INTRO TO DL
PART 2

Skandan Subramanian

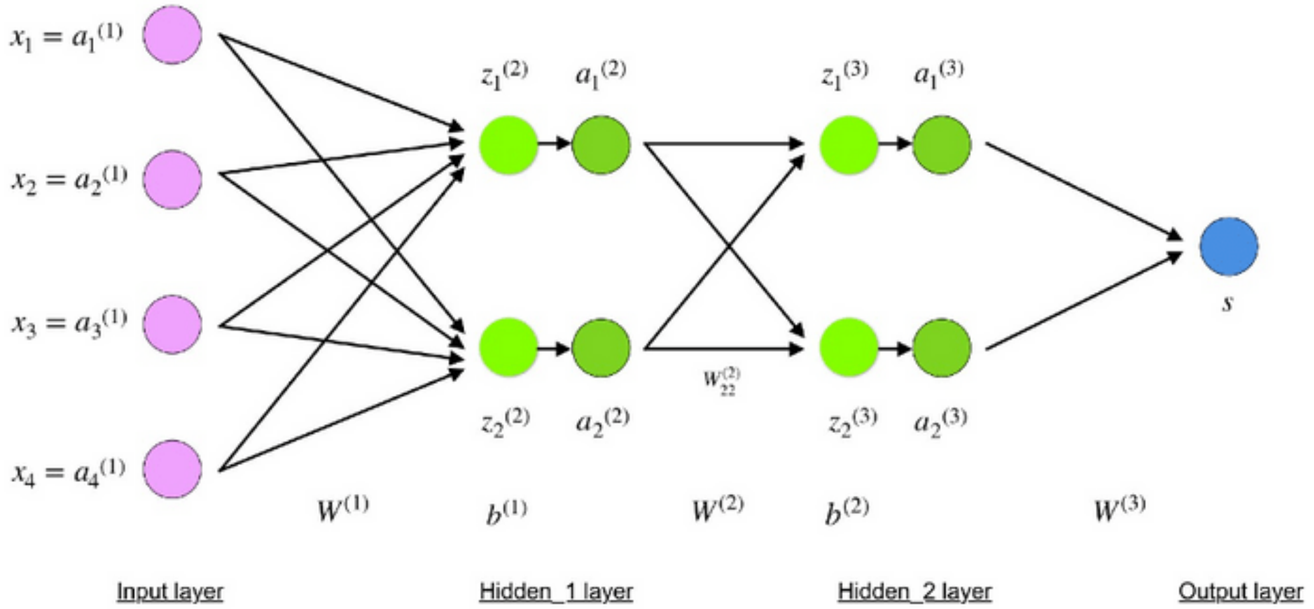
Kishore K

Eshan Prasad Kulkarni

Table of Contents

Forward propogation - a recap

Consider the following simple neural network



This network has :

- 1 input layer with 4 neurons
- 2 hidden layers with 2 neurons each
- 1 output layer
- All layers are fully connected.

Notation

$a_i -$ value of the i^{th} input node

w_{ij}^k—weight of the connecting i^{th} node of kth layer and j^{th} node of $k+1^{th} layer$

$b_i^k -$ bias value of the i^{th} node in $k+1^{th}$ layer.

$$x = a^{(1)} \qquad \textit{Input layer}$$

$$z^{(2)} = W^{(1)}x + b^{(1)} \qquad \textit{neuron value at } Hidden_1 \textit{ layer}$$

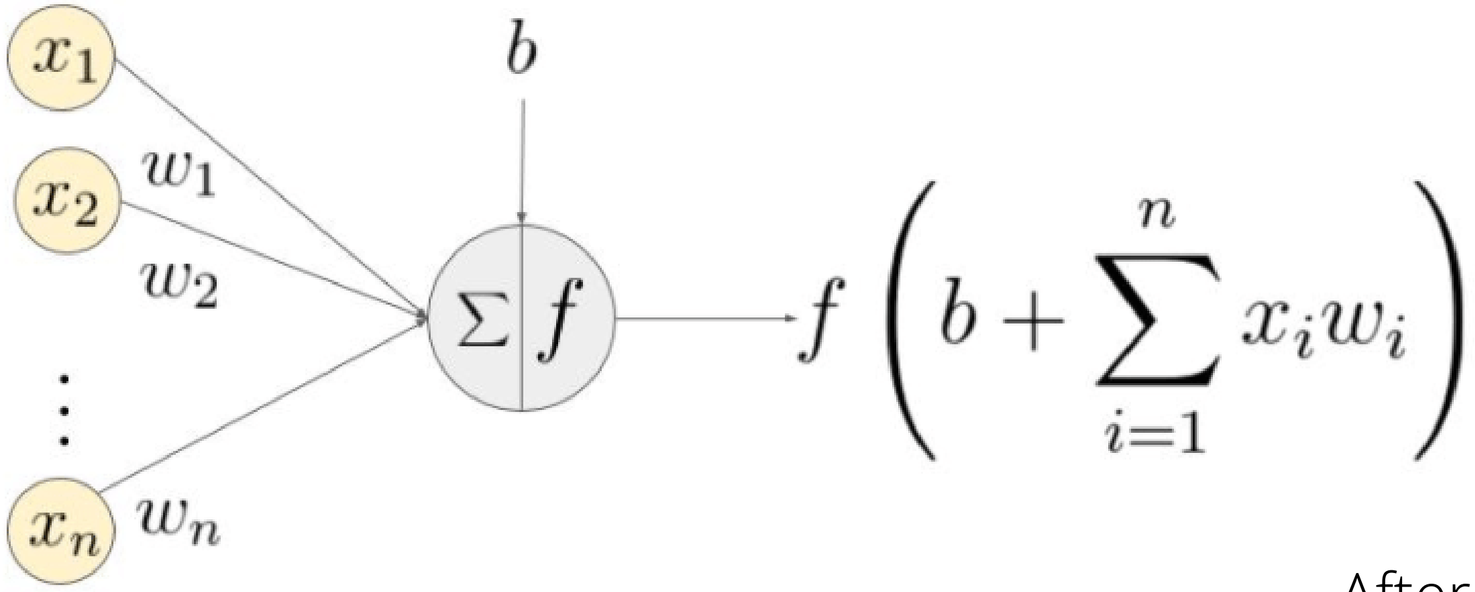
$$a^{(2)} = f(z^{(2)}) \qquad \textit{activation value at } Hidden_1 \textit{ layer}$$

$$z^{(3)} = W^{(2)}a^{(2)} + b^{(2)} \qquad \textit{neuron value at } Hidden_2 \textit{ layer}$$

$$a^{(3)} = f(z^{(3)}) \qquad \textit{activation value at } Hidden_2 \textit{ layer}$$

$$s = W^{(3)}a^{(3)} \qquad \textit{Output layer}$$

Activation Functions



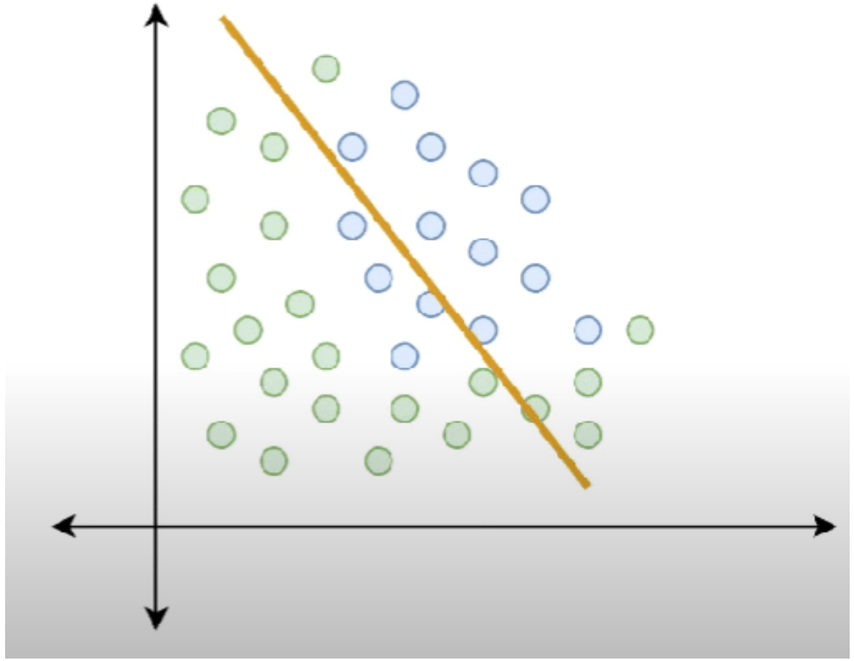
An example of a neuron showing the input (x_1 - x_n), their corresponding weights (w_1 - w_n), a bias (b) and the activation function f applied to the weighted sum of the inputs.

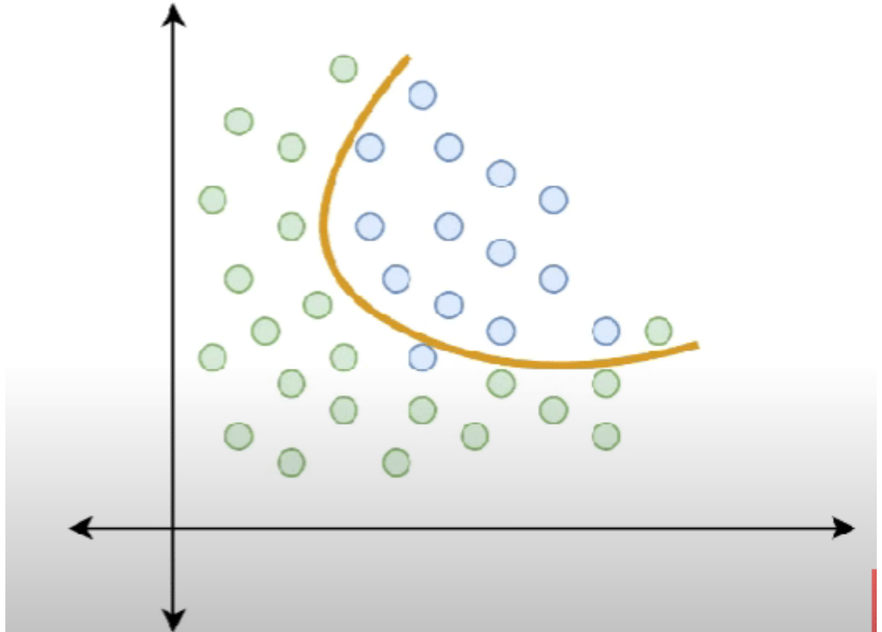
After the weighted sum of outputs of the previous layer is added with the bias, each neuron applies an **activation function** on the resultant value, this value is then used in forward propagation for the next layer

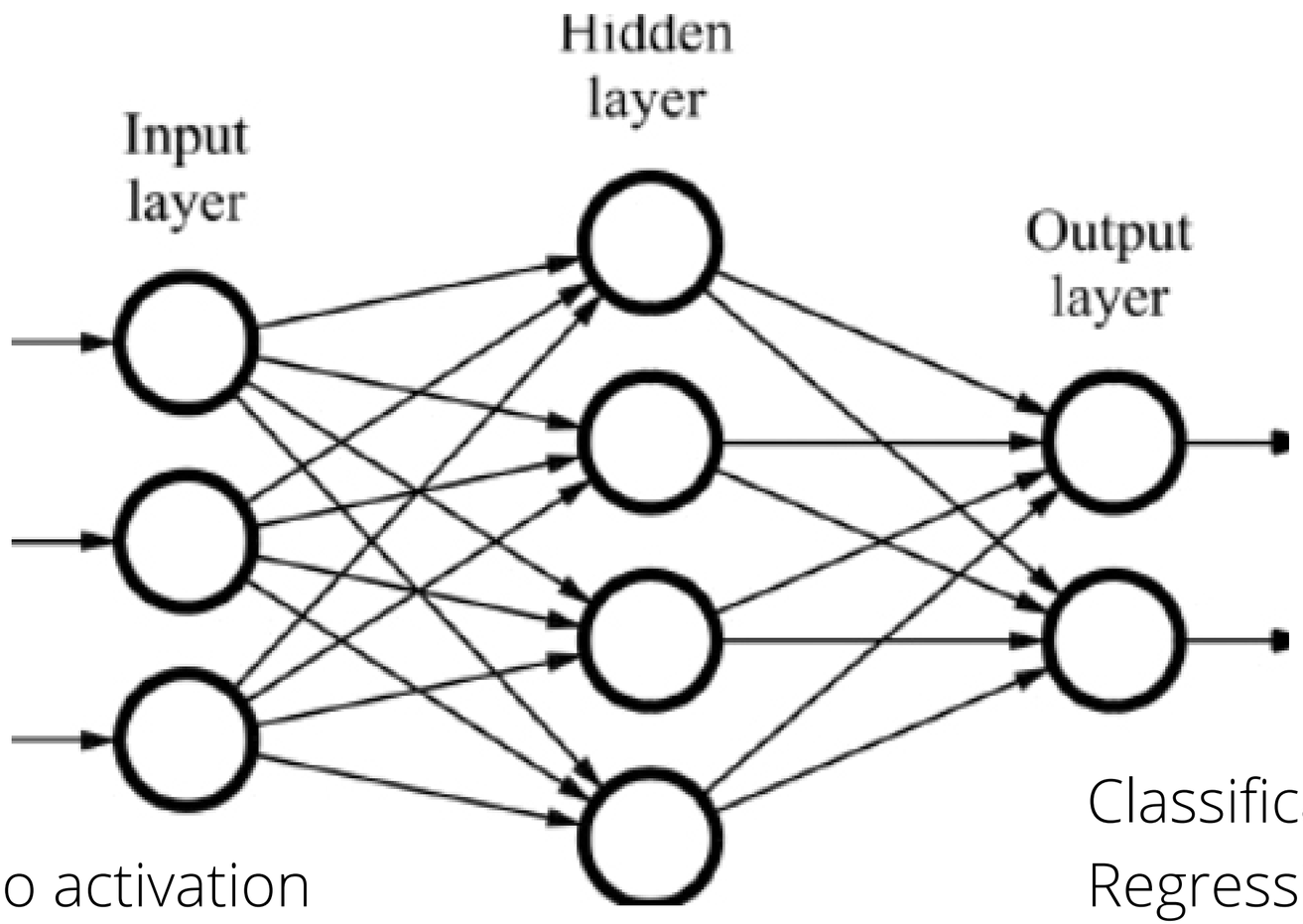
But Why ?

We know that any composition of linear maps is also a linear map, so without any activation function our neural network can only behave as a linear fit model, or in case of classification it can only distuinguish between linearly separable data

WE NEED TO INTRODUCE SOME NON-LINEARITY!







No activation

Classification: Softmax

Regression: No activation

Typically used functions:

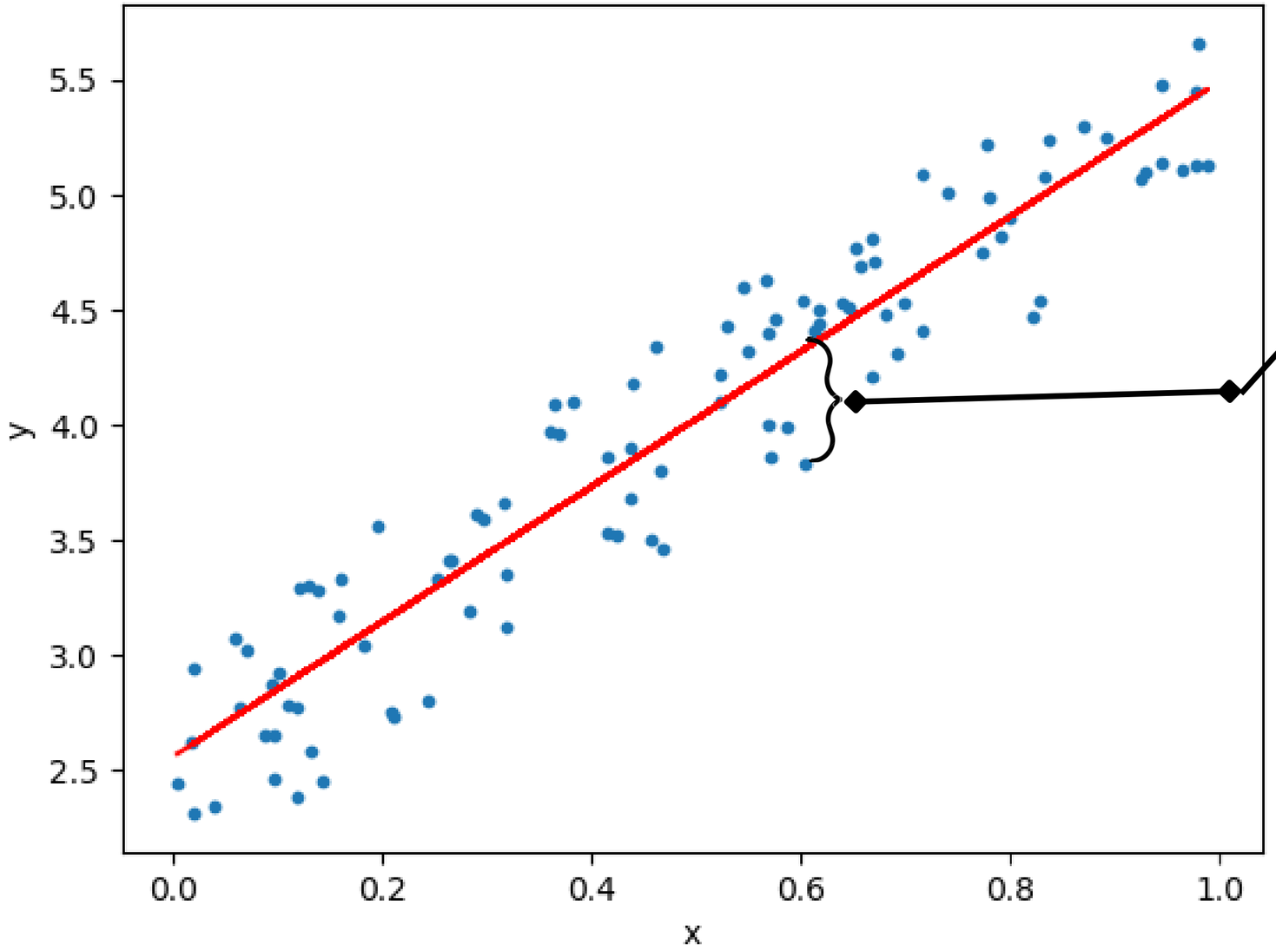
- sigmoid
- tanh (hyperbolic tan)
- ReLU
- Leaky ReLU
- Maxout

Cost function

- A cost function is a measure of how wrong the model is in terms of its ability to estimate the relationship between X and y.
- It is a measure of how far are our estimates/predicted values from the true value.
- In regression,the model predicts an output value for each training data during the training phase

Some of the loss functions for regression problems are:

- Mean error
- Mean squared loss(MSE)
- Mean absolute loss



- Error = y(actual) - y(predicted)

Linear Regression

Mean error

- In this cost function ,the error for each training data is calculated and then the mean value of all of these errors is derived .Calculating mean of the errors is the simplest and most intuitive way possible . But there is a catch .
- The errors can be both negative or positive and during summation, they will tend to cancel each other out. In fact, it is theoretically possible that the errors are such that positive and negatives cancel each other to give zero error mean error for the model.

Y (Actual)	Y' (Predicted)	Error = Y-Y'
10.2	9.4	0.8
3.5	1.7	1.8
7.1	6.9	0.2
14.5	15.4	-0.9
17.2	18.4	-1.2
9.5	11.3	-1.8
2.7	2.5	0.2
11.5	11.1	0.4
5.9	6.7	-0.8
15.3	15.2	0.1
Sum		-1.2
Mean Error		(-1.2/10) = -0.12

Mean Absolute Error (MAE)

- This addresses the shortcoming of ME. Here an absolute difference between the actual and predicted value is calculated to avoid any possibility of negative error. MAE loss is also called L1 loss.

$$MAE = \frac{\sum_{i=1}^{n} | y_i - \hat{y}_i |}{n}$$

Y (Actual)	Y' (Predicted)	Error = |Y-Y'|
10.2	9.4	0.8
3.5	1.7	1.8
7.1	6.9	0.2
14.5	15.4	0.9
17.2	18.4	1.2
9.5	11.3	1.8
2.7	2.5	0.2
11.5	11.1	0.4
5.9	6.7	0.8
15.3	15.2	0.1
Sum		8.2
Mean Absolute Error		(8.2/10) = 0.82

Mean Squared Error (MSE)

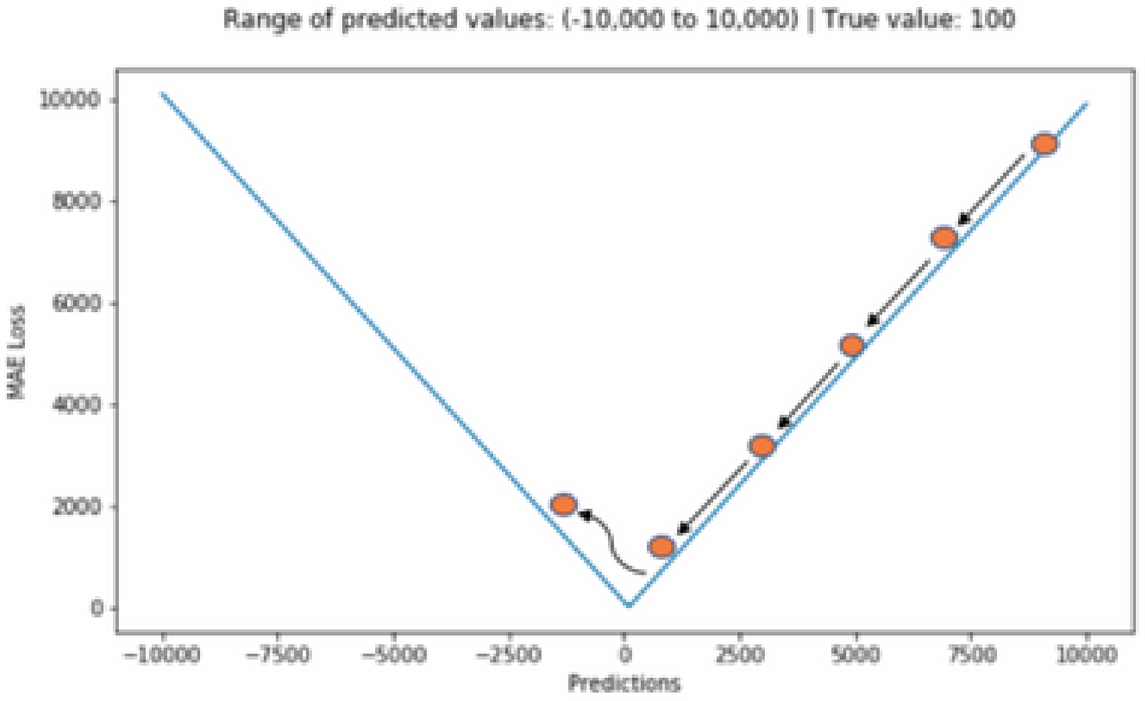
Here the square of the difference between the actual and predicted value is calculated to avoid any possibility of negative error. MSE Loss is also called L2 loss.

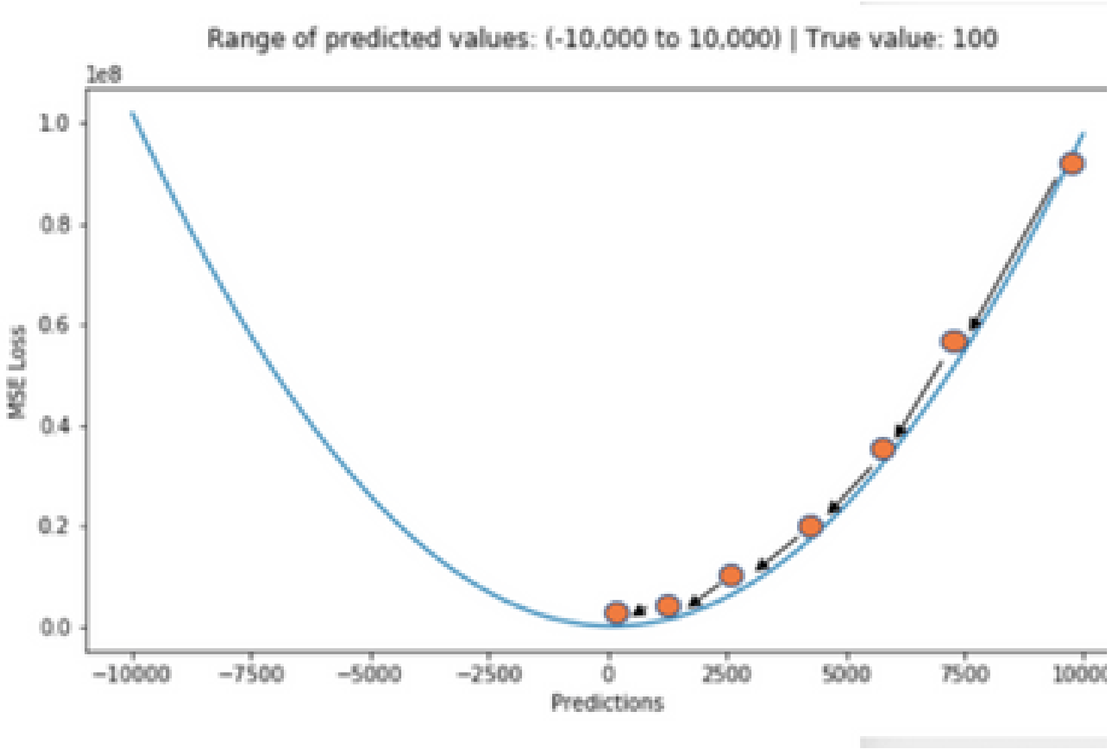
$$MSE = \frac{\sum_{i=1}^{n}(y_i - \hat{y}_i)^2}{n}$$

Y (Actual)	Y' (Predicted)	Error = $(Y-Y')^2$
10.2	9.4	0.64
3.5	1.7	3.24
7.1	6.9	0.04
14.5	15.4	0.81
17.2	18.4	1.44
9.5	11.3	3.24
2.7	2.5	0.04
11.5	11.1	0.16
5.9	6.7	0.64
15.3	15.2	0.01
Sum		10.26
Mean Square Error		(10.26/10) = 1.02

- L1 loss is more robust to outliers than L2, or we can say that when the difference is higher ,L1 is more stable than L2

- L2 loss is more stable than the L1 loss, especially when the difference is higher ,L1 is more stable than L2


Range of predicted values: (-10,000 to 10,000) | True value: 100
MAE Loss
Predictions
10000
8000
6000
4000
2000
0
−10000
−7500
−5000
−2500
0
2500
5000
7500
10000


Range of predicted values: (-10,000 to 10,000) | True value: 100
1e8
MSE Loss
Predictions
1.0
0.8
0.6
0.4
0.2
0.0
−10000
−7500
−5000
−2500
0
2500
5000
7500

Classification loss function

Cost functions used in classification problems are different than what we saw in the regression problem above. There is a reason why we don't use regression cost functions for classification problem and we will see it later. But before that let us see the classification cost functions.

The most commonly used loss function is cross entropy

CrossEntropy(A,P) = – (y1*log(y1') + y2*log(y2') + y3*log(y3') + … + yM*log(yM'))

We cant use MSEloss for classfication and hence we use methods like cross entropy

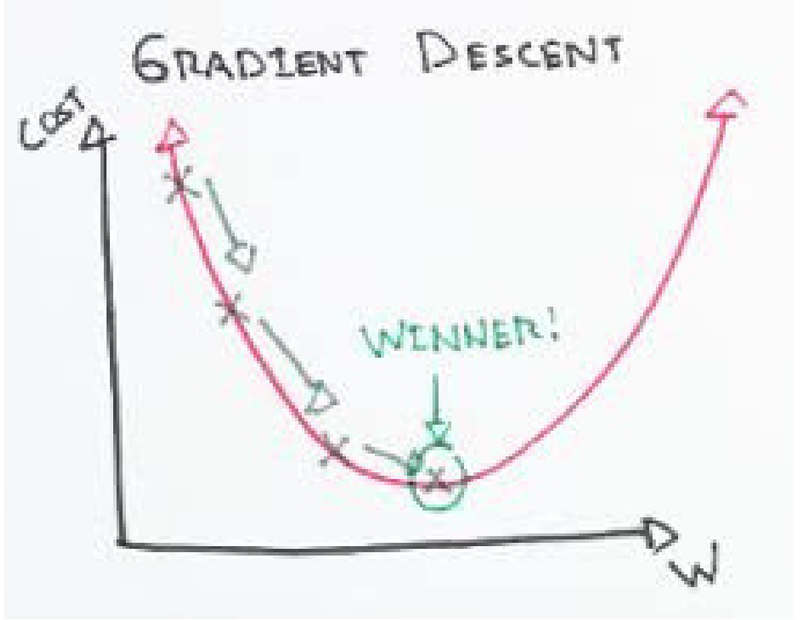
GRADIENT DESCENT

But what is Gradient Descent?

The main aim of a neural network is to achieve maximum accuracy, and minimum deviation of the predicted value from the actual value... But how do we achieve this? How can we go from some initial value to the final desired value of minimum error? Gradient Descent is an optimization technique in ML that allows us to reach the minima of the Loss/Cost Function in an effective way by taking help from the mathematical concept of Gradients (thus the name!)!

In the adjacent figure, we have a graph of the 'Cost Function' Vs 'Weight' for visual reference. Initially, we feed some random values for weights and biases in the neural network, and thus start off from some random point on the graph. With these values we get some output value, and then we calculate the Loss with the help of Loss Function. With the help of this value of Loss, we update the weights and biases with a suitable formula, and the same process of calculation of Loss and update of weights and biases keeps going on until we reach the minima. Each round of this process is given a fancy name called 'Epoch'.

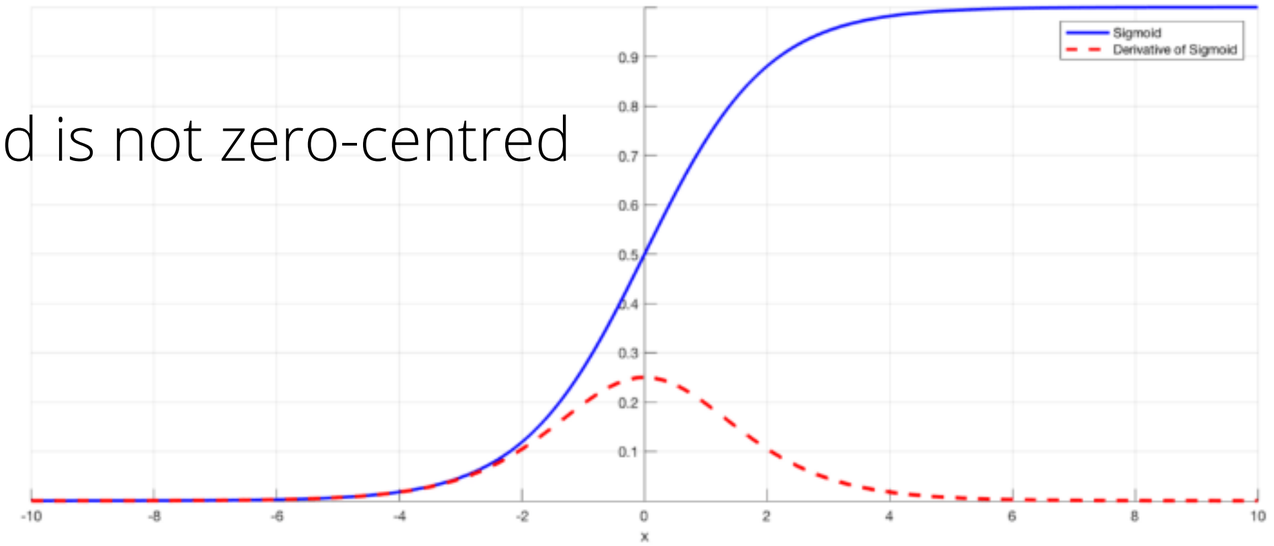
So... what is this formula? Must be complex and scary? No! In fact, it is a simple and a quite intuitive formula which we will understand now!



Back-Propogation Algorithm

- Our aim is to tune the weights of the neural network such that it minimizes the desired cost function and behaves as the required map
- In each iteration/epoch outputs are calculated using forward propagation with the current set of weights and the error with the training samples is used as a feedback to determine how much to adjust the weights in the next iteration
- This process is repeated until the rate of change of error with epochs is no longer noticeable or alternately if there is no convergence (in case of an error)

Sigmoid is not zero-centred

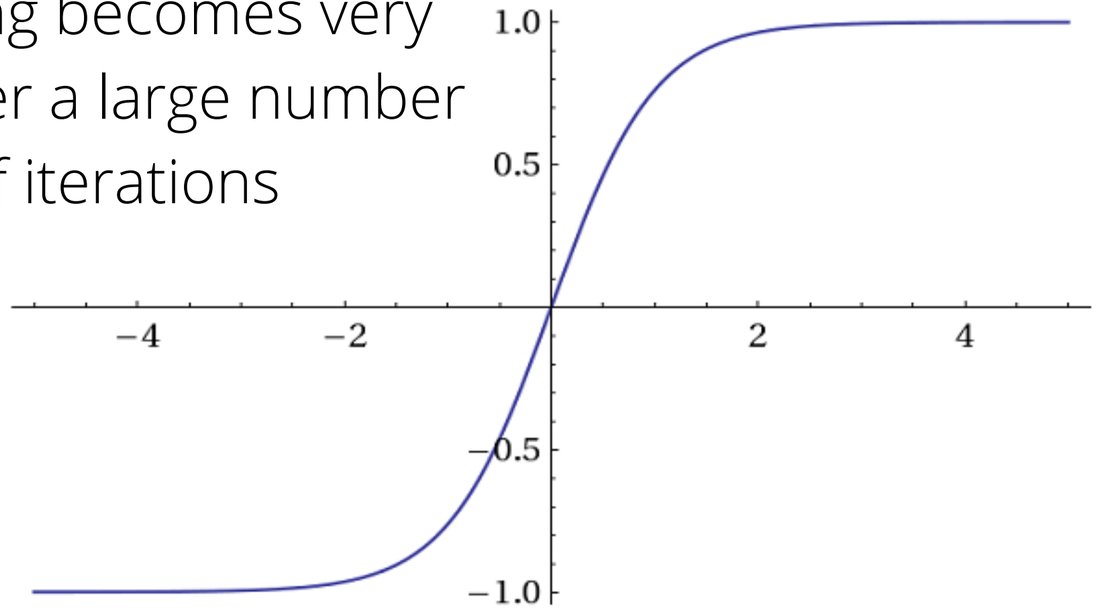


$$g(x) = \frac{e^x}{1 + e^x}$$

$$tanh(x) = 2g(2x) - 1$$

$$tanh(x) = \frac{e^x - e^{-x}}{e^x + e^{-x}}.$$

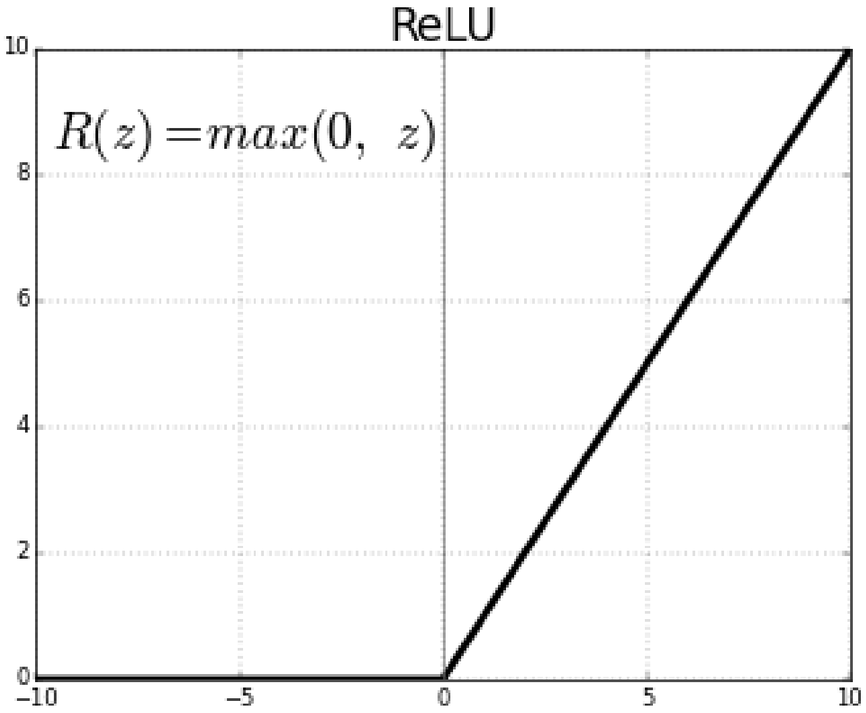
Learning becomes very slow after a large number of iterations

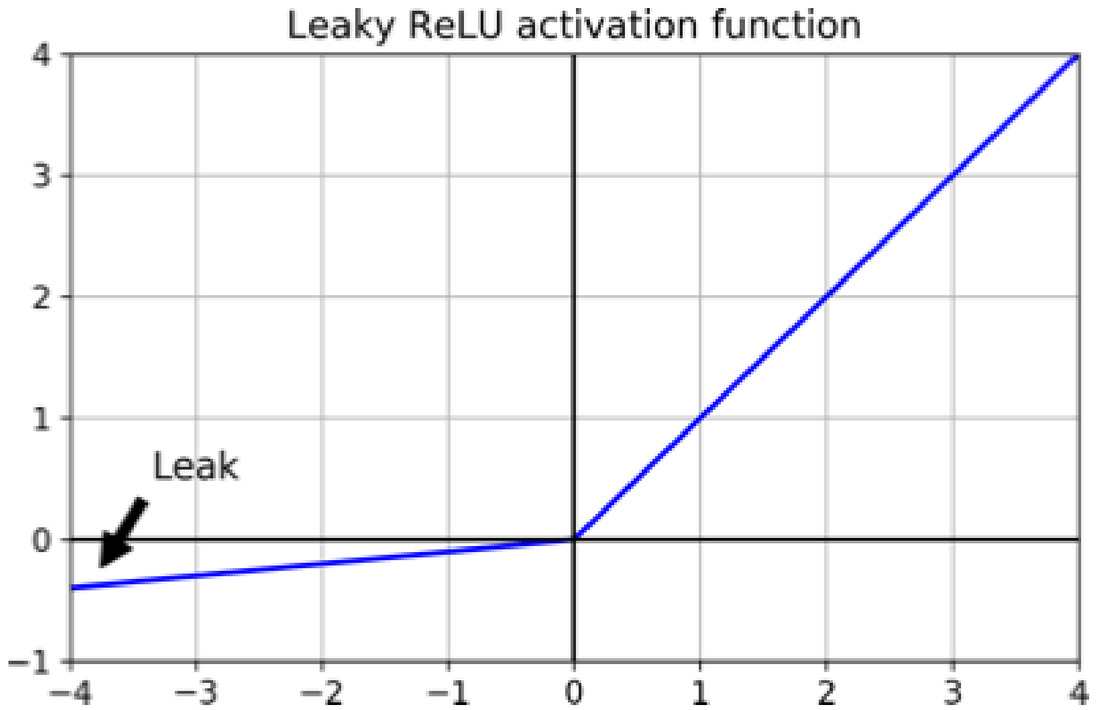


Disadvantages:

- Saturation and vanishing gradient at large values
- This is further compounded by chain rule as the gradients of one layer are recursively used to calculate gradients of the previous layer (problem in a large nn)

Alternatives





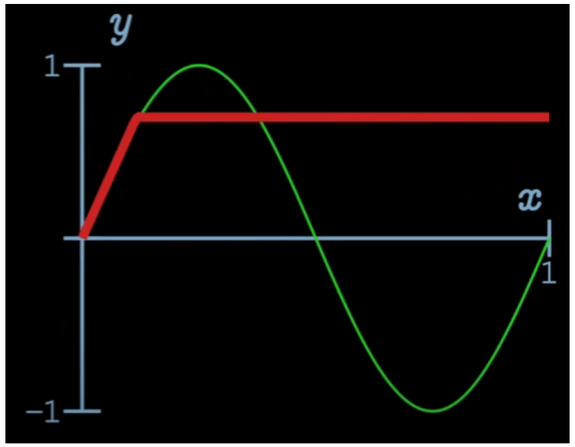
- Adv: Very fast to train as gradient is easy to calculate
- Disadv: Can result in dead neurons (learning stops)

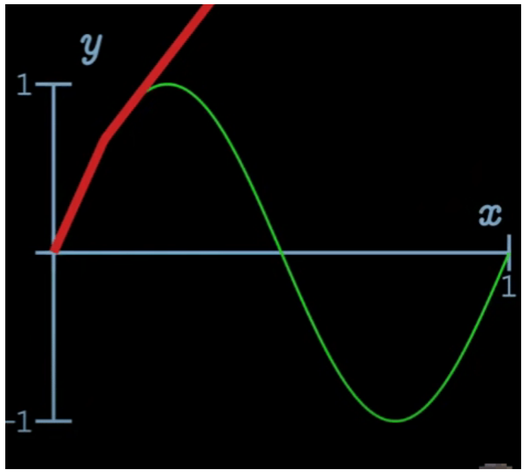
Leaky ReLU has a small non-zero slope when negative

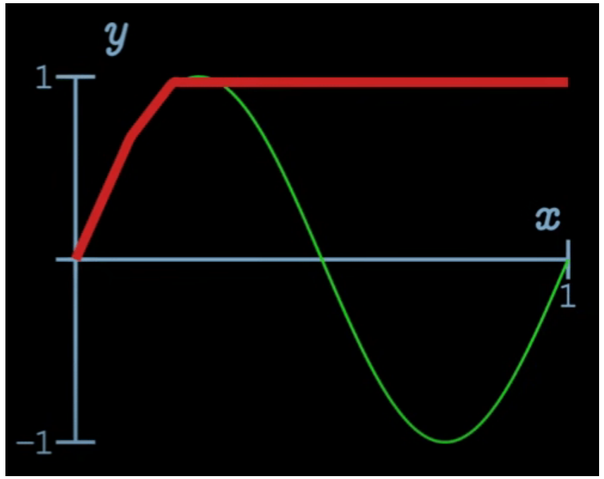
How does ReLU introduce non-linearity?

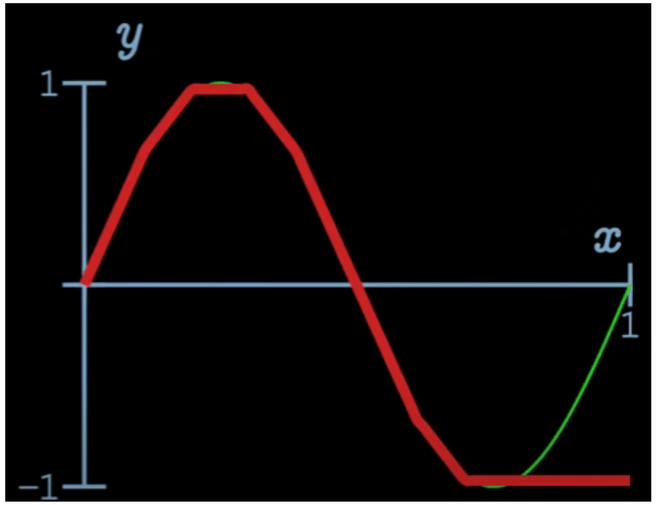
Linear compositions of ReLU's can approximate any non-linear function using piece-wise linear functions

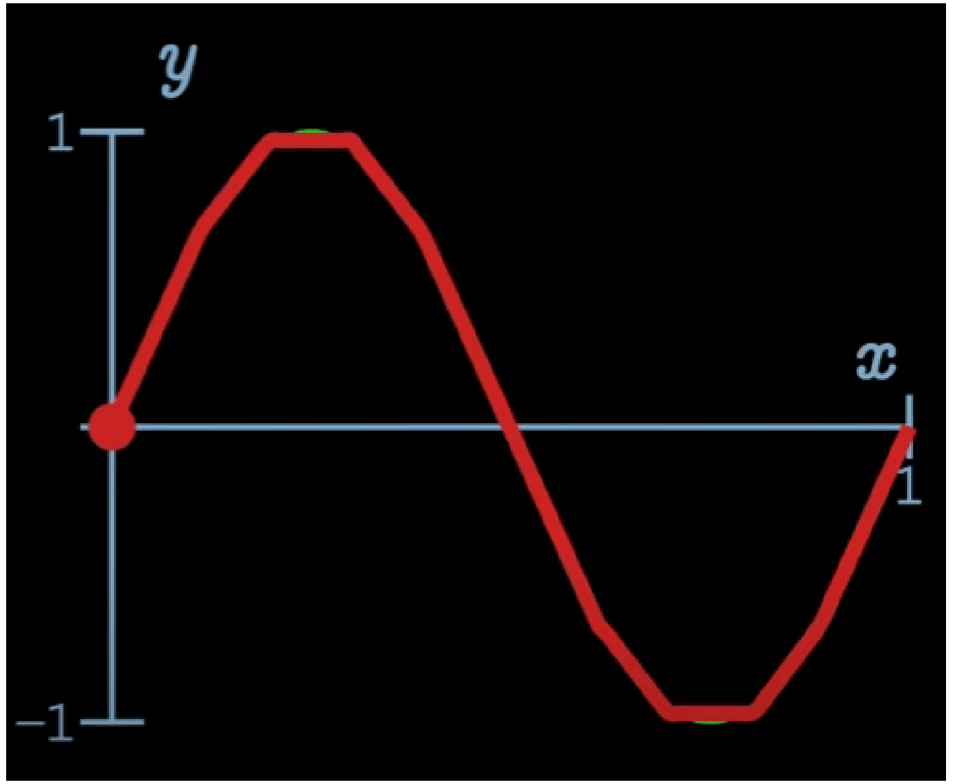
Training Process



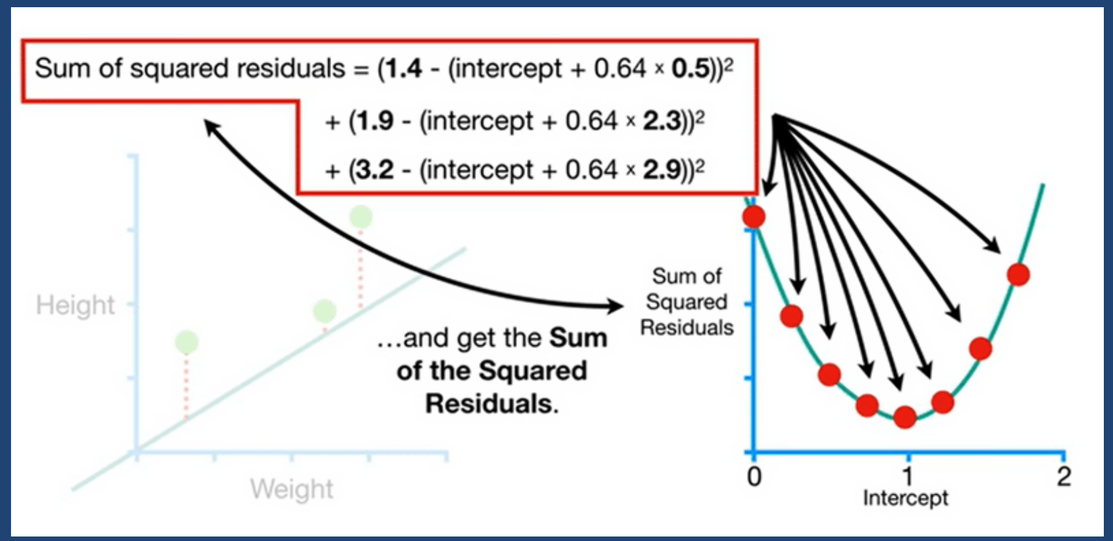


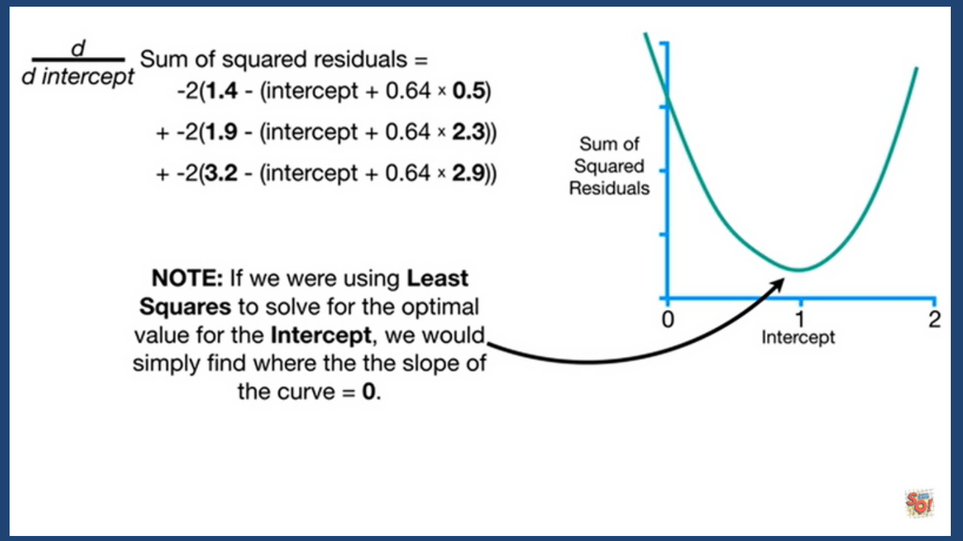


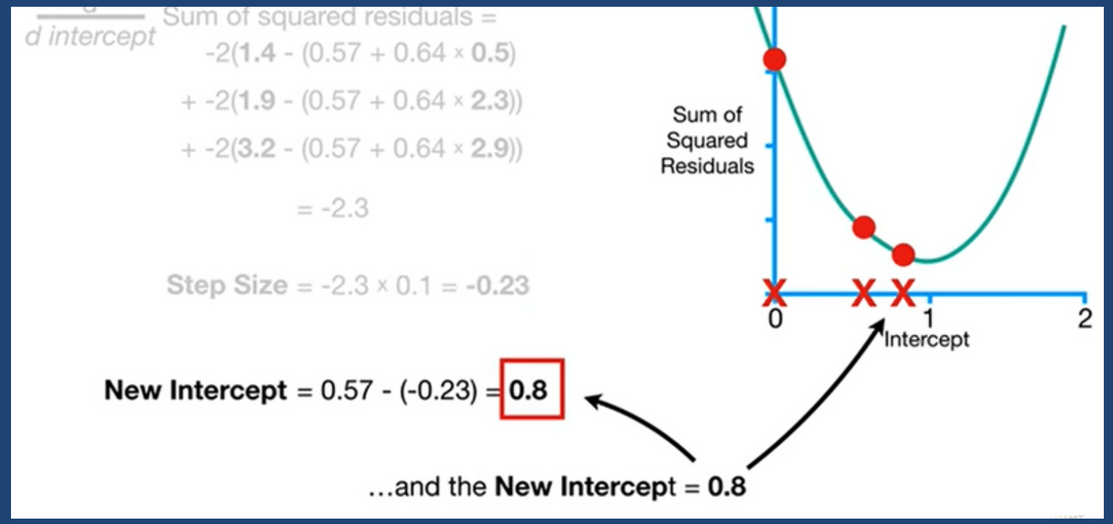


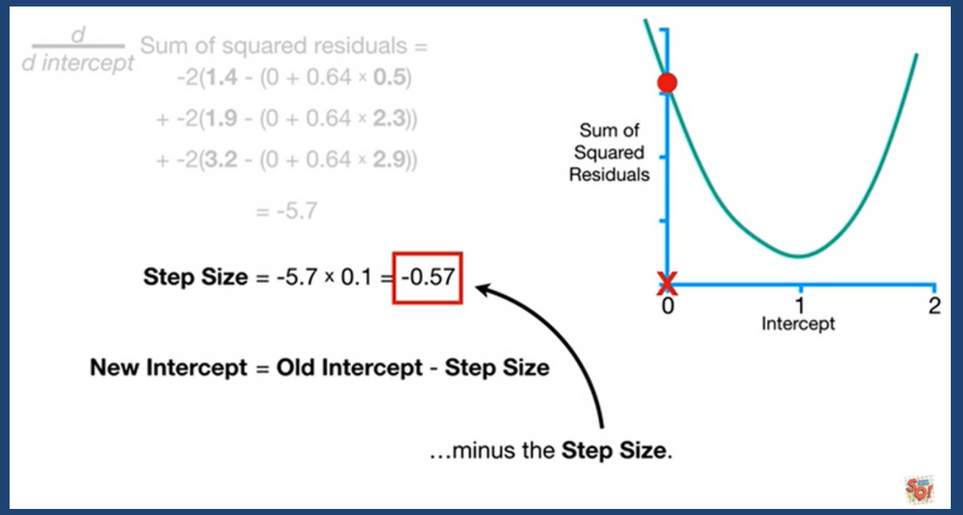


Gradient Descent









$$X = X - lr * \frac{d}{dX} f$$

Where,

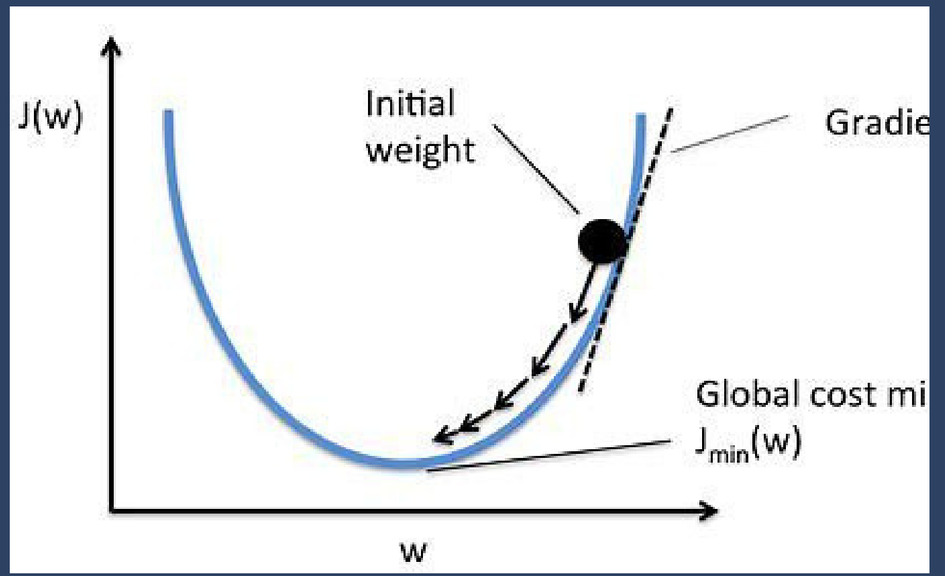
X = *input*

$F(X)$ = *output based on X*

lr = *learning rate*

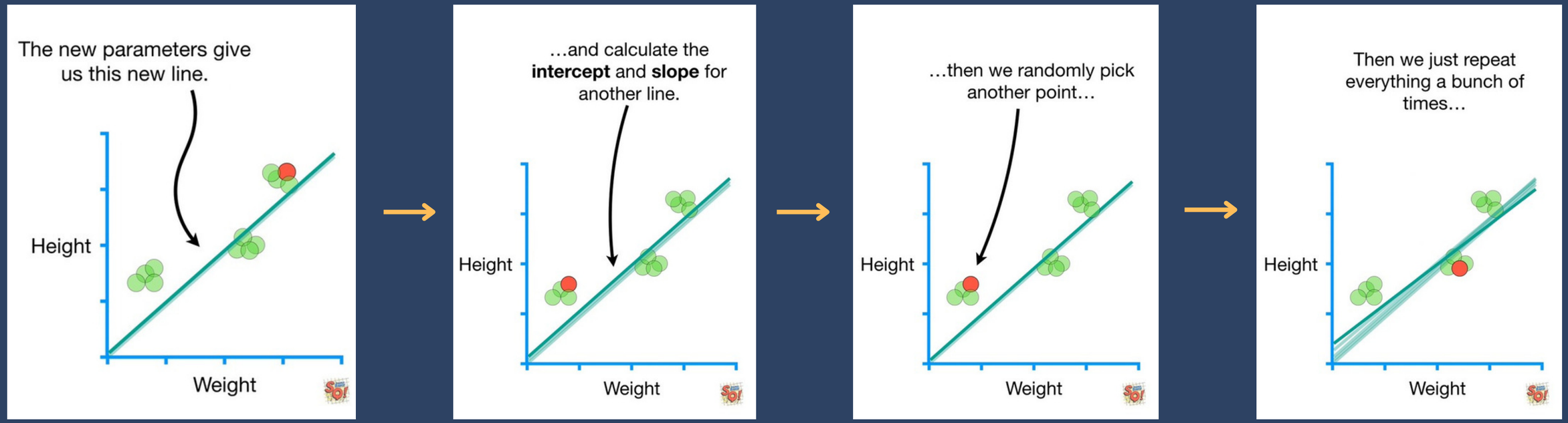
- So this is the formula!
- The X on the left is the value of new/updated input value (of weights/biases), and the X on the right is the old input value.
- The 'lr' term is called the learning rate, to which we will come shortly, and the derivative term is the gradient of the Loss Function with respect to X, i.e the input variable in consideration.

- The learning rate is the measure of length of steps that we take while going from the X(old) to X(new).
- The learning rate should not be too high, as with big steps, though we might reach close to the minima in lesser time, we will never ever be able to reach the desired accuracy. Thus, there has to be a trade-off between accuracy and amount of time required to reach the minima.
- With the need for a trade-off came various methods of optimization, with and without use of gradient descent!



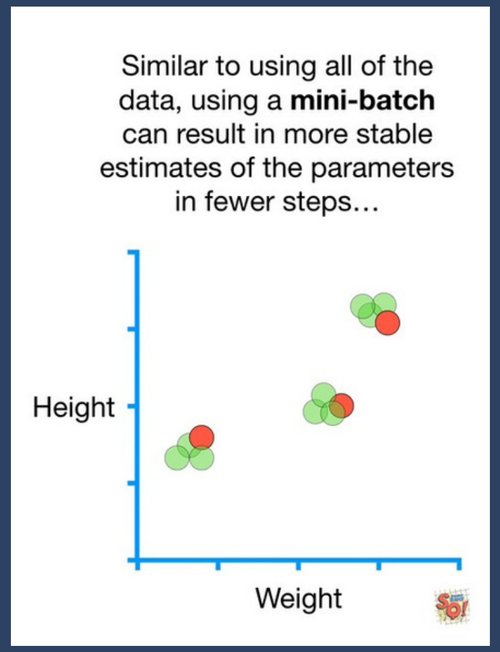
Stochastic Gradient Descent

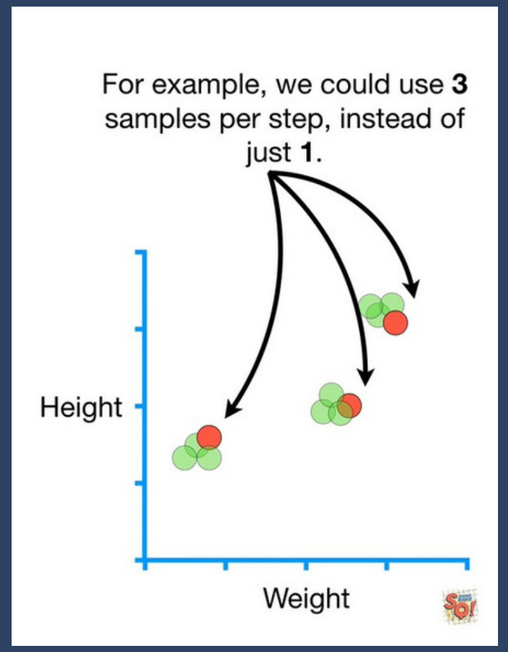
- Stochastic Gradient Descent is used particularly when the number of data points are large in number or their are redundancies (like clusters).
- Instead of having complicated gradient terms involving all the variables at once, we take just 1 point into consideration every time while running an epoch and calculate the Loss Function.
- Let us try to understand this with the help of a simple example.

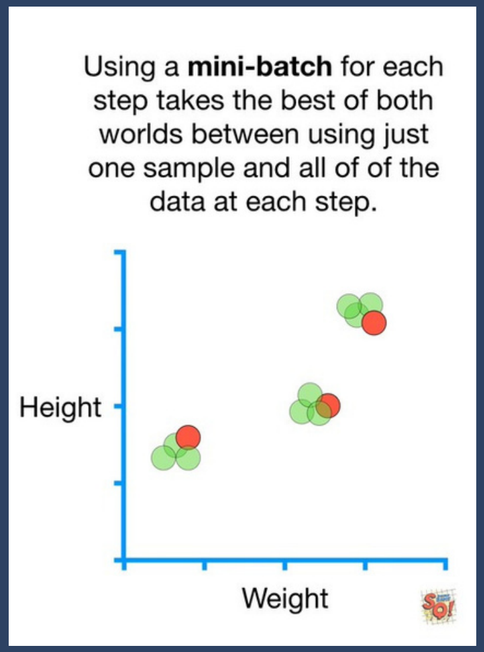


- The previous example is of a model, that predicts the the height of a person, given the value of his/her weight.
- Now, our aim is to obtain a line, that would most accurately predict the Heights based on the weights.
- We assume some initial random line, say with intercepts (0,0) and slope 1, and we start of with a single point belonging to one of the clusters and apply the formula of gradient descent to it.
- After the calculation of the loss, we update the value of the slope and intercepts, and then run the formula for gradient descent on some point from another cluster and again update the value of slope and intercepts.
- This goes on until we reach the minima of the Loss Function.

Mini-Batch Gradient Descent







Adaptive Learning Methods

- As the name suggests, Adaptive Learning Methods adapt to suitable changes during every epoch.
- AdaGrad, RMSProp, Adam, etc. are some of the adaptive learning methods that modify the learning rate at every epoch.
- Learning Rates should be changed in accordance wi
  th the need that cost function should be minimized inminimum number of steps, minimum amount of time and without over-shooting.
- •The various methods mentioned above keep modifying the value of learning rate in accordance with certain formulae, that help the model learn faster with good efficiency.
- As these methods are mathematical, we will come to them in more details during their implementation.

OTHER OPTIMIZERS

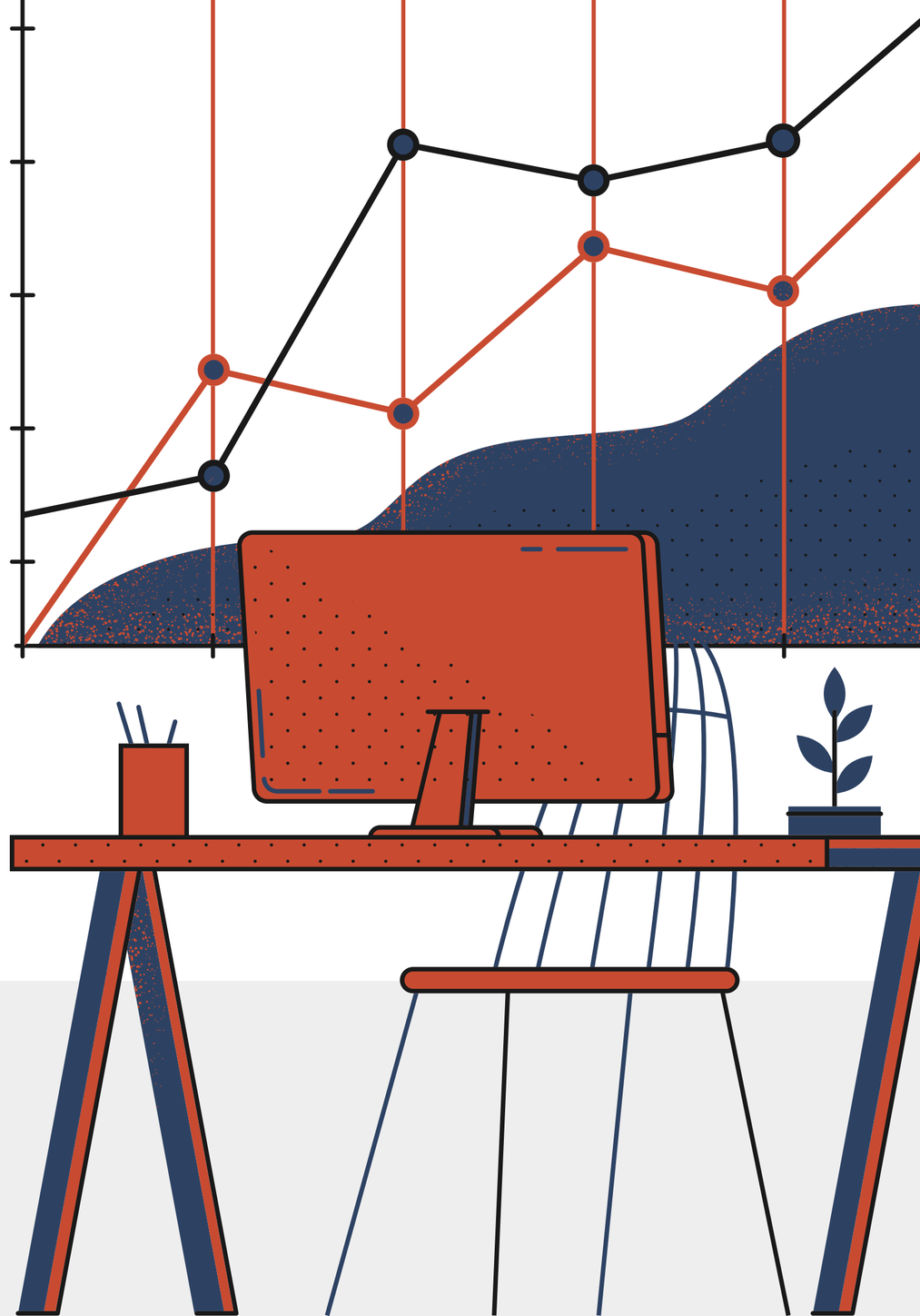


Table of Contents

AdaGrad Optimizer

$$\text{SGD} \Rightarrow \; w_t = w_{t-1} - \eta \frac{\partial L}{\partial w_{t-1}}$$

$$\text{Adagrad} \; \Rightarrow w_t = w_{t-1} - \eta_t^{'} \frac{\partial L}{\partial w_{t-1}}$$

Disadvantage: alpha continuously increases with iterations and after a large number of iterations, learning rate becomes low for all parameters

$$\text{where } \eta_t^{'} = \frac{\eta}{\sqrt{\alpha_t + \varepsilon}}$$

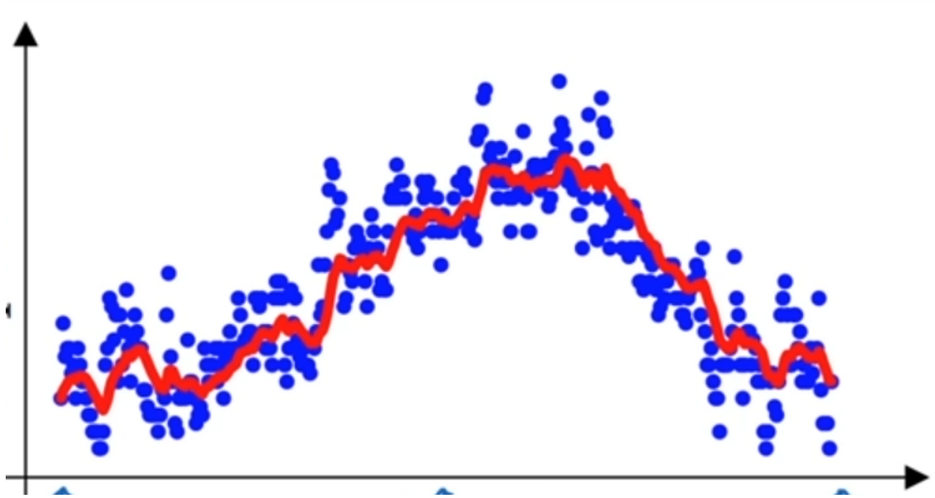
$$\varepsilon \text{ is a small } +\text{ve number to avoid divisibilty by } 0$$

Accumulates history of gradients from previous iterations allowing weights with lower gradients to have higher learning rates and vice-versa

$$\longrightarrow \; \alpha_t = \sum_{i=1}^{t} \left(\frac{\partial L}{\partial w_{t-1}}\right)^2 \quad \textit{summation of gradient square}$$

- In grad descent, certain weights may change very slowly, due to their gradient being very small
- Whereas other weights may change at a faster rate, this could cause a rapid divergence in their values

Exponentially weighted moving averages (EMWA)



Consider the following noisy data (blue dots)

$$V_t = \beta V_{t-1} + (1-\beta) S_t$$
$$\beta \in [0,1]$$

Formula for exponentially weighted moving average

Vt -> Tth average

St -> current value

$$V_t = \beta V_{t-1} + (1-\beta) S_t$$
$$V_{t-1} = \beta V_{t-2} + (1-\beta) S_{t-1}$$
$$V_{t-2} = \beta V_{t-3} + (1-\beta) S_{t-2}$$

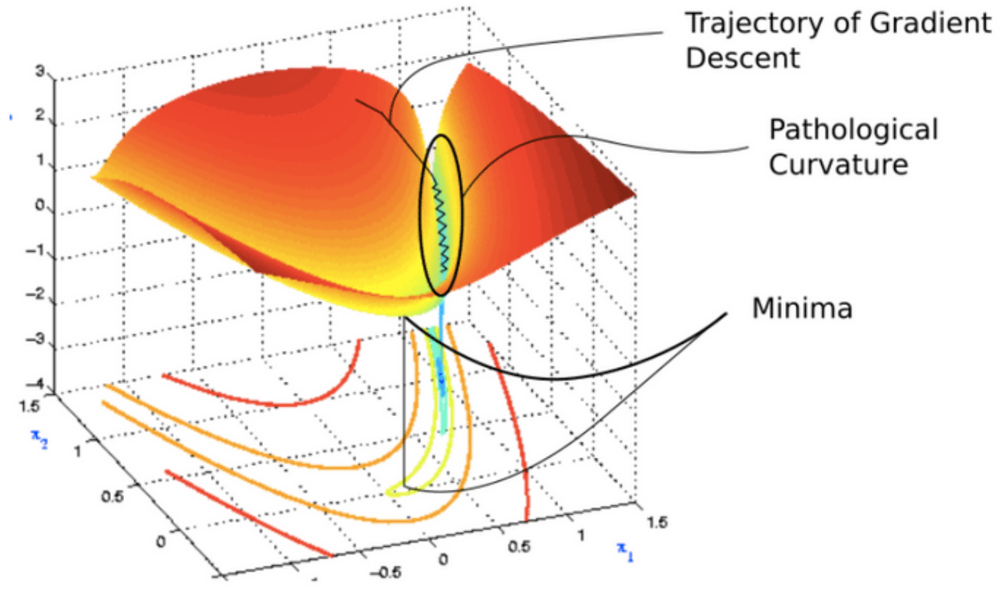
$$V_t = \beta(\beta(\beta V_{t-3} + (1-\beta) S_{t-2}) + (1-\beta) S_{t-1}) + (1-\beta) S_t$$

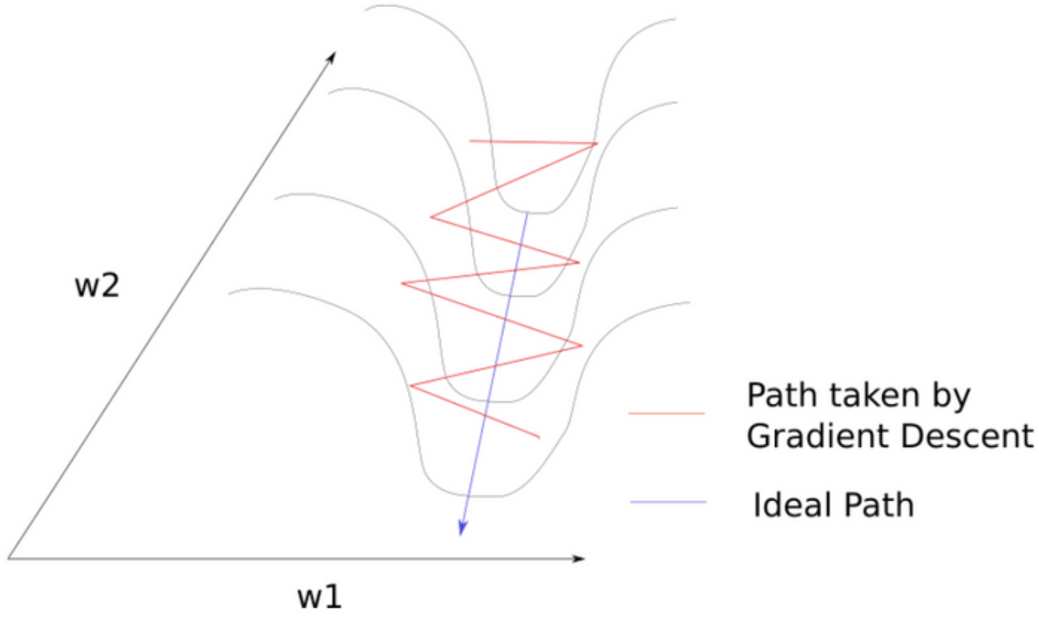
On simplifying ,we get

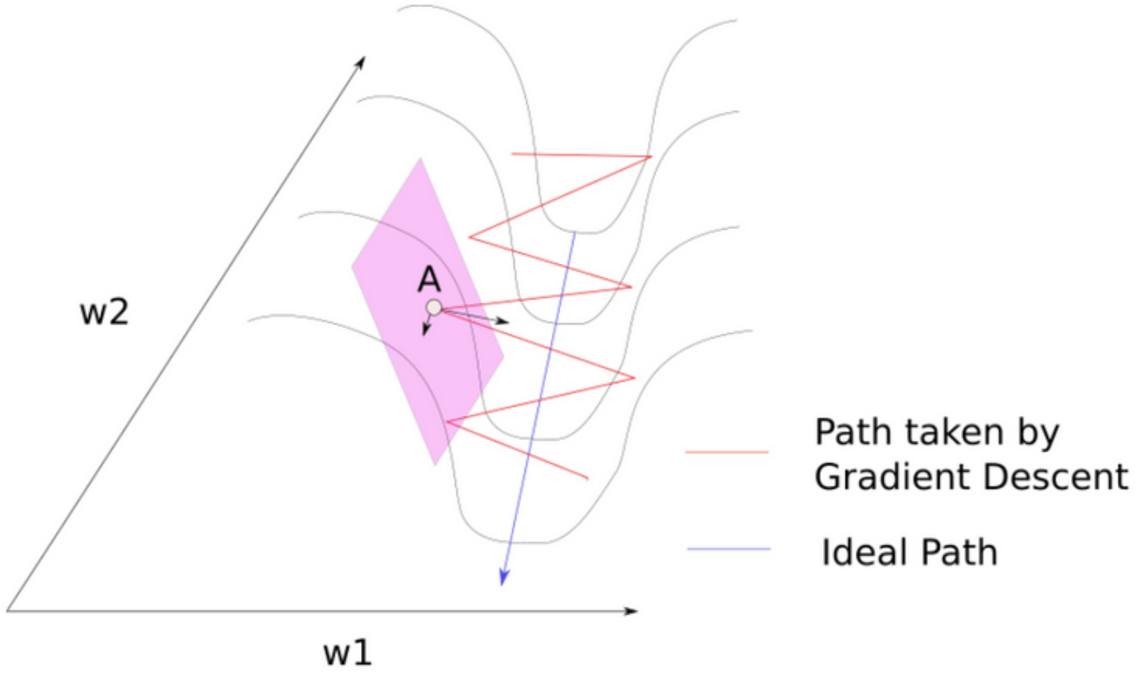
$$V_t = \beta\beta(1-\beta) S_{t-2} + \ldots + \beta(1-\beta) S_{t-1} + \ldots + (1-\beta) S_t$$

Momentum

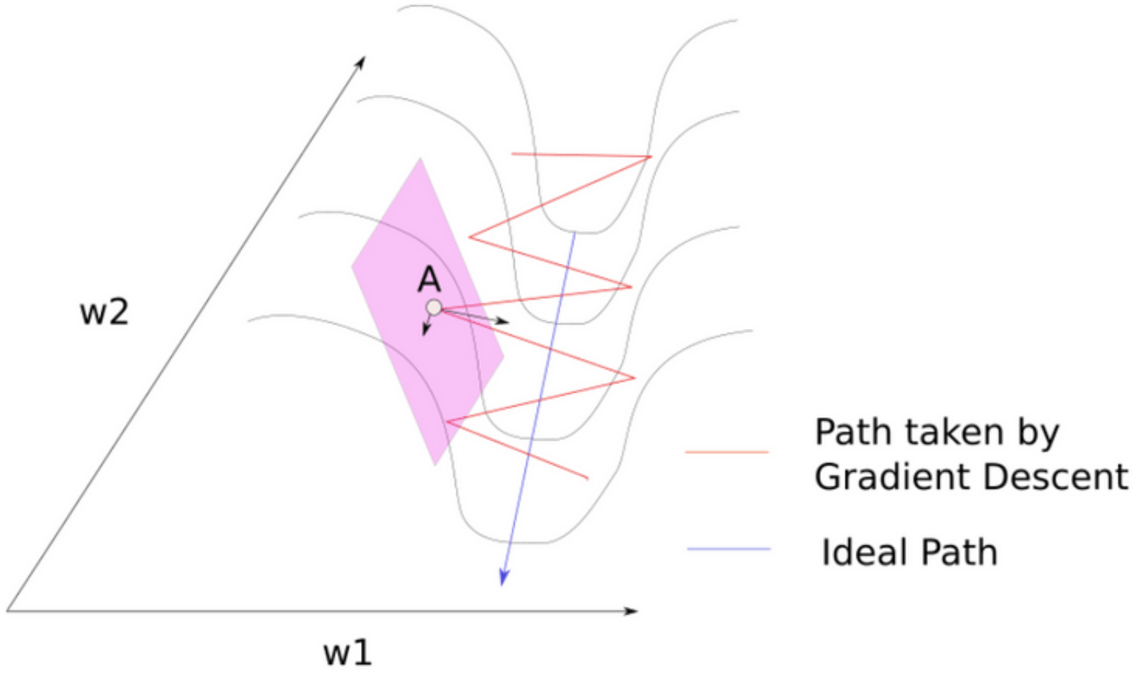
Consider the 3d plot of the cost function. Assume that this cost function depends on 2 parameters w1 and w2.







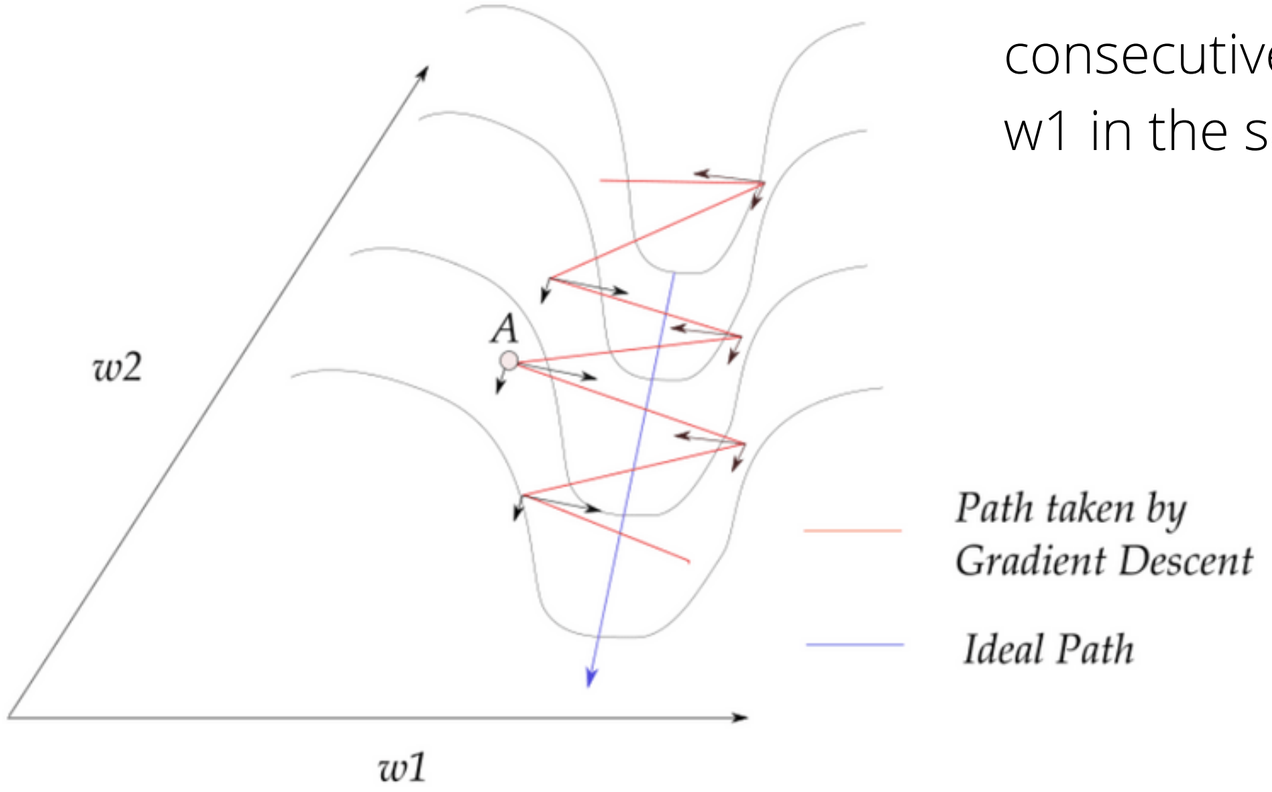
The trajectory in red colour is the path taken by SGD



The trajectory in red colour is the path taken by SGD

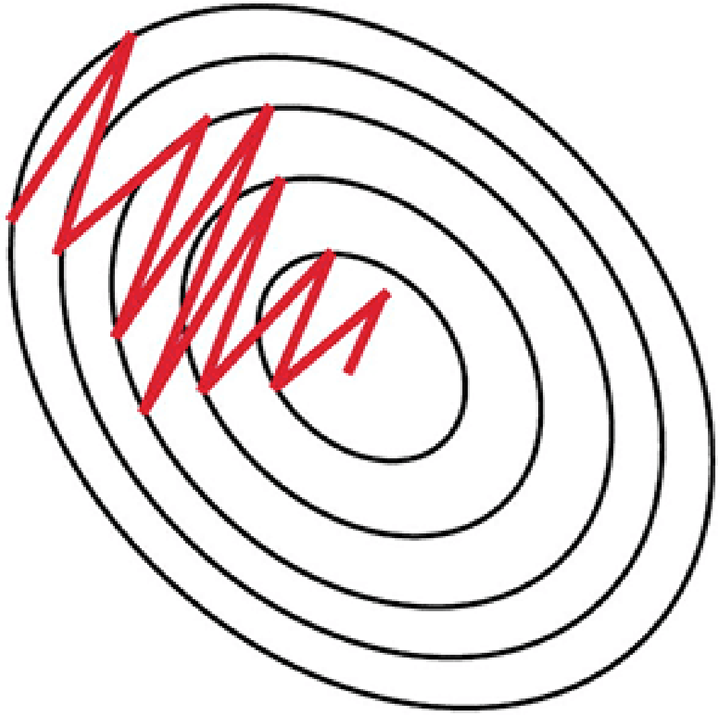
$$V_t = \beta V_{t-1} + (1-\beta)\nabla_w L(W, X, y)$$
$$W = W - \alpha V_t$$

why does it work?

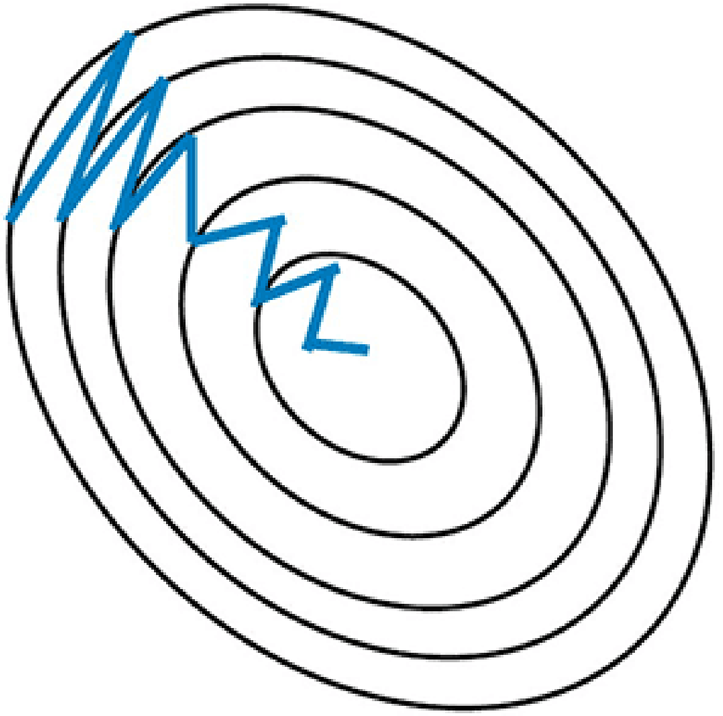


Notice that components of the gradient along w1 at consecutive points are opposite in direction and along w1 in the same direction

And hence while taking the EWMA components along w1 are cancelled out and components along w2 reinforced.



Stochastic Gradient Descent **withhout** Momentum



Stochastic Gradient Descent **with** Momentum

RMSProp

This optimizer is very similar to Adagrad except for the fact that this uses EMWA of squared gradients instead of just the actual sum of all previous gradients in the learning rate expression.

This eliminates the problem of rapidly slowing down when moving very close to the minima.

$$v_t = \beta v_{t-1} + (1-\beta)\left[\frac{\partial L}{\partial w_t}\right]^2$$

$$w_{t+1} = w_t - \frac{\alpha}{\sqrt{v_t + \epsilon}} \cdot \frac{\partial L}{\partial w_t}$$

Fun Fact!

RMSprop, has an interesting history. It was devised by the legendary Geoffrey Hinton, while suggesting a random idea during a Coursera class.

Ada Delta

Like RMSprop, Adadelta is also another improvement from AdaGrad, focusing on the learning rate component. Adadelta is probably short for ‘adaptive delta’, where delta here refers to the difference between the current weight and the newly updated weight.

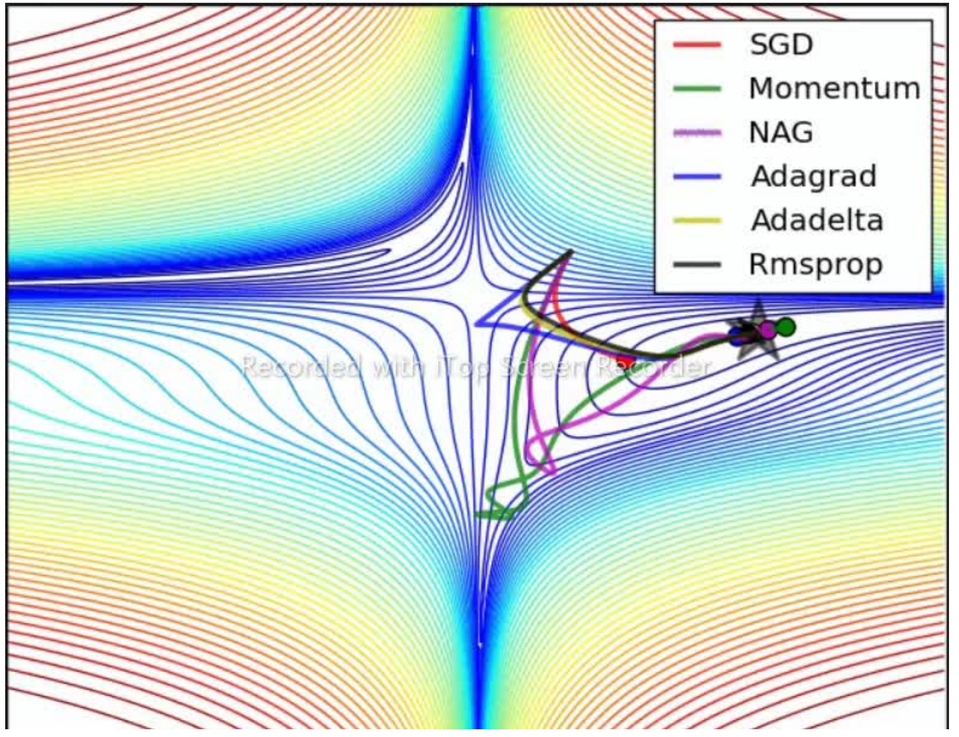
The difference between Adadelta and RMSprop is that Adadelta removes the use of the learning rate parameter completely by replacing it with *D,* the exponential moving average of squared *deltas*.

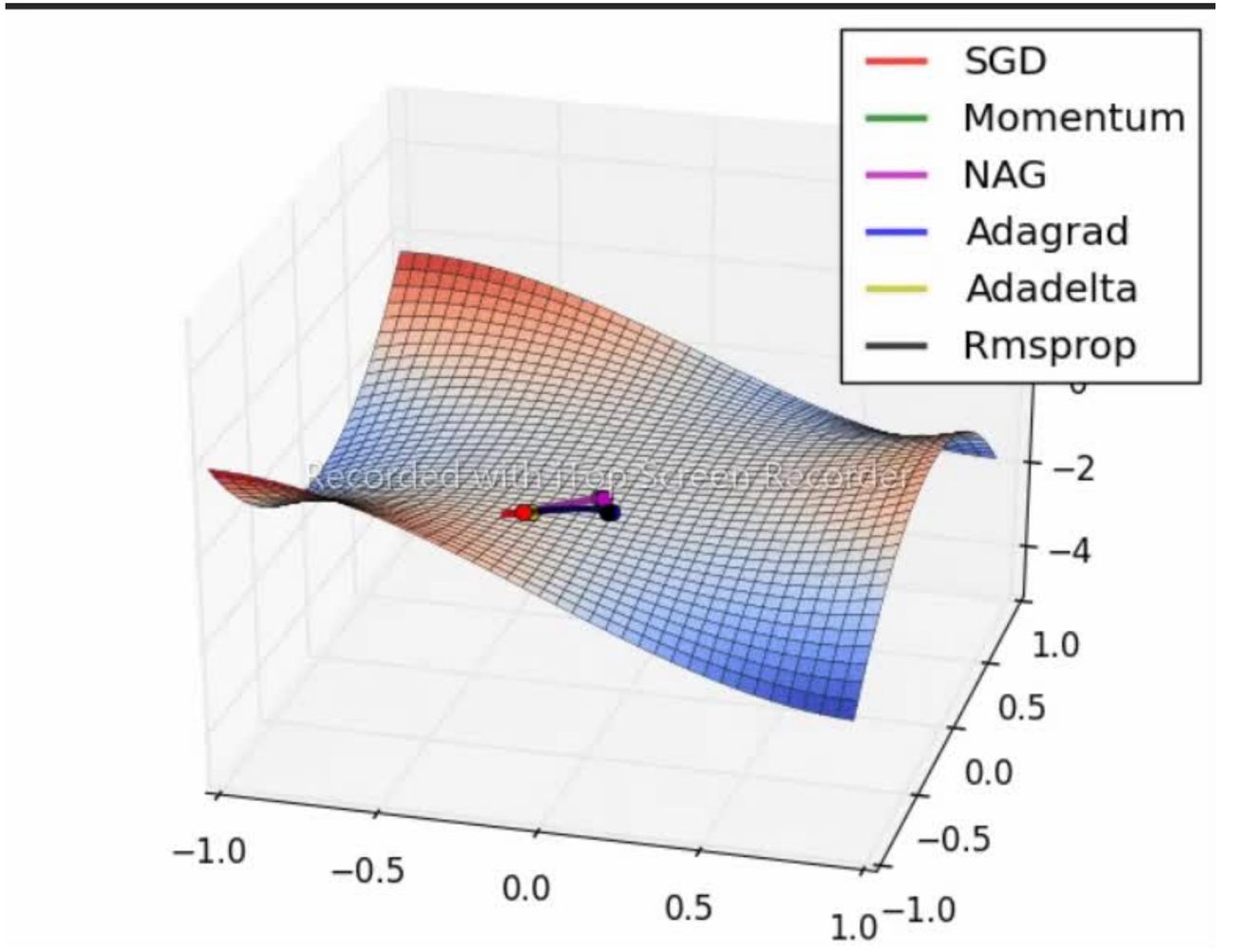
$$\Delta w_t = w_t - w_{t-1}$$

$$D_t = \beta D_{t-1} + (1-\beta)[\Delta w_t]^2$$

$$v_t = \beta v_{t-1} + (1-\beta)\left[\frac{\partial L}{\partial w_t}\right]^2$$

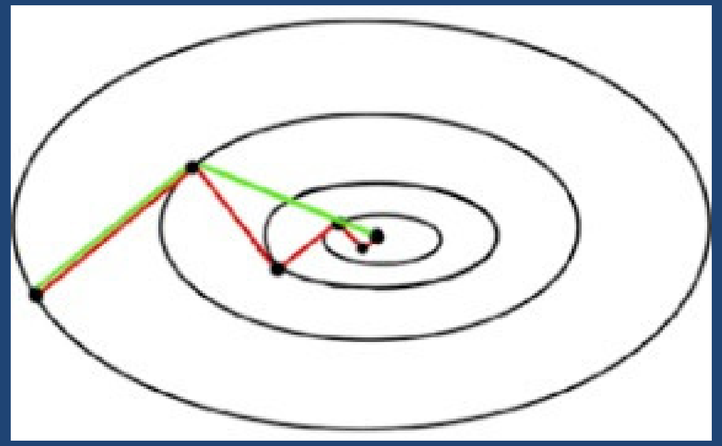
$$w_{t+1} = w_t - \frac{\sqrt{D_{t-1}+\epsilon}}{\sqrt{v_t+\epsilon}} \cdot \frac{\partial L}{\partial w_t}$$

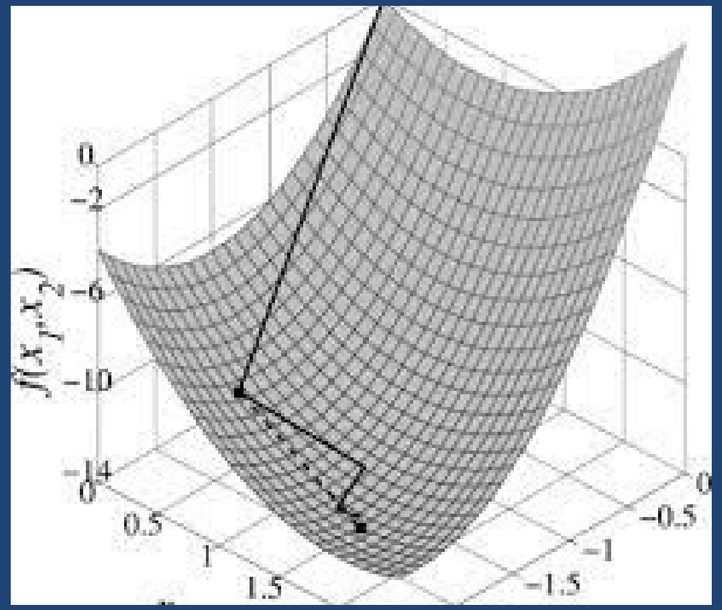

SGD
Momentum
NAG
Adagrad
Adadelta
Rmsprop
Recorded with iTop Screen Recorder


SGD
Momentum
NAG
Adagrad
Adadelta
Rmsprop
Recorded with iTop Screen Recorder
−1.0
−0.5
0.0
0.5
1.0
−1.0
−0.5
0.0
0.5
1.0
−2
−4

Conjugate Gradient Method

- Conjugate Gradient Method is one of the most beautiful applications of mathematics in ML !
- In this method, we basically try to reach the global minima in the MOST effective way by using Gradient Descent.
- In the adjacent figures, by using GD, we take curves in random directions, find their minima, and again take the next random curve, starting with the previous minima obtained.
- This randomness in choosing the consecutive curves might cost us a lot of time and energy.
- Thus, CGM finds the ideal curves to be considered in every consecutive steps (epochs), such that we reach the minima as quickly and efficiently as possible.
- The name, "Conjugate Gradient" backs the fact that we make use of Conjugate Matrices and their properties to set up the model.
- We will not go into the mathematics as of now, but you can surely take a look at it on your own !



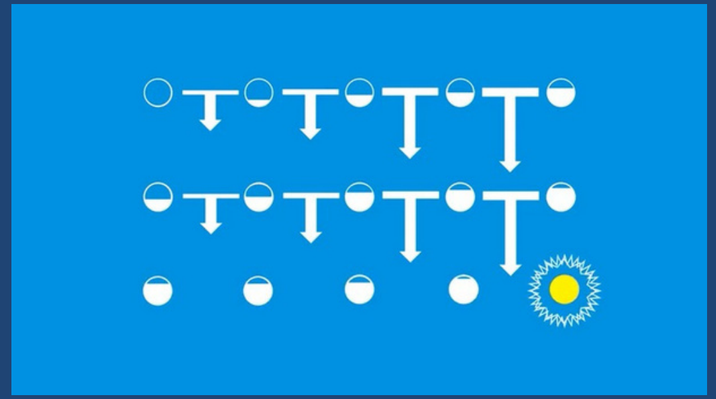


Gradient Free Methods

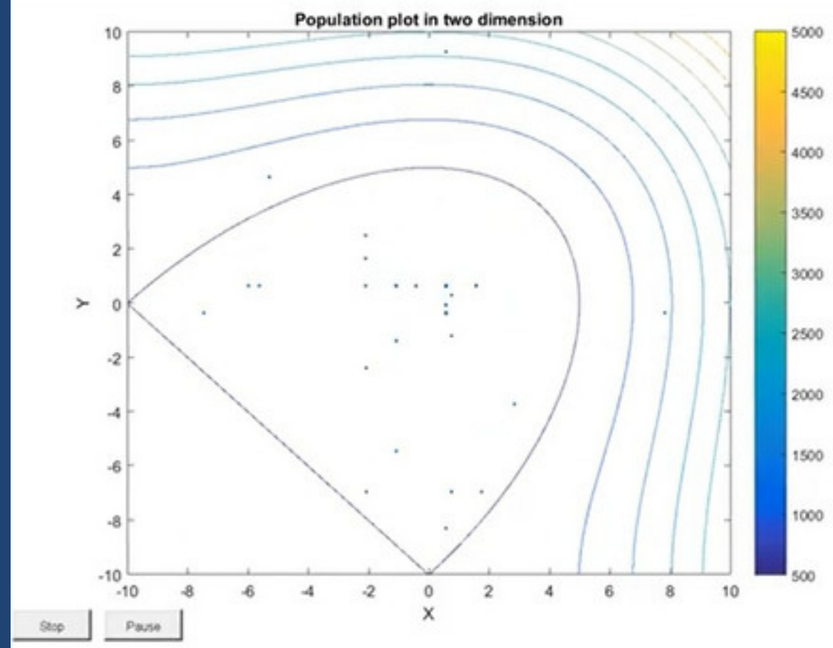
- As the name suggests, Gradient Free Methods optimise the efficiency of our model without the use of gradients.
- These methods can be used on problems, where the derivatives are not defined or difficult to obtain. (discrete, discontinuous or noisy functions)
- Exhaustive Search, Genetic Algorithms, Particle Swarm, Simulated Annealing and Nelder-mead are some of the methods used.
- These methods fail on large-scaled data sets as compared to Gradient used methods.
- Let's have a look at 1-2 of these methods to get and idea of how these types of methods actually work over !
- The exact functioning of these methods is much complex than what we have studied till now, and therefore, we will not go much deep into these methods as of now. (You are all free to explore these methods on your own just as a matter of curiosity!)

Genetic Algorithm

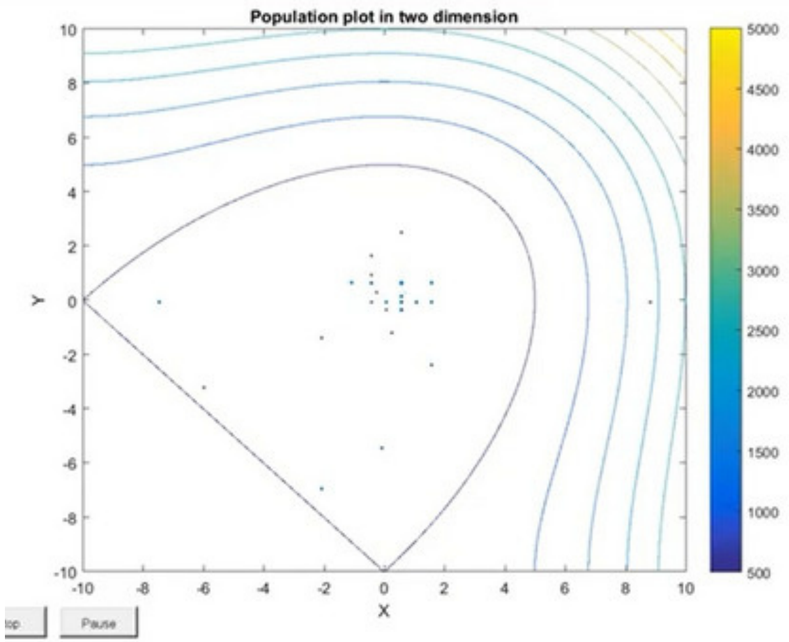
- The genetic algorithm starts by generating an initial population. This initial population consists of all the probable solutions to the given problem. The most popular technique for initialization is the use of random binary strings.
- The fitness function helps in establishing the fitness of all individuals in the population. It assigns a fitness score to every individual, which further determines the probability of being chosen for reproduction. The higher the fitness score, the higher the chances of being chosen for reproduction.
- After this, individual parents are paired and reproduce to form a generation with better scores.
- The process of how this is done is a bit complex, and therefore, we won't delve into it as of now, but we will surely look with the help of an example of how this method works.



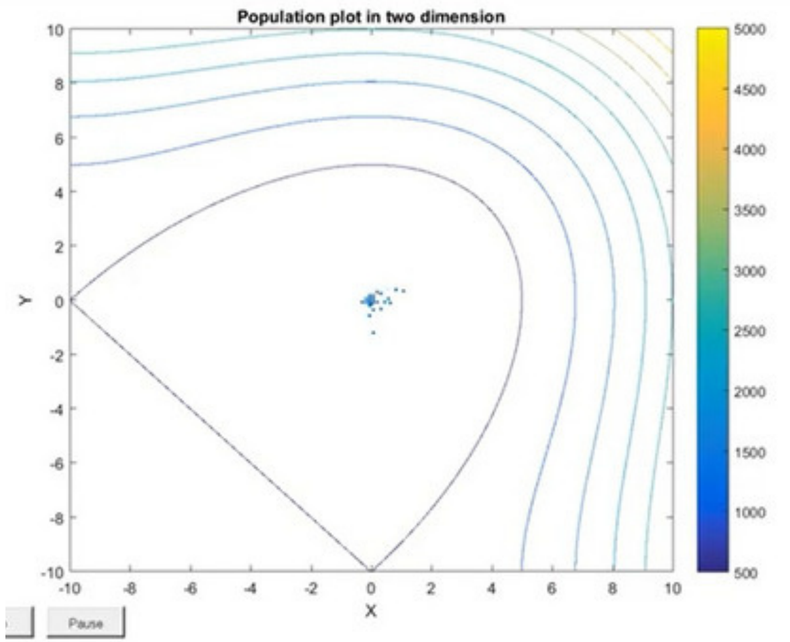
$$f(x,y) = x^3 + 15x^2 + y^3 + 15y^2$$



$$f(x,y) = x^3 + 15x^2 + y^3 + 15y^2$$



$$f(x,y) = x^3 + 15x^2 + y^3 + 15y^2$$



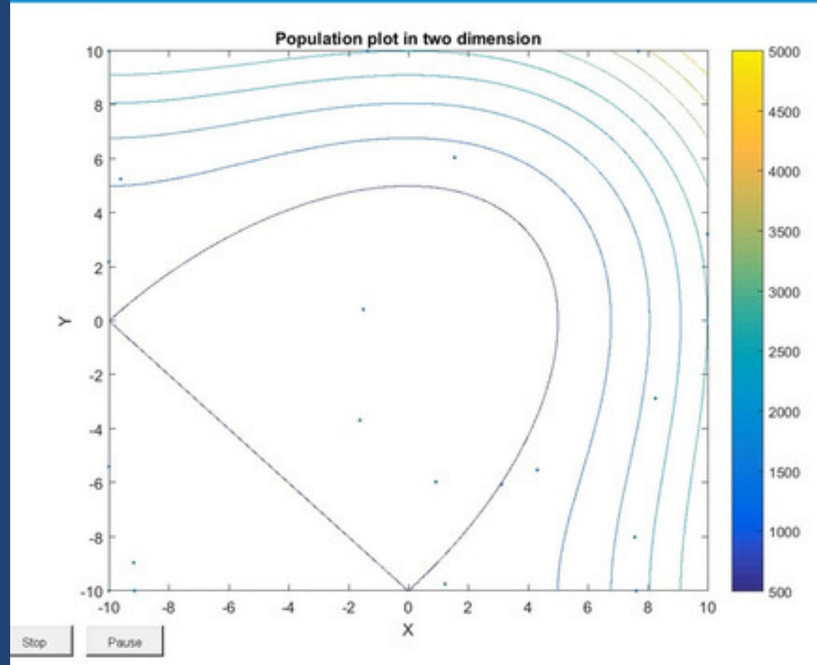
Particle Swarm

- Similar to GA, we collect a set of possible solutions called the Swarm, and assign a certain direction and velocity to every particle that depends on the the current direction (initially random), the direction of the best solution the point has found in the past, and the direction that the whole swarm has found in the past.

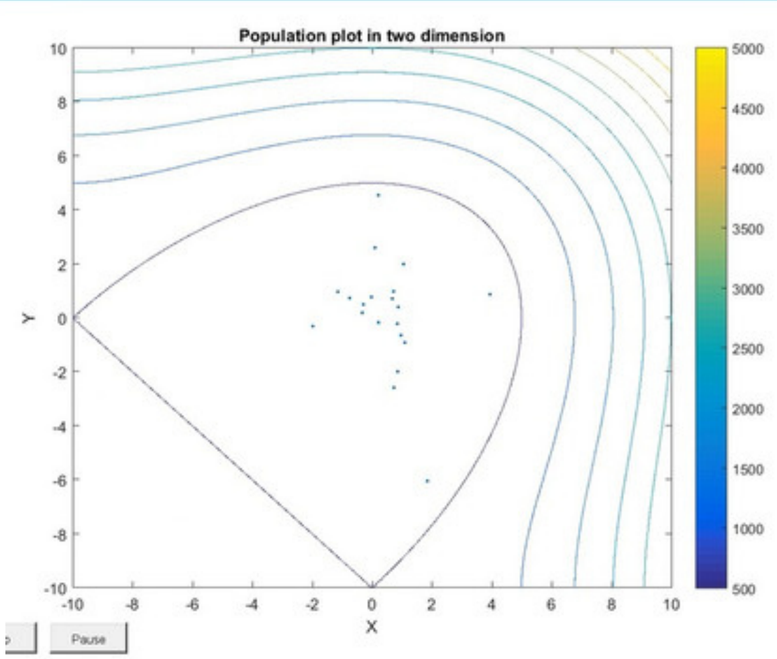
- Intuitively, what this means that the direction of the best solution determined by the swarm, directs the whole swarm towards the best solution, whereas the best solution found by the individual points directs individual points towards the optimal solution.

- Let's have a look about how this method works with the help of some images.

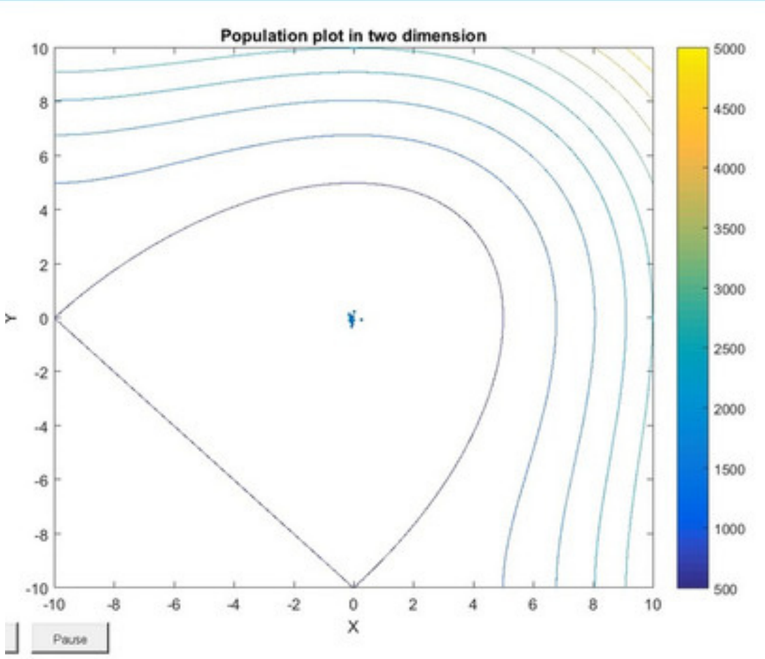
$$f(x,y) = x^3 + 15x^2 + y^3 + 15y^2$$



$$f(x,y) = x^3 + 15x^2 + y^3 + 15y^2$$



$$f(x,y) = x^3 + 15x^2 + y^3 + 15y^2$$



Zeroth Order Optimization

- Zeroth Order Optimization is a very recent and a very powerful method of Optimization.
- It is being used by the leading data scientists in the recent times.
- The functioning of this method is extremely complex and much beyond our understanding as of now.
- The motive of introducing this concept was only to make you all aware of the recent advancements in the field of ML, and all of you are encouraged to look upon this method as a matter of curiosity !