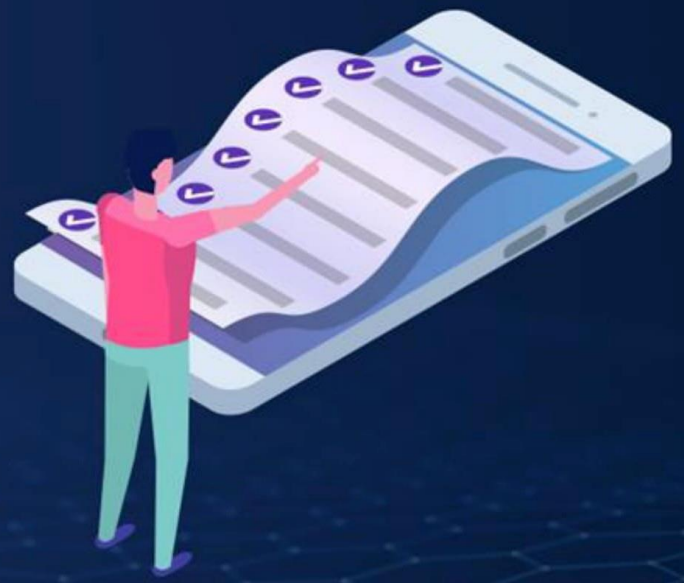
Artificial Neural Network

BY PRABHAT KUMAR

Learning Objectives

By the end of this lesson, you will be able to:

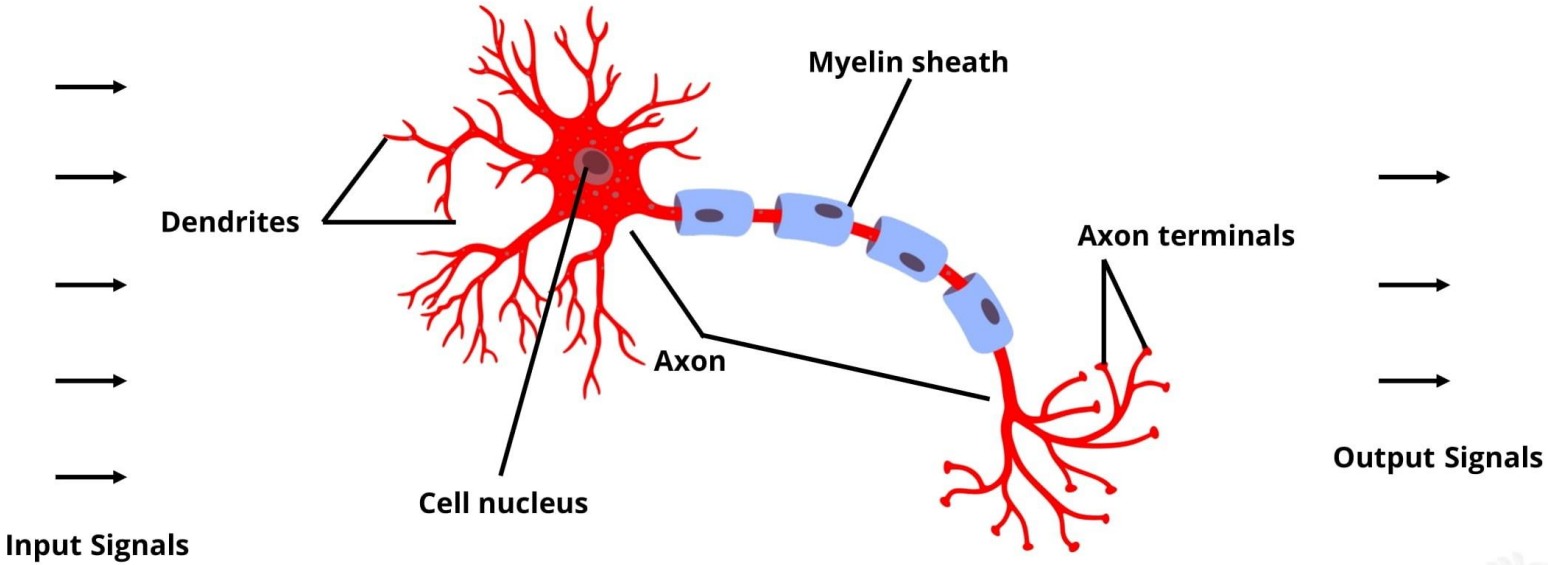
- Explore neural networks
- Perform weight updation using different activation functions
- Deduce and implement backpropagation algorithm in Python
- Optimize the performance of your neural network using L2 regularization and dropout layers



DATA AND
ARTIFICIAL INTELLIGENCE

Biological Neuron vs. Artificial Neuron

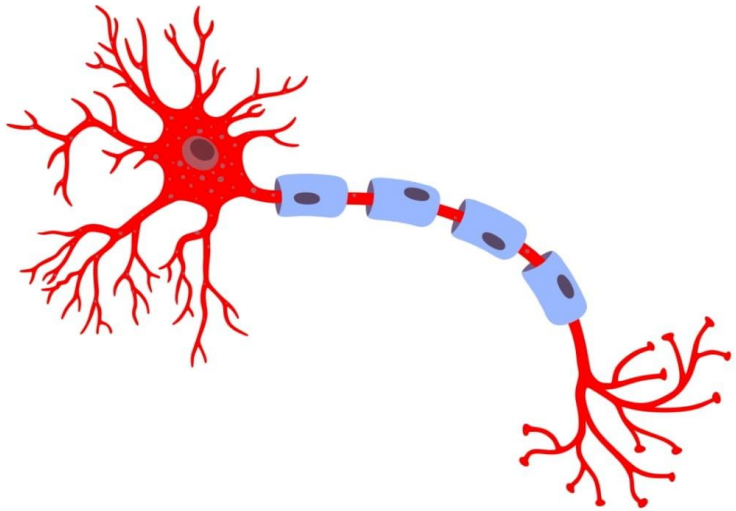
Biological Neurons



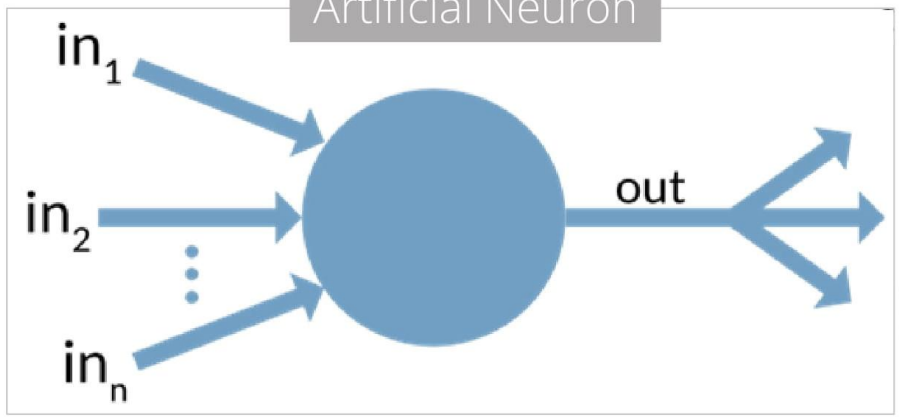
- Neurons are interconnected nerve cells that build the nervous system and transmit information throughout the body.
- **Dendrites** are extension of a nerve cell that receive impulses from other neurons.
- **Cell** nucleus stores cell's hereditary material and coordinates cell's activities.
- **Axon** is a nerve fiber that is used by neurons to transmit impulses.
- **Synapse** is the connection between two nerve cells.

Rise of Artificial Neurons

Biological Neuron



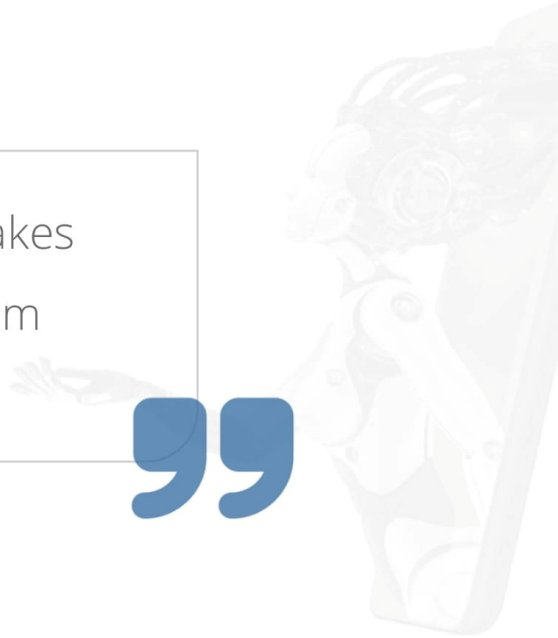
Artificial Neuron



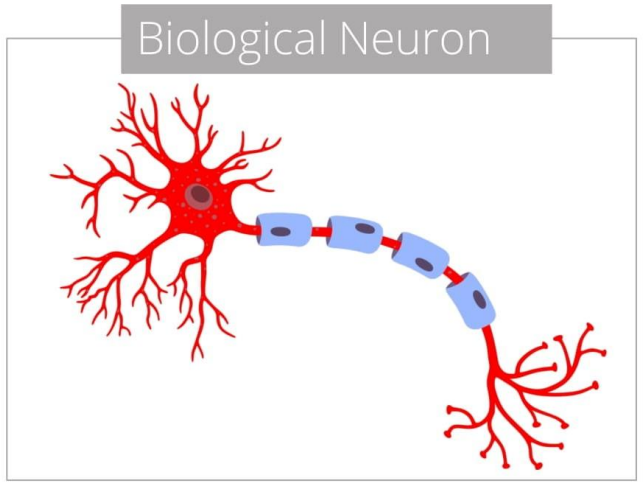
- Researchers Warren McCullock and Walter Pitts published their first concept of simplified brain cell in 1943.
- Nerve cell was considered similar to a simple logic gate with binary outputs.
- Dendrites can be assumed to process the input signal with a certain threshold such that if the signal exceeds the threshold, the output signal is generated.

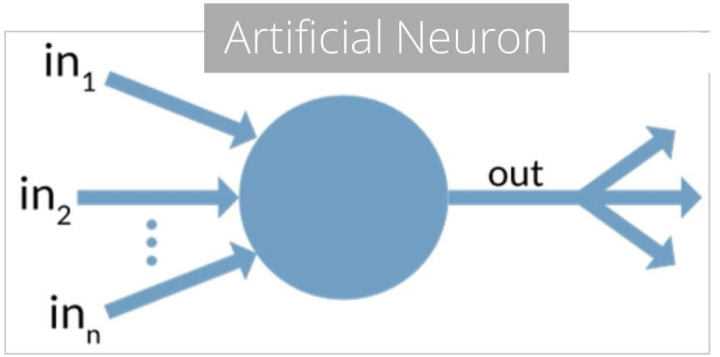
Definition of Artificial Neuron

> An artificial neuron is analogous to biological neurons, where each neuron takes inputs, adds weights to them separately, sums them up, and passes this sum through a transfer function to produce a nonlinear output.



Biological Neurons and Artificial Neurons: A Comparison





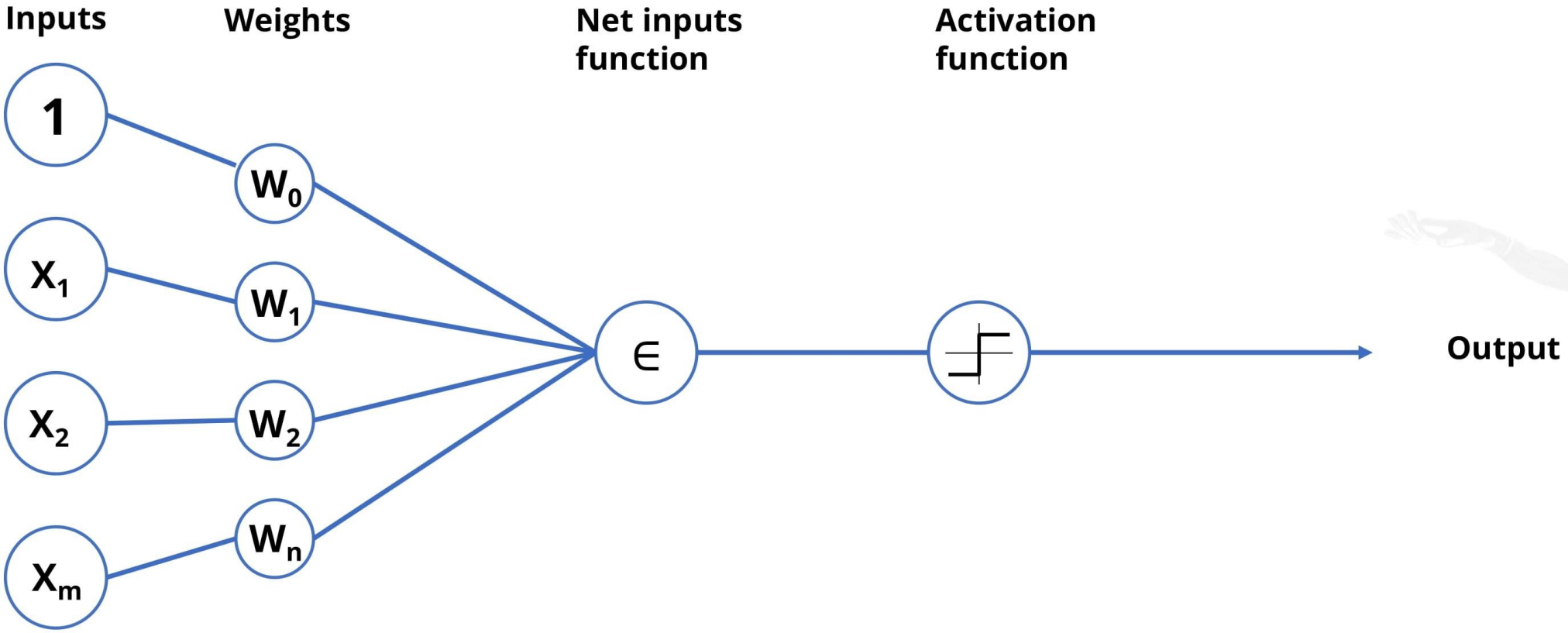
Biological Neurons	Artificial Neurons
Cell Nucleus	Node
Dendrites	Input
Synapse	Weights or interconnections
Axon	Output

DATA AND
ARTIFICIAL INTELLIGENCE

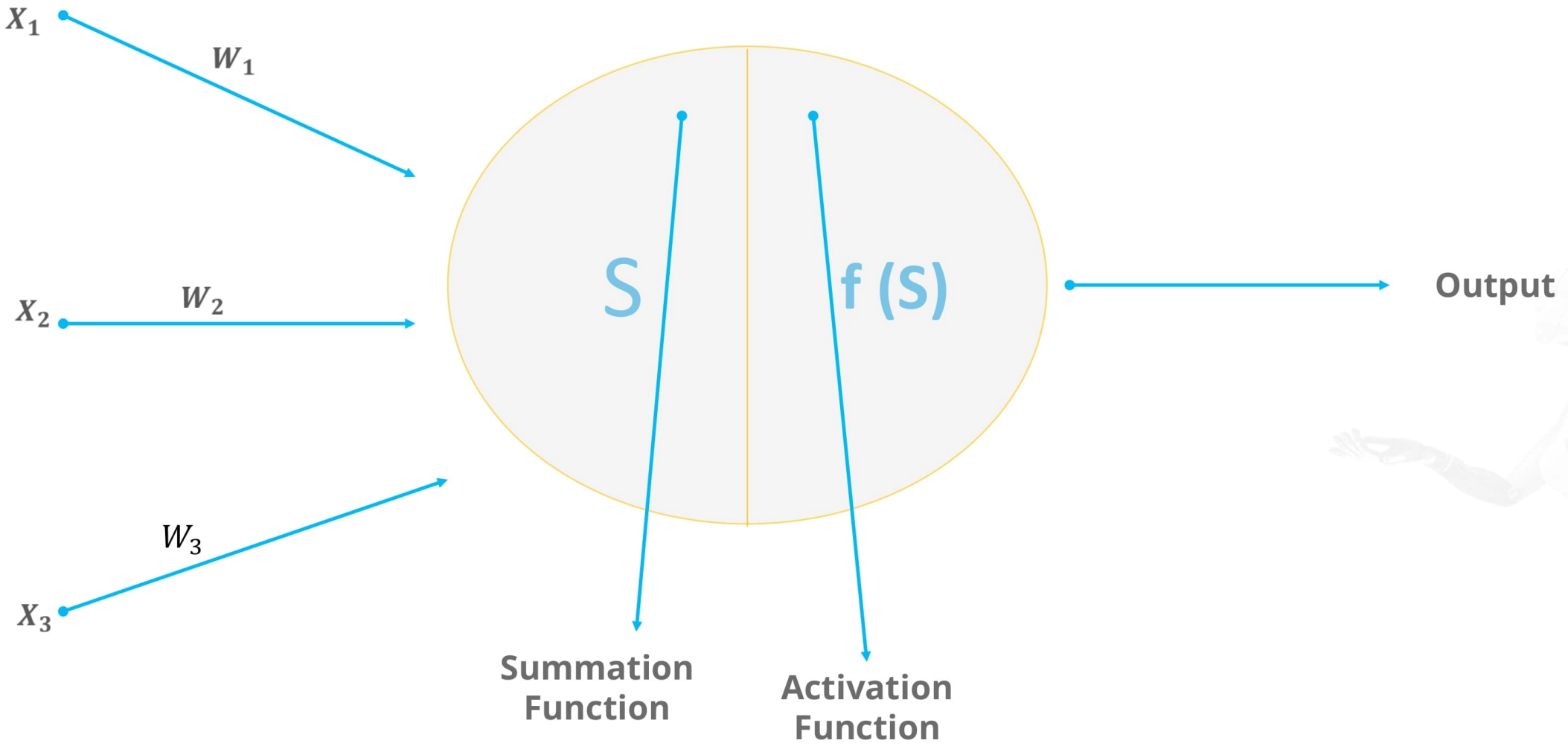
Neural Networks

Perceptron

- Single layer neural network
- Consists of weights, the summation processor, and an activation function



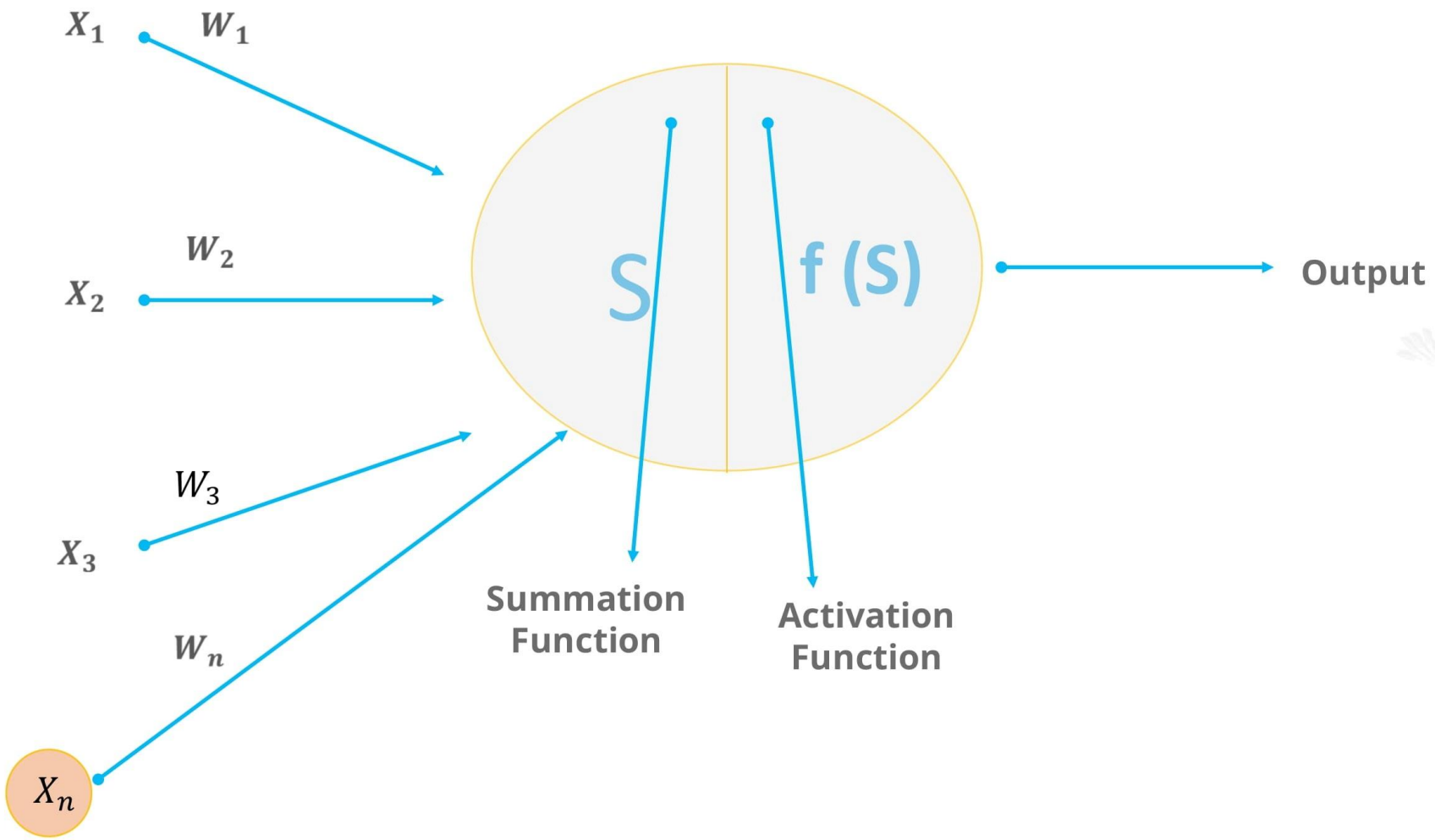
Perceptron: The Main Processing Unit



Note: Inputs X and weights W are real values.

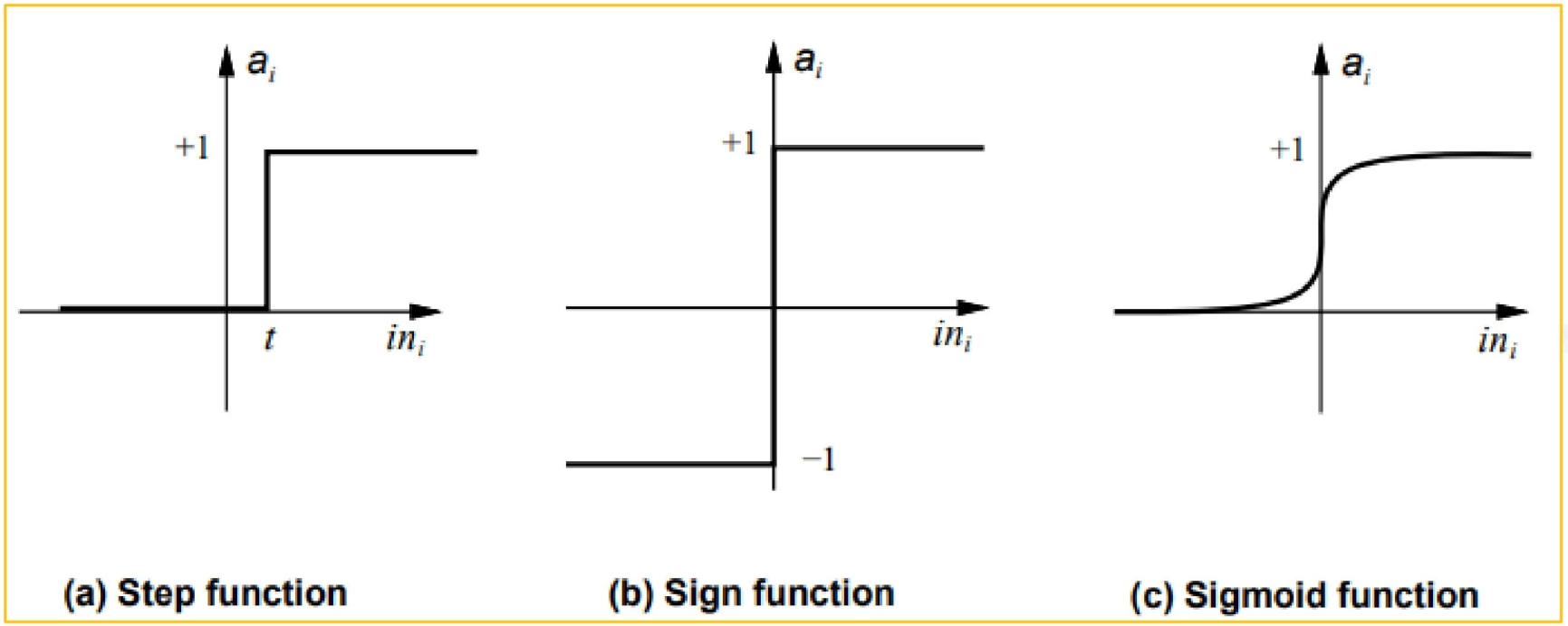
Weights and Biases in a Perceptron

While the weights determine the slope of the equation, bias shifts the output line towards left or right.



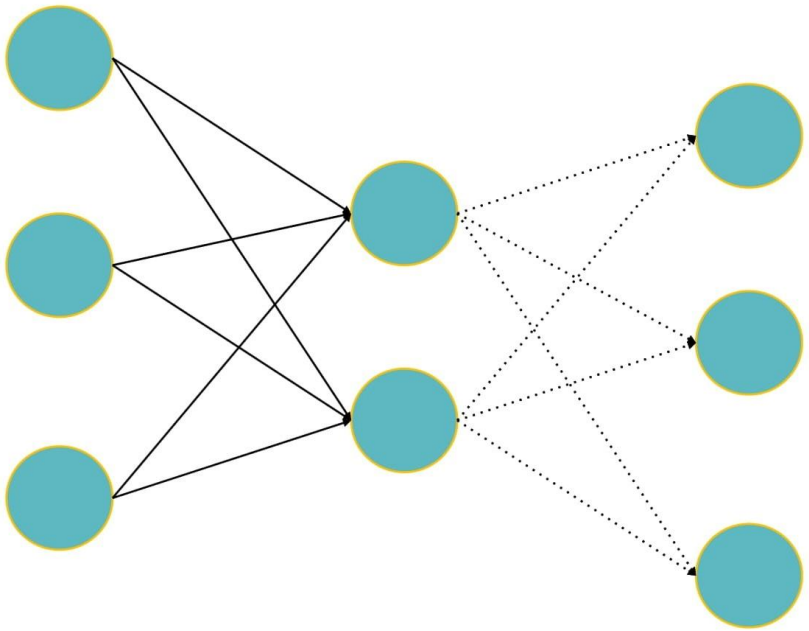
Activation Functions

Activation functions squash inputs to outputs by using a threshold value.



(a) Step function (b) Sign function (c) Sigmoid function

Feedforward Nets



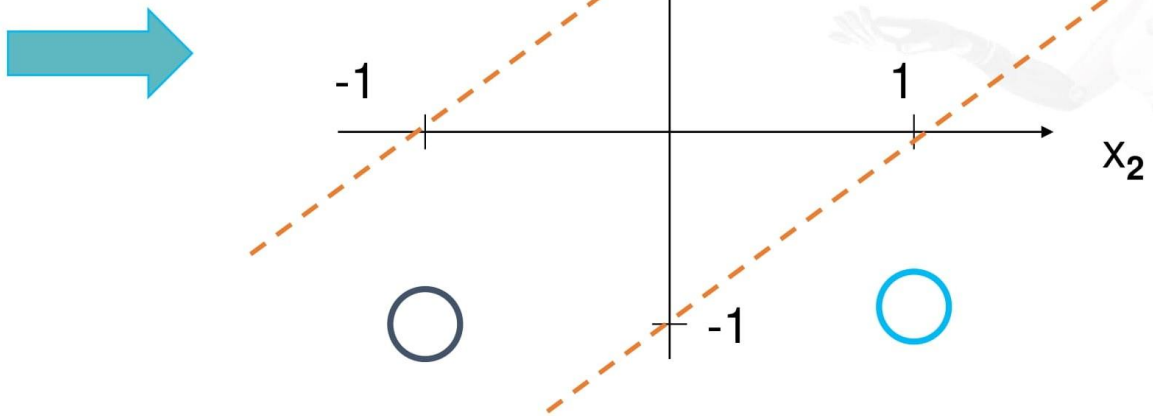
- Information flow is unidirectional
- Information is distributed
- Information processing is parallel

The XOR Problem

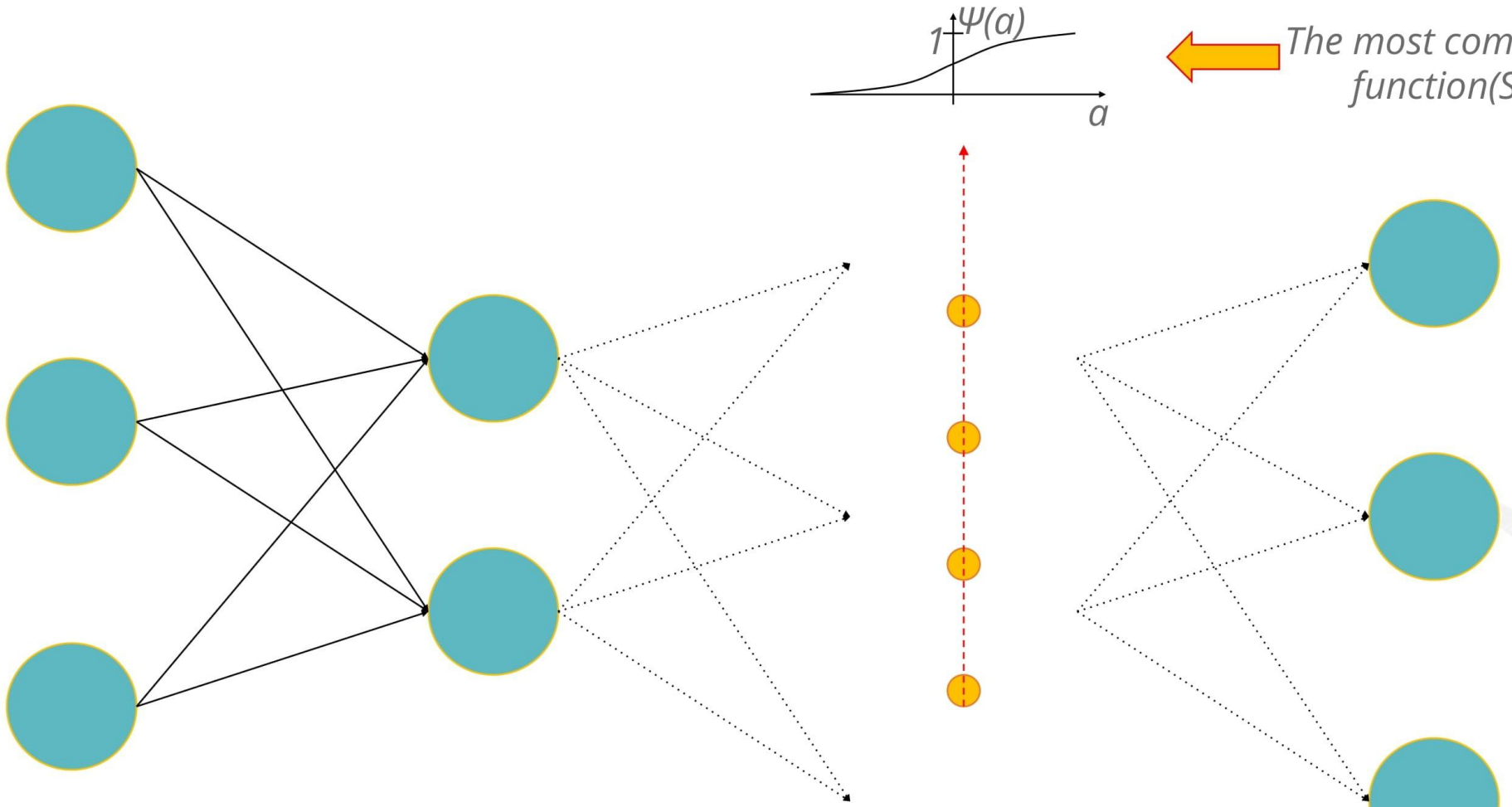
A perceptron can learn anything that it can represent, i.e., anything separable with a hyperplane.

However, it cannot represent Exclusive OR since it is not linearly separable.

x_1	x_2	x_1 xor x_2
-1	-1	-1
-1	1	1
1	-1	1
1	1	-1



Multilayer Perceptrons



The most common output function(Sigmoid)

Input nodes

One or more layers of hidden units (hidden layers)

Output neurons

Perceptron

Problem Scenario: You are hired by one of the major AI giants, planning to build the best image classifier model available till date. In the first phase of model development, the input is passed from the MNIST dataset. MNIST dataset is one of the most common datasets used for image classification and is accessible from many different sources. It is a subset of a larger set available from NIST and contains 60,000 training images and 10,000 testing images on handwritten digits taken from American Census Bureau employees and American high school students.

Objective:

Build a perceptron-based classification model to:

- Classify the handwritten digits properly
- Make predictions
- Evaluate model efficiency

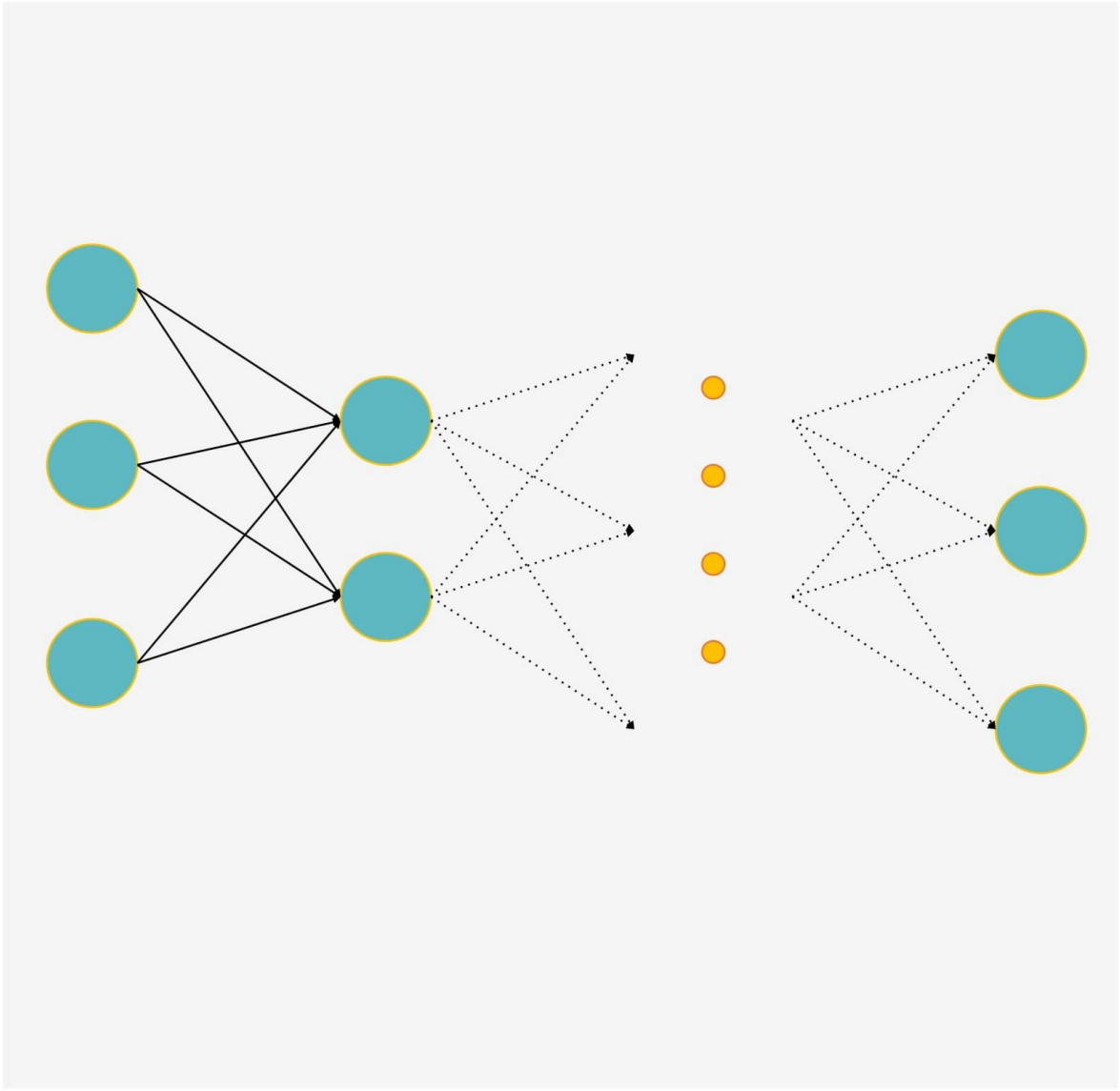
Access: Click the Practice Labs tab on the left panel. Now, click on the START LAB button and wait while the lab prepares itself. Then, click on the LAUNCH LAB button. A full-fledged jupyter lab opens, which you can use for your hands-on practice and projects.

ASSISTED PRACTICE

DATA AND
ARTIFICIAL INTELLIGENCE

Backpropagation

Learning Networks



- Learn from the input data or from training examples and generalize from learned data
- By training the network, it tries to find a line, plane or a hyperplane which can correctly separate two classes by adjusting weights and biases
- The network configures itself to solve a problem

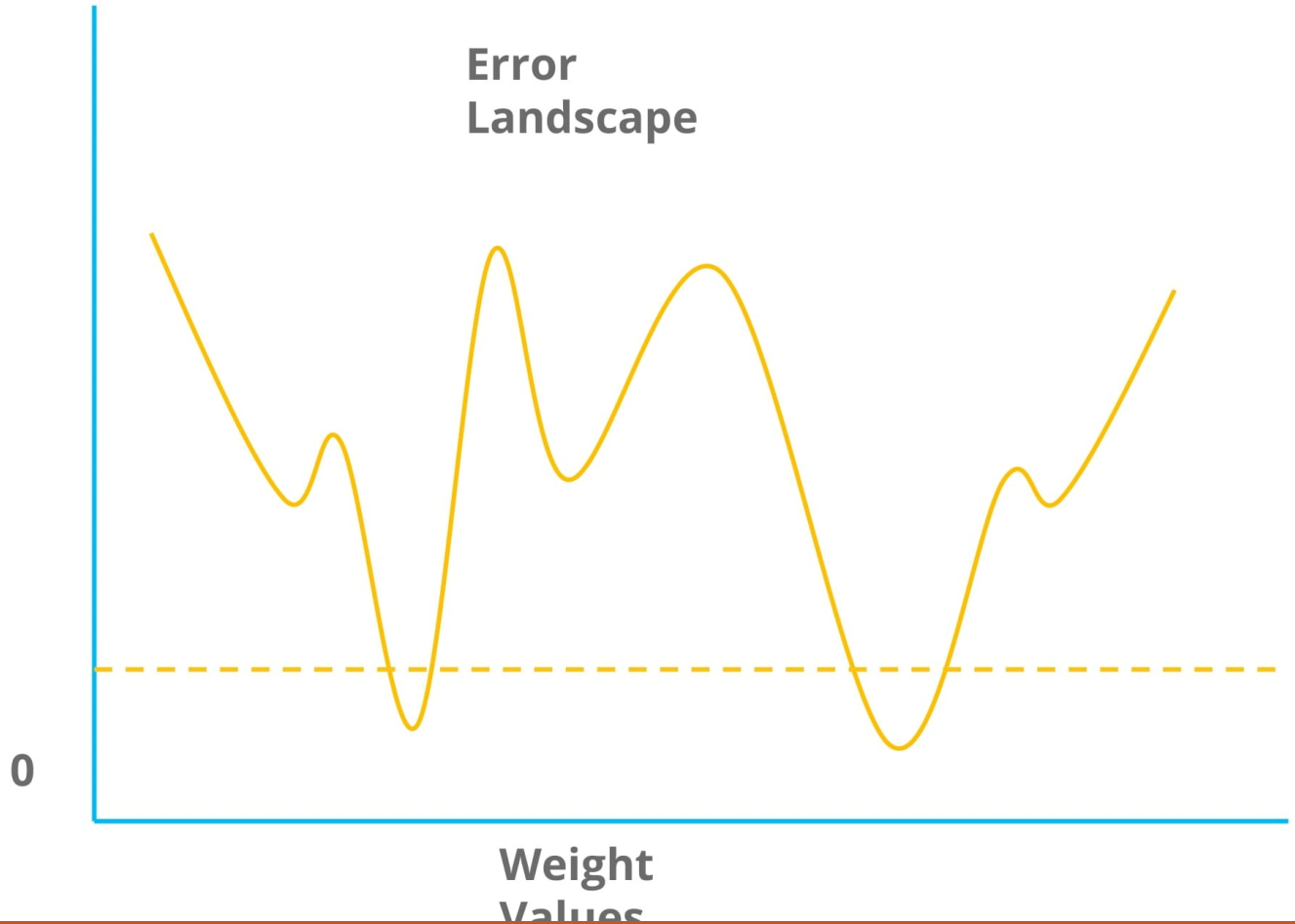
The Backpropagation Algorithm

Initialize the weights and the threshold	Provide the input and calculate the output	Update the weights	Repeat the initial steps
Let W_i be the initial weight	Let X be the input and Y be the output	W_i (t+1) = W_i (t) +n(d-y)X, where d is the desired output and y is the actual output	The former steps are iterated continuously by changing the values of n till a considerable output is obtained

The Error Landscape

The objective is to minimize the SSE (Sum Squared Error)

SSE:
Sum Squared Error
S $(t_i - z_i)^2$

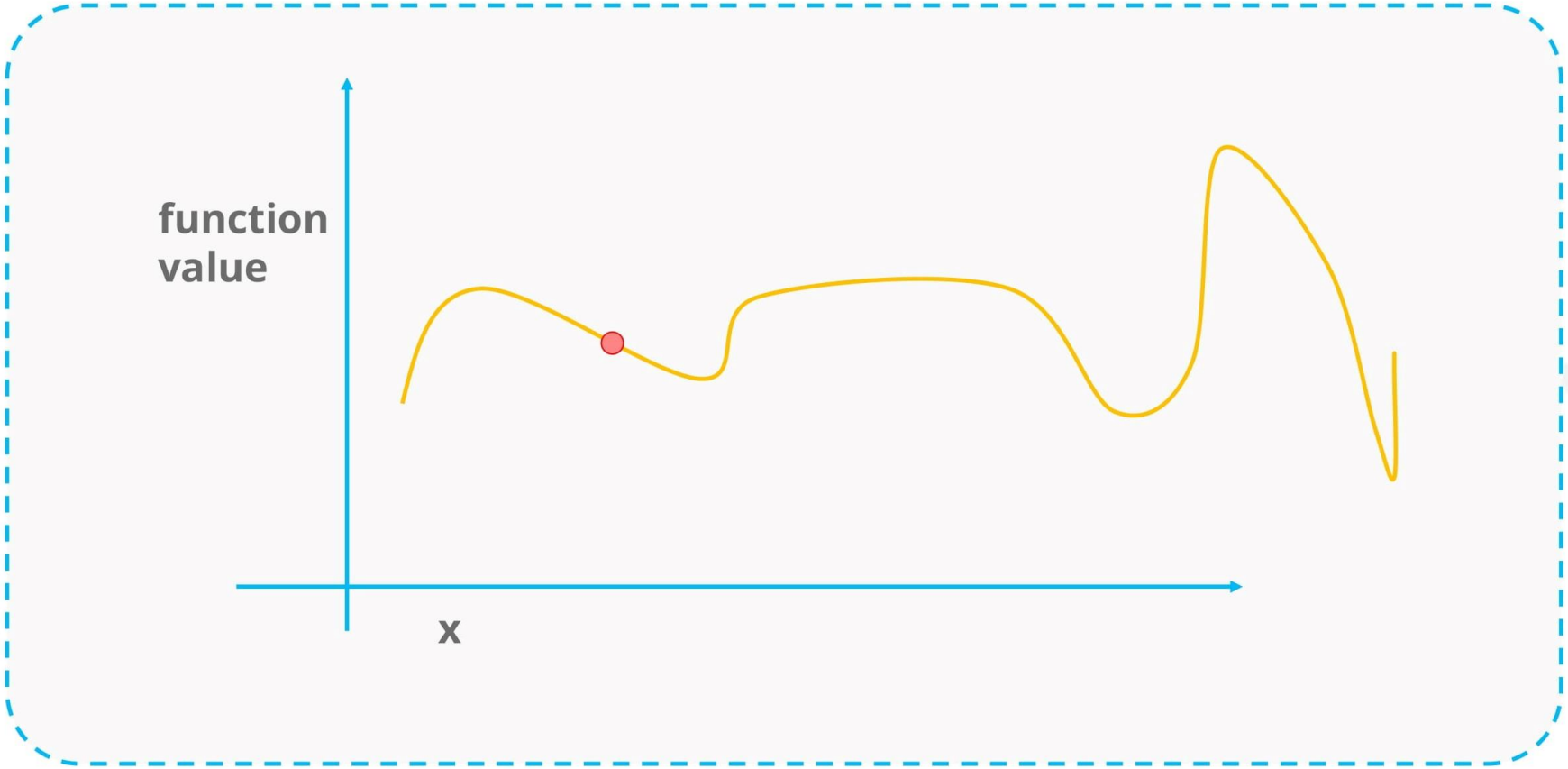


Deriving a Gradient Descent or Ascent Algorithm

- The idea of the algorithm is to decrease overall error (or other objective function) each time a weight is changed.
- Look at local gradient: the direction of largest change.
- Take a step in that direction such that the step size is proportional to gradient.
- Gradient descent tends to yield much faster convergence to maximum.

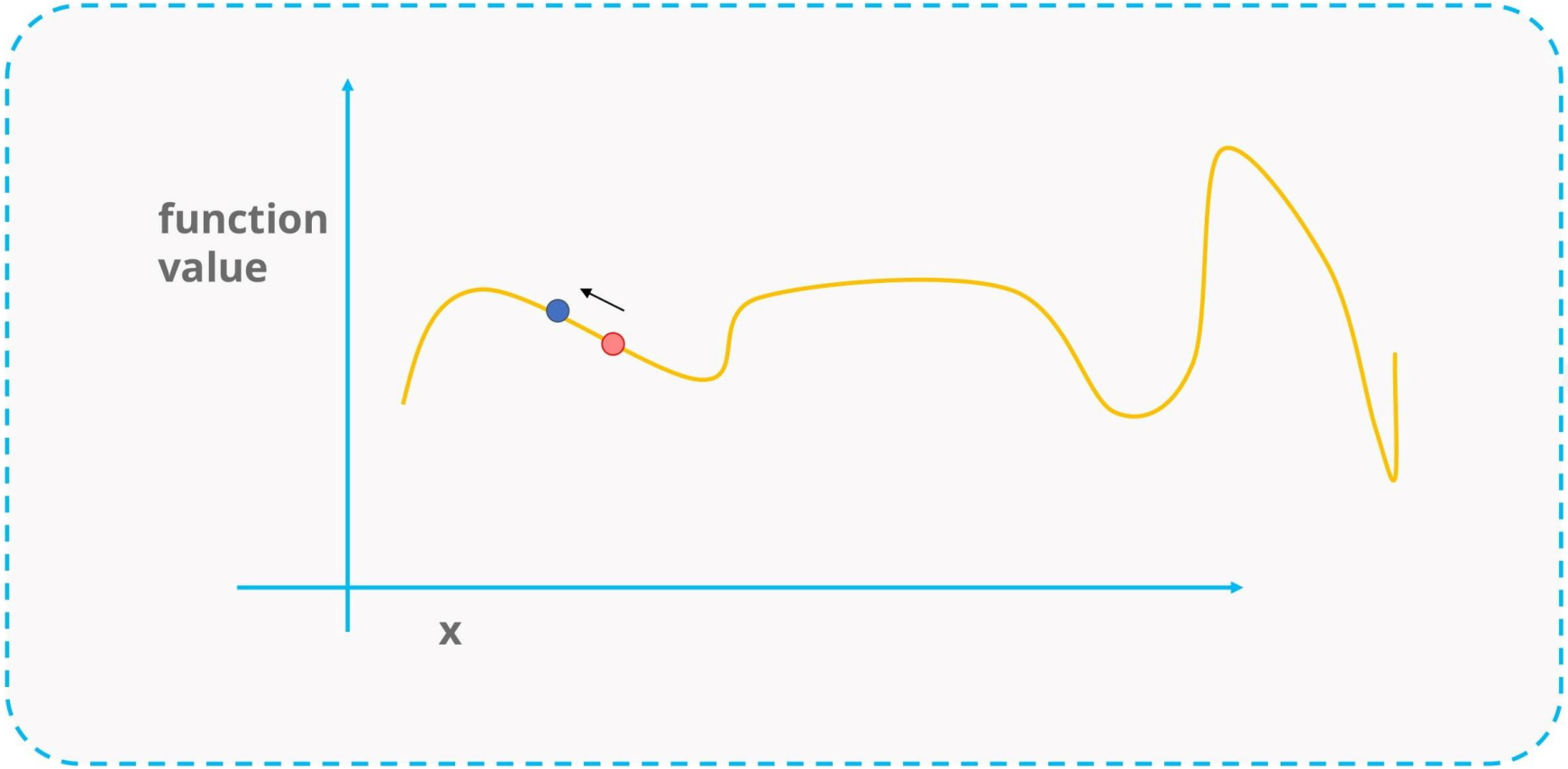
Gradient Ascent: Step 01

Initialize a random starting point



Gradient Ascent: Step 02

Take a step in the direction of the largest increase

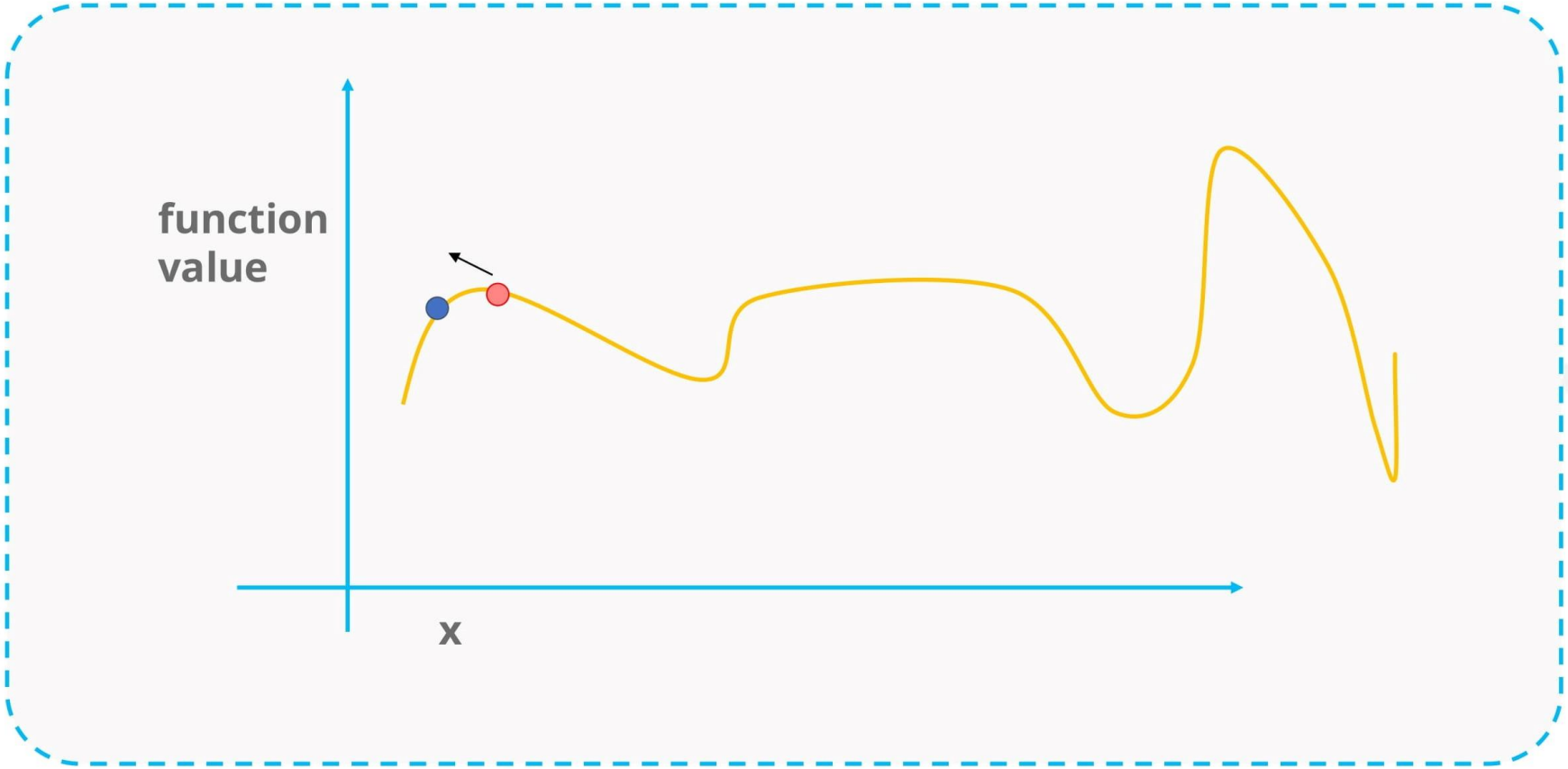


Gradient Ascent: Step 03

Repeat

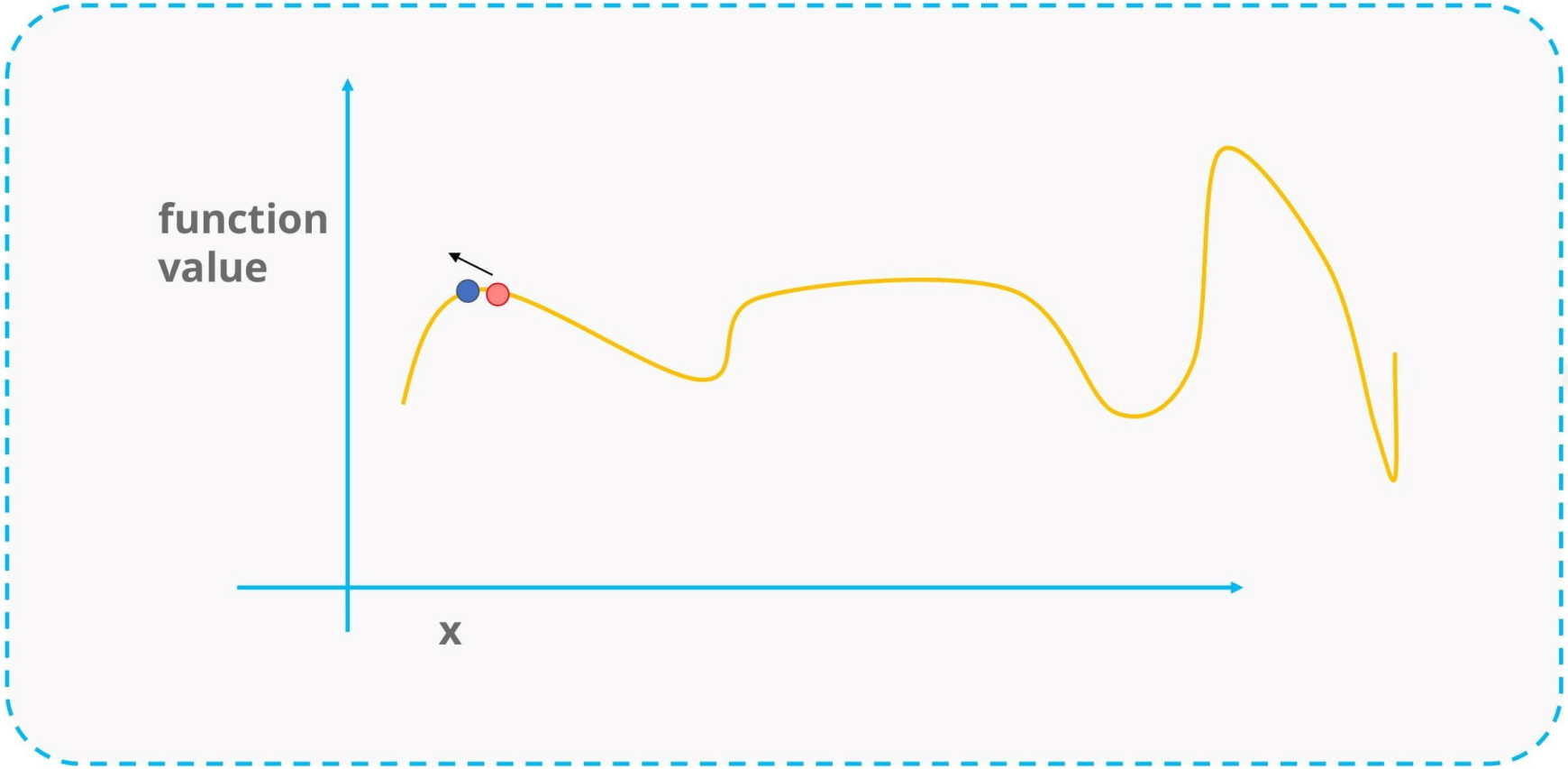
Gradient Ascent: Step 04

Next step is lower, so stop



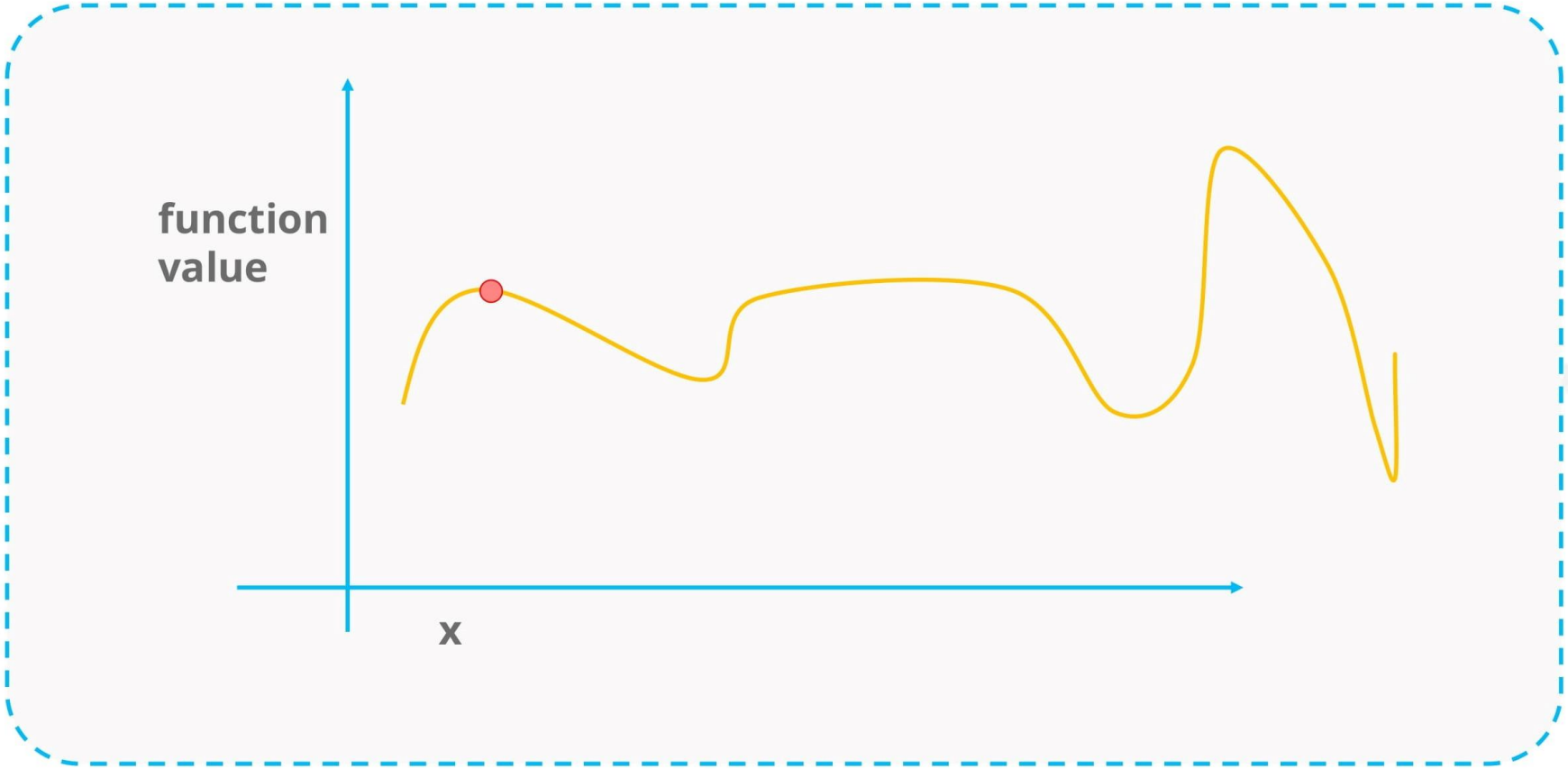
Gradient Ascent: Step 05

Reduce step size to “hone in”



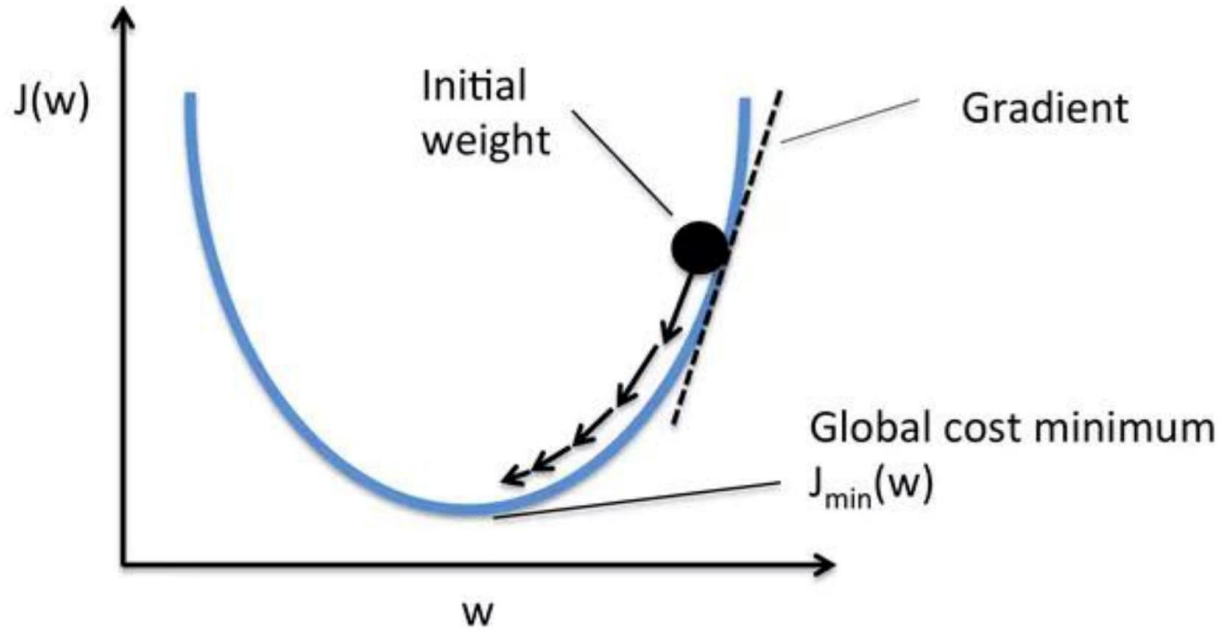
Gradient Ascent: Step 06

Converge to (local) maximum



The Learning Rate

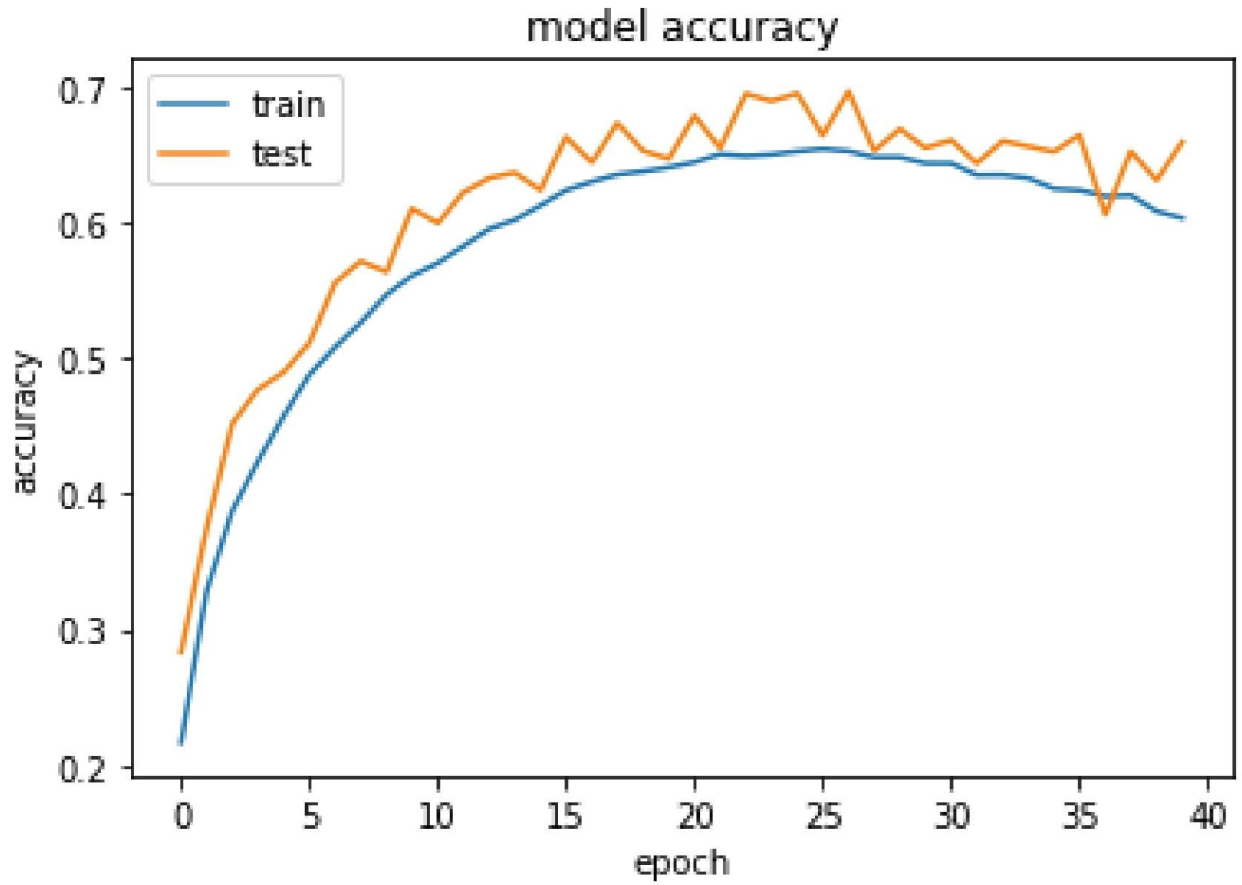
- Learning rate is used to control the changes made to the weights and biases in order to reduce the error.
- It is used to analyze how an error will change when the values of weights and biases are changed by a unit.



Note: Generally, a learning rate of 0.01 is a safe bet.

Epoch

- One epoch consists of one full cycle of training data
- Preferred value of number of training epochs is set to 1000

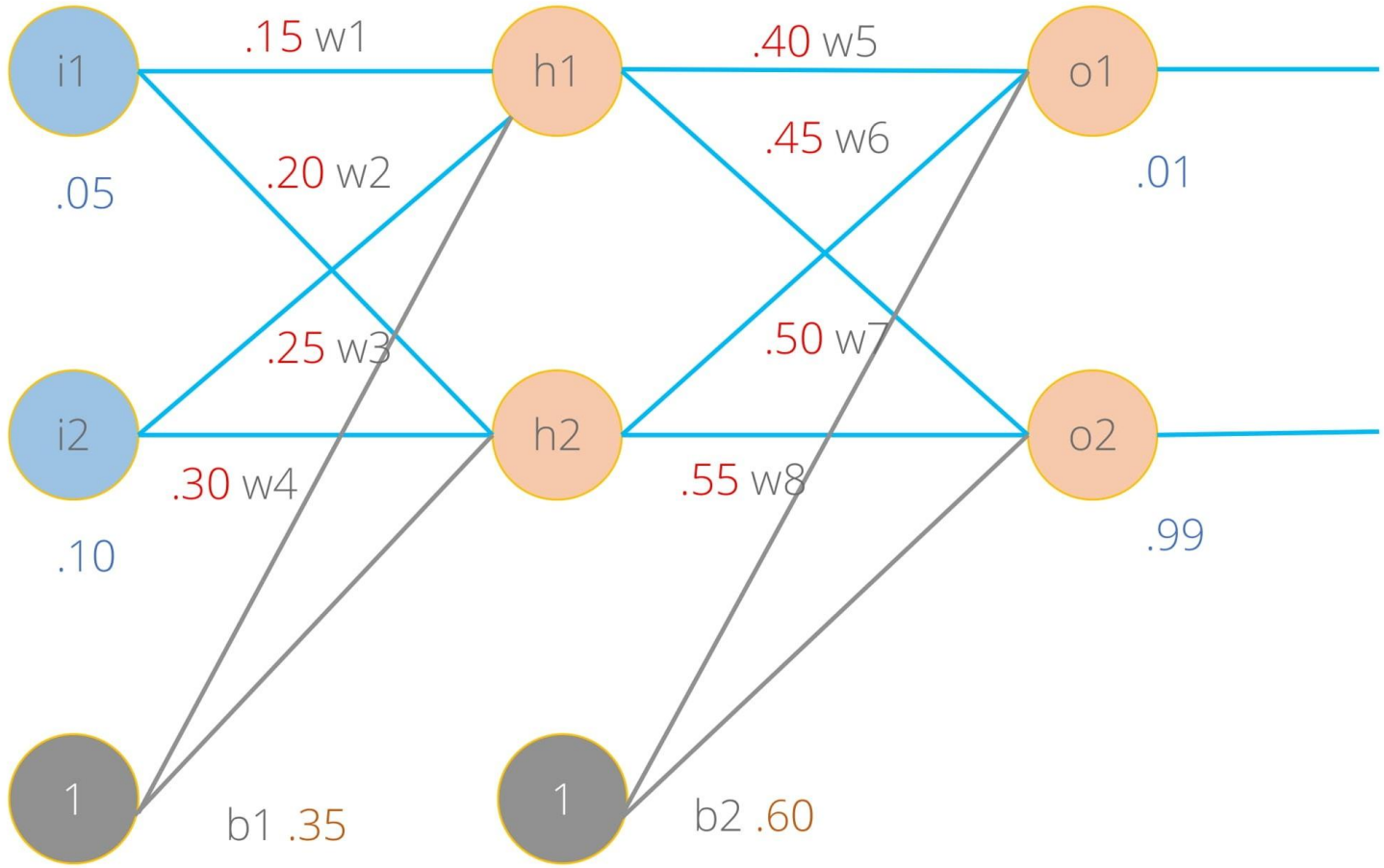


DATA AND
ARTIFICIAL INTELLIGENCE

Backpropagation: Example

A Feed Forward Network

Consider a forward pass network



Forward Pass

net_{h1} = w1 * i1 + w2 * i2 + b1*1

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

Squash it using logistic function to get the output:

Out_{h1} = 1/1 + $e^{-net_{h1}}$ = 0.593269992

Out_{h2} = 0.596884378

Forward Pass

net_{o1} = w5 * Out_{h1} + w6 * Out_{h2} + b2*1

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

Out_{o1} = 1/1 + $e^{-net_{o1}}$ = 0.75136507

Carrying out the same process for o2, we get:

Out_{02} = 0.772928465

Calculating Total Error

$E_{Total} = \sum 1/2(target - output)^2$

$E_{o1} = 1/2(Target_{o1} - Out_{o1})^2$ = ½(0.01 – 0.753136507)^2 = 0.274811083

E_{o2} = 0.023560026

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

The Backward Pass

Let us consider W_5:

$$\frac{dE_{Total}}{dw_5} = \frac{dE_{Total}}{dout_{o1}} * \frac{dout_{o1}}{dnet_{o1}} * \frac{dnet_{o1}}{dw_5}$$

$$\delta_{01} = \frac{dE_{Total}}{dout_{o1}} * \frac{dout_{o1}}{dnet_{o1}} = \frac{dE_{Total}}{dnet_{o1}}$$

$$E_{Total} = E_{01} + E_{02}$$
$$E_{01} = ½ (target_{01} - out_{01})\text{^}2$$
$$out_{01} = 1/1+e^{-net_{01}}$$
$$net_{01} = w5 * out_{h1} + w6 * out_{h2} + b2 * 1$$

The Backward Pass

$$\frac{dE_{Total}}{dw_5} = \frac{dE_{Total}}{dout_{o1}} * \frac{dout_{o1}}{dnet_{o1}} * \frac{dnet_{o1}}{dw_5}$$

$$E_{Total} = ½ (target_{01} - out_{01})^2 + ½(target_{02} + out_{02})^2$$

$$\frac{dE_{Total}}{dout_{01}} = 2* ½ (target_{01} - out_{01})^2-1 * -1 + 0$$

$$\frac{dE_{Total}}{dout_{01}} = -(target_{01} - out_{01}) = -(0.01-0.75136507) = 0.74136507$$

The Backward Pass

$$\frac{dE_{Total}}{dw_5} = \frac{dE_{Total}}{dout_{o1}} * \frac{dout_{o1}}{dnet_{o1}} * \frac{dnet_{o1}}{dw_5}$$

$$out_{01} = 1/1+ e^{-net_{01}}$$

$$\frac{dout_{01}}{dnet_{01}} = out_{01} (1 - out_{01})$$

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

$$\frac{df(x)}{dx} = f(x)(1- (f(x))$$

$$\frac{dout_{01}}{dnet_{01}} = 0.75136507(1-0.75136507) = 0.186815602$$

The Backward Pass

net_{01} = w5 * out_{h1} + w6 * out_{h2} + b2 *1

$\frac{dnet_{01}}{dw_5}$ = 1 * out_{h1} * w_5 ^ (1-1) + 0 +0 = out_{h1} = 0.593269992

$\frac{dE_{Total}}{dw_5} = \frac{dE_{Total}}{dout_{o1}} * \frac{dout_{o1}}{dnet_{o1}} * \frac{dnet_{o1}}{dw_5}$

$\frac{dE_{Total}}{dw_5}$ = 0.74136507 * 0.186815602 * 0.593269992 = 0.082167041

Weight Updation

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

$$w_5^+ = w_5 - \eta * \frac{dE_{total}}{dw_5} = 0.4 - 0.5*0.082167041 = 0.35891648$$

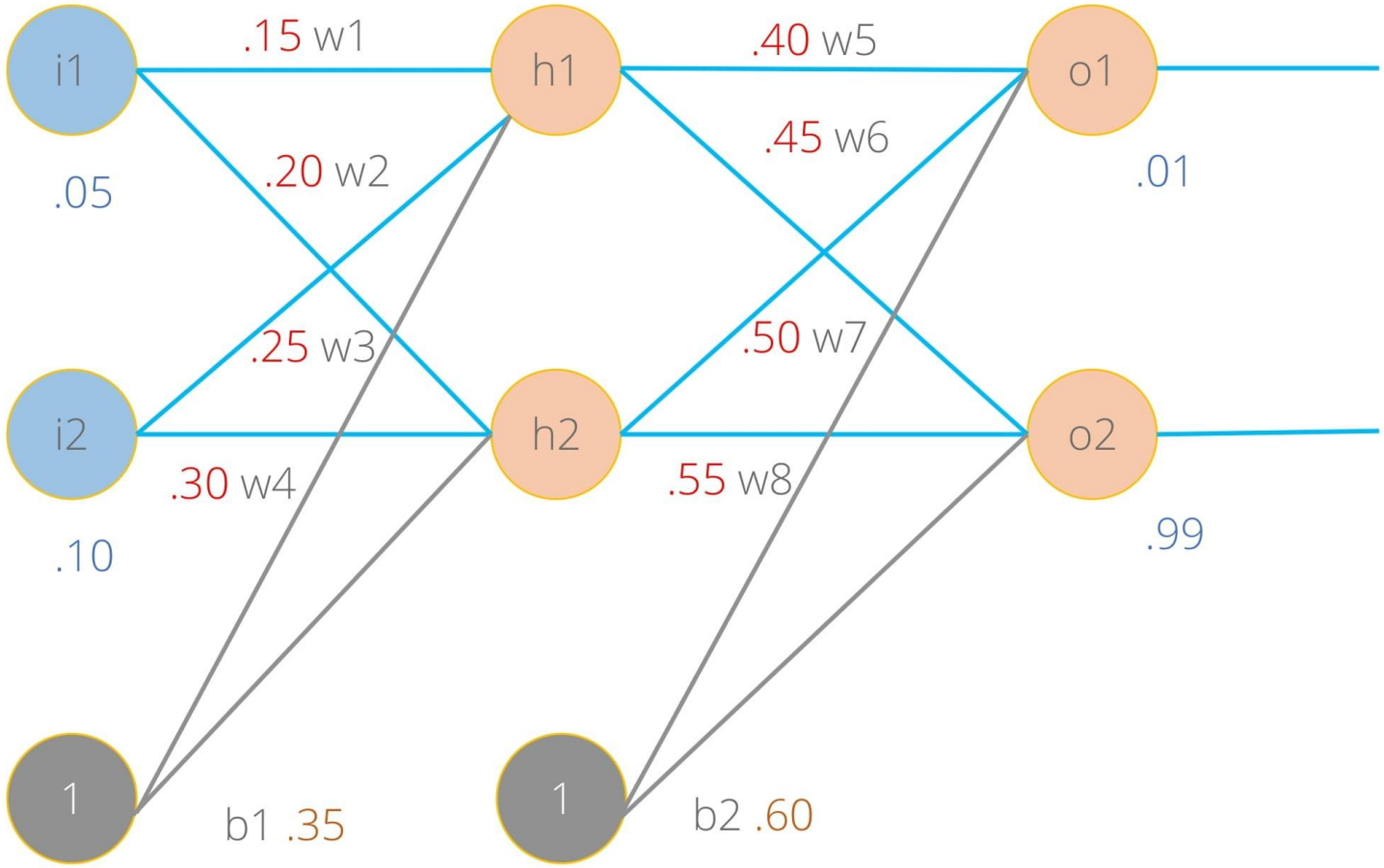
η = Learning rate

$w_6^+ = 0.408666186$

$w_7^+ = 0.511301270$

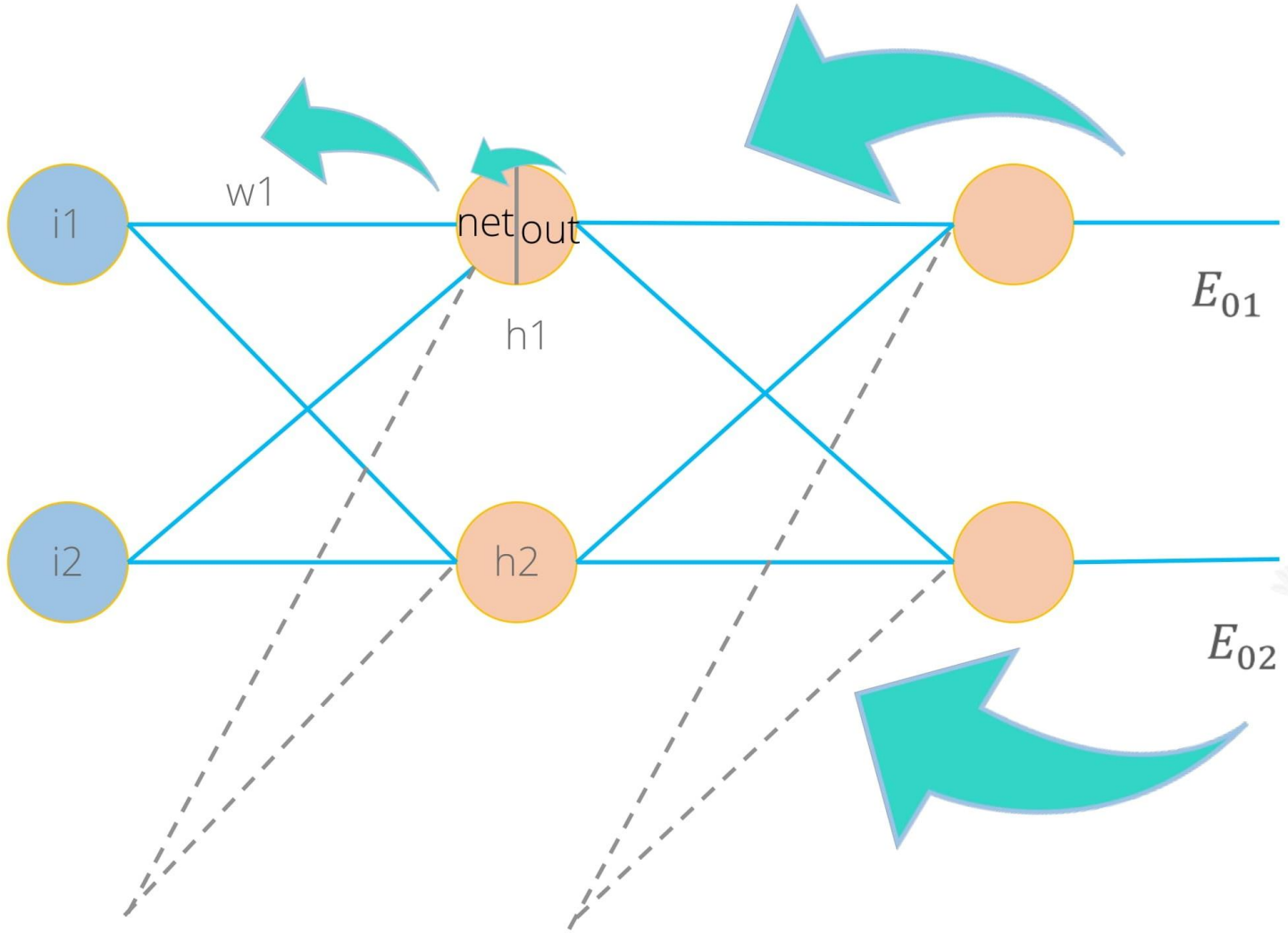
$w_8^+ = 0.561370121$

Weight Updation



Note: While using the backpropagation mechanism, we use the original weights, not the updated weights.

Hidden Layer Weight Assignment



$$\frac{dE_{Total}}{dw_1} = \frac{dE_{Total}}{dout_{h1}} * \frac{dout_{h1}}{dnet_{h1}} * \frac{dnet_{h1}}{dw_1}$$

Hidden Layer Weight Assignment

$$\frac{dE_{Total}}{dout_{h1}} = \frac{dE_{o1}}{dout_{h1}} + \frac{dE_{o2}}{dout_{h1}}$$

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

$$out_{o1} = 1/1 + e^{-net_{01}}$$

$$net_{01} = w_5 * out_{h1} + w_6 * out_{h2} + b_2 * 1$$

$$\frac{dE_{Total}}{dout_{h1}} = \frac{dE_{o1}}{dnet_{o1}} * \frac{dnet_{o1}}{dout_{h1}}$$

$$\frac{dE_{o1}}{dnet_{o1}} = \frac{dE_{o1}}{dout_{o1}} * \frac{dout_{o1}}{dnet_{o1}} = \frac{dnet_{o1}}{dout_{h1}} = w_5$$

Hidden Layer Weight Assignment

$$\frac{dE_{01}}{dout_{h1}} = \frac{dE_{o1}}{dnet_{o1}} * \frac{dnet_{o1}}{dout_{h1}} = 0.138498562*0.40 = 0.055399425$$

$$\frac{dE_{02}}{dout_{h1}} = -0.019049119$$

$$\frac{dE_{total}}{dout_{h1}} = \frac{dE_{01}}{dout_{h1}} + \frac{dE_{02}}{dout_{h1}} = 0.55399425 + -0.019049119 = 0.036350306$$

Hidden Layer Weight Assignment

$$\frac{dE_{total}}{dw_1} = \frac{dE_{total}}{dout_{h1}} * \frac{dout_{h1}}{dnet_{h1}} * \frac{dnet_{h1}}{dw_1}$$

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

$$\frac{dout_{h1}}{dnet_{h1}} = out_{h1}\,(1- out_{h1}\,) = 0.59326999(1-0.59326999) = 0.241300709$$

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

$$\frac{dnet_{h1}}{dw_1} = i_1 = 0.05$$

$$\frac{dE_{total}}{dw_1} = 0.036350306 * 0.241300709 * 0.05 = 0.000438568$$

Hidden Layer Weight Assignment

W_1 can be updated now:

(1- out_{h1}) = 0.59326999(1-0.59326999) = 0.241300709

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

$$w_1^+ = w_1 - \eta * \frac{dE_{total}}{dw_1} = 0.15 - 0.5 * 0.000438568 = 0.149780716$$

The same needs to be repeated for w_2, w_3, and w_4.

The same needs to be repeated for W_2, W_3, and W_4.

Initially the error was 0.298371109. However, post first iteration of the backpropagation algorithm the error value lowered down to 0.291027924.

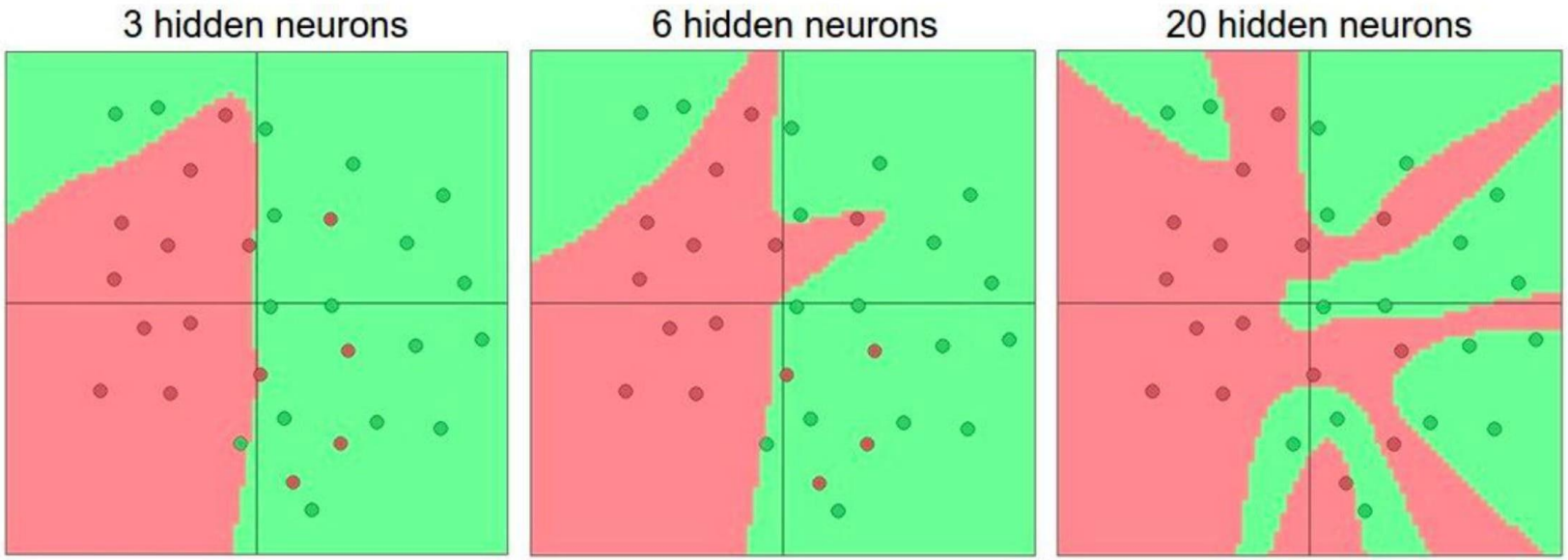
The above error can be reduced significantly by repeating this process 10,000 times or more.

DATA AND
ARTIFICIAL INTELLIGENCE

Activation Functions

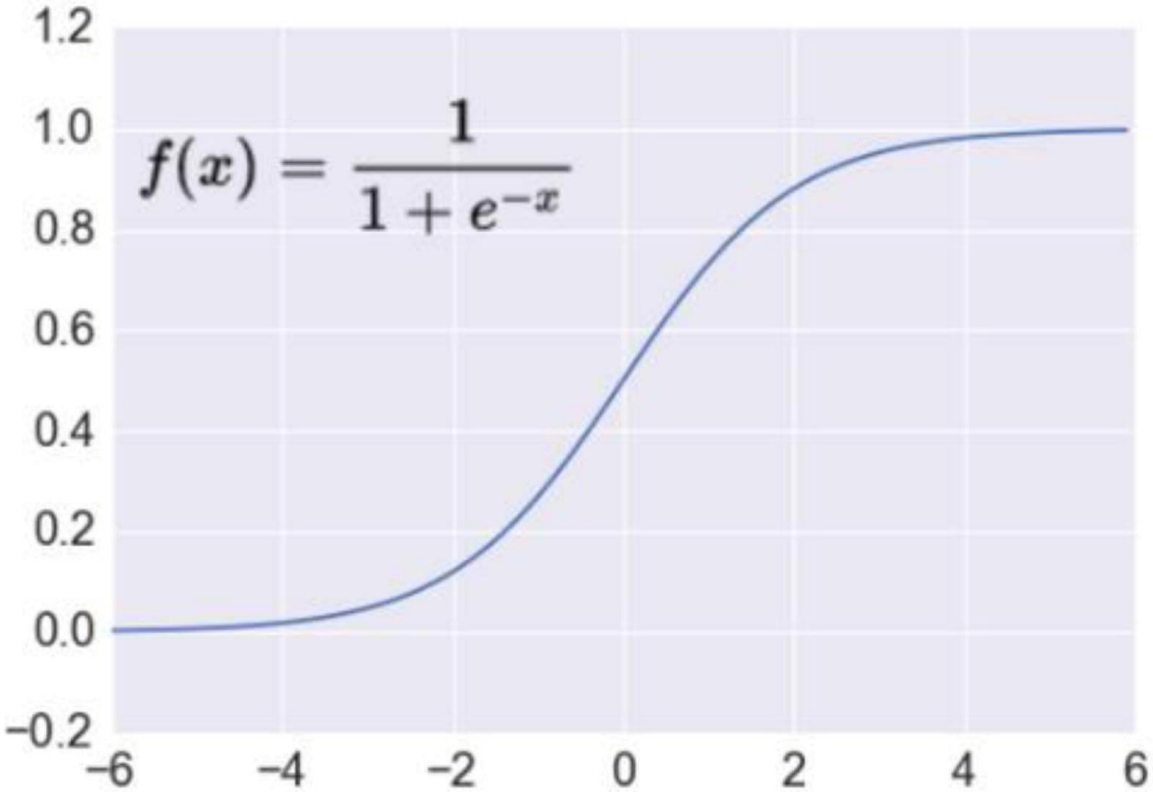
Activation Functions

Nonlinearities are needed to learn complex (non-linear) representations of data, otherwise the neural networks would be just a linear function



Activation Functions

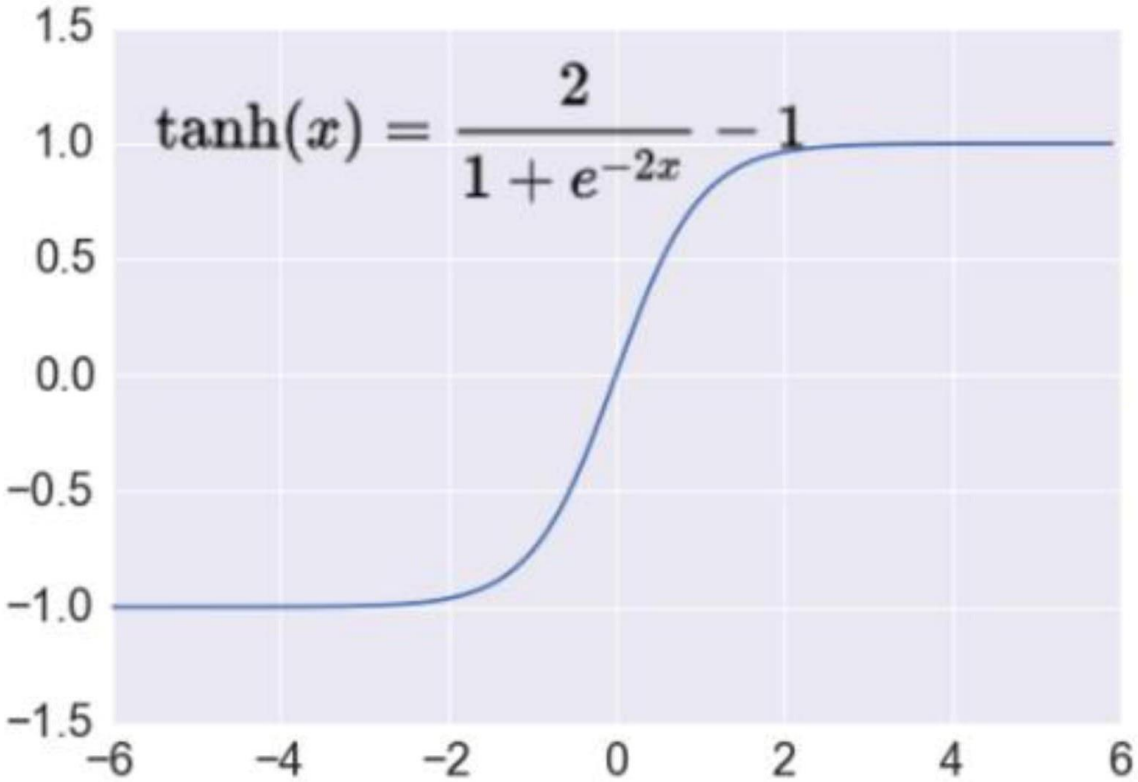
- Sigmoid
- Tanh
- ReLU



- Takes a real-valued number and **squashes** it into a range of 0 to 1
- Sigmoid neurons saturate and kill gradients, thus NN will barely learn

Activation Functions

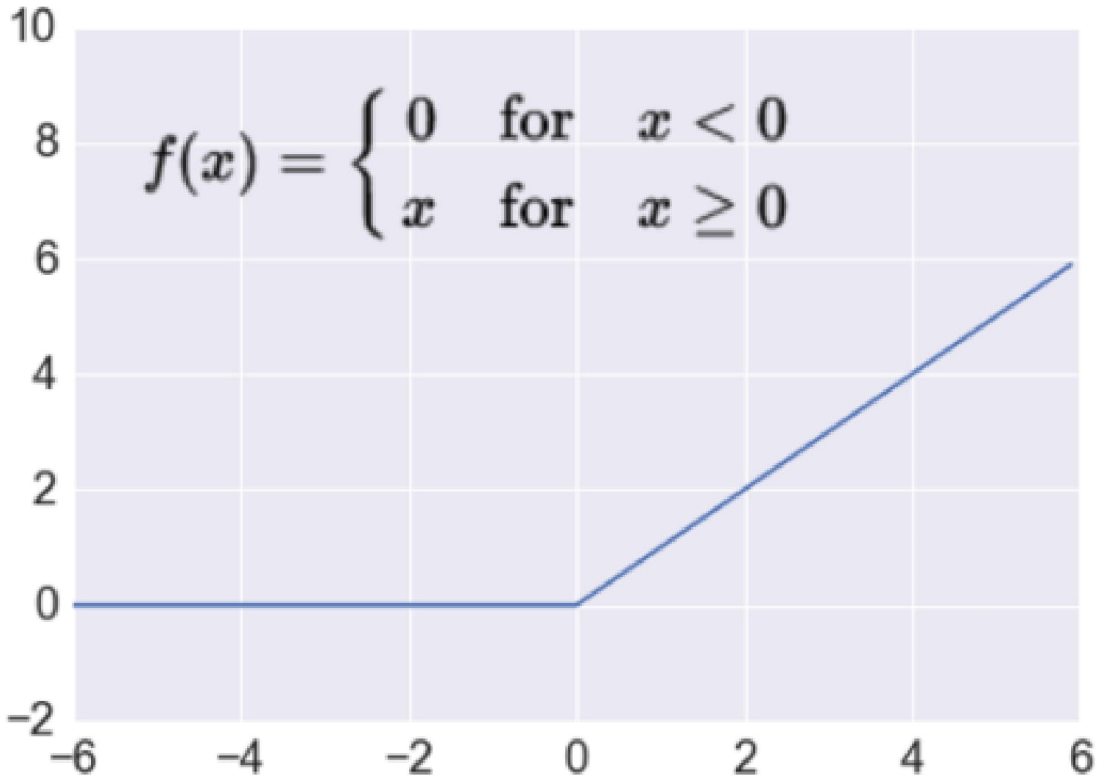
- Sigmoid
- Tanh
- ReLU



- Takes a real-valued number and squashes it into a range of -1 to 1
- Like sigmoid, tanh neurons saturate
- Unlike sigmoid, output is zero-centered

Activation Functions

- Sigmoid
- Tanh
- ReLU



- Takes a real-valued number and thresholds it at zero
- Most deep networks use ReLU nowadays
- Trains much faster
- Prevents the vanishing gradient problem

Backpropagation

Problem Scenario: The backpropagation algorithm plays a key role in training a feedforward artificial neural network. It models a given function by modifying internal weights of input neurons to produce an expected output neuron.

Objective:

Build a neural network which takes **tanh** as the activation function and updates weights with respect to the tanh gradients.

Access: Click the Practice Labs tab on the left panel. Now, click on the START LAB button and wait while the lab prepares itself. Then, click on the LAUNCH LAB button. A full-fledged jupyter lab opens, which you can use for your hands-on practice and projects.

ASSISTED PRACTICE

Activation Function

Problem Scenario: Neural networks are the crux of deep learning, a field which has practical applications in many different areas**.** They become more accurate and effective with multiple layers. The creation of multilayered neural network is not feasible. However, developing the source code for a shallow neural network will help understand the functioning of deep neural networks much better.

Objective: Write a simple neural network in Python considering the activation function as sigmoid.

Access: Click the Practice Labs tab on the left panel. Now, click on the START LAB button and wait while the lab prepares itself. Then, click on the LAUNCH LAB button. A full-fledged jupyter lab opens, which you can use for your hands-on practice and projects.

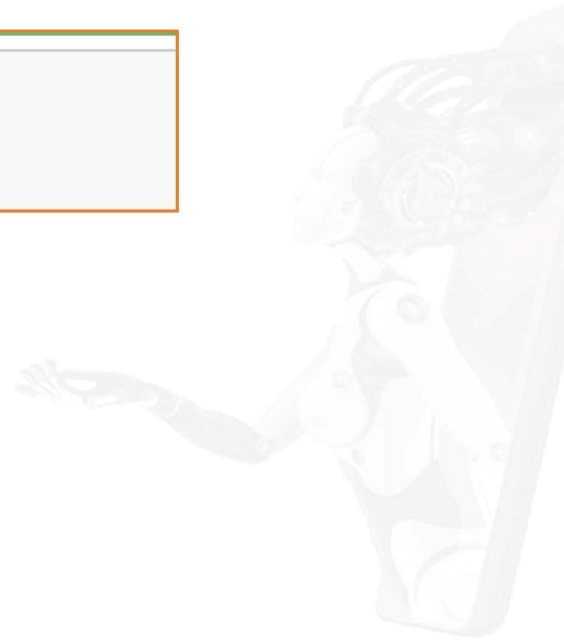
UNASSISTED PRACTICE

Defining Elements

Import the necessary libraries and define a class in the name of NeuralNetwork

```
In [23]: from numpy import exp, array, random, dot

         class NeuralNetwork():
```



Initialize Weights and Assign Activation Functions

```
class NeuralNetwork():
    def __init__(self):
        # Seed the random number generator, so it generates the same numbers
        # every time the program runs.
        random.seed(2)

        # We model a single neuron, with 3 input connections and 1 output connection.
        # We assign random weights to a 3 x 1 matrix, with values in the range -1 to 1
        # and mean 0.
        self.synaptic_weights = 2 * random.random((3, 1)) - 1

    # The Sigmoid function, which describes an S shaped curve.
    # We pass the weighted sum of the inputs through this function to
    # normalise them between 0 and 1.
    def __sigmoid(self, x):
        return 1 / (1 + exp(-x))

    # The derivative of the Sigmoid function.
    # This is the gradient of the Sigmoid curve.
    # It indicates how confident we are about the existing weight.
    def __sigmoid_derivative(self, x):
        return x * (1 - x)
```

Weight Adjustment

```
# We train the neural network through a process of trial and error.
# Adjusting the synaptic weights each time.
def train(self, training_set_inputs, training_set_outputs, number_of_training_iterations):
    for iteration in range(number_of_training_iterations):
        # Pass the training set through our neural network (a single neuron).
        output = self.think(training_set_inputs)

        # Calculate the error (The difference between the desired output
        # and the predicted output).
        error = training_set_outputs - output

        # Multiply the error by the input and again by the gradient of the Sigmoid curve.
        # This means less confident weights are adjusted more.
        # This means inputs, which are zero, do not cause changes to the weights.
        adjustment = dot(training_set_inputs.T, error * self.__sigmoid_derivative(output))

        # Adjust the weights.
        self.synaptic_weights += adjustment
```

Note: The above functions are defined within the class named NeuralNetworks.

Weight Adjustment

```
# We train the neural network through a process of trial and error.
# Adjusting the synaptic weights each time.
def train(self, training_set_inputs, training_set_outputs, number_of_training_iterations):
    for iteration in range(number_of_training_iterations):
        # Pass the training set through our neural network (a single neuron).
        output = self.think(training_set_inputs)

        # Calculate the error (The difference between the desired output
        # and the predicted output).
        error = training_set_outputs - output

        # Multiply the error by the input and again by the gradient of the Sigmoid curve.
        # This means less confident weights are adjusted more.
        # This means inputs, which are zero, do not cause changes to the weights.
        adjustment = dot(training_set_inputs.T, error * self.__sigmoid_derivative(output))

        # Adjust the weights.
        self.synaptic_weights += adjustment
```

Note: The above functions are defined within the class named NeuralNetworks.

Initialize the think Function

```
    # The neural network thinks.
    def think(self, inputs):
        # Pass inputs through our neural network (our single neuron).
        return self.__sigmoid(dot(inputs, self.synaptic_weights))
```

Initialize the Neural Network

```
if __name__ == "__main__":

    #Intialise a single neuron neural network.
    neural_network = NeuralNetwork()

    print ("Random starting synaptic weights: ")
    print (neural_network.synaptic_weights)

    # The training set. We have 4 examples, each consisting of 3 input values
    # and 1 output value.
    training_set_inputs = array([[0, 0, 1], [1, 1, 1], [1, 0, 1], [0, 1, 1]])
    training_set_outputs = array([[0, 1, 1, 0]]).T
```

Train the Neural Network

```
# Train the neural network using a training set.
# Do it 10,000 times and make small adjustments each time.
neural_network.train(training_set_inputs, training_set_outputs, 10000)

print ("New synaptic weights after training: ")
print (neural_network.synaptic_weights)

# Test the neural network with a new situation.
print ("Considering new situation [1, 0, 0] -> ?: ")
print (neural_network.think(array([1, 0, 0])))
```

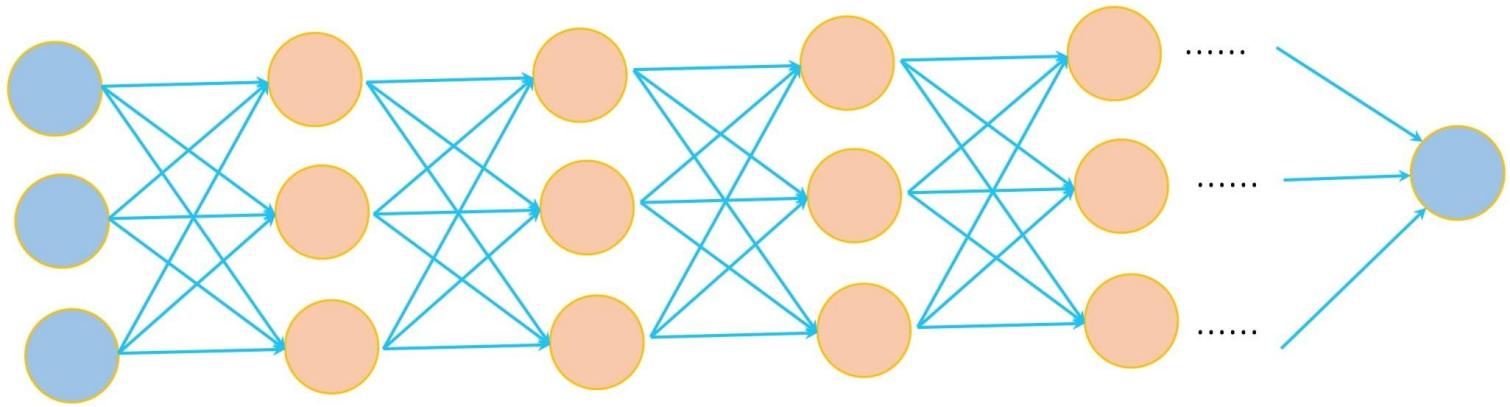
```
Random starting synaptic weights:
[[-0.1280102 ]
 [-0.94814754]
 [ 0.09932496]]
New synaptic weights after training:
[[ 9.67282529]
 [-0.20892653]
 [-4.62890667]]
Considering new situation [1, 0, 0] -> ?:
[0.99993703]
```

DATA AND
ARTIFICIAL INTELLIGENCE

Regularization

Deep Neural Networks

When a neural network contains more than one hidden layer it becomes a Deep Neural Network.

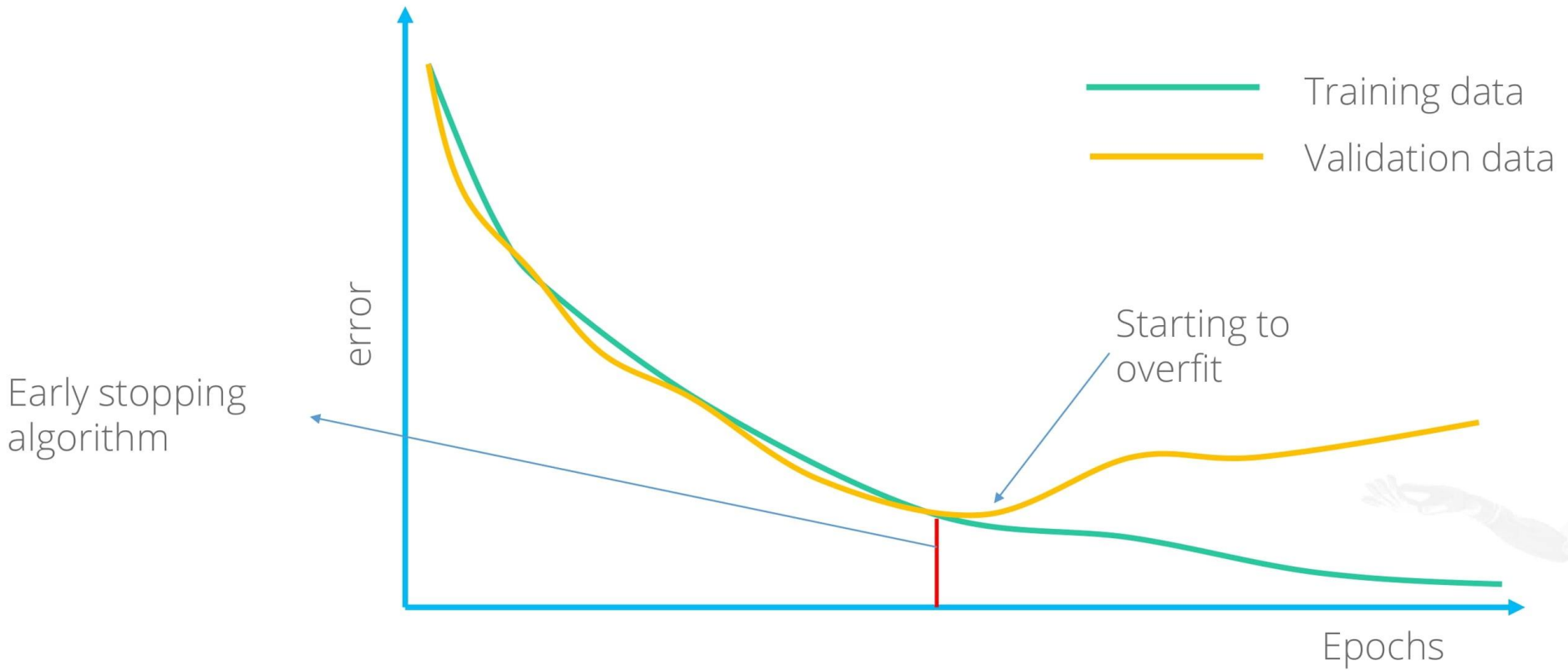


Input layer

Multiple hidden layers

Output layer

The Overfitting Problem



Learned hypothesis may **fit** the training data and the outliers (**noise**) very well but fail to **generalize** test data.

Dealing with the Overfitting Problem

L2 Regularization

Dropout Regularization

- Regularization penalizes big weights, in addition to the overall cost function
- Weight decay value determines how dominant regularization is during gradient computation
- Big weight decay coefficient implies big penalty for big weights

$$C = C_0 + \frac{\lambda}{2n} \sum_w w^2$$

Regularization term

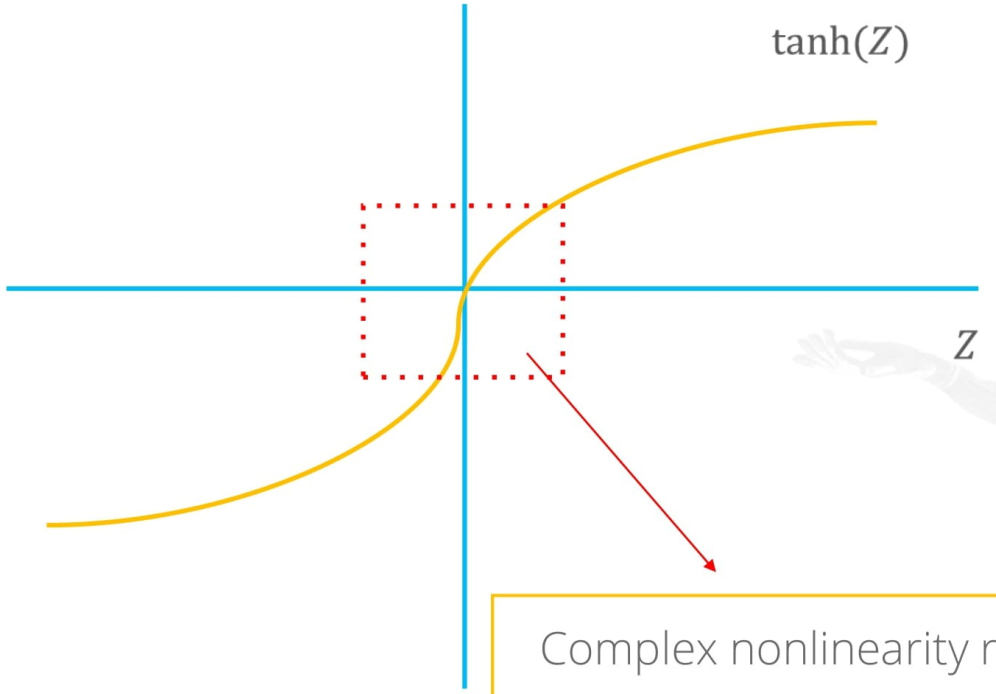
Squared Weights

Dealing with the Overfitting Problem

L2 Regularization

Dropout Regularization

$$C = C_0 + \frac{\lambda}{2n} \sum_w w^2,$$

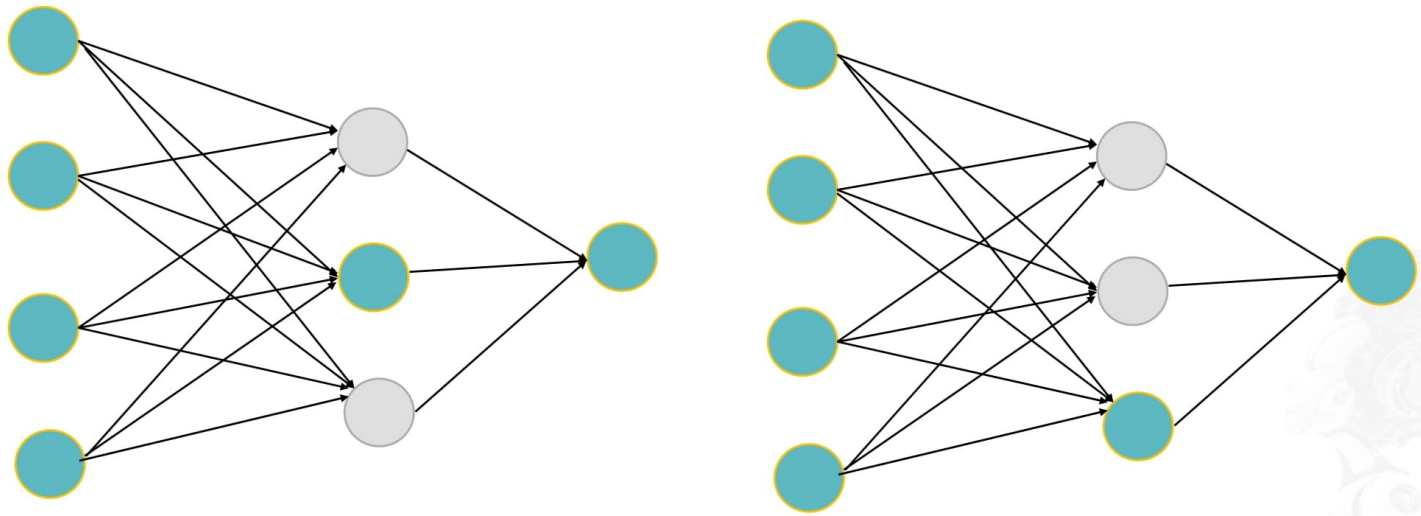


Complex nonlinearity reduced to linear function after application of L2 regularization, thus reducing the complexity due to hidden layers

Dealing with the Overfitting Problem

L2 Regularization

Dropout Regularization

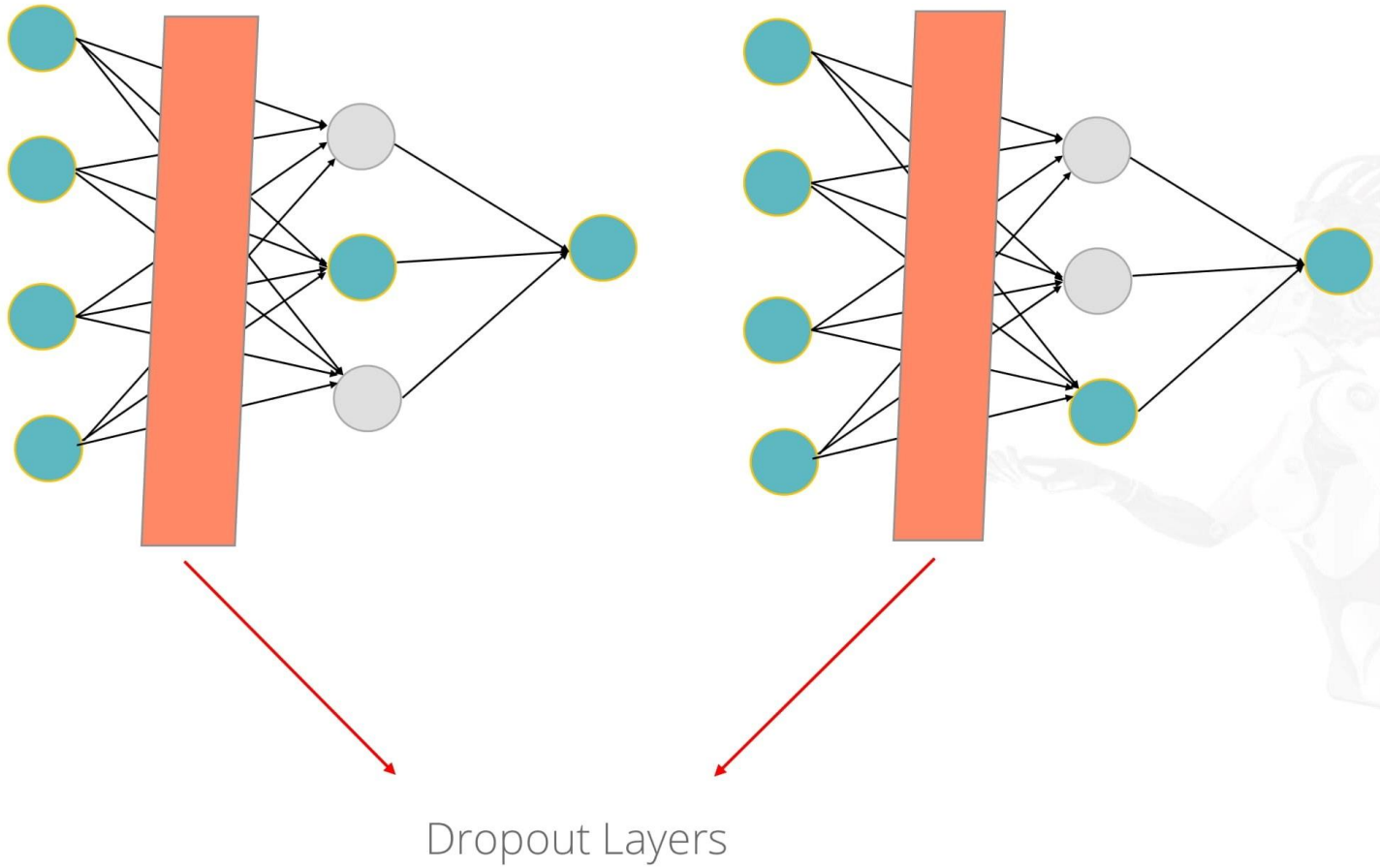


- Randomly drops units (along with their connections) during training
- Each unit is retained with fixed probability p, independent of other units
- $0 < p < 1$
- Hyper-parameter p has to be chosen (tuned)
- While testing the entire network gets activated while the weights get scaled by a factor of p

Dealing with the Overfitting Problem

L2 Regularization

Dropout Regularization



Dropout Layers

Dropout Experiment

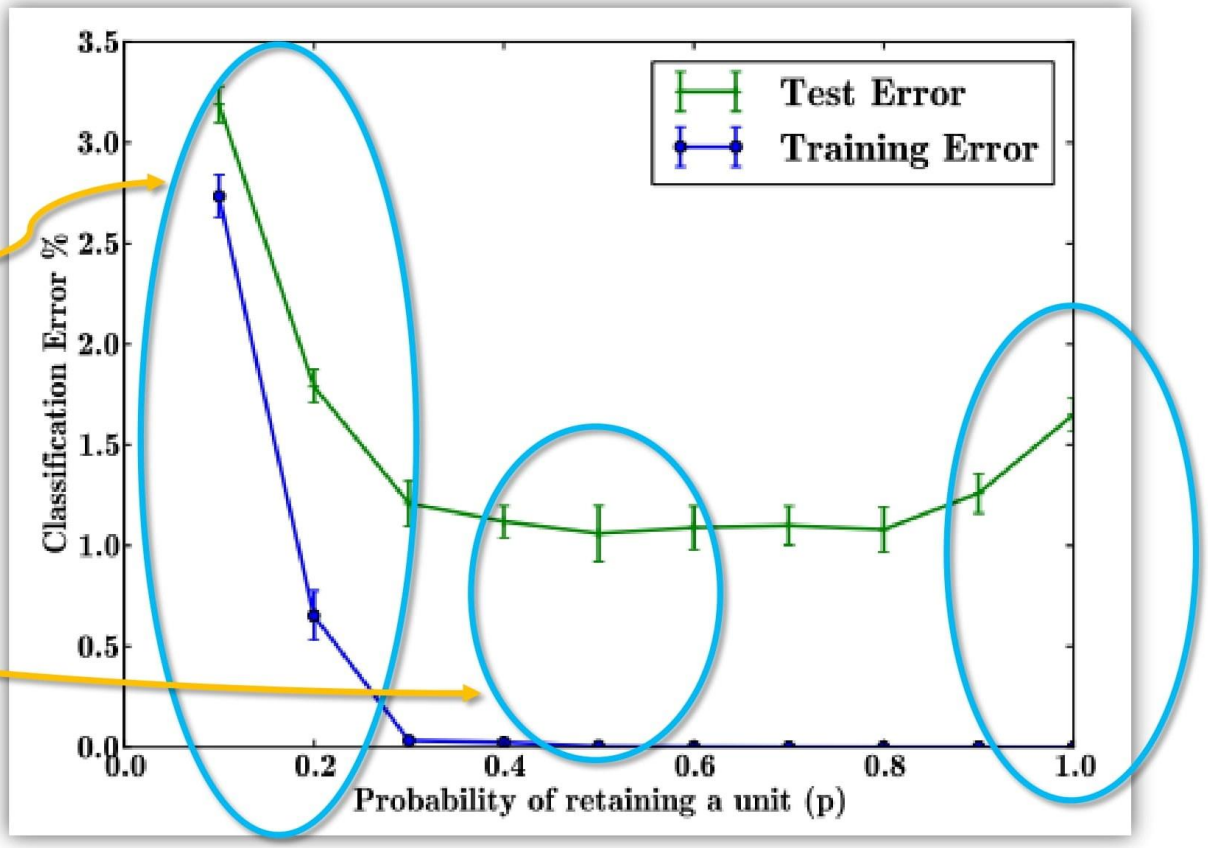
An architecture of 784-2048-2048-2048-10 is used on the MNIST dataset. The dropout rate p was changed from small numbers (most units are dropped out) to 1.0 (no dropout).

High rate of dropout ($p < 0.3$)

- Underfitting
- Very few units are turned on during training

Best dropout rate ($p = 0.5$)

- Training error is low
- Test error is low



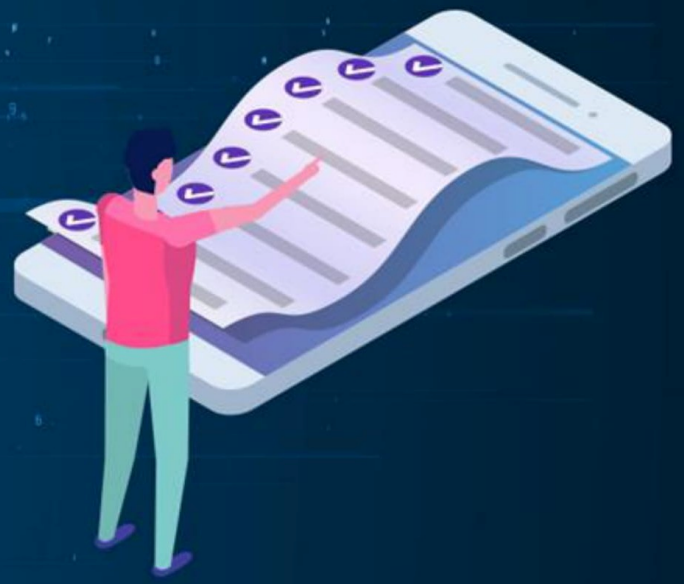
No dropout ($p = 1.0$)

- Training error is low
- Test error is high

Key Takeaways

Now, you are able to:

- Explore neural networks
- Perform weight updation using different activation functions
- Deduce and implement backpropagation algorithm in Python
- Optimize the performance of your neural network using L2 regularization and dropout layers



DATA AND
ARTIFICIAL INTELLIGENCE

Knowledge Check

Knowledge Check 1

After the perceptron algorithm finishes training, how can the learned weights be expressed in terms of the initial weight vector and the input vectors?

a. It requires one bit per data point

b. It requires one integer per data point

c. It requires one real number per data point

d. It is impossible

Knowledge Check 1

After the perceptron algorithm finishes training, how can the learned weights be expressed in terms of the initial weight vector and the input vectors?

a. It requires one bit per data point

b. It requires one integer per data point

c. It requires one real number per data point

d. It is impossible

The correct answer is **b**

During the perceptron training algorithm, the weights are updated by adding or subtracting the input vector. Moreover, this might happen multiple times for the same input vector. Therefore, the weight vector = initial vector + c1 * data point 1 + c2 * data point 2 + ... cn * data point n, where c_i = number of times data point i was added - number of times data point i was subtracted

Knowledge Check 2

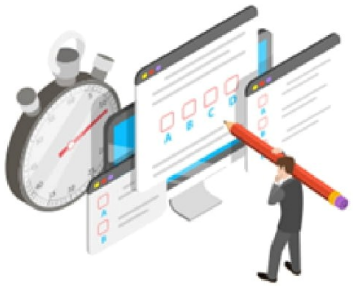
Which of the following techniques performs similar operations as dropout in a neural network?

a. Bagging

b. Boosting

c. Stacking

d. None of these



Knowledge Check 2

Which of the following techniques performs similar operations as dropout in a neural network?

a. Bagging

b. Boosting

c. Stacking

d. None of these

The correct answer is **a**

Dropout can be seen as an extreme form of bagging in which each model is trained on a single case and each parameter of the model is very strongly regularized by sharing it with the corresponding parameter in all the other models

MNIST Image Classification

Problem Scenario: The MNIST dataset is widely used for image classification. However, while validating the same, researchers found out that the classification model was overfitting as it was not giving acceptable accuracy on the testing data.

Use the mnist_test.csv and mnist_train.csv for model optimization (using dropout layers). Also, you will have to use one hot encoding for training and testing labels.

Objective:

Optimize a neural network based classification model using dropout regularization such that the p value is 0.70 for input and hidden layers.

Access: Click the Practice Labs tab on the left panel. Now, click on the START LAB button and wait while the lab prepares itself. Then, click on the LAUNCH LAB button. A full-fledged jupyter lab opens, which you can use for your hands-on practice and projects.

LESSON-END PROJECT

DATA AND
ARTIFICIAL INTELLIGENCE

Thank You