Artificial Intelligence

Deep Dive into Neural Networks

Agenda

01 Limitations of a Single-layer Perceptron

02 Use Case 1

03 Feedforward Neural Network

04 Multi-layer Perceptron

05 Use Case 2

06 Backpropagation Algorithm

07 Gradient Descent

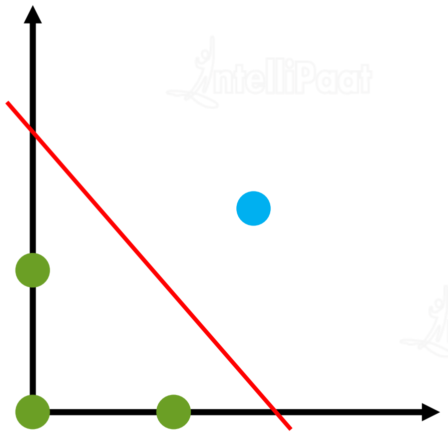
08 Stochastic Gradient Descent

09 Adam Optimization Algorithm

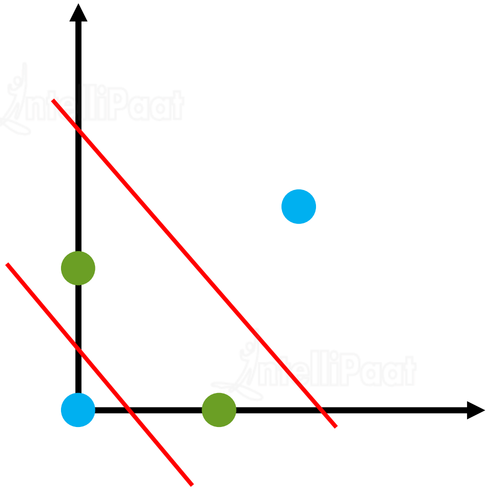
10 Demo

Limitations of a Single-layer Perceptron

- A single-layer perceptron can only learn linearly separable problems
- If the problem is not linearly separable, the learning process of a perceptron will never reach a point where all points are classified correctly
- Boolean ‘AND’ and ‘OR’ functions are linearly separable, whereas Boolean ‘XOR’ is not



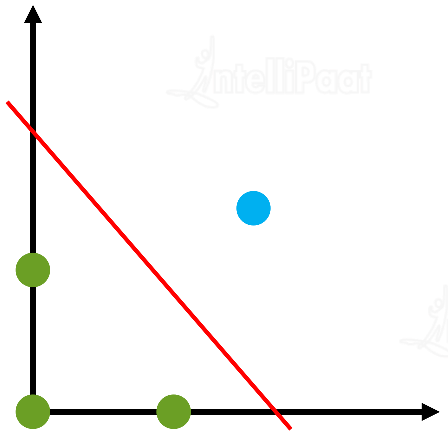
Boolean ‘AND’: Linearly Separable



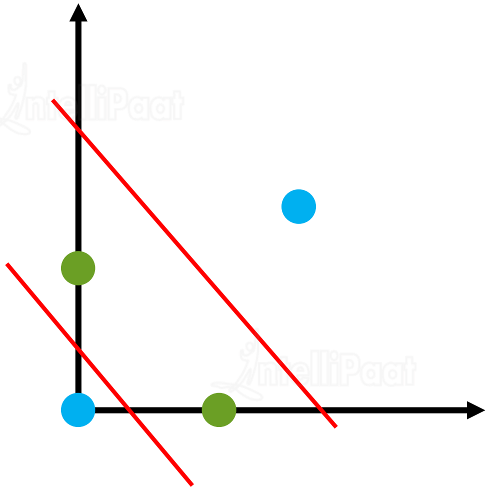
Boolean ‘XOR’: Non-Linearly Separable

Limitations of a Single-layer Perceptron

- A single-layer perceptron can only learn linearly separable problems
- If the problem is not linearly separable, the learning process of a perceptron will never reach a point where all points are classified correctly
- Boolean ‘AND’ and ‘OR’ functions are linearly separable, whereas Boolean ‘XOR’ is not



Boolean ‘AND’: Linearly Separable



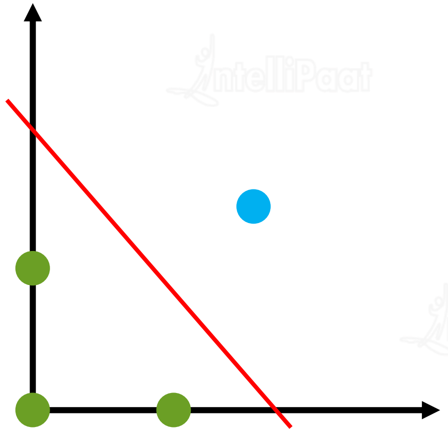
Boolean ‘XOR’: Non-Linearly Separable

For solving this problem, we can use a multi-layer perceptron

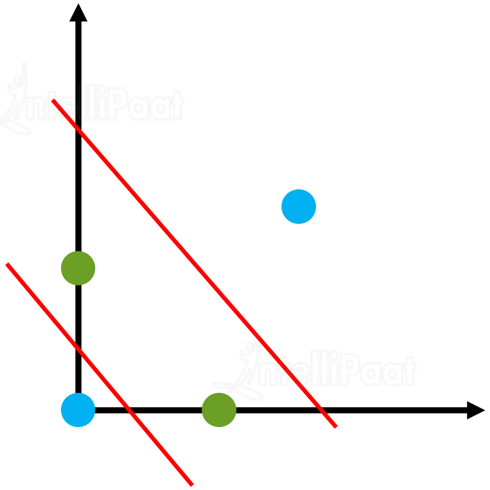
Limitations of a Single-layer Perceptron

For solving this problem, we can use a multi-layer perceptron

- A single-layer perceptron won't be able to solve complex problems such as *Image Classification*
- In such kinds of problems, the *dimensionality* and *complexity* of the classification is very high



Boolean 'AND': Linearly Separable



Boolean 'XOR': Non-Linearly Separable

Let us see some real-life problems which cannot be solved by single-layer perceptron

Use Case 1

Complex problems that involve a lot of parameters cannot be solved by a single-layer perceptron

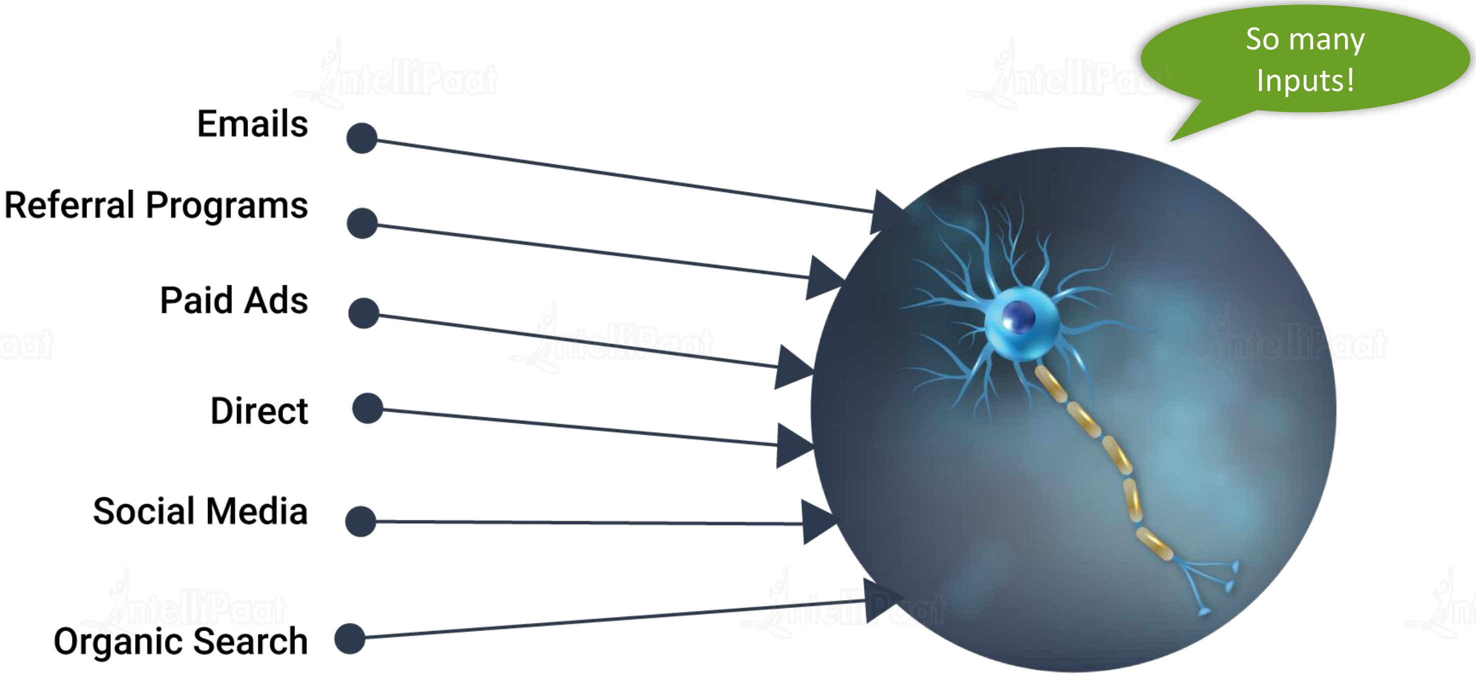
- Consider a case where you own an e-commerce firm. You have planned to increase traffic on your site by providing a special discount on the products and services. Now, you want to create awareness among people regarding this end-season sale by marketing on different portals like:
  - Google ads
  - Personal emails
  - Sales advertisements on relevant sites
  - YouTube ads
  - Ads on different sites
  - linkedin
  - Blogs and so on
- This task is too complex for a human to analyze, as you can see that the number of parameters is quite high
- Let us try to solve it using Deep Learning

Use Case 1

- You can either use just one platform for publicity or use a variety of them
- Each of them has its own advantages and disadvantages, but lots of factors would have to be considered
- The increased traffic on your portal or the number of sales that would happen is dependent on different categorical inputs, their sub-categories, and their parameters

Computing and calculating profit in terms of popularity and sales, from so many inputs and their sub-categories, is not possible just through one perceptron

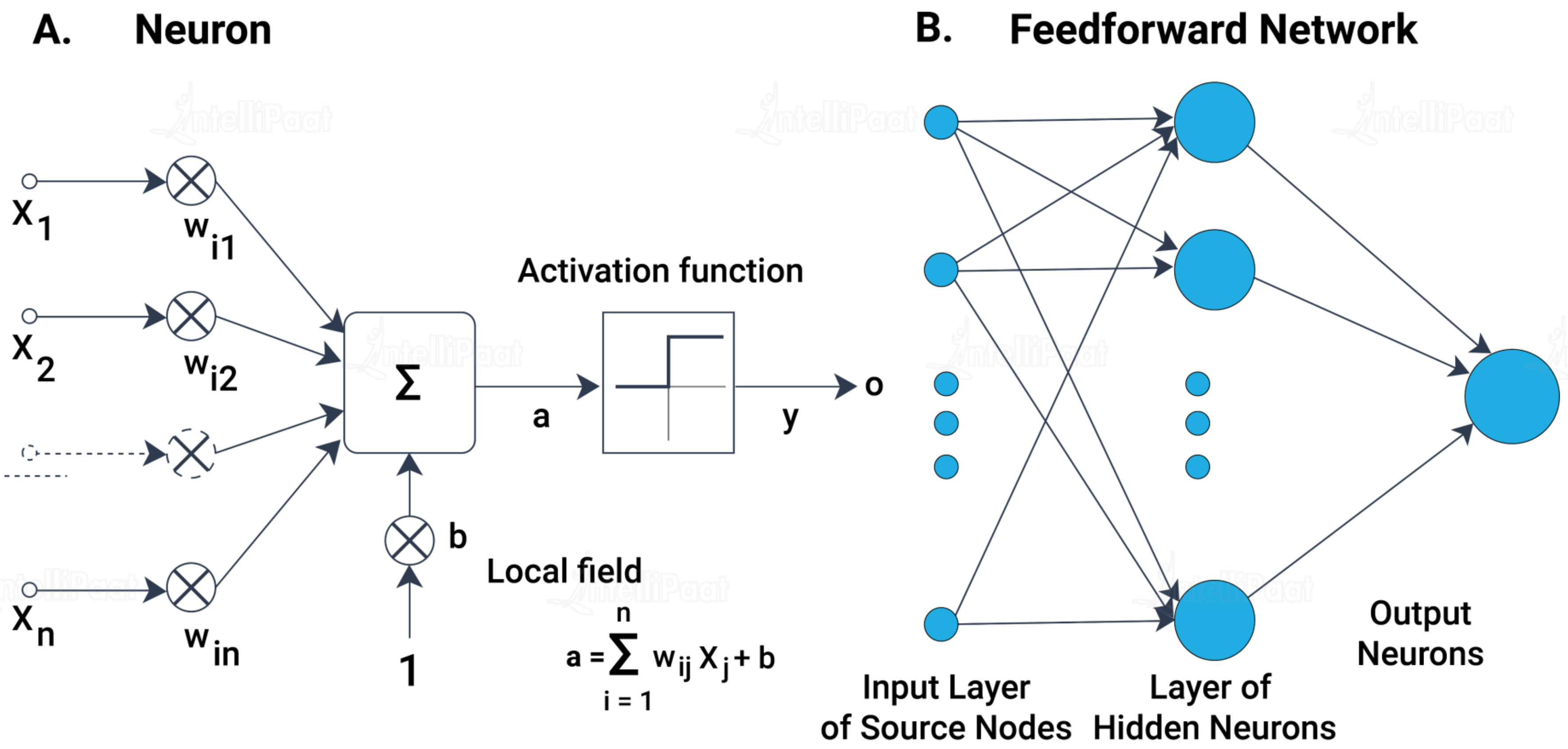

So now, you know why a single perceptron cannot be used for complex non-linear problems


Emails
Referral Programs
Paid Ads
Direct
Social Media
Organic Search
So many Inputs!

Before getting into the actual solution to our problem, let us recall one of the previously discussed topics: Feedforward Neural Network

Feedforward Neural Network

Feedforward neural network is the most simple artificial neural network containing multiple nodes arranged in multiple layers. Adjacent layer nodes have connections or edges where all connections are weighted



Feedforward Neural Network

Feedforward neural network is the most simple artificial neural network containing multiple nodes arranged in multiple layers. Adjacent layer nodes have connections or edges where all connections are weighted

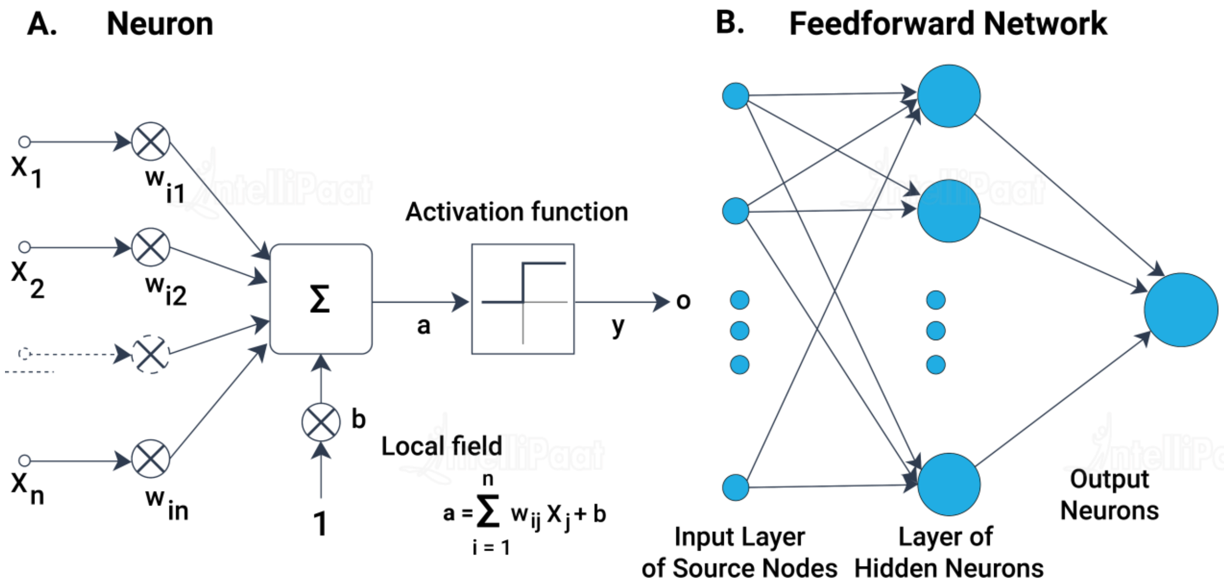
"A feedforward neural network can contain two kinds of nodes"

Monolayer

This is the simplest feedforward neural network that does not contain any hidden layers

Multi-layer Perceptron

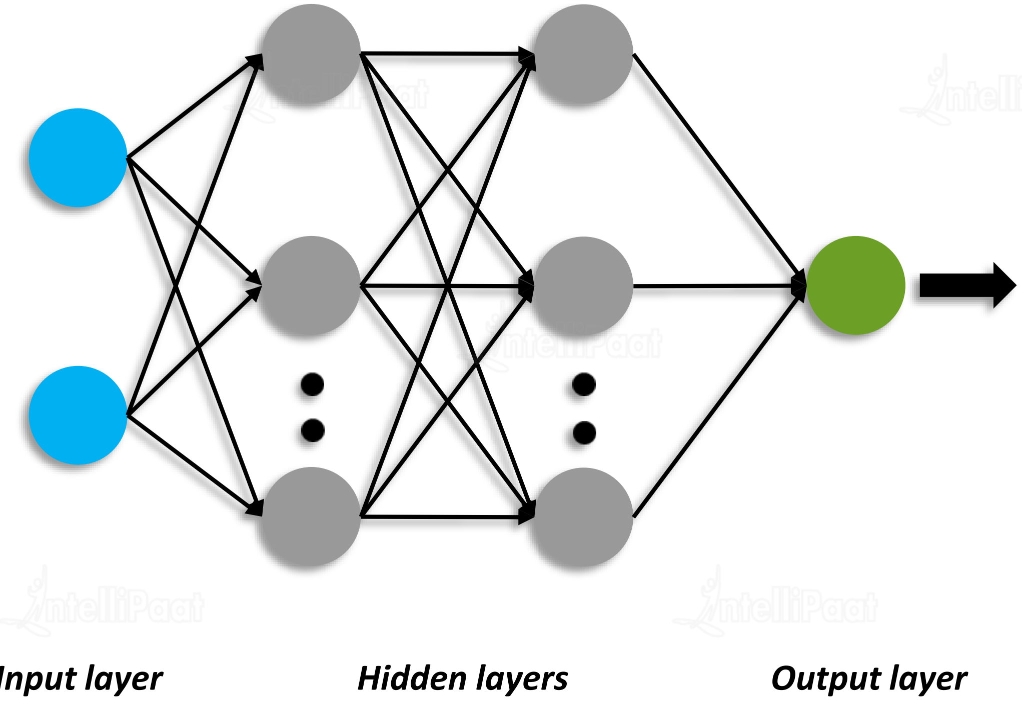
Multi-layer Perceptron (MLP) includes at least one hidden layer (except for one input layer and one output layer)



Let us now discuss the ultimate solution to our previous problem, i.e., MLP

Multi-layer Perceptron

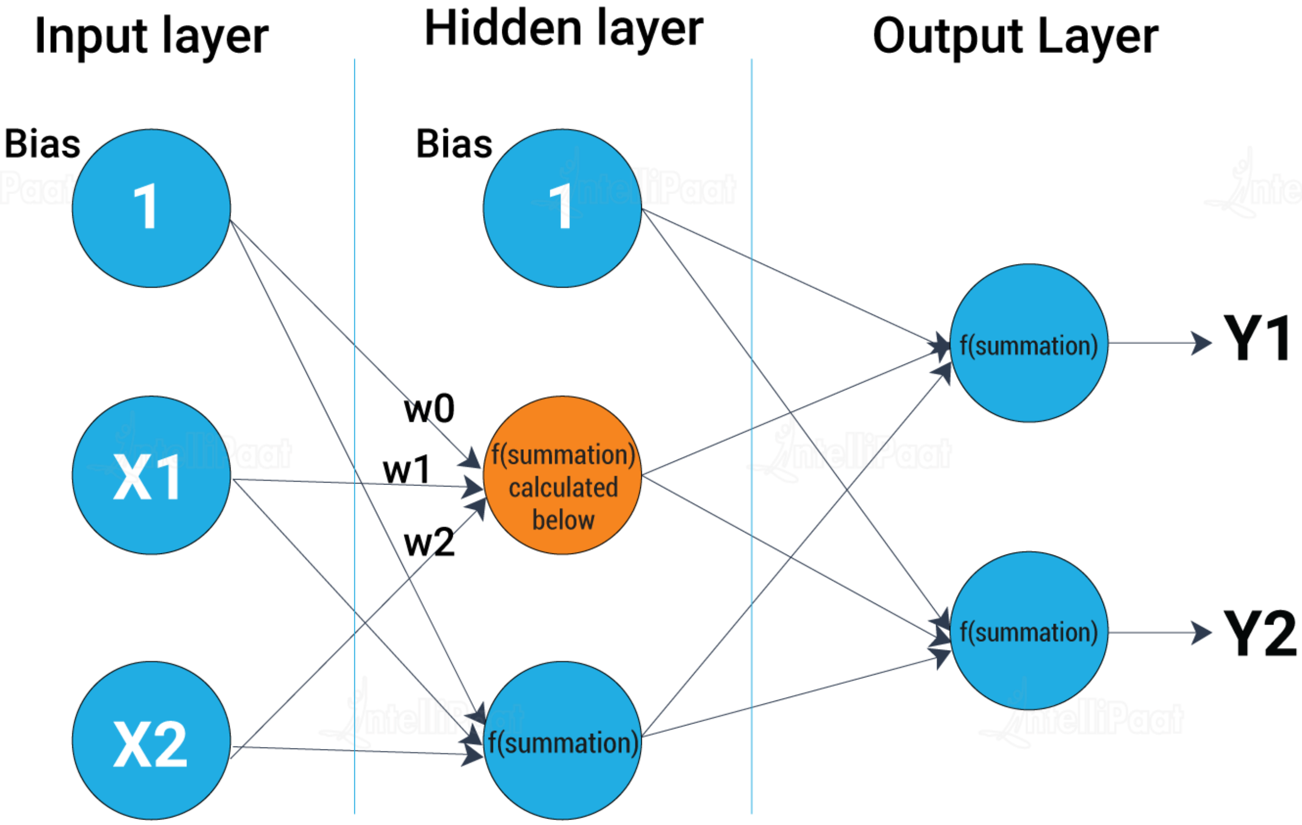
- A multi-layer perceptron (MLP) is a deep, artificial neural network
- It is composed of more than one perceptrons
- An MLP is comprised of:
  - An *input layer* to receive the signal
  - An *output layer* that makes a decision or prediction about the input
  - An *arbitrary number of hidden layers*
- Each node, apart from the input nodes, has a nonlinear activation function
- An MLP uses backpropagation as a supervised learning technique



MLP is widely used for solving problems that require supervised learning and research into computational neuroscience and parallel distributed processing. Such applications include speech recognition, image recognition, and machine translation

Multi-layer Perceptron

The figure shows a multi-layer perceptron with a single hidden layer. All connections have weights associated with them, but only three weights (w0, w1, and w2) are shown in the figure

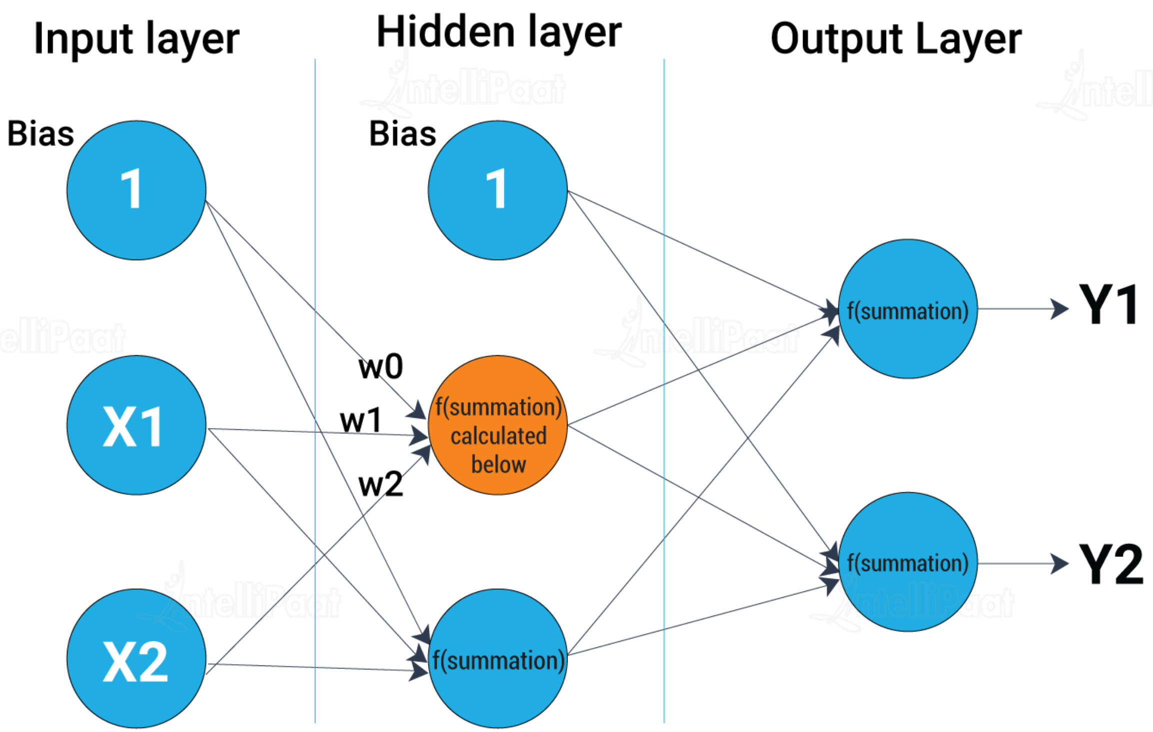


Output from the highlighted Neuron = f(summation) = f(w0. 1 + w1 . X1 + w2 . X2)

Multi-layer Perceptron

Input Layer:

- It has three nodes
- Bias (offset) node has a value of 1
- The other two nodes take X1 and X2 as external inputs
- Outputs from nodes in the input layer are 1, X1, and X2, respectively, which are fed into the hidden layer

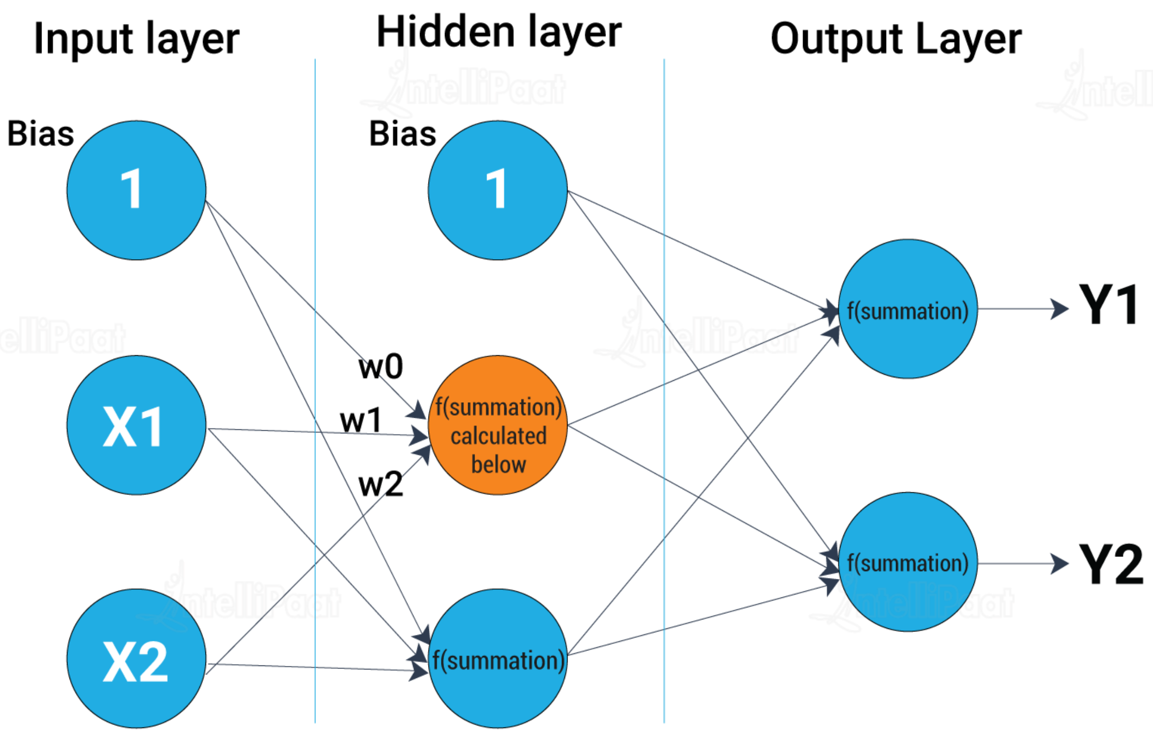


Output from the highlighted Neuron = f(summation) = f(w0. 1 + w1 . X1 + w2 . X2)

Multi-layer Perceptron

Hidden Layer:

- It also has three nodes with the Bias node having an output of 1
- The output of the other two nodes in the hidden layer depends on the outputs from the input layer (1, X1, and X2) as well as the weights associated with the connections (edges)
- The figure shows the output calculation for one of the hidden nodes
- Similarly, the output from the other hidden node can be calculated
- Here, ‘f’ refers to the activation function. These outputs are then fed to the nodes in the output layer

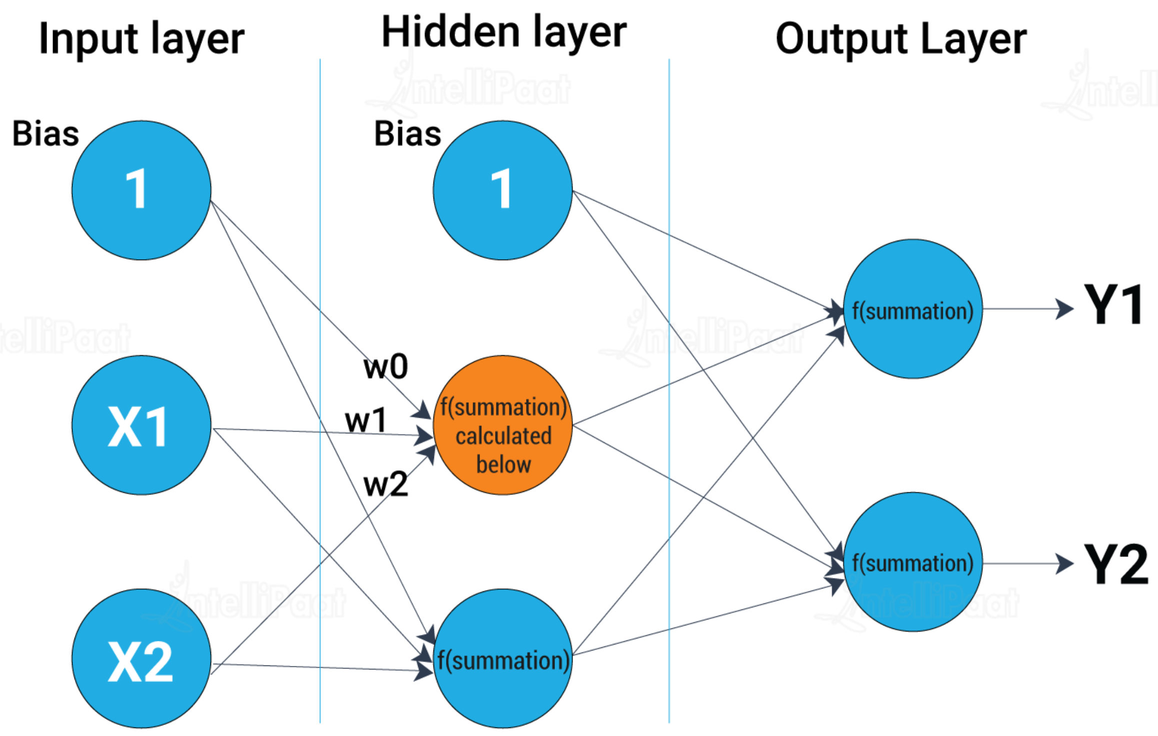


Output from the highlighted Neuron = f(summation) = f(w0. 1 + w1 . X1 + w2 . X2)

Multi-layer Perceptron

Output Layer:

- The output layer has two nodes which take inputs from the hidden layer and perform similar computations as shown for the highlighted hidden node
- Values calculated (Y1 and Y2) as a result of these computations act as outputs of the multi-layer perceptron

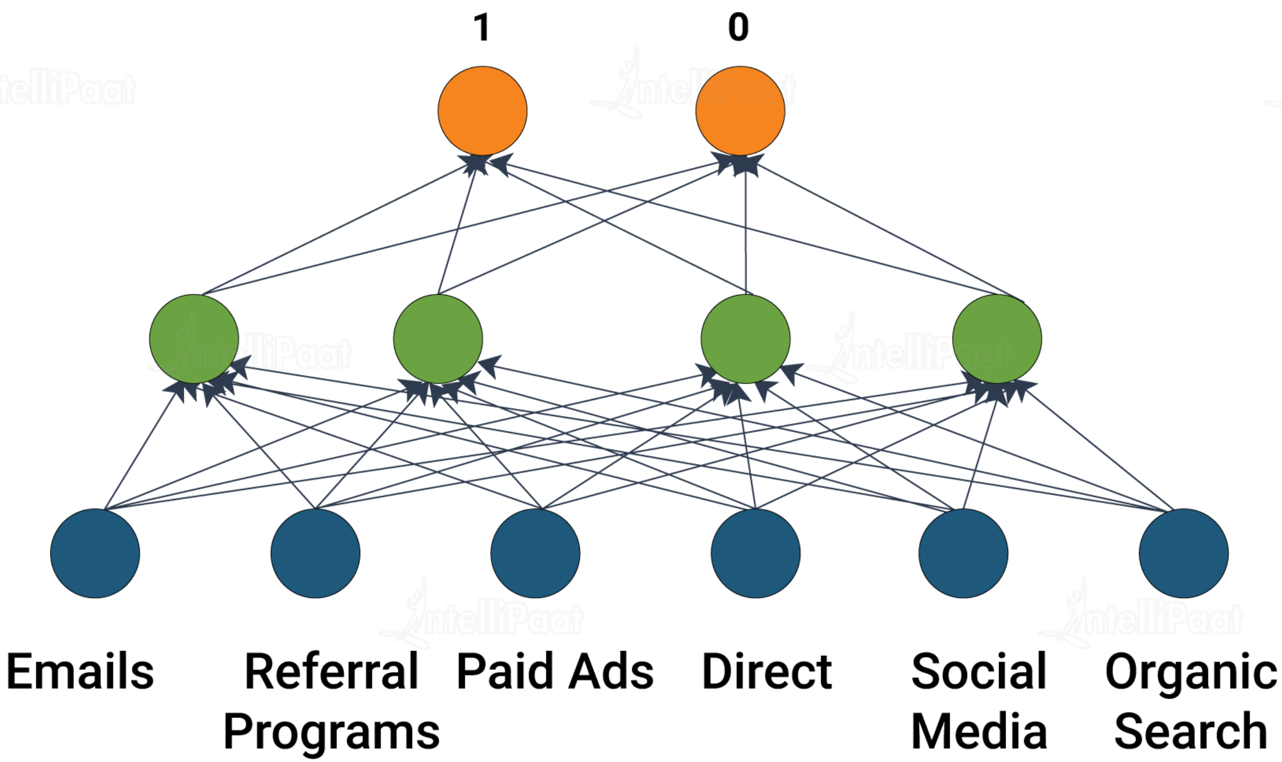


Output from the highlighted Neuron = f(summation) = f(w0. 1 + w1 . X1 + w2 . X2)

Let us now see how MLP helps us in providing a solution to our Use Case 1

Use Case 1: Solution

- Every source behaves as an input to the neural network
- Once all sources are fed into the system, the neural network calculates the output after the computation is done



Lets take another example to understand a multi-layer perceptron better!

Use Case 2

Suppose, we have the following student-marks dataset

Hours Studied	Mid-term Marks	Final Results
35	67	1
12	75	0
16	89	1
45	56	1
10	90	0

- The two input columns show the number of hours each student has studied and the mid-term marks obtained by the student, respectively
- The Final Results column can have two values 1 or 0 indicating whether the student passed (1) in the final term or failed (0)

Now, suppose, we want to predict whether a student studying 25 hours and having 70 marks in the mid term will pass the final term

Use Case 2

Hours Studied	Mid-term Marks	Final Results
25	70	?

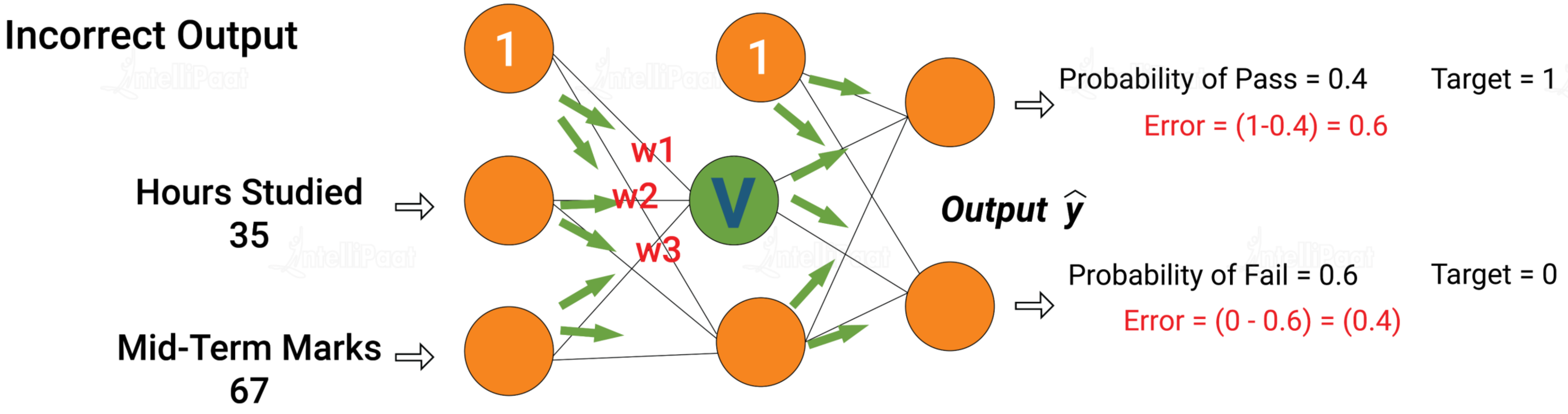
This is a binary classification problem where a multi-layer perceptron can learn from the given examples (the training data) and make an informed prediction when given a new data point. We will see now, how a multi-layer perceptron learns such relationships

The process by which a multi-layer perceptron learns

is called the *Backpropagation algorithm.*

We will discuss this in details after completing MLP!

Use Case 2: Solution

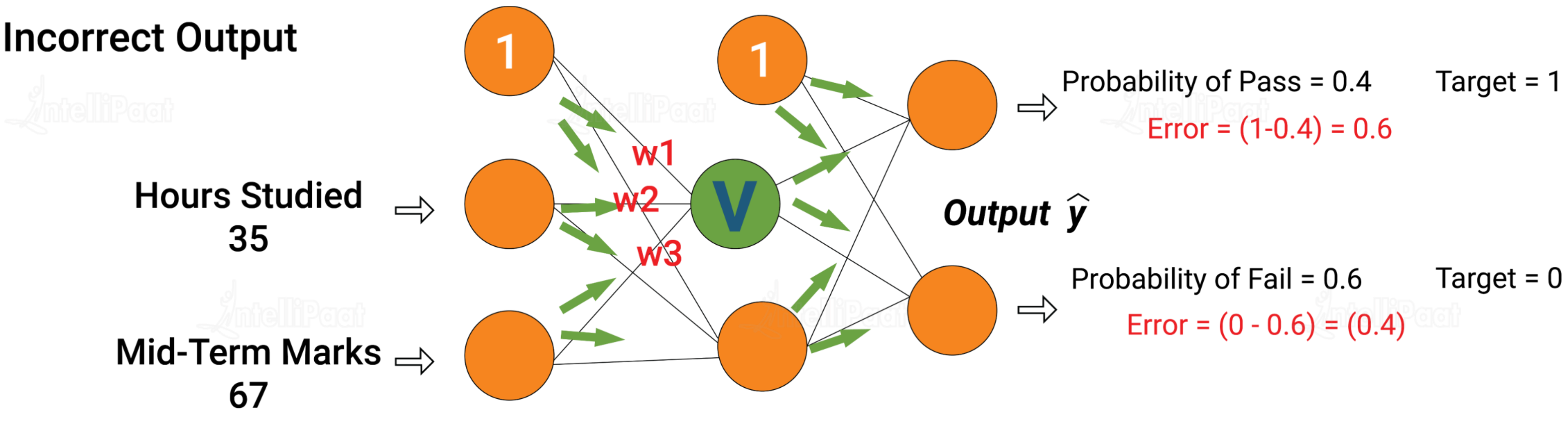


- The figure has two nodes in the input layer (apart from the Bias node) which take the inputs *Hours Studied* and *Mid-term Marks*
- It also has a hidden layer with two nodes (apart from the Bias node)
- The output layer has two nodes as well: the upper node outputs the *probability of 'Pass'* while the lower node outputs the *probability of 'Fail'*

In classification, we generally use a Softmax function as the activation function to ensure that the outputs are probabilities and they add up to 1. So, in this case,

Probability (Pass) + Probability (Fail) = 1

Use Case 2: Solution

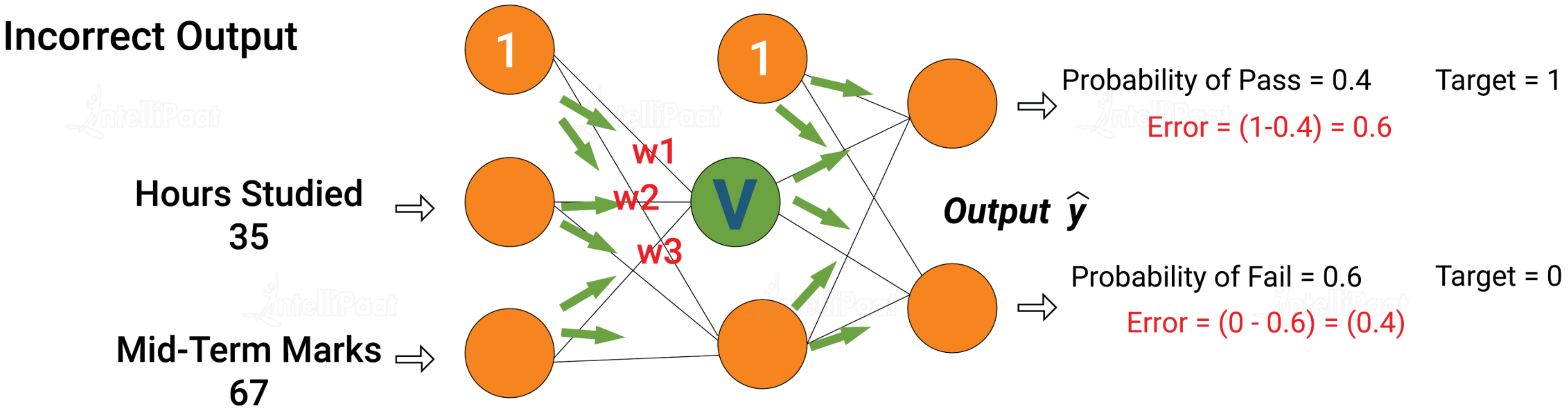


- **Step 1:** ***Forward Propagation***
  - Let's consider the hidden layer node, marked V, in the figure
  - Assume that the **weights** of the connections from the inputs to that node are **w1**, **w2**, and **w3** (as shown)
  - The first training example as input:
    - Input to the network = **[35, 67]**
    - Desired output from the network (target) = **[1, 0]**
    - The output V from the node can be calculated as follows (where 'f' is an activation function): **V = f (1*w1 + 35*w2 + 67*w3)**

Suppose, the output probabilities from the two nodes in the output layer are 0.4 and 0.6, respectively (since the weights are randomly assigned, outputs will also be random)

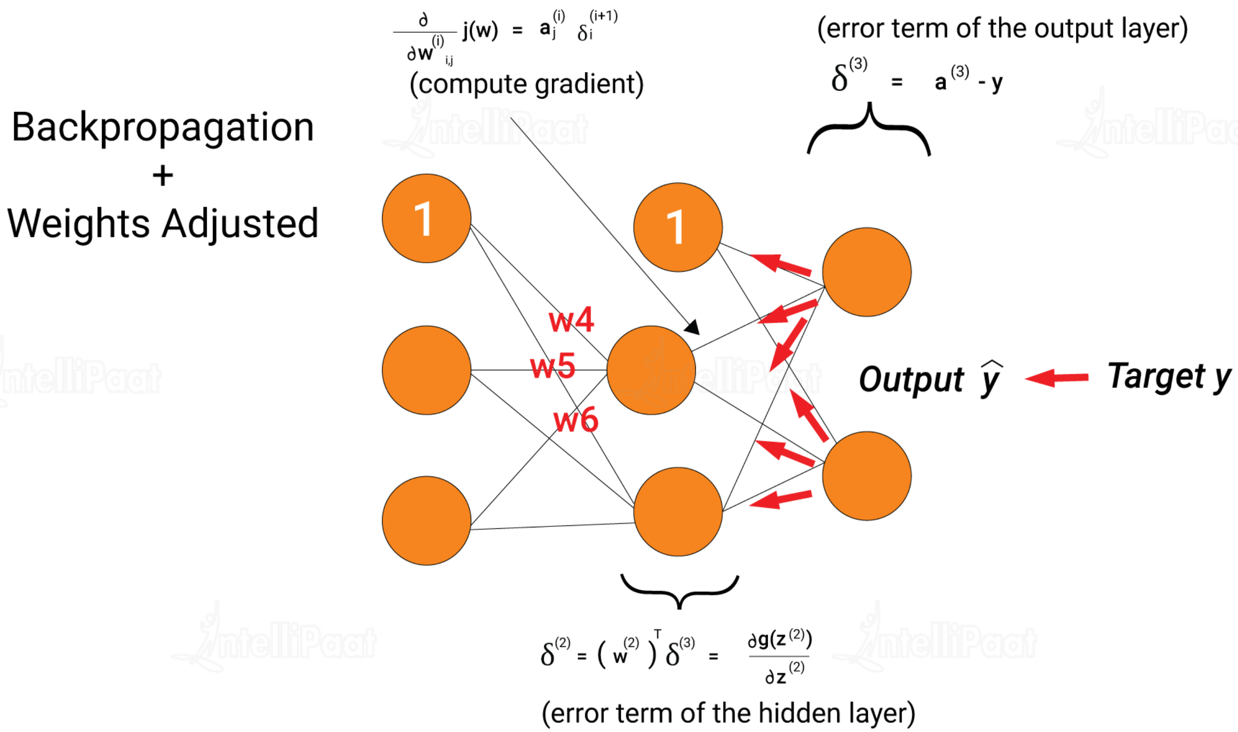
We can see that the calculated probabilities (0.4 and 0.6) are very far from the desired probabilities (1 and 0, respectively); hence, the network in the figure is said to have an ‘Incorrect Output’

Use Case 2: Solution



- **Step 2:** ***Backpropagation and Weight Updates***
  - We calculate the total errors at the output nodes and propagate these errors back through the network using backpropagation to calculate the gradients
  - Then, we use an optimization method such as gradient descent to 'adjust' all weights in the network with an aim of reducing errors at the output layer
  - This is shown in the next figure

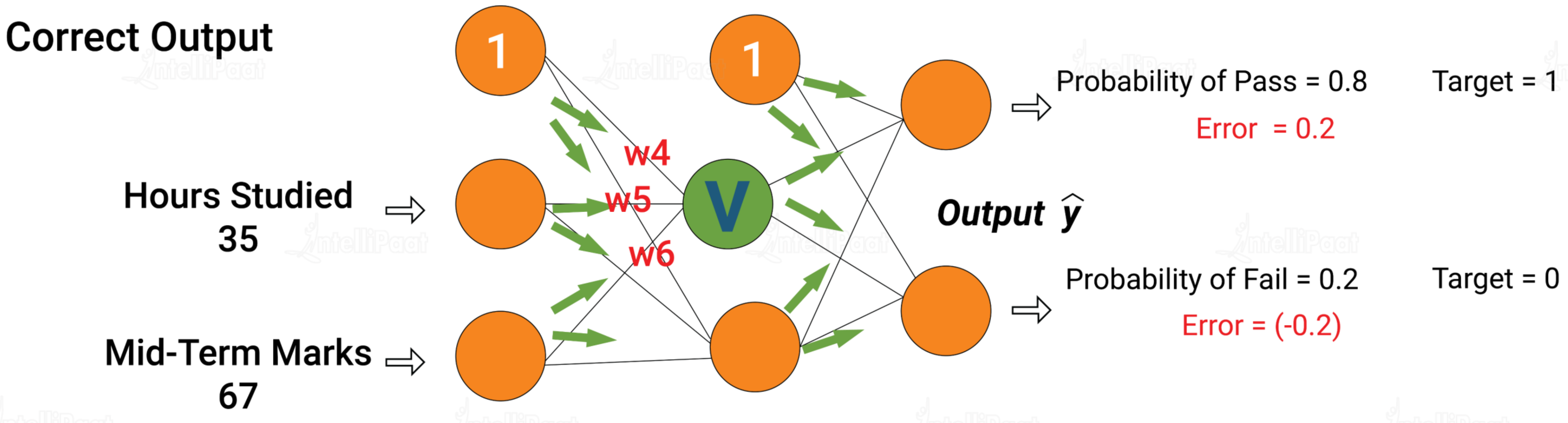
Use Case 2: Solution



- **Step 2:** ***Backpropagation and Weight Updates***
  - Suppose that the new weights associated with the node in consideration are w4, w5, and w6 (after backpropagation and adjusting weights)

If we now input the same example to the network again, the network should perform better than before since the weights have now been adjusted to minimize errors in prediction

Use Case 2: Solution



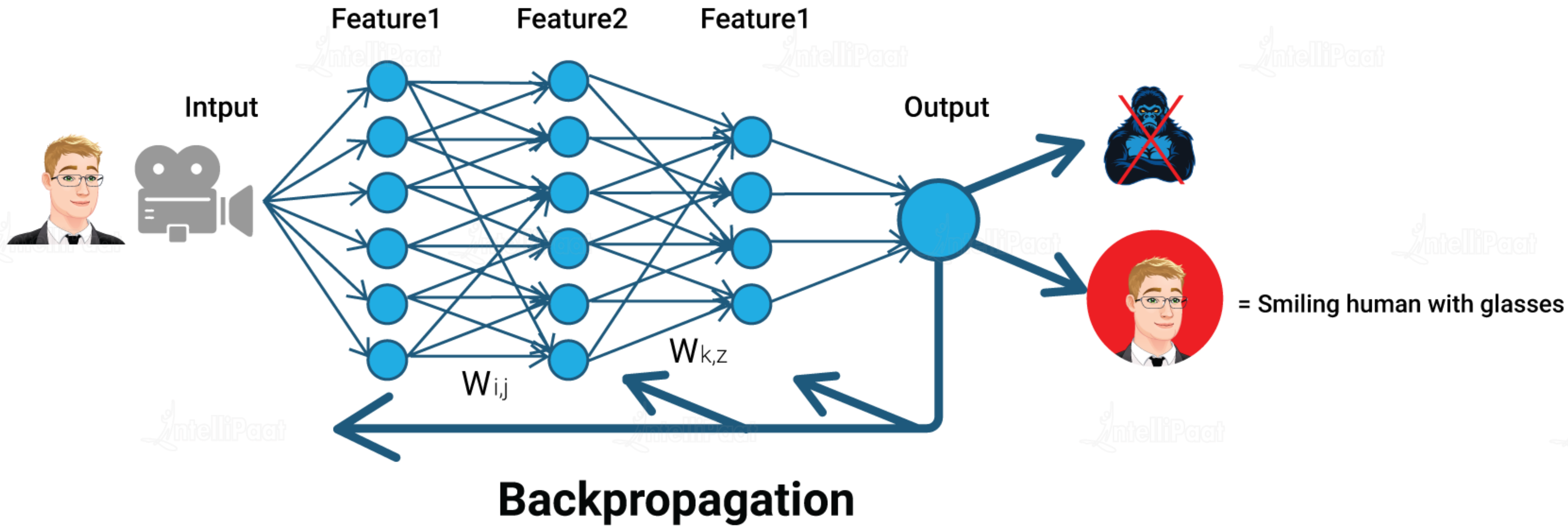
- As shown in Figure, errors at the output nodes have now reduced to [0.2, -0.2] as compared to [0.6, -0.4] earlier
- This means that our network has learned to correctly classify our first training example
- We repeat this process with all other training examples in our dataset. Then, our network will learn those examples as well

If we now want to predict whether a student studying 25 hours and having 70 marks in the mid term will pass the final term, we go through the forward propagation step and find the output probabilities for Pass and Fail

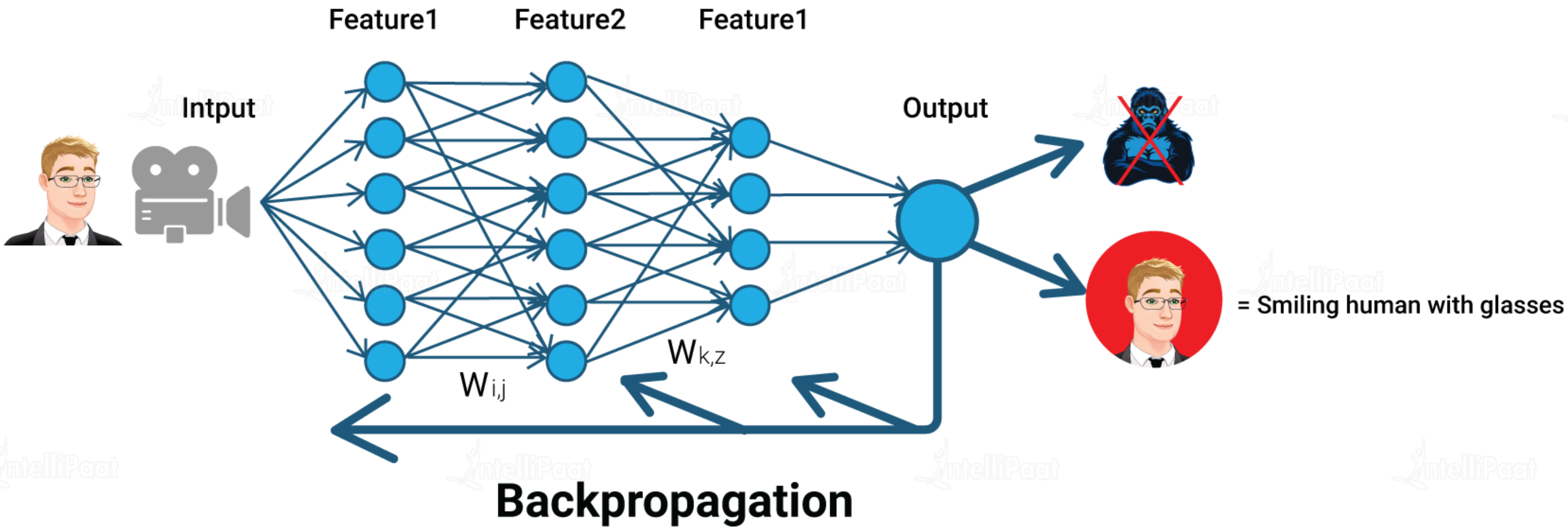
So now, let us understand backpropagation in detail as you have already heard a lot about it!

Backpropagation Algorithm

The backpropagation algorithm is a supervised learning method for multi-layer feedforward networks from the field of Artificial Neural Networks



Backpropagation Algorithm



- The principle of this approach is to model a given function by modifying internal weightings of input signals to produce an expected output signal
- The system is trained using a supervised learning method, where the error between the system's output and a known expected output is presented to the system and used to modify its internal state

Let us understand its working with the help of an example!

Backpropagation Algorithm: How Does it Work?

Consider the following table

Input	Desired Output
0	0
1	2
2	4

Backpropagation Algorithm: How Does it Work?

Consider the initial value of weight as 3

Input	Desired Output	Model Output (W=3)
0	0	0
1	2	3
2	4	6

Backpropagation Algorithm: How Does it Work?

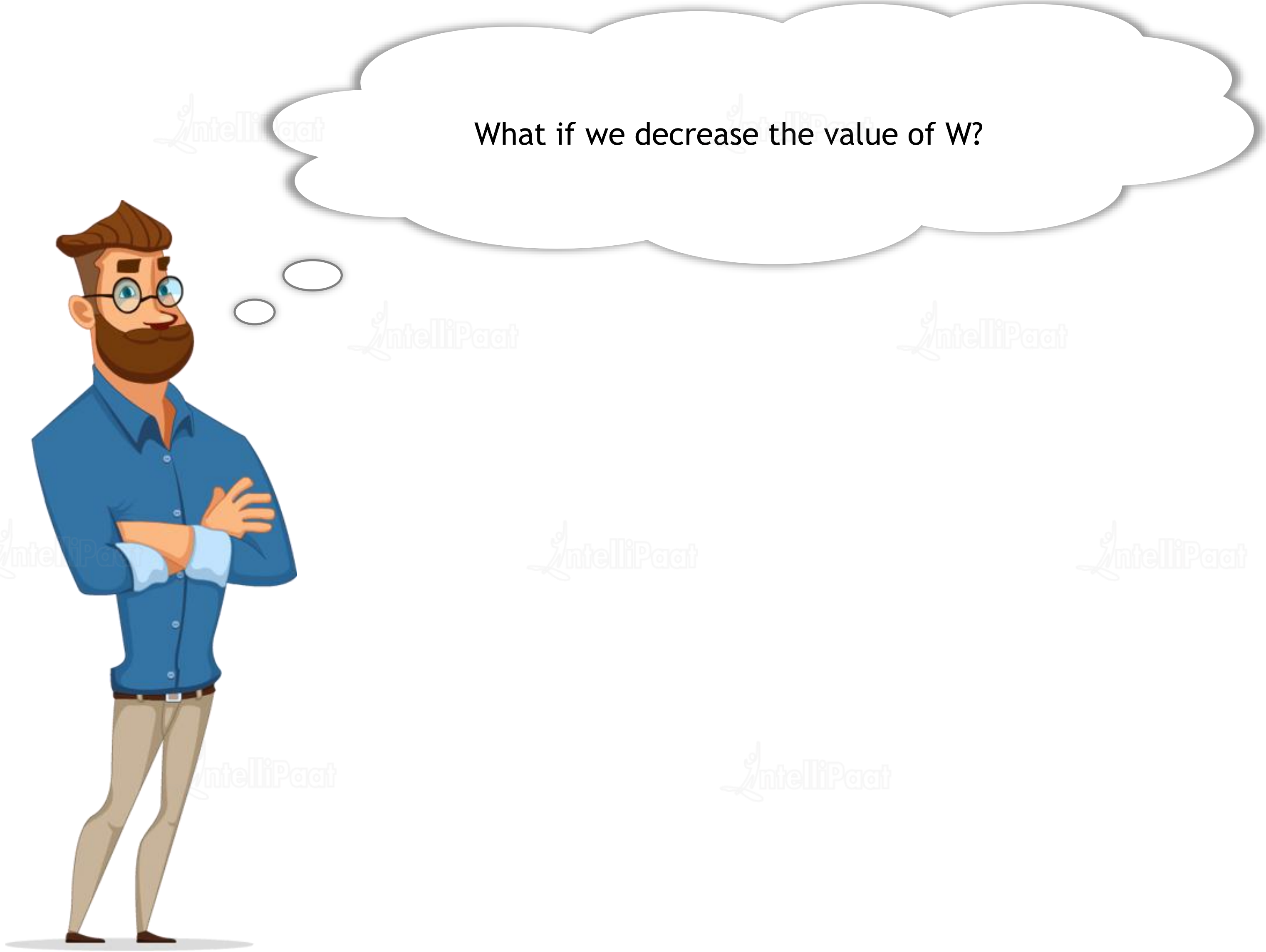
Observe the difference between the actual output and the desired output

Input	Desired Output	Model Output (W=3)	Absolute Error	Square Error
0	0	0	0	0
1	2	3	1	1
2	4	6	2	4

Backpropagation Algorithm: How Does it Work?

Observe the error when changing the value of W to 4

Input	Desired Output	Model Output (W=3)	Absolute Error	Square Error	Model Output (W=4)	Square Error
0	0	0	0	0	0	0
1	2	3	1	1	4	4
2	4	6	2	4	8	16


What if we decrease the value of W?

Backpropagation Algorithm: How Does it Work?

Consider the value of weight as 2

Input	Desired Output	Model Output (W=3)	Absolute Error	Square Error	Model Output (W=2)	Square Error
0	0	0	0	0	0	0
1	2	3	1	1	3	1
2	4	6	2	4	4	0

Backpropagation Algorithm: How Does it Work?

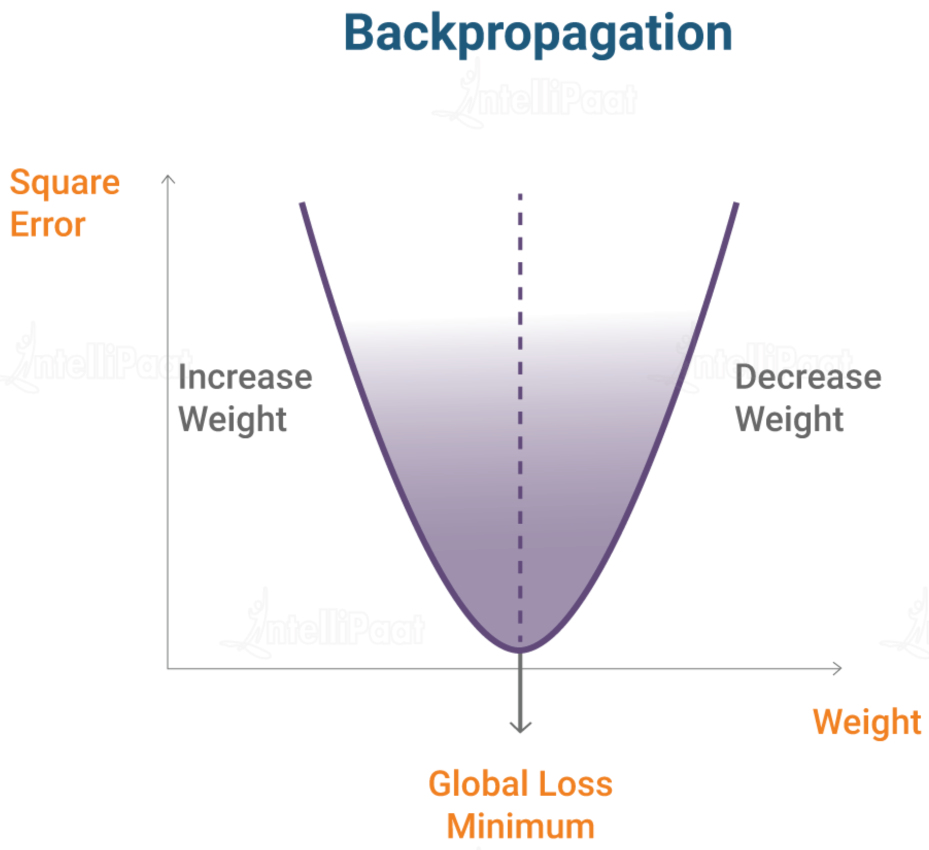
Consider the value of weight as 2

Input	Desired Output	Model Output (W=3)	Absolute Error	Square Error	Model Output (W=2)	Square Error
0	0	0	0	0	0	0
1	2	3	1	1	3	1
2	4	6	2	4	4	0

We see that when the weight is reduced, the error also decreases

Backpropagation Algorithm: How Does it Work?

Consider the following graph



Backpropagation Algorithm: How Does it Work?

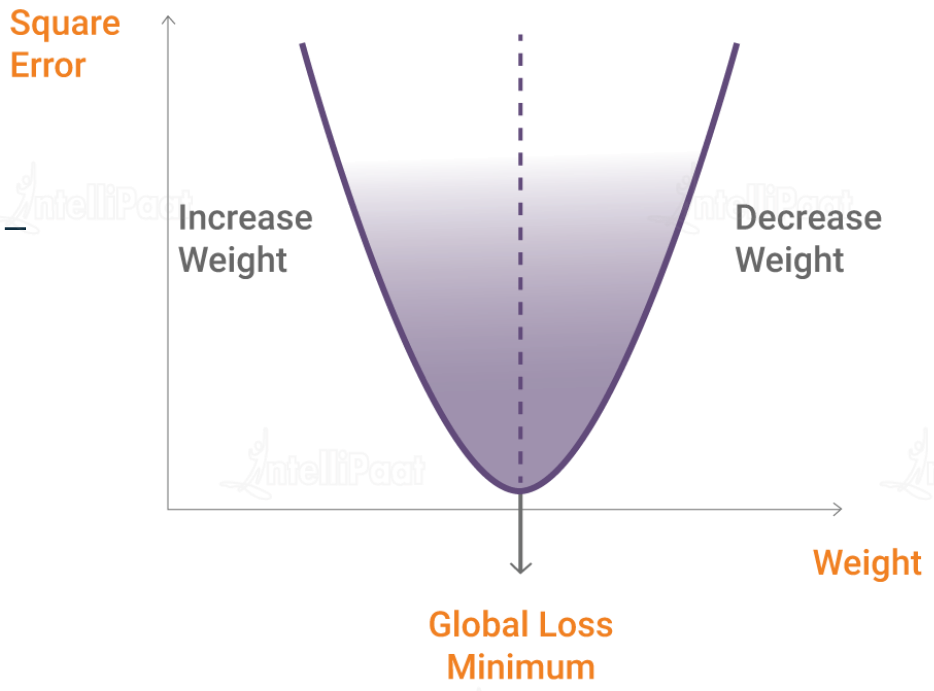
We need to reach the *Global Loss Minimum*. This is nothing but backpropagation

Backpropagation

When the gradient is negative,

increase in weight

leads to decrease in error



When the gradient is positive,

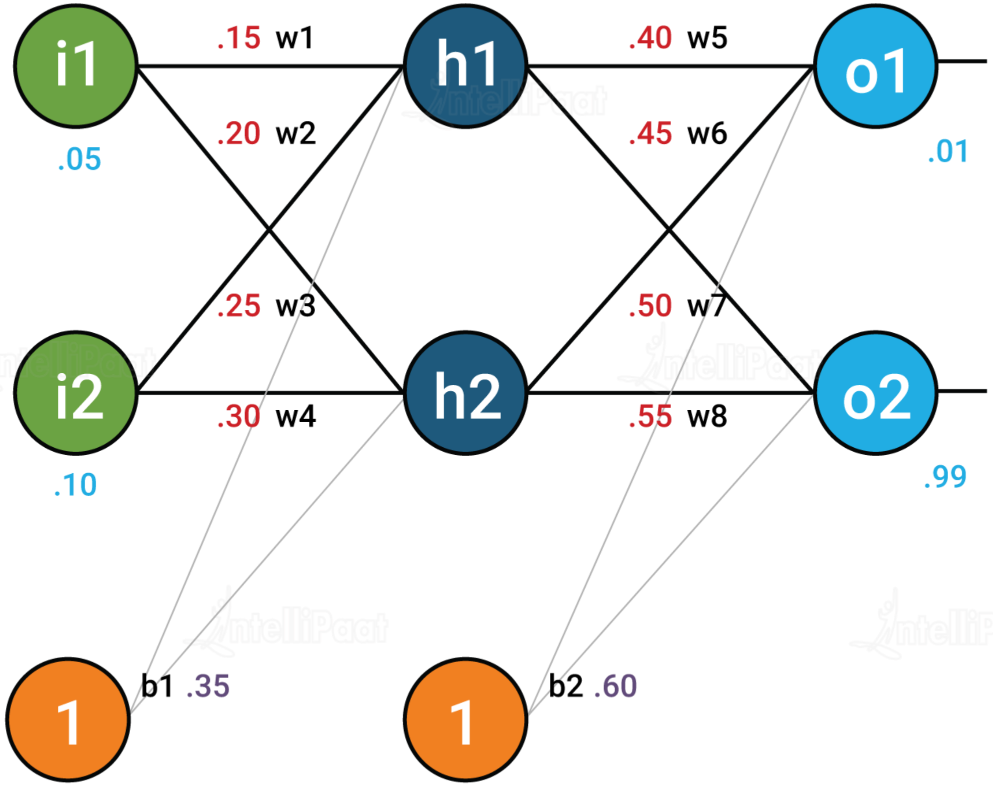
decrease in weight

leads to decrease in error

Let us now understand the math behind backpropagation

Backpropagation Algorithm: How Does it Work?

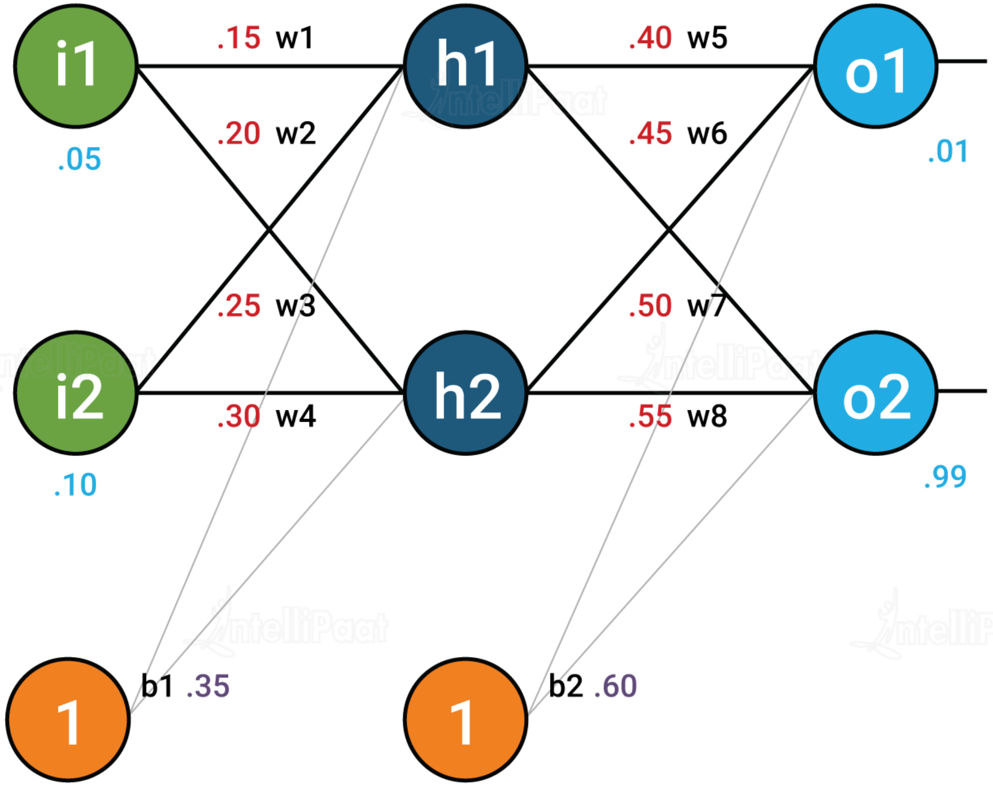
In order to have some numbers to work with, here are *initial weights*, *biases*, and *training inputs/outputs*



- The goal of backpropagation is to optimize the weights so that the neural network can learn how to correctly map arbitrary inputs to outputs
- 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

Backpropagation Algorithm: How Does it Work?

Steps Involved in Backpropagation



Step 1: The Forward Pass

Step 2: The Backward Pass

Backpropagation Algorithm: How Does it Work?

Step 1: The Forward Pass

The total net input for h1:

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

The output for h1:

out h1 = $1/(1 + e^{-net\ h1}) = 1/(1 + e^{-0.3775}) = 0.593269992$

Carrying out the same process for h2:

out h2 = 0.596884378

We repeat this process for the output layer neurons, using the output from the hidden layer neurons as their input

Backpropagation Algorithm: How Does it Work?

Step 1: The Forward Pass

The output for o1:

$$net\ o1 = w_5 * out\ h1 + w_6 * out\ h2 + b_2 * 1$$

$$net\ o1 = 0.4 * 0.593269992 + 0.45 * 0.596884378 + 0.6 * 1 = 1.105905967$$

$$out\ o1 = 1/(1 + e^{-net\ o1}) = 1/(1 + e^{-1.105905967}) = 0.75136507$$

Carrying out the same process for o2:

$$out\ o2 = 0.772928465$$

Backpropagation Algorithm: How Does it Work?

Calculating the Total Error

We can now calculate the error for each output neuron using the squared error function and sum them to get the total error: $E_{total} = \Sigma 1/2(target - output)^2$

The target output for o1 is 0.01, but the neural network output is 0.75136507; therefore, its error is:

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

By repeating this process for o2 (remembering that the target is 0.99), we get:

$E_{o2} = 0.023560026$

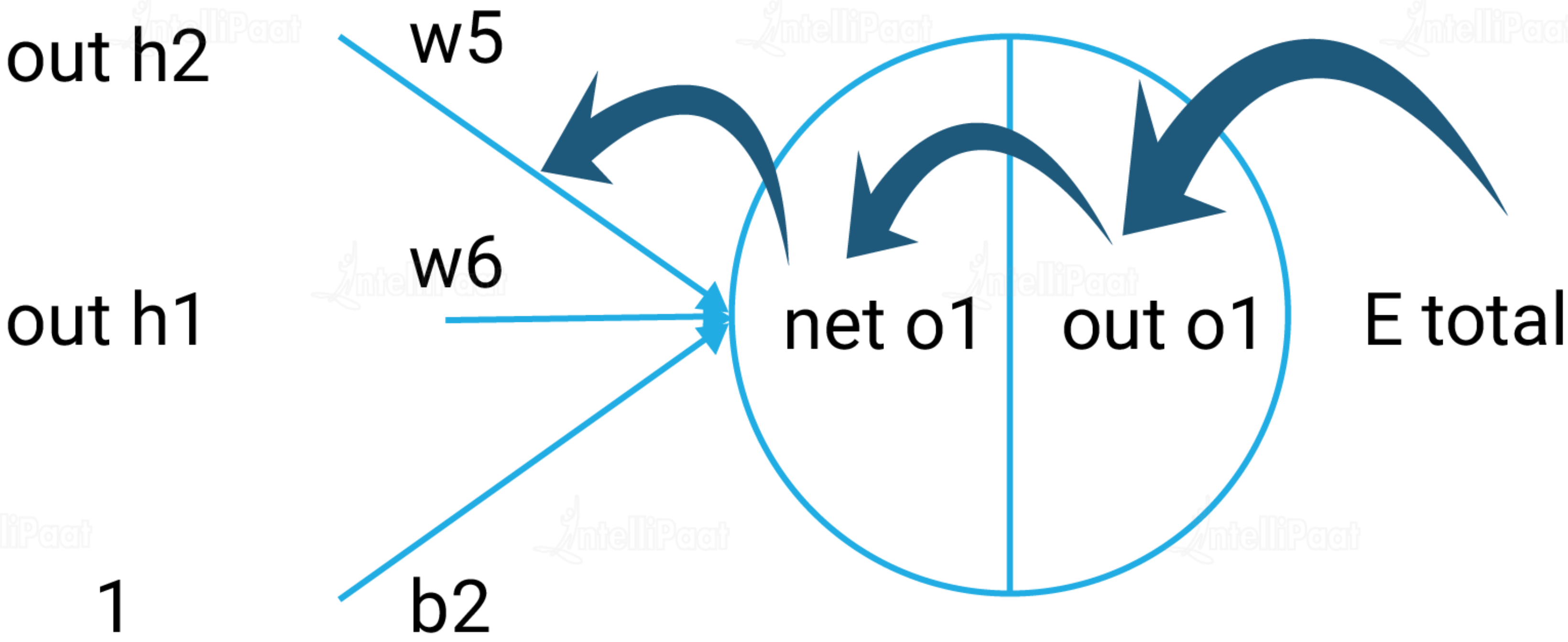
The total error for the neural network is the sum of these errors:

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

Backpropagation Algorithm: How Does it Work?

Step 2: The Backward Pass

Our goal with backpropagation is to update each of the weights in the network so that the actual output is closer to the target output, thereby minimizing the error for each output neuron and the network as a whole



Backpropagation Algorithm: How Does it Work?

Step 2: The Backward Pass

Our goal with backpropagation is to update each of the weights in the network so that the actual output is closer to the target output, thereby minimizing the error for each output neuron and the network as a whole

Consider w5, we will calculate the rate of change of error w.r.t the change in weight w5:

$(\partial E\ total)/\partial w_5 = (\partial E\ total)/(\partial out\ o1) * (\partial out\ o1)/(\partial net\ o1) * (\partial net\ o1)/\partial w_5$

Since we are propagating backwards, the first thing we need to do is to calculate the change in total errors w.r.t. the outputs o1 and o2:

$E\ total = (1/2)*(target\ o1 - out\ o1)^2 + (1/2)*(target\ o2 - out\ o2)^2$

$(\partial E\ total)/(\partial out\ o1) = -(target\ o1 - out\ o1) = -(0.01 - 0.75136507) = 0.74136507$

Now, we will propagate further backward and calculate the change in the output o1 w.r.t to its total net input:

$out\ o1 = 1/(1 + e^{-net\ o1})$

$(\partial out\ o1)/(\partial net\ o1) = out\ o1(1-out\ o1) = 0.75136507(1 - 0.75136507) = 0.186815602$

How much does the total net input of o1 change w.r.t. w5?

$net\ o1 = w_5*out\ h1 + w_6*out\ h2 + b_2*1$

$(\partial net\ o1)/\partial w_5 = 1*out\ h1*w_5^{(1-1)} + 0 + 0 = out\ h1 = 0.593269992$

Backpropagation Algorithm: How Does it Work?

Step 2: The Backward Pass

Putting all values together and calculating the updated weight value

let's put all values together:

$(\partial E\ total)/\partial w_5 = (\partial E\ total)/(\partial out\ o1) * (\partial out\ o1)/(\partial net\ o1) * (\partial net\ o1)/\partial w_5 = 0.082167041$

Calculate the updated value of w5:

$W_5^+ = W_5 - n*(\partial E\ total)/\partial w_5 = 0.4 - 0.5 * 0.082167041 = \mathbf{0.35891648}$

We can repeat this process to get the new weights w6, w7, and w8

$W_6^+ = 0.408666186$

$W_7^+ = 0.511301270$

$W_8^+ = 0.561370121$

We perform the actual updates in the neural network after we have the new weights leading into the hidden layer neurons

Backpropagation Algorithm: How Does it Work?

Step 2: The Backward Pass Hidden Layer

We’ll continue the backward pass by calculating new values for w_1, w_2, w_3, and w_4

Start with w_1:

$(\partial E\ total)/\partial w_1 = (\partial E\ total)/(\partial out\ h1) * (\partial out\ h1)/(\partial net\ h1) * (\partial net\ h1)/\partial w_1$

$(\partial E\ total)/(\partial out\ h1) = (\partial E\ o1)/(\partial out\ h1) + (\partial E\ o2)/(\partial out\ h1)$

We’re going to use a similar process as we did for the output layer, but slightly different to account for the fact that the output of each hidden layer neuron contributes to the output. Thus, we need to take E o1 and E o2 into consideration

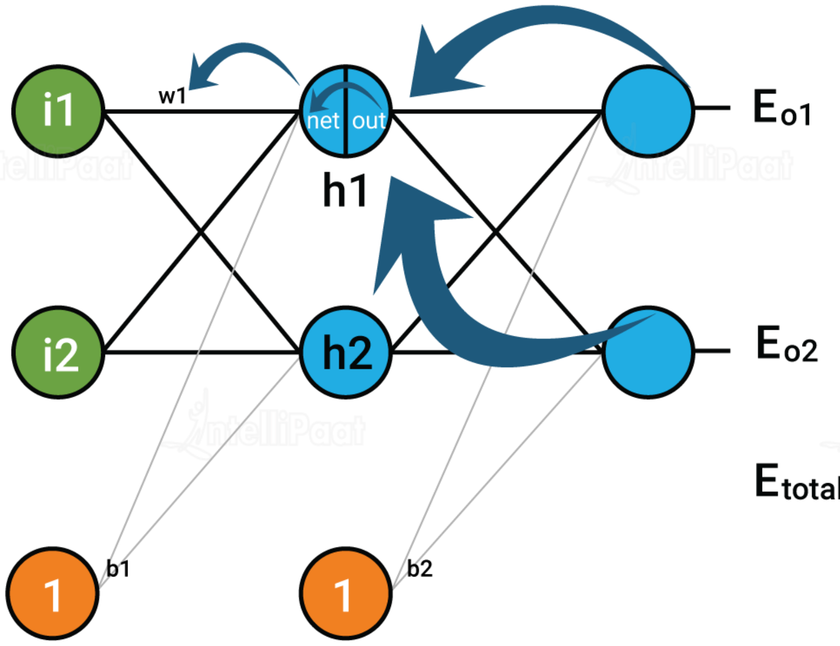
Backpropagation Algorithm: How Does it Work?

Step 2: The Backward Pass Hidden Layer

We can visualize it as:

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

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



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

Backpropagation Algorithm: How Does it Work?

Step 2: The Backward Pass Hidden Layer

Starting with:

$(\partial E\ total)/(\partial out\ h1) = (\partial E\ o1)/(\partial out\ h1) + (\partial E\ o2)/(\partial out\ h1)$

$\mathbf{(\partial E\ o1)/(\partial out\ h1)} = (\partial E\ o1)/(\partial net\ o1) * (\partial net\ o1)/(\partial out\ h1)$

We can calculate $(\partial E\ o1)/(\partial net\ o1)$ *using values calculated earlier:*

$(\partial E\ o1)/(\partial net\ o1) = (\partial E\ o1)/(\partial out\ h1) * (\partial out\ h1)/(\partial net\ o1)$
$= 0.74136507 * 0.186815602 = 0.138498562$

$net\ o1 = w_5 * out\ h1 + w_6 * out\ h2 + b_2 * 1$
$(\partial net\ o1)/(\partial out\ h1) = w_5 = 0.40$

Backpropagation Algorithm: How Does it Work?

Step 2: The Backward Pass
Hidden Layer

Put the values in the equation:

$$(\partial E\ o1)/(\partial out\ h1) = (\partial E\ o1)/(\partial net\ o1) * (\partial net\ o1)/(\partial out\ h1)$$
$$= 0.138498562 * 0.40 = 0.055399425$$

Following the same process for $(\partial E\ o2)/(\partial out\ h1)$, *we get:*

$$(\partial E\ o2)/(\partial out\ h1) = -0.019049119$$

Backpropagation Algorithm: How Does it Work?

Step 2: The Backward Pass Hidden Layer

We can calculate:

$$(\partial E\ total)/(\partial out\ h1) = (\partial E\ o1)/(\partial out\ h1) + (\partial E\ o2)/(\partial out\ h1)$$
$$= 0.055399425 + (-0.019049119) = 0.036350306$$

Now that we have $(\partial E\ total)/(\partial out\ h1)$, *we need to figure out* $(\partial out\ h1)/(\partial net\ h1)$ *and* $(\partial net\ h1)/\partial w$ *for each weight*

$$out\ h1 = 1/(1 + e^{-net\ h1})$$
$$(\partial out\ h1)/(\partial net\ h1) = out\ h1(1- out\ h1) = 0.59326999(1 - 0.59326999) = 0.241300709$$

We calculate the partial derivative of the total net input to h1 with respect to w_1 *the same as we did for the output neuron:*

$$net\ h1 = w_1 * i_1 + w_3 * i_2 + b_1 * 1$$
$$(\partial net\ h1)/\partial w_1 = i_1 = 0.05$$

Backpropagation Algorithm: How Does it Work?

Step 2: The Backward Pass
Hidden Layer

Put it all together:

$$(\partial E\ total)/\partial w_1 = (\partial E\ total)/(\partial out\ h1) * (\partial out\ h1)/(\partial net\ h1) * (\partial net\ h1)/\partial w_1$$

$$(\partial E\ total)/\partial w_1 = 0.036350306 * 0.241300709 * 0.05 = 0.000438568$$

We can now update w_1:

$$w_1^+ = w_1 - n * (\partial E\ total)/\partial w_1 = 0.15 - 0.5 * 0.000438568 = 0.149780716$$

Update other weights similarly:

$$w_2^+ = 0.19956143$$

$$w_3^+ = 0.24975114$$

$$w_4^+ = 0.29950229$$

- *When we fed forward 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 output neurons generate 0.015912196 (vs. 0.01 target) and 0.984065734 (vs. 0.99 target)

Optimization is a big part of Machine Learning. Almost every Machine Learning algorithm has an optimization algorithm at its core

Gradient Descent

- Gradient descent is by far the most popular optimization strategy, used in Machine Learning and Deep Learning at the moment
- It is used while training your model, can be combined with every algorithm, and is easy to understand and implement

Gradient measures how much the output of a function changes if you change the inputs a little bit

- You can also think of a gradient as the slope of a function. *The higher the gradient, the steeper the slope and the faster the model learns*

Gradient Descent

- Gradient descent is by far the most popular optimization strategy, used in Machine Learning and Deep Learning at the moment
- It is used while training your model, can be combined with every algorithm, and is easy to understand and implement

Gradient measures how much the output of a function changes if you change the inputs a little bit

- You can also think of a gradient as the slope of a function. *The higher the gradient, the steeper the slope and the faster the model learns*

How Does it Work?

Gradient Descent

- Gradient descent is by far the most popular optimization strategy, used in Machine Learning and Deep Learning at the moment
- It is used while training your model, can be combined with every algorithm, and is easy to understand and implement

Gradient measures how much the output of a function changes if you change the inputs a little bit

- You can also think of a gradient as the slope of a function. *The higher the gradient, the steeper the slope and the faster the model learns*

How Does it Work?

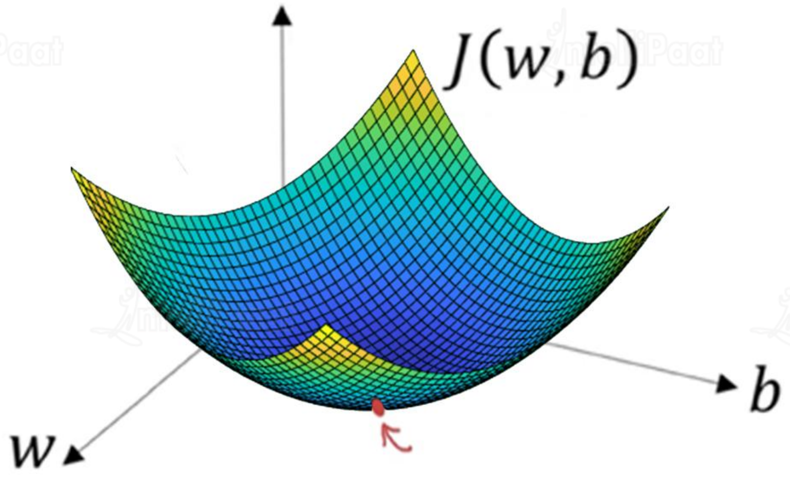
$$b = a - \gamma \nabla f(a)$$

- *b = next value*
- *a = current value*
- *'–' refers to the minimization part of the gradient descent*
- *γ in the middle is the learning rate, and the gradient term $\nabla f(a)$ is simply the direction of the steepest descent*

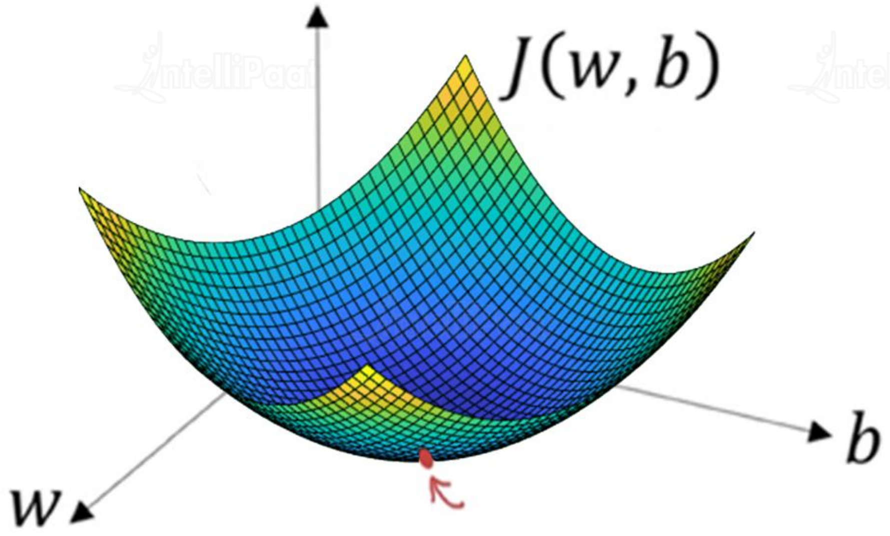
Gradient Descent

$$b = a - \gamma \nabla f(a)$$

- This formula basically tells you the next position where you need to go, which is the direction of the steepest descent
- Gradient descent can be thought of climbing down to the bottom of a valley, instead of climbing up a hill. This is because it is a *minimization algorithm* that minimizes a given function
- Consider the graph below where we need to find the *values of w and b* that correspond to the *minimum of the cost function* (marked with the red arrow)



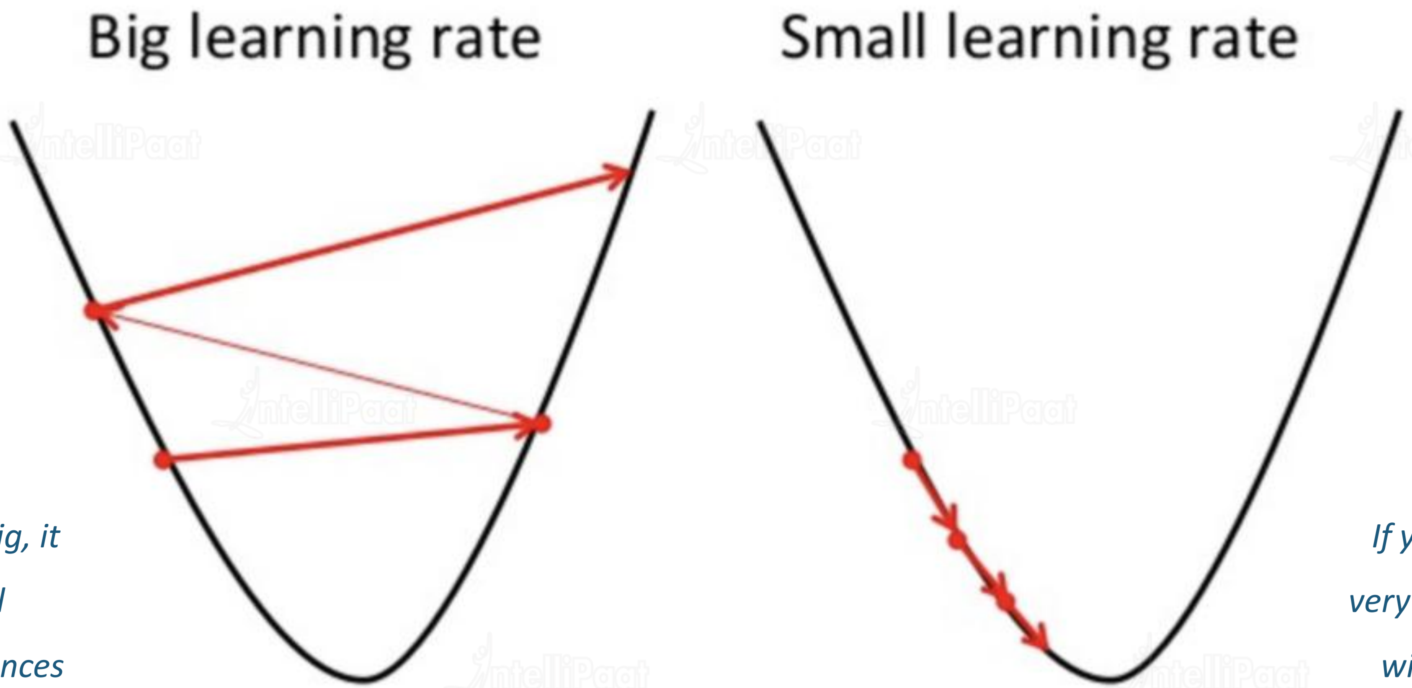
Gradient Descent



- To start with finding the right values, we initialize the values of w and b with some random numbers, and gradient descent then starts at that point (somewhere around the top)
- Then, it takes one step after the other in the steepest downside direction (e.g., from top to bottom) till it reaches the point where the cost function is as small as possible

Importance of the Learning Rate

- Learning rate determines how fast or slow we will move toward the optimal weights
- In order for gradient descent to reach the local minimum, we have to set the learning rate to an appropriate value, which is neither too low nor too high

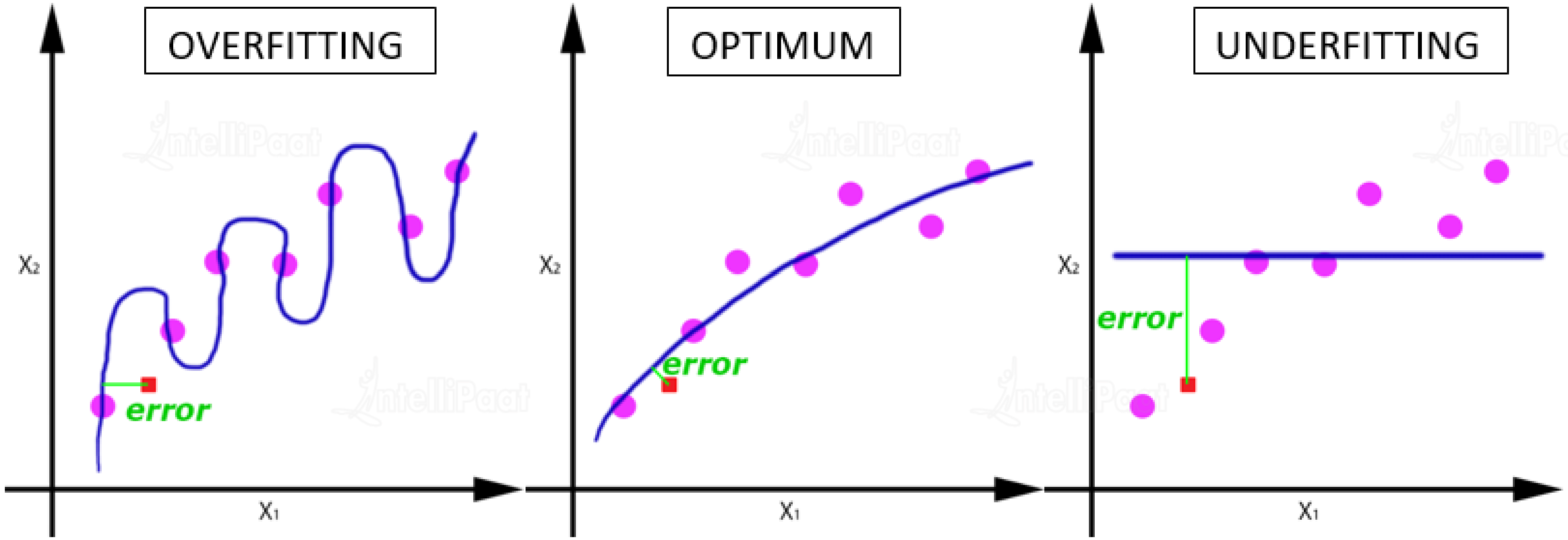


If the steps it takes are too big, it might not reach the local minimum because it just bounces back and forth between the convex function of gradient descent

If you set the learning rate to a very small value, gradient descent will eventually reach the local minimum, but it might take too much time

Understanding Epoch

- *One epoch is when an ENTIRE dataset is passed forward and backward through the neural network only ONCE*
- One epoch leads to underfitting of the curve in the graph



- As the number of epochs increases, more number of times the weights are changed in the neural network and the curve goes from **underfitting** to **optimal** to **overfitting**

Batches and Iterations

Batch Size

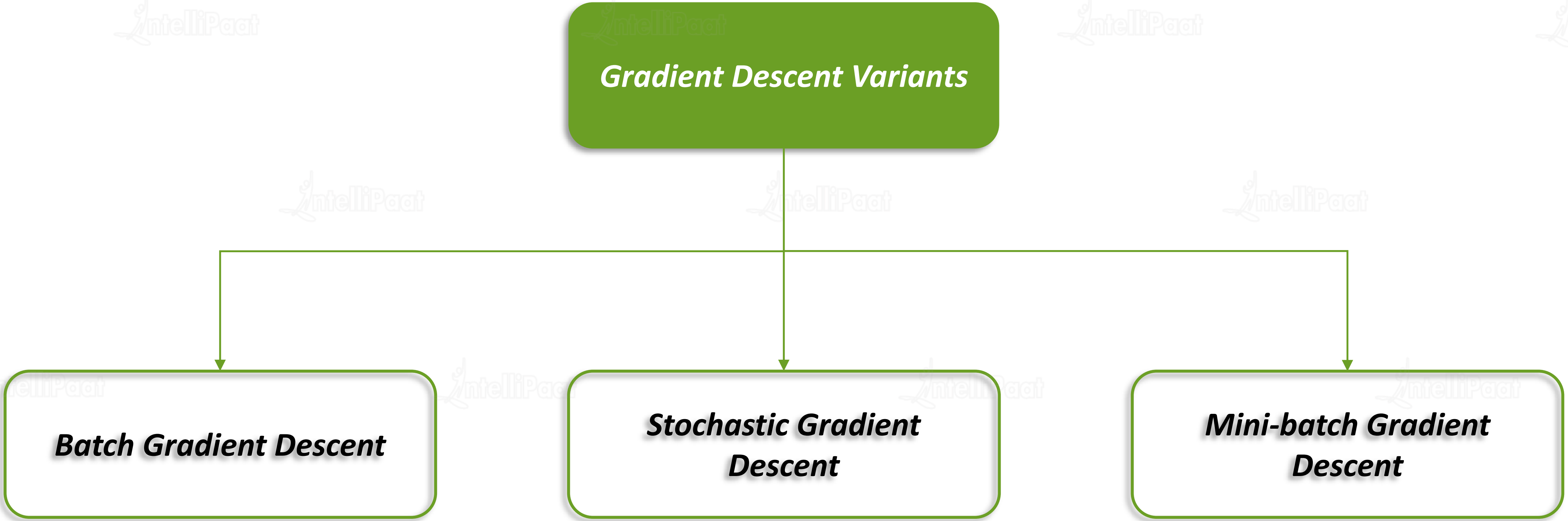
- Total number of training examples present in a single batch is referred to as the batch size
- Since we can’t pass the entire dataset into the neural net at once, we divide the dataset into number of batches or sets or parts

Iterations

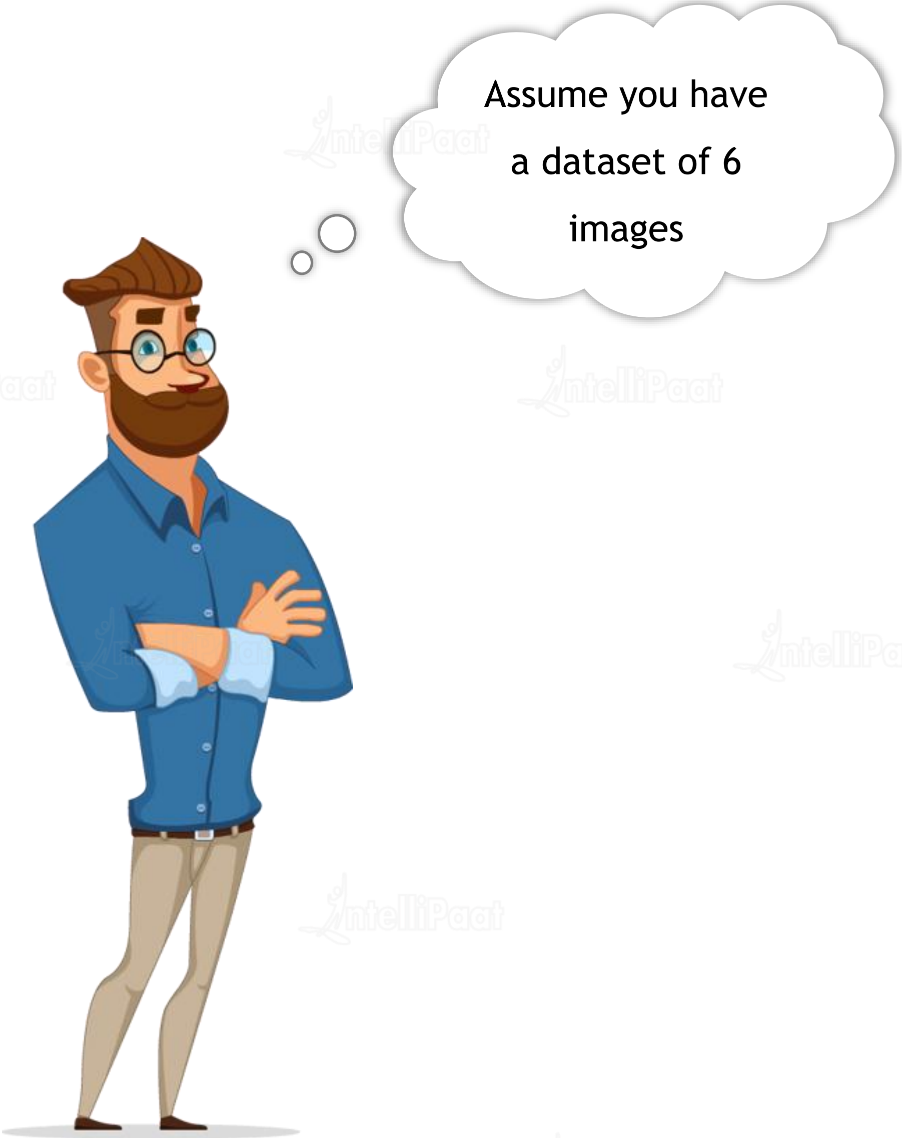
- Iteration is the number of batches needed to complete one epoch

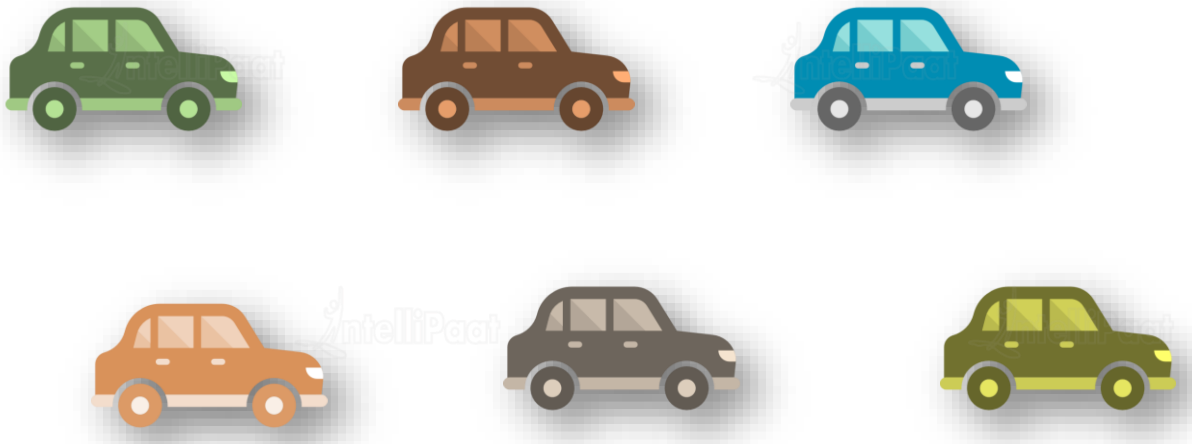
Let’s say, we have 2,000 training examples that we are going to use. We can divide the dataset of 2,000 examples into batches of 500, and then it will take four iterations to complete one epoch

Gradient Descent Variants



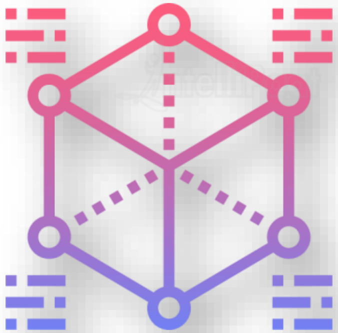
Gradient Descent Variants





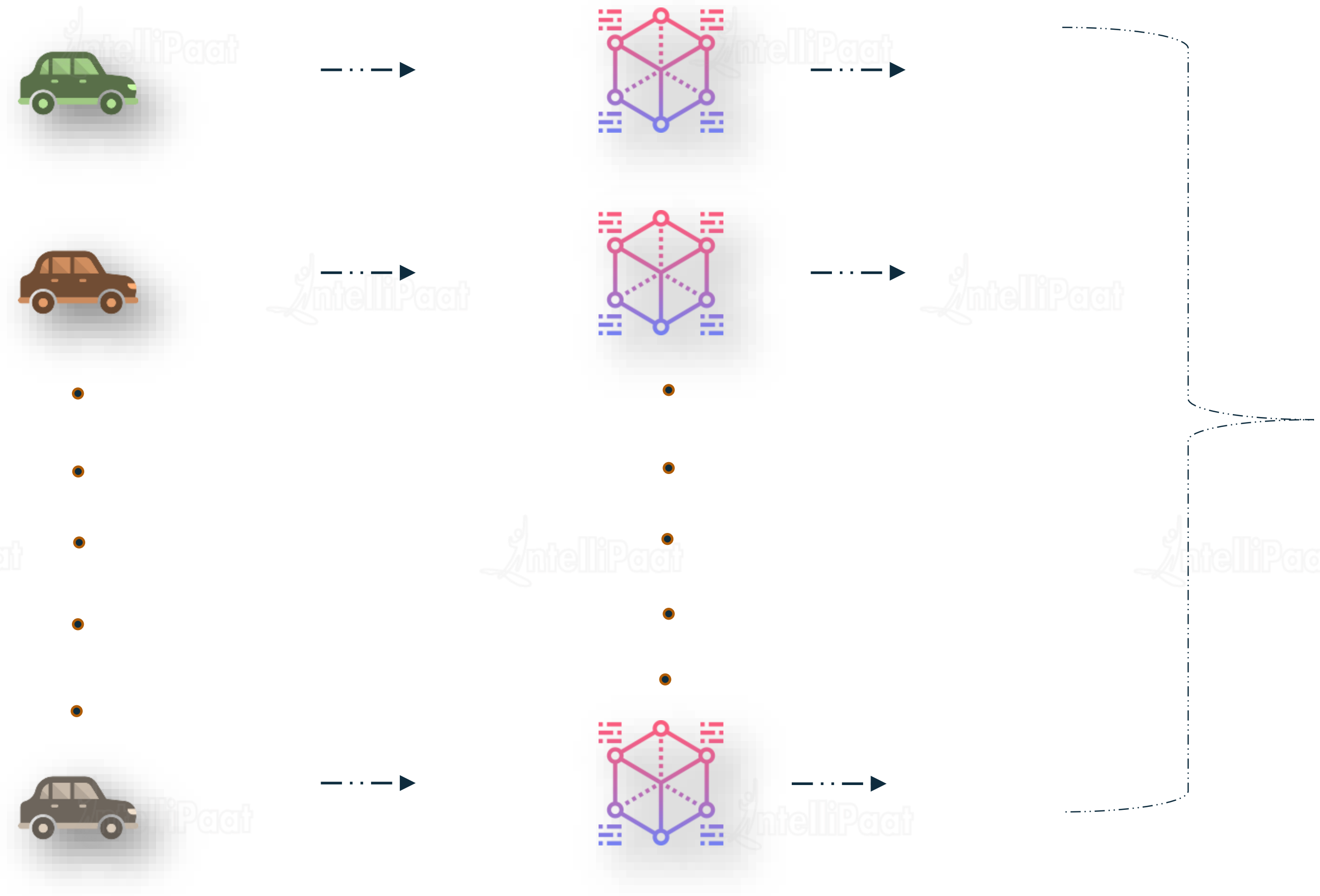
Batch Gradient Descent

Takes the entire dataset at a time



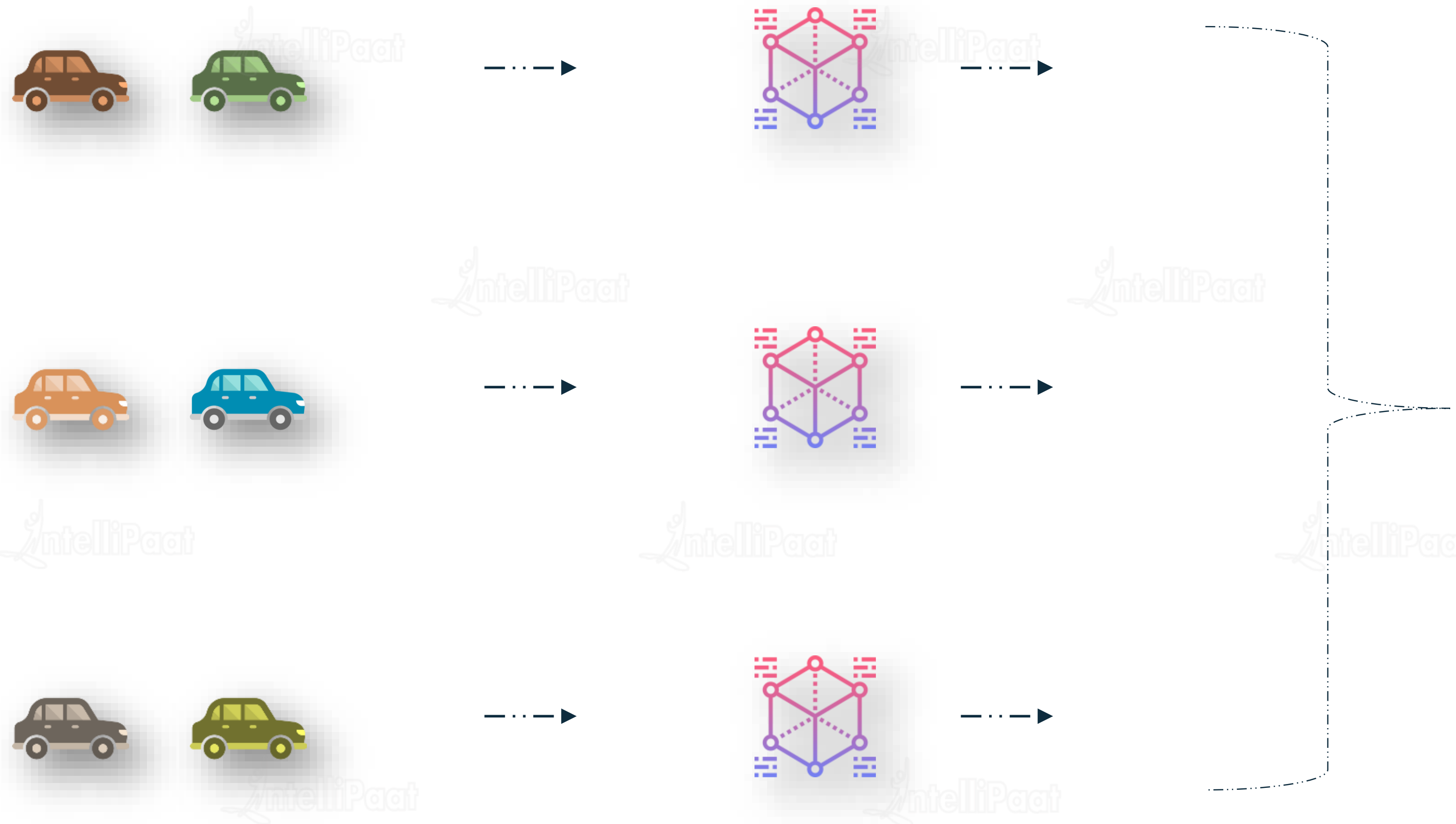
Back-propagates the loss for all 10 images at a time

Stochastic Gradient Descent



Back-propagates the loss for each image and updates the gradient

Mini-Batch Gradient Descent



Back-propagates the loss for each mini batch

Tips for Gradient Descent

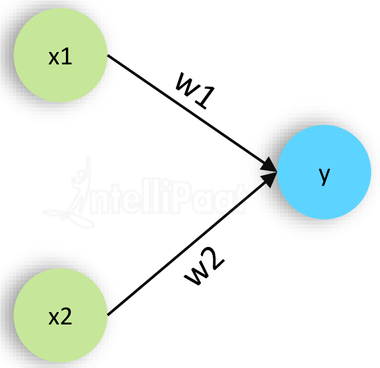
- ***Plot Cost versus Time:*** Collect and plot the cost values calculated by the algorithm for each iteration. The expectation for a well-performing gradient descent run is a decrease in cost at every iteration. If it does not decrease, try reducing your learning rate
- ***Learning Rate:*** The learning rate value is a small real value such as 0.1, 0.001, or 0.0001. Try different values for your problem and see which works best
- ***Rescale Inputs:*** The algorithm will reach the minimum cost faster if the shape of the cost function is not skewed and distorted. You can achieve this by rescaling all of the input variables (*X*) to the same range, such as [0, 1] or [-1, 1]

Adam Optimization Algorithm

- The Adaptive Moment Estimation or Adam optimization algorithm is a combination of gradient descent with momentum and RMSprop algorithms
- Adam is an adaptive learning rate method, which means that it computes individual learning rates for different parameters

Implementing a Simple Neural Network

Implementing a Simple Neural Network



Let x1 = 2, x2 = 5, and y = 31

Initializing the values of x1, x2, and y:

```
In [8]: x1=2
        x2=5
        y=31
```

Implementing a Simple Neural Network

Implementing forward propagation and backpropagation:

```
In [10]: lr=0.01
         w1=9
         w2=7

         for epoch in range(25):
             y_pred=w1*x1 + w2*x2
             error=(y-y_pred)**2
             dEW1=2*(y-y_pred)*(-x1)
             dEW2=2*(y-y_pred)*(-x2)

             w1=w1-(lr*dEW1)
             w2=w2-(lr*dEW2)
             print("Value of w1 - ", w1, "Value of w2 - ", w2, "Error is - ", error)
```

```
Value of w1 -  8.12 Value of w2 -  4.8 Error is -  484
Value of w1 -  7.750399999999999 Value of w2 -  3.8760000000000003 Error is -  85.3775999999999
Value of w1 -  7.595167999999999 Value of w2 -  3.4879200000000004 Error is -  15.060608640000005
Value of w1 -  7.529970559999999 Value of w2 -  3.3249264000000003 Error is -  2.656691364096002
Value of w1 -  7.5025876351999985 Value of w2 -  3.2564690880000002 Error is -  0.4686403566265386
Value of w1 -  7.491086806783999 Value of w2 -  3.2277170169600007 Error is -  0.0826681589089180７
Value of w1 -  7.486256458849279 Value of w2 -  3.2156411471232005 Error is -  0.014582663231533663
Value of w1 -  7.484227712716696 Value of w2 -  3.210569281791744 Error is -  0.002572381794042668
Value of w1 -  7.483375639341012 Value of w2 -  3.208439098352533 Error is -  0.0004537681484689692
Value of w1 -  7.483017768523224 Value of w2 -  3.2075444213080644 Error is -  8.00447013899338e-05
Value of w1 -  7.482867462779753 Value of w2 -  3.2071686569493876 Error is -  1.411988532519286e-05
Value of w1 -  7.482804334367495 Value of w2 -  3.207010835918743 Error is -  2.490747771367834e-06
Value of w1 -  7.48277782043434７ Value of w2 -  3.2069445510858725 Error is -  4.393679068673078７e-07
Value of w1 -  7.482766684582424 Value of w2 -  3.2069167114560666 Error is -  7.75044987719865４e-08
Value of w1 -  7.4827620075246175 Value of w2 -  3.2069050188115487 Error is -  1.3671793582514382e-08
Value of w1 -  7.482760043160338 Value of w2 -  3.206900107900851 Error is -  2.411704387857833e-09
Value of w1 -  7.482759218127341 Value of w2 -  3.2068980453183578 Error is -  4.254246540708817e-10
Value of w1 -  7.482758871613482 Value of w2 -  3.206897179033711 Error is -  7.504490893870946e-11
Value of w1 -  7.482758726077662 Value of w2 -  3.2068968151941593 Error is -  1.3237921953333858e-11
Value of w1 -  7.482758664952617 Value of w2 -  3.2068966623815474 Error is -  2.335169438214246７e-12
Value of w1 -  7.482758639280099 Value of w2 -  3.2068965982002506 Error is -  4.119238848878803５e-13
Value of w1 -  7.48275862849764 Value of w2 -  3.2068965712441053 Error is -  7.266337750799056e-14
Value of w1 -  7.482758623969008 Value of w2 -  3.2068965599225243 Error is -  1.2817819245385772e-14
Value of w1 -  7.482758622066982 Value of w2 -  3.20689655516746１ Error is -  2.261062929716986e-15
Value of w1 -  7.482758621268132 Value of w2 -  3.206896553170334 Error is -  3.988515178306028e-16
```

Multi-variable Equation

Multi-variable Equation

1

Loading the NumPy package:

```
In [36]: import numpy as np
```

2

Setting initial values for x1, x2, x3, and y:

```
In [21]: x1 = np.random.randint(3, 12, 10)
         x2 = np.random.randint(9, 18, 10)
         x3 = np.random.randint(12, 20, 10)
         y = x1*7 + 5*x2 + 4*x3
```

3

Having a glance at x1, x2, x3, and y:

```
In [38]: x1,x2,x3
```

```
Out[40]: (array([8, 7, 3, 8, 4, 4, 7, 3, 7, 7]),
          array([12, 13, 17, 15,  9, 14, 12, 15, 15, 10]),
          array([13, 12, 19, 12, 15, 19, 16, 15, 18, 13]))
```

Multi-variable Equation

Reshaping 'y':

4

```
In [41]: y = y.reshape(SHAPE,1)
         y
```

```
Out[41]:  array([[168],
                 [162],
                 [182],
                 [179],
                 [133],
                 [174],
                 [173],
                 [156],
                 [196],
                 [151]])
```

Creating a NumPy array from 'x1', 'x2' , and 'x3':

5

```
In [32]: X = np.array([x1, x2, x3])
```

Transposing 'X':

6

```
In [41]: y = y.reshape(SHAPE,1)
         y
```

```
Out[33]:  array([[ 8, 12, 13],
                 [ 7, 13, 12],
                 [ 3, 17, 19],
                 [ 8, 15, 12],
                 [ 4,  9, 15],
                 [ 4, 14, 19],
                 [ 7, 12, 16],
                 [ 3, 15, 15],
                 [ 7, 15, 18],
                 [ 7, 10, 13]])
```

Multi-variable Equation

Setting the learning rate value and initializing random values:

7

```
In [34]: lr = 0.0001
         w = np.array([np.random.randn(1),
                       np.random.randn(1),
                       np.random.randn(1)])
```

Implementing forward propagation and backpropagation:

8

```
In [35]: error_list = []
         for i in range(50):
             y_pred = np.matmul(X, w)
             error = (y - y_pred)**2
             dEw1 = 2*np.matmul((y - y_pred).T, (-X[:,0].reshape(SHAPE, 1)))
             dEw2 = 2*np.matmul((y - y_pred).T, (-X[:,1].reshape(SHAPE, 1)))
             dEw3 = 2*np.matmul((y - y_pred).T, (-X[:,2].reshape(SHAPE, 1)))

             w[0][0] = w[0][0] - lr * dEw1
             w[1][0] = w[1][0] - lr * dEw2
             w[2][0] = w[2][0] - lr * dEw3

             error_list.append(error.mean())
             print("value of w1 - ", w[0][0], "value of w2 - ", w[1][0], "value of w3 - ", w[2][0])
             plt.plot(error_list)
```

Quiz

Quiz 1

Single Layer Perceptron is easy to use than Multi Layer Perceptron

A	True
B	False

Answer 1

Single Layer Perceptron is easy to use than Multi Layer Perceptron

A	True
B	False

Quiz 2

The variants of Gradient Descent are...

A	Batch Gradient Descent
B	Adams Optimizer
C	Ada-Delta Optimizer
D	All of these

Answer 2

The variants of Gradient Descent are...

A	Batch Gradient Descent
B	Adams Optimizer
C	Ada-Delta Optimizer
D	All of these

Quiz 3

The backpropagation algorithm is a supervised learning method for multi-layer feedforward networks

A	Yes
B	No

Answer 3

The backpropagation algorithm is a supervised learning method for multi-layer feedforward networks

A	Yes
B	No

Thank you!

India: +91-7847955955

US: 1-800-216-8930 (TOLL FREE)

sales@intellipaat.com

24/7 Chat with Our Course Advisor