Neural Networks

Neural Networks are models that take inspiration from brain's functioning

They aim to replicate the working of a biological neuron, that is brain cells

Neural networks are the backbone of deep learning and are very useful and complex structures

They are different from other ML models like decision trees that rely on probability or SVMs that rely on distance.

They are modelling of our brain, our human intelligence

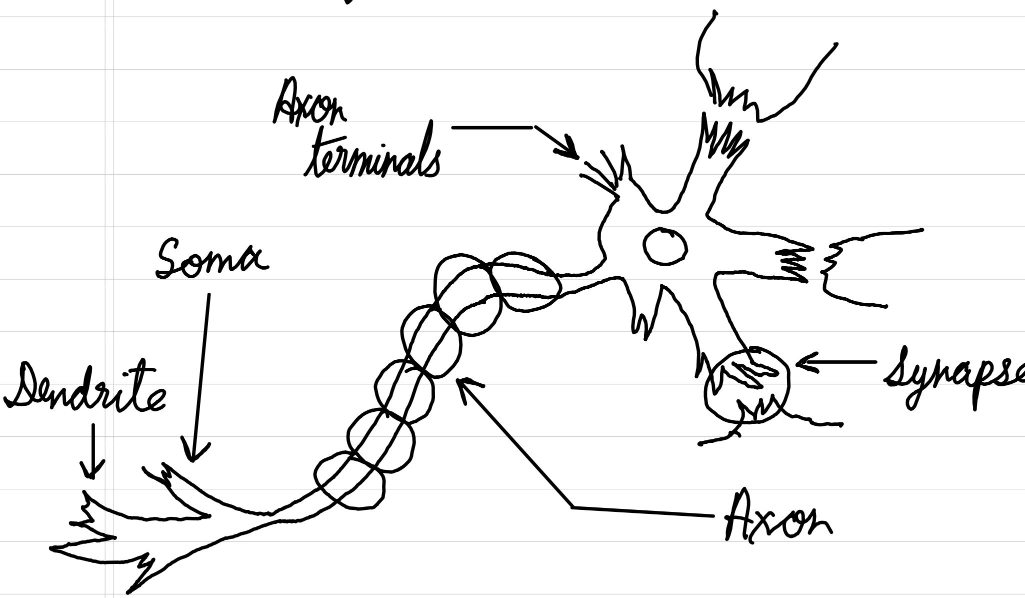
Applications of Neural Networks

1. classification
2. Recognition → OCR
3. Prediction eg crop yeild forcasting, stock Market

Neural Networks can learn and solve almost everything

This is because neural networks are universal function approximators

Biological Neurons



Structure of a biological Neuron

Dendrite : A brush of very thin fibre

Axon : A long cylinderical fibre

Soma : Cell body

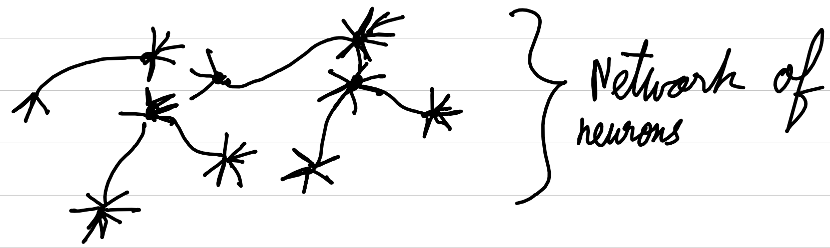
Synapse : Junction where axon makes contact with dendrites of neighboring dendrites

Biological Networks

Each neuron is 10 µm long. Human brains have 10^{11} neurons

These neurons operate in parallel and form a network of neurons.

These neurons communicate with each other with the help of electric impulses.



① Dendrite (Input) → Recieves signal from other neurons

② Soma (Processing Unit) → Sums up all input signals. Consists of a threshold value

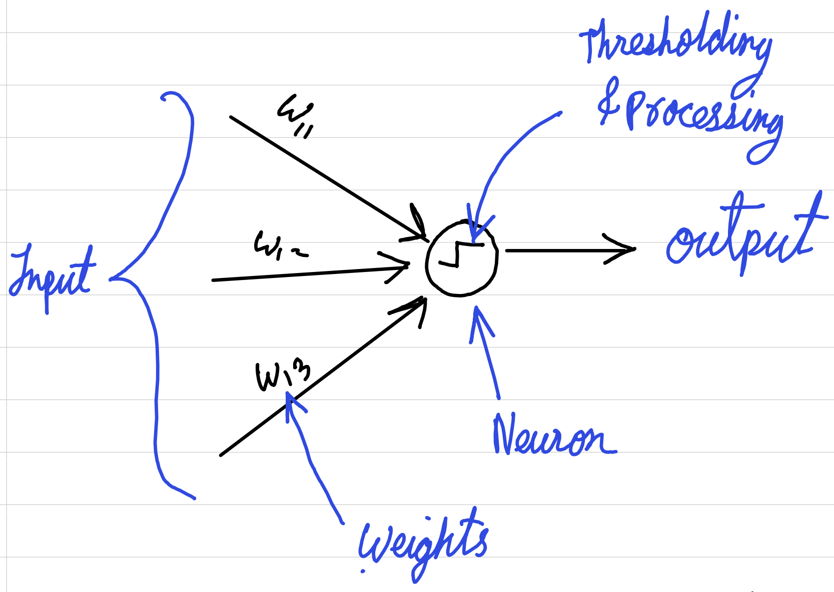
③ Synapse (Weighted Connections) →

Point of communication between neurons. Amount of signal transferred depends upon the strength (synaptic weight) of the connection

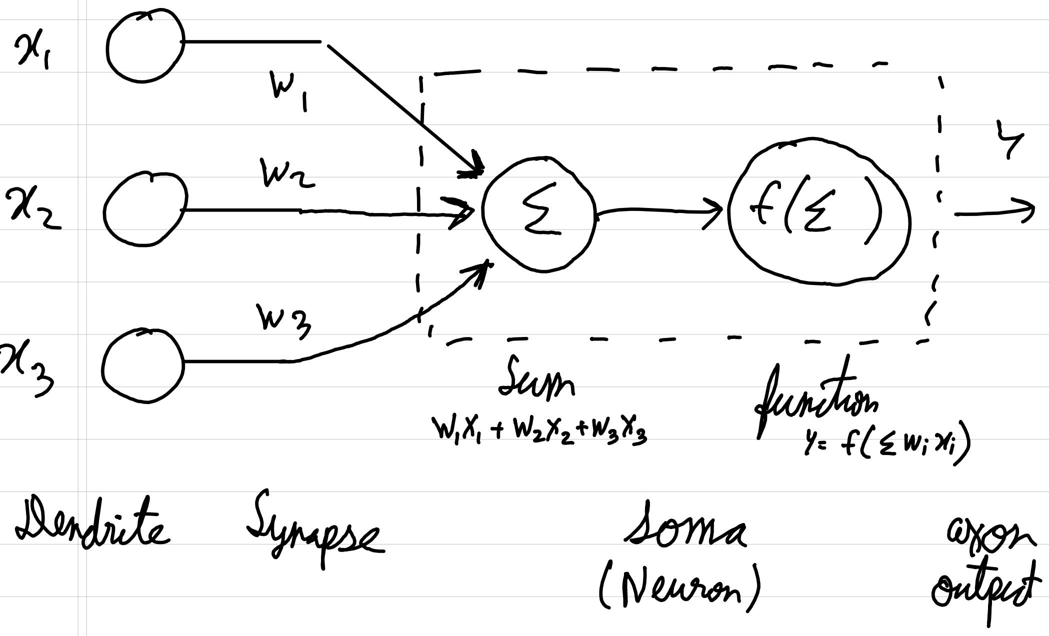
④ Axon terminals (Output) → Transmit signal. Neuron fires depending on threshold.

Artificial Neural Networks

ANN is a (vague) simulation of neural networks

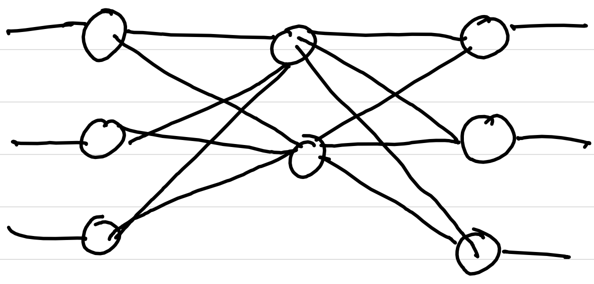


Biological	Artificial
Cell	Neuron
Dendrites	Weights
Soma	Net Input
Axon	Output



A neural network consists of a laarge number of neurons

These neurons are interconnected with each other



Activation of a neuron is the input to the next neuron

A Neural Network has parameters like weights that need to be decided when model is trained

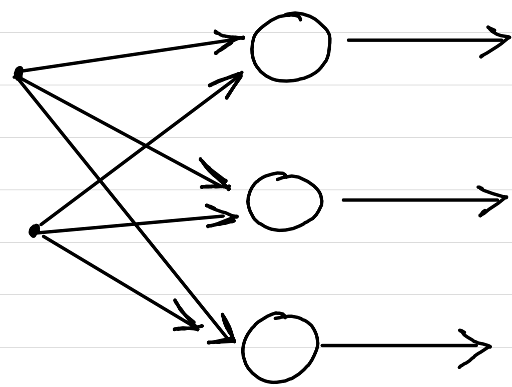
Hyperparameters include →

1. Learning rate
2. Learning rules (eg activation functions)
3. Number of layers
4. Arrangement of Neurons & Connection pattern

Network architecture

Single layer feed forward network

Simplest form of neural network



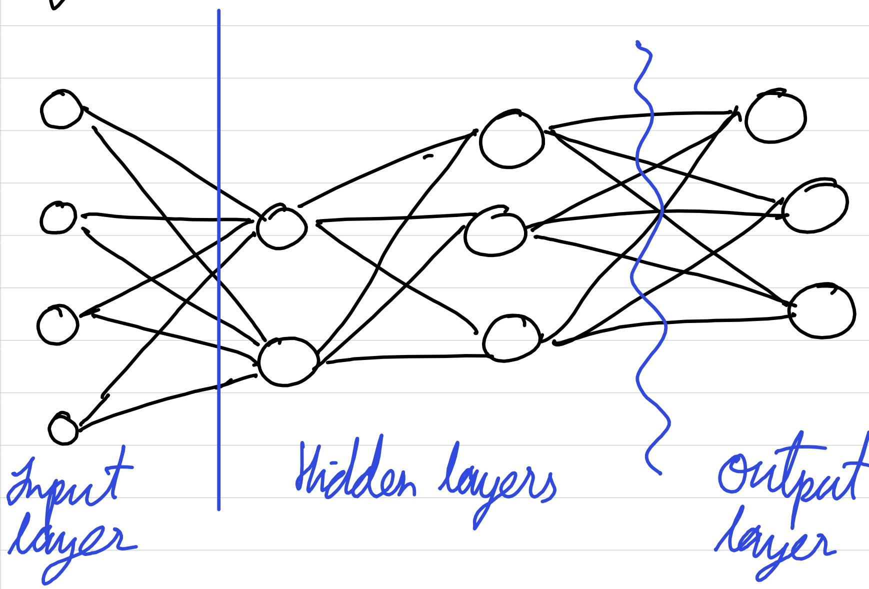
Outputs are $f(\sum w_i x_i)$

where w_i are determined (learnt) by the network

Very simple model useful only for small classification and function approximation tasks

Multilayer Feed Forward Network

Formed by interconnection of several layers



Multiple layers are present in the network

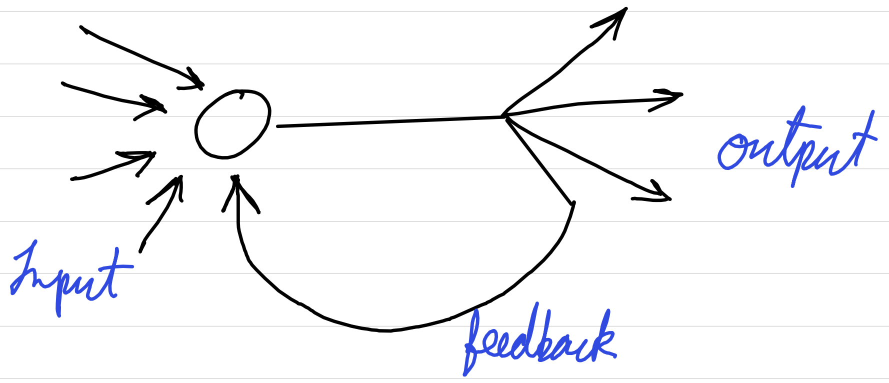
More hidden layers, more complexity of the model

Neural networks are universal function approximators because they can approximate any function

There are theoritical gurantees that neural networks with enough neurons ($\to \infty$) can approximate any function

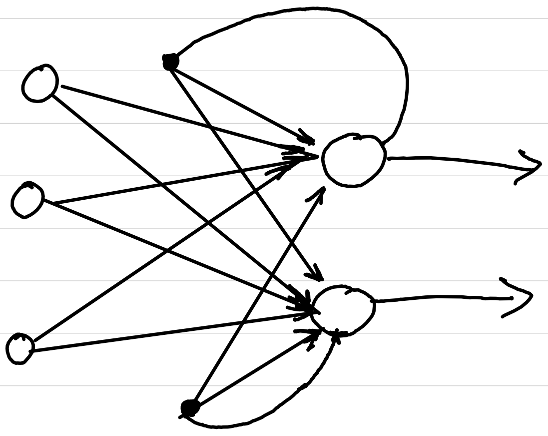
Feed Back Networks

A feedback is given to the network
That is, the output of the neurons is fed as input to the model.



Single layer Recurrent layer

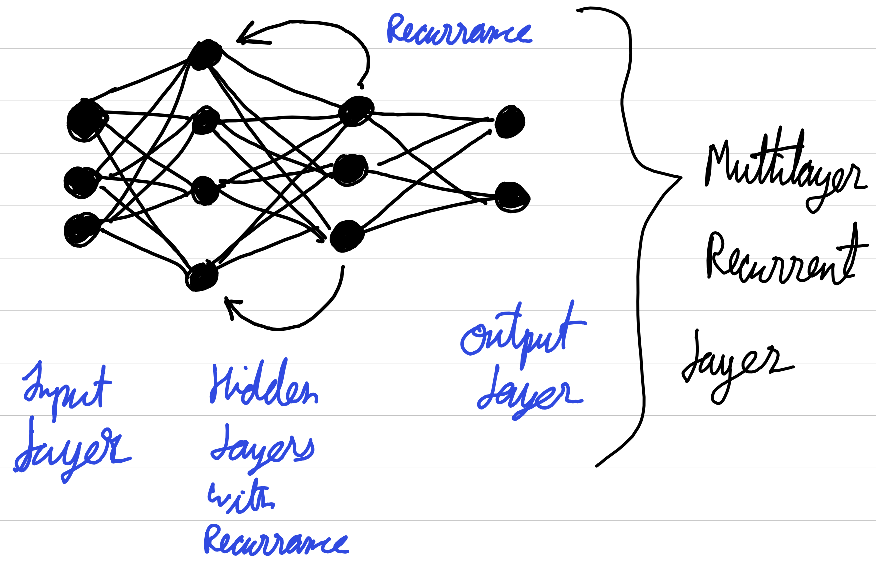
Output is fed back into entire layer



If output is directed back to same layer, then it is lateral feedback

Recurrent Networks

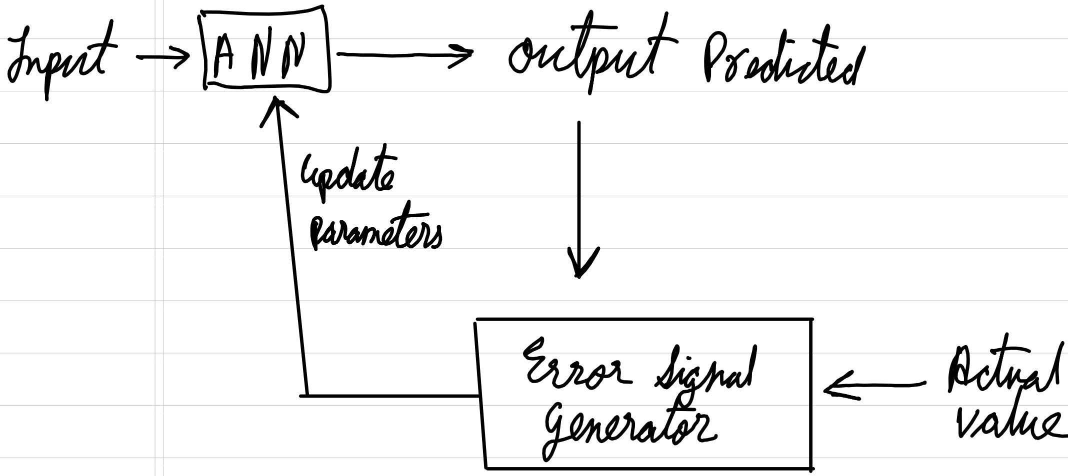
Recurrent networks are networks with feedback networks in closed loop



RNNs maintain an internal state memory helping them to recognize patterns

Helpful for NLP, speech recognition & time series prediction

Supervised Learning in ANN



The actual value & the predicted values are matched and the error value is calculated

Depending on error value, the parameters (weights) are updated

This is much like linear regression where we use gradient descent to update values of m & c

Unsupervised Neural Networks

The expected output is not known hence no explicit error function present

ANN will try to find some kind of pattern using input data set without any external aids

Known as self organizing networks

Network recieves input patterns and organizes to form clusters

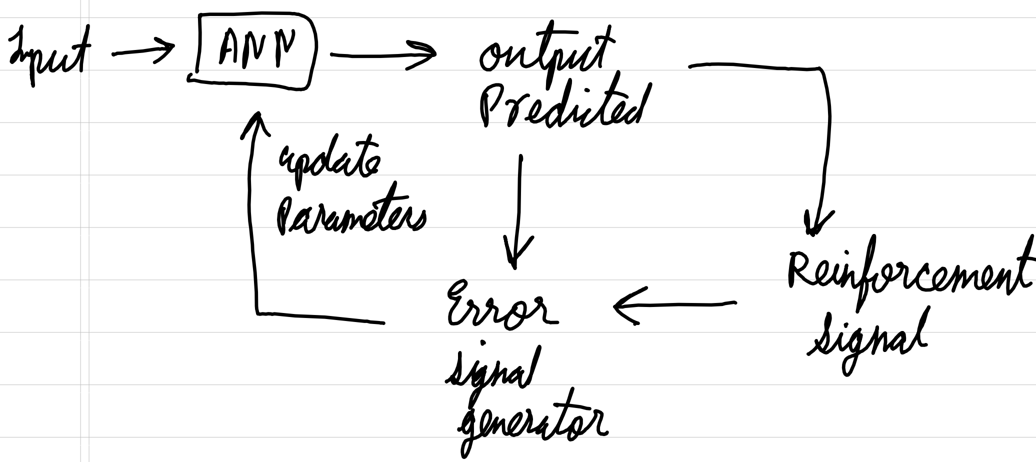
Example → GAN (Generative Adverserial Networks)

NN. Reinforcement learning

Exact information about the output is not known.

Only critic information is known

Example network might be told that only 50% of the information is correct



Neural Networks

* Advantages →

Complex Pattern Recognition

Non linearity

Feature learning

* Disadvantages →

Complexity

Blackbox Nature ∴ less interpretability

Overfitting

Data dependance → Requires large datasets

Neuron as generalization of linear regression

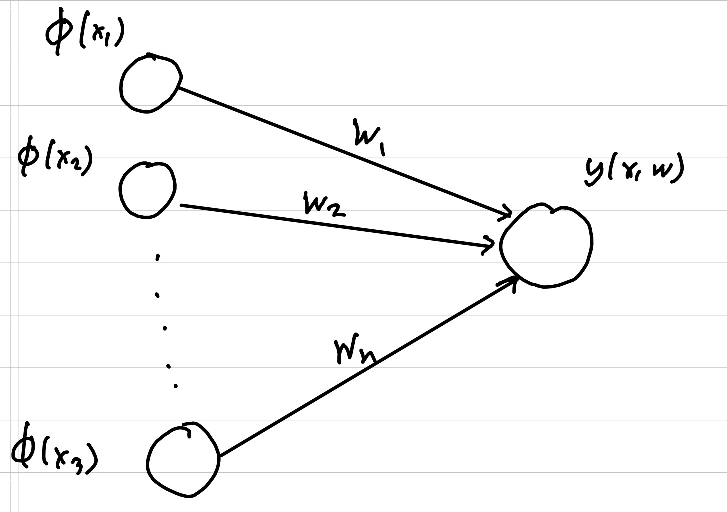
Predict y as a linear function of x

$$y(x, w) = w_0 + w_1 x_1 + \cdots w_D x_D = \sum w_i x_i$$

or derived from $\phi()$

$$y = \sum_{i=0}^{N-1} w_i \phi(x_i) = w^T \Phi(x) \qquad \Phi(x) \in \mathbb{R}^d$$

We can represent it as N.N.



Data is given by

$$\{(x_1, y_1), (x_2, y_2), \ldots (x_D, y_D)\}$$

$$y \sim N(w^T x_i, \sigma^2)$$

We need to minimize the loss function

$$\text{loss} = \sum_{i=1}^{N} \left(y_i - w^T \Phi(x_i) \right)^2$$

$$w^* = \underset{w}{\text{argmin}} \text{ loss}$$

We can have ① Multiple possible w
② Overfitting when choosing ϕ

In order to avoid we use regularization

$$\text{loss} = \sum_{i=1}^{N} \left(y_i - w^T \Phi(x_i) \right)^2 + \lambda w^T w$$

$$w^* = \underset{w}{\text{argmin}} \text{ loss}$$

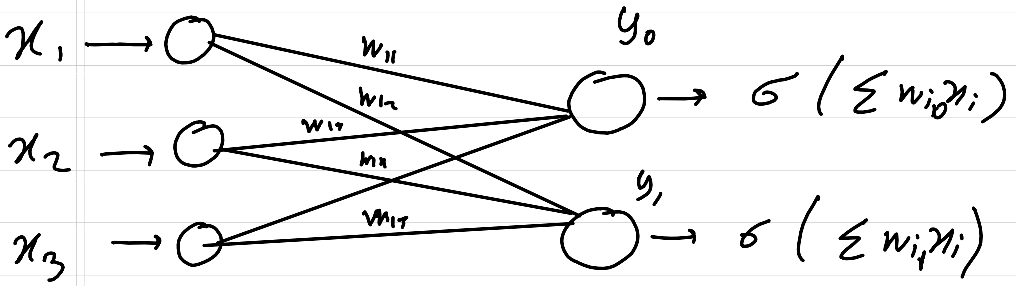
But still we are not sure about choosing ϕ

ϕ should have enough expressive power so linear model works well.

ϕ cant be too powerfull, else overfit

So let us learn ϕ automatically from data which is the idea of N.N.

A N.N. is a network such that



Every line has a weight

$$y_j = \sigma\left(\sum w_{ij} x_i\right) \quad \text{eg } y_0 = \sigma(w_{10} x_1 + w_{20} x_2 + w_{30} x_2)$$

σ: Non linear function

w_{ij}: Weight

σ is called activation function

Above is called a single layer NN

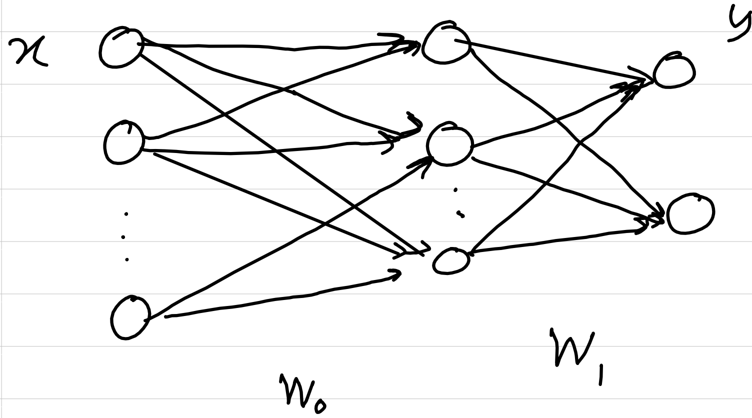
It has no hidden layers.

We can represent y as matrix

$$\sigma\left(\begin{bmatrix} w_{11} & w_{12} & w_{13} \\ w_{11} & w_{12} & w_{13} \end{bmatrix} \begin{bmatrix} x_1 \\ x_2 \\ x_3 \end{bmatrix}\right) = \sigma\left(\begin{bmatrix} \sum w_{i} x_i \\ \sum w_i' x_i \end{bmatrix}\right)$$

can be written as $\sigma(Wx)$

Consider the network



$$y = W_1 \left(\sigma \left(W_0 x \right) \right)$$

Loss $\quad \mathcal{L}(W) = \sum_i \left(y_i - W_1 \left(\sigma(W_0 x) \right)_i \right)$

We want to find weights that minimize the loss

Since we have n neurons in hidden layer. & we are minimizing weights we can have any permutation of hidden layer

eg Given a W_0 & W_1

We can have another set

$W_0' = W_{0(1 \leftrightarrow 2)}$ ie interchange row

$W_1' = W_{1\,(\cdot,\, 1 \leftrightarrow 2)}$ ie interchange columns

Hence we can have $n!$ such weights

Multiple Hidden layers

If we have multiple hidden layers, then

$$y = W_k \, \sigma(W_{k-1} \, \sigma(\cdots \sigma(W_0 x)) \cdots)$$

We can find $\frac{\partial L}{\partial W_i}$ using chain rule

Consider $\hat{y}$ as

$$\hat{y} = f_1 (W_k h_1)$$

$$h_1 = f_2 (W_{k-1} h_2)$$

$$h_2 = \cdots$$

$$\vdots$$

$$h_k = f_k (W_0 x)$$

$$\frac{\partial L}{\partial W_k} = \frac{\partial L}{\partial \hat{y}} \frac{\partial y}{\partial W_k}$$

$$\frac{\partial y}{\partial W_k} = f_1'(W_k h_1) \, h_1^T$$

$$\frac{\partial \hat{y}}{\partial W_{k-1}} = \frac{\partial \hat{y}}{\partial h_1} \frac{\partial h_1}{\partial W_{k-1}}$$

$$\frac{\partial \hat{y}}{\partial h_1} = f_1'(W_k h_1) \, W_k^T$$

$$\frac{\partial \hat{y}}{\partial W_{k-2}} = \frac{\partial \hat{y}}{\partial h} \frac{\partial h_1}{\partial h_2} \frac{\partial h_2}{\partial W_{k-2}}$$

$$\frac{\partial h_1}{\partial h_2} = f_2'(W_{k-1} h_2) \, W_{k-1}^T$$

$$\vdots$$

similarly proceed

$$\vdots$$

$$\frac{\partial h_k}{\partial W_1} = f_k'(W_0 x) \, x^T$$

Back propagation

Most common training algorithm

step 1: Randomly initialize the weights

step 2: Apply input to the network

step 3: Work out gradient for last layer (k^{th})

Error is difference in expectation vs reality.

what you want — what you get

$$\mathcal{L} = (\hat{y} - y)^2$$

We minimize the error by gradient descent

$$\frac{\partial \mathcal{L}}{\partial W_k} = \frac{\partial \mathcal{L}}{\partial y} \frac{\partial y}{\partial W_k}$$

$$\frac{\partial \mathcal{L}}{\partial W_k} = 2(\hat{y} - y) \; f_1'(W_k h_1) \; h^T$$

$$\text{Let } \delta_1 = (\hat{y} - y) \, f_1'(W_k h_1)$$

Calculate the term seperately. Note 2 doesnt matter as we are

4. Update weight

$$W_k \leftarrow W_k + \alpha \frac{\partial \mathcal{L}}{\partial W_k}$$

5. for hidden layer, back propagate from the last layer

$$\frac{\partial \mathcal{L}}{\partial W_{k-1}} = \frac{\partial \mathcal{L}}{\partial h_1} \frac{\partial h_1}{\partial W_{k-1}}$$

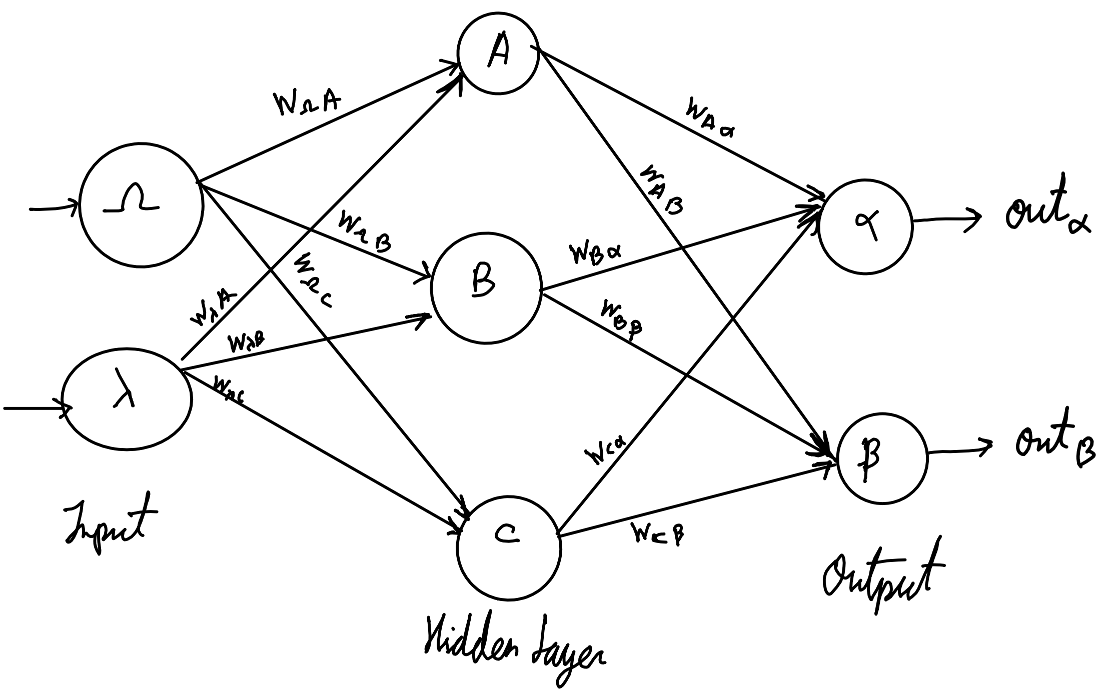
$$= 2(\hat{y} - y) \, f_1'(W_k h_1) \, W_k^T \; f_2'(W_{k-1} h_2) \, h_2^T$$

$$\frac{\partial \mathcal{L}}{\partial W_{k-1}} = \delta_1 W_k^T \; f_2'(W_{k-1} h_2) \; h_2$$

$$\text{Let } \delta_2 = \delta_1 W_k^T f_2'(W_{k-1} h_2)$$

Repeat

$$\frac{\partial \mathcal{L}}{\partial W_{k-i}} = \delta_i \; h_{i+1}$$



σ → sigmoid activation in hidden & output layer

1.

$$\delta_\alpha = out_\alpha (1 - out_\alpha)(Target_\alpha - out_\alpha)$$
$$\delta_\beta = out_\beta (1 - out_\beta)(Target_\beta - out_\beta)$$

2. Change output layer weights

$W_{A\alpha} \leftarrow W_{A\alpha} + \eta\, \delta_\alpha\, out_A$ $\quad$ $W_{A\beta} \leftarrow W_{A\beta} + \eta\, \delta_\beta\, out_A$

$W_{B\alpha} \leftarrow W_{B\alpha} + \eta\, \delta_\alpha\, out_B$ $\quad$ $W_{B\beta} \leftarrow W_{B\beta} + \eta\, \delta_\beta\, out_B$

$W_{C\alpha} \leftarrow W_{C\alpha} + \eta\, \delta_\alpha\, out_C$ $\quad$ $W_{C\beta} \leftarrow W_{C\beta} + \eta\, \delta_\beta\, out_C$

3. Backpropagate hidden layer errors

$$\delta_A = out_A (1 - out_A)(\delta_\alpha W_{A\alpha} + \delta_\beta W_{A\beta})$$
$$\delta_B = out_B (1 - out_B)(\delta_\alpha W_{B\alpha} + \delta_\beta W_{B\beta})$$
$$\delta_C = out_C (1 - out_C)(\delta_\alpha W_{C\alpha} + \delta_\beta W_{C\beta})$$

4. Change hidden layer weights

$W_{\lambda A} \leftarrow W_{\lambda A} + \eta\, \delta_A\, input_\lambda$ $\quad$ $W_{\Omega A} \leftarrow W_{\Omega A} + \eta\, \delta_A\, input_\Omega$

$W_{\lambda B} \leftarrow W_{\lambda B} + \eta\, \delta_B\, input_\lambda$ $\quad$ $W_{\Omega B} \leftarrow W_{\Omega B} + \eta\, \delta_B\, input_\Omega$

$W_{\lambda C} \leftarrow W_{\lambda C} + \eta\, \delta_C\, input_\lambda$ $\quad$ $W_{\Omega C} \leftarrow W_{\Omega C} + \eta\, \delta_C\, input_\Omega$

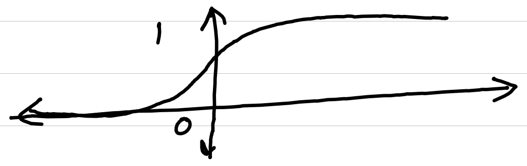
Activation functions

Activation functions are mathematical functions that are applied to the output of a neuron in a network.

Introduce non linearity

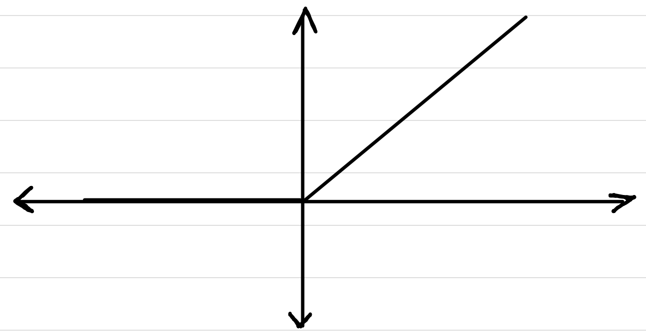
Common activation functions →

① Sigmoid : Maps output between 0 and 1



② Tanh : Maps output between -1 and 1

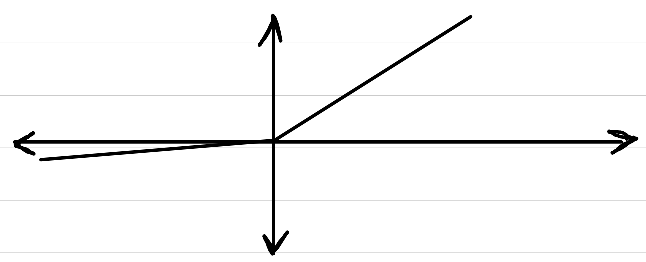
③ ReLU : Most commonly used
Easy to calculate
$\text{Relu}(x) = \text{Max}(0, x)$



④ Leaky relu :

$$f(x) \rightarrow \begin{cases} x & \text{if } x > 0 \\ \alpha x & \text{if } x < 0 \end{cases}$$

$\alpha \rightarrow$ small positive constant (eg 0.01)



In standard ReLU all negative units are mapped to 0.

During training if a neuron consistantly receives negative inputs, its gradient becomes zero, preventing weight updates. Such neurons become "dead" and stop contributing to learning

Leaky ReLU addresses this problem by allowing a small slope for negative inputs, ensuring that gradients still flow and neurons continue to learn

⑤ Softmax → Used in last layer for classification tasks

Without activation functions, N.N. collapses into a linear regression

Proof $\rightarrow$ Consider a neural network with k hidden layers each with activation function $\sigma_1\ \sigma_2 \ldots \sigma_k$

$$\hat{y} = \sigma_1(W_1 \sigma_2(W_2 \sigma_3 \cdots (W_k \sigma_k(x)) \cdots)$$

If activation functions are not used, ie $\sigma_i(x) = x \ \forall i$

$$\hat{y} = W_1 W_2 W_3 \ldots W_k \ x$$

$$\hat{y} = W' x$$

which becomes a linear regression problem.

Hence $\sigma(x) = x$ should not be used as a activation function in any layer of the network

Activation functions introduce the necessary non linearity in the model

Regularization

① Introduce a loss term that penalizes the squared magnitude of all parameters
That is, for every w, introduce a penalty

→ $\lambda \|w\|_2^2$ for L2

This is same as

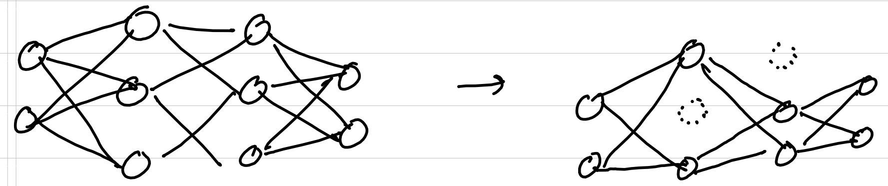
$$\mathcal{L}oss = \mathcal{L}oss + \frac{\lambda}{2}\|w\|_2^2$$

$$w^{t+1} \leftarrow w^t - \eta \nabla_\theta \mathcal{L} - \lambda w^t$$

if $\lambda < 1$, it decays the weight

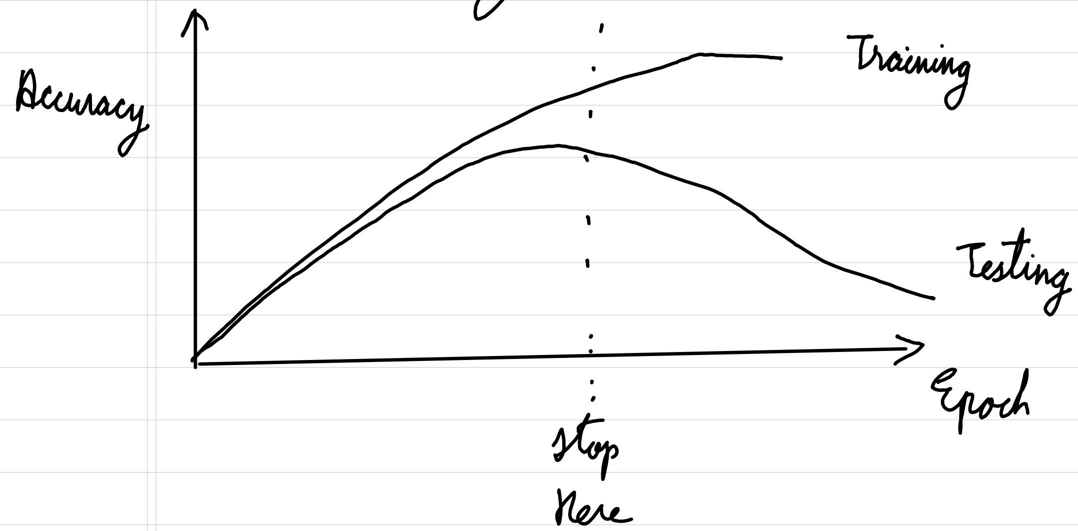
→ for L1 $\lambda \|w\|$

② Dropout → at some iterations, dont consider some neurons



Dont train every weight everytime

③ Early stopping



stop when the validation loss does not improve for a certain number of epochs

The number is called patience

Batch Gradient Descent

Uses all training samples (all N) in an epoch. Then update the model weights once per epoch

$$W \leftarrow w + \eta \frac{1}{N} \sum_{i=1}^{N} \frac{\partial \mathcal{L}(x_i; y_i)}{\partial w}$$

$(x_i, y_i) \rightarrow i^{th}$ training sample

$N \rightarrow$ Total no of training samples.

For epoch in numepochs:

    For all $x_i \in D$

        $\hat{y}_i \leftarrow$ forward (x_i)

        loss$_i \leftarrow$ loss $(y_i, \hat{y}_i)$

        ∇loss$_i \leftarrow$ Gradient (loss$_i$)

    ∇loss avg $\leftarrow$ average (∇loss$_i$)

    $W \leftarrow W + \eta \nabla$loss avg

Advantages

1. Accurate gradient direction
2. No randomness

Works well for small datasets

Disadvantages

1. Very slow for large datasets
2. Can get stuck in local minima

Note $\rightarrow$ Batch uses the true gradient direction

Stochastic Gradient descent

Updates weights after each individual training sample (x_i, y_i)

However this is a biased estimate since we are moving in the direction of expected slope and not the true direction

For epoch in numepochs:

$D \leftarrow$ shuffle (D)

For all $x_i \in D$

$\hat{y}_i \leftarrow$ forward (x_i)

$loss_i \leftarrow$ loss $(y_i, \hat{y}_i)$

$\nabla loss_i \leftarrow$ Gradient $(loss_i)$

$W \leftarrow W + \eta \nabla loss$.

Advantages →

1) Can escape local minima due to noisy updates

Disadvantages →

1) High variance in updates
2) Noisy convergence (zig zag path)
3) May never converge exactly - oscillate around minimum
4) Sensative to learning rate

Mini Batch Gradient descent

Middle ground between batch & stochastic

Most commonly used.

For epoch in numepochs:

Partition D into m minibatches randomly

For all $K \in (0, 1, \ldots m)$

For all $x_i \in D_K$

$\hat{y}_i \leftarrow \text{forward}(x_i)$

$\text{loss}_i \leftarrow \text{loss}(y_i, \hat{y}_i)$

$\nabla \text{loss}_i \leftarrow \text{Gradient}(\text{loss}_i)$

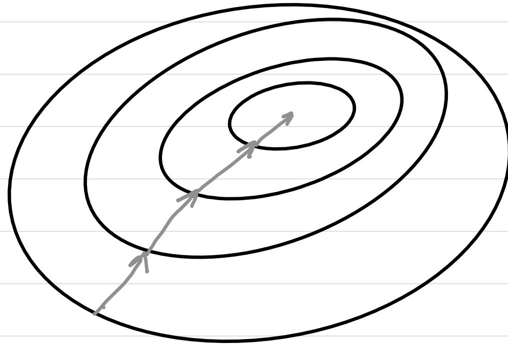
$\nabla \text{loss}_{avg} \leftarrow \text{average}(\nabla \text{loss}_i)$

$W \leftarrow W + \eta \nabla \text{loss}_{avg}$

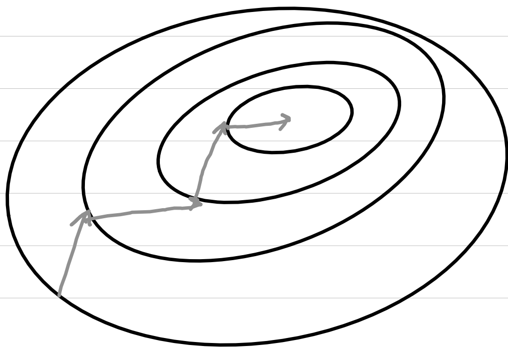
Advantages:

1) Balances speed & stability
2) $x_i \in D_n$ can be processed parallely
3) Implicit regularization due to small batch noise

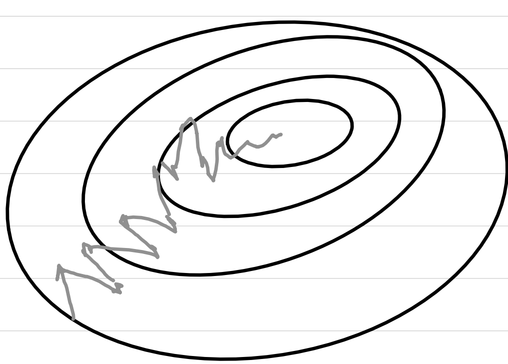
Batch
G.D.



Mini Batch
G.D.



stochastic
G.D.



Batch Normalization

In deep neural networks, as you go through many layers, the distribution of inputs to each layer can change during training

This is called "Internal Covariate shift"

If each layers see inputs that keep changing their "scale" or "mean", its harder for the layer to learn

This slows down learning.

Hence batch normalization is used

Normalize the inputs of each layer so that they have mean 0 and standard deviation of 1

Sometimes learnable scaling & shifting parameters can be used.

Parameter Initialization

We can initialize initial biases = 0,
weights $\sim N(0, \sigma^2)$

① If σ^2 is tiny (eg 1e-5) then in a forward pass, deeper layer outputs may decay towards 0

Because each linear layer does a weighted average with tiny weights

If activation function is ReLU, then it even further cuts down weights

This is called as the vanishing gradient problem

② If σ^2 is large (eg 1e+5) then in a forward pass, deeper layers outputs may grow towards ∞

This is called as exploding gradient

These problems can happen even when the σ^2 isn't too small or large

It is difficult to tune σ correctly