    outinp= outinp1+ bout
    output = sigmoid(outinp)
#Backpropagation
    EO = y-output
    outgrad = derivatives_sigmoid(output)
    d_output = EO* outgrad
    EH = d_output.dot(wout.T)
    hiddengrad = derivatives_sigmoid(hlayer_act)#how much hidden layer wts contributed to error
    d_hiddenlayer = EH * hiddengrad
    wout += hlayer_act.T.dot(d_output) *lr# dotproduct of nextlayererror and currentlayerop
   # bout += np.sum(d_output, axis=0,keepdims=True) *lr
    wh += X.T.dot(d_hiddenlayer) *lr
    #bh += np.sum(d_hiddenlayer, axis=0,keepdims=True) *lr
print("Input: \n" + str(X))
print("Actual Output: \n" + str(y))
```

```
print("Predicted Output: \n" ,output)
```

//output

```
Input:
[[ 0.66666667  1.        ]
 [ 0.33333333  0.55555556]
 [ 1.          0.66666667]]
Actual Output:
[[ 0.92]
 [ 0.86]
 [ 0.89]]
Predicted Output:
 [[ 0.90224607]
 [ 0.87341151]
 [ 0.8925683 ]]
```