DEEP LEARNING

Chapter 1:- Intro to Neural Networks

Deep Learning is very exciting part of any AI applications. It is used in anything from self-driving cars to computer vision & speech to text recognition etc.

Process:- Neural networks

Deep Neural Networks

Recursive Neural Networks

Convolutional Neural Networks

Architectures like Generate Adversarial Network

Reinforcement Learning

1.1

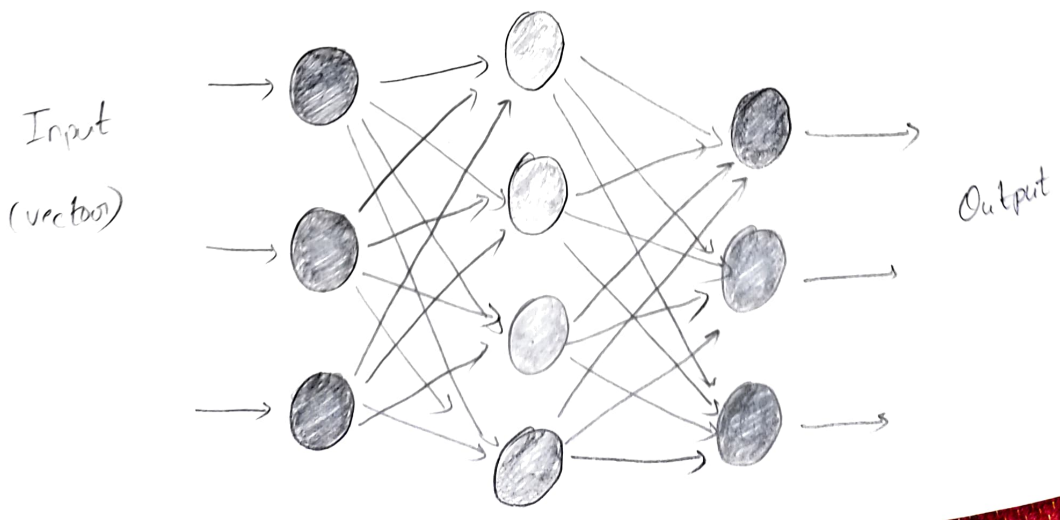
Introduction to Neural Networks

Background on Neural Networks:-

- use biology as inspiration for maths model
- Get signals from previous neurons
- Generate signals (or not) according to inputs
- Pass signals on to next neurons
- By layering many neurons, can create a complex model

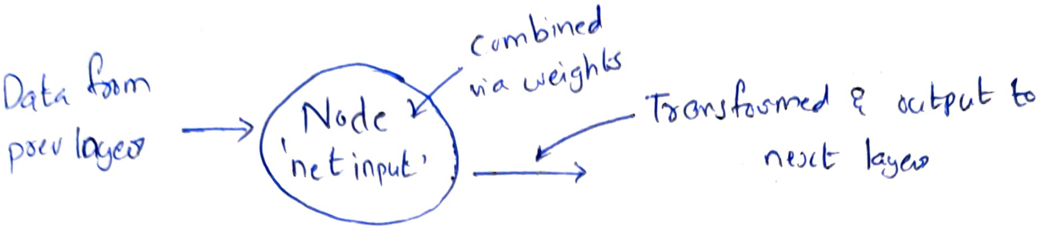
Neural Net Structure:-

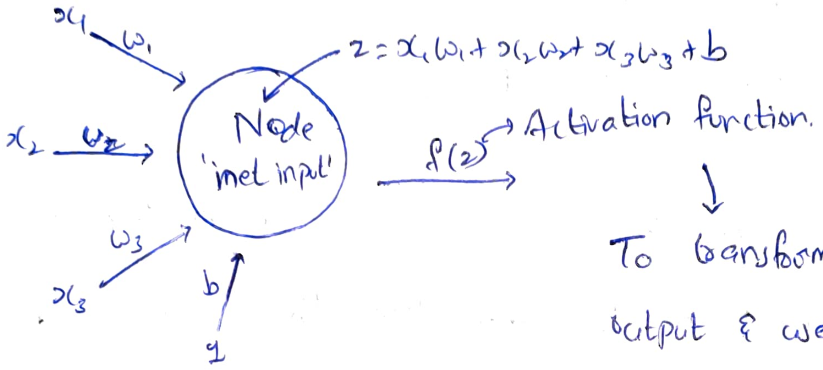
- Can think of it is a complicated computation engine
- we will "train it" using our training data
- Then use it to generate our predictions.
- Similar like Supervised Machine learning Algorithms in the process of prediction.



1.2 Basics of Neurons

Basic Neuron Visualisation:-





To transform that output & use that value as input for the next layer.

→ If we don't use Activation function, we will be only restricted to the linear output / linear combinations. No matter how deep we go we still remain with linear output. This function allows for the great flexibility with respect to how we consider the model outputs.

In Vector Notation:-

z = "net input b = "bias term" f = "activation term"

a = output to next layer.

Relation to Logistic Regression:-

$$f(z) = \frac{1}{1+e^{-z}} \qquad z = b + \sum_{i=1}^{m} x_i w_i$$

Then a neuron is simply a 'unit' of logistic regression

weights ⟷ coefficients

inputs ⟷ variable

bias term ⟷ constant term

As LR & NN in a way can acomplish the same task. If we're trying to accomplish the same task. If we're trying to accomplish classification, we want to ensure that when we move to a neural network that we actually need a more complex model. We should switch over NN only when we have multiple units & perhaps multiple layers

→ More complex boundary with Neural network than LR &.

→ Loss of lot's of explanatory value that you have with LR in Neural networks

1.3 Neural Networks with Sklearn

Nice property of Sigmoid function:

$$\sigma(z) = \frac{1}{1+e^{-z}}$$

$$\frac{d}{dx}\frac{y_1}{y_2} = \frac{y_2 y_1' - y_1 y_2'}{y^2}$$

$$\sigma'(z) = \frac{0-(-e^{-z})}{(1+e^{-z})^2} = \frac{e^{-z}}{(1+e^{-z})^2} = \frac{e^{-z}+1-1}{(1+e^{-z})^2} = \frac{\cancel{1+e^{-z}}}{(1+e^{-z})^{\cancel{2}}} - \frac{1}{(1+e^{-z})^2}$$

$$\sigma'(z) = \frac{1}{1+e^{-z}} - \frac{1}{(1+e^{-z})^2} = \frac{1}{1+e^{-z}}\left(1 - \frac{1}{1+e^{-z}}\right)$$

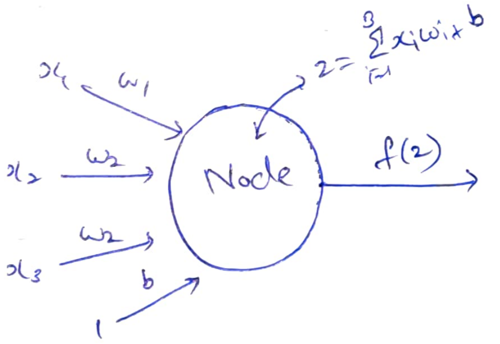
$$\boxed{\sigma'(z) = \sigma(z)(1-\sigma(z))}$$

$$\boxed{f'(z) = f(z)[1-f(z)]}$$

This will be useful as we tried to compute our neural network

Perceptron :-

single neuron.



We're going to use LR using that sigmoid activation fn. To see this neuron in ACTION!

$x_1 = 0.9$ $\quad w_1 = 2$

$x_2 = 0.2$ $\quad w_2 = 3$

$x_3 = 0.3$ $\quad w_3 = -1$

$b = 0.5$

$$z = (0.9\times2)+(0.2\times3)+(0.3\times-1)+0.5$$

$$z = 2.6$$

$f(z) = f(2.6) = \dfrac{1}{1+e^{-2.6}} = 0.93$

Node output = 0.93

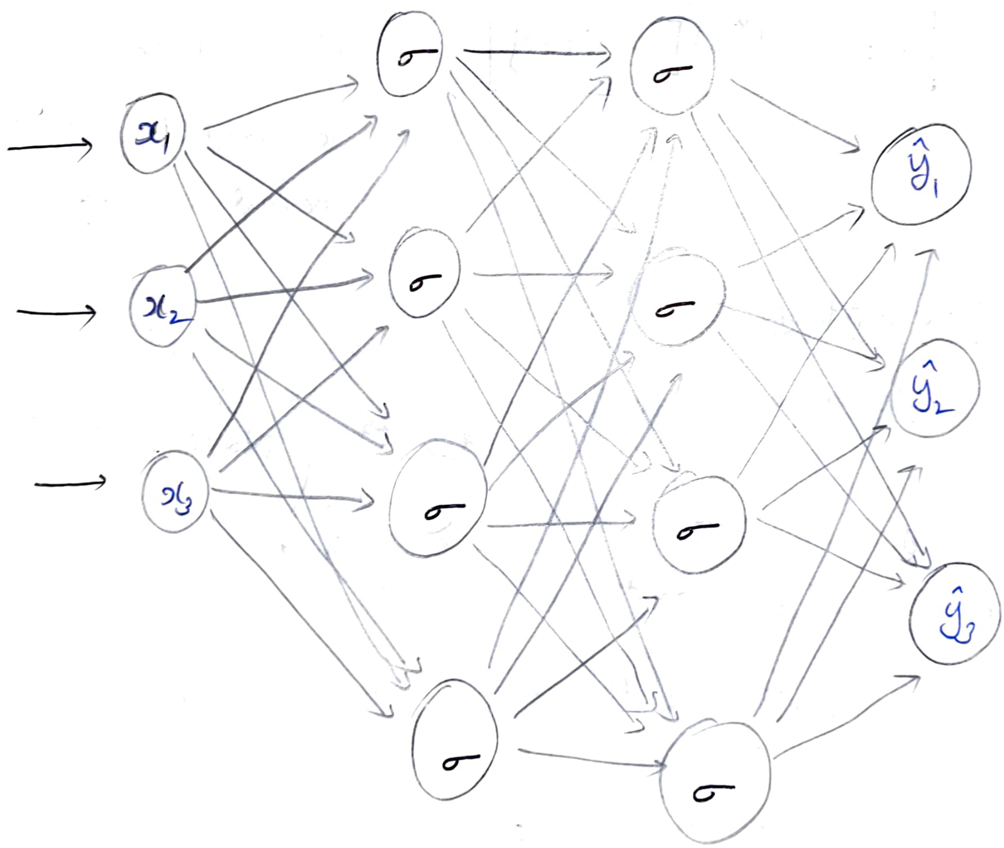
Why Neural Networks?

Why not just use a single neuron? Why do we need a larger network?

A single neuron only permits a linear density decision boundary

Most real world problems are considerably more complicated

Feed Forwards Networks :- Multi Layer perceptron :-



Every value in ~~on. first~~ one layer is connected to every value in it's succeding layer.

Multi Layer Perception Syntax :- Sckit Learn

~~from~~

```
from sklearn.neural_network import MLPClassifier

mlp= MLPClassifier(hidden_layer_sizes=(5,2), activation="logistic")

mlp.fit(X_train, y_train)

mlp.predict (X_test).
```

1.4 Forward Propagation

Multi-Layer perceptron :- Weights

→ The arrows that connects all the layers signifies the weights & how to combine each one of these different layers

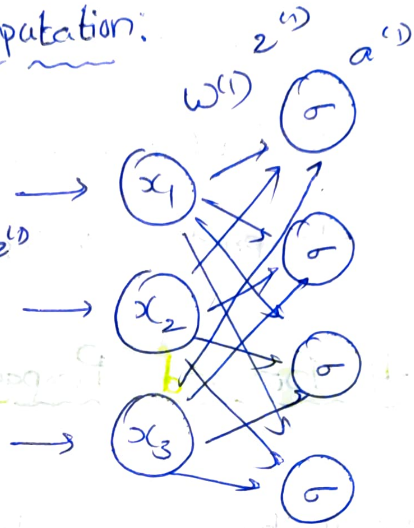
Input Layer :- which is our input dataset, thes are features

Hidden Layers :- Every layer between output & input layers

Output Layers :- This is our output of the given input processed through hidden layers

→ weights will be represented as matrices & each of those diff matrices will again, just be the way that we combine each layer step-by-step

Matrix Representation of Computation:



$W^{(1)}$ is a 3 x 4 matrix

$z^{(1)}$ is a 4-vector → $\frac{1}{3} \times \frac{3}{4} = \frac{1}{4} \to z^{(1)}$

$a^{(1)}$ is a 4-vector

$x = a^{(0)}$

$z^{(1)} = x W^{(1)}$

$a^{(1)} = \sigma(z^{(1)})$

Continuing the Computation:

For a single training instance:

Input: vector x (a row vector of length 3)

Output: vector y (a row vector of length 3)

$z^{(1)} = x \cdot W^{(1)} \longrightarrow a^{(1)} = \sigma(z^{(1)})$

$z^{(2)} = a^{(1)} W^{(2)} \longrightarrow a^{(2)} = \sigma(z^{(2)})$

$z^{(3)} = a^{(2)} W^{(3)} \longrightarrow \cancel{a^{(3)} = \sigma(} \cdot \hat{y} = \text{softmax}(z^{(3)})$

Multiple Data Points:-

In practice, we do these computation for many data points at the same time, by "stacking" the rows into a matrix. But the equations looks the same!

Inputs: matrix X (an n×3 matrix) (each row a single instance)

Output: matrix $\hat{y}$ (an n×3 matrix) (each row a single pred)

$$z^{(1)} = x\,\omega^{(1)} \longrightarrow a^{(1)} = \sigma(z^{(1)})$$

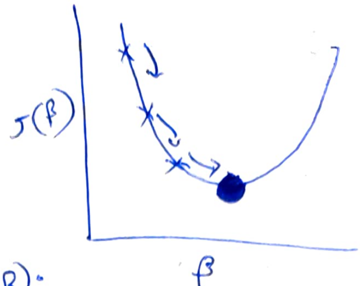
$$z^{(2)} = a^{(1)}\omega^{(2)} \longrightarrow a^{(2)} = \sigma(z^{(2)})$$

$$z^{(3)} = a^{(2)}\omega^{(3)} \longrightarrow a^{(3)} = \hat{y} = \text{softmax}(z^{(3)})$$

1.5 Deep Learning Model Summary (Informative)

S. No	Method	Use Cases
1	Neural Network Models Multi Layer perceptron, Feedforward net	Applied to many traditional predictive probs (classification & regression, tabular data)
2	Recurrent Neural Networks (RNN, LSTM)	Useful for modelling sequences& (time series forecasting, sentence prediction)
3	Convolutional Neural Networks (CNN)	useful for feature & object recognition in visual data (image, video). Also applied in other contexts (forecasting)
4	Unsupervised pre-trained nets:- Autoencoders, Deep Belief Nets, & Generative Adversarial Nets,	Many uses including generating images, labelling outcomes, dimensionality reduction.

1.6 Gradient Descent



Gradients Descent:-

start with a cost function J(β):

Then gradually move towards the minimum. This β_{min} value will be effective β value for J(β) which is our cost function.

Gradient Descent with Linear Regression:-

Imagine there are two parameters (β_0, β_1)

This is more complicated surface on which the minimum must be found.

How can we do this without knowing what $J(\beta_0, \beta_1)$ looks like?

① Start at a random point

② Then compute gradients of points in respect to β_0 & β_1
The gradients will always point in direction of largest increase

③ we take negative of that gradients, & now we are pointing in the direction of the largest decrease.

[These gradients will be a vector in that same dimensional space as our parameters, consisting of the partial derivatives of each one of those parameters]

→ These gradients will tell us direction of descent for each one of our individuals parameters

$$\bar{\nabla} J(\beta_0, \dots, \beta_n) = \left\langle \frac{\partial J}{\partial \beta_0}, \dots, \frac{\partial J}{\partial \beta_n} \right\rangle$$

4 Then use the gradient ($\bar{\nabla}$) & the cost function to calculate the next point (w_1) & from the current one (w_0):

$$w_1 = w_0 - \alpha \bar{\nabla} \frac{1}{2} \sum_{i=1}^{m} \left(\left(\beta_0 + \beta_1 x_{obs}^{(i)} \right) - y_{obs}^{(i)} \right)^2$$

The learning rate is going to be a tunable parameter, that will ~~let~~ tell us know how large we want to make each one of our steps within our cost function.

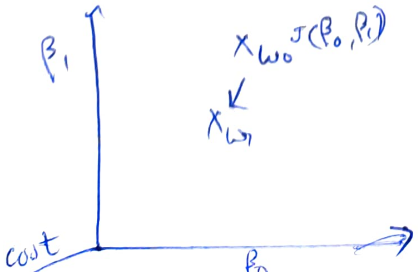
* Too large steps, → overshooting our minimum
* Too small steps → Too long to optimize our model (time taking process)

5 We can iterate the same concept of subtracting the gradient to move closer & closer to minimum value from that last step.

$$w_2 = w_1 - \alpha \bar{\nabla} \frac{1}{2} \sum_{i=1}^{m} \left(\left(\beta_0 + \beta_1 x_{obs}^{(i)} \right) - y_{obs}^{(i)} \right)^2$$

$$w_3 = w_2 - \alpha \bar{\nabla} \frac{1}{2} \sum_{i=1}^{m} \left(\left(\beta_0 + \beta_1 x_{obs}^{(i)} \right) - y_{obs}^{(i)} \right)^2$$

& eventually we end ~~and~~ up with a global minimum.



1.6.1 Stochastic Gradient Descent :-

This method is just to speed up the process by taking a single data point to determine the gradient & the cost function.

→ Here we calculate our weights by subtracting from w_0 ~~the~~ & not not summesion part as we are using only one data point. So the formula would look like this

$$w_1 = w_0 - \alpha \nabla \frac{1}{2} ((\beta_0 + \beta_1 x_{obs}^{(0)}) - y_{obs}^{(0)})^2$$

→ Then we use the single point, we can again iterate through to continue updates of weights so we can use for w_1 & each one can be diff random point but we still use a single point itself.

→ We keep updating our weight's moving down our cost function & eventually we end up nearer to global minimum.

→ Not exactly bcz of the noise with working with single data point & over all randomness in the data points we have taken

→ It being a Stochastic Gradient Descent.

Summary:- Iteration of a single data point insted of numerous in gradient descent method, end up nearer to global minimum due to the noise with working with single data point.

1.6.2 Mini Batch Gradient Descent

Let n be number between 1 & the size of entire dataset. Perform an update for every n training egs:-

$$w_1 = w_0 - \alpha \hat{\nabla} \frac{1}{2} \sum_{i=1}^{n} \left(\left(\beta_0 + \beta_1 x_{obs}^{(i)} \right) - y_{obs}^{(i)} \right)^2$$

→ we are summing over a random subset of original data, seeing our error & taking the gradients & moving down that line given the gradients for the subset of values.

→ we can reduce memory relative to "vanilla" gradient descent

→ Less noisy than stochastic gradient descent