Introduction to Deep Learning

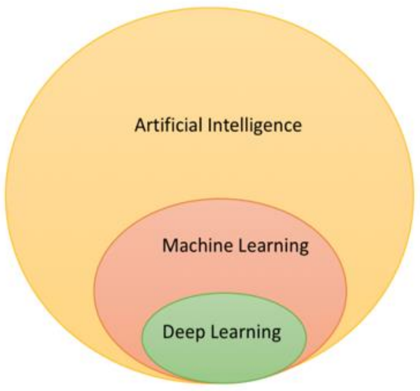
1. What Is Deep Learning

1.1. Deep learning definition:

Deep learning is a method in artificial intelligence (AI) that teaches the computers to process the data by using neural networks like human brain.

1.2. Deep learning part in machine learning:

- Deep learning is a sub part of machine learning and machine learning is a support of artificial intelligence.
- Deep learning is an advanced technique in machine learning.
- Deep learning can solve complex problems which cannot able to resolve in machine learning old algorithms.

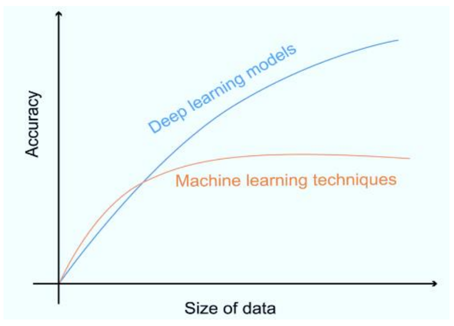


2. Why Deep Learning so Popular

There are mainly three reasons to explain popularity of the deep learning technique:

1. High accuracy at big size of data.
2. Implicit feature extraction models.
3. Solve complex problems.

2.1. High accuracy at big size of data:



Nowadays the Data increasing drastically in every business. Apply machine learning is the major problem on this huge amount of data. The old machine learning algorithms cannot give perfect accuracy on huge amount of data.

- If you observe the image the accuracy went to a saturated point in machine learning techniques when size of data getting increased.
- But in deep learning algorithms accuracy exponentially increased with size of data. So, the deep learning algorithms can give more accuracy when size of data increasing.

2.2. Implicit feature extraction models:

- In old machine learning algorithms needs feature extraction steps separately.

- But in deep learning, feature extraction can handle by models, which means this feature extraction step includes in model training and each model has its own way to handle the feature extraction.

2.3. Solve complex problems:

- Deep learning algorithms can solve complex problems when compared with old machine learning algorithms.
- For example, deep learning can be used image classification, NLP, chatbot kind of projects, but normal machine learning algorithms can’t work on these complex areas.

Artificial Neural Network

1. Introduction to ANN

1.1. Definition

An Artificial Neural Network (ANN) is a computational model that mimics the way of nerve cells work in the human brain.

1.2. History

In 1949, Donald Hebb published "The Organization of Behavior," which illustrated a law for synaptic neuron learning. This law, later known as Hebbian Learning in honor of Donald Hebb, is one of the most straight-forward and simple learning rules for artificial neural networks.

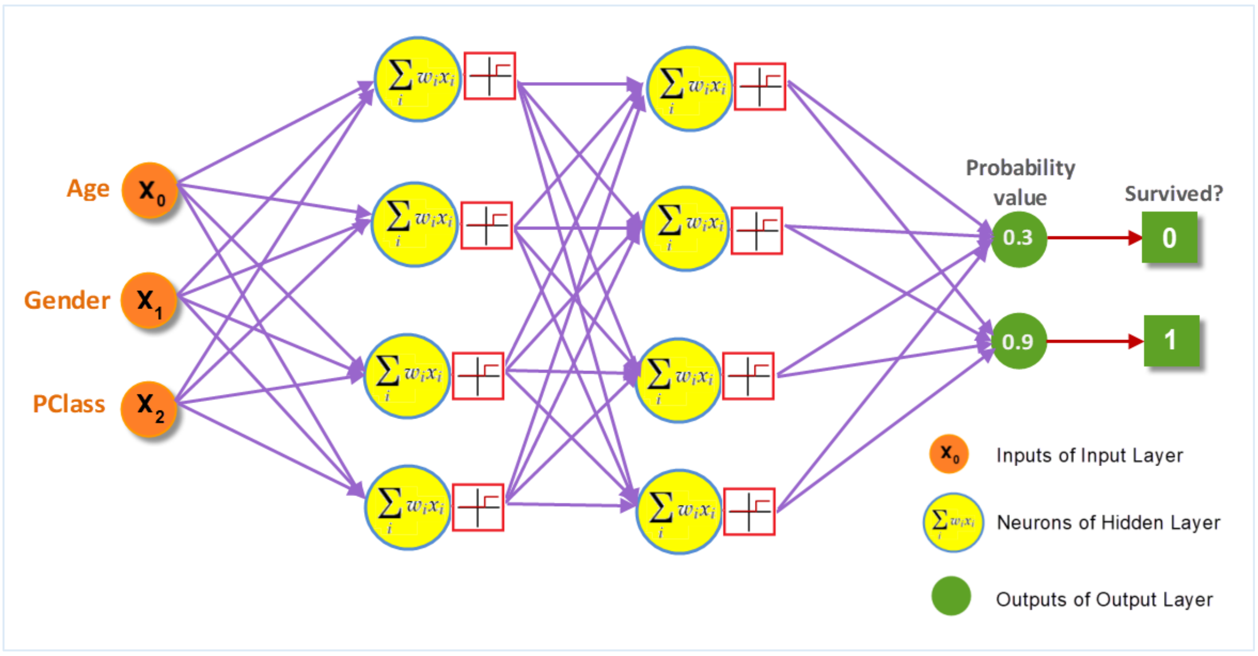
In 1951, Narvin Minsky made the first Artificial Neural Network (ANN) while working at Princeton.

In 1958, "The Computer and the Brain" were published, a year after Jhon von Neumann's death. In that book, von Neumann proposed numerous extreme changes to how analysts had been modeling the brain.

2. ANN architecture

- Artificial Neural Networks (ANNs) are a fundamental building block of deep learning.
- ANNs are inspired by the structure and function of the human brain, with interconnected nodes that mimic neurons and layers that perform specific tasks.

Basic Architecture of an Artificial Neural Network



2.1. Key Components:

1. Neurons (Nodes):

- Neuron is a fundamental processing unit, inspired by biological neurons.
- Neuron taking input signals, perform calculations, and produce output signals.
- Neurons are arranged in layers.

2. Layers:

- Groups of interconnected neurons are called layer.
- Common types of layers in ANN:
  - ✓ Input layer: Receives raw input data.
  - ✓ Hidden layers: Intermediate layers for processing the data and perform feature extraction.
  - ✓ Output layer: Produces final results (predictions, classifications).

3. Connections (Weights):

- Numerical values (weights) representing the strength of connections between neurons.
- These weights adjusted during training to learn patterns in data.

4. Activation Function:

- Determines whether a neuron "fires" or not, based on its input.
- Introduces non-linearity, enabling complex patterns to be captured.
- Common examples: ReLU, sigmoid, tanh.

5. Training:

- Process of adjusting weights to minimize errors and achieve desired outcomes.
- Involves feeding training data through the network and updating weights using backpropagation.

2.2. Key Characteristics:

- Deep: Multiple hidden layers enabled to learning the complex representations.
- Non-linear: Activation functions allow for modeling the non-linear relationships.

- Adaptive: Weights are adjusted during training to fit the data.

2.3. Additional Considerations:

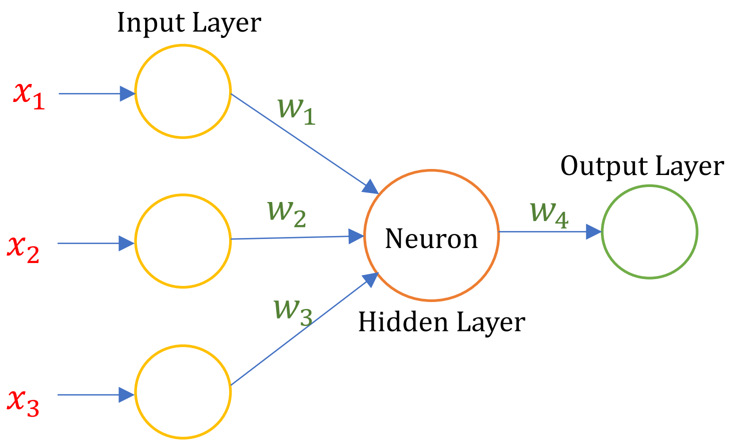
- Deeper Networks: Can learn more complex patterns, but require more data and computational resources.
- Regularization: Techniques to prevent overfitting and improve generalization.
- Optimization Algorithms: Used to efficiently update weights during training.
- Hyperparameter Tuning: Finding optimal settings for network architecture and training process.

Neural Network Working

1. NN working.
2. Internal work of neuron

1. Neural Network Working

Consider simple neural network example as show in below:



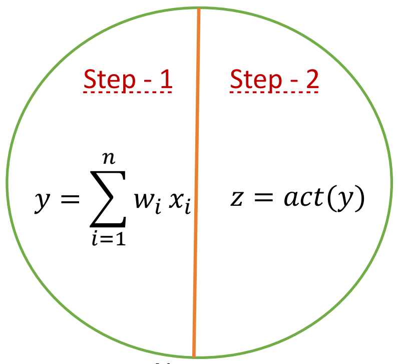
Here x_1, x_2, x_3 are input features. w_1, w_2, w_3 are connection weights features.

The X1 X2 and X3 are the input features to neural network, these features also known as independent variables.

- The input features X1, X2 and X3 are providing to input notes in input layer. These features will go to neuron through weighted connections W1, W2 under W3.
- Once the feature inputs and the weights received to the neuron, it will start the mathematical operations and will generate the activation signal.
- The generated output signal passed to the output node through weighted output connection W4.
- For example, if we kept a hot object on the hand then all hand related neurons immediately activated and weighted, that information passed to the brain through neurons.
- Consider here the hot object is input X, and neuron activation weight is (W). when weight is occurred then neuron will be activated.
- Here the big question is, how to know weight it is occurred or not it? can be known by activation function result.

2. Internal Work of Neuron:

The Neuron performs mainly two works are shown as step-1 and step-2 in the diagram. Let’s learn about those two steps now:



Neuron

Step -1:

In this stage the neuron generates the y values by adding the multiply values of weights (w) and feature (x) values at each ith index (ith row in dataset).

It receives the weights (w1, w2, w3, ...) from connections and feature values (x1, x2, x3,...) from input node or from previous neurons.

$$y = \sum_{i=1}^{n} w_i \, x_i + bias$$

$$y = w_1x_1 + w_2x_2 + w_3x_3 + \cdots \ldots \ldots + w_nx_n + bias$$

Step -2:

In the second step it will generate activation value of the Y by applying the activation function. This activation value (z) tells that neuron is activated or not. If the Z value is zero or similar to zero, then the neuron is in off state otherwise it is on state.

$$z = act(y)$$

Activation Functions

1. Introduction about activation function.
2. Sigmoid activation function
3. Relu activation function

The main work of activation functions is activation or deactivate the neuron based on the value Y. The input of activation function is hypothesis value which is generated from below equation.

$$y = \sum_{i=1}^{n} w_i x_i + b_i$$

Types of Activation Functions:

1. Sigmoid activation function
2. Relu activation function

1. Sigmoid Activation Function

The formula of sigmoid activation function is:

$$\sigma = \frac{1}{1 + e^{-y}}$$

$$\sigma = \begin{cases} > 0.5 & if\ y > 0 \\ \leq 0.5 & if\ y \leq 0 \end{cases}$$

$$Sigmoid\ act\ output = \begin{cases} 1 & if\ \sigma > 0.5 \\ 0 & if\ \sigma \leq 0.5 \end{cases}$$

- The sigmoid function gives the values between 0 to 1 only. We can observe these levels in figure 1.
- The output of neuron considered as 1 if the $\sigma > 0.5$. It indicates that particular neuron is activated.
- The output of neuron considered as 0 if the $\sigma < 0.5$. It indicates that particular neuron is not activated.

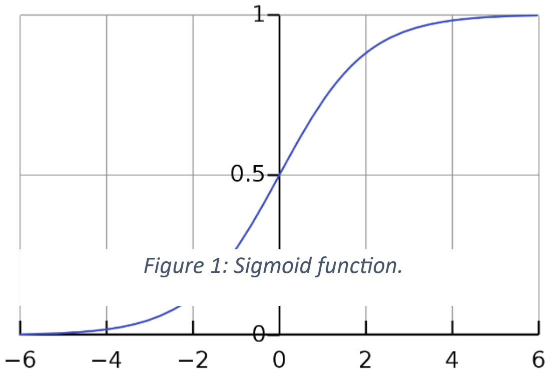


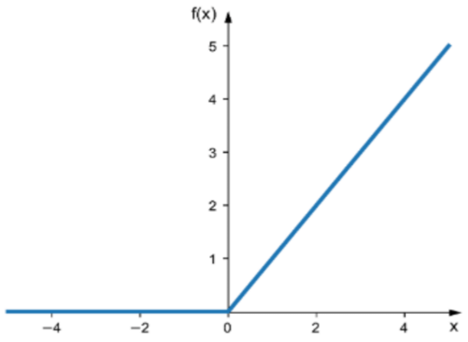
Figure 1: Sigmoid function.

2. Relu Activation Function

The formula of sigmoid activation function is:

$$Relu\ = max(y, 0)$$

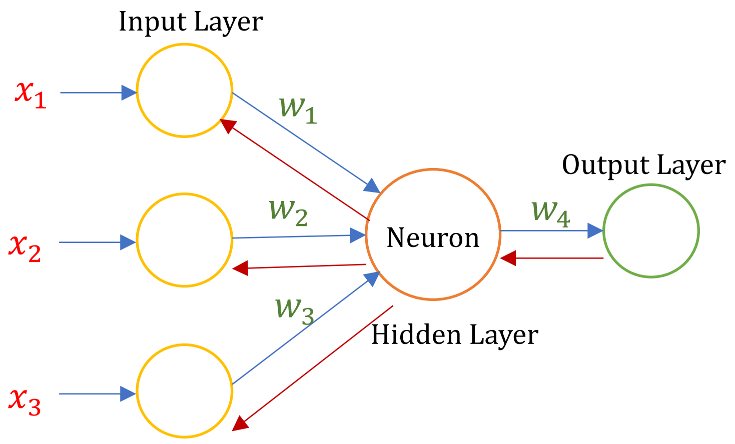
$$Relu = \begin{cases} y & if\ y > 0 \\ 0 & if\ y \leq 0 \end{cases}$$



- This is most popular technique than sigmoid activation function.
- The Relu Activation function output will be “y” if the value of y is +ve ($y > 0$). This is the neuron activate case.
- The Relu Activation function output will be “0” if the value of y is -ve ($y \leq 0$). This is the neuron deactivate case.

Forward and Backward Propagation in NN Training

1. Forward propagation
2. Backward propagation



Let's consider an example data as shown in below to understand these concepts better. It is a time management data of different students.

study	play	sleep	Good time maintenance
2hr	4hr	9hr	1
1hr	2hr	8hr	0
5hr	1hr	9hr	1
2hr	2hr	5hr	0

Here x_1 = Study

x_2 = Play

x_3 = Sleep

Output = Good time maintenance

In the about diagram blue arrows indicates the forward propagation and red arrows indicates the backward propagation

1. Forward Propagation:

Now we need to pass this data to above neural network for training purpose. This process is called forward propagation because the data and the training flow is in forward direction in neural network.

Let's understand forward propagation very clearly with below steps:

Step - 1:

Providing input data (x_1, x_2, x_3) to neural network.

Step - 2:

Neuron will calculate Y value with below mathematical equation.

$$z = w_1x_1 + w_2x_2 + w_3x_3 + bias$$

Step - 3:

Neuron will calculate the activation function value by using above y value, and pass the generated z value to output layer.

$$z = act(y)$$

2. Backward Propagation:

- The backward propagation starts immediately after received the output in forward propagation. The following steps involved in backward propagation.
- By using this backward propagation, we will decrease the loss and optimize the training.

Step - 1:

First it will calculate the loss value by using predicted and actual Y values with below equation.

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

Now the neural network task is reducing this loss by update the weights (w_1, w_2, w_3, w_4), because this loss is nothing but the total generated error in the neural network.

Step - 2:

In order to reduce the loss, it will update the weights (w_1, w_2, w_3, w_4) by using below equation. This equation provides the new weight by using old weight (w_{old}), learning rate (α) and derivative of loss function.

$$\omega_{new} = \omega_{old} - \alpha \frac{\partial L}{\partial \omega_{old}}$$

There is a huge subject behind the above equation we can learn all those things in gradient decent topic.

Step - 3:

After update the all the weights in the neural network with this backward propagation it will again start the forward propagation operation. This cycle continues until to get the best accuracy with less loss value.

Multilayer NN Working

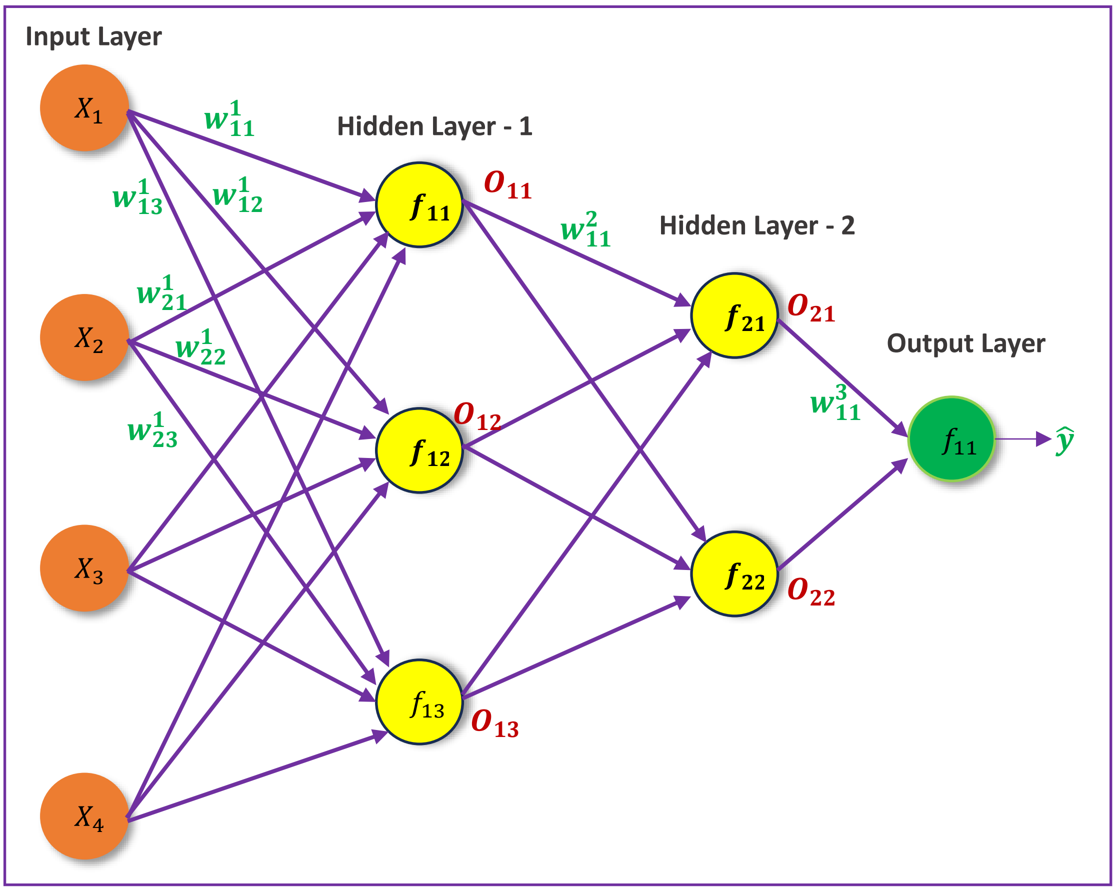


Figure 2: Multilayer neural network

The denotation of weight: $W^{layer\ number}_{(input\ node\ number)(output\ node\ number)}$

The weights matrix at first layer connections: $\begin{bmatrix} w^1_{11} & w^1_{12} & w^1_{13} \\ w^1_{21} & w^1_{22} & w^1_{23} \\ w^1_{31} & w^1_{32} & w^1_{33} \\ w^1_{41} & w^1_{42} & w^1_{43} \end{bmatrix}$

The weights matrix at second layer connections: $\begin{bmatrix} w_{11}^2 & w_{12}^2 \\ w_{21}^2 & w_{22}^2 \\ w_{31}^2 & w_{32}^2 \end{bmatrix}$

The weights matrix at third layer connections: $\begin{bmatrix} w_{11}^3 \\ w_{21}^3 \end{bmatrix}$

Let's understand this multilayer neural network working in three stages show in below. Observe the above multilayer neural network diagram and link with below explanation for better understanding:

1. Forward propagation
2. Lose calculation
3. Backward propagation

1. Forward Propagation:

- First it will take the input features (x_1, x_2, x_3, x_4) and pass them to each neuron $(f_{11},\ f_{12},\ f_{13})$ in the first hidden layer and weights also passed through input connection.
- Once the neuron receives the weights and inputs then it will start the *Hypothesis function*, and *Activation function* calculation internally. To understand it better let's consider the first neuron (f_{11}) working in the first hidden layer work.
  Step-1: Calculating the hypothesis value:
  $$\boldsymbol{f_{11} = w_{11}^1\, x_1 + w_{21}^1\, x_2 + w_{31}^1\, x_3 + w_{41}^1\, x_4 + b}$$
  Step-2: Calculating the output value by applying the activation function:
  $$\boldsymbol{O_{11} = act(f_{11})}$$
- Next the generated output (O_{11}) Passing as an input of next layer neurons.
- Again, the same neuron operations will happen inside the second layer neurons, as explained in the 3rd point above.
- Finally, the predicted y value will be generated at output layer.

2. Loss Calculation:

- The cost/ loss function value will calculate after complete the one round of forward training.
- The cost function is: $L = (y_i - \hat{y}_i)^2$

3. Backward Propagation:

- Updating the weights is the main work of backward propagation.
- We need to select a suitable optimizer (like Gradient descent, stochastic gradient descent etc..) for update the weights. Let's consider *Gradient Descent* for simple understand and consider w_{11}^3 weight updating process. The resultant equation is:
  $$w_{11\,new}^3 = w_{11}^3 - \alpha \frac{\partial L}{\partial w_{11}^3}$$
- Once all the weights got updated then it will again start the forward propagation. This cycle repeats until to get the global minimum point of the loss function.

There are four steps involved for update weights:

Step-1: Find out the predicted value $\hat{y}$.

Step-2: Pass the predicted $\hat{y}$ value to find the loss function L.

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

Step-3: Calculate the loss function.

Step-4: Substitute the loss function in the weight formula and update the weight by back propagation process.

- These four steps are continued to update all the weights and this back propagation process continues till old $\hat{y}$ predicted is similar to new $\hat{y}$ predicted.

Chain Rule in Back Propagation

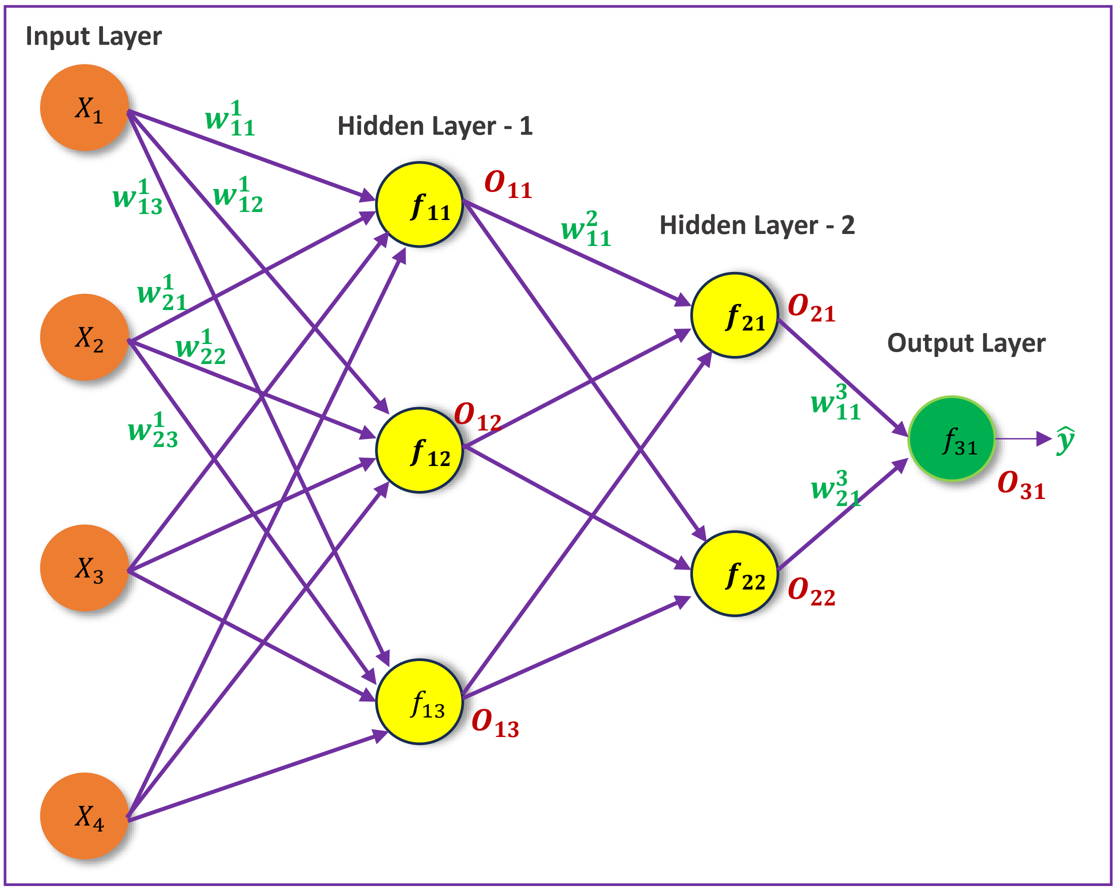


Figure 3: Multilayer neural network

Let's take gradient descent wide updating formula for reference:

$$\omega_{new} = \omega_{old} - \alpha \frac{\partial L}{\partial \omega_{old}} \dots\dots\dots\dots \quad (1)$$

The chain rule formula for update w^3_{11} at first step from output side is:

$$w^3_{11new} = w^3_{11} - \alpha \frac{\partial L}{\partial w^3_{11}} \dots\dots\dots\dots \quad (2)$$

$$\frac{\partial L}{\partial w^3_{11}} = \frac{\partial L}{\partial O_{31}} \cdot \frac{\partial O_{31}}{\partial w^3_{11}}$$

We should calculate $\frac{\partial L}{\partial w^3_{11}}$ by using above formula and substitute in w^3_{11new} (equation- 2)

Similarly, the formula for update the w^3_{21} is:

$$w^3_{21new} = w^3_{21} - \alpha \frac{\partial L}{\partial w^3_{21}} \dots\dots\dots\dots \quad (3)$$

$$\frac{\partial L}{\partial w_{21}^{3}} = \frac{\partial L}{\partial O_{31}} \cdot \frac{\partial O_{31}}{\partial w_{21}^{3}}$$

The formula for update the w_{12}^{2} is:

$$w_{12new}^{2} = w_{12}^{2} - \alpha \frac{\partial L}{\partial w_{12}^{2}} \ldots\ldots\ldots\ldots \quad (2)$$

$$\frac{\partial L}{\partial w_{12}^{2}} = \left[\frac{\partial L}{\partial O_{31}} \cdot \frac{\partial O_{31}}{\partial O_{21}} \cdot \frac{\partial O_{21}}{\partial w_{11}^{2}}\right] + \left[\frac{\partial L}{\partial O_{31}} \cdot \frac{\partial O_{31}}{\partial O_{22}} \cdot \frac{\partial O_{22}}{\partial w_{12}^{2}}\right]$$

It will absorb the above equation The second term is extra added after + Because when we want to change the weight w_{12}^{2} the output of O_{31} affected in two paths are $w_{11}^{2} \rightarrow O_{21} \rightarrow O_{31}$ and $w_{12}^{2} \rightarrow O_{22} \rightarrow O_{31}$. That's why second chain added in above equation.

Note: This is just a basic chain calculation, if you assume 100 connections of neurons are existed, then the formula will be bigger and many differentiations will be performed in that formula.

Gradient Problems

1. Vanishing gradient problem.
2. Exploding gradient problem.

1. Vanishing Gradient Problem:

Vanishing gradient problem is a phenomenon that occurs during the training of deep neural networks, where the gradients that are used to update the network become extremely small or "vanish" as they are backpropagated from the output layers to the earlier layers.

- We should use chain rule for finding $\frac{\partial L}{\partial \omega_{old}}$ if we use gradient descent to reduced loss function.
- If we observe the child role poll differentiation terms are multiplying each other. In mathematics the differentiation term is always a very less value those values may exist like 0 .1 or 0 .001 or 0 .00025 etc.
- When we multiply these small values in chain rule, then it will give very small value like 10^{-10} or 10^{-15} etc. The problem is when we try to find the local minima of loss function somewhere the point will be vanished because it treated $\frac{\partial L}{\partial \omega_{old}}$ as 0 $(\frac{\partial L}{\partial \omega_{old}} \approx 0)$.
- Actually, the differentiation loss function $(\frac{\partial L}{\partial \omega_{old}})$ is not zero. This problem is called vanishing gradient problem.

1.1. Vanishing Gradient Mathematical Proof

- Let's understand this vanishing gradient problem mathematically.

$$\omega_{new} = \omega_{old} - \alpha \frac{\partial L}{\partial \omega_{old}} \ldots\ldots\ldots\ldots \quad (1)$$

As per our previous explanation the derivative term is very less value and again, we multiply with Learning rate α.

Consider $\alpha = 0.001$ and $\frac{\partial L}{\partial \omega_{old}} = 2 \times 10^{-10}$

$$\alpha \frac{\partial L}{\partial \omega_{old}} = 0.001 \times 0.0000000002$$

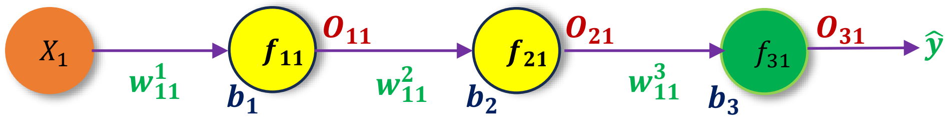
$$\alpha \frac{\partial L}{\partial \omega_{old}} \approx 0$$

- The $\alpha \frac{\partial L}{\partial \omega_{old}}$ is approximately zero and if we subtract this value from ω_{old} then there is no change in weights (ω_{new}), if weighs not much changed then predicted $\hat{y}$ value is almost same, and loss function also will not change, finally the back propagation takes it as minimum point of loss function but actually it is not.

2. Exploding Gradient Problem:

If the gradients are large, the multiplication of these gradients will become huge over time.

- In this case the weights of neural network are very high. These higher weights are not good to reach global minima point of loss function in GD.
- Let's understand this problem mathematically for that consider single line from neural network as shown in below:



Let's take f_{21} neuron hypothesis function $Z = O_{11}\, w_{11}^2 + b_2$

The sigmoid of Z is $\oint (Z) = O_{21}$

Now let's try to update the weight w_{11}^1 :

$$w_{11new}^1 = w_{11}^1 - \alpha \frac{\partial L}{\partial w_{11}^1}$$

$$\frac{\partial L}{\partial w_{11}^1} = \left[\frac{\partial\, L}{\partial\, O_{31}} \cdot \frac{\partial\, O_{31}}{\partial\, O_{21}} \cdot \frac{\partial\, O_{21}}{\partial\, O_{11}} \cdot \frac{\partial\, O_{11}}{\partial\, w_{11}^1}\right] \ldots\ldots\ldots\ldots(1)$$

- We know that sigmoid activation function lies between zero to one.
$$0 \le \oint (Z) \le 1$$
- And derivative of same sigmoid function lies between 0 to 0.25.
$$0 \le \frac{\partial \oint (Z)}{\partial Z} \le 0.25$$
- Now let's apply simple giant rule to $\frac{\partial\, O_{21}}{\partial\, O_{11}}$:

$$\frac{\partial\, O_{21}}{\partial\, O_{11}} = \frac{\partial\, O_{21}}{\partial\, Z} \cdot \frac{\partial\, Z}{\partial\, O_{11}}$$

$$= \frac{\partial \oint (Z)}{\partial\, Z} \cdot \frac{\partial\, Z}{\partial\, O_{11}}$$

$$= \left(0 \le \frac{\partial \oint (Z)}{\partial Z} \le 0.25\right) \cdot \left[\frac{\partial\, (O_{11}\, w_{11}^2 + b_2)}{\partial\, O_{11}}\right]$$

$$= \left(0 \le \frac{\partial \oint (Z)}{\partial Z} \le 0.25\right) \cdot [w_{11}^2]$$

- Now take care scenario $\frac{\partial \oint (Z)}{\partial Z} = 0.25$ and $w_{11}^{2} = 500$

  $$\frac{\partial \ O_{21}}{\partial \ O_{11}} = 0.25 \ \times 500$$

  $$= 125$$

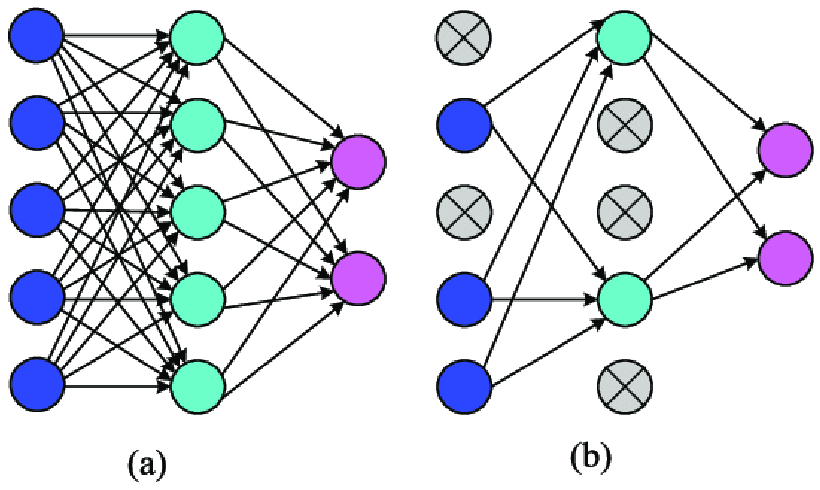
- Let's substitute all values derivative in equation one

  $$\frac{\partial L}{\partial w_{11}^{1}} = 300 \ \times 200 \times 125 \times 1001$$

- The value of $\frac{\partial L}{\partial w_{11}^{1}}$ is very large. Whenever $\frac{\partial L}{\partial w_{11}^{1}}$ is large value then w_{11new}^{1} will be *negative value*. If w_{11new}^{1} is negative, then weights are jumping positive to negative side of the loss function graph. it will never reach to global minima point.

Note: What we did till now is for simple neural network, if we consider a large neural network and weights updating something like this, that will never take the global minimum point.

Dropout Layers in ANN



- Removing or dropping off some layers randomly during training this process is called dropout layering.
- It is one of the regularization techniques in ANN.
- ANN has many weights and biases when it has a huge depth, in this case ANN might be overfit to the data.
- Overfit and underfit conditions in ANN are:
  - ✓ Less number of layers : UNDERFIT
  - ✓ More number of layers : OVERFIT
- Always over fit is the main problem in deep ANN and the underfit condition never happened in deep ANN.
- If you observe the above image, image(a) indicate the ANN before dropout and image (b) indicates after dropout.

Working:

Step -1:

In first step should select a dropout ratio for each layer

$$p = Dropout\ ration$$

$$0 \leq P \leq 1$$

Step -2:

ANN will select some neurons in each layer based on the dropout ratio and remaining neurons will be deactivated.

Step -3:

Now ANN will start the training with forward and backward propagation, and find the cost function then minimizing it.

Step -4:

ANN testing will start after complete the training. In the testing ANN connect with all layers, neurons and weights will be changed with a new term i.e:

$$w_{new} = w \ \times p$$

Here p = Dropout ratio selected for each layer.

w = The weight which was selected in training with optimizer.

Note: Learn about the hyperparameter tuning concept in machine learning to find the best P value.

Rectified Linear Unit

1. Rectified linear unit introduction
2. Relu functionality
3. Leaky relu.

- The main aim of rectifying linear unit is removing the vanishing GD problem.
- Normally the varnishing GD problem coming when a greater number of derivative terms get multiplied.
- If we take a sigmoid activation function, the values of derivative of sigmoid lies between 0 to 0.25. these are very small values. After multiply all these small values in chain rule will cause varnishing GD problem.
- Not only in sigmoid function, in threshold activation function also creates same problem.

Note: The vanishing gradient descent occurs due to two reasons:

1. Low values of derivative terms.
2. Chain rule in loss function.

Relu Functionality:

- This problem overcome by using *Relu* activation function. now let's understand how *Relu* works.

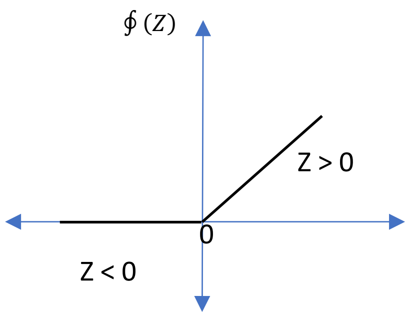


Figure 5 - Relu graph

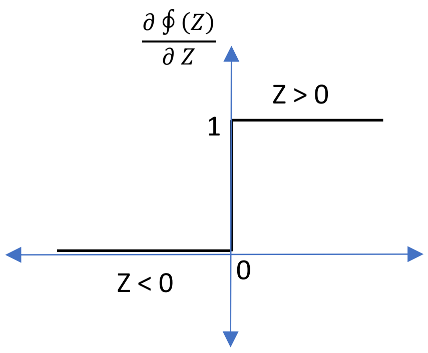


Figure 4 - Derivative Relu graph

- The formula of *Relu* activation function is:

$$\oint(Z) = max\,(Z, 0)$$

$$At\ normal\ condition: \begin{cases} When\ z\ is\ negitive\ \oint(Z) = 0 \\ When\ z\ is\ positive\ \oint(Z) = z \end{cases}$$

- The derivative of *Relu* activation function values are:

$$\frac{\partial \oint(Z)}{\partial Z} = \begin{cases} 1 & if\ \ z = +ve\ or\ z > 0 \\ 0 & if\ \ z = -ve\ or\ z < 0 \end{cases}$$

- Let's check above values in two cases with chain rule for updating weights.

Case – 1:

Consider the values of all derivative terms as 1 in chain rule of $\frac{\partial L}{\partial \omega_{old}}$.

$$\frac{\partial L}{\partial \omega_{old}} = 1 \times 1 \times 1$$

And consider $\alpha = 1$, then ω_{new} is very different from ω_{old}. ω_{new} will update normally and there is no problem in this case.

Case – 2:

In this case take one of the derivative terms as 0 in the child rule.

$$\frac{\partial L}{\partial \omega_{old}} = 1 \times 0 \times 1$$

Finally, $\omega_{new} = \omega_{old} - \alpha\ [0]$

$$\omega_{new} = \omega_{old}$$

But actually, it is not correct because Relu function value zero even the value of z is not -ve in other derivative terms.

If we observe in this case no weight changes happening. And this case exactly creates the dead neuron means this particular neuron will be deactivated due to a single zero of derivative term in child rule.

Note: This particular problem we can overcome by using leaky Relu concept.

Leaky Relu:

- The *leaky relu* takes a small functional value instead of derivative term value when $z < 0$.

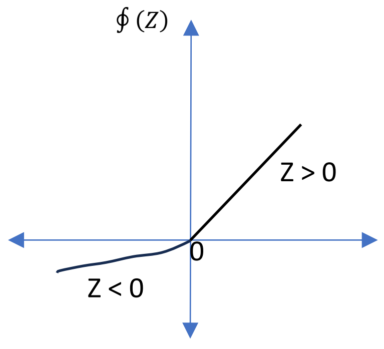


Figure 6 – Leaky Relu graph

- In this case the *Leaky Relu* graph will change as shown in the figure.

$$\frac{\partial \oint(Z)}{\partial Z} = \begin{cases} 1 & if\ \ z > 0 \\ 0.01(z) & if\ \ z < 0 \end{cases}$$

- Now the derivative terms will not be zero, but those are considered as nearby zero. And it will consider the dead neurons which are created by zero values of derivative terms.

Weight Initialization Techniques

1. Key points in weight initialization
2. Weight initialization techniques

Different types of weights initializing techniques are available in deep learning. Each technique follows a unique way to initialize the weights. The accuracy of the NN is also depends on the type of the weight initialization technique.

1. Key Points in Weights Initialization

The below key point should remember before Initialize the weights:

1. Weight should be small but not similar to zero ($w \not\approx 0$).
2. Weight should not be same.
3. Weights should have good variance.

1.1. Weight should be small:

- Faster Training: Smaller weights can lead to faster convergence during training.
- Regularization: Small weights act as implicit regularization, preventing overfitting. Regularization techniques like L1 or L2 are explicitly penalize the large weights, encouraging the network to learn simpler patterns that generalize better on unseen data.
- Numerical Stability: Smaller weights give numerical stability in math calculations and prevent overflow issues.
- Memory Efficiency: Small weights enhance memory efficiency, crucial for resource-constrained environments.

1.2. Weight should not be same:

If all weights are same then all neurons performance will be same, and the outputs of each neuron also same. In this case ANN can't separate the data in classification. That's why weights should not be same.

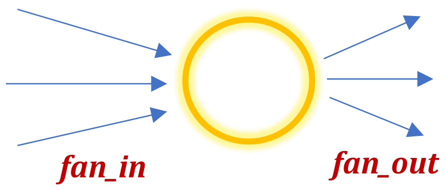
1.3. Weights should have good variance:

The learning rate is bad when the difference between weights is less. Show weights should have good variance.

Terms involved in weight initialization:

Mainly there are two terms involved in every weight initialization technique formula those are:

Inputs	→	fan_in
Outputs	→	fan_out



2. Weights Initialization Techniques:

Mainly there are four mostly used weight initialization techniques are available in deep learning:

1. Uniform distribution
2. Xavier / Gorat distribution
3. He init distribution

2.1. Uniform Distribution:

Here weight will take in between subrange depending on the formula

Normal Uniform
$\boldsymbol{w_{ij} \approx Uniform\left[\frac{-1}{\sqrt{fan\_in}}, \quad \frac{1}{\sqrt{fan\_in}}\right]}$

The distribution performed between these two points only.

2.2. Xavier / Gorat Distribution:

Xavier Normal	Xavier Uniform
$\boldsymbol{w_{ij} \approx N(0, \sigma)}$ $\boldsymbol{\sigma = \sqrt{\frac{2}{[fan\_in + fan\_out]}}}$	$w_{ij} \approx U\left[\frac{-\sqrt{6}}{\sqrt{fan\_in + fan\_out}}, \quad \frac{\sqrt{6}}{\sqrt{fan\_in + fan\_out}}\right]$

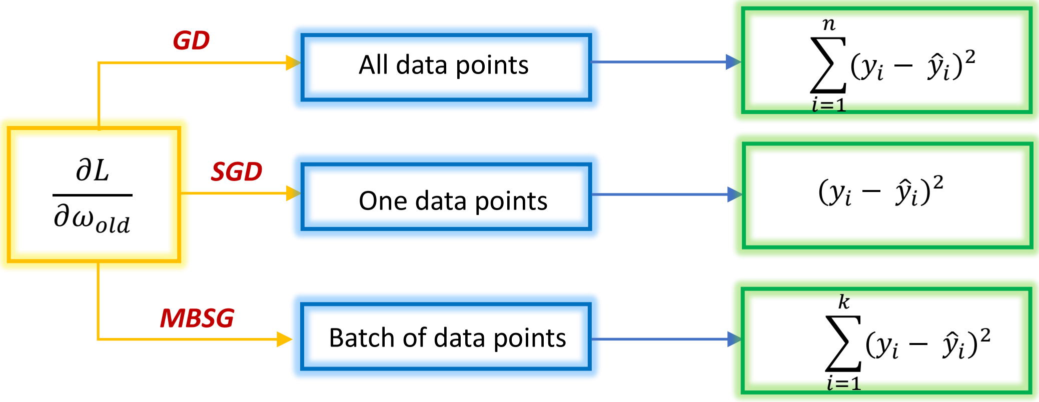
2.3. He Init Distribution:

He Normal	He Uniform
$\boldsymbol{w_{ij} \approx N(0, \sigma)}$ $\boldsymbol{\sigma = \sqrt{\frac{2}{fan\_in}}}$	$w_{ij} \approx U\left[-\sqrt{\frac{6}{fan\_in}}, \quad \sqrt{\frac{6}{fan\_in}}\right]$

Different Types of Gradient Decent Techniques Working Nature

1. How All GD Techniques Works:

- In *Gradient Descent* all data points will consider in $\frac{\partial L}{\partial \omega_{old}}$ calculation.
- In *Stochastic Gradient Descent* (SGD) only one data point will consider in $\frac{\partial L}{\partial \omega_{old}}$ calculation. In SGD, ANN takes only one data point for forward propagation and calculate $\frac{\partial L}{\partial \omega_{old}}$, Then update the weights in back propagation. This process repeats for all data points in data set.
- Mini batch stochastic gradient descent considers a batch of data points (k) from the dataset to calculate the $\frac{\partial L}{\partial \omega_{old}}$. It will do the forward propagation with a batch of data points k and calculate $\frac{\partial L}{\partial \omega_{old}}$, then update the weights in back propagation. This process rebates for all batches of the data set.

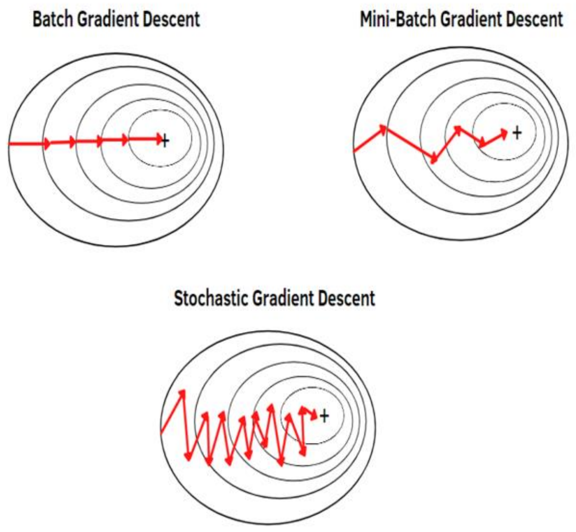

GD
SGD
MBSG
$\frac{\partial L}{\partial \omega_{old}}$
All data points
One data points
Batch of data points
$\sum_{i=1}^{n}(y_i - \hat{y}_i)^2$
$(y_i - \hat{y}_i)^2$
$\sum_{i=1}^{k}(y_i - \hat{y}_i)^2$

- The most popular technique is MBSGD. Because it is more accurate than GD and SGD.

1.1. Global Minima Path Loss Function:

- The GED path is straightforward to get the global minimum in loss function.
- But SGD and MBSGD follows a jig-jog path to get the global minimum. Due to this jig-jog path it has some noise. Momentum SGD overcome this noise problem.

2. Momentum SGD:



Local Minima Problem

1. Introduction about local minima problem
2. How local minima confuse the ANN

1. Introduction about Local Minima problem:

- The local minima point in the loss function is one of the problems while updating the weights. The local minima points confuse ANN and ANN will treat the local minima as a global minima.
- Till now we consider the loss function graph as a normal bell curve as shown in the fig-1. This graph has only one global minima point.
- But in real world scenario the curve looks like fig - 2 with local minima, local maxima, global minima and global maxima points.

2. How Local Minima Confuse the ANN:

Now let's understand how ANN treat the local minima as global minima due to tangent confusion:

Consider weight updating formula.

$$\omega_{new} = \omega_{old} - \alpha \frac{\partial L}{\partial \omega_{old}}$$

Here the loss function is.

$$L = \sum_{i=1}^{k} (y_i - \hat{y}_i)^2$$

- The meaning of loss derivatization $(\frac{\partial L}{\partial \omega_{old}})$ is, calculating the tangent of the loss function. This tangent gives the slope of loss function at a particular point.
- Which means if the graph doesn't have any slope at any point that place treated as a global minima. This is actual working of gradient descent optimizer.
- But if we observe the slope of local minima points also zero, Due to this reason ANN takes this local minima as a global minima.

At local minima point 1 the slope is zero means $\frac{\partial L}{\partial \omega_{old}} = 0.$

Then $$\omega_{new} = \omega_{old} - \alpha\,(0)$$
$$\omega_{new} = \omega_{old}$$

- ANN stops the training and considering global minima got reached whenever the weights ω_{new} and ω_{old} are same.

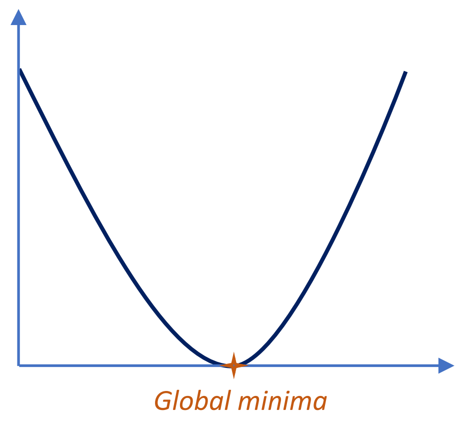


Figure 1 - Convex function

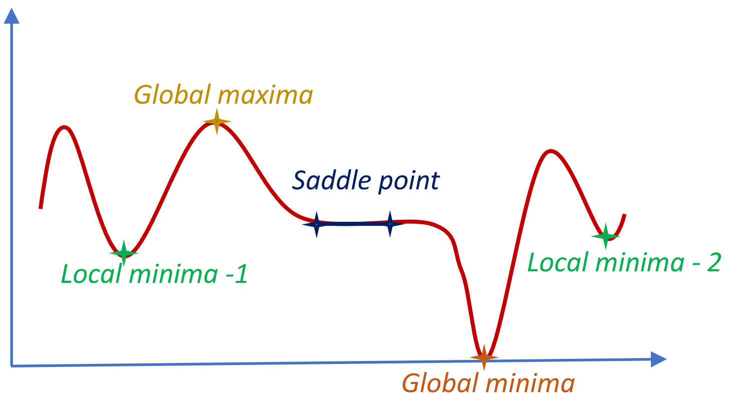


Figure 2 – Non-Convex function

- The saddle point is also a one of the tangent problems, we can see it in fig-2. The ANN treats this saddle point as global minima, because tangent of saddle point is 0.

Learning Rate Optimizers

1. How Adagrad Optimizer:

- Up to now we knew some optimizers like GD, SGD, MBSGD. These all optimizers are used for reduce the loss function.
- The weight updating formula which is used in GD, SGD, MBSGD is:

$$\omega_{new} = \omega_{old} - \alpha \frac{\partial L}{\partial \omega_{old}} \ldots\ldots\ldots\ldots \quad (1)$$

- In Adagrad Optimizer the formula has changed as:

$$\omega_t = \omega_{t-1} - \alpha_t^1 \frac{\partial L}{\partial \omega_{old}} \ldots\ldots\ldots\ldots \quad (2)$$

- The main change in this optimization is, we are taking different learning rates for each neuron in each layer at each iteration. That's why the α is replaced with α_t^1.
  Here α_t = Present weight;
  t = Iteration number;
  α_{t-1} = Previous weight
- Adagrad Optimizer taking different alpha values to avoid DENSE, SPARSE, BOW problems.

1.1. α_t^1 Value Calculation:

- Now the task is finding α_t^1 value. the correspond formula is:

$$\alpha_t^1 = \frac{\alpha}{\sqrt{\alpha_t + \varepsilon}}$$

- Here ε is a small +ve value Which is added to α_t to avoid the zero values problem. let's consider a formula without ε.

$$\alpha_t^1 = \frac{\alpha}{\sqrt{\alpha_t}}$$

- If we consider the above formula. If $\alpha_t = 0$ then $\alpha_t^1 = \infty$ finally the $w_t = \infty$. to avoid this problem, we add a small positive value ε.

1.2. α_t Values Calculation:

- α_t is the sum of derivative of loss function from starting iteration to current iteration.

$$\alpha_t = \sum_{i=1}^{t} \left[\frac{\partial L}{\partial w_i}\right]^2$$

- Let's consider we calculate α_t for 3rd iteration then the formula become:

$$\alpha_3 = \sum_{i=1}^{3} \left[\frac{\partial L}{\partial w_i}\right]^2$$

- α_t value is always higher value, because it is the sum-up component of derivative functions. which means we are adding all derivative terms to calculate α_t.
- Whenever α_t value is high, then α_t^1 value will be decreased because α_t^1 is inversely proportional to $\sqrt{\alpha_t}$.

$$\alpha_t^1 \propto \frac{1}{\sqrt{\alpha_t}}$$

- α_t^1 decreases for every iteration then the weight also will decrease slowly if weights are decreased then cost function will coverage to minimum point.

1.3. Important Graph Explanation:

- In Adagrad Optimization the weights are decreasing slowly and finally converge to minima as shown in the graph.
- Initially the weights are very high and continuously those will decrease step by step.

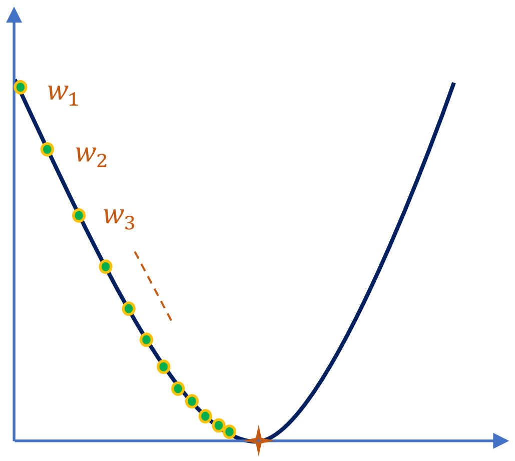


Figure 1 - Convex function

1.4. Disadvantage:

This Adagrad Optimizer is also having one disadvantage due to α_t. Actually α_t is some of squares of derivative terms.

- Consider if number of iterations are increasing α_t value will also increase. if α_t increasing drastically α_t^1 value will be very low. This situation can create approximate zero values of α_t^1.
- So, this optimization technique fails at a higher number of iterations of training.

2. Adadelta Optimizer:

- Adadelta is an advanced version of Adagrad Optimizer. It Addresses the issues which are in Adagrad.
- Adagrad has *Diminishing Learning Rates* problem that can resolve by Adadelta.
- The α_t is the main reason of *Diminishing Learning Rates* problem, the Adadelta Optimizer formula created by w_{avg} instead α_t.

$$\alpha_t^1 = \frac{\alpha}{\sqrt{w_{avg} + \varepsilon}}$$

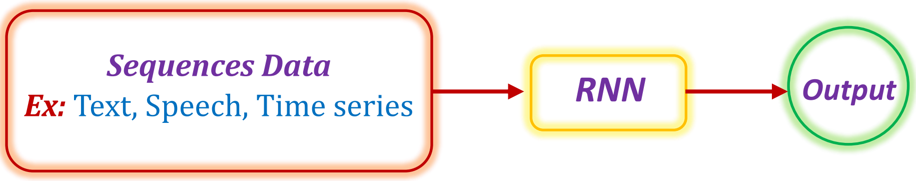
$$w_{avg(t)} = \gamma\, w_{avg(t-1)} + (1-\gamma) \left[\frac{\partial L}{\partial w_t}\right]^2$$

- The main difference between. w_{avg} and α_t is high value. α_t value is always high when we compare α_t with w_{avg}.
- The main reason is w_{avg} don't have any summation terms and it haven't all iterations loss function terms. It has only one loss function term. so, it doesn't have any problem about high number of iterations.
- Here γ is a small fractional multiplier and it is 0.95 in most cases.

Recurrent Neural Network

1. Why we Can't Use ANN for Text Data
2. RNN Architecture
3. Problems in Simple RNN

A Recurrent Neural Network (RNN) is a type of artificial neural network designed to recognize patterns in sequences of data, such as text, speech, time series, or any sequence of data points.



RNN ---> Recurrent Neural Network

Recurrent --- > Carrying repeatedly

1. Why we Can't Use ANN for Text Data:

There are mainly two problems with the text data if we use in ANN:

1. High volume of input values
2. Unrecognizable sequence problem

1.1. High Volume of Input Values:

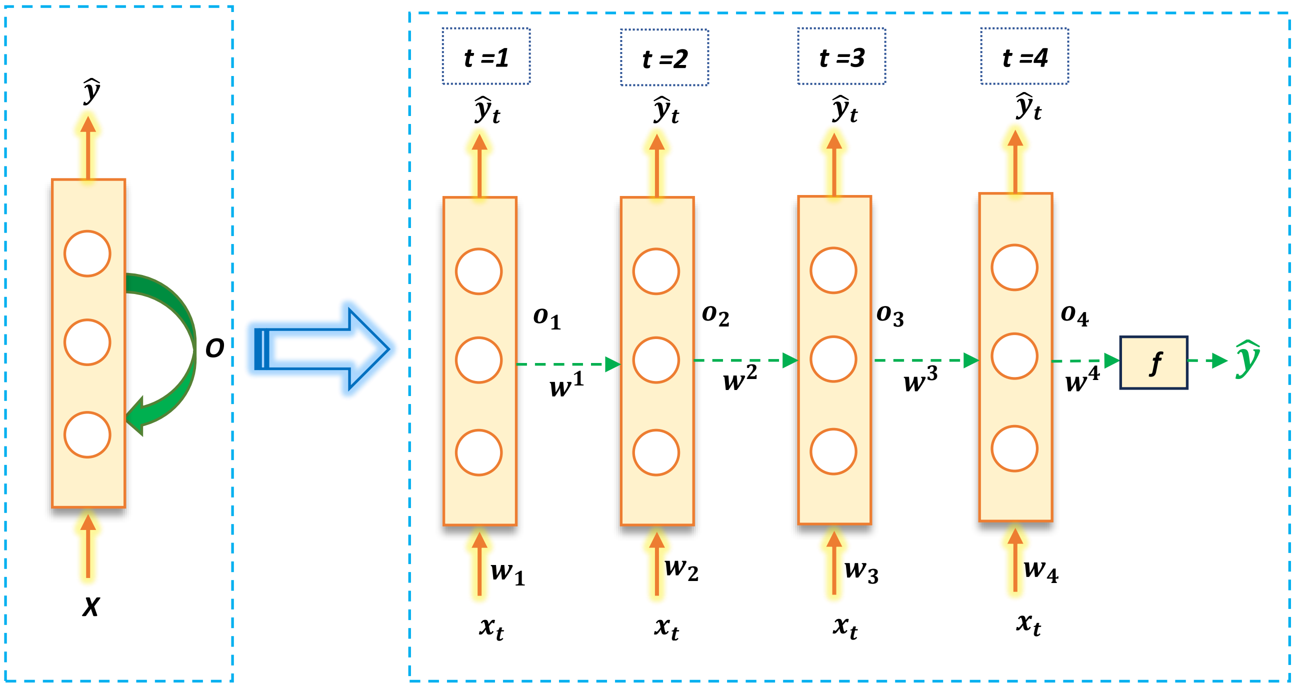
- If we observe artificial neural network hidden neurons and depth is depends on the number of input values.
- But while processing the text data has thousands or hundreds of words, in this case the ANN will be more competitional with huge depth.
- ANN increases the cost and taking too much time to complete the training while processing the text data.

1.2. Unrecognizable Sequence Problem:

- ANN not consider the sequence of the words but in text data processing the sequence of words is very important.
- If we pass two sentences as an input to ANN by changing the words in the two sentences internally, but ANN gives same output for these two sentences because of the words contains two sentences are same.
- Actually, these two sentences meaning is not same, the expected output should not be same.

Recurrent Neural Network can overcome the above problem and it can work with ***sequence data*** and identifying the output based on the sequence along with value**.**

2. RNN Architecture:



Let's consider 4 words (x_1, x_2, x_3, x_4) of sentence X.

$$X = < \ x_1, x_2, x_3, x_4 \ >$$

2.1. Forward Propagation:

- At time t = 1 the first word x_1 taking as input along with weight w_1. The first hidden layer processed that word x_1 and generated the output $\hat{y}_1$. Thes first word output O_1 will pass as an input to seconded layer by applying activation function.
- The second layer will take two inputs at time t = 2, one is x_2 along with w_2 and seconded is output of first layer O_1.
- This process continues for all words and final output $\hat{y}$ will come output layer.
- Let's see the output functions in each layer:
  $O_1 = f(x_1\, w_1)$
  $O_2 = f(x_2\, w_2 + O_1\, w^1)$
  $O_3 = f(x_3\, w_3 + O_2\, w^2)$
  $O_4 = f(x_4\, w_4 + O_3\, w^3)$
- Based on the above equations, the output O_4 is a depends on the input x_4 and output O_3. Again output O_3 is a depends on output O_2 and the output O_2 is depends on output O_1. It indicates every output considering the previous outputs as sequence. That's why RNN is worked based on the sequence.

2.2. Back Propagation:

- First it will calculate the loss function after got the predicted $\hat{y}$
- Next it will update the weight by using chain rule with back propagation as like as ANN. Let's observe the weight updating equations:

$$\omega_{new} = \omega_{old} - \alpha \frac{\partial L}{\partial \omega_{old}} \ldots\ldots\ldots\ldots \quad (1)$$

Updating w^4:

$$\omega^4 = \omega^4 - \alpha \frac{\partial L}{\partial \omega^4} \ldots\ldots\ldots\ldots \quad (2)$$

$$\frac{\partial L}{\partial \omega^4} = \frac{\partial L}{\partial \hat{y}} \cdot \frac{\partial \hat{y}}{\partial \omega^4}$$

Updating ω_4:

$$\omega_{4(new)} = \omega_4 - \alpha \frac{\partial L}{\partial \omega_4} \ldots\ldots\ldots\ldots \quad (3)$$

$$\frac{\partial L}{\partial \omega_4} = \frac{\partial L}{\partial \hat{y}} \cdot \frac{\partial \hat{y}}{\partial O_4} \cdot \frac{\partial O_4}{\partial \omega_4}$$

- The remaining weights also will update as shown above.

3. Problems in Simple RNN:

- If we use *Sigmoid Activation* function will create varnishing gradient problem because of the derivative terms are very less values (exist between 0 to 1) in chain rule. The multiplication of these less values will create very less change in weights. Finally, it creates vanishing gradient problem.
- If we use *Relu Activation* function will create exploding gradient problem because of the derivative terms are very high values (exist > 1) in chain rule. The multiplication of these high values will create exploding gradient problem.

*The **LSTM*** can overcome these ***Exploding, Varnishing*** gradient problems***.***