Rizvi College of Engineering

DEEP LEARNING

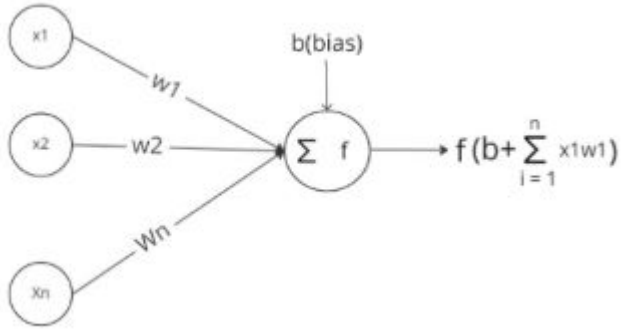
MODULE 2

2		**Training, Optimization and Regularization of Deep Neural Network**
	2.1	**Training Feedforward DNN** Multi Layered Feed Forward Neural Network, Learning Factors, Activation functions: Tanh, Logistic, Linear, Softmax, ReLU, Leaky ReLU, Loss functions: Squared Error loss, Cross Entropy, Choosing output function and loss function
	2.2	**Optimization** Learning with backpropagation, Learning Parameters: Gradient Descent (GD), Stochastic and Mini Batch GD, Momentum Based GD, Nesterov Accelerated GD, AdaGrad, Adam, RMSProp
	2.3	**Regularization** Overview of Overfitting, Types of biases, Bias Variance Tradeoff Regularization Methods: L1, L2 regularization, Parameter sharing, Dropout, Weight Decay, Batch normalization, Early stopping, Data Augmentation, Adding noise to input and output

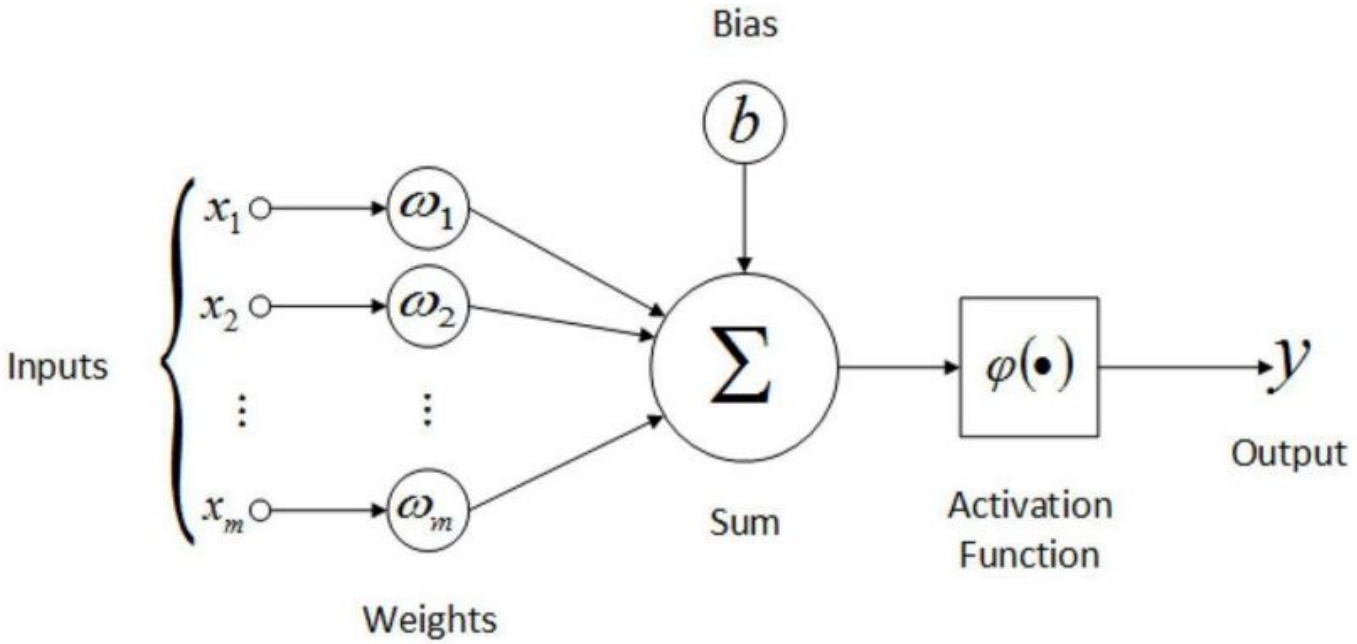
Basic Introduction to Feed-Forward Network in Deep Learning

A Feed Forward Neural Network is an artificial Neural Network in which the nodes are connected circularly. A feed-forward neural network, in which some routes are cycled, is the polar opposite of a Recurrent Neural Network. The feed-forward model is the basic type of neural network because the input is only processed in one direction. The data always flows in one direction and never backwards/opposite.

- The complexity of the function is inversely correlated with the number of layers.
- It cannot spread backward; it can only go forward. In this scenario, the weights are unchanged.
- Weights are added to the inputs before being passed to an activation function.

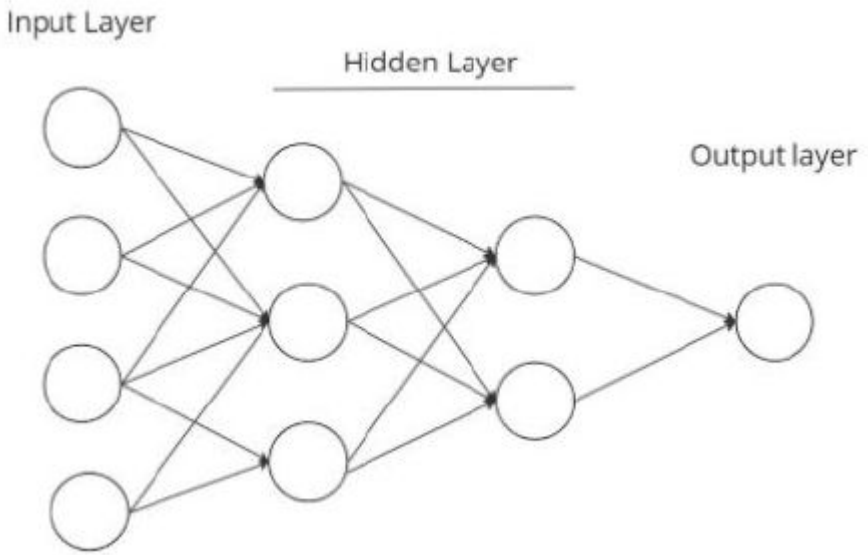


- ***In its most basic form, a Feed-Forward Neural Network is a single layer perceptron. A sequence of inputs enter the layer and are multiplied by the weights in this model. The weighted input values are then summed together to form a total***. If the sum of the values is more than a predetermined threshold, which is normally set at zero, the output value is usually 1, and if the sum is less than the threshold, the output value is usually -1. The single-layer perceptron is a popular feed-forward neural network model that is frequently used for classification. Single-layer perceptrons can also contain machine learning features.



Multi-layer Feed Forward Neural Network

- An entrance point into sophisticated neural networks, where incoming data is routed through several layers of artificial neurons. Every node is linked to every neuron in the following layer, resulting in a fully connected neural network.
- There are input and output layers, as well as several hidden levels, for a total of at least three or more layers. It possesses bidirectional propagation, which means it can propagate both forward and backward.
- Inputs are multiplied by weights and supplied into the activation function, where they are adjusted to minimize loss during backpropagation. Weights are just machine-learned values from Neural Networks.
- They modify themselves based on the gap between projected and training outcomes. Softmax is used as an output layer activation function after nonlinear activation functions.



The Architecture of the Multilayer Feed-Forward Neural Network:

- This Neural Network or Artificial Neural Network has multiple hidden layers that make it a multilayer neural Network and it is feed-forward because it is a network that follows a top-down approach to train the network. In this network there are the following layers:
- **Input Layer:** It is starting layer of the network that has a weight associated with the signals.
- **Hidden Layer:** This layer lies after the input layer and contains multiple neurons that perform all computations and pass the result to the output unit.
- **Output Layer:** It is a layer that contains output units or neurons and receives processed data from the hidden layer, if there are further hidden layers connected to it then it passes the weighted unit to the connected hidden layer for further processing to get the desired result.
- The input and hidden layers use sigmoid and linear activation functions whereas the output layer uses a Heaviside step activation function at nodes because it is a two-step activation function that helps in predicting results as per requirements. All units also known as neurons have weights and calculation at the hidden layer is the summation of the dot product of all weights and their signals and finally the sigmoid function of the calculated sum. Multiple hidden and output layer increases the accuracy of the output.

Learning Factors

- Initial weight: Initial weight is chosen so that training the network requires a greater number of training cycles. Weight changes will improve network training.
- Fixing up the desired output: If we do not appropriately decide the desired output based on input patterns, the network will not be trained.
- Non separable patterns: If two classes of patterns have features that are similar to one another, a network cannot be trained successfully. Features must to be distinct. It is important to appropriately specify the Euclidean distance between the two clusters in order to make pattern recognition simple. Performance will increase as the number of hidden layers increases.
- Learning constant: The learning constant needs to be changed. The training progresses more quickly as the learning rate rises. Example: For learning rate, $\mathbf{10}^{-3}$ to 10 values have been successfully demonstrated. For a few training cycles, the learning rate must be raised; after that, it must be lowered.

Activation Function

- Node(neurons) when receive the input signals, depending on that nature of the signals, intensity we combine them and then send it to activation function. This defines and generates a output on which further actions depend on.
- Types: Linear, Heaviside, Sigmoidal
- Linear: f(v)= a+v, Addition of Bias + Weighted sum
- What is bias and weighted sum?
- The bias in a linear activation function is a constant that is added to the output of the function. This allows the function to be shifted up or down, which can be useful for adjusting the output of the function to fit the data. Weighted sum is addition of input signals to the node.

- Heaviside function: This function works in either 0 or 1 only.
- f(v) is either 1 or 0. f(v)=1 when v>a i.e weighted sum is greater than threshold. Here a is NOT your bias. Eg: Human body part sends signals to the neurons after a threshold value is reached and action is generated.
- Sigmoid function: $f(v) = 1/(1 + e^{-v})$
- Here –v stands for the weighted function.
  A sigmoid function is a mathematical function that has a characteristic "S"-shaped curve or sigmoid curve. It transforms any value in the domain to a number between 0 and 1. The sigmoid function is a S-shaped curve that is often used in classification problems. The output of the sigmoid function from 0 to 1, which makes it well-suited for representing probabilities.
- Artificial Neural Networks use activation function to compute many complex calculations in the hidden layer and then forward the result to the output layer.
- The main aim of activation functions is to introduce non-linear properties in the neural network.

Why do we need non-linearity?

- If we do not use activation function then the output signal would be a linear function. Even if linear equations are easy to solve but they still do have limited complexity quotient.

- Less power to learn complex mapping of data. (Eg. Cats and Dogs)
- Will be called linear regression model with limited functionalities.
- This cannot happen e.g. Speech, images, audio etc
- Activation functions help neural networks make sense of complicated, high dimensional, non linear data that have complex architecture with multiple hidden layers between the input and output.
- A neural network with too many linear layers can become too complex and difficult to train. However, a neural network with a few non-linear layers can be much easier to train and can still learn to approximate complex functions.

Soft and Hard Limiting Function Types

- Soft and hard limiting functions are two types of activation functions used in deep learning. They both introduce **non-linearity** into the neural network, but they do so in different ways.
- **Soft limiting** is a smooth function that limits the output of a neuron to a certain range. The most common soft limiting function is the sigmoid function, which is a S-shaped curve that **ranges from 0 to 1**. The **sigmoid function is often used** in classification problems because it can be used to represent probabilities.
- **Hard limiting** is a step function that limits the output of a neuron **to two values, typically 0 and 1**. The most common hard limiting function is the **Heaviside step** function, which is a step function that is equal to 1 for positive inputs and 0 for negative inputs. Hard limiting functions are often used in regression problems because they can be used to represent binary decisions.

- Hard limiting activation function forces a neuron to output 1 if it's net input reaches a threshold otherwise it gives the output 0. This allows neuron to make decision or classification.

Activation Function	Range	Output	S-shaped	Step Function
Soft limiting	0 to 1	Probability	Yes	No
Hard limiting	0 to 1	Binary	No	Yes

- Soft limiting used to classify the data
- Hard limiting used for prediction.

1. **Linear**: The output of a linear activation function is

the same as the input, or the net value. Range is real values ranging from -∞ to ∞ used in hidden layer in output layer.

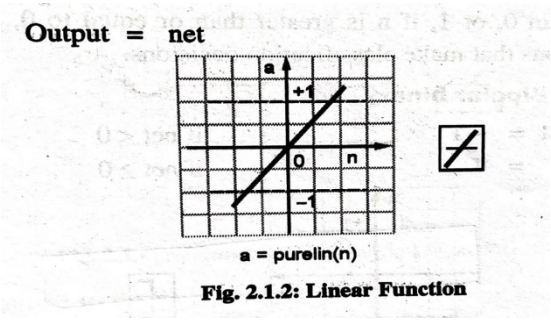
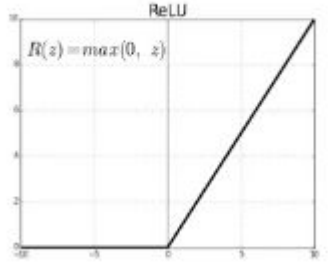


Fig. 2.1.2: Linear Function

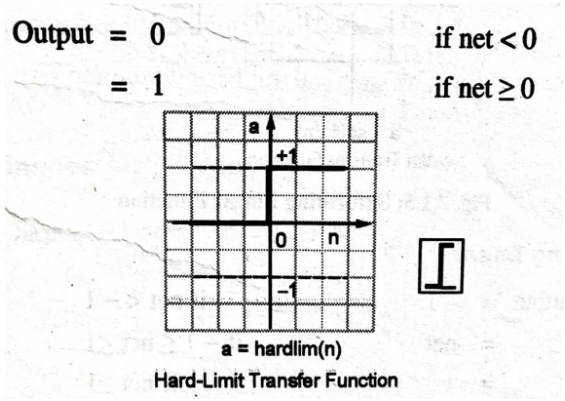
2. **Rectified Linear unit(ReLU**): This is one of the most frequently used activation function in DL models. It is a fast learning function with assured delivery of correct results. This function is nearly linear function that retains the properties of linear models which makes them easy to optimize with gradient-descent methods. ReLU performs a threshold operation on each input element where all values less than 0 are set to 0.

Output= max (0, net)= net if net ≥ 0

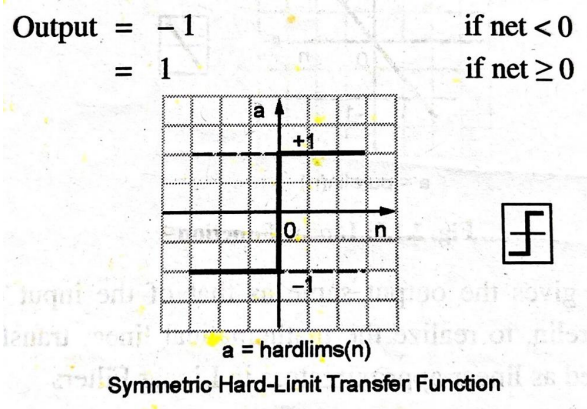
= max (0, net)= 0 if net<0



- Hard limit/ Unipolar binary: The hard limit transfer function depicted above restricts the neuron's output to either 0 or 1 depending on whether the net input argument n is greater than or equal to 0. Perceptron's use this function to build classification-decision-making neurons.

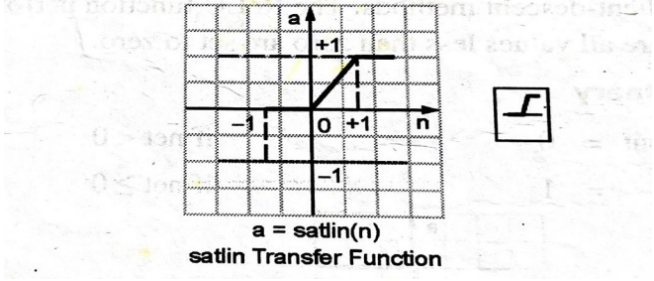


- Symmetrical Hard limit/ Bipolar binary: The symmetrical hard-limit transfer function shown limits the output of the neuron to either -1 if the net input argument n is less than 0, or 1 if n is greater than or equal to 0. This function is used in perceptron’s to make neurons that make classification decisions.



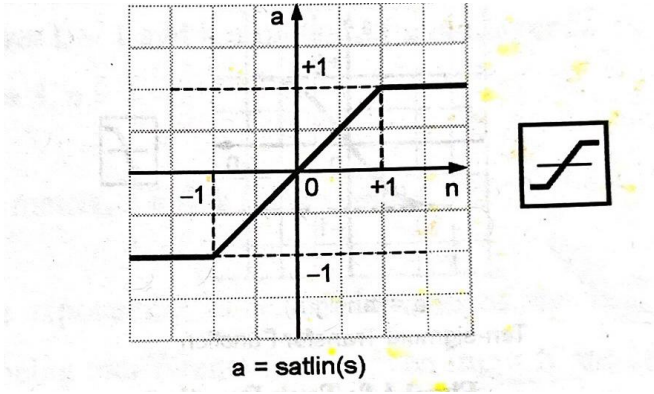
- Saturating linear:

Output = 0 if net < 0
= net if 0 ≤ net < 1
= 1 if net ≥1



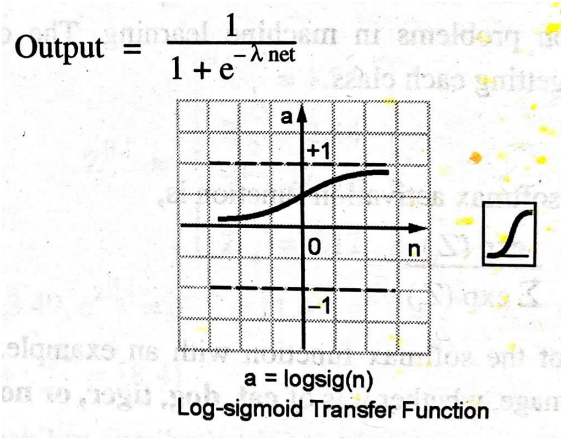
- Symmetrical Saturating linear:

Output = -1 if net< -1
= net if -1≤ net <1
= 1 if net ≥ 1



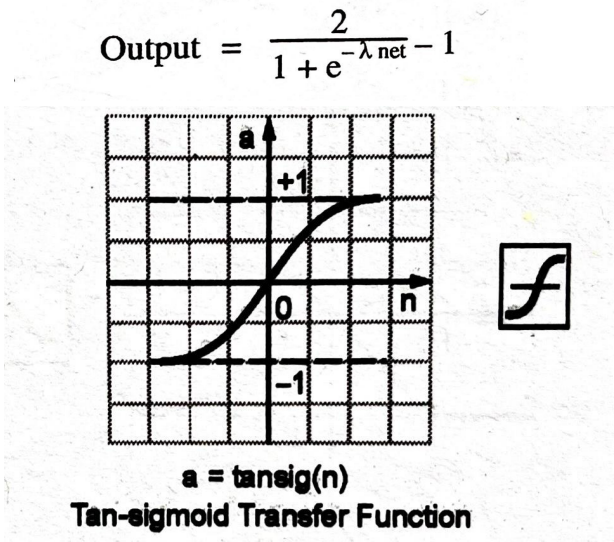
- **Logistic/ Unipolar continuous:**

This function takes the input which can have any value between + and – infinity, and squashes the output into the range 0 to 1. This is commonly used in back propagation networks.



- **Tanh/ Bipolar continuous:**

This function takes the input which can have any value between + and – infinity, and squashes the output into the range -1 to 1. This is used in case of multi class classification problems in ML.



- **Softmax**: Softmax is an activation function that scales numbers/logits into probabilities. This function is normally used in last layer of neural network.

$$\text{Softmax}(Z_i) = \frac{\exp(Z_i)}{\Sigma \exp(Z_i)}$$

- It converts a vector of real numbers into a probability distribution, where each element represents the likelihood of the corresponding class. The softmax function is particularly useful when you have more than two classes, and you want the output to represent the probabilities of each class.

$$\sigma(\vec{z})_i = \frac{e^{z_i}}{\sum_{j=1}^{K} e^{z_j}}$$

σ = softmax

$\vec{z}$ = input vector

e^{z_i} = standard exponential function for input vector

K = number of classes in the multi-class classifier

e^{z_j} = standard exponential function for output vector

Range: (0,1), sum of output = 1

Pros: *Can handle multiple classes and give the probability of belonging to each class*

Cons: *Should not be used in hidden layers as we want the neurons to be independent. If we apply it then they will be linearly dependent.*

- **Leaky ReLU**: Leaky Rectified Linear Unit, or Leaky ReLU, is a type of activation function based on a ReLU, but it has a small slope for negative values instead of a flat slope. If the input to the function x is positive then the output will be x, otherwise the output will be ax, where a is a small magnitude. (eg: 0.01, 0.03 etc).
- Mathematically this will be:

$$F(X)= \max(0.05X, X)$$

Rather than completely blocking negative values, it allows them to slightly leak.

Example 2.1.1 : Given a 2 input neuron with following parameters, b = 1.2, w = [3, 2], x = [–5, 6]. Calculate neuron's output for following transfer functions.

1. Hard limit
2. Symmetrical Hard limit
3. Linear
4. Saturating linear
5. Symmetrical saturating linear
6. Unipolar continuous
7. Bipolar continuous

☑ **Soln. :**

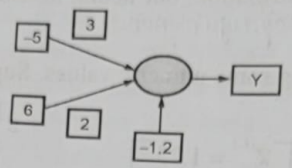


Fig. Ex. 2.1.1

First we calculate the net value which can be calculated as follows

$$net_i = \sum_{j=1}^{n} W_{ij} X_j$$

But, in this particular problem bias value is also given, Bias is a value which is added in the net to improve the performance of the network so we will use the following formula

$$net_i = \sum_{j=1}^{n} W_{ij} X_j + b$$

$$net = (-5*3) + (6*2) + 1.2 = -1.8$$

Now we will calculate the final output for the given activation functions

1. Hard limit

$$\begin{aligned} \text{Output} &= 0 && \text{if, net} < 0 \\ &= 1 && \text{if, net} \geq 0 \\ \text{Hence, } Y &= 0 \end{aligned}$$

2. Symmetrical Hard limit

$$\begin{aligned} \text{Output} &= -1 && \text{if, net} < 0 \\ &= 1 && \text{if, net} \geq 0 \\ \text{Hence, } Y &= -1 \end{aligned}$$

3. Linear

$$\begin{aligned} \text{Output} &= \text{net} \\ \text{Hence, } Y &= -0.8 \end{aligned}$$

4. Saturating linear

$$\begin{aligned} \text{Output} &= 0 && \text{if, net} < 0 \\ &= \text{net} && \text{if, } 0 \leq \text{net} < 1 \\ &= 1 && \text{if, net} \geq 1 \\ \text{Hence, } Y &= 0 \end{aligned}$$

5. Symmetrical saturating linear

$$\begin{aligned} \text{Output} &= -1 && \text{if, net} < -1 \\ &= \text{net} && \text{if, } -1 \leq \text{net} < 1 \\ &= 1 && \text{if, net} \geq 1 \\ \text{Hence, } Y &= -1 \end{aligned}$$

6. Unipolar continuous

$$\text{Output} = \frac{1}{(1+\exp(-\lambda*\text{net}))}$$

We assume the value of λ equal to 1(If the value of λ is not given then assume the standard value as equal to 1)

Hence, Y = 0.1418

7. Bipolar continuous

$$\text{Output} = \frac{2}{(1+\exp(-\lambda*\text{net}))} - 1$$

$$\text{Hence, } Y = -0.71$$

Loss function

- A loss function is a function that compares the target and predicted output values; measures how well the neural network models the training data. When training, we aim to minimize this loss between the predicted and target outputs.
- The hyperparameters are adjusted to minimize the average loss — we find the weights (wt) and biases (b) that minimize the value of J (average loss).

Types of loss functions

- In supervised learning, there are two main types of loss functions — these correlate to the 2 major types of neural networks:
- Regression and Classification loss functions

Categories

- Regression Loss Functions — used in regression neural networks; given an input value, the model predicts a corresponding output value (rather than pre-selected labels); Ex. Mean Squared Error, Mean Absolute Error.
- We predict continuous values eg: predicting the price of a house based on size location etc. It measures how far our predictions are from the actual values/ close we are from the correct answer.
- Classification Loss Functions — used in classification neural networks; given an input, the neural network produces a vector of probabilities of the input belonging to various pre-set categories — can then select the category with the highest probability of belonging; Ex. Binary Cross-Entropy, Categorical Cross-Entropy
- Eg: Identifying whether image is cat or dog. Classification loss helps measure how well our model can make predictions i.e. how much accurate the choices are.

Mean Squared error

- MSE finds the average of the squared differences between the target and the predicted outputs

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

- This function has numerous properties that make it especially suited for calculating loss. The difference is squared, which means it does not matter whether the predicted value is above or below the target value; however, values with a large error are penalized.
- The result is always positive regardless the sign of predicted and actual value.

Binary Cross Entropy:

- Used for classification only when two classes are present (Yes or No)
- BCE loss is the default loss function used for the binary classification tasks. It requires one output layer to classify the data into two classes and the range of output is (0–1) i.e. should use the sigmoid function.

$$L = -\frac{1}{m}\sum_{i=1}^{m} y_i \cdot \log(\hat{y}_i)$$

where y is the actual label, ŷ is the classifier's predicted probability distributions for predicting one class and m is the number of records.

Range: (0,inf)

Categorical Cross Entropy

It is the default loss function when we have a multi-class classification task. It requires the same number of output nodes as the classes with the final layer going through a *softmax* activation so that each output node has a probability value between (0–1).

$$logloss = -\frac{1}{N}\sum_{i}^{N}\sum_{j}^{M} y_{ij}\log(p_{ij})$$

- N is the number of rows
- M is the number of classes

where y is the actual label and p is the classifier's predicted probability distributions for predicting the class j

Range: (0,inf)

Choosing Output function and Loss function

CASE 1: When the output is numerical value that you are trying to predict.

- Eg: Consider predicting the price of a house. A neural network structure where the final layer or the output layer will consist of only one neuron that reverts the numerical value. For computing the accuracy score, the predicted values are compared to true numeric values.
- Activation Function: Linear activation and ReLU activation.

The linear activation function allows the network to directly output a numeric value without restricting it to a specific range, making it suitable for regression tasks.

- Loss function to be used in such cases: Mean Squared Error (MSE):

This loss function is responsible to compute average squared difference between the true and the predicted values.

CASE 2: When the output you are trying to predict is Binary.

- Eg: Consider where we are supposed to predict if a student is going to pass or fail. In such cases the output will consist of only one neuron that is responsible to result in value that is between 0 and 1, also called as probabilistic scores.
- For computing the accuracy of the prediction, the value is again compared with the true labels. The true value is 1 if data belongs to that class or else it is 0.
- Activation Function: Sigmoid.

It squashes the output into a range between 0 and 1, which can be interpreted as a probability.

The output can be seen as the probability that the given input belongs to the positive class

- Loss function to be used in such cases: BCE

BCE loss measures the difference between the predicted probability and the true label (0 or 1).

CASE 3: Predicting a single class from many classes

- Consider predicting name of the fruit among the 4 different fruits given. In this case the output layer will consist of only one neuron for every class and it will revert a value between 0 and 1.
- Each output is checked with its respective true value to get the accuracy.
- The output will be 1 for the correct class and 0 for the wrong
- Activation Function: Softmax.

Softmax is typically used in the output layer for multiclass classification problems.

It transforms the raw output scores into probabilities, ensuring that the sum of the probabilities for all classes is equal to 1.

- Loss function to be used in such cases: Cross entropy

Cross-Entropy loss, or categorical cross-entropy, is commonly used as the loss function for multiclass classification problems. It measures the dissimilarity between the predicted probability distribution and the true distribution (one-hot encoded vectors representing the true class).

CASE 4: Predicting multiple labels from multiple class.

- Consider predicting different objects in an image having multiple objects. This is also known as multiclass classification. In this type, output layer consists of only one neuron that is responsible to result in value between 0 and 1. For computing the accuracy it is again compared with true labels. The true value is 1 if data belongs to that class or else it is 0.
- Activation Function: Sigmoid

The softmax activation function is commonly used in the output layer for multiclass classification problems. It transforms the raw output scores into probabilities, ensuring that the sum of the probabilities for all classes is equal to 1.

- Loss function to be used in such cases: BCE

Categorical Cross-Entropy loss is a suitable loss function for multiclass classification problems. It measures the dissimilarity between the predicted probability distribution and the true distribution

Activation and loss functions

Problem	Output Type	Activation Function	Loss Function
Regression	Numerical	Linear	Mean squared error
Classification	Binary	Sigmoid	Binary cross entropy
Classification	Single label, multiple class	Softmax	Cross entropy
Classification	Multiple label, multiple class	Sigmoid	Binary cross entropy

OPTIMIZATION

Back propagation is a supervised learning algorithm that aims to minimize the error between the predicted outputs of the neural network and the actual target values.

The process involves adjusting the weights of the network based on the gradient of the loss function with respect to the weights.

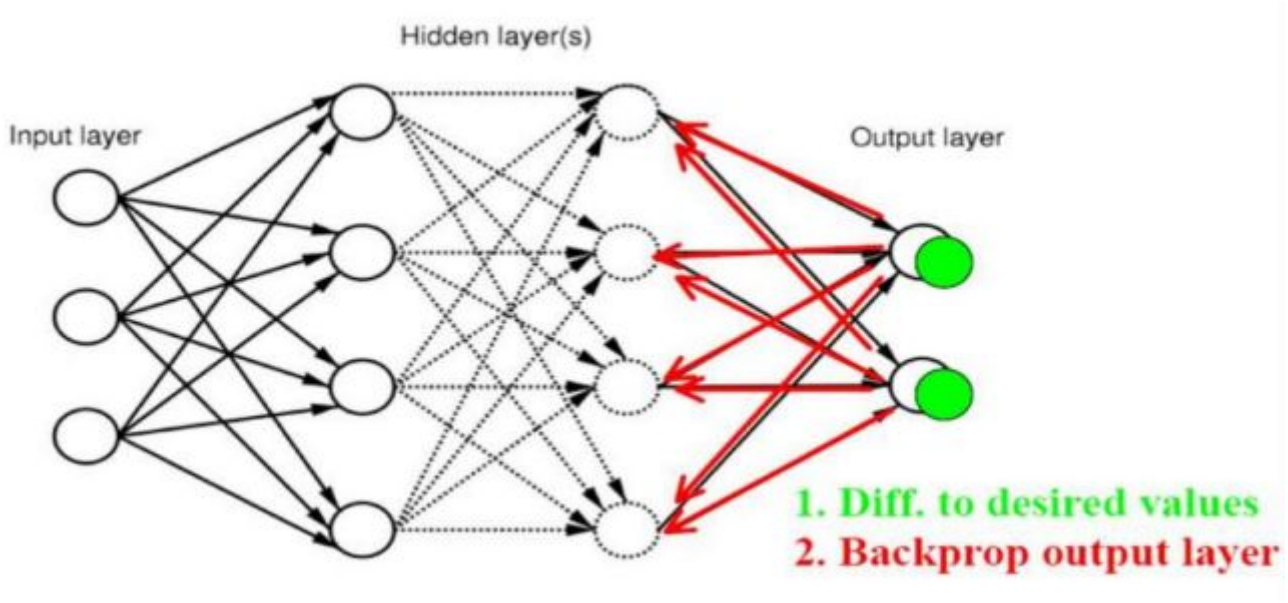
Back Propagation Algorithm:

Steps:

1. First input is applied to the input layer to calculate the output of hidden layer. Output of hidden layer becomes input of next layer. Finally we get output of output layer.
2. The desired and the actual output at the out layer are compared with each other and an error signal is generated.
3. Error of output layer is back propagated to the hidden layer so that weights connected in each layer of the network can be adjusted.
4. Back propagation network is trained for correct classification for training data then testing data is applied in order to check if patterns are correctly classified or not.

Back propagation

Learning with back propagation is an iterative process, and through repeated adjustments to the weights, the neural network gradually learns to map input data to the desired output. The process involves finding a set of weights that minimizes the error on the training data, allowing the model to make accurate predictions on new, unseen data.



LEARNING PARAMETERS

- Gradient Descent: Gradient descent is an iterative optimization algorithm for finding the minimum of a function. The goal is to minimize the given function i.e. the loss in the neural networks. It acts in two steps:

I. Compute the gradient slope that is the first order derivative of the function at the current point.

II. Move-in the opposite direction of the slope increase from the current point by the computed amount.

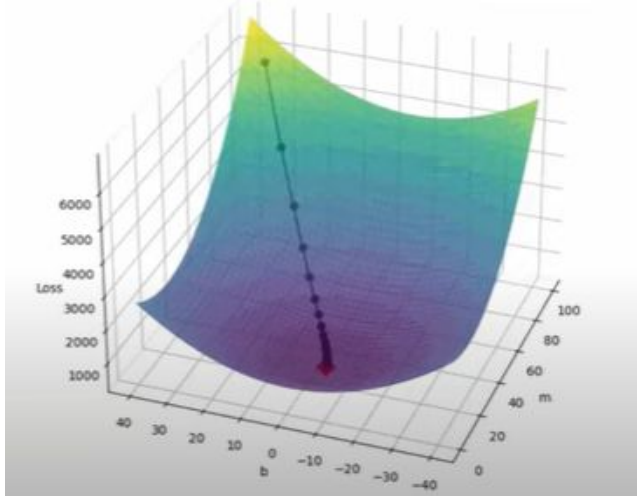
Basically idea is to pass the training set through the hidden layers of neural network and then update the parameters of layers by computing the gradients using training samples from training dataset.

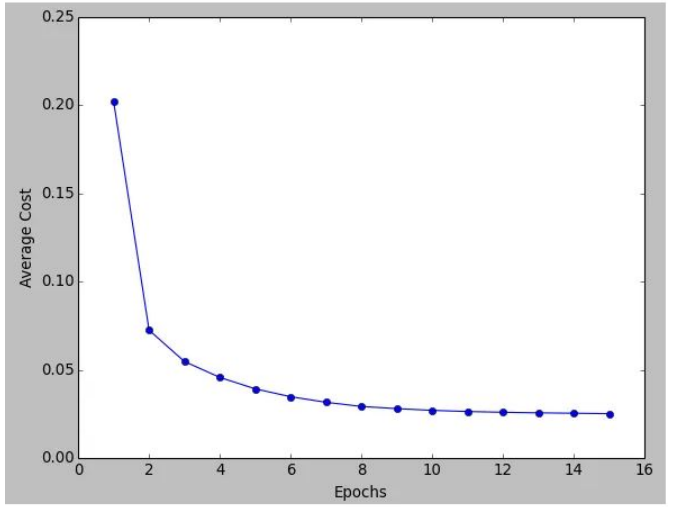
Types of Gradient Descent

- Batch Gradient Descent optimiser
- Stochastic Gradient Descent
- Minibatch Gradient descent optimiser
- Momentum based Gradient
- Nesterov Accelerated Gradient Optimizer
- AdaGrad
- RMSProp
- Adam

Batch Gradient Descent

- Batch Gradient Descent involves calculations over the full training set at each step as a result of which it is very slow on very large training data. Thus, it becomes very computationally expensive to do Batch GD.
- This is great for convex or relatively smooth error manifolds. Also, Batch Gradient Descent scales well with the number of features.
- So that means just one step of gradient descent in one epoch.





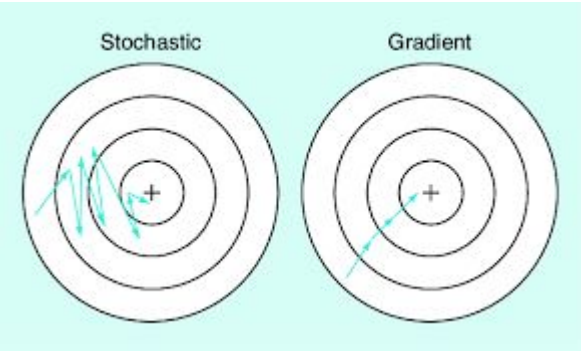
Stochastic Gradient Descent:

- What if the we have huge data. For one step we will have to calculate gradient of all the examples.
- Does not seem efficient. SGD is stochastic in nature i.e. it picks up a “random” instance of training data at each step and then computes the gradient, making it much faster as there is much fewer data to manipulate at a single time, unlike Batch GD.
- In SGD consider one example at a time to take single step:
- Steps in one epoch for SGD:

Steps in one epoch for SGD:

- Take the example Feed it to Neural Network Calculate the gradient Use the gradient to update the weights Repeat the steps for all examples in training dataset.

- In BGD only 1 random point is used but in Stochastic all the points, dataset is used hence variation.



Mini Batch Gradient Descent:

- Mini-batch gradient descent optimizer combines both BGD and SGD concepts. It splits the whole dataset and takes the subsets to calculate the loss function.
- These subsets are called batches. The parameters are updated on each of those batches. So, it requires fewer iterations and a lot less time to find the local minimum.
- This makes the process faster than both batch and stochastic gradient descent algorithms.
- This process requires the memory to only store the data needed for training and calculating the error. This makes it more efficient.

Momentum based Gradient Descent:

- Momentum is an extension to the gradient descent optimization algorithm that allows the search to build inertia in a direction in the search space and overcome the oscillations of noisy gradients and coast across flat spots of the search space.
- Faster convergence: If the gradient points consistently in the same direction, MGD builds up momentum and takes bigger steps, reaching the minimum faster.
- Smoother progress: MGD can "glide" over shallow valleys and avoid getting stuck in local minima. Noise in the gradients gets averaged out, leading to a more stable descent.
- Better escape from saddle points: Saddle points are like flat spots surrounded by hills. MGD's momentum can help the algorithm push through these points and find the true minimum.

- The choice of the momentum coefficient (β) is important. A higher β leads to more momentum but can also make the algorithm less stable.
- MGD can be combined with other optimization techniques, such as adaptive learning rate methods, for even better performance.

$$\Delta w = -\eta * \nabla L(w) + \beta * \Delta w\_prev$$

Δw is the update to the weights (parameters)

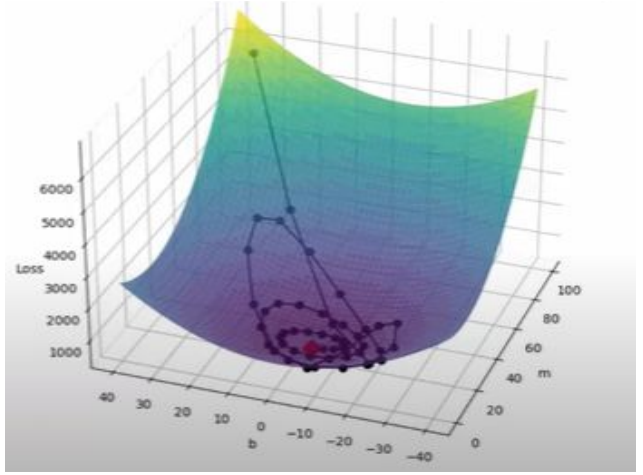
η is the learning rate

∇L(w) is the gradient of the loss function at the current weights

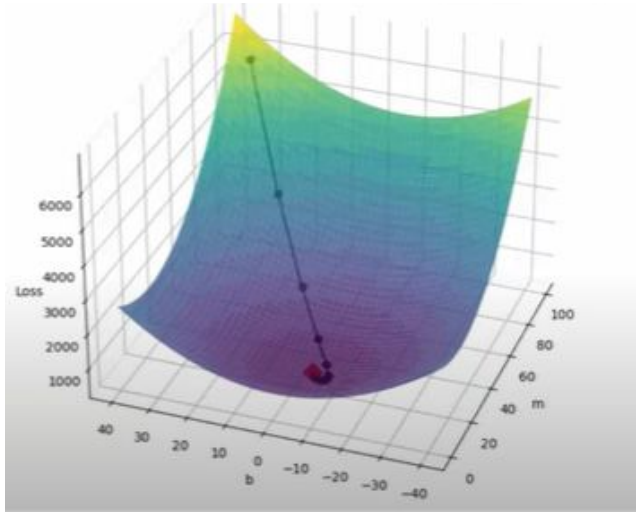
β is the momentum coefficient (usually between 0 and 1)

Δw_prev is the previous weight update

- Momentum with suppose 0.8 decay (β)



- Momentum with suppose 0.4 decay (β)



Vanishing and Exploding Gradient

In a deep neural network, if the model is trained with gradient descent and backpropagation, there can occur two more issues other than local minima and saddle point.

- Vanishing Gradients:

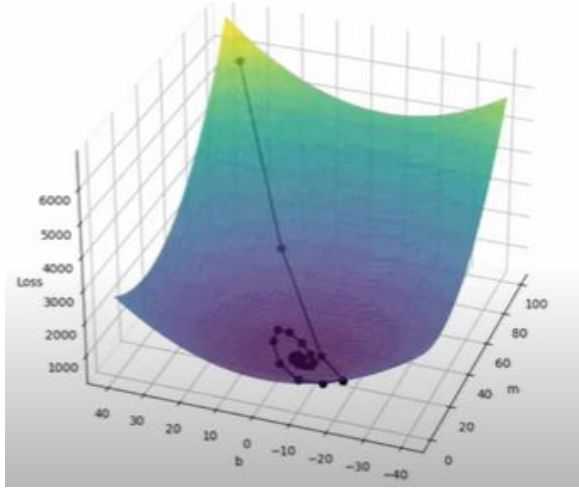
Vanishing Gradient occurs when the gradient is smaller than expected. During backpropagation, this gradient becomes smaller that causing the decrease in the learning rate of earlier layers than the later layer of the network. Once this happens, the weight parameters update until they become insignificant.

- Exploding Gradient:

Exploding gradient is just opposite to the vanishing gradient as it occurs when the Gradient is too large and creates a stable model. Further, in this scenario, model weight increases, and they will be represented as NaN. This problem can be solved using the dimensionality reduction technique, which helps to minimize complexity within the model.

Nesterov Accelerated Gradient (NAG):

- Though the momentum based gradient optimizer can be considered a good approach, it has its downside. Depending on the momentum, the algorithm finds the convergence.
- For higher momentum, this process can miss the minima.
- To overcome this problem, Nesterov accelerated gradient optimizer is used and it is called the look-ahead approach, Where the momentum optimizer moves toward the direction of updated gradients, this algorithm moves toward the direction of previous gradients and makes corrections. Therefore, updating the model and slowly approaching the minimum point. This results in fewer iterations and saves more time.
- NAG is an extension of Momentum Gradient Descent. It evaluates the gradient at a hypothetical position ahead of the current position based on the current momentum vector, instead of evaluating the gradient at the current position. This can result in faster convergence and better performance.



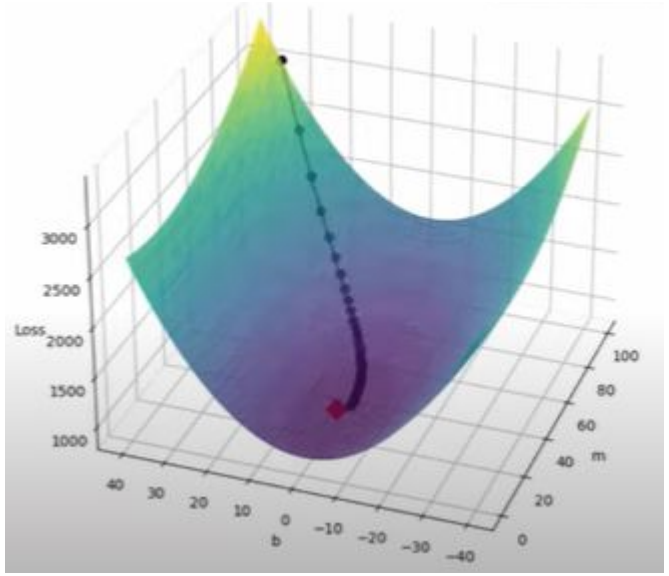
Adagrad (Adaptive gradient):

- In this variant, the learning rate is adaptively adjusted for each parameter based on the historical gradient information.
- Adaptive gradient is a technique used in machine learning to help algorithms learn more efficiently. It adjusts the learning rate for each parameter in a model based on how quickly or slowly that parameter is changing.
- This way, it can speed up learning for slow-changing parameters and slow down learning for fast-changing ones, leading to better and faster convergence during training.
- Adagrad is mostly used when the data is sparse. (0 output data)

- Adagrad focuses on the learning rate for training the models.
- The key idea is to have an adaptive learning rate that can change according to the updates and no need to tune manually
- It means the learning rate decreases if there are larger updates and the accumulated history of squared gradients keeps growing.
- Unfortunately, in this case, the learning rate can decrease massively and approach zero at one point but the speed of computation is comparatively faster in AdaGrad.

- Adagrad with suppose 0.8 decay (β)

- It uses more optimized way for calculations.

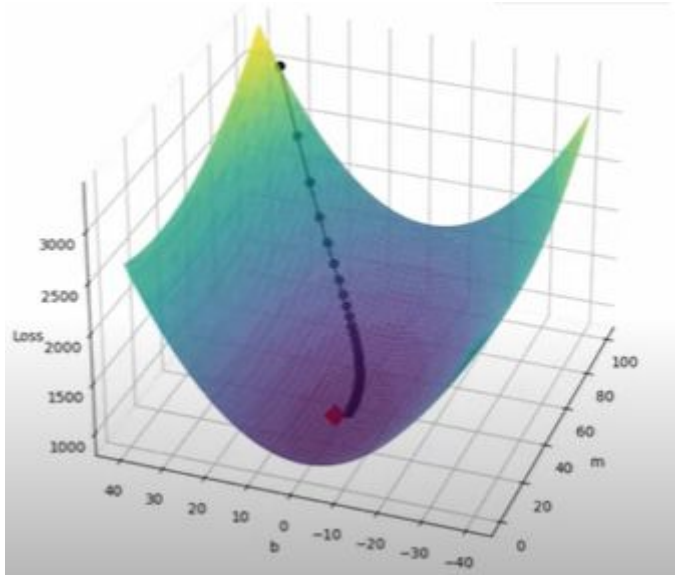


RMSProp (Root Mean Square Propagation):

- In this variant, the learning rate is adaptively adjusted for each parameter based on the moving average of the squared gradient. This helps the algorithm to converge faster in the presence of noisy gradients.
- RMSProp helps the model take the right step size during training. It keeps track of how steep the slope is for each parameter and adjusts the step size accordingly. If the slope is steep (gradients are large), it takes smaller steps to avoid overshooting. If the slope is gentle (gradients are small), it takes bigger steps to speed up progress.
- Doing this the process get smoother and faster, helping the model find the best value for its parameters and become better. It works as a guide to help model navigate up and down.

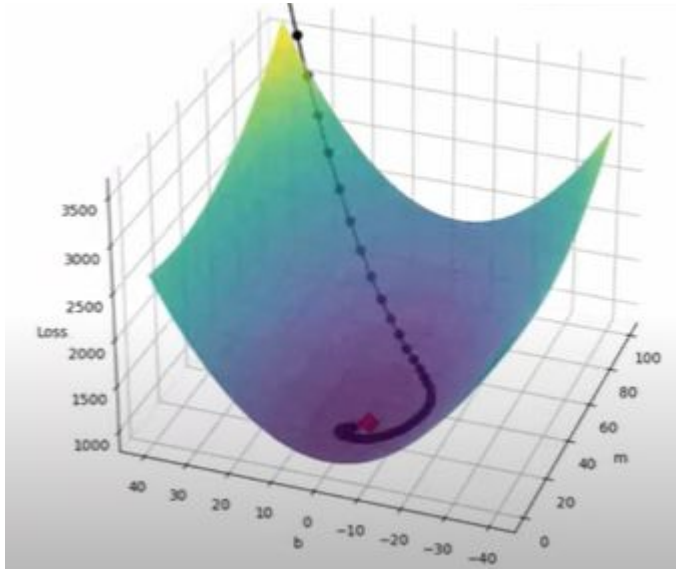
RMSProp

- Adagrad will not reach the minima value in case of complex problems but RMSProp can.
- They both look similar in the graphical way but this one reaches the minimal value.



Adam: Adaptive Moment Estimation

- Most effective method. Adam is an extension to gradient descent and a natural successor to techniques which automatically adapts a learning rate for each input variable for the objective function and further smooths the search process by using an exponentially decreasing moving average of the gradient to make updates to variables.
- It calculates an exponentially weighted average of past gradients, stores it in variables v (before bias correction) and v_corrected (after bias correction)
- It calculates an exponentially weighted average of the squares of the past gradients, stores it in variables s (before bias correction) and s_corrected (after bias correction)
- Adam graph shows both momentum and learning rate decay functions. Graph for sparse data is seen with momentum in the end.



Adadelta

An improvement to the learning rate decreasing problem of AdaGrad optimizer is AdaDelta.

Instead of taking the sum of accumulated gradients, AdaDelta takes the average of the squared gradients. Based on this, it tunes the learning rate.

The average is calculated from a fixed number of past squared gradients. There is no need to set an initial learning rate in AdaDelta.

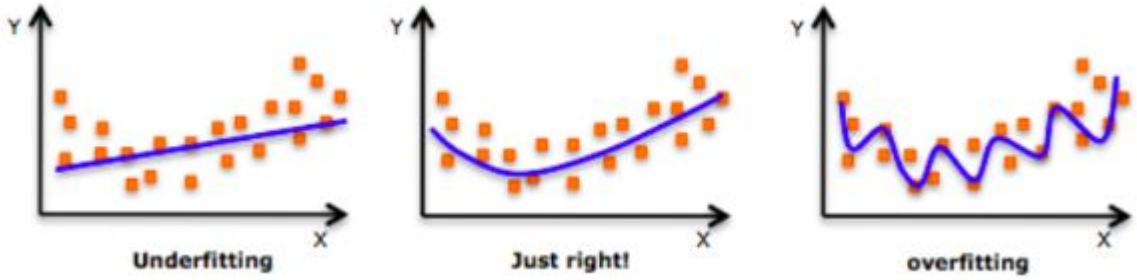
If the gradients are pointing in the right direction, the step size increases. But for the opposite case, the step size is reduced and the weights are updated accordingly.

Different optimizers comparison

OPTIMIZER	DETAILS	ADVANTAGES	ISSUES
Batch Gradient Descent	Updates the parameters once the gradient of the entire dataset is calculated	Easy to compute, understand and implement	Can be very slow and requires large memory
Stochastic Gradient Descent	Instead of the entire dataset the calculation is done on few samples of data	Faster than BGD and takes less memory	Gradient results can be noisy and takes a lot of time to find minima
Mini-batch Gradient Descent	Splits whole dataset into subsets and parameters are updated after calculating the loss function of the subsets	Faster and more efficient than SGD	For too small learning rate, the process can be very slow and the updated gradients can be noisy
Momentum Based Gradient Descent	Reduces the noise of updated gradients and makes the process faster	Faster convergence and takes less memory	Computation of a new parameter at each update
Nesterov Accelerated Gradient	Moves toward the direction of past gradients, makes corrections and slowly approaches minima	Decreases the number of iterations and makes the process faster	Computation of a new parameter at each update
AdaGrad	Focuses on the learning rate and it can adjust according to the updates based on the sum of past gradients	Learning rate changes automatically with iterations	Massive decrease in learning rate can lead to slow convergence
AdaDelta	Adjusts learning rate based on the average of past squared gradients	Learning rate does not decrease massively	Computation cost can be high
Adam	Computation is based on both the average of past gradients and past squared gradients	Faster than others	Computation cost can be high

REGULARIZATION

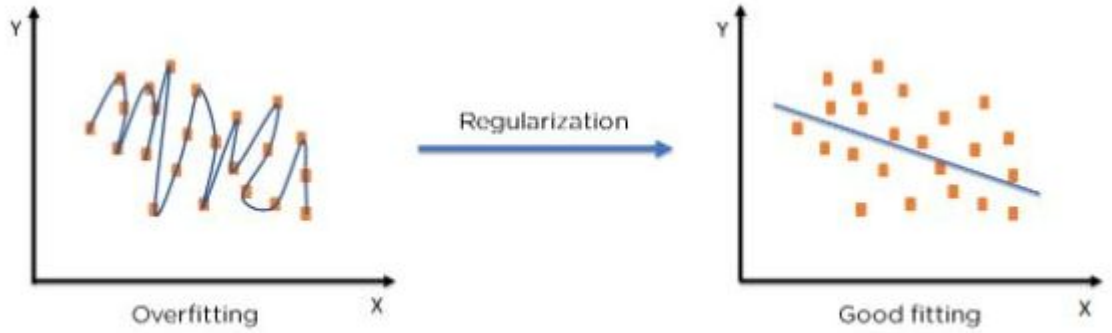
- **Regularization** is a technique which makes slight modifications to the learning algorithm such that the model generalizes better. This in turn improves the model's performance on the unseen data as well.



- As we move towards the right in this image, our model tries to learn too well the details and the noise from the training data, which ultimately results in poor performance on the unseen data.
- Regularization helps to keep the model's complexity in check by adding a penalty term to the loss function during training. This penalty discourages the model from relying too heavily on any specific feature or combination of features, making it more generalizable and better to handle.
- Example: Levels while playing games, recognizing cats and dogs based on particular features or background.

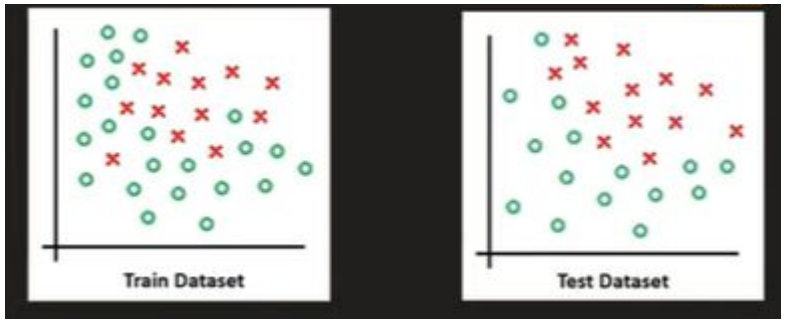
Overfitting

- Overfitting: A statistical model is said to be overfitted when the model does not make accurate predictions on testing data. When a model gets trained with so much data, it starts learning from the noise and inaccurate data entries in our data set.
- Then the model does not categorize the data correctly, because of too many details and noise. The causes of overfitting are the non-parametric and non-linear methods because these types of machine learning algorithms have more freedom in building the model based on the dataset and therefore they can really build unrealistic models.
- When Machine Learning model is trained with large amount of data, it starts capturing noise and inaccurate data. It affects the performance of the system. Here train error is small and test error is large.
- ANN, decision tree examples will try to capture the minute details with an intention of reducing the error on the data. So a boundary curve will be created on the training dataset.
- The same boundary curve will not be seen in the test dataset case which will lead into bad performance. Accuracy will be less and this concept is known as Overfitting.



Causes of Overfitting

1. High variance and low bias
2. The model is too complex
3. The size of the training data



Techniques to reduce overfitting:

- Increase training data
- Reduce complexity of model
- Early stopping during the training phase(have an eye over the loss over the training period as soon as loss begins to increase stop the training)
- Data augmentation(Bunch of fruits with specific color, size rules. If too complicated the robot might struggle. Be more general)
- Regularization
- Main reason behind this is using non-linear methods which build non-realistic data.
- We can overcome this by using linear data, increasing training data and reducing the noise.
- (Train error: If we apply model to the data it was trained on, we calculate the train error)

Underfitting

- Underfitting: A statistical model or a machine learning algorithm is said to have underfitting when it cannot capture the underlying trend of the data, i.e., it only performs well on training data but performs poorly on testing data. (It's just like trying to fit undersized pants!)
- Machine Learning is trained with less data. It provides incomplete and inaccurate data. Model initially gives us the incorrect predictions. It occurs when model is too simple to understand and basic.
- Underfitting destroys the accuracy of our machine learning model. Its occurrence simply means that our model or the algorithm does not fit the data well enough.

Causes for Underfitting

- High bias and low variance
- The size of the training dataset used is not enough.
- The model is too simple.
- Training data is not cleaned and also contains noise in it.

Techniques to reduce Underfitting

- Increase model complexity
- Increase the number of features, performing feature engineering
- Remove noise from the data.
- Increase the number of epochs or increase the duration of training to get better results.
- Main reason it happens is when there is limited data and we try to build linear model with non-linear data.
- We can overcome this by increasing complexity, removing noise, training or increasing better features.
- (Test error: If we calculate error on data which was unknown, we calculate test error)

Bias-Variance Trade-off

- **Bias**:

The bias is known as the difference between the prediction of the values by the ML model and the correct value. Being high in biasing gives a large error in training as well as testing data. Its recommended that an algorithm should always be low biased to avoid the problem of underfitting.

By high bias, the data predicted is in a straight line format, thus not fitting accurately in the data in the data set. Such fitting is known as Underfitting of Data. This happens when the hypothesis is too simple or linear in nature. Refer to the graph given below for an example of such a situation.

- **Variance**:

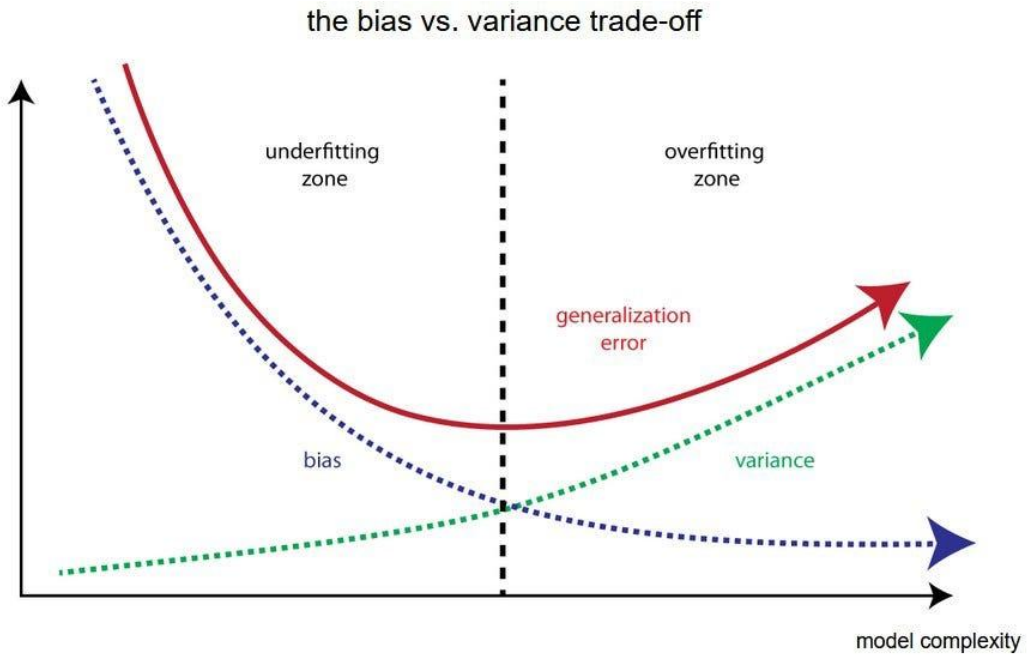
The variability of model prediction for a given data point which tells us spread of our data is called the variance of the model. The model with high variance has a very complex fit to the training data and thus is not able to fit accurately on the data which it hasn't seen before. As a result, such models perform very well on training data but has high error rates on test data.

When a model is high on variance, it is then said to as Overfitting of Data. Overfitting is fitting the training set accurately via complex curve and high order hypothesis but is not the solution as the error with unseen data is high.

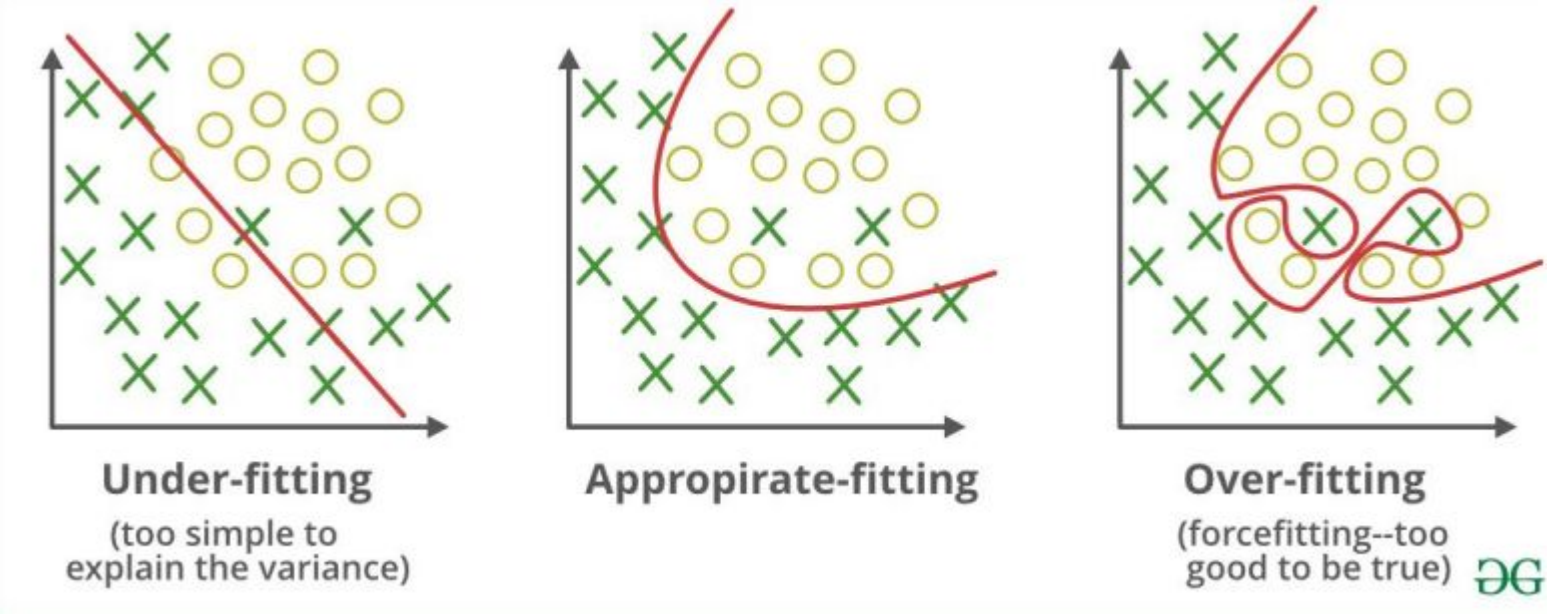
While training a data model variance should be kept low.

Bias-Variance Trade-off

- The bias is known as the difference between the prediction of the values by the model and the correct value. It is recommended that an algorithm should always be low-biased to avoid the problem of underfitting.
- The variability of model prediction for a given data point which tells us the spread of our data is called the variance of the model. When a model is high on variance, it is then said to as Overfitting of Data. While training a data model variance should be kept low.



- If the algorithm is too simple then it may be on high bias and low variance condition and thus is error-prone. If algorithms fit too complex then it may be on high variance and low bias. In the latter condition, the new entries will not perform well. Well, there is something between both of these conditions, known as a Trade-off or Bias Variance Trade-off.
- So, the Bias-Variance Trade-off reminds us to find an algorithm's sweet spot, where it's just right in complexity, striking the right balance between bias and variance for better overall performance on various datasets.



Regularization

- Regularization is one of the most important concepts of machine learning. It is a technique to prevent the model from overfitting by adding extra information to it.
- Sometimes the machine learning model performs well with the training data but does not perform well with the test data. It means the model is not able to predict the output when deals with unseen data by introducing noise in the output, and hence the model is called overfitted. This problem can be deal with the help of a regularization technique.
- This technique can be used in such a way that it will allow to maintain all variables or features in the model by reducing the magnitude of the variables. Hence, it maintains accuracy as well as a generalization of the model.
- It mainly regularizes or reduces the coefficient of features toward zero. In simple words, "In regularization technique, we reduce the magnitude of the features by keeping the same number of features."

$$y= \beta0+\beta1x1+\beta2x2+\beta3x3+\cdots+\beta nxn +b$$

In the above equation, Y represents the value to be predicted, X1, X2, ...Xn are the features for Y.

β0,β1,.....βn are the weights or magnitude attached to the features, respectively. Here represents the bias of the model, and b represents the intercept.

Regularization Methods

1. Lasso
2. Ridge

Difference between L1 and L2 regularisation

- Ridge regression is mostly used to reduce the overfitting in the model, and it includes all the features present in the model. It reduces the complexity of the model by shrinking the coefficients.
- Lasso regression helps to reduce the overfitting in the model as well as feature selection.

L1 Regularization (Lasso): This method adds a penalty to the neural network's loss function proportional to the absolute values of the model's weights. It encourages the model to use only the most important features, effectively leading to sparse weight values, as some weights may become exactly zero. This helps in feature selection and simplifies the model. Ex: Predicting student's performance depending on factors.

- Lasso regression is another regularization technique to reduce the complexity of the model. It stands for Least Absolute and Selection Operator.
- It is similar to the Ridge Regression except that the penalty term contains only the absolute weights instead of a square of weights. Since it takes absolute values, hence, it can shrink the slope to 0, whereas Ridge Regression can only shrink it near to 0. It is also called as L1 regularization.

The equation for the cost function of Lasso regression will be:

$$\sum_{i=1}^{M}(y_i - y'_i)^2 = \sum_{i=1}^{M}\left(y_i - \sum_{j=0}^{n}\beta_j * x_{ij}\right)^2 + \lambda\sum_{j=0}^{n}|\beta_j|$$

L2 Regularization (Ridge):

L2 regularization adds a penalty to the loss function based on the squared values of the model's weights. It discourages large weight values and forces the model to use all the features but with smaller weights. L2 regularization prevents overemphasis on any particular feature and promotes a more balanced usage of features. Ex: Predicting the cost of a house depending on the factors.

- Ridge regression is one of the types of linear regression in which a small amount of bias is introduced so that we can get better long-term predictions.
- Ridge regression is a regularization technique, which is used to reduce the complexity of the model. It is also called as L2 regularization.
- In this technique, the cost function is altered by adding the penalty term to it. The amount of bias added to the model is called Ridge Regression penalty. We can calculate it by multiplying with the lambda to the squared weight of each individual feature.

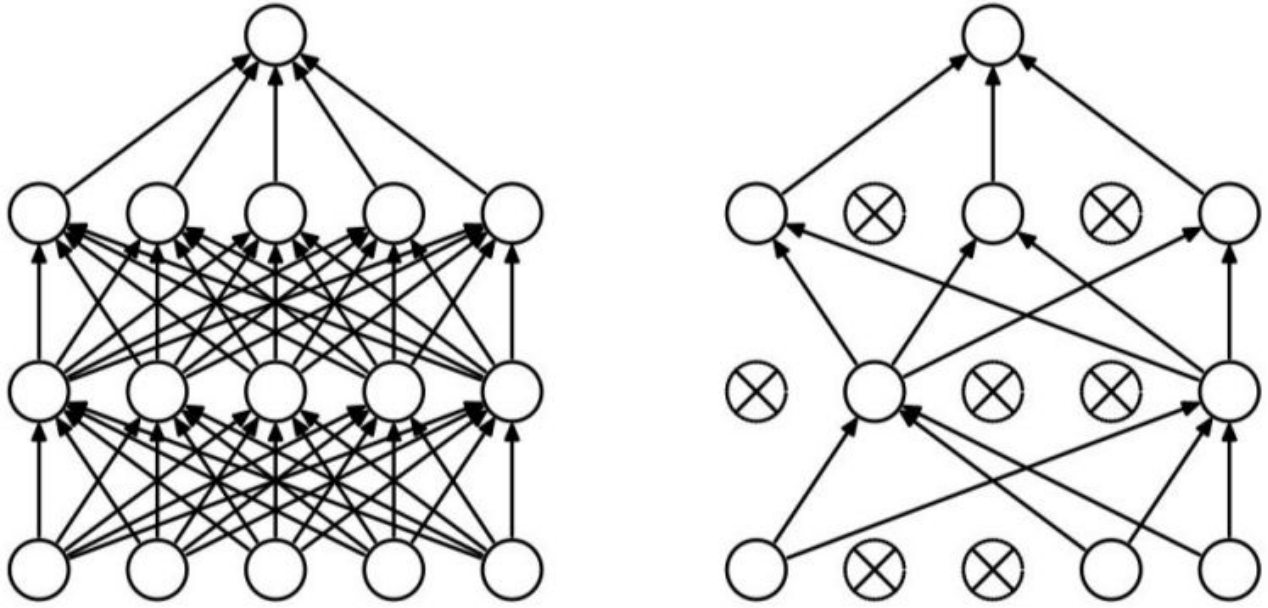
$$\sum_{i=1}^{M}(y_i - y'_i)^2 = \sum_{i=1}^{M}\left(y_i - \sum_{j=0}^{n}\beta_j * x_{ij}\right)^2 + \lambda\sum_{j=0}^{n}\beta_j{}^2$$

In the above equation, the penalty term regularizes the coefficients of the model, and hence ridge regression reduces the amplitudes of the coefficients that decreases the complexity of the model.

As we can see from the above equation, if the values of λ tend to zero, the equation becomes the cost function of the linear regression model. Hence, for the minimum value of λ, the model will resemble the linear regression model.

Dropout:

Dropout is a technique where, during training, random neurons (units) in the neural network are temporarily dropped or ignored with a certain probability. This prevents neurons from becoming overly dependent on each other, promoting robustness and reducing overfitting. Dropout essentially simulates an ensemble of multiple subnetworks and improves the model's generalization. Ex: Teacher trying to train a group of students to solve, you need a deep learning model to help identify areas where each student needs improvement.



(a) Standard Neural Net

(b) After applying dropout.

Early Stopping:

Early stopping is a method that allows you to specify an arbitrary large number of training epochs and stop training once the model performance stops improving on a hold out validation dataset. Early stopping is a simple technique where the training process is halted when the performance on a validation set stops improving. It prevents the model from overfitting by finding the optimal point where the model's performance on new data is the best.

Ex: Imagine preparing for GATE exam. You have a model that takes input as factors related to your study habits, practice time, and previous exam performances. The model's goal is to predict your GATE performance accurately and help you identify areas where you need to focus more.

Key benefits of early stopping:

- Prevents overfitting: Helps to avoid models that are too complex and memorize the training data, leading to better generalization to new data.
- Reduces training time: Can often save time by stopping training before it becomes unnecessary.

- **Batch Normalization**: Batch normalization normalizes the outputs of each layer in a neural network, making the training process more stable. It fixes the means and variances of the input by bringing the features in the same range. It can implicitly regularize model and is preferred over Dropout. Ex: You are preparing a recipe that requires baking a cake. You have a deep learning model that takes as input various factors, such as the ingredients used, the baking time, and the oven temperature. The model's goal is to predict the cake's taste and texture based on these factors.
- **Data Augmentation**: Data augmentation is not a traditional regularization technique, but it is a widely used method in deep learning. It involves generating additional training data by applying various transformations, such as rotations, flips, translations, and brightness adjustments, to the original data. This artificially increases the size of the training dataset and helps the model generalize better.
- **Parameter sharing**: It is a technique used in machine learning and deep learning to reduce the number of learnable parameters in a model. It involves using the same set of parameters (weights) for multiple parts of the model, allowing the model to learn shared patterns and features more efficiently.

- **Weight decay:** This is also known as L2 regularization, is a regularization technique used in deep learning to prevent overfitting. It involves adding a penalty to the loss function during training that encourages the model's weights to be small, reducing the complexity of the model.
- **Adding noise to input and output**: It is a regularization technique used in deep learning to improve the generalization of a model. It involves intentionally introducing random noise to the input data and/or the output data during training. By doing so, the model becomes more robust and less likely to overfit to the training data.