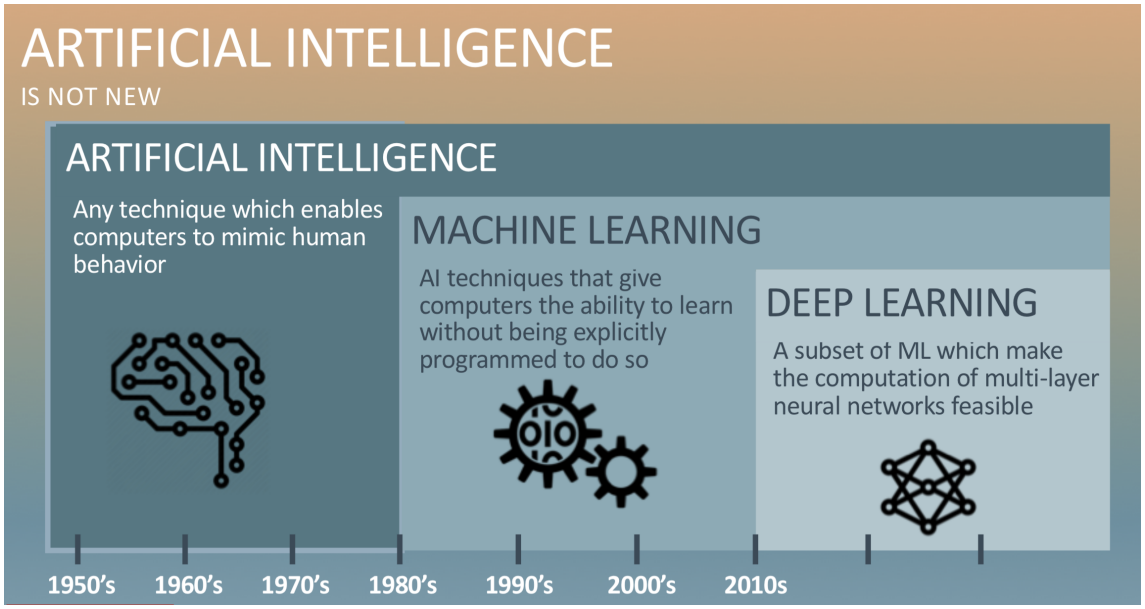


Deep Learning Introduction

- Deep learning is a branch of machine learning which is completely based on artificial neural networks, as neural network is going to mimic the human brain so deep learning is also a kind of mimic of human brain.
- it is an artificial intelligence (AI) function that imitates the workings of the human brain in processing data and creating patterns for use in decision making.

KEY TAKEAWAYS

- Deep learning is an AI function that mimics the workings of the human brain in processing data for use in detecting objects, recognizing speech, translating languages, and making decisions.
- Deep learning AI is able to learn without human supervision, drawing from data that is both unstructured and unlabeled.
- Deep learning, a form of machine learning, can be used to help detect fraud or money laundering, among other functions.

Key Difference between Machine Learning and Deep Learning :

S.No.	Machine Learning	Deep Learning
1.	Machine Learning is a superset of Deep Learning	Deep Learning is a subset of Machine Learning
2.	The data represented in Machine Learning is quite different as compared to Deep Learning as it uses structured data	The data representation is used in Deep Learning is quite different as it uses neural networks(ANN).
3.	Machine Learning is an evolution of AI	Deep Learning is an evolution to Machine Learning. Basically it is how deep is the machine learning.
4.	Machine learning consists of thousands of data points.	Big Data: Millions of data points.
5.	Outputs: Numerical Value, like classification of score	Anything from numerical values to free-form elements, such as free text and sound.
6.	Uses various types of automated algorithms that turn to model functions and predict future action from data.	Uses neural network that passes data through processing layers to the interpret data features and relations.
7.	Algorithms are detected by data analysts to examine specific variables in data sets.	Algorithms are largely self-depicted on data analysis once they're put into production.
8.	Machine Learning is highly used to stay in the competition and learn new things.	Deep Learning solves complex machine learning issues.

Types of Deep Learning Algorithms That I Coverd In Notebook

1. Multilayer Perceptrons (MLPs)
2. Convolutional Neural Networks (CNNs)
3. Recurrent Neural Networks (RNNs)
4. Long Short Term Memory Networks (LSTMs)
5. Generative Adversarial Networks (GANs)
6. Restricted Boltzmann Machines(RBMs)
7. Autoencoders
8. Self Organizing Maps (SOMs)

There are so many techniques but in my note book i will focus on this 8 topic.

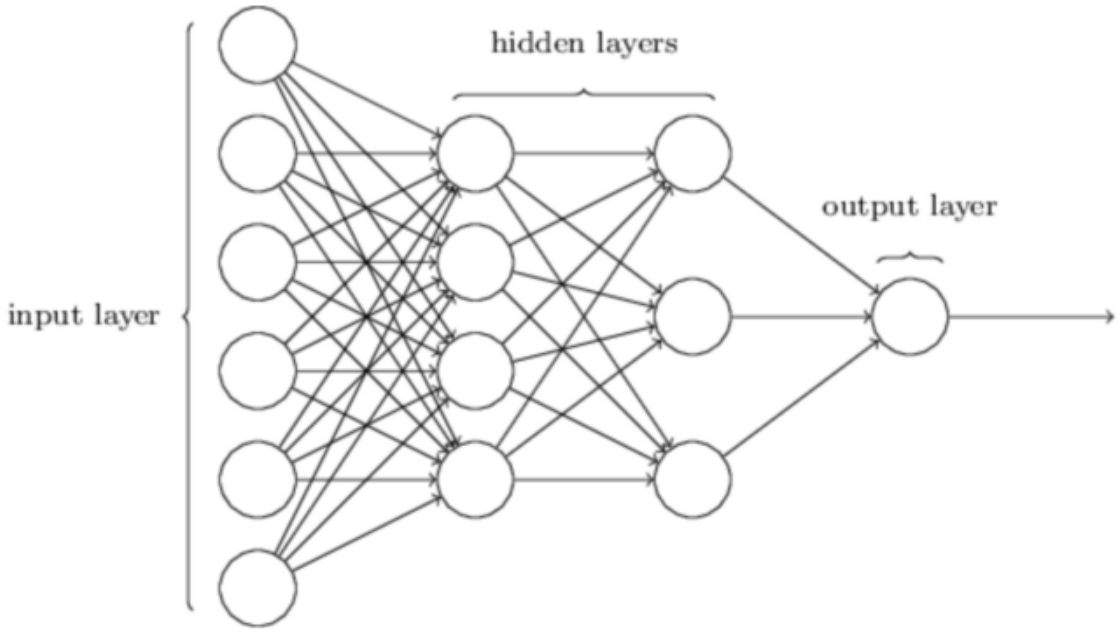
Neural Networks

- Before Deep Dive in to deep learning first see some important terminology regarding this

Q1. What are Neural networks?

- Neural networks are set of algorithms inspired by the functioning of human brian. Generally when you open your eyes, what you see is called data and is processed by the Nuerons(data processing cells) in your brain, and recognises what is around you. That's how similar the Neural Networks works. They takes a large set of data, process the data(draws out the patterns from data), and outputs what it is.

- A neural network is composed of layers, which is a collection of neurons, with connections between different layers. These layers transform data by first calculating the weighted sum of inputs and then normalizing it using the activation functions assigned to the neurons.

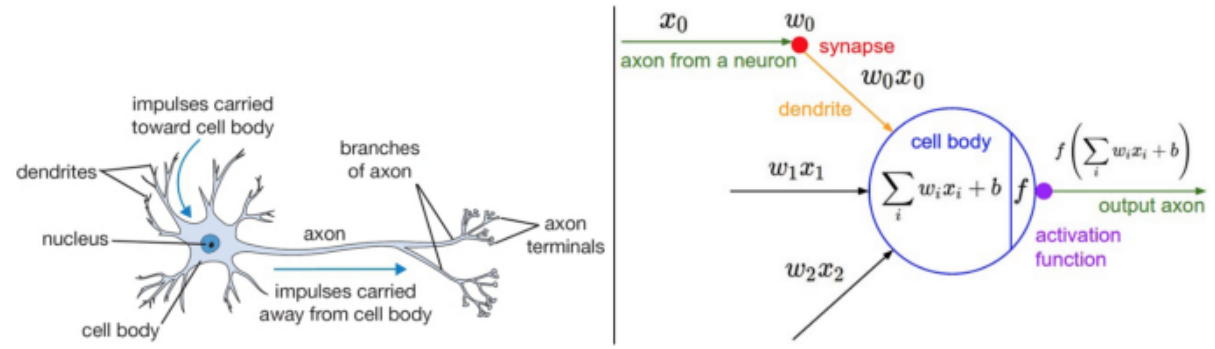


- The leftmost layer in a Neural Network is called the input layer, and the rightmost layer is called the output layer. The layers between the input and the output, are called the hidden layers. Any Neural Network has 1 input layer and 1 output layer.
- The number of hidden layers differ between different networks depending on the complexity of the problem. Also, each hidden layer can have its own activation function.

- Here 3 terms Comes in picture 1. Neuron , 2. Weights , 3. Bias , 4. Actiation_Function

1. Neuron

- Like in a human brain, the basic building block of a Neural Network is a Neuron. Its functionality is similar to a human brain, i.e, it takes in some inputs and fires an output. Each neuron is a small computing unit that takes a set of real valued numbers as input, performs some computation on them, and produces a single output value.
- The basic unit of computation in a neural network is the neuron, often called as a node or unit. It receives input from some other nodes, or from an external source and computes an output. Each input has an associated weight (w), which is assigned on the basis of its relative importance to other inputs. The node applies a activation function f (defined below) to the weighted sum of its inputs as in figure below.



A cartoon drawing of a biological neuron (left) and its mathematical model (right).

The above network have:

- numerical inputs X1 and X2
- weights w1 and w2 associated with those inputs
- b (called the Bias) associated with it.

The Left side Picture is Neuron Of Human Brain ,The Right Side is Artificial Neuron

Biological Neuron Work:

- Information from other neurons, in the form of electrical impulses, enters the dendrites at connection points called synapses. The information flows from the dendrites to the cell where it is processed. The output signal, a train of impulses, is then sent down the axon to the synapse of other neurons.

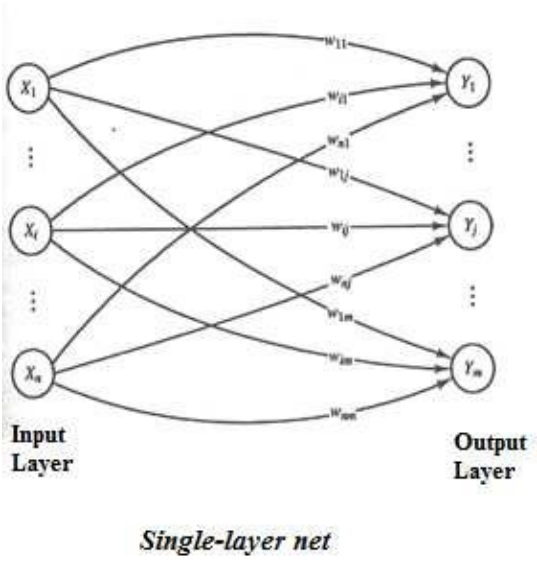
Artificial Neuron Work:

- The arrangements and connections of the neurons made up the network and have three layers.
- The first layer is called the input layer and is the only layer exposed to external signals.
- The input layer transmits signals to the neurons in the next layer, which is called a hidden layer. The hidden layer extracts relevant features or patterns from the received signals.
- Those features or patterns that are considered important are then directed to the output layer, which is the final layer of the network.

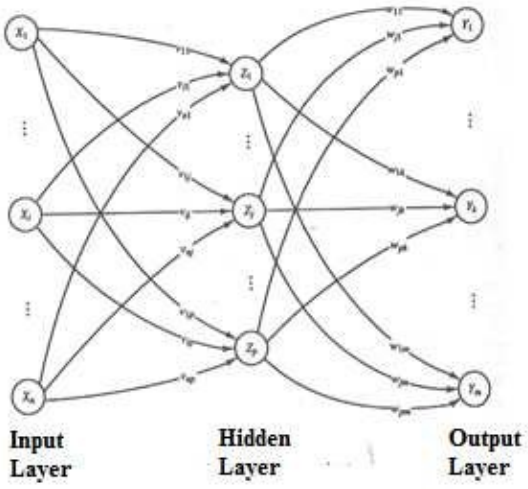
Difference between the human brain and computers in terms of how information is processed.

Human Brain(Biological Neuron Network)	Computers(Artificial Neuron Network)
The human brain works asynchronously	Computers(ANN) work synchronously.
Biological Neurons compute slowly (several ms per computation)	Artificial Neurons compute fast (<1 nanosecond per computation)
The brain represents information in a distributed way because neurons are unreliable and could die any time.	In computer programs every bit has to function as intended otherwise these programs would crash.
Our brain changes their connectivity over time to represents new information and requirements imposed on us.	The connectivity between the electronic components in a computer never change unless we replace its components.
Biological neural networks have complicated topologies.	ANNs are often in a tree structure.
Researchers are still to find out how the brain actually learns.	ANNs use Gradient Descent for learning.

Single Layers And Multi Layer Netwrok



Single-layer net



Multi-layer net

- In Multi Layer net there are many number of hidden layer in between input and output layer,but in single only one or not hidden layer.

Weight:

- Every input(x) to a neuron has an associated weight(w), which is assigned on the basis of its relative importance to other inputs.
- The way a neuron works is, if the weighted sum of inputs is greater than a specific threshold, it would give an output 1, otherwise an output 0. This is the mathematical model of a neuron, also known as the Perceptron.
- Every neural unit takes in a weighted sum of its inputs, with an additional term in the sum called a Bias.

Bias:

- Bias is a constant which is used to adjust the output along with the weighted sum of inputs, so that the model can best fit for the given data.

$$z=w.x+b$$

I defined

- weighted sum z
- weight vector w
- input vector x
- bias value b.

$$y=a=f(z)$$

- The output(y) of the neuron is a function f of the weighted sum of inputs z. The function f is non linear and is called the Activation Function.

Activation Function: ¶

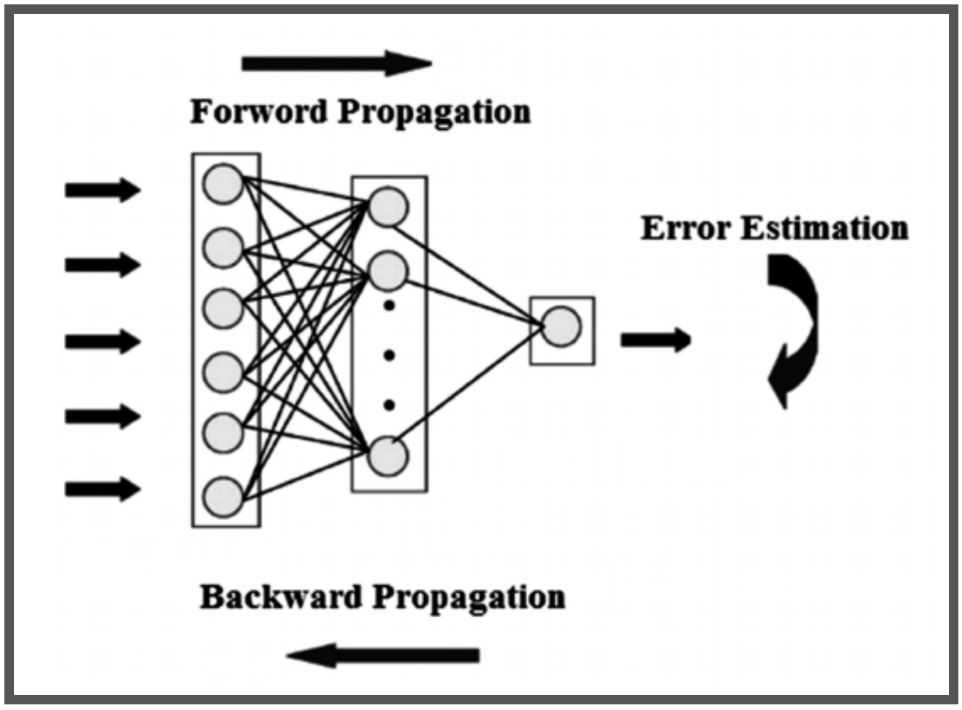
- The purpose of activation function is to introduce non-linearity into the output of neuron. It takes a single number, and performs some mathematical operation on it. There are several activation functions used in practice:

    1. Sigmoid
    2. Tanh
    3. ReLU
    4. Leaky relu
    5. Softmax function
- These Are most widly use activation function that i covered in subsequent notebook.

Forward And Backward Propogation

Every Neural Network has 2 main parts:

1. Feed Forward Propogation/Forward Propogation.
2. Backward Propogation/Back propogation.



Feed Forward Propogation:

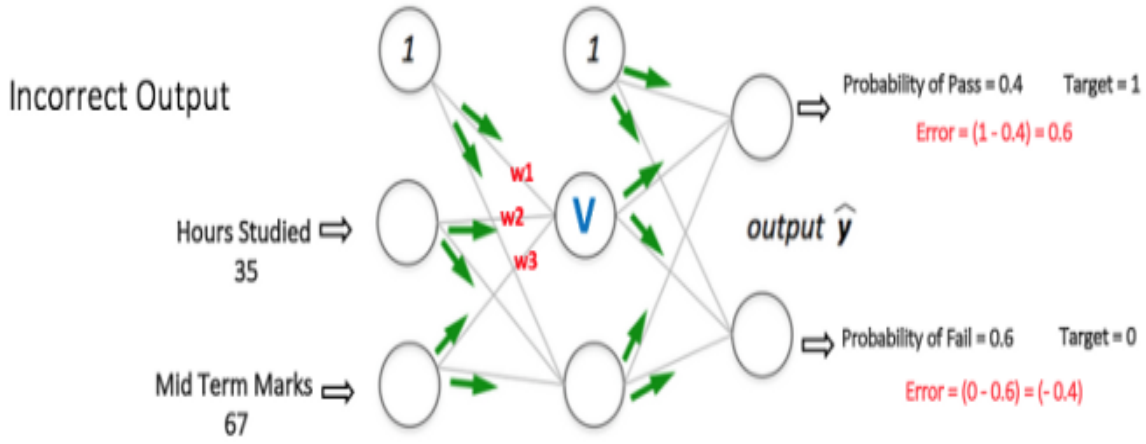
- All weights in the network are randomly assigned. Assume the weights of the connections from the inputs to that node are w1, w2 and w3.
- Here I take an example to better understand of this two concept
- Let take an example of if you study 35 Hour per day then you definitly Pass the exam with 67 Mark. Now we apply this.

    - Input to the network = [35, 67]
    - Desired output from the network (target) = [1, 0] 1 means PASS and 0 means Fail

Then output V from the node in consideration can be calculated as below (f is an activation function such as sigmoid):

$$V = f(1*w1 + 35*w2 + 67*w3)$$

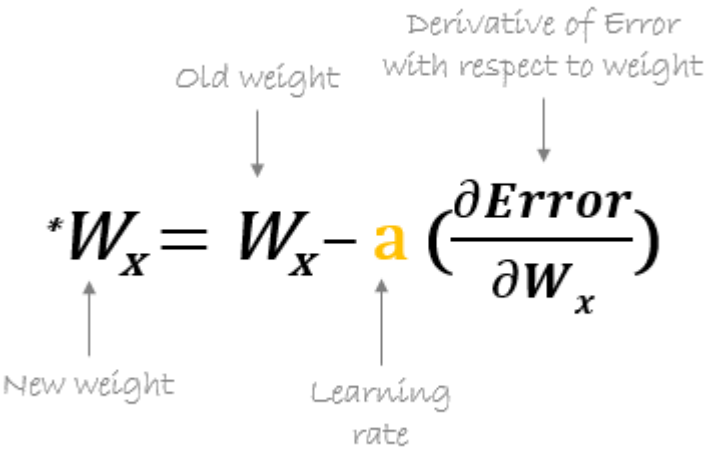
- Similarly, outputs from the other node in the hidden layer is also calculated. The outputs of the two nodes in the hidden layer act as inputs to the two nodes in the output layer. This enables us to calculate output probabilities from the two nodes in output layer.



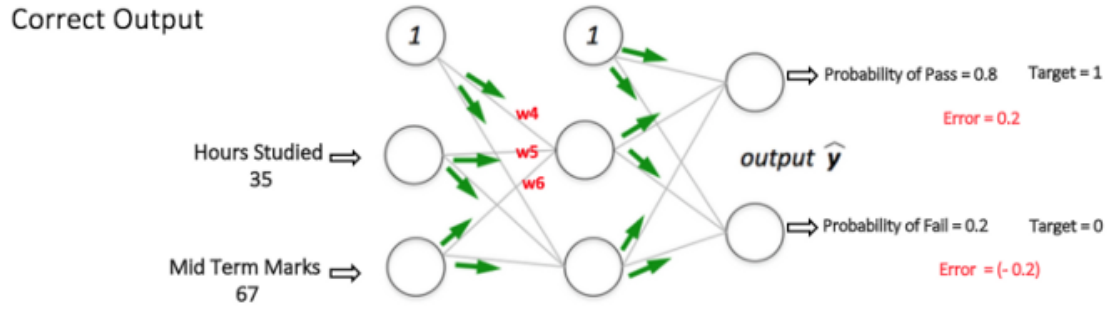
- Suppose the output probabilities from the two nodes in the output layer are 0.4 and 0.6 respectively (since the weights are randomly assigned, outputs will also be random). We can see that the calculated probabilities (0.4 and 0.6) are very far from the desired probabilities (1 and 0 respectively), hence the network in above Figure is said to have an 'Incorrect Output'. As it give output as fail but it not happen that one study 35 hr and secure 67.
- here Weight are randomly assigned so now we do Back Propagation and with Weight Updation.

Back Propagation and Weight Updation:

- Here We calculate the total error at the output nodes and propagate these errors back through the network using Backpropagation to calculate the gradients.
- Then we use an optimization method such as Gradient Descent to 'adjust' all weights in the network with an aim of reducing the error at the output layer.



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



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

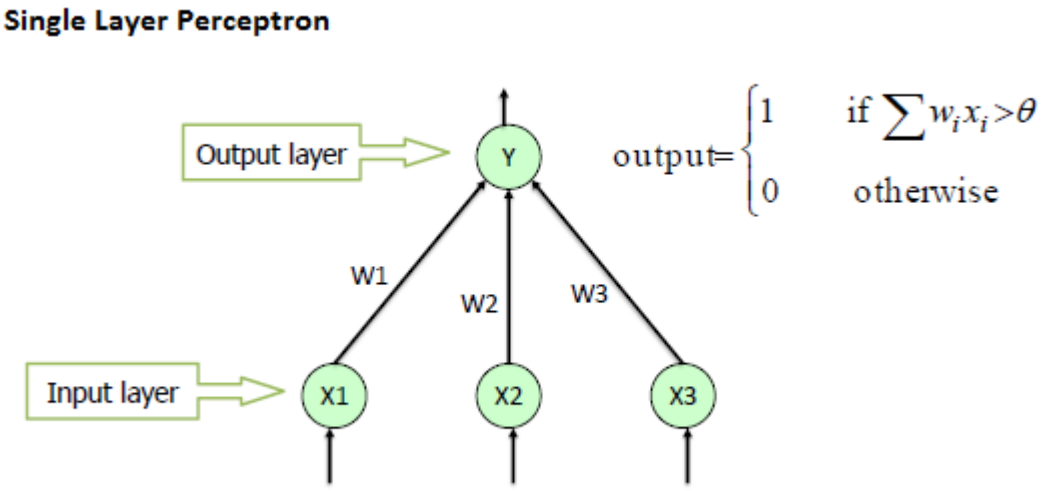
```
1. Thanks To https://medium.com/@purnasaigudikandula/a-beginner-intro-to-neural-networks-543267bda3c8#:~:text=Neural%20networks%20are%20set%20of,the%20functioning%20of%20human%20brian.&text=That's%20how%20similar%20the%20Neural,and%20outputs%20what%20it%20is.
2. http://home.agh.edu.pl/~vlsi/AI/backp_t_en/backprop.html
```

What is Perceptron?

Perceptron is a single layer neural network and a multi-layer perceptron is called Neural Networks.

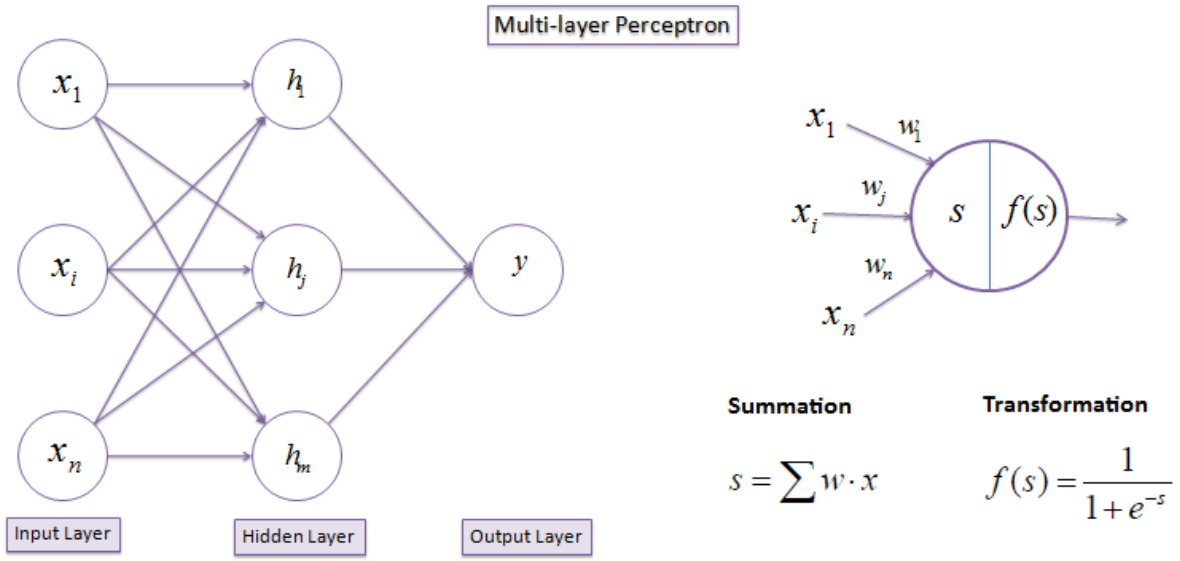
1. Single-layered perceptron model
2. Multi-layered perceptron model.

1. Single-layered perceptron model



- If you talk about the functioning of the single-layered perceptron model, its algorithm doesn't have previous information, so initially, weights are allocated inconstantly, then the algorithm adds up all the weighted inputs,
- if the added value is more than some pre-determined value(or, threshold value) then single-layered perceptron is stated as activated and delivered output as +1.
- In simple words, multiple input values feed up to the perceptron model, model executes with input values, and if the estimated value is the same as the required output, then the model performance is found out to be satisfied, therefore weights demand no changes. In fact, if the model doesn't meet the required result then few changes are made up in weights to minimize errors.

2. Multi-layered perceptron model



- In the forward stage, activation functions are originated from the input layer to the output layer, and in the backward stage, the error between the actual observed value and demanded given value is originated backward in the output layer for modifying weights and bias values.
- In simple terms, multi-layered perceptron can be treated as a network of numerous artificial neurons overhead varied layers, the activation function is no longer linear, instead, non-linear activation functions such as Sigmoid functions, TanH, ReLU activation Functions, etc are deployed for execution.

Activation Function

- Activation function decides, whether a neuron should be activated or not by calculating weighted sum and further adding bias with it. The purpose of the activation function is to introduce non-linearity into the output of a neuron.

Explanation :-

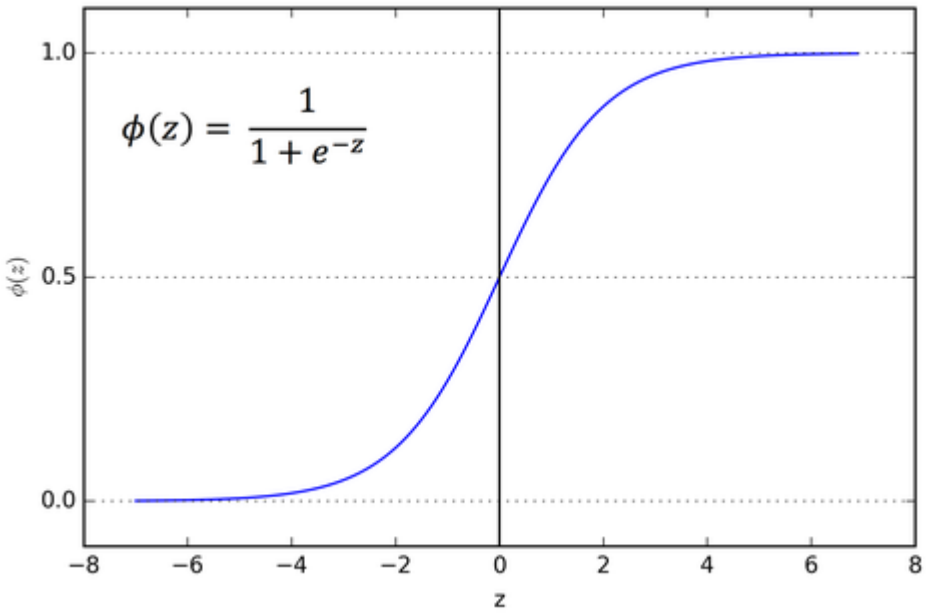
We know, neural network has neurons that work in correspondence of weight, bias and their respective activation function. In a neural network, we would update the weights and biases of the neurons on the basis of the error at the output. This process is known as back-propagation. Activation functions make the back-propagation possible since the gradients are supplied along with the error to update the weights and biases.
Why do we need Non-linear activation functions :-
A neural network without an activation function is essentially just a linear regression model. The activation function does the non-linear transformation to the input making it capable to learn and perform more complex tasks.

There are several type of Activation But here i dicuss some of them:

1. Sigmoid (Binary Classification)
2. Tanh
3. Relu
4. Leaky Relu
5. Linear
6. Softmax (Use Multiclass Classification)

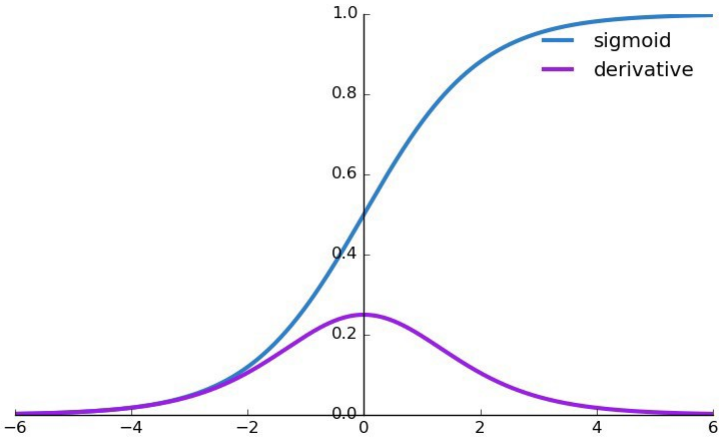
1. Sigmoid

- The Sigmoid Function curve looks like a S-shape.



- The main reason why we use sigmoid function is because it exists between (0 to 1). Therefore, it is especially used for models where we have to predict the probability as an output.Since probability of anything exists only between the range of 0 and 1, sigmoid is the right choice.

Derivative Of Sigmoid Function

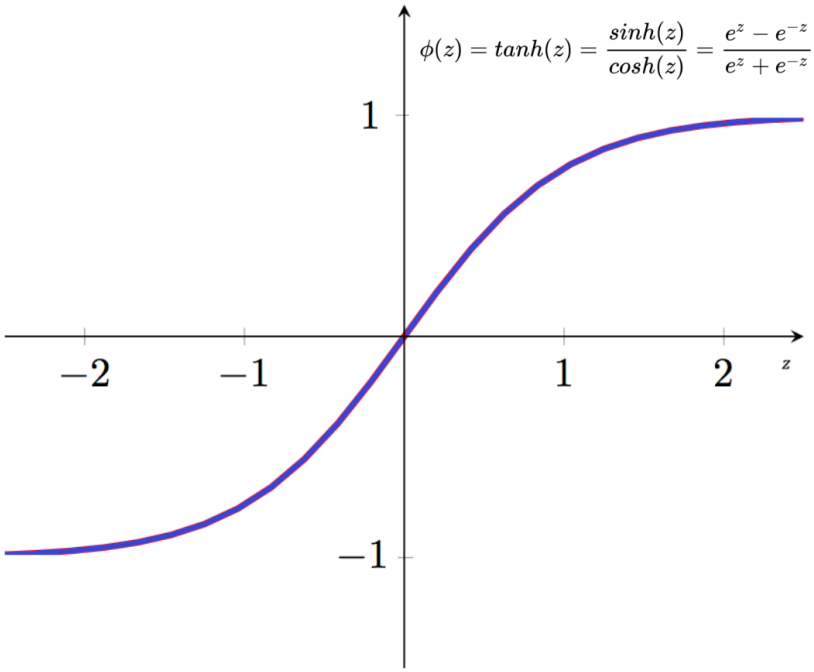


- Sigmoid Function Derivative range from 0 to 0.25

Uses : Usually used in output layer of a binary classification, where result is either 0 or 1, as value for sigmoid function lies between 0 and 1 only so, result can be predicted easily to be 1 if value is greater than 0.5 and 0 otherwise.

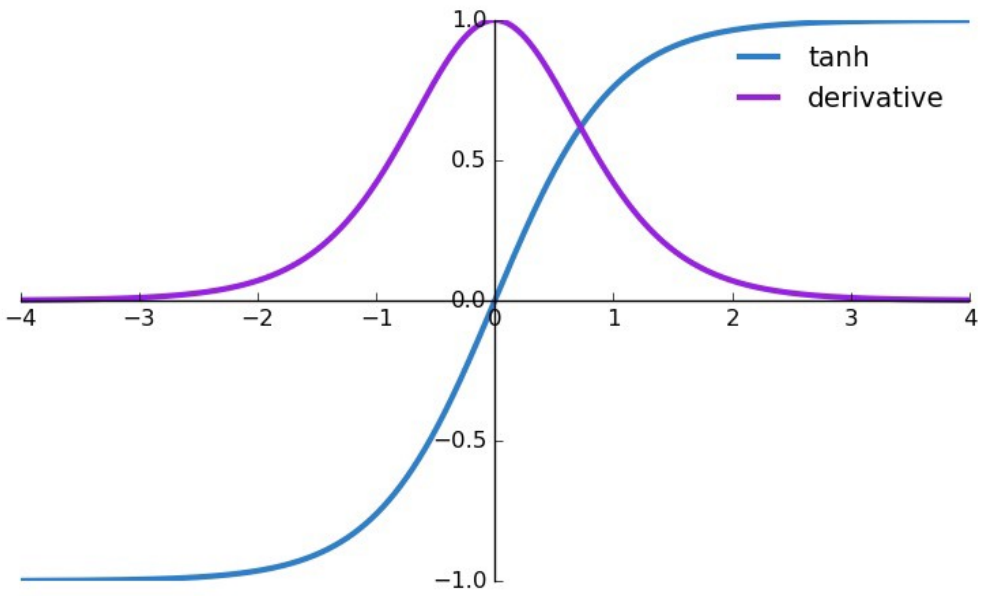
2. Tanh or hyperbolic tangent Activation Function

- tanh is also like logistic sigmoid but better. The range of the tanh function is from (-1 to 1). tanh is also sigmoidal (s - shaped).



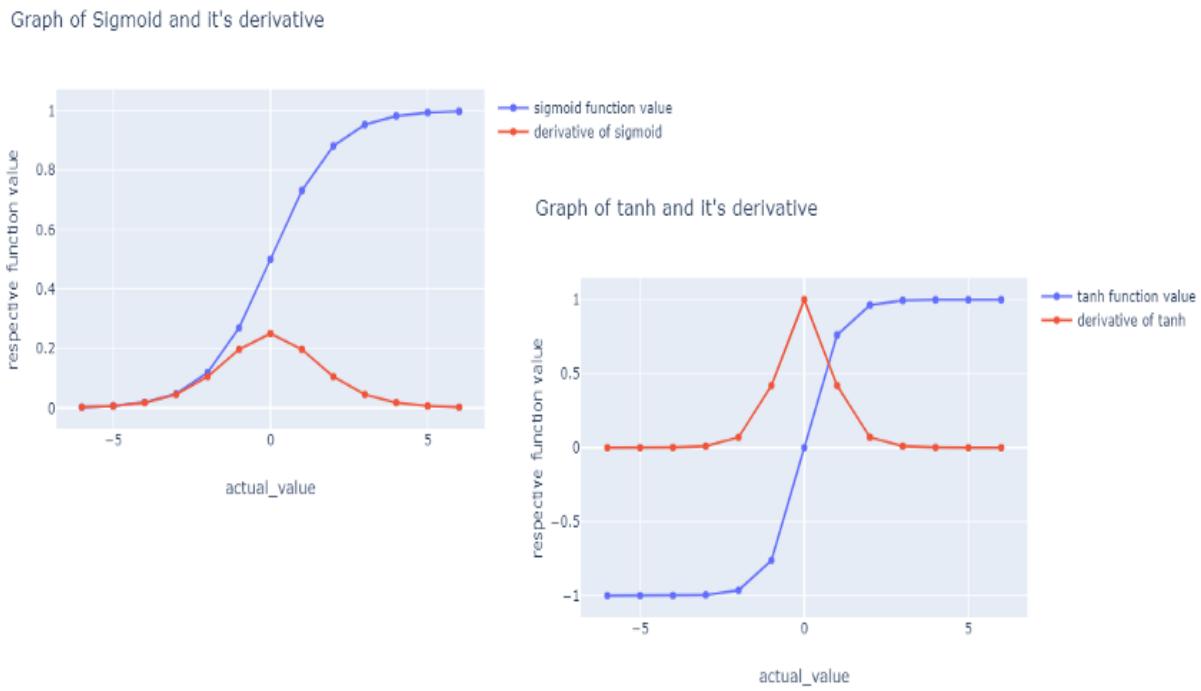
- The advantage is that the negative inputs will be mapped strongly negative and the zero inputs will be mapped near zero in the tanh graph.
- The activation that works almost always better than sigmoid function is Tanh function also knows as Tangent Hyperbolic function. It's actually mathematically shifted version of the sigmoid function. Both are similar and can be derived from each other.

Derivative Of Tanh Function:-



Sigmoid Function Derivative range from 0 to 1

Uses :- Usually used in hidden layers of a neural network as it's values lies between -1 to 1 hence the mean for the hidden layer comes out be 0 or very close to it, hence helps in centering the data by bringing mean close to 0. This makes learning for the next layer much easier.



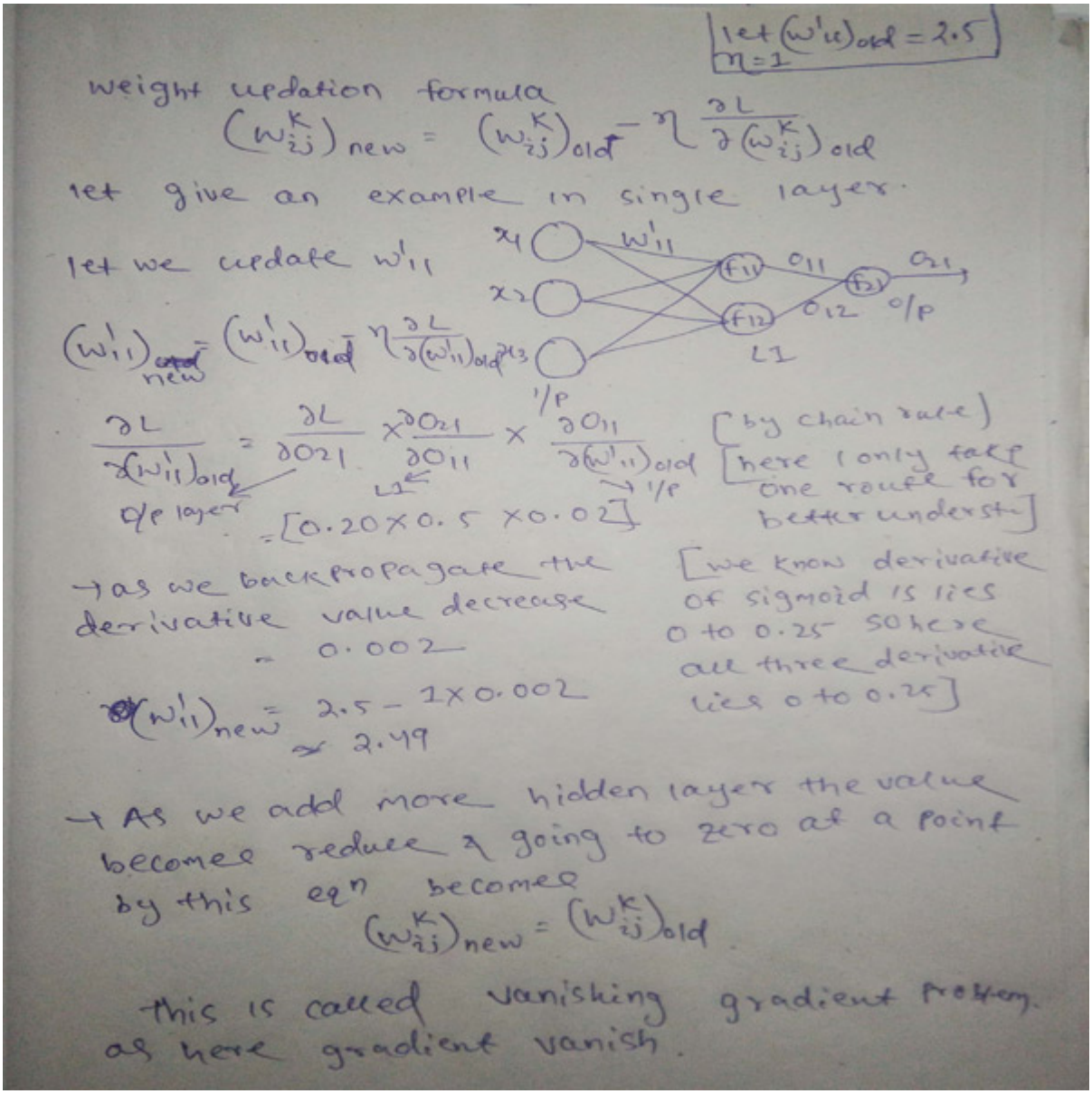
Point :-

1. tanh and logistic sigmoid are the most popular activation functions in 90's but because of their Vanishing gradient problem and sometimes Exploding gradient problem (because of weights), they aren't mostly used now.
2. These days Relu activation function is widely used. Even though, it sometimes gets into vanishing gradient problem, variants of Relu help solving such cases.
3. tanh is preferred to sigmoid for faster convergence BUT again, this might change based on data. Data will also play an important role in deciding which activation function is best to choose.

Vanishing Gradient Problem

- Vanishing gradient problem is a common problem that we face while training deep neural networks.Gradients of neural networks are found during back propagation.
- Generally, adding more hidden layers will make the network able to learn more complex arbitrary functions, and thus do a better job in predicting future outcomes. This is where Deep Learning is making a big difference.

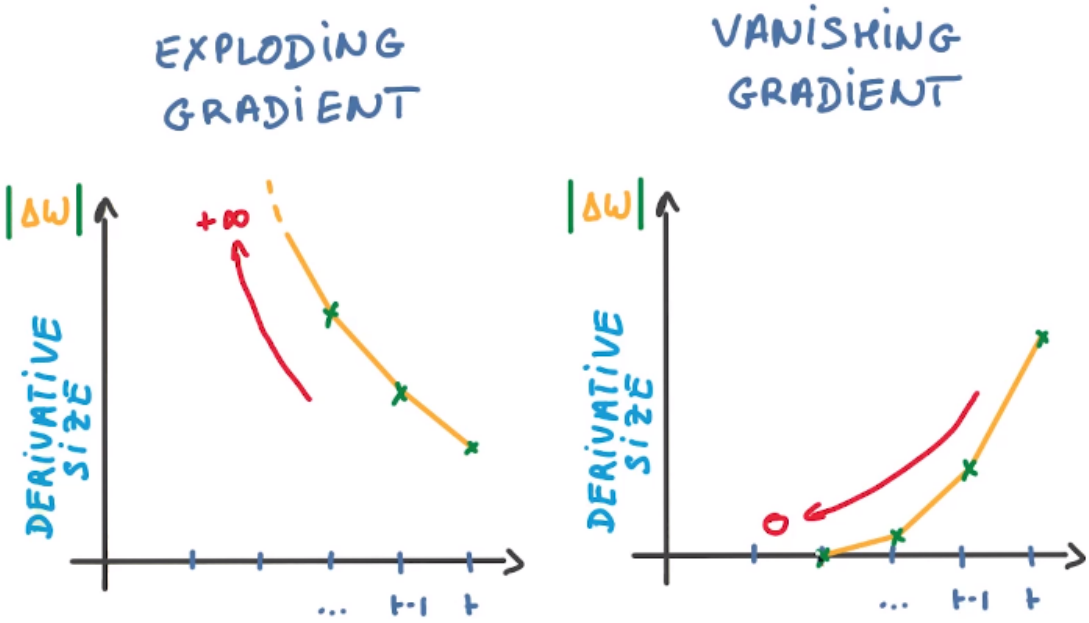
Now during back-propagation i.e moving backward in the Network and calculating gradients, it tends to get smaller and smaller as we keep on moving backward in the Network . Below is Just a simple demonstration of Vanishing Gradient Problem in single layer.



- This Happen because of we use sigmoid and tanh activation function in hidden layer. As sigmoid and tanh deriative 0.25,1 respectively. so by calculating number of hidden layer the derivative becomes 0 so avoid it we use RELU activation function in hidden layer.

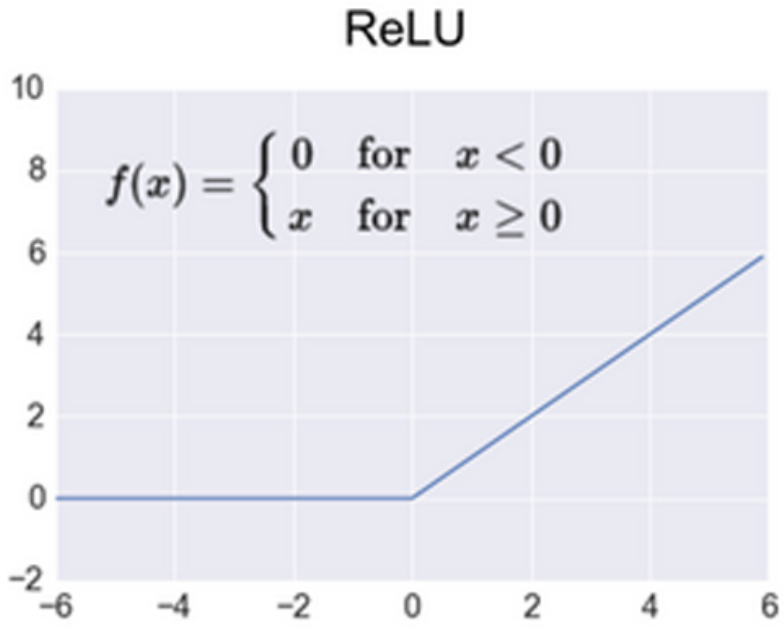
Exploding gradient Problem

- We have discussed about vanishing gradient problem.Now we will get in to exploding gradient problem.Earlier we discussed what happens when our gradient becomes very small.Now we will discuss what will happen if it gets large.
- In deep networks or recurrent neural networks, error gradients can accumulate during an update and result in very large gradients.
- These in turn result in large updates to the network weights, and in turn, an unstable network.The explosion occurs through exponential growth by repeatedly multiplying gradients through the network layers that have values larger than 1.0.This will ultimately led to an total unstable network.



3. Relu

- By Using of Sigmoid And Tanh function there is vanishing gradient problem occure so the convergence rate slow down.
- To overcome slighlty we use Relu Function. it not totally overcome this problem but here the convergence rate is faster than sigmoid and tanh.
- Value Range :- [0, inf)
- Its Nature non-linear, which means we can easily backpropagate the errors and have multiple layers of neurons being activated by the ReLU function.



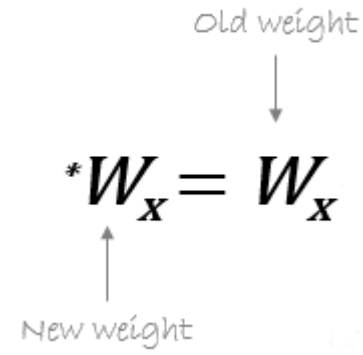
Point:

1. Vanishing Gradient Problem occure due to multiple number of derivative.
2. As sigmoid derivative 0 to 0.25 so by multiple by this sigmoid derivative the result might vanishing gradient as number of hidden layer increase.
3. But in Relu here its derivative 0 to 1 so here no problem of any vanishing gradient problem as its derivative cant be 0.2,0.3 like.
4. But as its derivative can be 0 so here a problem aries that called Dead_Activation

Uses :- ReLu is less computationally expensive than tanh and sigmoid because it involves simpler mathematical operations. At a time only a few neurons are activated making the network sparse making it efficient and easy for computation.

What is Dead_Activation ?

- When Derivative equal to 0 in Relu Then New Weight = Old Weight :



which is not good for any model.Here Weight Cant be update. it occure when value of z is negative.This state called Dead Activation state. to overcome this we use Leaky Relu.

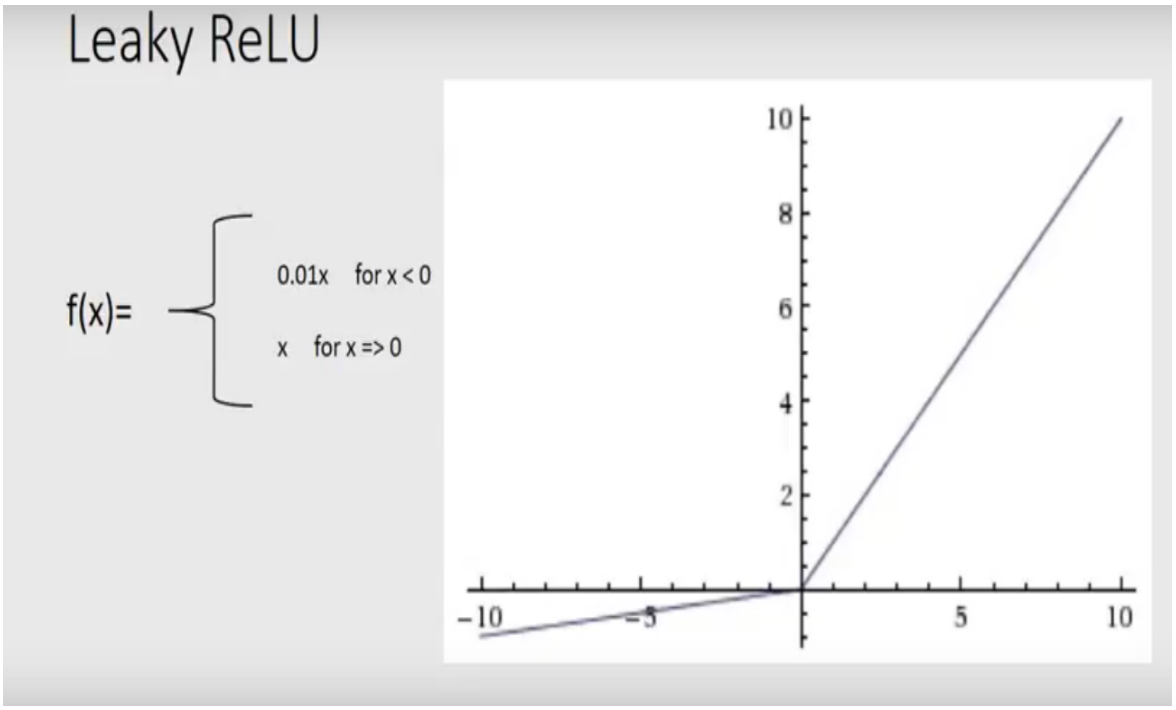
```
Epoch 1/25
32428/32428 [==============================] - 41s 1ms/step - loss: nan - accuracy: 0.5900 - val_loss: nan - val_accuracy: 0.59
44
Epoch 2/25
32428/32428 [==============================] - 39s 1ms/step - loss: nan - accuracy: 0.5907 - val_loss: nan - val_accuracy: 0.59
44
Epoch 3/25
32428/32428 [==============================] - 36s 1ms/step - loss: nan - accuracy: 0.5907 - val_loss: nan - val_accuracy: 0.59
44
Epoch 4/25
32428/32428 [==============================] - 34s 1ms/step - loss: nan - accuracy: 0.5907 - val_loss: nan - val_accuracy: 0.59
44
Epoch 5/25
32428/32428 [==============================] - 36s 1ms/step - loss: nan - accuracy: 0.5907 - val_loss: nan - val_accuracy: 0.59
44
Epoch 6/25
32428/32428 [==============================] - 37s 1ms/step - loss: nan - accuracy: 0.5907 - val_loss: nan - val_accuracy: 0.59
44
Epoch 7/25
32428/32428 [==============================] - 36s 1ms/step - loss: nan - accuracy: 0.5907 - val_loss: nan - val_accuracy: 0.59
44
Epoch 8/25
32428/32428 [==============================] - 38s 1ms/step - loss: nan - accuracy: 0.5907 - val_loss: nan - val_accuracy: 0.59
```

- See here no updation happen the value remain same and ‘nan’ for validation loss is an unexpected very large or very small number. This is dead activation state to overcome this we use leaky relu

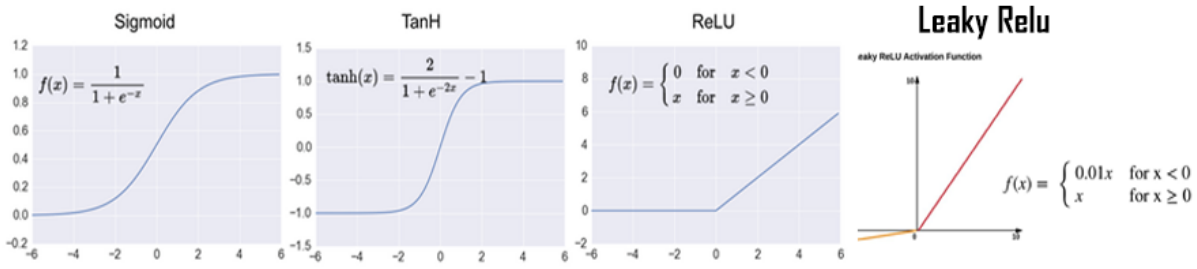
4. Leaky Relu

- Leaky ReLU is an improved version of the ReLU function.
- ReLU function, the gradient is 0 for x < 0(-ve), which made the neurons die for activations in that region.
- Leaky ReLU is defined to address this problem. Instead of defining the Relu function as 0 for x less than 0, we define it as a small linear component of x.
- Leaky ReLUs are one attempt to fix the Dying ReLU problem. Instead of the function being zero when x < 0, a leaky ReLU will instead have a small negative slope (of 0.01, or so). That is, the function computes:

$$f(x) = 1(x < 0)(\alpha x) + 1(x >= 0)(x)$$



Sigmoid,Tanh,Relu,Leaky Relu



Softmax

- Softmax is used as the activation function for multi-class classification tasks, usually the last layer.
- We talked about its role transforming numbers (aka logits) into probabilities that sum to one.
- Let's not forget it is also an activation function which means it helps our model achieve non-linearity. Linear combinations of linear combinations will always be linear but adding activation function helps gives our model ability to handle non-linear data.
- Output of other activation functions such as sigmoid does not necessarily sum to one. Having outputs summing to one makes softmax function great for probability analysis.
- The function is great for classification problems, especially if you're dealing with multi-class classification problems, as it will report back the "confidence score" for each class. Since we're dealing with probabilities here, the scores returned by the softmax function will add up to 1.

Mathematical representation

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

Where:

σ = softmax

$\vec{z}$=input vector

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

K = number of classes in the multi-class classifier

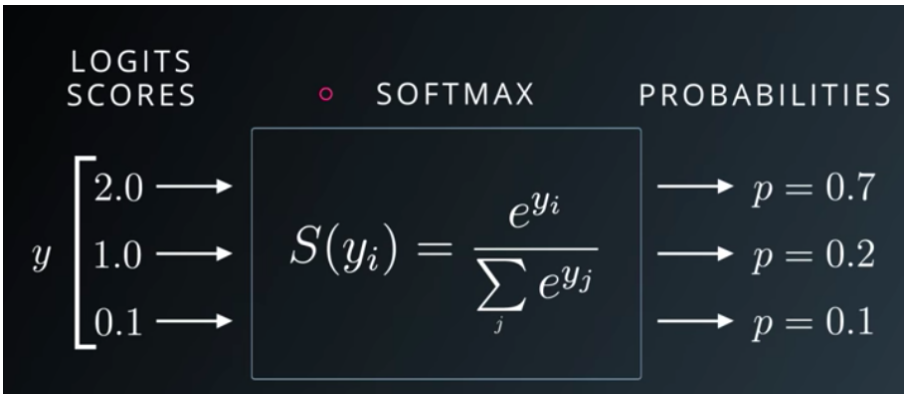
e^{z_j} =standard exponential function for output vector

- It states that we need to apply a standard exponential function to each element of the output layer, and then normalize these values by dividing by the sum of all the exponentials. Doing so ensures the sum of all exponentiated values adds up to 1.

Here are the steps For Softmax:

1. Exponentiate every element of the output layer and sum the results
2. Take each element of the output layer, exponentiate it and divide by the sum obtained in step 1

Example Implementation



To start, let's declare an array which imitates the output layer of a neural network:

In [1]:
```
## as we know softmax used in output layer so here i take a outputlayer value
import numpy as np
output_layer = np.array([2.0,1.0,0.1])
output_layer
```

Out[1]:
```
array([2. , 1. , 0.1])
```

By step 1 we need to exponentiate each of the elements of the output layer:

In [2]:
```
exponentiated = np.exp(output_layer)
exponentiated
```

Out[2]:
```
array([7.3890561 , 2.71828183, 1.10517092])
```

According to step 2 calculate probabilities! We can use Numpy to divide each element by exponentiated sum and store results in another array

In [3]:
```
probabilities = exponentiated / np.sum(exponentiated)
print(probabilities)
print(sum(probabilities))
print(np.argmax(probabilities))
```

```
[0.65900114 0.24243297 0.09856589]
1.0
0
```

Here see the output are formed. If we sum three then we get probability 1. after this you use argmax function which return highest value index number. See here return 0 as 0 index have 0.65 value which is highest among three value.

When you use softmax in your dataset you should use argmax function to predict output.

- From a probabilistic perspective, if the argmax() function returns 1 in the large value, it returns 0 for the other two array indexes.here it giving full weight to index 0 and no weight to index 1

and index 2 for the largest value in the list [0.65,0.24,0.09].

In the Keras deep learning library with a three-class classification task, use of softmax in the output layer may look as follows:

model.add(Dense(no.of output layer, activation='softmax'))

- It apply when you have multiclass problem aries

The Differences between Sigmoid and Softmax Activation Functions

In [4]:
```
import numpy as np
### sigmoid function
def sigmoid(x):
    s = 1 / (1 + np.exp(-x))
    return s

## softmax function
def softmax(x):
    exponentiated = np.exp(x)
    probabilities = exponentiated / np.sum(exponentiated)
    return probabilities
```

In [5]:
```
x = np.array([-0.5, 1.2, -0.1, 2.4])
a = sigmoid(x)
print("--- Sigmoid---")
print(a.round(2))
print(sum(a).round(2))

print(50*"*")

output_layer = np.array([-0.5, 1.2, -0.1, 2.4])
b = softmax(output_layer)
print("---Softmax---")
print(b.round(2))
print(sum(b))
```

```
--- Sigmoid---
[0.38 0.77 0.48 0.92]
2.54
**************************************************
---Softmax---
[0.04 0.21 0.06 0.7 ]
1.0
```

The key takeaway from this example is:

- **Sigmoid:** probabilities produced by a Sigmoid are independent. Furthermore, they are not constrained to sum to one: 0.38 + 0.77 + 0.48 + 0.92 = 2.54. The reason for this is because the Sigmoid looks at each raw output value separately.
- **Softmax:** the outputs are interrelated. The Softmax probabilities will always sum to one by design: 0.04 + 0.21 + 0.06 + 0.7 = 1.00. In this case, if we want to increase the likelihood of one class, the other has to decrease by an equal amount.

Summary..

Characteristics of a Sigmoid Activation Function:

1. Used for Binary Classification in the Logistic Regression model
2. The probabilities sum does not need to be 1
3. Used as an Activation Function while building a Neural Network

Characteristics of a Softmax Activation Function

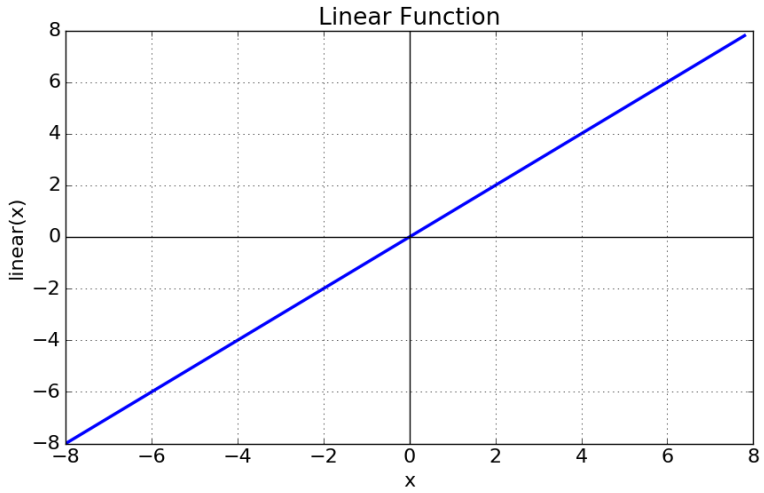
1. Used for Multi-classification in the Logistics Regression model
2. The probabilities sum will be 1
3. Used in the different layers of Neural Networks

Activation Function For Regression Problem

- Linear Activation Function:-

Equation : f(x) = x

Range : (-infinity to infinity)



- No matter how many layers we have, if all are linear in nature, the final activation function of last layer is nothing but just a linear function of the input of first layer.
- Linear activation function is used at just one place i.e. output layer.
- If we will differentiate linear function to bring non-linearity, result will no more depend on input “x” and function will become constant, it won’t introduce any ground-breaking behavior to our algorithm.

algorithm.

Uses: Calculation of price of a house is a regression problem. House price may have any big/small value, so we can apply linear activation at output layer. Even in this case neural net must have any non-linear function at hidden layers.

Loss Functions

- The loss function is the function that computes the distance between the current output of the algorithm and the expected output. It's a method to evaluate how your algorithm models the data. It can be categorized into two groups. One for classification (discrete values, 0,1,2…) and the other for regression (continuous values).

TYPES OF LOSS FUNCTION:

1. Regression Loss Functions

```
Mean Absolute Error
Mean Squared Error
Root Mean Square error (RMSE)
```

2. Binary Classification Loss Functions

```
Binary Cross-Entropy
```

3. Multi-Class Classification Loss Functions

```
Multi-Class Cross-Entropy Loss
Sparse Multiclass Cross-Entropy Loss
```

Regression Losses

We know all of this regression loss function but here i discuss a brief

Mean Absolute Error

- Regression metric which measures the average magnitude of errors in a group of predictions, without considering their directions. In other words, it's a mean of absolute differences among predictions and expected results where all individual deviations have even importance.

$$\mathrm{MAE} = \underbrace{\frac{1}{n}\sum_{i=1}^{n}}_{\text{test set}} \underbrace{|y_i}_{\text{predicted vaue}} - \underbrace{\hat{y}_i|}_{\text{actual value}}$$

where:

i — index of sample,

ŷ — predicted value,

y — expected value,

m — number of samples in dataset.

Sometimes it is possible to see the form of formula with swapped predicted value and expected value, but it works the same.

Mean Squared Error

- One of the most commonly used and firstly explained regression metrics. Average squared difference between the predictions and expected results. In other words, an alteration of MAE where instead of taking the absolute value of differences, they are squared.
- In MAE, the partial error values were equal to the distances between points in the coordinate system. Regarding MSE, each partial error is equivalent to the area of the square created out of the geometrical distance between the measured points. All region areas are summed up and averaged.

$$\underbrace{\mathrm{MSE} = \frac{1}{n}\sum_{i=1}^{n}}_{\text{test set}} \underbrace{(y_i}_{\text{predicted vaue}} - \underbrace{\hat{y}_i}_{\text{actual value}})^2$$

Where

i — index of sample,

ŷ — predicted value,

y — expected value,

m — number of samples in dataset.

Root Mean Square error (RMSE)

- Root Mean Square error is the extension of MSE — measured as the average of square root of sum of squared differences between predictions and actual observations.

$$RMSE = \sqrt{\frac{\sum_{i=1}^{N}(Predicted_i - Actual_i)^2}{N}}$$

Classification Losses

Binary Classification Loss Functions

Binary Cross Entropy

- Also called Sigmoid Cross-Entropy loss. It is a Sigmoid activation plus a Cross-Entropy loss. Unlike Softmax loss it is independent for each vector component (class), meaning that the loss computed for every CNN output vector component is not affected by other component values.
- Binary cross entropy measures how far away from the true value (which is either 0 or 1) the prediction is for each of the classes and then averages these class-wise errors to obtain the final loss.
- We can define cross entropy as the difference between two probability distributions p and q, where p is our true output and q is our estimate of this true output.
- it Only use for binary classification problem

$$H(x) = \sum_{i=1}^{N} \underbrace{p(x)}_{\text{true label}} \log \underbrace{q(x)}_{\text{estimate}}$$

Multi-Class Classification Loss Functions

Categorical cross-entropy

- Used binary and multiclass problem, the label needs to be encoded as categorical, one-hot encoding representation (for 3 classes: [0, 1, 0], [1,0,0]…)
- It is a loss function that is used for single label categorization. This is when only one category is applicable for each data point. In other words, an example can belong to one class only.

$$L(y, \hat{y}) = -\sum_{j=0}^{M} \sum_{i=0}^{N} (y_{ij} * log(\hat{y}_{ij}))$$

- Use categorical crossentropy in classification problems where only one result can be correct.
- Example: In the MNIST problem where you have images of the numbers 0,1, 2, 3, 4, 5, 6, 7, 8, and 9. Categorical crossentropy gives the probability that an image of a number is, for example, a 4 or a 9.

- Categorical cross-entropy will compare the distribution of the predictions (the activations in the output layer, one for each class) with the true distribution, where the probability of the true class is set to 1 and 0 for the other classes. To put it in a different way, the true class is represented as a one-hot encoded vector, and the closer the model's outputs are to that vector, the lower the loss.

Sparse Categorical cross-entropy

- Used binary and multiclass problem (the label is an integer — 0 or 1 or … n, depends on the number of labels)

If your targets are **one-hot encoded**, use `categorical_crossentropy`.

- Examples of one-hot encodings:
  - `[1,0,0]`
  - `[0,1,0]`
  - `[0,0,1]`

But if your targets are **integers**, use `sparse_categorical_crossentropy`.

- Examples of integer encodings (*for the sake of completion*):
  - `1`
  - `2`
  - `3`

All Losses : [https://keras.io/api/losses/ (https://keras.io/api/losses/)](https://keras.io/api/losses/)

Summary

There are three kinds of classification tasks:

1. Binary classification: two exclusive classes
2. Multi-class classification: more than two exclusive classes
3. Multi-label classification: just non-exclusive classes

Here, we can say

1. In the case of (1), you need to use binary cross entropy.
2. In the case of (2), you need to use categorical cross entropy.
3. In the case of (3), you need to use binary cross entropy.

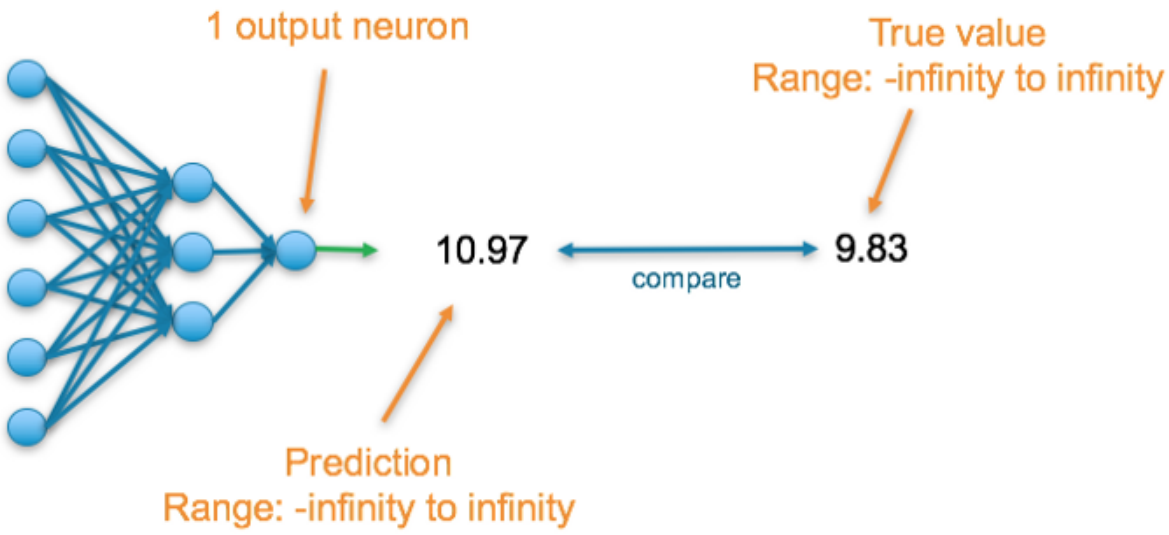
Which Loss and Activation Functions should I use?

- The motive of the blog is to give you some ideas on the usage of “Activation Function” & “Loss function” in different scenarios.
- Choosing an activation function and loss function is directly dependent upon the output you want to predict. There are different cases and different outputs of a predictive model. Before I introduce you to such cases let see an introduction to the activation function and loss function.
- The activation function activates the neuron that is required for the desired output, converts linear input to non-linear output. If you are not aware of the different activation functions I would recommend you visit my activation pdf to get an in-depth explanation of different activation functions click here : https://github.com/pratyusa98/ML_Algo_pdf/tree/main/01_Deep_Learning_PDF (https://github.com/pratyusa98/ML_Algo_pdf/tree/main/01_Deep_Learning_PDF).
- Loss function helps you figure out the performance of your model in prediction, how good the model is able to generalize. It computes the error for every training. You can read more about loss functions and how to reduce the loss https://github.com/pratyusa98/ML_Algo_pdf/tree/main/01_Deep_Learning_PDF (https://github.com/pratyusa98/ML_Algo_pdf/tree/main/01_Deep_Learning_PDF)..

Let’s see the different cases:

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

- Ex:- Consider predicting the prices of houses provided with different features of the house. A neural network structure where the final layer or the output later 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 to be used in Output layer such cases,

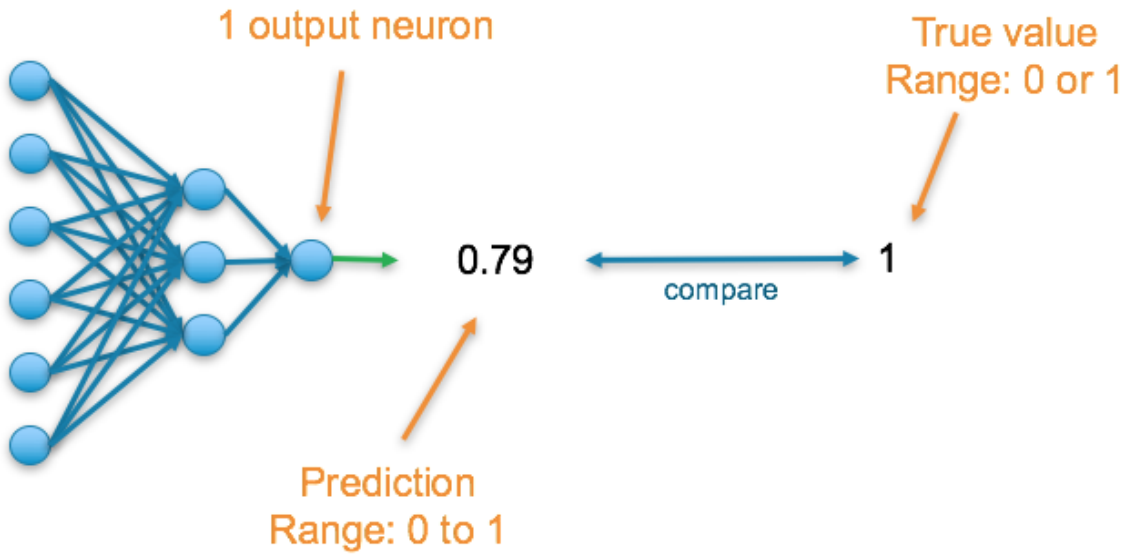
```
        * Linear Activation - it gives output in a numeric form th
at is the demand for this case. Or
        * ReLU Activation - This activation function gives you pos
itive numeric outputs as a result.
```

- Loss function to be used in such cases,

```
        * Mean Squared Error (MSE) - This loss function is respons
ible to compute the average squared difference    between the true v
alues and the predicted values.
```

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

- Ex:- Consider a case where the aim is to predict whether a loan applicant will default or not. In these types of cases, the output layer consists of only one neuron that is responsible to result in a value that is between 0 and 1 that can be also called probabilistic scores.
- For computing the accuracy of the prediction, it is again compared with the true labels. The true value is 1 if the data belongs to that class or else it is 0.



- Activation Function to be used in Output layer such cases,

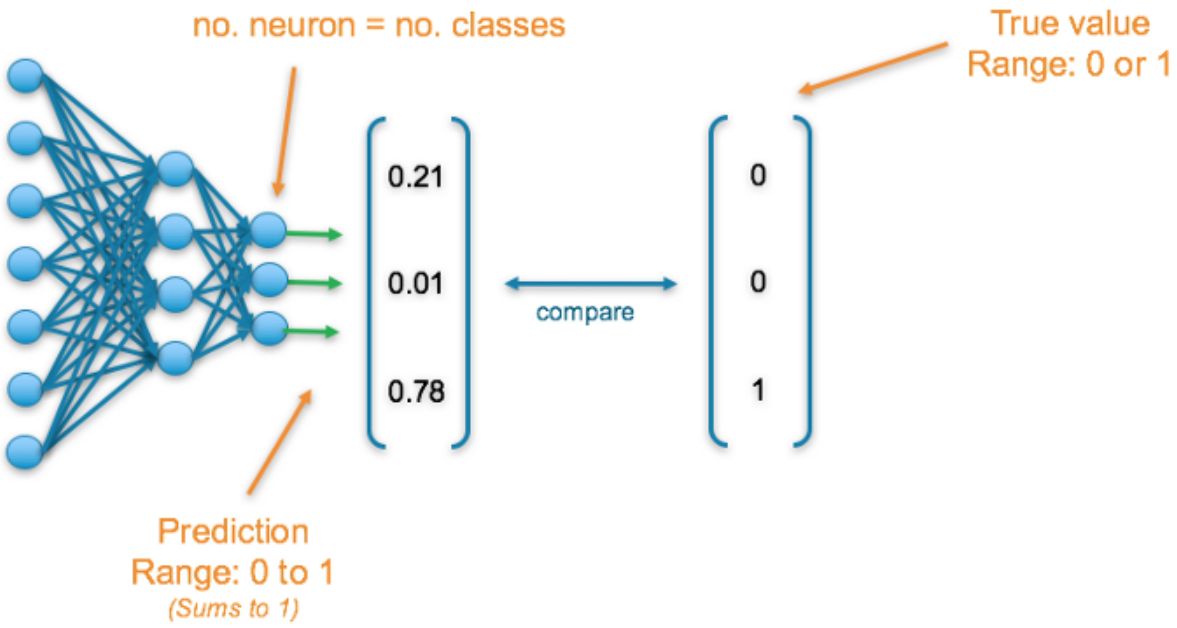
```
        * Sigmoid Activation -  This activation function gives
 the output as 0 and 1.
```

- Loss function to be used in such cases,

```
        * Binary Cross Entropy - The difference between the tw
o probability distributions is given by binary  cross-entropy. (p,1-
p) is the model distribution predicted by the model, to compare it w
ith true distribution, the   binary cross-entropy is used.
```

CASE 3: Predicting a single class from many classes

- Ex:- Consider a case where you are predicting the name of the fruit amongst 5 different fruits. In the case, the output layer will consist of only one neuron for every class and it will revert a value between 0 and 1, the output is the probability distribution that results in 1 when all are added.
- Each output is checked with its respective true value to get the accuracy. These values are one-hot-encoded which means if will be 1 for the correct class or else for others it would be zero.



- Activation Function to be used in Output layer such cases,

```
                * Softmax Activation -  This activation function gives
    the output between 0 and 1 that are the probability scores which if
     added gives the result as 1.
```

- Loss function to be used in such cases,

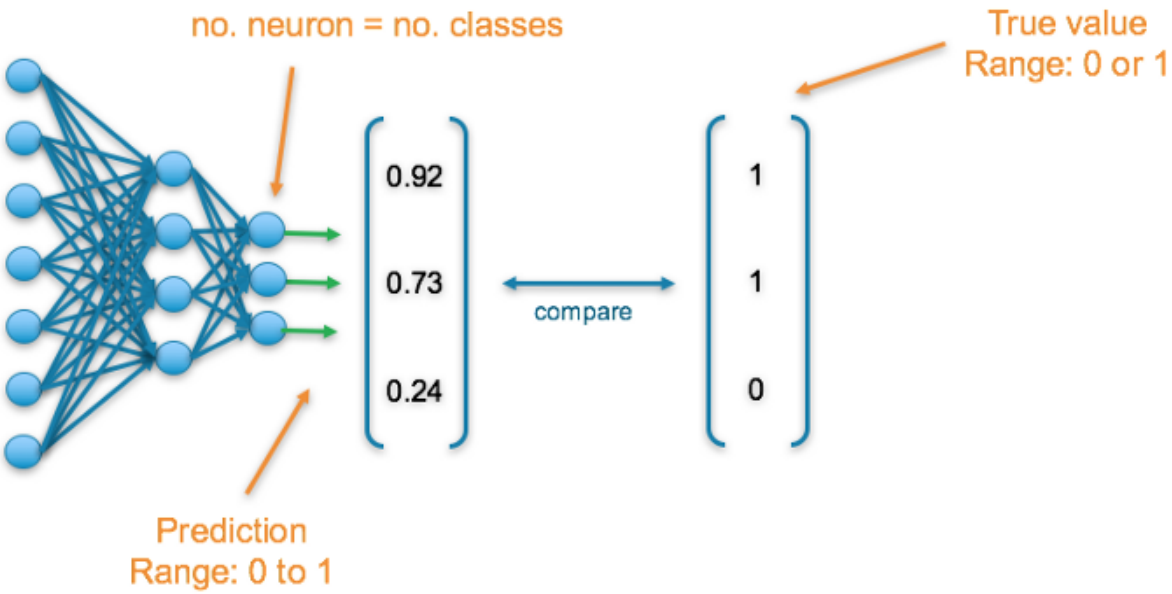
```
                * Cross-Entropy - It computes the difference between t
    wo probability distributions.
                * (p1,p2,p3) is the model distribution that is predict
    ed by the model where p1+p2+p3=1. This is compared with the true dis
    tribution using cross-entropy.
```

CASE 4: Predicting multiple labels from multiple class

- Ex:- Consider the case of predicting different objects in an image having multiple objects. This is termed as multiclass classification. In these types of cases, the output layer consists of only one neuron that is responsible to result in a value that is between 0 and 1 that can be also

called probabilistic scores.

- For computing the accuracy of the prediction, it is again compared with the true labels. The true value is 1 if the data belongs to that class or else it is 0.



- Activation Function to be used in Output layer such cases,

```
                * Sigmoid Activation -  This activation function giv
es the output as 0 and 1.
```

- Loss function to be used in such cases,

```
                * Binary Cross Entropy - The difference between the
 two probability distributions is given by binary cross-entropy. (p,
1-p) is the model distribution predicted by the model, to compare it
with true distribution, the binary cross-entropy is used.
```

All Losses : https://keras.io/api/losses/ (https://keras.io/api/losses/)
All Activation : https://keras.io/api/layers/activations/ (https://keras.io/api/layers/activations/)

Summary

- This activation use only output layer and in hidden layer you can use Relu or Leaky Relu.
- The following table summarizes the above information to allow you to quickly find the final layer activation function and loss function that is appropriate to your use-case

Problem Type	Output Type	Final Activation Function	Loss Function
Regression	Numerical value	Linear	Mean Squared Error (MSE)
Classification	Binary outcome	Sigmoid	Binary Cross Entropy
Classification	Single label, multiple classes	Softmax	Cross Entropy
Classification	Multiple labels, multiple classes	Sigmoid	Binary Cross Entropy

Weight Initialization

- The weight initialization technique you choose for your neural network can determine how quickly the network converges or whether it converges at all. Although the initial values of these weights are just one parameter among many to tune, they are incredibly important. Their distribution affects the gradients and, therefore, the effectiveness of training.

Why is weight initialization important? ¶

- Improperly initialized weights can negatively affect the training process by contributing to the vanishing or exploding gradient problem.
- With the vanishing gradient problem, the weight update is minor and results in slower convergence — this makes the optimization of the loss function slow and in a worst case scenario, may stop the network from converging altogether.
- Conversely, initializing with weights that are too large may result in exploding gradient values during forward propagation or back-propagation.

1. Zero initialization :

- If all the weights are initialized with 0, the derivative with respect to loss function is the same for every weight(w), thus all weights have the same value in subsequent iterations.
- This makes hidden units symmetric and continues for all the n iterations i.e. setting weights to 0 does not make it better than a linear model.
- An important thing to keep in mind is that biases have no effect what so ever when initialized with 0.
- It also gives problems like vanishing gradient problem.

2. initialization With -ve Number :

- If all weigth can be negative then it affect Relu Activation Function.As in -ve Relu comes under dead activation problem. so we cant use this technique.
- Weights can’t be too high as gives problems like exploding Gradient problem(weights of the model explode to infinity), which means that a large space is made available to search for global minima hence convergence becomes slow.

To prevent the gradients of the network’s activations from vanishing or exploding, we need to have following rules:

```
        1. The mean of the activations should be zero.
        2. The variance of the activations should stay the same across e
very layer.
```

Idea 1 : Normal or Naïve Initialization:

- In normal distribution weights can be a part of normal or gaussian distribution with mean as zero and a unit standard deviation.

$$W \approx N(0, \sigma) \quad \mu = 0 \qquad \sigma = \text{small number}$$

- Random initialization is done so that convergence is not to a false minima.
- In Keras it can be simply written as hyperparameter as - kernel_initializer='random_normal'

Idea 2: Uniform Initialization:

- In uniform initialization of weights , weights belong to a uniform distribution in range a,b with values of a and b as below:

$$W \approx U(a,b) \qquad a = \frac{-1}{\sqrt{f\_in}} \;, \; b = \frac{1}{\sqrt{f\_in}}$$

- Whenever **sigmoid** activation function is used as , Uniform works well.
- In Keras it can be simply written as hyperparameter as - kernel_initializer='random_uniform'

Idea 3: Xavier/ Glorot Weight Initialization:

- The variance of weights in the case normal distribution was not taken care of which resulted in too large or too small activation values which again led to exploding gradient and vanishing gradient problems respectively, when back propagation was done.
- In order to overcome this problem Xavier Initialization was introduced. It keeps the variance the same across every layer. We will assume that our layer's activations are normally distributed around zero.
- Glorot or Xavier had a belief that if they maintain variance of activations in all the layers going forward and backward convergence will be fast as compared to using standard initialization where gap was larger.
- It have Two Varient

```
a. Normal Distribution - kernel_initializer='glorot_normal'
b. Uniform Distribution - kernel_initializer='glorot_uniform'
```

Point : Works well with **tanh , sigmoid** activation functions.

a. Normal Distribution:

- In Normal Distribution, weights belong to normal distribution where mean is zero and standard deviation is as below:

$$W \approx N(\mu, \sigma)$$

$$\mu = 0 \qquad \sigma = \sqrt{\frac{2}{f\_in + f\_out}}$$

b. Uniform Distribution:

- Uniform Distribution , weights belong to uniform distribution in range of a and b defined as below:

$$W \approx U(a,b)$$

$$a = -\sqrt{\frac{6}{f\_in + f\_out}}, \qquad b = \sqrt{\frac{6}{f\_in + f\_out}}$$

Idea 4: He-Initialization:

- When using activation functions that were zero centered and have output range between-1,1 for activation functions like tanh and softsign, activation outputs were having mean of 0 and standard deviation around 1 average wise.
- But if ReLu is used instead of tanh, it was observed that on average it has standard deviation very close to square root of 2 divided by input connections.
  - It have Two Varient

```
a. Normal Distribution - kernel_initializer='he_normal'
b. Uniform Distribution - kernel_initializer='he_uniform'
```

Point : Works well with **Relu And Leaky Relu** activation functions.

a. Normal Distribution:

- In He-Normal initialization method, weights belong to normal distribution where mean is zero and standard deviation is as below:

$$W \approx N(\mu, \sigma)$$

$$\mu = 0 \qquad \sigma = \sqrt{\frac{2}{f\_in}}$$

b. Uniform Initialization :

- In He Uniform Initialization weights belong to uniform distribution in range as shown below:

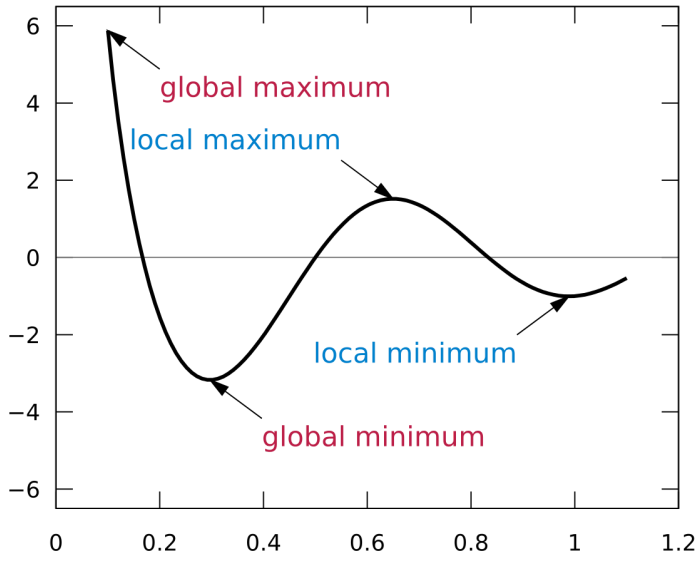
$$W \approx U(a,b) \qquad a = -\sqrt{\frac{6}{f\_in}} \quad , \quad b = \sqrt{\frac{6}{f\_in}}$$

Optimization Techniques

- Optimization algorithms are responsible for reducing losses and provide most accurate results possible.
- The weight is initialized using some initialization strategies and is updated with each epoch according to the equation. The best results are achieved using some optimization strategies or algorithms called Optimizer.

- Some of the techniques that we will be discussing in this article is-

```
* Gradient Descent
* Stochastic Gradient Descent (SGD)
* Mini-Batch Stochastic Gradient Descent (MB – SGD)
* SGD with Momentum
* Nesterov Accelerated Gradient (NAG)
* Adaptive Gradient (AdaGrad)
* AdaDelta
* RMSProp
* Adam
```



1. Gradient Descent or Batch Gradient Descent

- A Gradient Descent is an iterative algorithm, that starts from a random point on the function and traverses down its slope in steps until it reaches lowest point (global minima) of that function.
- This algorithm is apt for cases where optimal points cannot be found by equating the slope of the function to 0. For the function to reach minimum value, the weights should be altered.
- With the help of back propagation, loss is transferred from one layer to another and “weights” parameter are also modified depending on loss so that loss can be minimized.

Point :

1. Use all training Sample for a forward pass and adjust the weights.
2. This makes it computationally intensive. 3. Another drawback is there are chances the iteration values may get stuck at local minima or saddle point and never converge to minima. To obtain the best solution, the must reach global minima. 4. Good For Small training data.

Cost function: $\theta=\theta-\alpha\cdot\nabla J(\theta)$

Advantages:

- Easy computation.
- Easy to implement.
- Easy to understand.

Disadvantages:

- May trap at local minima.
- Weights are changed after calculating gradient on the whole dataset. So, if the dataset is too large than this may take years to converge to the minima.
- Requires large memory to calculate gradient on the whole dataset.

2. Stochastic Gradient Descent

- Stochastic Gradient Descent is an extension of Gradient Descent, where it overcomes some of the disadvantages of Gradient Descent algorithm.
- SGD tries to overcome the disadvantage of computationally intensive by computing the derivative of one point at a time.
- Due to this fact, SGD takes more number of iterations compared to GD to reach minimum and also contains some noise when compared to Gradient Descent.
- As SGD computes derivatives of only 1 point at a time, the time taken to complete one epoch is large compared to Gradient Descent algorithm.

Point :

1. Use One (Randomly Picked) Sample for a forward pass and adjust the weights.
2. Good when training set is very big and we dont want too much computation.

cost function $\theta=\theta-\alpha\cdot\nabla J(\theta;x(i);y(i))$, where {x(i) ,y(i)} are the training examples.

Advantages:

- Frequent updates of model parameters hence, converges in less time.
- Requires less memory as no need to store values of loss functions.
- May get new minima’s.

Disadvantages:

- High variance(noisey) in model parameters.

- May shoot even after achieving global minima.
- To get the same convergence as gradient descent needs to slowly reduce the value of learning rate.

3. Mini Batch — Stochastic Gradient Descent

- MB-SGD is an extension of SGD algorithm. It overcomes the time-consuming complexity of SGD by taking a batch of points / subset of points from dataset to compute derivative.
- It's best among all the variations of gradient descent algorithms. It is an improvement on both SGD and standard gradient descent. It updates the model parameters after every batch. So, the dataset is divided into various batches and after every batch, the parameters are updated.
- This is a mixture of both stochastic and batch gradient descent.
- The training set is divided into multiple groups called batches. Each batch has a number of training samples in it.
- At a time a single batch is passed through the network which computes the loss of every sample in the batch and uses their average to update the parameters of the neural network.
- For example, say the training set has 100 training examples which is divided into 5 batches with each batch containing 20 training examples. This means that the equation in figure2 will be iterated over 5 times (number of batches).

Point:

1. Use a Batch Of (Randomly Picked) Sample for a forward pass and adjust the weights.
2. It is observed that the derivative of loss function of MB-SGD is similar to the loss function of GD after some iterations. But the number iterations to achieve minima in MB-SGD is large compared to GD and is computationally expensive. The update of weights in much noisier because the derivative is not always towards minima.

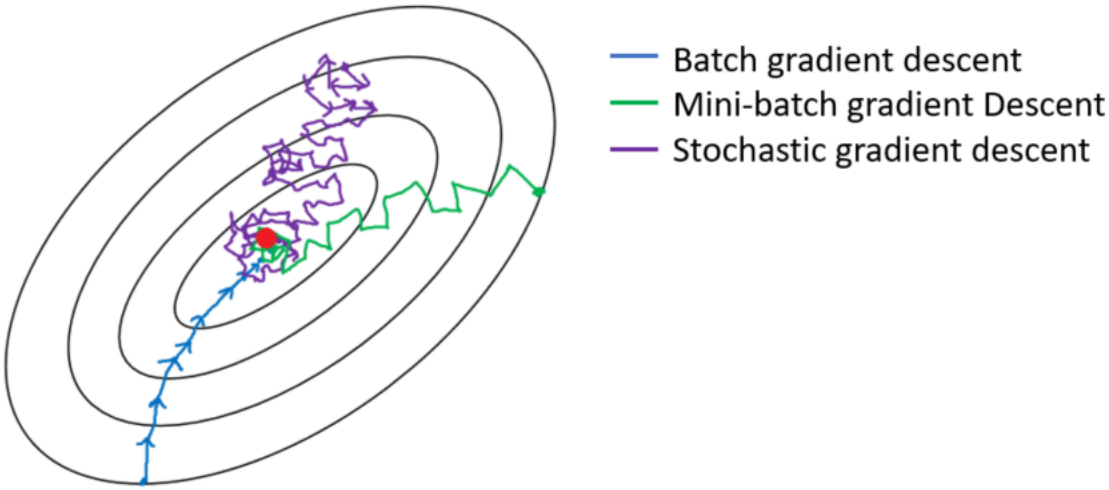
$\theta=\theta-\alpha\cdot\nabla J(\theta; B(i))$, where $\{B(i)\}$ are the batches of training examples.

Advantages:

- Frequently updates the model parameters and also has less variance.
- Requires medium amount of memory.
- Easily fits in the memory
- It is computationally efficient
- Benefit from vectorization
- If stuck in local minimums, some noisy steps can lead the way out of them
- Average of the training samples produces stable error gradients and convergence

!!!! This ensures the following advantages of both stochastic and batch gradient descent are used due to which Mini Batch Gradient Descent is most commonly used in practice.

See How in this above three convergence Occure towards minima point



Here We see in SGD Due to frequent updates the steps taken towards the minima are very noisy. This can often lead the gradient descent into other directions. Also, due to noisy steps it may take longer to achieve convergence to the minima of the loss function. to reduce this we can use SGD with Momentum.

4. SGD with Momentum

- Momentum was invented for reducing high variance in SGD and softens the convergence.
- It accelerates the convergence towards the relevant direction and reduces the fluctuation to the irrelevant direction. One more hyperparameter is used in this method known as momentum symbolized by 'γ'(gamma).
- It is an adaptive optimization algorithm which exponentially uses weighted average gradients over previous iterations to stabilize the convergence, resulting in quicker optimization.
- This is done by adding a fraction (gamma) to the previous iteration values.
- Essentially the momentum term increase when the gradient points are in the same directions and reduce when gradients fluctuate. As a result, the value of loss function converges faster than expected.

$$v_t = \gamma v_{t-1} + \eta \nabla J(w_t)$$

$$w_t = w_{t-1} - v_t$$

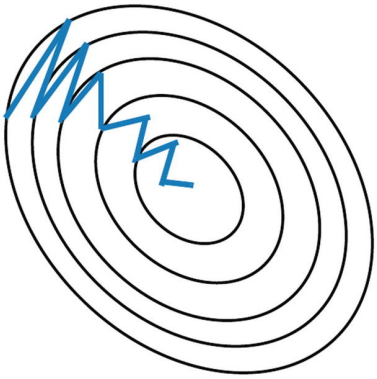
Advantages:

- Reduces the oscillations and high variance of the parameters.
- Converges faster than gradient descent.

Disadvantages:

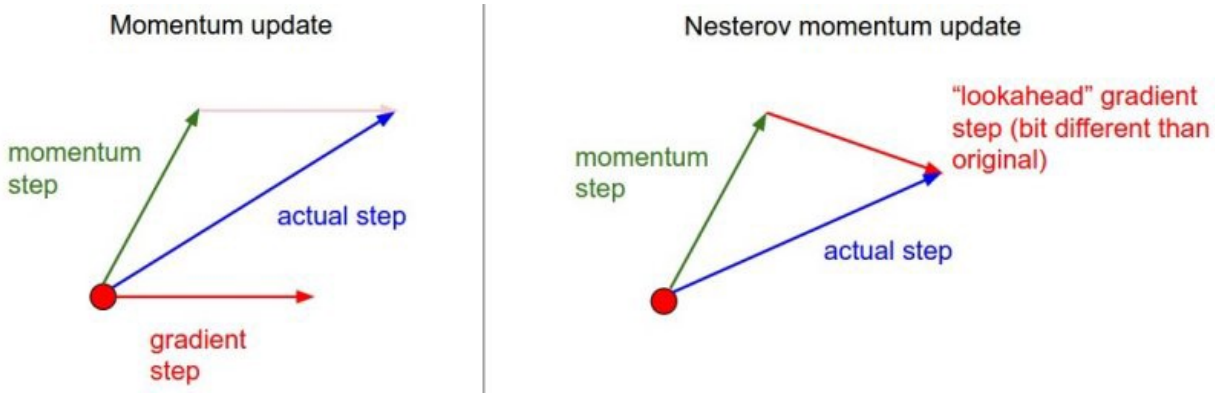
- One more hyper-parameter is added which needs to be selected manually and accurately.

Stochastic Gradient Descent **withhout** Momentum



Stochastic Gradient Descent **with** Momentum

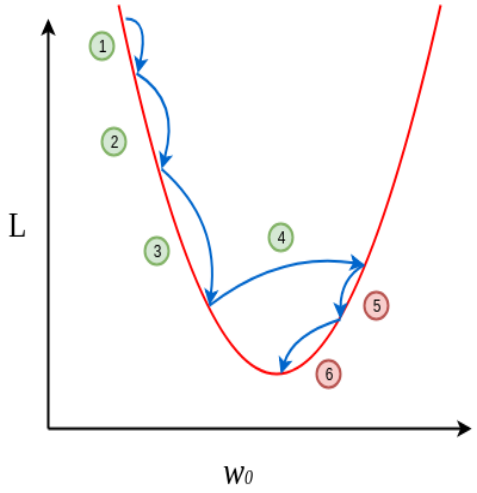
5. Nesterov accelerated gradient(NAG)



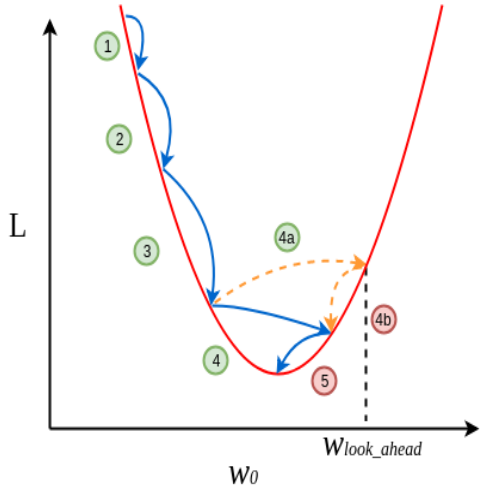
- Momentum may be a good method but if the momentum is too high the algorithm may miss the local minima and may continue to rise up. So, to resolve this issue the NAG algorithm was developed.
- Nesterov accelerated gradient (NAG) is a way to give momentum more precision.
- The idea of the NAG algorithm is very similar to SGD with momentum with a slight variant. In the case of SGD with momentum algorithm, the momentum and gradient are computed on previous updated weight.
- Both NAG and SGD with momentum algorithms work equally well and share the same advantages and disadvantages.

$$v_t = \gamma v_{t-1} + \eta \nabla J(w_t - \gamma v_{t-1})$$

$$w_t = w_{t-1} - v_t$$



(a) Momentum-Based Gradient Descent



(b) Nesterov Accelerated Gradient Descent

$$\text{○} \Longrightarrow \frac{\partial L}{\partial w_0} = \frac{Negative(-)}{Positive(+)} \qquad \text{○} \Longrightarrow \frac{\partial L}{\partial w_0} = \frac{Negative(-)}{Negative(-)}$$

figure (a) :

- In figure (a), update 1 is positive i.e., the gradient is negative because as w_0 increases L decreases. Even update 2 is positive as well and you can see that the update is slightly larger than update 1 because of momentum.
- By now, you should be convinced that update 3 will be bigger than both update 1 and 2 simply because of momentum and the positive update history.
- Update 4 is where things get interesting. In SGD with Momentum case, due to the positive history, the update overshoots and the descent recovers by doing negative updates.

figure (b) :

- But in NAG's case, every update happens in two steps — first, a partial update, where we get to the look_ahead point and then the final update (see the NAG update rule), see figure (b).
- First 3 updates of NAG are pretty similar to the momentum-based method as both the updates (partial and final) are positive in those cases. But the real difference becomes apparent during update 4.
- As usual, each update happens in two stages, the partial update (4a) is positive, but the final update (4b) would be negative as the calculated gradient at w_lookahead would be negative (convince yourself by observing the graph).
- This negative final update slightly reduces the overall magnitude of the update, still resulting in an overshoot but a smaller one when compared to the vanilla momentum-based gradient descent. And that my friend, is how NAG helps us in reducing the overshoots, i.e. making us take shorter U-turns.

Advantages:

- Does not miss the local minima.
- Slows if minima's are occurring.

Disadvantages:

- Still, the hyperparameter needs to be selected manually.

Point :

- By using NAG technique, we are now able to adapt error function with the help of previous and future values and thus eventually speed up the convergence. Now, in the next techniques we will try to adapt alter or vary the individual parameters depending on the importance factor it plays in each case.

6. Adaptive Gradient (AdaGrad)

- Adaptive Gradient as the name suggests adopts the learning rate of parameters by updating it at each iteration depending on the position it is present, i.e- by adapting slower learning rates when features are occurring frequently and adapting higher learning rate when features are infrequent.
- The motivation behind Adagrad is to have different learning rates for each neuron of each hidden layer for each iteration.

But why do we need different learning rates?

Data sets have two types of features:

- Dense features, e.g. House Price Data set (Large number of non-zero valued features), where we should perform smaller updates on such features; and
- Sparse Features, e.g. Bag of words (Large number of zero valued features), where we should perform larger updates on such features.

It has been found that Adagrad greatly improved the robustness of SGD, and is used for training large-scale neural nets at Google.

$$\text{For Gradient Descent: } w_t = w_{t-1} - \eta \nabla J(w)$$

$$\text{For ADA Grad : } w_t = w_{t-1} - \eta'_t \nabla J(w)$$

$$\eta'_t = \frac{\eta}{\sqrt{(\alpha_t + \varepsilon)}} \qquad \alpha_t = \sum_{i=1}^{t-1} (\nabla J(w))^2$$

η : initial Learning rate

ϵ : smoothing term that avoids division by zero

w: Weight of parameters

- In SGD learning Rate same for all weight but in Adagrad this is different for all.

Advantage:

- No need to update the learning rate manually as it changes adaptively with iterations.
- If we have some Sparse and Dense feature it automatically takes out what learning rate is suitable.

Disadvantage:

- As the number of iteration becomes very large learning rate decreases to a very small number which leads to slow convergence.
- Computationally expensive as a need to calculate the second order derivative.

Adadelta, RMSProp, and adam tries to resolve Adagrad's radically diminishing learning rates.

7. AdaDelta

- It is simply an extension of AdaGrad that seeks to reduce its monotonically decreasing learning rate.
- Instead of summing all the past gradients, AdaDelta restricts the no. of summation values to a limit (w).
- In AdaDelta, the sum of past gradients (w) is defined as "Decaying Average of all past squared gradients". The current average at the iteration then depends only on the previous average and current gradient.

$$\text{For Gradient Descent: } w_t = w_{t-1} - \eta \nabla J(w)$$

$$\text{For ADA Grad : } w_t = w_{t-1} - \eta'_t \nabla J(w)$$

$$v_t = \gamma v_{t-1} + (1-\gamma)\nabla J(w_{t-1})^2$$

$$\eta'_t = \frac{\eta_{t-1}}{\sqrt{(V_t + \varepsilon)}}$$

- Instead of inefficiently storing all previous squared gradients, we recursively define a decaying average of all past squared gradients. The running average at each time step then depends (as a fraction γ , similarly to the Momentum term) only on the previous average and the current gradient.

Advantages:

Now the learning rate does not decay and the training does not stop.

Disadvantages:

Computationally expensive.

8. RMSProp

- RMSProp is Root Mean Square Propagation. It was devised by Geoffrey Hinton.
- RMSProp tries to resolve Adagrad's radically diminishing learning rates by using a moving average of the squared gradient. It utilizes the magnitude of the recent gradient descents to normalize the gradient.
- In RMSProp learning rate gets adjusted automatically and it chooses a different learning rate for each parameter.
- RMSProp divides the learning rate by the average of the exponential decay of squared gradients
- Its cost function same as Adadelta

9. Adam — Adaptive Moment Estimation

- It is a combination of RMSProp and Momentum.
- This method computes adaptive learning rate for each parameter.
- In addition to storing the previous decaying average of squared gradients, it also holds the average of past gradient similar to Momentum. Thus, Adam behaves like a heavy ball with friction which prefers flat minima in error surface.
- Another method that calculates the individual adaptive learning rate for each parameter from estimates of first (Momentum) and second (RMSProp) moments of the gradients.

Momentum

$$v_t = \gamma v_{t-1} + \eta \nabla J(w_t)$$

$$w_t = w_{t-1} - v_t$$

RMS Prop

$$v_t = \gamma v_{t-1} + (1-\gamma) \nabla J(w_{t-1})^2$$

$$\eta'_t = \frac{\eta_{t-1}}{\sqrt{(V_t + \varepsilon)}}$$

ADAM

Momentum Component ⟶ $m_t = \beta_1 m_{t-1} + (1-\beta_1) \nabla J(w)$

RMS Prop Component ⟶ $v_t = \beta_2 v_{t-1} + (1-\beta_2) \nabla J(w)^2$

$$w_{t+1} = w_t - \frac{\eta}{\sqrt{v_t + \varepsilon}} * m_t$$

Advantages:

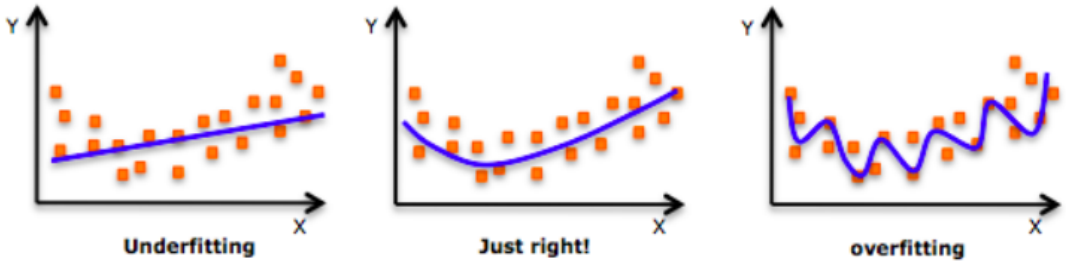
- The method is too fast and converges rapidly.
- Rectifies vanishing learning rate, high variance.

Disadvantages:

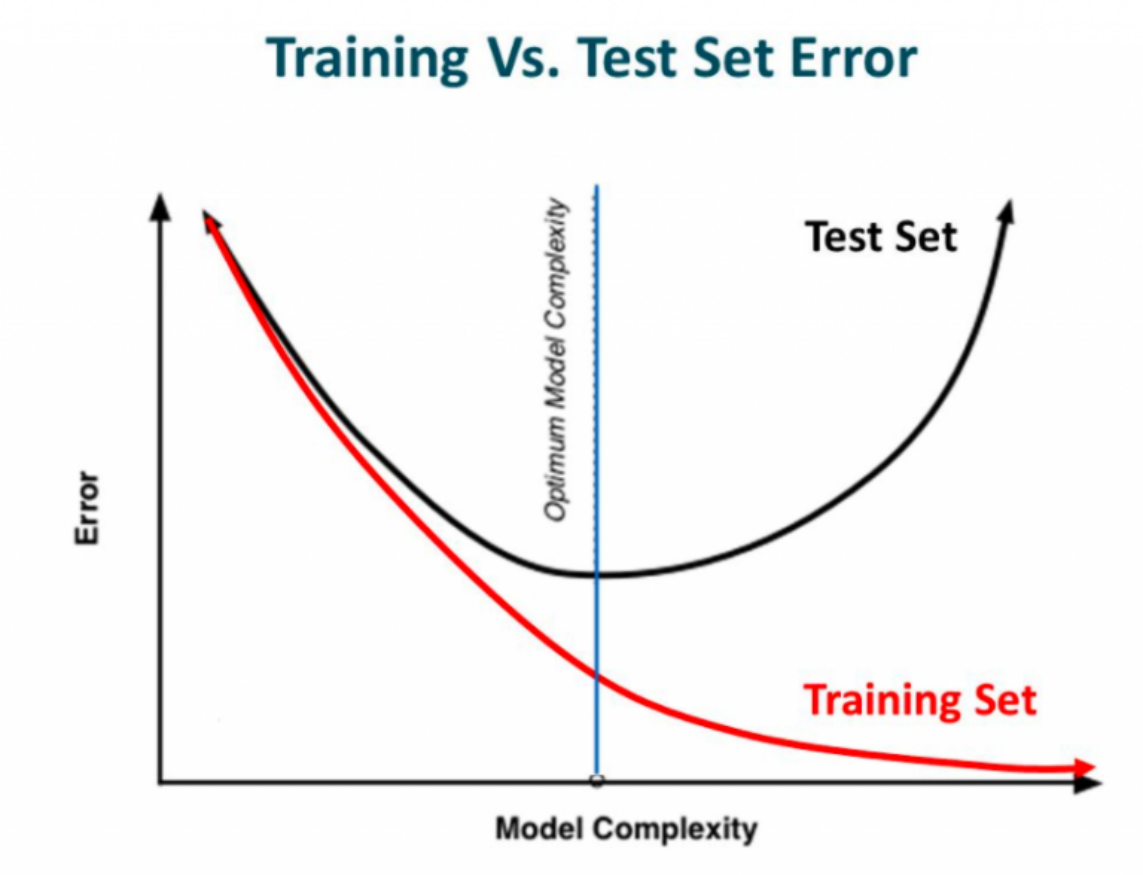
- Computationally costly.

Regularization Techniques

- One of the most common problem data science professionals face is to avoid overfitting. Have you come across a situation where your model performed exceptionally well on train data, but was not able to predict test data.



- Have you seen this image before? 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.
- In other words, while going towards the right, the complexity of the model increases such that the training error reduces but the testing error doesn't. This is shown in the image below.



- If you've built a neural network before, you know how complex they are. This makes them more prone to overfitting. 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.

.

Different Regularization Techniques in Deep Learning

1. L1 & L2 regularization

- L1 and L2 are the most common types of regularization. These update the general cost function by adding another term known as the regularization term.

```
                                Cost function = Loss (say, binary cross entrop
    y) + Regularization term
```

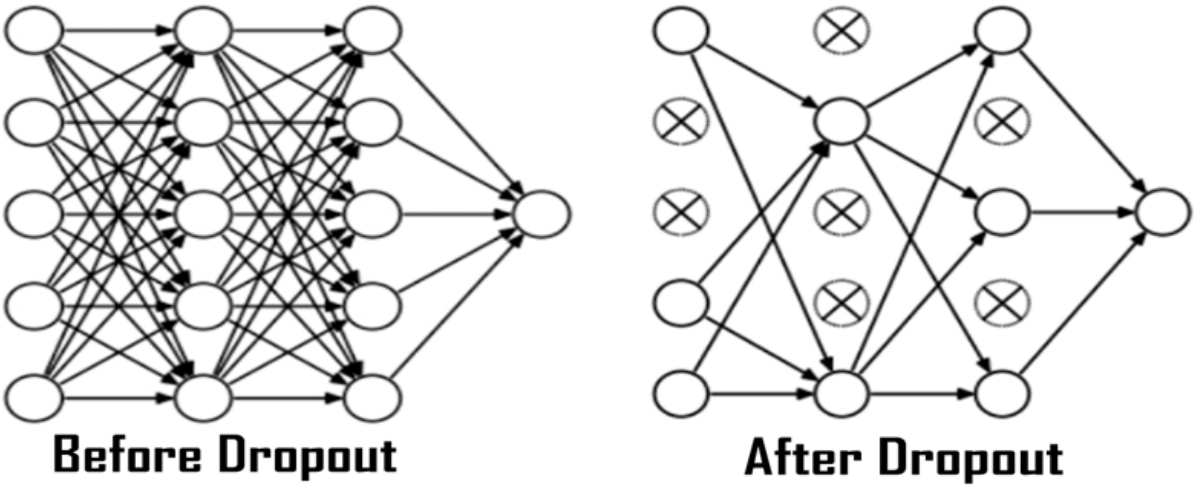
- Due to the addition of this regularization term, the values of weight matrices decrease because it assumes that a neural network with smaller weight matrices leads to simpler models. Therefore, it will also reduce overfitting to quite an extent.

```
## Below is the sample code to apply L2 regularization to a Dense layer.

from keras import regularizers
model.add(Dense(64, input_dim=64,
                kernel_regularizer=regularizers.l2(0.01)


## Below is the sample code to apply L1 regularization to a Dense layer.

from keras import regularizers
model.add(Dense(64, input_dim=64,
                kernel_regularizer=regularizers.l1(0.01)
```

Note: Here the value 0.01 is the value of regularization parameter, i.e., lambda, which we need to optimize further. We can optimize it using the hyper parameter tuning method.

2. Dropout

- This is the one of the most interesting types of regularization techniques. It also produces very good results and is consequently the most frequently used regularization technique in the field of deep learning.
- To understand dropout, let’s say our neural network structure is akin to the one shown below:



- So what does dropout do? At every iteration, it randomly selects some nodes and removes them along with all of their incoming and outgoing connections as shown below. So each iteration has a different set of nodes and this results in a different set of outputs. It can also be thought of as an ensemble technique in machine learning.

Point:-

- dropout is usually preferred when we have a large neural network structure in order to introduce more randomness.

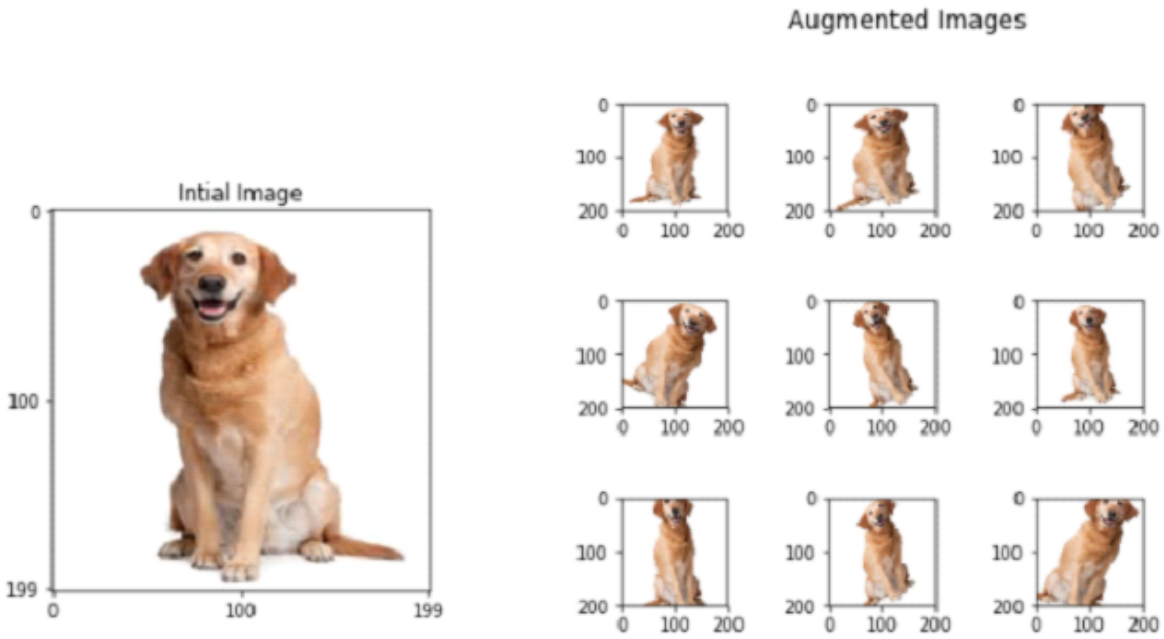
```
## In keras, we can implement dropout using the keras core layer. Below is the
python code for it:

from keras.layers.core import Dropout

model = Sequential([
 Dense(output_dim=hidden1_num_units, input_dim=input_num_units,
activation='relu'),
 Dropout(0.25),

Dense(output_dim=output_num_units, input_dim=hidden5_num_units,
activation='softmax'),
 ])
```

Note : As you can see, we have defined 0.25 as the probability of dropping. We can tune it further for better results using the Hyper parameter tuning method.

3. Data Augmentation (For Image Data)

- The simplest way to reduce overfitting is to increase the size of the training data. In machine learning, we were not able to increase the size of training data as the labeled data was too costly.
- But, now let's consider we are dealing with images. In this case, there are a few ways of increasing the size of the training data – rotating the image, flipping, scaling, shifting, etc. In the below image, some transformation has been done on the handwritten digits dataset.

- This technique is known as data augmentation. This usually provides a big leap in improving the accuracy of the model. It can be considered as a mandatory trick in order to improve our predictions.
- In keras, we can perform all of these transformations using ImageDataGenerator. It has a big list of arguments which you you can use to pre-process your training data.



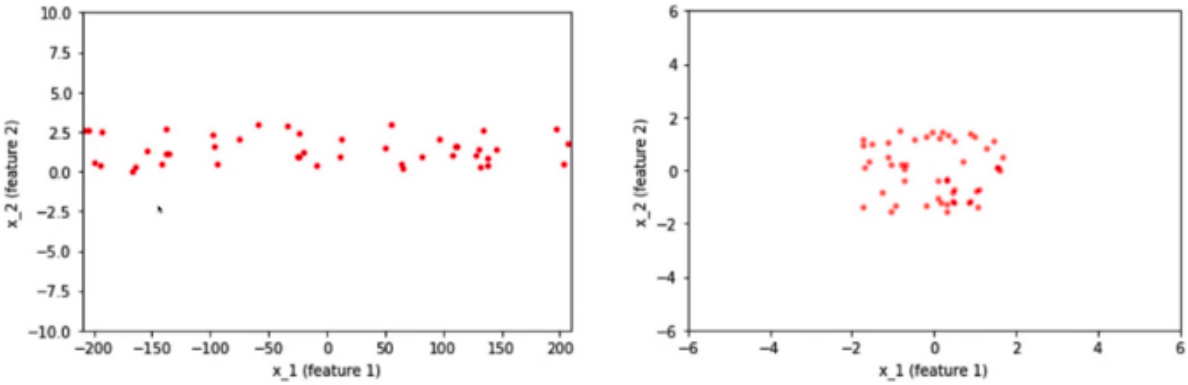

```
## Below is the sample code to implement it.
from keras.preprocessing.image import ImageDataGenerator
datagen = ImageDataGenerator(horizontal flip=True)
datagen.fit(train)
```

4. Batch Normalization

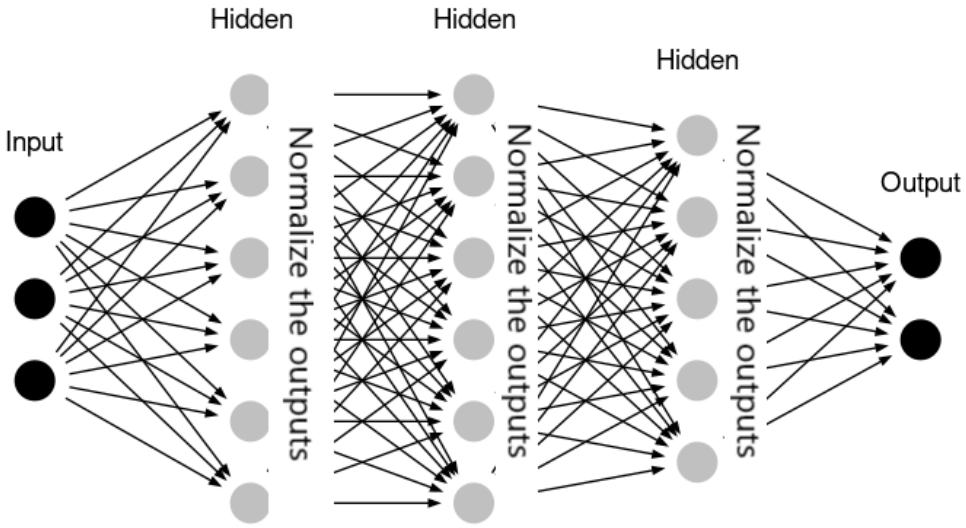
- Batch normalization is a technique for improving the speed, performance, and stability of artificial neural networks, also known as batch norm. The idea is to normalize the inputs of each layer in such a way that, they have a mean activation output zero and a unit standard deviation.
- The reason for the ‘batch’ in the term Batch Normalization is because neural networks are usually trained with a collated set of data at a time, this set or group of data is referred to as a batch. The operation within the BN technique occurs to an entire batch of input values as opposed to a single input value.

Why should we normalize the input?

- Let say we have 2D data, X1, and X2. X1 feature has a very wider spread between 200 to -200 whereas the X2 feature has a very narrow spread. The left graph shows the variance of the data which has different ranges. The right graph shows data lies between -2 to 2 and it’s normally distributed with 0 mean and unit variance.



- Essentially, scaling the inputs through normalization gives the error surface a more spherical shape, where it would otherwise be a very high curvature ellipse. Having an error surface with high curvature will mean that we take many steps that aren't necessarily in the optimal direction.
- When we scale the inputs, we reduce the curvature, which makes methods that ignore curvature like gradient descent work much better. When the error surface is circular or spherical, the gradient points right at the minimum.



Benefits of Batch Normalization

- Inclusion of Batch Normalization technique in deep neural networks improves training time
- BN enables the utilization of larger learning rates, this shortness the time of convergence when training neural networks
- Reduces the common problem of vanishing gradients
- Covariate shift within neural network is reduced

Point: In Batch normalization just as we standardize the inputs, the same way we standardize the activation at all the layers so that, at each layer we have 0 mean and unit standard deviation.

```
## In keras, we can implement BatchNormalization using the keras layer. Below
is the python code for it:
model = Sequential([
 Dense(output_dim=hidden1_num_units, input_dim=input_num_units,
activation='relu'),
 keras.layers.BatchNormalization(),

Dense(output_dim=output_num_units, input_dim=hidden5_num_units,
activation='softmax'),
 ])
```

In [1]:

```
import pandas as pd
from matplotlib import pyplot as plt
import numpy as np
%matplotlib inline

from sklearn.model_selection import train_test_split
from sklearn.preprocessing import LabelEncoder
from sklearn.preprocessing import StandardScaler

from sklearn.metrics import mean_absolute_error,mean_squared_error
from sklearn.metrics import confusion_matrix , classification_report,accuracy_score


import tensorflow as tf
from tensorflow import keras
from keras.models import Sequential
from keras.layers import Dense
```

Binary Classifier

In [2]:

```
df = pd.read_csv("kaggle_diabetes.csv")
df.head()
```

Out[2]:

	Pregnancies	Glucose	BloodPressure	SkinThickness	Insulin	BMI	DiabetesPedigreeFunctio
0	2	138	62	35	0	33.6	0.12
1	0	84	82	31	125	38.2	0.23
2	0	145	0	0	0	44.2	0.63
3	0	135	68	42	250	42.3	0.36
4	1	139	62	41	480	40.7	0.53

In [3]:

```
df.shape
```

Out[3]:

```
(2000, 9)
```

In [4]:

```
X = df.drop('Outcome',axis=1)
y = df['Outcome']

X_train, X_test, y_train, y_test = train_test_split(X,y,test_size=0.2,random_state=5)
```

Apply ANN

In [5]:

```
classifier = Sequential()
##input 1st layer
classifier.add(Dense(16,activation='relu',input_dim=8))


## second hidden layer
classifier.add(Dense(8,activation='relu'))


## output layer
classifier.add(Dense(1,activation='sigmoid'))
```

In [6]:

```
classifier.compile(optimizer='adam',
              loss='binary_crossentropy',
              metrics=['accuracy'])
```

In [7]:

```
classifier.fit(X_train, y_train, epochs=150)
```

```
Epoch 1/150
50/50 [==============================] - 1s 1ms/step - loss: 6.1302 - accu
racy: 0.4691
Epoch 2/150
50/50 [==============================] - 0s 1ms/step - loss: 1.5064 - accu
racy: 0.6145
Epoch 3/150
50/50 [==============================] - 0s 1ms/step - loss: 1.1917 - accu
racy: 0.6264
Epoch 4/150
50/50 [==============================] - 0s 1ms/step - loss: 0.9443 - accu
racy: 0.6452
Epoch 5/150
50/50 [==============================] - 0s 1ms/step - loss: 0.8631 - accu
racy: 0.6444
Epoch 6/150
50/50 [==============================] - 0s 2ms/step - loss: 0.8198 - accu
racy: 0.6555
Epoch 7/150
```

In [8]:

```
classifier.evaluate(X_test, y_test)
```

```
13/13 [==============================] - 0s 1ms/step - loss: 0.4997 - accura
cy: 0.7775
```

Out[8]:

```
[0.4997430741786957, 0.7774999737739563]
```

In [9]:

```
y_pred = classifier.predict(X_test)
```

In [10]:

```
yp = classifier.predict(X_test)
yp[:5]
```

Out[10]:

```
array([[0.1072953 ],
       [0.03356129],
       [0.20200807],
       [0.16860384],
       [0.50361   ]], dtype=float32)
```

In [11]:

```
y_pred = []
for element in yp:
    if element > 0.5:
        y_pred.append(1)
    else:
        y_pred.append(0)
```

In [12]:

```
print(classification_report(y_test,y_pred))
```

```
              precision    recall  f1-score   support

           0       0.79      0.92      0.85       269
           1       0.74      0.49      0.59       131

    accuracy                           0.78       400
   macro avg       0.77      0.70      0.72       400
weighted avg       0.77      0.78      0.76       400
```

Multi Classification Using ANN

In [13]:

```
data = pd.read_csv('https://gist.githubusercontent.com/curran/a08a1080b88344b0c8a7/raw/0e7a
data.shape
```

Out[13]:

```
(150, 5)
```

In [14]:

```
data.head()
```

Out[14]:

	sepal_length	sepal_width	petal_length	petal_width	species
0	5.1	3.5	1.4	0.2	setosa
1	4.9	3.0	1.4	0.2	setosa
2	4.7	3.2	1.3	0.2	setosa
3	4.6	3.1	1.5	0.2	setosa
4	5.0	3.6	1.4	0.2	setosa

Split in to X and y

In [15]:

```
X = data.drop('species',axis=1)
y = data['species']
```

Encoding target variable Using Label or Dummy

imp1:- If you use dummy then in loss function you use categorical_crossentropy

imp2:- if you use label_encoding then in loss function you use sparse_categorical_crossentropy

If your targets are **one-hot encoded**, use `categorical_crossentropy`.

- Examples of one-hot encodings:
  - `[1,0,0]`
  - `[0,1,0]`
  - `[0,0,1]`

But if your targets are **integers**, use `sparse_categorical_crossentropy`.

- Examples of integer encodings (*for the sake of completion*):
  - `1`
  - `2`
  - `3`

In [16]:

```
## Label

lb = LabelEncoder()
y_enc = lb.fit_transform(y)
y_enc
```

Out[16]:

```
array([0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0,
       0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0,
       0, 0, 0, 0, 0, 0, 1, 1, 1, 1, 1, 1, 1, 1, 1, 1, 1, 1, 1, 1, 1, 1,
       1, 1, 1, 1, 1, 1, 1, 1, 1, 1, 1, 1, 1, 1, 1, 1, 1, 1, 1, 1, 1, 1,
       1, 1, 1, 1, 1, 1, 1, 1, 1, 1, 1, 2, 2, 2, 2, 2, 2, 2, 2, 2, 2,
       2, 2, 2, 2, 2, 2, 2, 2, 2, 2, 2, 2, 2, 2, 2, 2, 2, 2, 2, 2, 2,
       2, 2, 2, 2, 2, 2, 2, 2, 2, 2, 2, 2, 2, 2, 2, 2, 2])
```

In [17]:

```
# dummy
# y_dummy = pd.get_dummies(y).values
```

In [18]:

```
X_train,X_test,y_train,y_test = train_test_split(X,y_enc,test_size=0.25,random_state=4)
```

In [19]:

```
sc = StandardScaler()
X_train_scaled = sc.fit_transform(X_train)
X_test_scaled = sc.transform(X_test)
```

In [20]:

```
X.shape
```

Out[20]:

```
(150, 4)
```

Apply ANN

In [21]:

```
classifier = Sequential()
classifier.add(Dense(10,input_dim = 4,activation = "relu"))

classifier.add(Dense(3,activation = "softmax"))
```

In [22]:

```
## if target dummy encoding use categorical_crossentropy if use label encoding use sparse_c
classifier.compile(optimizer = 'adam' , loss = 'sparse_categorical_crossentropy',
                   metrics = ['accuracy'] )
```

In [23]:

```
classifier.fit(X_train_scaled , y_train ,epochs = 100)
```

```
Epoch 1/100
4/4 [==============================] - 0s 2ms/step - loss: 0.8949 - accura
cy: 0.6033
Epoch 2/100
4/4 [==============================] - 0s 2ms/step - loss: 0.9165 - accura
cy: 0.5705
Epoch 3/100
4/4 [==============================] - 0s 2ms/step - loss: 0.8442 - accura
cy: 0.6268
Epoch 4/100
4/4 [==============================] - 0s 4ms/step - loss: 0.8412 - accura
cy: 0.6121
Epoch 5/100
4/4 [==============================] - 0s 4ms/step - loss: 0.8268 - accura
cy: 0.5933
Epoch 6/100
4/4 [==============================] - 0s 3ms/step - loss: 0.8125 - accura
cy: 0.6260
Epoch 7/100
```

In [24]:

```
y_pred = classifier.predict(X_test_scaled)
# y_test= np.argmax(y_test,axis=1) # when use dummy encoding in target
y_pred = np.argmax(y_pred,axis=1)
```

In [25]:

```
print(classification_report(y_test,y_pred))
```

```
              precision    recall  f1-score   support

           0       1.00      1.00      1.00        18
           1       1.00      0.88      0.93         8
           2       0.92      1.00      0.96        12

    accuracy                           0.97        38
   macro avg       0.97      0.96      0.96        38
weighted avg       0.98      0.97      0.97        38
```

In [ ]:

```

```

In [ ]:

```

```

Regression

In [26]:

```
df=pd.read_csv('https://raw.githubusercontent.com/krishnaik06/Keras-Tuner/main/Real_Combine
```

In [27]:

```
df.head()
```

Out[27]:

	T	TM	Tm	SLP	H	VV	V	VM	PM 2.5
0	7.4	9.8	4.8	1017.6	93.0	0.5	4.3	9.4	219.720833
1	7.8	12.7	4.4	1018.5	87.0	0.6	4.4	11.1	182.187500
2	6.7	13.4	2.4	1019.4	82.0	0.6	4.8	11.1	154.037500
3	8.6	15.5	3.3	1018.7	72.0	0.8	8.1	20.6	223.208333
4	12.4	20.9	4.4	1017.3	61.0	1.3	8.7	22.2	200.645833

In [28]:

```
df.dropna(inplace=True)
```

In [29]:

```
X=df.drop('PM 2.5',axis=1) ## independent features
y=df['PM 2.5'] ## dependent features
```

In [30]:

```
X_train, X_test, y_train, y_test = train_test_split(X, y, test_size=0.3, random_state=0)
```

In [31]:

```
sc = StandardScaler()
X_train = sc.fit_transform(X_train)
X_test = sc.transform(X_test)
```

In [32]:

```
X.shape
```

Out[32]:

```
(1092, 8)
```

Apply ANN

In [33]:

```
classifier = Sequential()
classifier.add(Dense(10,input_dim = 8,activation = "relu"))

## second hidden layer
classifier.add(Dense(8,activation='relu'))

classifier.add(Dense(1,activation = "linear"))
```

In [34]:

```
## if target dummy encoding use categorical_crossentropy if use label encoding use sparse_c
classifier.compile(optimizer = 'adam' , loss = 'mse',
                   metrics = ['mse'] )
```

In [35]:

```
classifier.fit(X_train , y_train ,epochs = 100)
```

```
Epoch 1/100
24/24 [==============================] - 1s 2ms/step - loss: 18824.7756 -
mse: 18824.7756
Epoch 2/100
24/24 [==============================] - 0s 2ms/step - loss: 20844.0150 -
mse: 20844.0150
Epoch 3/100
24/24 [==============================] - 0s 2ms/step - loss: 21034.4066 -
mse: 21034.4066
Epoch 4/100
24/24 [==============================] - 0s 2ms/step - loss: 21785.1067 -
mse: 21785.1067
Epoch 5/100
24/24 [==============================] - 0s 2ms/step - loss: 16766.3743 -
mse: 16766.3743
Epoch 6/100
24/24 [==============================] - 0s 2ms/step - loss: 19805.6353 -
mse: 19805.6353
Epoch 7/100
```

In [36]:

```
classifier.evaluate(X_test, y_test)
```

```
11/11 [==============================] - 0s 2ms/step - loss: 3349.8369 - ms
e: 3349.8369
```

Out[36]:

```
[3349.8369140625, 3349.8369140625]
```

In [37]:

```
y_pred = classifier.predict(X_test)
```

In [38]:

```
print("MAE",mean_absolute_error(y_test,y_pred))
print("MSE",mean_squared_error(y_test,y_pred))
print("RMSE",mean_squared_error(y_test,y_pred,squared=False))
```

```
MAE 40.52856917885261
MSE 3349.837197332338
RMSE 57.87777809602177
```