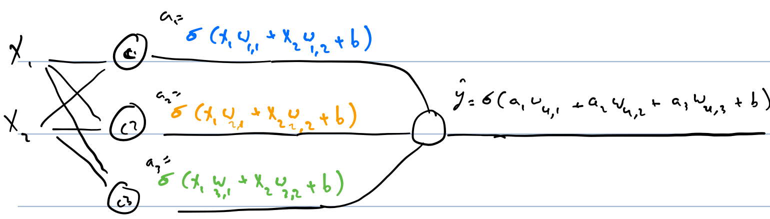
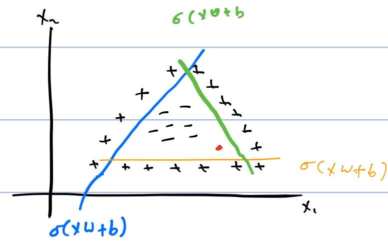




$c1 = 0.1$
$c2 = 0.4$
$c3 = 0.4$

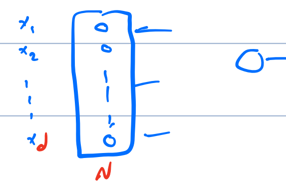
use them as features and input them to another classifier

- nn = learn non-linear boundry
- we use σ in training
- if you don't put non-linearity in hidden layer = you will have another Logistics Regression

2 layers

Standerd nn (FFNN, FCNN):- * of neurons = $dN + W$

- * of features = * of neurons
- many features = less * of neurons
- don't make any hiddin layer < output layer



* How to make prediction?

first neuron

$$z_1^{[1]} = w_{11}^{[1]} x_1 + w_{12}^{[1]} x_2 + \cdots + b_1^{[1]}$$

layer; first neuron second wieght

$$a_1^{[1]} = \sigma(z_1^{[1]})$$

first hiddin layer

$$\Rightarrow \vec{a}^{[1]} = \sigma(WX + \vec{b})$$

$$z^{[2]} = w_1^{[2]} \cdot a_1^{[1]} + w_2^{[2]} a_2^{[1]} + \cdots + b^{[2]}$$

$$\hat{y} = a^{[2]} = \sigma(z^{[2]})$$

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

How to training ?

backprobagation → to update the parameters

for any nn you will not get same accurecy every time

$$\frac{dJ}{dw_j^{[2]}} = \frac{1}{m} \Sigma (a^{[2]} - y) \cdot a_j^{[1]}$$

$$\frac{dJ}{db^{[2]}} = \frac{1}{m} \Sigma (a^{[2]} - y)$$

initailze w and b with Random [not zeros] → to get different gradients

$$\frac{dJ}{dw_j^{[1]}} = \frac{dJ}{dz^{[2]}} \times \frac{dz^{[2]}}{da^{[1]}} \times \frac{da^{[1]}}{dz^{[1]}} \times \frac{dz^{[1]}}{dw^{[1]}}$$

$$(a^2 - y) \times w^2 \times \sigma(z^{[1]})(1 - \sigma(z^{[1]})) \times$$

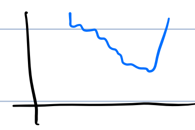
Loss function is not convex!

but

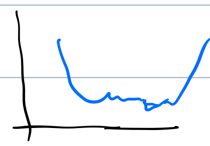
in highly dimention feature space :-

→ many local minimums

→ local minimums are close to global one



→ will not happened



→ what happened

GD for NN :-

1 → initial w, b with random (small)

2 → farward pass

z^1, a^1, z^2, a^2

3 → compute loss, J

4 → Backpropgation dw^2, db^2, dw^1, db^1

5 → update $w^1 \leftarrow$

$w^2 \leftarrow$

$b^1 \leftarrow$

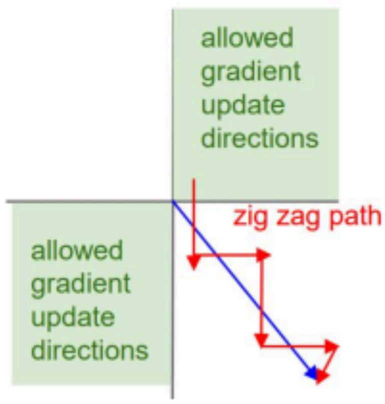
$b^2 \leftarrow$

repeat 2 → 5 until converge $|J^k - J^{k-1}| < \epsilon$

Three Problems with activation functions:-

- **Non zero-centered:** (Slower in converge)

  the problem of not having zero-centered activation function is slow convergence, the gradient will move in a zig-zag path. To illustrate the problem more let us see this picture:



The blue line is the optimal path, but the non zero-center will move like the red path (zig-zag, always move vertical or horizontal) which lead to slow convergence.

- **Saturation:** (no learning)

the problem with saturation is that the gradient will be zero and this kill the gradient since if the gradient is zero ($dW \; = \; 0$) then the update will be:

$$W = W - \; \alpha * dw = W - \alpha * 0 = W$$

We can see that the value of W does not change, this lead to make the network stop learning.

- **Computational Cost:**

the computational time for some activation function is more than other. For example, calculating the value of ReLU function is faster than sigmoid and tanh since both of them contain exponential calculation (e^x).

problem with sigmoid:- $\sigma(z) = \frac{1}{1+e^{-z}}$

- not zero centered   values [0, 1]   all positive (0.5 centered)
- graident saturate easily
- exponintal computation   derivative = $\sigma(x)(1-\sigma(x))$

ReLU:- $R(x) = \max(0, x)$

- less computation
- not zero centered
- gradiant not all saturate just negatives →

derivative (gradient)
$\begin{cases} 0 & \text{if } z \le 0 \\ 1 & \text{other} \end{cases}$

Tanh:-

- zero centered
- gradient saturated easily
- gradient → $1 - \tanh^2$

	ReLU	Tanh	Sigmoid
zero-centered	Not zero-centered	Zero-centred	Not zero-centered
Saturation	Dose not saturated	saturated	saturated
Computational Cost	efficient	slow	slow

Why we go deep

to avoid overfitting

being more efficient (less ~~#~~ of neurons in each layer)

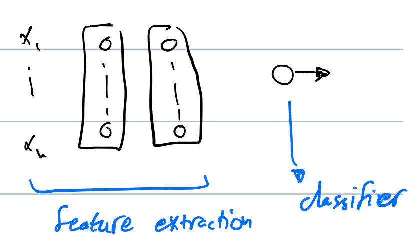
DL vs ML:-

Deep Learning   machine Learning

having more data ⇒ DL

having more feature ⇒ DL

there is no spesific features (images..) ⇒ DL



Structured data ⇒ ML

Prosperity	Machine Learning	Deep learning
Data structure	Structure data	Unstructured data
Size of the data	Could works on small or medium data	Required huge amount of data
Training time	Less training time	Large training time
Feature Extraction	Need to understand the features	Don't have to understand the feature, it could learn the features by itself
Accuracy	Less accuracy than deep learning	Higher accuracy than machine learning

\# binary loss cross-entropy for binary (Logistics)

\# Multiclass

Data:

convert y ⇒ one-hot-encoding

wider layers: overfitting

Deeper Layers less overfitting

Architecture:-

→ # neurons in output layer = # of classes

→ apply Softmax

→ loss ⇒ caterogal cross-entropy $\left(\sum_{i=1}^{K} -y \log(\hat{y})\right)$

\# of classes

TARGET	PREDICTION
1	0.5
0	0.3
0	0.2

for i=1

for multi-label classification:
outputs # of classes
+
sigmoid

What we change

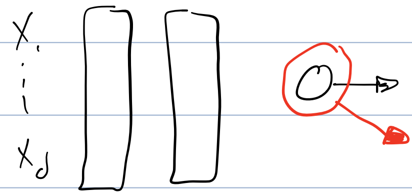
- # of output layer
- activation of output layer
- Loss

$$\text{SoftMax}(X) = \frac{e^{x}}{\sum e^{i}}$$

all output nurons

Regression by nn

Predict the age of image



no activation
$z = wx + b$

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

MSE

$$Cost = \frac{1}{2m} \sum (\hat{y} - y)^2$$

What we change
- activation in the output layer
- loss

big data ⟹ 1 step is costly and may out of memory

Solutions

Gradient decent:-

you have to do FP / bp / update for all samples to do just one step → costly + out of memory

Stochastic GD:

you have to do FP / bp / update for just one sample to do just one step → noisy

mini-batch GD:-

you have to do FP / bp / update for just one batch to do just one decent → the best

batch size : 16, 32, 64

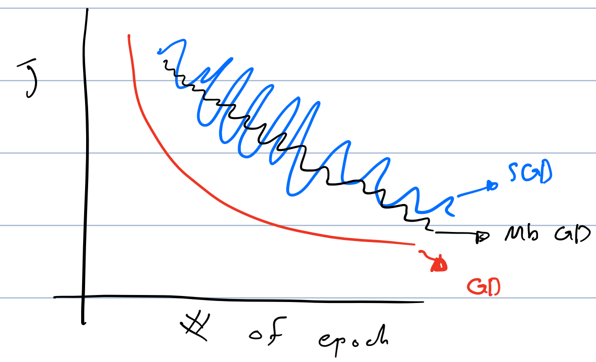
# of samples

Example:- m = 10000 and we loop on m

do GD ⇒ 1 step

do SGD ⇒ 10000 steps

do Mb GD ⇒ 100 steps
batch size = 100



epoch ⇒ cover all samples

Optimizers to optimize the update step

insted of ($w_i = w_i - \alpha \, dw_i$)

GD with momentum :- to have a capability to escape local minimal (Ball)

GD with RMSprop:- to have faster GD

Adam conpine GDM and RMSprop

Dealing With Underfitting (Bias)

- A more complex model: More neurons, layers.
- Train longer
- More features

Dealing With Overfitting (Variance)

- More data
- L_p regularization
- Dropout
  - Decide dropout rate {0.5, 0.2,…}
  - Set some activations to be 0 and rescale all the remaining activations
  - Only during training
- Data augmentation
- Early stopping