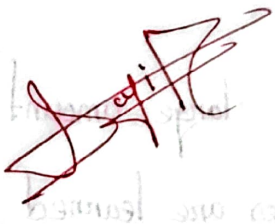
What is Deep Learning

Deep learning is a part of machine learning that uses artificial neural networks with multiple layers to learn pattern from data.

Key idea

Instead of humans designing features manually, deep learning model automatically discover the best features during training. This makes them powerful for solving non linear data as well.

Importance



(i) Powers modern AI applications
(ii) Works well with large dataset
(iii) Improve with better hardware like gpu's.

Applications

(i) Image recognition
(ii) Natural language processing.
(iii) Speech recognition
(iv) Recommandation systems etc.

Difference between ML and DL.

ML

(i) Works well with smaller dataset

(ii) Features are manually design by human's.

(iii) Use methods like decision tree, SVM, regression etc.

(iv) Requires less computational power.

(v) Great explainability

DL

(i) Needs large amount of data.

(ii) Features are learned automatiely by models

(iii) Uses multi-layer neural networks.

(iv) Needs GPUs / TPUs for training

(v) Less explainability

Perceptron

=> Simplest artificial neural networks.

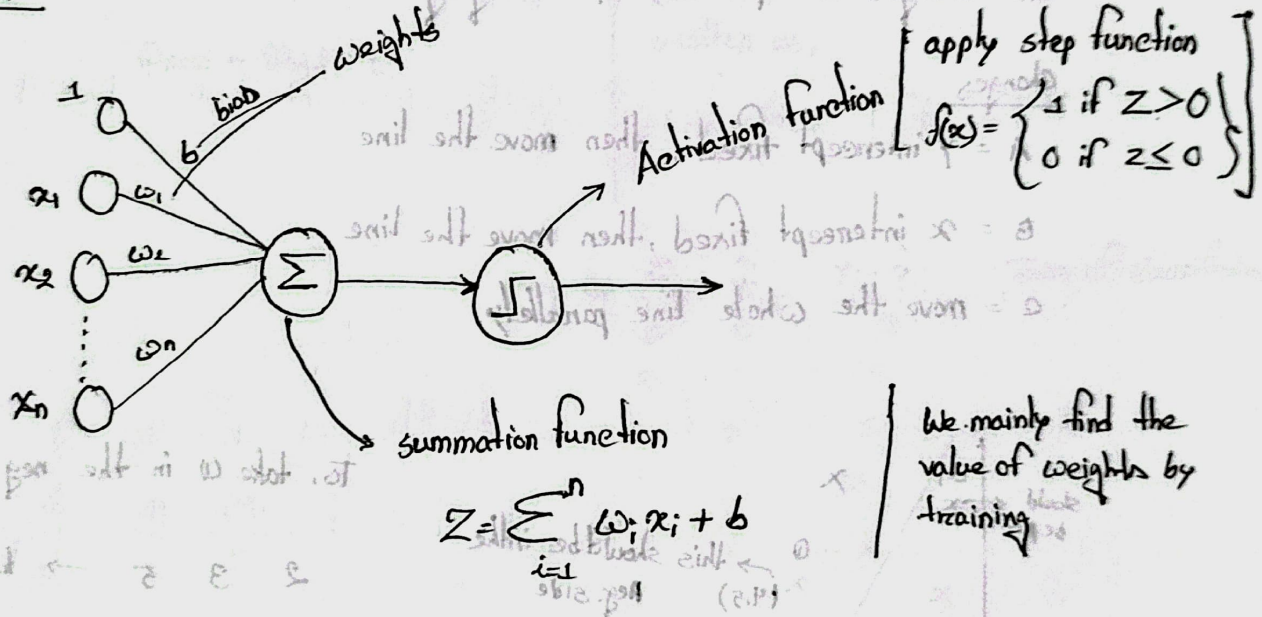
=> its a binary classifier

=> learns using a supervised learning rules

=> works if a linear boundary exists (AND, OR)

=> Cannot solve problems that are not linearly seperable. (XOR)

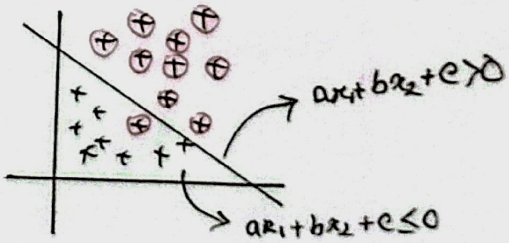
Diagram



For, 2D, it will create a line which separate the classes.

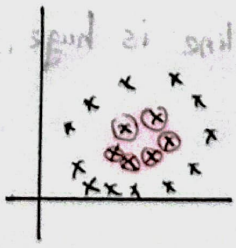
For, 3D, → Plane

Above, → hyperplane

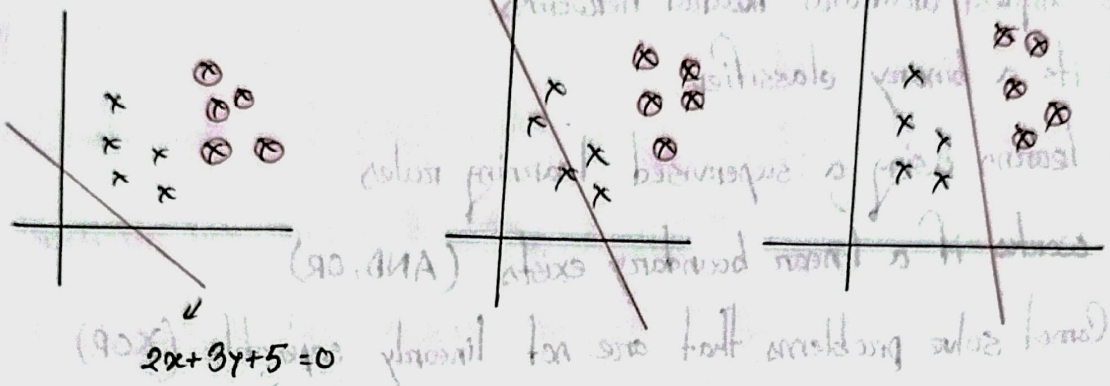


For, these kind of graph, not ideal,

Not linearly separable.



How it creates the line



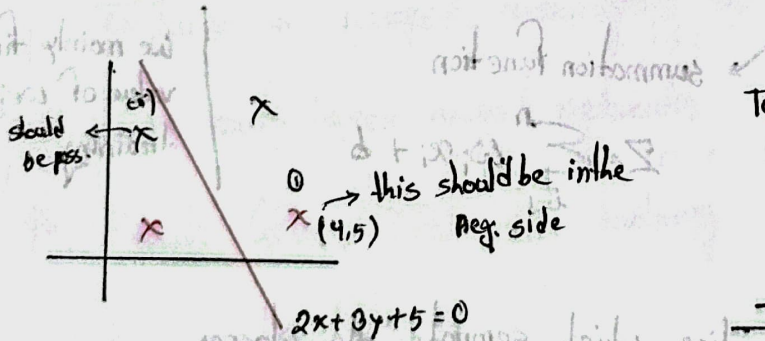
Line changes its position by changing the value of A, B, and C.

Changes

A = y intercept fixed, then move the line

B = x intercept fixed, then move the line

C = move the whole line parallely.



To, take (1) in the neg,

2 3 5 → line coff.

4 5 1 → coordinat and extra 1

− − − −

−2 −2 4

new line,

−2x−2y+4 =0

For, (1) add, instade of subtract.

The changes of line is huge, so we use learning rate (η)

Algorithm

epoch → 1000 ; $\eta = 0.01$

for in in range (epoch) :

take a random row data.

if ($\underbrace{x_i \in N}_{y_i}$ and $\underbrace{z \geq 0}_{\hat{y}_i}$) {

$\omega_{new} = \omega_{old} - \eta x_i$

}

elif ($x_i \in p$ and $z \leq 0$) {

$\omega_{new} = \omega_{old} - \eta x_i$

}

$\omega = [\omega_0, \omega_1, \omega_2]$

a A B

x_i = class

η = learning rate

total if and elif part can be written as,

$\omega_n = \omega_0 + \eta (y_i - \hat{y}_i) x_i$

y_i	$\hat{y}_i$	$y_i - \hat{y}_i$	Term of classification
0	0	0	
0	1	-1	
1	0	1	
1	1	0	

Loss function (Hinze)

$$L = \frac{1}{n} \sum_{i=1}^{n} \max(0, -y_i f(x_i))$$

n = number of sample
y_i = actual output
$f(x_i) = z$ = predicted output
$z = \omega_0 + \omega_1 x_1 + \omega_2 x_2$

We need to find such a line which's Loss function value will be minimum.

$$L = \underset{\omega_1, \omega_2, b}{\operatorname{argmin}} \frac{1}{n} \sum_{i=1}^{n} \max(0, y_i f(x_i))$$

To do that, we will use gradien descent.

Explanation of Loss function

for, 2 sample, the Loss function will look like,

$$L = \frac{1}{2} \left[\max(0, -y_1 f(x_1)) + \max(0, -y_2 f(x_2)) \right]$$

Data		
x_1	x_2	y
x_{11}	x_{12}	y_1
x_{21}	x_{22}	y_2

① $\max(0, -1 \cdot 1) = -ve = 0$

② $\max(0, -1 \cdot (-1) = +ve$

③ $\max(0, +1 \cdot 1 = +ve$

④ $\max(0, (-1 \cdot -1)) = +ve = 0$

y_i	$\hat{y}_i$		
1	1	①	→ connect
1	-1	②	
-1	1	③	
-1	-1	④	→ connect

0 = -1

For, connect prediction, contribute = 0
or else, atleast someting.

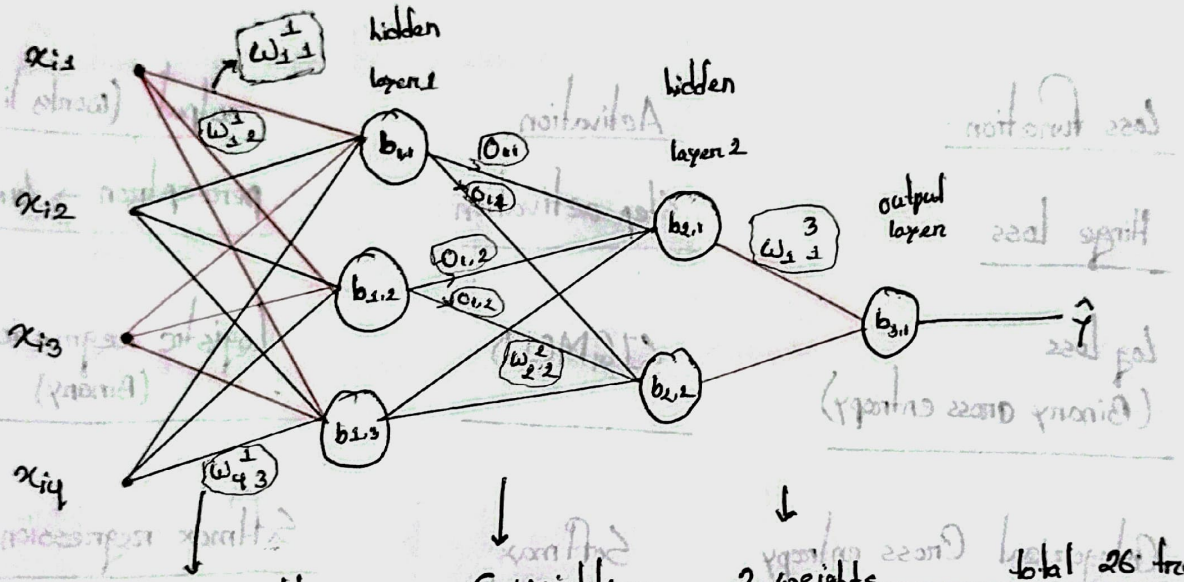
Benifites

Perceptron is very flexible. We can change or modify loss function and activation function for different purpose.

Here are some of the combination

Loss function	Activation	output (works like)
Hinge Loss	Step activation	perceptron → binary classi.
Log Loss (Binary cross entropy)	SIGMOID	Logistic regression (Binary)
Categorical Cross entropy	Softmax	Softmax regression (multi classi.)
Mean Squared error (MSE)	Linear (nothing)	Linear regression

Multi Layerd perceptron (MLP) Notation



Notations

Bias $\Rightarrow b_{ij}$

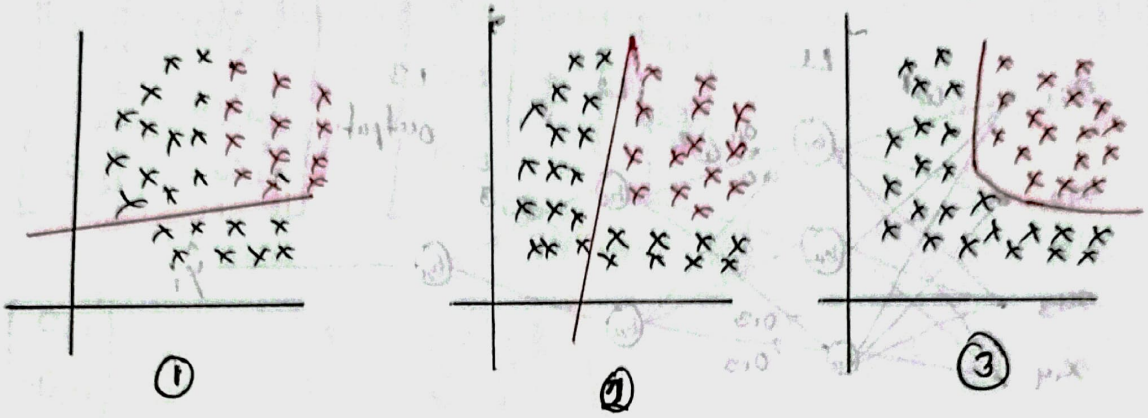
Output $\Rightarrow O_{ij}$

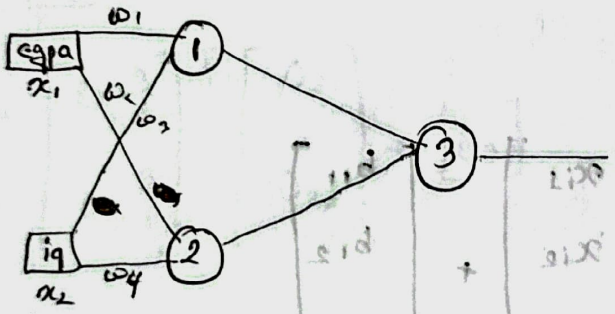
Weights $\Rightarrow \omega_{ij}^{K}$

i = layer no.
j = node no

K = entering layer no.
i = previous layer node no.
j = entering layer node no.

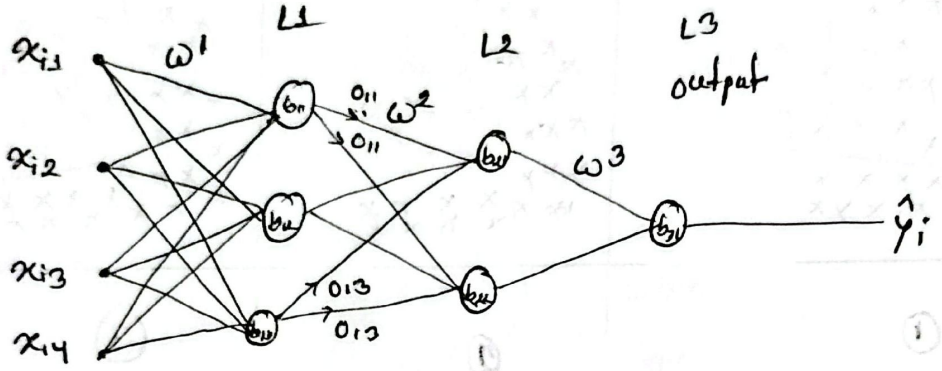
MLP intuition





So, It works great for non linear classification by multiple linear combination perceptrons. By increasing hidden layer and adjust loss function and activation function, complex non linear dataset can be classified.

Forward Propagation



Total parameter - 26

Perceptron $= \sigma(\omega^T x + b)$

Layer-1

$$= \sigma \begin{bmatrix} \omega^1_{11} & \omega^1_{12} & \omega^1_{13} \\ \omega^1_{21} & \omega^1_{22} & \omega^1_{23} \\ \omega^1_{31} & \omega^1_{32} & \omega^1_{33} \\ \omega^1_{41} & \omega^1_{42} & \omega^1_{43} \end{bmatrix}^T \begin{bmatrix} x_{i1} \\ x_{i2} \\ x_{i3} \\ x_{i4} \end{bmatrix} + \begin{bmatrix} b_{11} \\ b_{12} \\ b_{13} \\ [redacted] \end{bmatrix}$$

$(4\times3)^T \rightarrow (3\times4) \quad (4,1)$

(3×1)

$+ b$

$$= \begin{bmatrix} o_{11} \\ o_{12} \\ o_{13} \end{bmatrix}$$

Layer 2

$$\sigma \begin{bmatrix} \omega_{11}^2 & \omega_{12}^2 \\ \omega_{21}^2 & \omega_{22}^2 \\ \omega_{31}^2 & \omega_{32}^2 \end{bmatrix}^T \begin{bmatrix} o_{11} \\ o_{12} \\ o_{13} \end{bmatrix} + \begin{bmatrix} b_{21} \\ b_{22} \end{bmatrix}$$

$$= \begin{bmatrix} o_{21} \\ o_{22} \end{bmatrix}$$

Layer 3

$$\sigma \begin{bmatrix} \omega_{11}^3 \\ \omega_{21}^3 \end{bmatrix}^T \begin{bmatrix} o_{21} \\ o_{22} \end{bmatrix} + \begin{bmatrix} b_{31} \end{bmatrix}$$

$$= \hat{y}_i$$

Loss Functions

Loss Function is a method of evaluating how well your algorithm is modelling your dataset.

last activation → linear

① Mean squared error (MSE) or L2 loss

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

y_i = actual output
$\hat{y}_i$ = predicted output

$$\text{cost function} = \frac{1}{n} \sum_{i=1}^{n} (y_i - \hat{y}_i)^2$$

if no outliers

Advantages
- => Easy to interpret
- => Differentiable
- => 1 local minima

Disadvantages
- => Error unit (squared)
- => Not robust to outliers.

② Mean Absolute error (MAE) → L1 loss

$$L = |y_i - \hat{y}_i|$$

$$c = \frac{1}{n} \sum_{i=1}^{n} |y_i - \hat{y}_i|$$

if outlier

=> Robust to outlier

3. Huber loss

$$L = \begin{cases} \frac{1}{2}(y-\hat{y})^2 & \text{for } |y-\hat{y}| \le \delta \\ \delta|y-\hat{y}| - \frac{1}{2}\delta^2 & \text{otherwise} \end{cases}$$

(δ → hyperparameter)

If dataset have outliers, it will act like MAE, otherwise MSE.

④ Binary Cross Entropy or log loss

For binary classification

last Activation → Sigmoid

$$L = -y \log(\hat{y}) - (1-y)\log(1-\hat{y})$$

$$c = \frac{-1}{n} \sum_{i=1}^{n} y_i \log(\hat{y_i}) - (1-y_i)\log(1-\hat{y_i})$$

Ad.

* Differentiable

disad.

* multiple local minima
* not so intuitive

⑤ Categorical Cross Entropy

For, multiple classification

last activation → Softmax

$$L = -\sum_{j=1}^{k} Y_j \log(\hat{Y}_j)$$

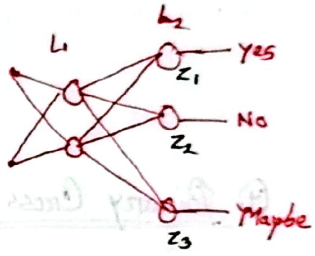
k = no. of classes in data

For, 3 class,

$$L = -Y_1 \log(\hat{Y}_1) - Y_2 \log(\hat{Y}_2) - Y_3 \log(\hat{Y}_3)$$

For that, target feature have to be one hot encoded.

Last layer neuron number = no. of classes



to avoid such thing,

We can use,

Sparse Categorical Cross Entropy.

It will handle internally and it is fast.

Softmax

$$f(z_i) = \frac{e^{z_i}}{\sum e^{z}}$$

$$f(z_1) = \frac{e^{z_1}}{e^{z_1} + e^{z_2} + e^{z_3}}$$

$$C = -\frac{1}{n} \sum_{i=1}^{n} \sum_{j=1}^{k} Y_{ij} \log(\hat{Y}_{ij})$$

- Regression
  - have outliers → MAE
  - No " → MSE
  - (linear)
- Classification
  - binary → BCE (Sigmoid)
  - Multi↓ → CCE
  - " ↑ → SCCE
  - (Softmax)

Backpropagation

Its a method used to train neural networks by sending errors backward through the layers to adjust the weights and bias by gradient descent to minimize loss.

Pre-requisite

- Complete understanding of Gradient descent
- Forward propagation

Algorithm and steps

```
for i in range (epochs):
    for j in range (x.shape[0]):
```

→ Select 1 row from data (random)

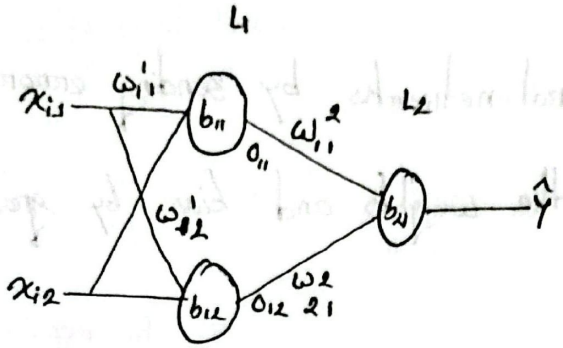
→ Predict output using forward propagation

→ Calculate logs using loss function

→ update weights a bias using G.D.

$$w_{new} = w_{old} - \eta \frac{\partial L}{\partial w}$$

$$b_{new} = b_{old} - \eta \frac{\partial L}{\partial b}$$



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

parameter = 9

Our main goal is to minimize the loss. The data inputs are constant. If I consider $\hat{y}$, it is effected by weights and biases. Loss is directely effected by $\hat{y}$ but if we want to change the $\hat{y}$ we have to adjust weights and bias.

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

If we notice this formula, $\frac{\partial L}{\partial \omega_{old}}$ is the partial derivative of all the ω and b. It adjust and update the values of ω and b. and find the best values of them.

Like, if Loss = $(y-\hat{y})^2$ and if it is the upper nural network, then, there are 9 parameter that we need to find and adjust.

$$\frac{\partial L}{\partial \omega^1_{11}}, \frac{\partial L}{\partial \omega^1_{12}}, \frac{\partial L}{\partial \omega^1_{21}}, \frac{\partial L}{\partial \omega^1_{22}}, \frac{\partial L}{\partial \omega^2_{11}}, \frac{\partial L}{\partial \omega^2_{21}}, \frac{\partial L}{\partial b_{11}}, \frac{\partial L}{\partial b_{12}}, \frac{\partial L}{\partial b_{21}}$$

These are the partial contribuation of weights and bias to $\hat{y}$.

Here, I calculate one,

$\frac{\partial L}{\partial \omega_{11}^2}$, this is Loss is not directly effect ω_{11}^2 rather, loss is effected by $\hat{y}$, then, $\hat{y}$ is effected by ω_{11}^2. It is like a chain.

That means,

$$\frac{\partial L}{\partial \omega_{11}^2} = \frac{\partial L}{\partial \hat{y}} \times \frac{\partial \hat{y}}{\partial \omega_{11}^2}$$

$$L = (y-\hat{y})^2$$
$$\hat{y} = o_{11}\omega_{11}^2 + o_{12}\omega_{21}^2 + b_{21}$$

Now, $\frac{\partial L}{\partial \hat{y}} = \frac{\partial}{\partial \hat{y}}(y-\hat{y})^2$

$$= -2(y-\hat{y})$$

$$\frac{\partial \hat{y}}{\partial \omega_{11}^2} = \frac{\partial}{\partial \omega_{11}^2}\left[o_{11}\omega_{11}^2 + o_{12}\omega_{21}^2 + b_{21}\right]$$

$$= o_{11}$$

So, $\frac{\partial L}{\partial \omega_{11}^2} = -2(y-\hat{y}) \cdot o_{11}$,

after, that, $\omega_{11(new)}^2 = \omega_{11(old)}^2 - \eta\left(-2(y-\hat{y}) \cdot o_{11}\right)$, new value of ω_{11}^2

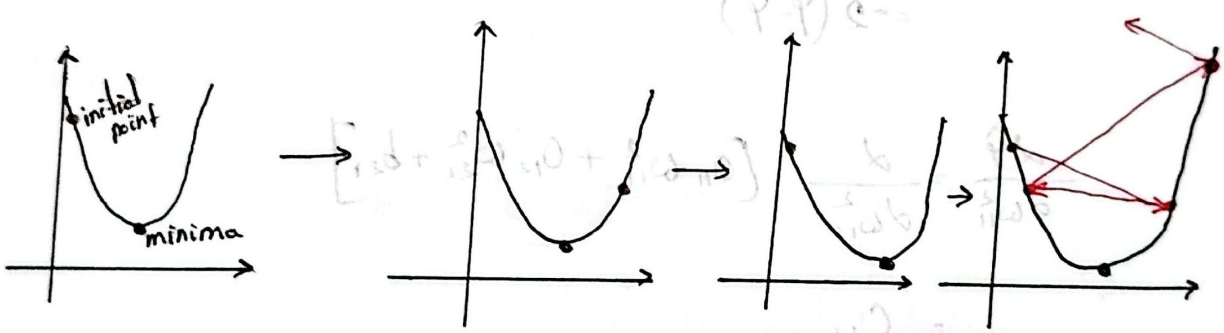
Similarly, all the other ω parameter is calculated, and it happens multiple iteration to reach the global minima.

Effect of learning rate

We introduce learning rate (η) on this formula just to get to the global minima gradually.

If, $\omega_{new} = \omega_{old} - \eta \frac{\partial L}{\partial \omega_{old}}$

learning rate is high, there is a posiblity that, we may skip the minima. We can observe a Z pattern in gradient deseent.



If, the learning rate is too less,

we may reach the minima, but it will take a lot of iteration. Means more computation and more time.

If, iteration is less, we will not reach the minima

Gradient Descent

GD is an optimization algorithm. In deep learning it is used to optimize millions of parameter (weights and bias) by minimizing a loss function.

Types

① Batch Gradient Descent

⇒ Uses the entire data set to compute the gradient before performing a weight update.

Advantages and disadvantages

⇒ Smoot and stable convergence

⇒ Slow + memory heavy.

⇒ Impractical for large dataset

② Stochastic GD (SGD)

⇒ Update parameter after each training example.

Ad. ans disad.

⇒ Can escape local minima

⇒ Updates are noisy

⇒ fast update

③ Mini batch GD

⇒ Splits dataset into small batches (e.g. 32, 64, 128) and update parameters per batch.

⇒ Efficient on Gpu

⇒ More stable then SGD and faster then BGD

⇒ Choosing batch size effects training efficiency.

Vanishing and Exploding Gradient

VG Gradient became extremely small as they propagate backward through layers.
Early layers learn very slowly or stop.

Why
⇒ Mostly happens if its a very deep network.
⇒ repeated multiplication of derivative < 1 (Sigmoid/tanh)
⇒ Slow or stop convergence.

Solution
⇒ Use Relu or variants
⇒ Use Batch normalization
⇒ Carefull weight initialization

EG
Oposite of vanishing Gradient.

Why
⇒ High learning rate
⇒ Impropor weight

Soution
⇒ Gradient clipping
⇒ Lower learning rate
⇒ Normalization layer

Fine Tuning Hyper-parameter

Layers and Neurons

more layer → capture complex patterns

more neurons → increase representation capasity (use more then sufficient)

Batch size

Preferable

⇒ Small batch size (e.g. 32, 64, 128) → better perform and generalization

⇒ large batch → faster but not so generalize. May cause ~~often~~ overfitting.

Number of epoch

⇒ Use early stopping to avoid overfitting

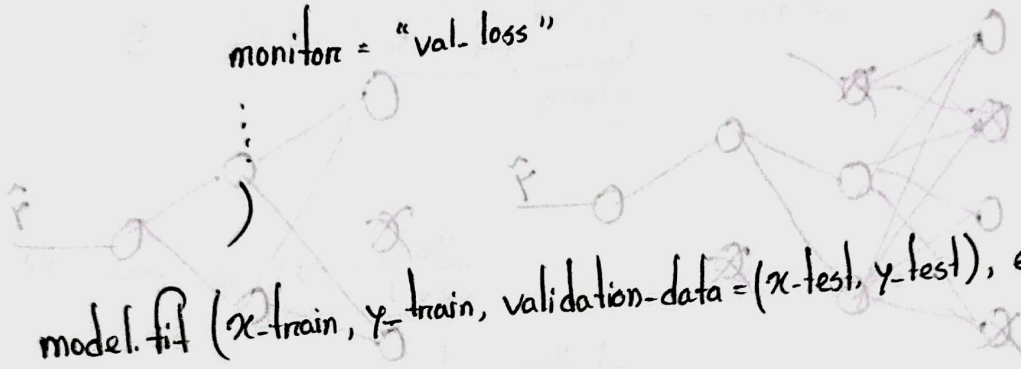
Early stopping

=> a regularization technique used to prevent overfitting. Training stoped before model starts overfitting.

=> Used to stop epoch before overfitting.

Code

```
callback = EarlyStopping (
            monitor = "val_loss"
            :
            )
```

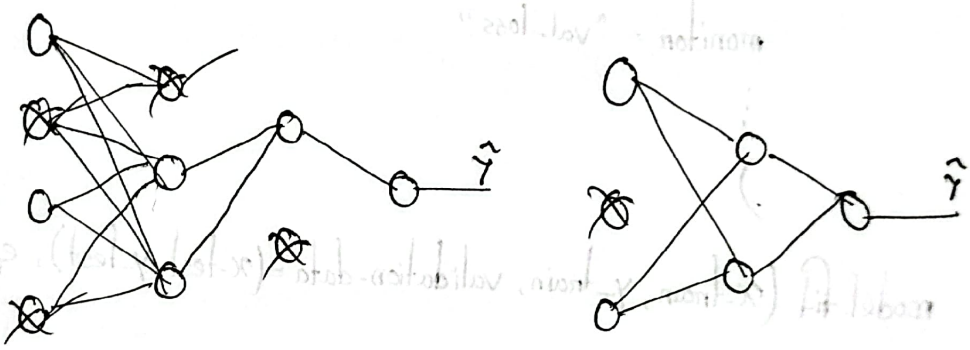


```
model.fit (x_train, y_train, validation_data = (x_test, y_test), epoch = 3000, callbacks = callback)
```

Dropout

Dropout is a regularization method used to reduce overfitting in neural network. It works by randomly droping or deactivate a subset of neurons during training forcing the network to learn redundant, robust representation.

→ forces each neurons to learn useful, general features independently.



If, dropout rate (d = 1-p) is too high, may tend to underfit.

Common dropout rate

ANN → 10 - 50%.
CNN → 40 - 50%
RNN → 20 - 30%.

```
from tensorflow.keras.layers import Dropout

model = Sequential([
    Dense(128, activation="relu", input_dim=),
    Dropout(0.2),
                                    ),
    Dense(
    Dropout(0.2),
])
```

Regularization

It's a set of technique used to reduce overfitting and improve generalization of machine learning and deep learning models. It introduces constraints or penalties to control model complexity, encouraging model to learn simple and generalize patterns. It reduce the values of weights and bring them closen to zero.

$$C = L(y, \hat{y}) + \underbrace{\frac{\lambda}{2n} \sum_{i=1}^{k} \|w_i\|^2}_{\text{panalty}} \longrightarrow L_2$$

For, L1, Panalty $\frac{\lambda}{2n} \sum_{i=1}^{k} \|w_i\|$

λ = hyperparameter = penalty strength.

We know, $w_n = \boxed{w_0 - \eta \frac{\partial L}{\partial w_0}}$

For, a sample, loss function will be,

$$L' = L + \frac{\lambda}{2} \sum \|w_i\|^2$$

Now, $\frac{\partial L'}{\partial w_0} = \frac{\partial L}{\partial w_0} + \lambda w_0$

$\therefore w_n = w_0 - \eta \left(\frac{\partial L}{\partial w_0} + \lambda w_0 \right)$

$= \underbrace{(1 - \eta\lambda)} \boxed{w_0 - \eta \frac{\partial L}{\partial w_0}}$ $\boxed{w \downarrow\downarrow}$

```
model = Sequential()
model.add(Dense(128, input_dim = 2, activation = "relu", kernal_regularizer = tensorflow.keras.
    regularizers.l2(0.005))
```

L2

Activation Function

It defines how the weighted and ~~biased~~ sum of input + bias is transformed before passing to the next layer in a neural network.

It introduce non linearity to the network.

$$z = w^T x + b$$

activation function, $a = f(z)$

~~Sigmoid~~

Ideal activation Function

1) Non linear
2) Differentiable
3) Computationally inexpensive
4) Zero centered (mean = 0) (tanh)
5) Non-saturating (relu)

Sigmoid

$$f(x) = \frac{1}{1 - e^{-x}}$$

$$f'(x) = f(x)\,(1 - f(x))$$

range : (0, 1)

- Commonly used in output layer

Advantages

=> Smooth and differentiable

=> Usefull for probabilistic interpretation (Binary classification)

=> Non linear

Disadv

=> Vanishing gradient due to saturation

=> Not zero centered

=> Slows convergence

Hyperbolic Tangent (tanh)

$$f(x) = \tanh(x) = \frac{e^{x} - e^{-x}}{e^{x} + e^{-x}}$$

$$f'(x) = 1 - f(x)^2$$

Range : (-1, 1)

- Hidden layers in early RNN and MLPs.

Adv

=> Zero centered others like sigmoid

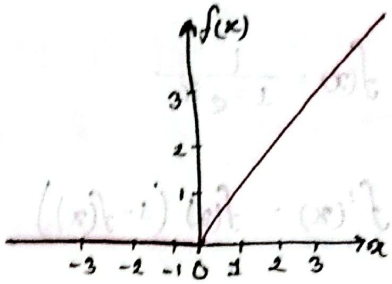
disadv

=> Vanishing gradient

Rectified linear unit (ReLU)

$$f(x) = \max(0, x)$$

$$f'(x) = \begin{cases} 1, & x > 0 \\ 0, & x \leq 0 \end{cases}$$



Advantages

=> Computational efficiency

=> Is non linear

=> Mitigates vanishing gradient

=> Introduce sparsity (Some neurons deactivate)

Disadv

Dying ReLU problem: neurons can inactive (output 0), if weight update incorrectly.

Non zero centered

Common

Hidden layers in ANN, RNN CNN, MLPs.

Dying Relu

It occurs when a neuron's output becomes 0 for all input and never recover during training. Means,

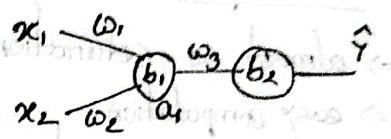
=> Neurons stop learning

=> Becomes inactive

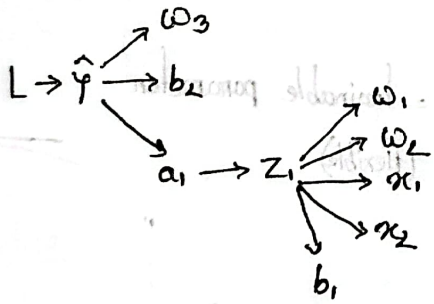
Why happens

During training, weights update.

$$\omega_n = \omega_0 - \eta \frac{\partial L}{\partial \omega_0}$$



Dependency



Relu

$a_1 = \max(0, z_1)$

main

$z_1 = \omega_1 x_1 + \omega_2 x_2 + b_1 < 0$

$\therefore a_1 = 0$

So, $\frac{\partial a_1}{\partial z_1} = 0$

For, $\frac{\partial L}{\partial \omega_1} = \frac{\partial L}{\partial \hat{y}} \frac{\partial \hat{y}}{\partial a_1} \frac{\partial a_1}{\partial z_1} \frac{\partial z_1}{\partial \omega_1}$

If, this derivation became zero, then there will be no weight update. That is when we will face dying relu.

When happens

=> Large learning rate

=> High neg. bias

Variation of Relu

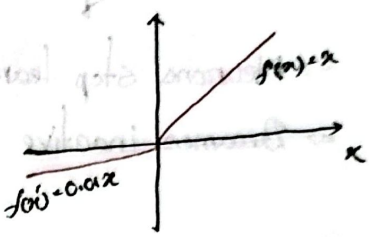
Most of them solve dying relu problem.

Leaky Relu

$$f(z) = \max(0.01 \cdot z, z)$$

(2)

Typically,
$\alpha = 0.01$
(reachened) value



activation = LeakyRelu(alpha=0.01)

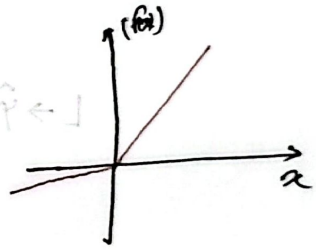
Advantages

⇒ almost no saturation
⇒ easy computation
⇒ closer to zero centered

Parametric Relu

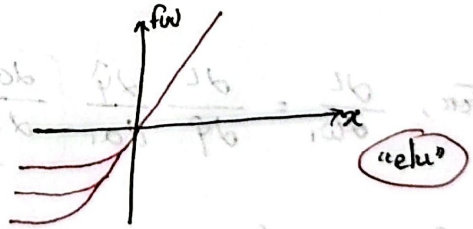
$$f(x) = \begin{cases} x & \text{if } x > 0 \\ ax & \text{if } x \leq 0 \end{cases}$$

a = trainable parameter (flexible)



Elu (Exponential Linear unit)

$$f(x) = \begin{cases} x & \text{if } x > 0 \\ \alpha(e^x - 1) & \text{if } x \leq 0 \end{cases}$$



Selu (Scaled Elu)

$$f(x) = \lambda \begin{cases} x & \text{if } x > 0 \\ \alpha e^x - \alpha & \text{if } x \leq 0 \end{cases}$$

$\alpha \approx 1.673$
$\lambda = 1.051$

"selu"

Weight initialization

What not to do,

1. All weights are zero
2. Are constants
3. Random small (-0.01 to 0.01)
4. Random large (-3 to 3)

What can be done

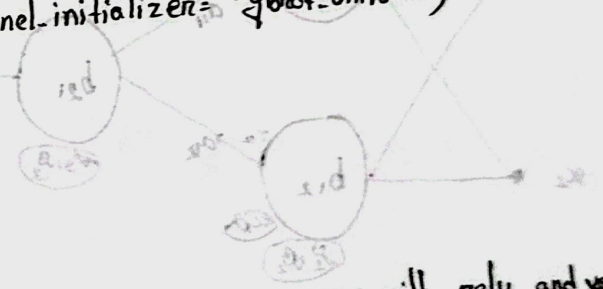
Xavier / Glorot initialize

np.random.randn(dim) * $\sqrt{\frac{1}{fan\_in}}$

fan_in = number of inputs coming to the node

use with (tanh), sigmoid

```
Dense(256, activation = 'tanh', kernel_initializer = 'glorot_uniform')
```



He initialize

use with relu and variant (selu) 'lecum_normal'

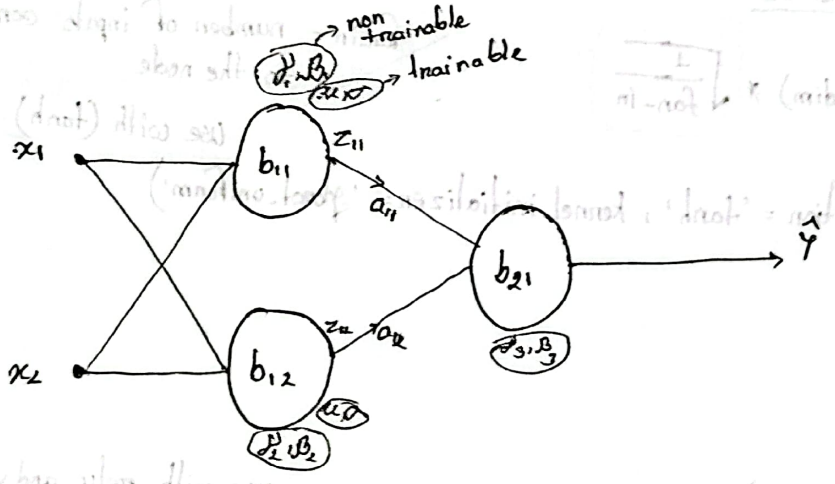
np.random.randn(dim) * $\sqrt{\frac{2}{fan\_in}}$

```
model = Sequential([
    Dense(256, activation = 'relu', kernel_initializer = 'he_normal')])
```

Batch Normalization

It is a technique to improve the training speed, stability, and performance of deep learning networks. It normalize the input of each layer so that they have a mean of 0 and variance of 1 within a mini-batch.

During training, as parameter update, the distribution of activations in each layer changes - it is called Internal Covariate shift.



For, each neuron,
γ, β → different
μ, σ → different

$z_{11} = \omega_{11}' x_1 + \omega_{21}' x_2 + b_{11}$

rest will be same process

Process

$$Z_{11} \rightarrow Z_{11}^{N} \longrightarrow Z_{11}^{BN} \longrightarrow g(Z_{11}^{BN}) \rightarrow a_{11}$$

$$z_{11} = \frac{z_{11} - \mu_B}{\sigma_B + \epsilon}$$ (m = batch size)

mean, $\mu_B = \frac{1}{m} \sum_{i=1}^{m} z_n$

variance $\sigma_B = \sqrt{\frac{1}{m} \sum_{i=1}^{m} (z_{11} - \mu_B)}$

$$Z_{11}^{BN} = \gamma Z_{11}^{N} + \beta$$

gives flexibility

Adv.

1) Stable — Wide range hyper parameters
2) Faster — higher learning rate
3) act like regularizer
4) Impact of weight initialization reduce

Code

```
model.add(Dense(3, activation='relu', input_dim=2))
model.add(BatchNormalization())
```

Optimizers

It's are algorithms used to update parameters of a neural network in order to minimize the loss function.

Till now, we use Gradient descent and its types.

There were many problem with these optimizers like,

1> Choosing right learning rate

2> get stuck in local minima

3> Saddle point where gradient $\frac{dL}{dw}$ is 0 or close

Optimizers that we will learn

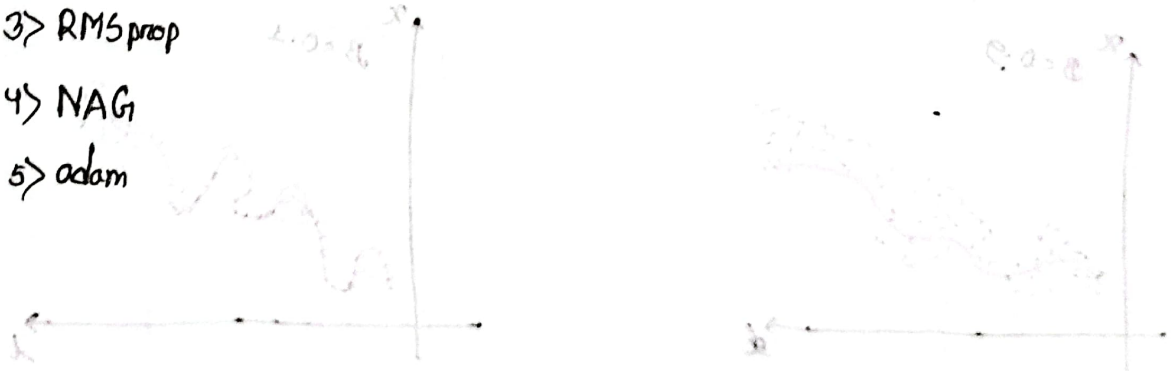
1> Momentum

2> AdaGrad

3> RMSprop

4> NAG

5> adam



Exponentially weighted moving avg (EWMA)

It's a technique to compute a smooth average of a sequence of numbers by giving more weight to recent observation and less to older ones.

$$V_t = \beta V_{t-1} + (1-\beta) x_t$$

V_t = current moving avg
V_{t-1} = previous " " "
x_t = current value
β = decay rate or smooth factor $(0 < \beta < 1)$

β determines how much weight is given to the past vs the present.

⇒ higher β (0.9) → smoother, slow react to present

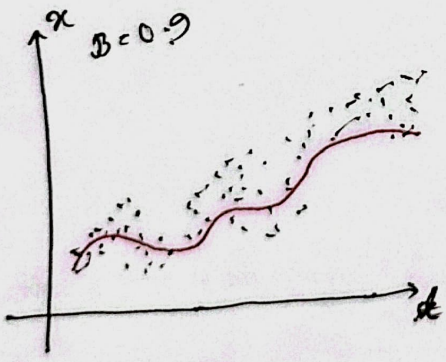
⇒ for lower (0.1-0.5) → opposite

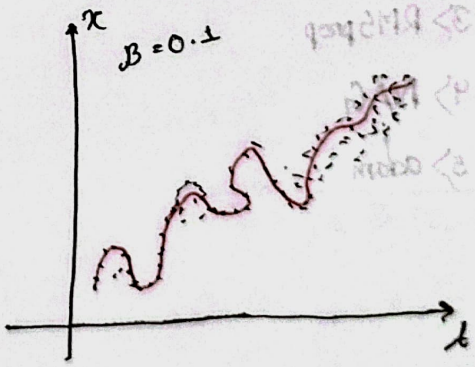
Day	x_t (temp)
D_1	43°
D_2	33°
D_3	26.5°
D_4	30.6°

Observation

$$V_t = (1-\beta)\left(x_t + \beta x_{t-1} + \beta^2 x_{t-2} + \beta^3 x_{t-3} + \cdots\right)$$

$\beta = 1 - \beta$






```
[x1 = df[' '].ewm(alpha = 0.9).mean()]
```

Momentum

An optimization technique used to accelerate gradient descent by taking account the past gradient.

It behaves like a ball rolling down a hill - gaining 'momentum' as it moves.

Gradient descent, $\omega_{t+1} = \omega_t - \eta \nabla \omega_t$

In momentum,

$$\omega_{t+1} = \omega_t - V_t$$

$$V_t = \beta V_{t-1} + \eta \nabla \omega_t$$

$\beta \rightarrow$ momentum coefficient normally (0.9)

Adv

1) faster convergence
2) Smooth updates
3) Escape local minima

Dis adv

1) May overshoot if β and η is too high
2) Requires hyper tuning

Nesterov Accelerated Gradient (NAG)

An improvement of Momentum. It adds a look ahead step - instead of computing the gradient at the current position, it looks ahead in the direction of the accumulated momentum, before computing the gradient.