REGRESSION:

Regression is a method of finding the relation between one dependant variable and other independent variables.

Let's just understand some calculation based on our sample data -
Say, we have two variables, x and y and we need to find the relationship between them, the effect that x has on y when y is dependent on x.

Sample data:

x	y
1	2
2	4
3	6
4	8

Well, from this simple example, we can all say that :

$$y = 2 * x$$

where 2 stands for weight which can be denoted by w. Hence re-writing the above equation, we have -

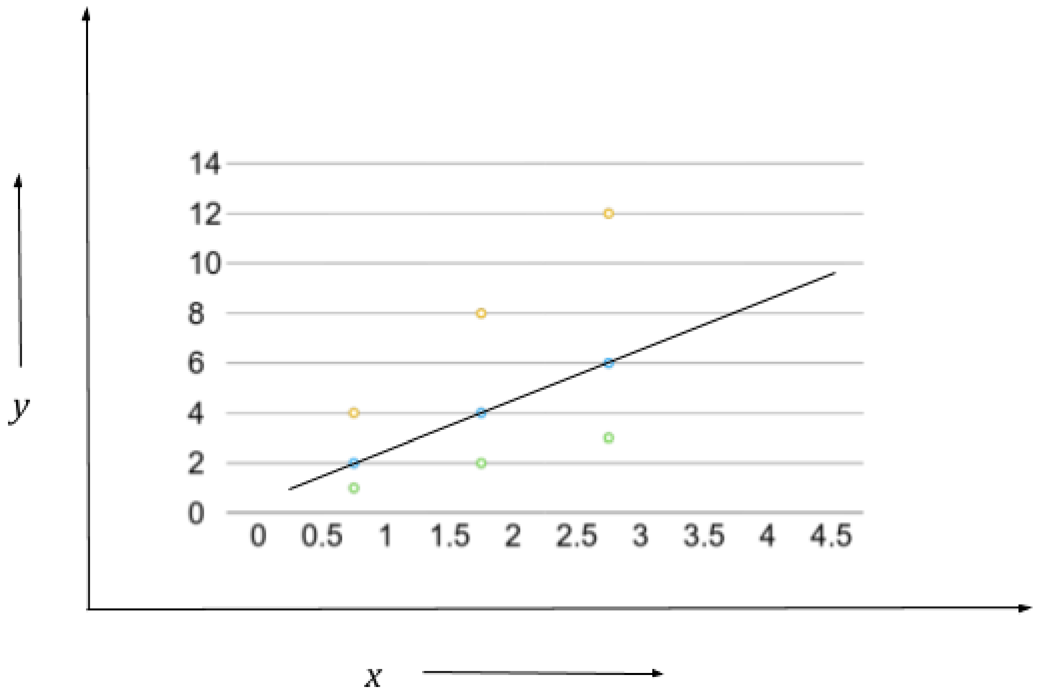
$$y = x * w$$

In reality there is a bias associated with this equation -

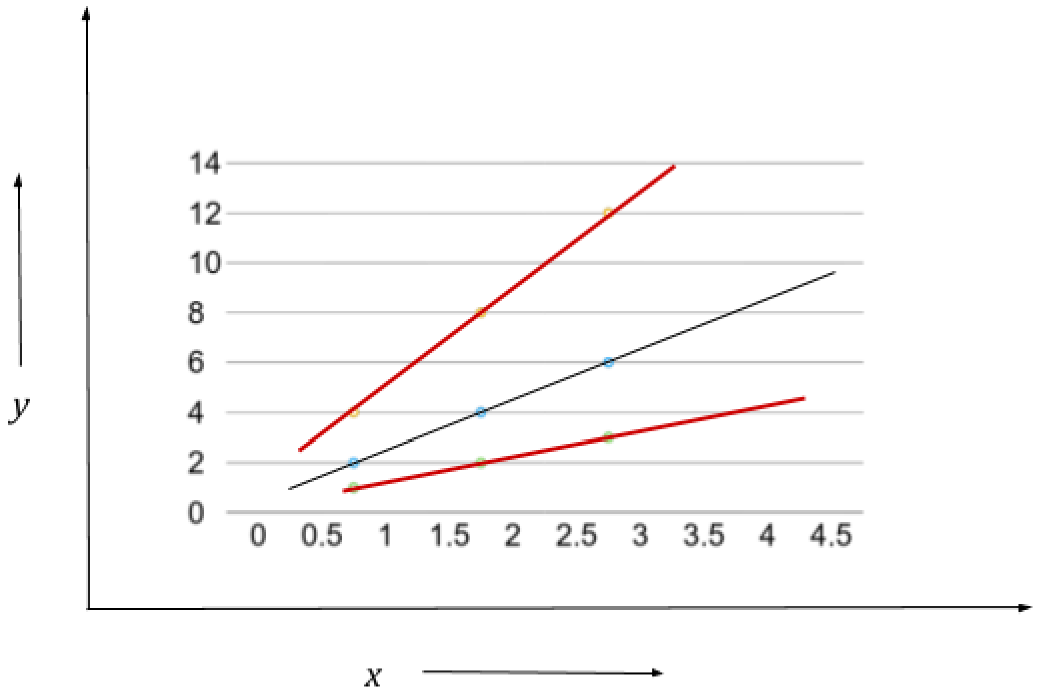
$$y = x * w + b$$

But for simplicity's sake, we won't consider the bias as of now.

Based on the simple linear equation, we can draw a line as below -

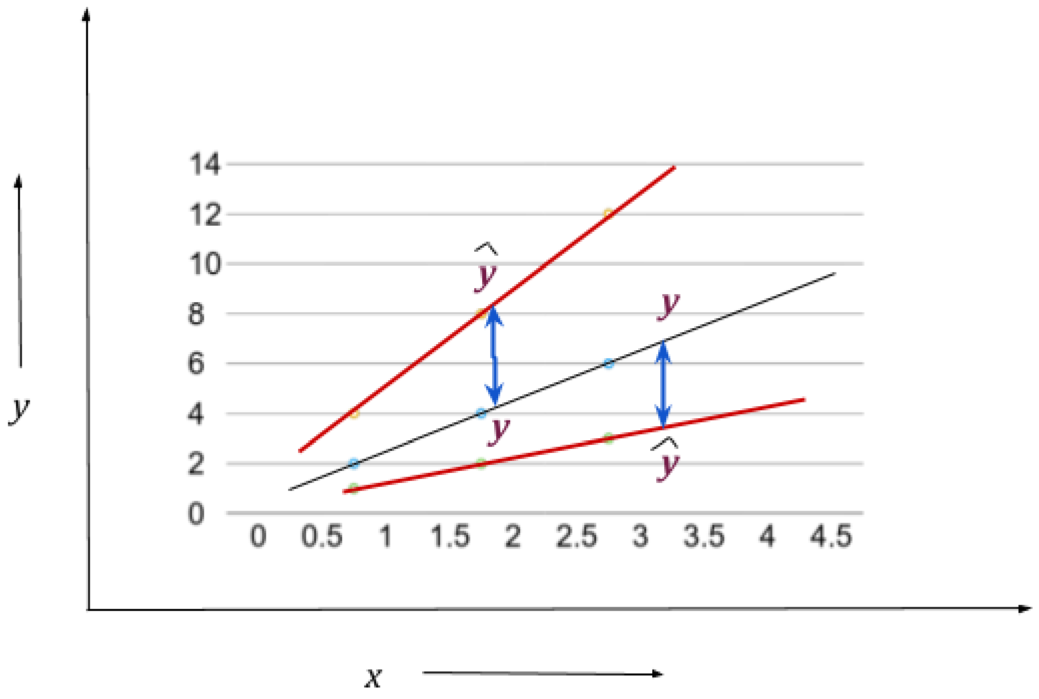


For the sample dataset taken, this becomes our true line. Since, machine does not have enough insights about the data above, we can start the weight at some random value. The chart below depicts two random guesses denoted by two red lines.



Our goal is to make these two random guesses closer to the true line. We also need to decide line to choose so that we can move towards the true line in least number of steps. The best way to measure this is by calculating the distance.

$$\widehat{y} - y$$



Since it is not required to know whether the difference is positive or negative, we simply square the difference and this gives us the error.

$$error \ = \ (\widehat{y} - y)^2$$

The goal of machine learning is to make this error very small so that our loss is significantly reduced.

Therefore, loss can be calculated as -

$$loss \ = \ (\widehat{y} - y)^2$$

Rephrasing the equation of a simple linear model above, based on the graph, we have -

$$\widehat{y} = x * w$$

Therefore, loss can be equated as -

$$loss \;=\; (\hat{y} - y)^2 = (x * w - y)^2$$

Now, based on certain random valueof weights, we can very easily calculate the y_prediction $\hat{y}$ and also the square of the error.

Therefore, we can rewrite the loss equation to present it formally -

$$loss(w) \;=\; \frac{1}{N} \sum_{n=1}^{N} (\hat{y}_n - y_n)^2$$

This can also be called as the mean square error.

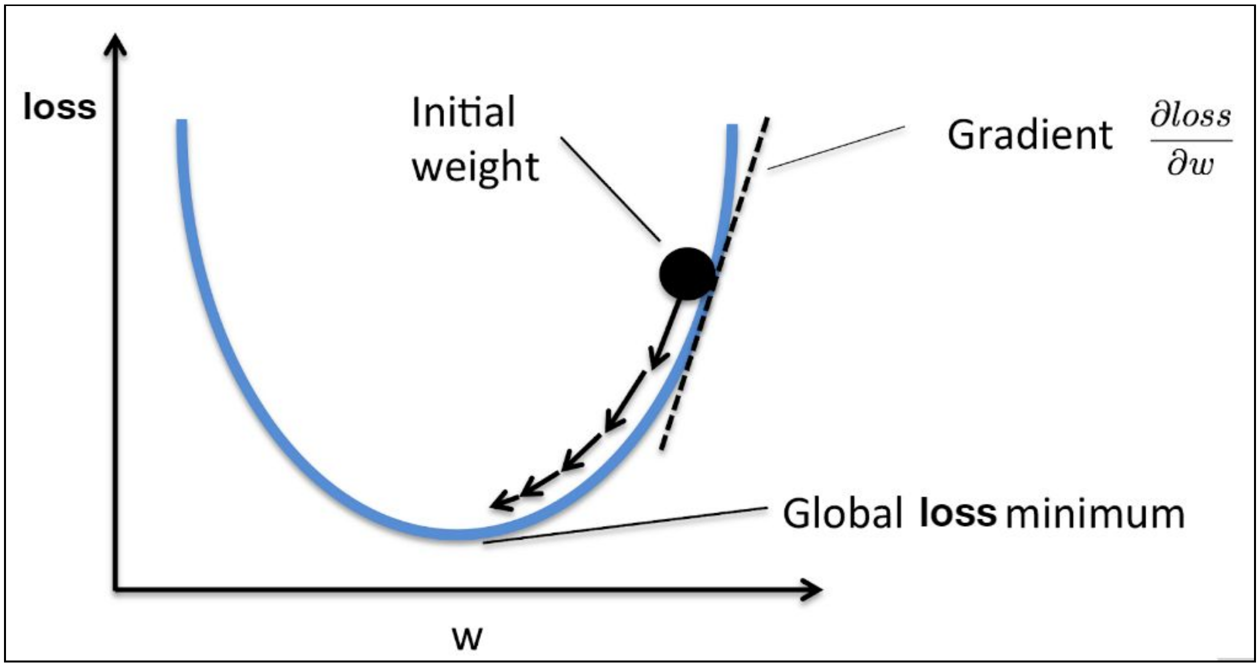
Now, it is easy to compute on small scale data. In case of very large dataset and complex relationship, it is impossible to predict each and every relationship value (weights). A mechanism is required where selection of weights is done automatically and it is capable of minimizing the loss. One of the most common method to do that is gradient descent. Let's discuss this in the next section.

GRADIENT DESCENT

In the regression section, we found out the loss function and how we can compute errors with the help of this formula -

$$loss(w) \;=\; \frac{1}{N} \sum_{n=1}^{N} (\hat{y}_n - y_n)^2$$

We have also seen the graph which is -



Graph: Obtaining gradient descent.
Courtesy: Sung Kim

In the training phase, we try to find out the value of the weight (w) which essentially minimizes the loss. Looking at the graph, we can say that we need to bring the initial random w value close to the true w value so as to reduce the loss error. Therefore we can define learning as finding a w that minimizes the loss. Mathematically, it is represented as -

$$arg\ min\ loss(w)$$

In case of neural networks where we have to calculate thousands of weights it is necessary to obtain this w value that significantly minimizes the loss automatically. The algorithm that does this systematically is known as gradient descent algorithm.

How is it done? - Looking at the loss graph above, since we are not aware of where to start with, we can start at any random value (the black circle marked as Initial weight in the above graph) . Now, based on the random value, we need to decide whether we need to decrease or increase this value to obtain the correct value. The best way to decide this is by computing the gradient -

$$\frac{\delta loss}{\delta w}$$

of the random weight w and based on this gradient we can decide whether we need to increase or decrease the value. So, if the gradient is positive (as of now, marked as initial weight in the graph), we will need to decrease its value to obtain the perfect weight which minimizes loss and hence w becomes negative, and if the gradient is negative (initial weight is on the left side of the graph) then we will need to increase its value to obtain the perfect weight that minimizes loss and hence our weight becomes positive. This can be expressed with the help of a mathematical notation -

$$w = w - \alpha \frac{\delta loss}{\delta w}$$

where,

> The w on the right hand side of the equation denotes the initial weight
> α is the value by which we need to define the increase or decrease in weight that is required to reach the optimum weight. This is a very small value usually 0.01 and so on.
> And, $\frac{\delta loss}{\delta w}$ is the gradient.

So, based on this equation, we move to the next position in the graph (increasing in case of negative initial weight and decreasing in case of positive initial weight) and again we calculate the gradient and adjust the position and so on until and unless we reach a point where the gradient is almost zero and this determines our perfect weight value which is capable of significantly minimizing the loss function.

Therefore, mathematically, we can define this whole systematic method in the form of two equations -

$$loss = (\widehat{y} - y)^2 = (x * w - y)^2$$

$$w = w - \alpha \frac{\delta loss}{\delta w}$$

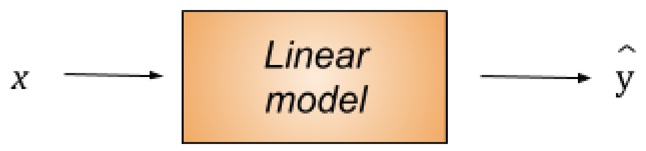
Now, let’s just code the entire process discussed above by assigning gradients manually.

GRADIENT DESCENT IN PyTorch:

BACKPROPAGATION:

In this section we are going to automatically calculate the above mentioned gradient , using backpropagation.

In the above section we discussed about training using a simple linear model which takes in x as input and gives $\hat{y}$ as output.

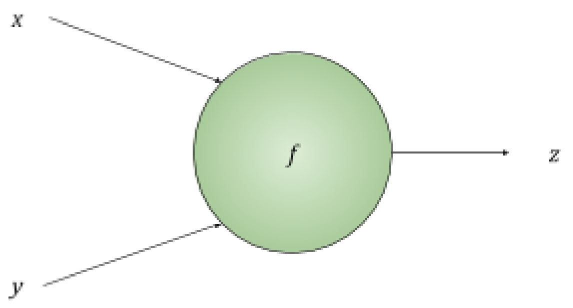


Then we calculate the gradient of the loss $\frac{\delta loss}{\delta w}$ with respect to the weight w, and applying it to gradient descent algorithm to find the perfect value of w which minimizes the loss. This is the training process and we just manually computed our gradient as our model was fairly simple.

However in case of complex neural network, we are dealing with non-linearities between each of the nodes and in which case, computing gradient descent becomes complex. Hence to overcome such situation, the best method to use is the chain rule or computational graphs.

Understanding Backpropagation using chain rule:

To begin with, let's just consider one node in a huge neural network.



From the above figure, there is a function f which takes input as x and y and produce z as output. It is obvious that this single node is connected to many other nodes and will eventually produce loss. We are interested in finding out the gradient of loss with respect to the input variables. Let's do this step by step.

<u>Step 1:</u>

We will start by computing the local gradient - we know what is function f , hence finding out the gradient of z with respect to its input x = $\left(\frac{\delta z}{\delta x}\right)$ and y = $\left(\frac{\delta z}{\delta y}\right)$ is easy.

<u>Step 2:</u>

Let's assume that the loss at the output z = $\left(\frac{\delta L}{\delta z}\right)$ is independent of our previous computations. Then, we use Chain rule to compute the derivative of the loss at the inputs viz. x = $\left(\frac{\delta L}{\delta x}\right)$ and y = $\left(\frac{\delta L}{\delta y}\right)$.

This calculation is simple as we can use chain rule to multiply the output gradient with the local gradient -

$$\frac{\delta L}{\delta x} = \frac{\delta L}{\delta z}\frac{\delta z}{\delta x}$$

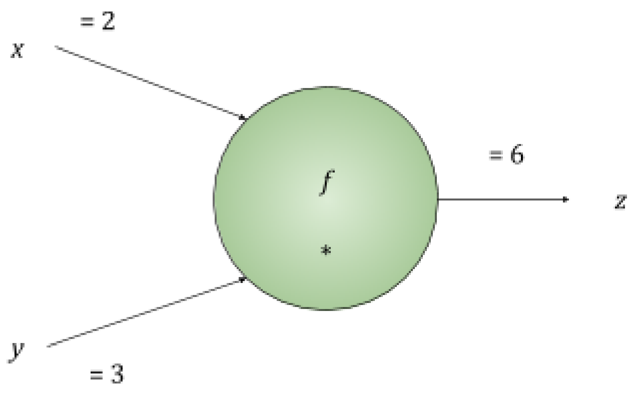
and,

$$\frac{\delta L}{\delta y} = \frac{\delta L}{\delta z}\frac{\delta z}{\delta y}$$

Now, let's see an example of forward pass with real value -

Let's say:

$$x = 2$$
$$y = 3$$



Now, calculating the local gradient -

Since we know that the function is a multiplication function, therefore:

$$z = x * y$$

Therefore, from step 1 above,

$$\frac{\delta z}{\delta x} = \frac{\delta xy}{\delta x} = y$$

and,

$$\frac{\delta z}{\delta y} = \frac{\delta xy}{\delta y} = x$$

Hence, accordingly we can calculate the local gradient based on the equations in step 1.

Now, consider the gradient at the output is given, let's say -

$$\frac{\delta L}{\delta z} = 6$$

and we can compute the value of loss at the input based on the equations in step 2.

This is how backward propagation works, by memorizing the value given in the forward pass and then doing backward propagation using the chain rule.

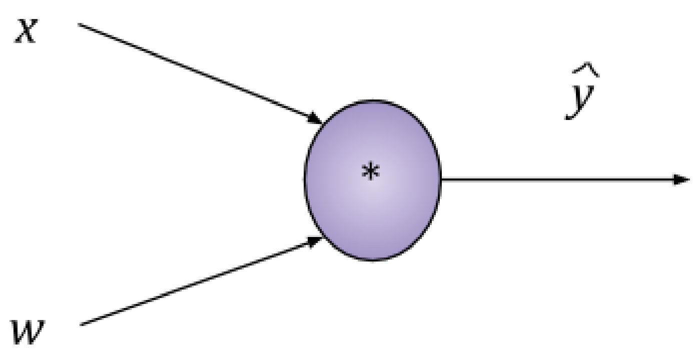
Understanding Backpropagation using computational graphs:

Applying the above idea to a simple linear model -

$$\widehat{y} = x * w$$

Let's draw a computation graph for the same. We will go step by step -

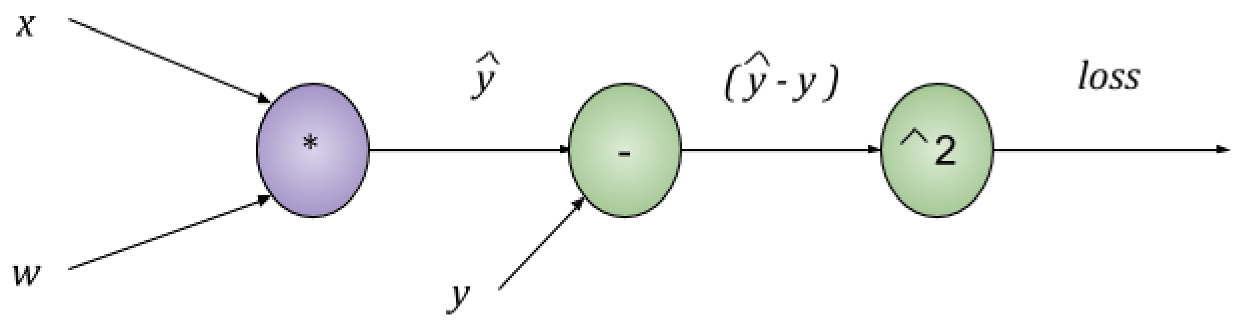
First, lets x and w and then put a function to get $\widehat{y}$ -



Our aim to calculate the gradient of the loss, now, lets look at the loss equation -

$$loss = (\hat{y} - y)^2 = (x * w - y)^2$$

Now, let's connect this loss to our computational graph above -

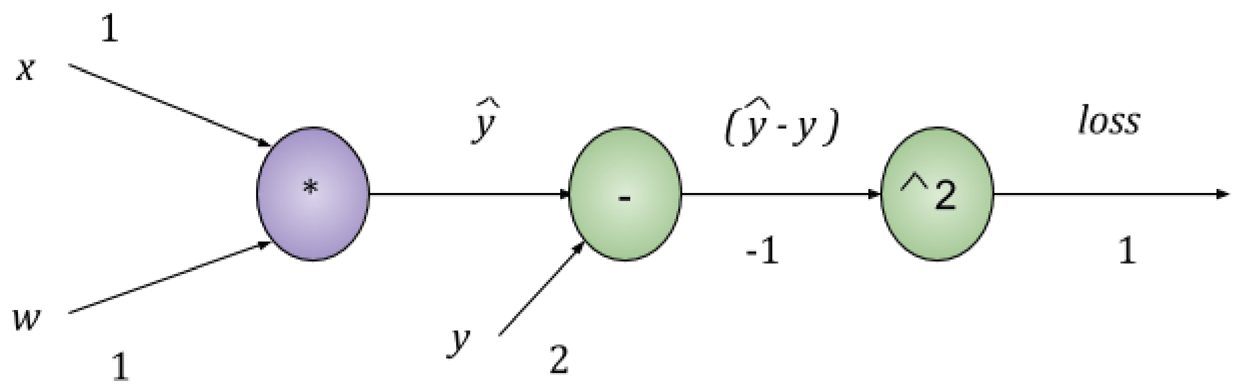


Now, let's apply the forward pass with some some real values -

Let's say:

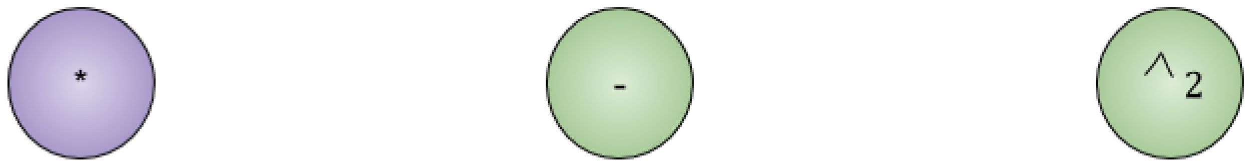
$$x = 1$$
$$y = 2$$
$$w = 1$$

So, by this the value becomes -



Now, to calculate backpropagation, we need to know the local gradient of each of the function in the above diagram -

So, we have 3 nodes with 3 different function -



node 1　　*node 2*　　*node 3*

Let's calculate the local gradient of node 1:

$$\frac{\delta xw}{\delta w} = x$$

Gradient of node 2 is:

$$\frac{\delta \widehat{y} - y}{\delta \widehat{y}} = \ -(-y)$$

Gradient of node 3 is:

$$\frac{\delta (\widehat{y} - y)^2}{\delta (\widehat{y} - y)} = 2(\widehat{y} - y)$$

Now, to calculate the loss gradient, we just need to compute each of these local gradient in a backward manner with each of the node function and compare the values -

$$\frac{\delta loss}{\delta(\hat{y}-y)} = 2(\hat{y} - y)$$

Then moving backward, computing the gradient of loss with respect to $\hat{y}$; so using the chain rule, we have the equation as -

$$\frac{\delta loss}{\delta \hat{y}} = \frac{\delta loss}{\delta(\hat{y}-y)} \frac{\delta \hat{y}-y}{\delta \hat{y}}$$

Now, calculating the gradient of loss against the weight w, applying chain rule we have -

$$\frac{\delta loss}{\delta w} = \frac{\delta loss}{\delta \hat{y}} \frac{\delta \hat{y}}{\delta w}$$

Then applying all the real value from above, we get the value of the loss function.

BACKPROPAGATION IN PyTorch:

You can very easily code this backpropagation in PyTorch without having to specify much. PyTorch will automatically build the computation graph and compute all the gradients for you. Now let's just dive into code -

LINEAR REGRESSION:

In this kind of regression, we model the relationship between inputs x and the predictions y using a linear approach. It is to understand the behaviour of the dependent variable y which is in relation to the independent variable x.

x	y
1	2
2	4
3	6
4	8

LINEAR REGRESSION IN PyTorch:

LOGISTIC REGRESSION:

In this, the terms remain the same as linear regression, but the values of y will be either 0 or 1. Hence, it is a kind of binary classification problem which can be used to predict problems such as pass or failure.

x	y
1	0
2	0
3	1
4	1

LOGISTIC REGRESSION IN PyTorch: