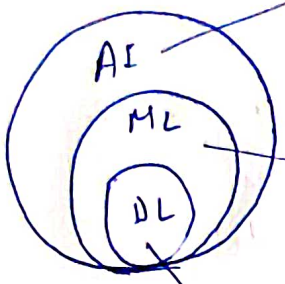
AI

Smartknower

Basics

* AI vs ML vs DL



→ computers that can imitate human intellect & behaviour

→ Statistical algorithm that enables AI implementation through data

→ subset of ML which follows neural networking

* ML

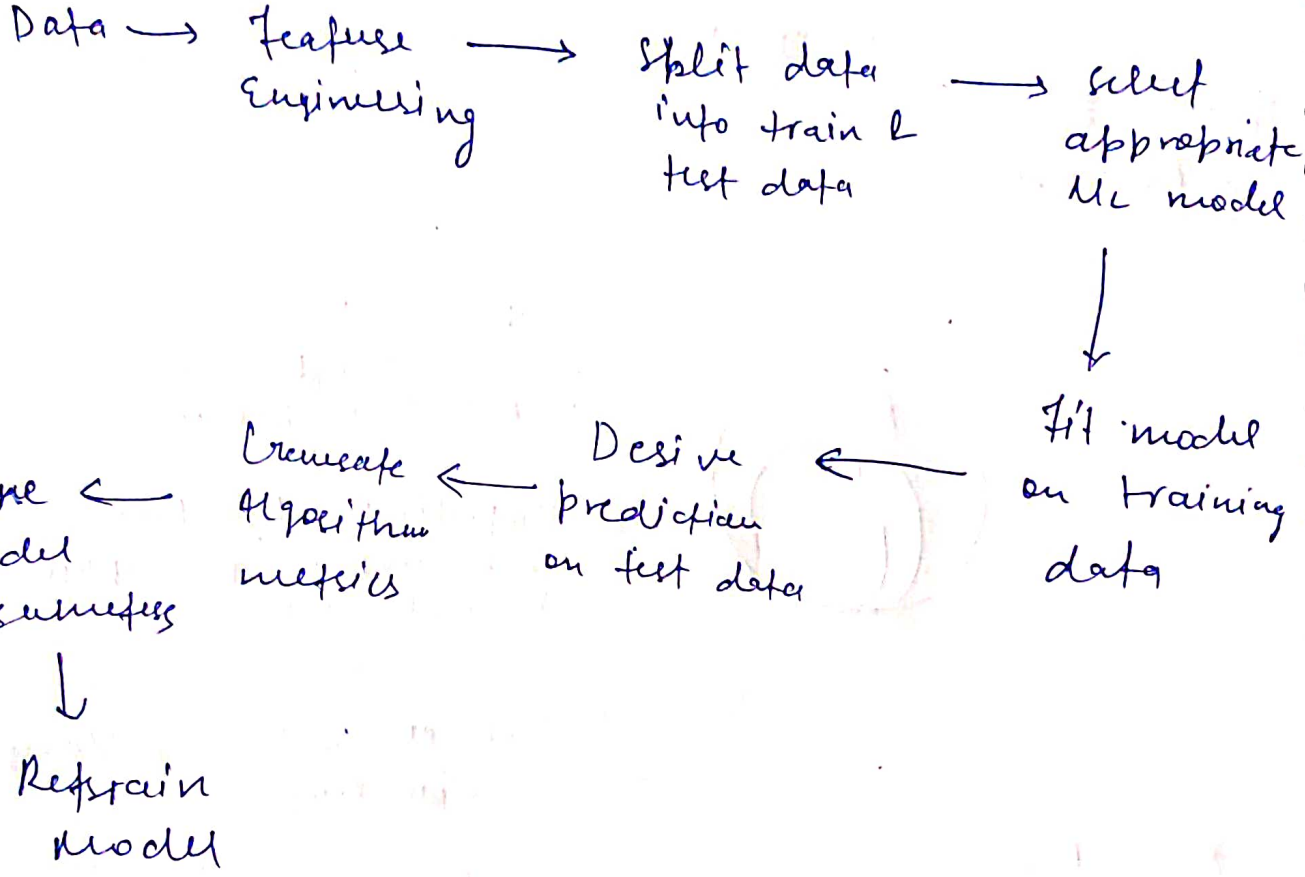
Supervised

- Classification
  - Logistic
  - Decision tree
  - Random forest
  - KNN
  - SVM
  - Naive Bayes
- Regression
  - simple linear reg.
  - multiple linear reg.
  - Ridge Reg
  - Lasso Reg

Unsupervised

- K-means Clustering
- Hierarchical clustering

* Prediction pipeline



* Application of AI

- Image classification
- Image De-noising
- Object detection
- Chatbot
- Pose estimation
- Stock price prediction
- Image segmentation
- Text Summarization
- Language translation

* Teachable machine

search internet

* Working of a neuron

- A neuron accepts a signal and sends a message to the brain to act in a particular way.
- It accepts a signal, generates a particular response in correspondence from its nucleus and then triggers a response.
- The response may pass to another neuron and the cycle proceeds in this manner.

* Neural Networks

① - Neurons
② - Weights
③ - Bias
④ - Synapse
⑤ - Architecture of a neural network
⑥ - Layers in NN
    a) Input layer
       Hidden layer
       Output layer
⑦ - Neural equation of a Neural network
⑧ - Activation functions
    a) Linear
    b) Sigmoid
    c) Relu
    d) Leaky relu
    e) Tanh
    f) Softmax
⑨ - Optimizers & Optimization Techniques
    a) Gradient descent
    b) Stochastic Gradient descent (SGD)
    c) Mini batch Gradient descent

d.) SGD with momentum
e.) Adagrad
f.) Adadelta
g.) RMS Prop
h.) Adam

(10) - Loss function -
a.) MSE
b.) MAE
c.) Binary Crossentropy
d.) Categorical Crossentropy
e.) Sparse - Categorical Crossentropy

(11) - Forward Propagation

(12) Backward Propagation

(13) Epochs

(14) Classification Metrics - Accuracy, Precision, Recall, F1 - Score

(15) Gradient descent from scratch (without tensorflow)

Neural network working

- Neural networks are a set of algorithms, modeled loosely after human brain, that are designed to recognize patterns. They interpret data through a kind of machine perception and process input data to solve a certain problem. The patterns they recognize are numerical, contained in vectors, into which all-real world data i.e text, images, sound can fit.

- Neural networks are a class of machine learning algorithms used to model complex patterns in datasets using multiple hidden layers & activation functions. A neural network takes an input, passes it through multiple layers hidden (mini-functions with unique coefficient that must be learned), and outputs a prediction representing the combined input of all the neurons.

- Neural networks are trained iteratively using optimization techniques like gradient descent. After each cycle of training, an error metric is calculated based on the difference between prediction and target. The derivatives of this error metric are calculated & propagated back through the network using a technique called backpropagation. Each neuron's coefficient (weights) are then adjusted relative to how much they contributed to the total error. This process is repeated iteratively until the network error drops below an acceptable threshold.

- In a neural network, each neuron (except those in the input layer) is actually a sum of all its inputs: which are in fact the outputs from the previous layer multiplied by some weights. An additional term called bias is added to this sum. And a non linear known as activation function is applied to the result.

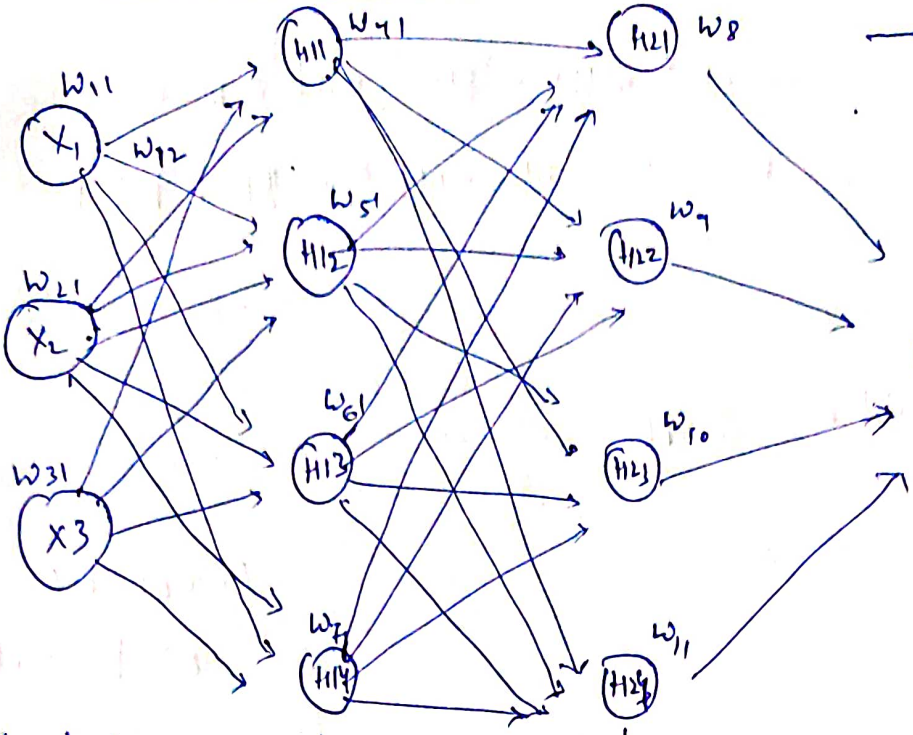
* Layers in Neural Network

(1) - Input layer - Holds the data your model will train on. Each neuron in the input layer represent a unique attribute in your dataset (e.g. height, hair color, etc).

(2) - Hidden layer - Sits b/w the input & output layers and applies an activation function before passing on the result. There are often multiple hidden layers in a network. In general, hidden layers are typically full-connected layers - each neuron receives input from all the previous layer's neurons & sends its output to every neuron in the next layer.

(3) - Output layer - The final layer in a ~~network~~ network. It receives inputs from the previous hidden layer, Optionally applies an activation function, and returns an output representing your model's prediction.

*



$x = x1, x2, x3$

$$Err = \left(\frac{1}{n}\right) * sum(yi - ypredi)^2$$

Total observation

W_{11} = weight

bias (constant term)

$$H11\_in = x1 * w11 + x2 * w21 + x3 * w31 + b11$$

$$H12\_in = x1 * w12 + x2 * w22 + x3 * w32 + b12$$

Input

$$H11\_out = Activation(H11\_in)$$

$$H12\_out = Activation(H11\_in)$$

output

$H21\_in = H11\_out * w41 + H12\_out * w51 + H13\_out * w61 + H14\_out * w71 + b21$

- The updated graph till now

→ forward propagation

till now

Iteration - 1

- ~~cost~~ errors also generating here
- weights & bias are the entities which is going to update in every iteration to minimise errors by backward propagation.
- how weights & bias get updated

Optimization technique.

① - Gradient descent

$$w' = w - alpha * d(Err)/d(w)$$

- w8, w9, w10, w11 get updated first

- we add bias in case weights are 0, it will have no impact. so bias helps in either shifting of Activation func. to left or right.
- Bias can be zero

*

Neuron - A neuron takes a group of weighted inputs applies an activation function, and returns an output. Inputs to a neuron can either be features from a training set or outputs from previous layer neurons.

Weight - Weights are applied to the inputs as they travel along synapse to reach the neuron & a bias term is added. The neuron then applies an activation function to the "sum of weighted inputs added to the bias" from each incoming synapse & passes the result on to all the neurons in the next layer.

Bias - Bias helps in controlling the value at which activation function will trigger. Bias value allow you to shift the activation func. to either left or right.

Activation function -

① - An Activation function is a neural network defines how the weighted sum of the input is transformed into an output from a node or nodes in a layer of the network.

② - An activation function is a function that is added into an artificial neural network in order to help the network learn complex patterns in the data.

③ - Activation function helps to normalize the output of any input in the range b/w 1 to -1 or 0 to 1

④ decides what is to be fired (activated) to the next neuron.

Synapse - Synapse are like roads in a neural network. They connect inputs to neurons to neurons, & neurons to outputs.

Optimizer - (1) - It's very important to tweak (change) the weights of the model during the training process, to make our predictions as correct & optimized as possible.

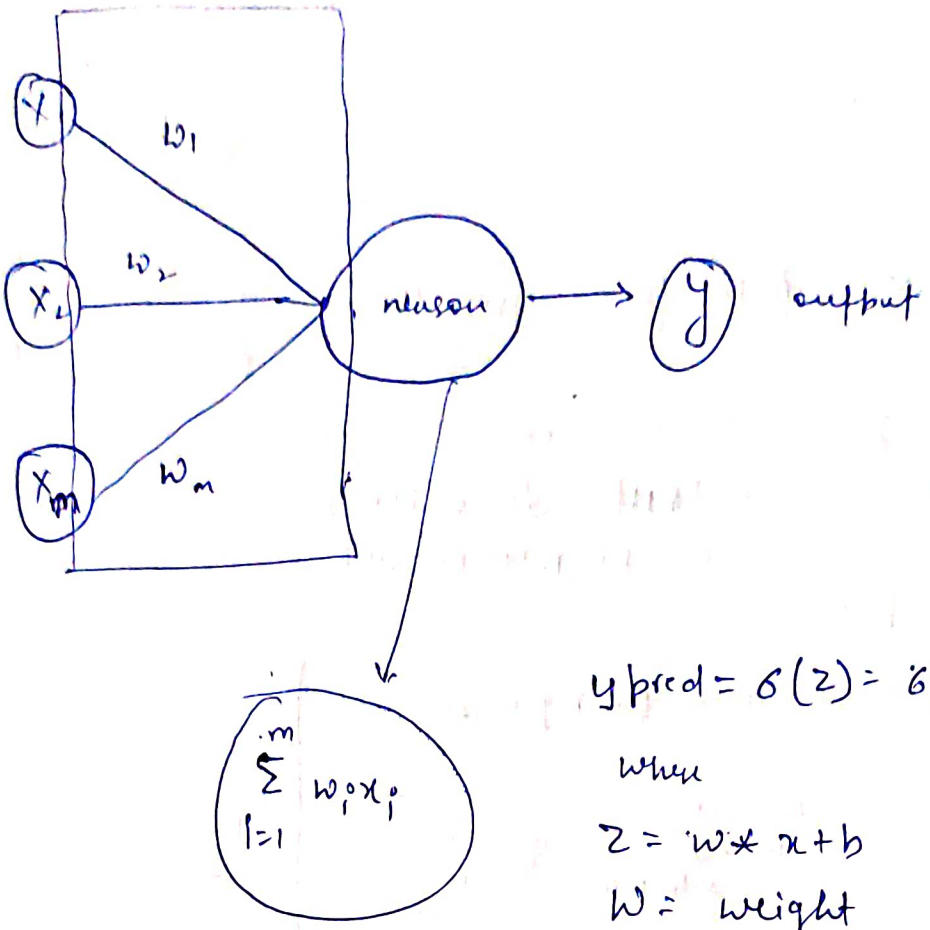
(2). This is where optimizers come into picture. They tie together the loss function & model parameters by updating the model in response to the output of the loss function.

Loss function / cost function

(1). It is a parameter in model's predict function that tells us "how good" the model is at making predictions for a given set of parameters.

(2). The loss function has its own curve & its own derivatives. The slope of this curve tells us how to change our parameters to make the model more accurate. We use the cost function to update our parameters.

* General Equation of Neural Network



$y_{pred} = \sigma(z) = \sigma(sum(w*x)+b)$

where

$z = w * x + b$

w = weight

x = input

b = bias

σ = Activation function

* Activation function

- properties

① - Non-Linear

② - Differentiable

- Types

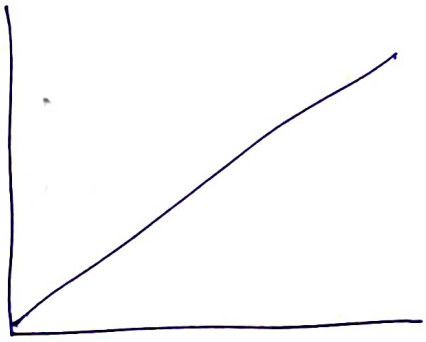
  - Liner
  - sigmoid
  - Tanh
  - Relu

- leaky Relu
- ELU
- softmax

•

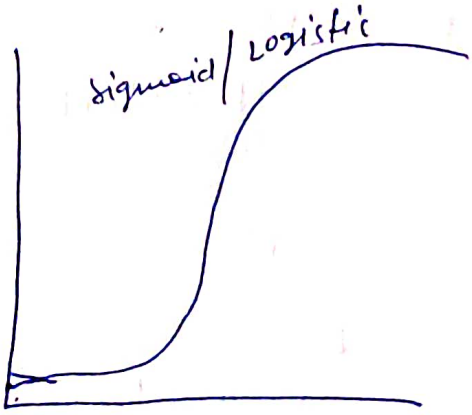
→ Linear Activation function

- Range $-\infty$ to $+\infty$
- It has constants derivative, so gradient has no relation with input
- Used in ANNs for regression
  ↓
  Artificial neural network
- $f(x) = x$



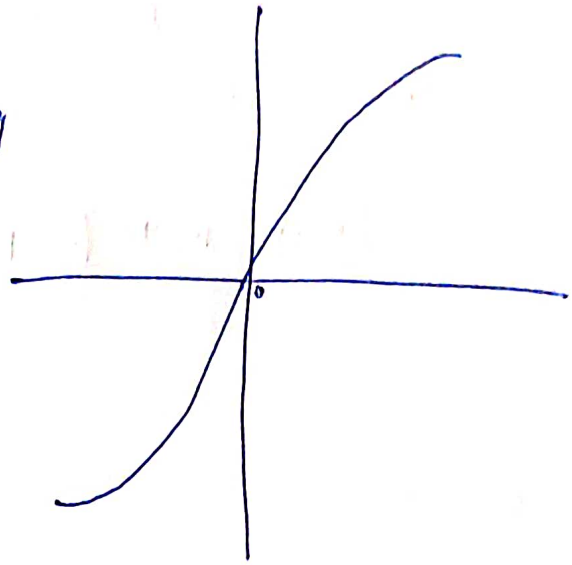
→ Sigmoid A. f.

- Range 0 to 1
- Used for binary classification models
- It suffers from vanishing Gradient
  (small) (slope)
- $f(x) = \frac{1}{1+e^{-x}}$



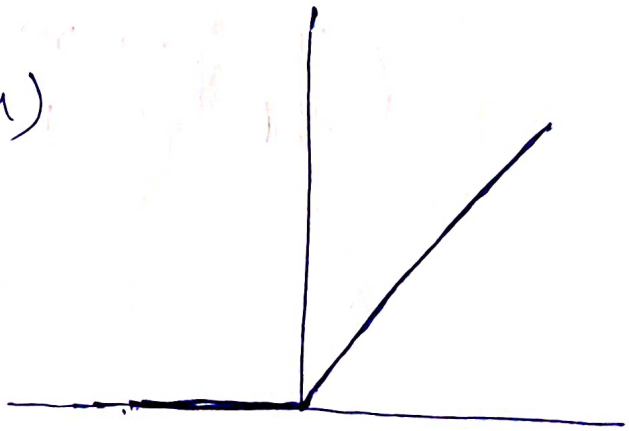
→ Tanh

- range -1 to 1
- used in RNNs
- It suffers from vanishing gradient.
- It is zero centered
- $f(x) = \frac{(e^x - e^{-x})}{(e^x + e^{-x})}$



→ RELU

- Relu means rectified linear unit.
- Range 0 to + infinity
- used in hidden layers
- It suffers from dying relu problem
- $f(x) = \max(0, x)$



~~~~ hll_out = Activation (hll_in)

hll_in = -ve

Activation = relu

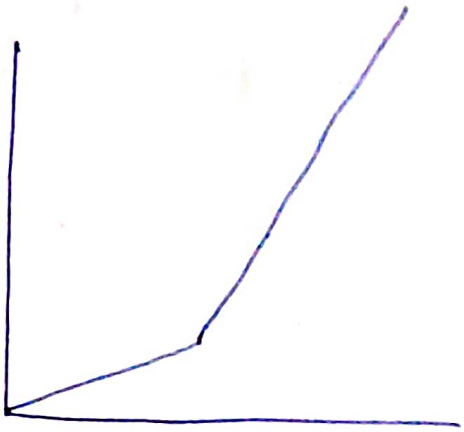
hll_out = 0

↳ so no contribution of this neuron; so it not activated

∴ suffers dying relu problem
~~~~

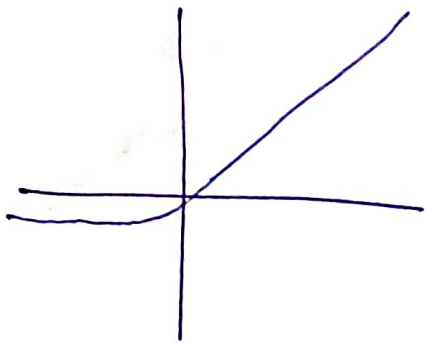
→ Leaky RELU

- leaky RELU is another variant of RELU
- used in hidden layers
- it overcomes the dying Relu problem
- $f(x) = \max(0.1x, x)$



→ ELU

- it stands for Exponential linear unit
- used in hidden layers
- it overcomes the Dying Relu problem
- suffers from exponential gradient.
- $\begin{cases} x & \text{for } x \geq 0 \\ \alpha(e^x - 1) & \text{for } x < 0 \end{cases}$



- → Activation function used in output layer is linear

* Vanishing gradient

- like the sigmoid function certain activation function squish an ample input space into a small output space b/w 0 & 1
- Therefore, a large change in the input of the sigmoid function will cause a small change in the output, Hence, the derivate becomes small. for shallow networks with only a few layers that use these activations, this isn't a big problem.
- However, when more layers are used, it can cause the gradient to be too small for training to work effectively.

* Exploding gradient

- Exploding gradient are problems where significant error gradients accumulates & result in very large updates to neural network model weights during training
- An unstable network can result when there are exploding gradients, & the learning cannot be completed.
- The values of the weights can also become so large as to overflow & result in something called NaN values.

* Optimizers

- point ① & ② already discussed previously

③ - Optimizers shape & mold your model into its most accurate possible form by maneuvering with the weights. The loss function is the guide to the terrain, telling the optimizer when it's moving in the right or wrong direction.

● Optimizers Functions

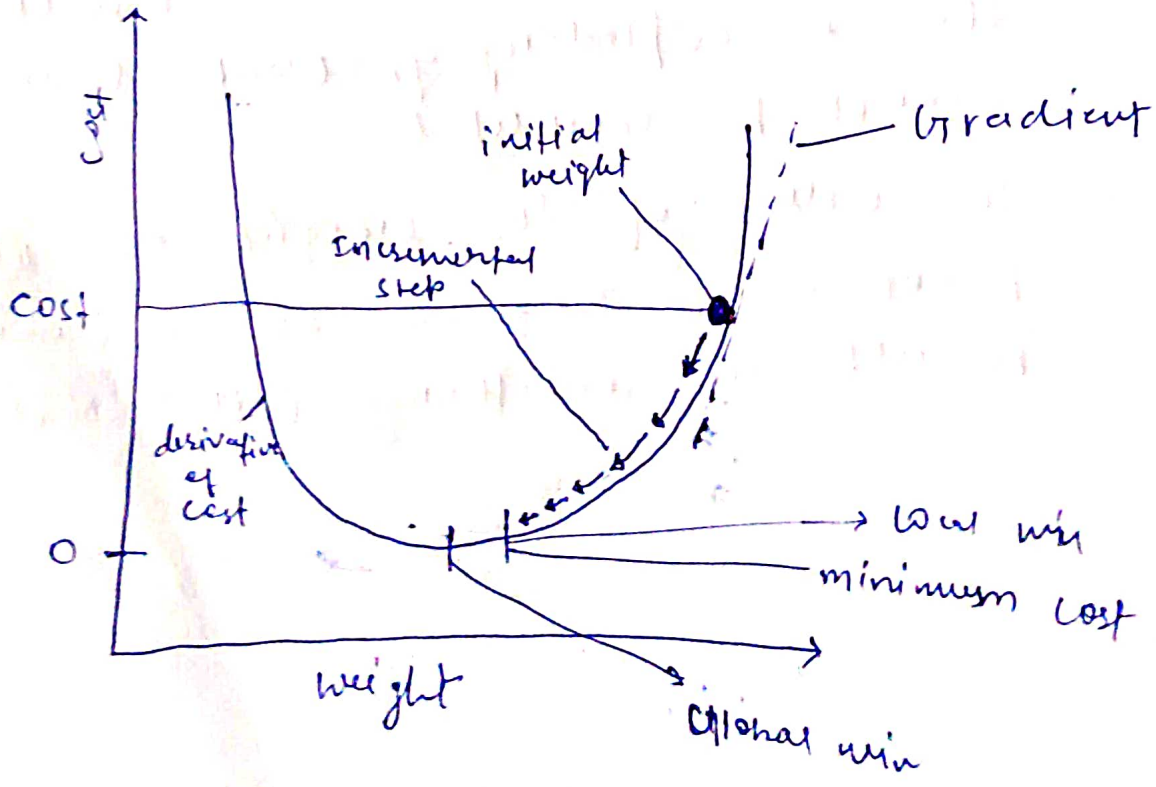
→ Gradient Descent

- It minimizes the given function to its local minimum.
- Gradient Descent iteratively reduces a loss function by moving in the direction opposite to that of steepest ascent.

→ $$W_{new} = W_{old} - \alpha \cdot \frac{\partial(Loss)}{\partial(W_{old})}$$

Loss = Cost = Error function

α → Learning Rate

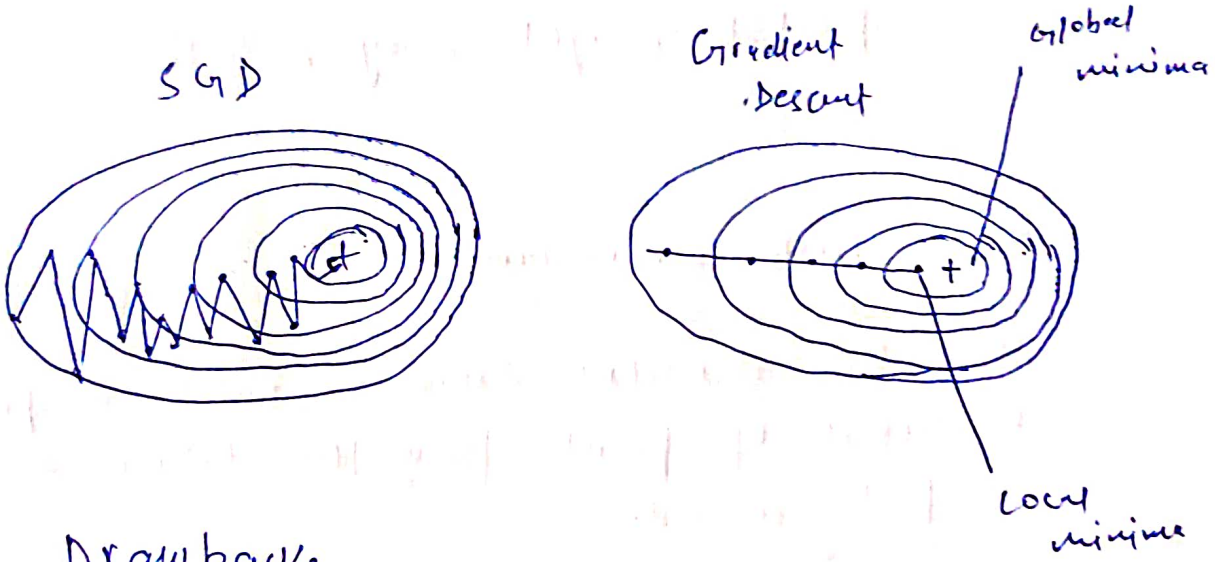


Drawbacks

- slow in computation
- Requires large memory
- computationally expensive

Stochastic Gradient Descent

- In the SGD algorithm derivative is computed taking one point at a time.
- Memory requirement is less than that of GD



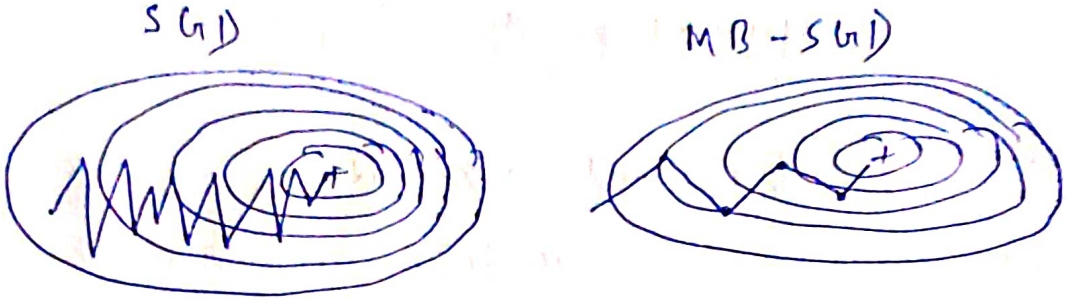
Drawbacks

- Takes a long time to converge
- May stuck at local minima

→ Mini-batch Gradient Descent

- MB-SGD algorithm takes a batch of points or subset of points from the dataset to compute derivate.
- MB-SGD divides the dataset into various batches & every batch, the parameters are updated.

- less time complexity as compared to SGD & more stab



Drawbacks

- may stuck at local minima
- updated weights may be too noisy

→ SGD with momentum

- MB-SGD algorithm takes a batch of points or subset of points from the dataset to compute derivate.
- MB-SGD divides the dataset into various batches & after every batch, the parameters are updated.
- The idea is to denois derivate using exponential weighting average that is to give more weightage to recent updates compared to the previous update.

$$V_{dw} = \beta \cdot V_{dw} + (1-\beta) \cdot dw$$

$$V_{db} = \beta \cdot V_{dw} + (1-\beta) \cdot db$$

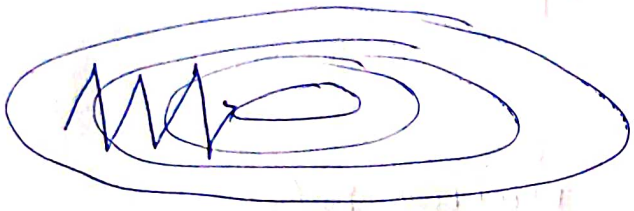
$$w = w - \alpha \cdot V_{dw}$$

$$b = b - \alpha \cdot V_{db}$$

- The problem with SGD is that while it tries to reach minima because of the high oscillation we can't increase the learning rate. so it takes time to converge

Drawbacks

- may stuck at local minima
- Updated weights may be too noisy



SGD without momentum

SGD with momentum

→ Adam (adaptive Momentum Estimation)

- It is a method that computes adaptive learning rate for each parameter. It stores both the decaying average of the past gradients, similar to momentum & also the decaying average of the past squared gradients, similar to RMS-Prop & Adadelta

$$V_{dw_t} = \beta V_{dw_{t-1}} + (1-\beta)\frac{\partial L}{\partial w_{t-1}}$$

$$V_{db_t} = \beta V_{db_{t-1}} + (1-\beta)\frac{dL}{\partial b_{t-1}}$$

exponential weighted avg for past gradients

$$\begin{cases} S_{dw_t} = \gamma S_{dw_{t-1}} + (1-\gamma)\left(\frac{\partial L}{\partial w_{t-1}}\right)^2 \\ S_{db_t} = \gamma S_{db_{t-1}} + (1-\gamma)\left(\frac{\partial L}{\partial b_{t-1}}\right)^2 \end{cases}$$

exponential avg for past squared gradients

$$\begin{cases} w_t = w_{t-1} - \frac{\eta}{\sqrt{S_{dw_t} - \varepsilon}} * V_{dw_t} \\ b_t = b_{t-1} - \frac{\eta}{\sqrt{S_{db_t} - \varepsilon}} * V_{db_t} \end{cases}$$

Adam optimizer

alpha = neta = learning rate

$$w' = w - \text{alpha} * \left(\frac{dL}{dw}\right)$$

* Loss functions

loss or cost or error function

- In a neural network, the weights get multiplied with the inputs & then activation function is applied to the element before going to the next layer. Finally we get the predicted value (yhat) through the output layer. But prediction is always close to the actual (y), which we term as error. So we define the loss/cost functions to capture

the essoss & try to optimize it through backpropagation.

- The essor function that compares actual value with its corresponding predicted value is professed to as loss function or cost function

- Types

→ MSE (Mean Square Essor)

- It is used in Regression
- MSE is the professed loss function if the distribution of the target variable is Gaussian. Mean Square essor is defined as the average of the squared differnce b/w the predicted & actual value

$$L = \frac{1}{n}\sum_{i=1}^{n}(y-\hat{y})^2$$

($\hat{y}$ → predicted value)

n is batch size.

→ MAE (Mean absolute essor)

- It is used in Regression
- It is defind as the average of the absolute differnce b/w the actual & predicted values.

$$L = \frac{1}{n} \sum_{i=1}^{n} |y - \hat{y}|$$

→ Binary Cross Entropy

- It is used in Binary classification.
- It measures the difference b/w two probability distributions. If the cross entropy is small, it suggest that two distributions are similar to each other.
- In case of a binary classification the predicted probability is compared to the target/actual (0 or 1). Binary cross entropy calculates a score that provides the negative average difference b/w the actual & predicted probabilities for predicting the class. 1. This score penalizes the probability based on the distance from the expected value.

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

$$= \begin{cases} -y * \log(\hat{y}), & \text{if } y = 1 \\ -(1-y) * \log(1-\hat{y}), & \text{if } y = 0 \end{cases}$$

→ Categorical cross Entropy

- It is used in Multiclass classification
- In case of a multi-class classification

the predicted probability is compared to the target/actual, where each class is assigned a unique integer value (0, 1, 3..., t), assuming data has t unique class. It calculates a score that provides the negative average difference b/w the actual & the predicted probabilities for all classes

- for multi-class cross entropy, actual targets (y) are one-hot encoded. For a 3-class classification [[0,0,1], [1,0,0] [0,1,0]]

$$L = -\sum_{j=1}^{t} y_{ij} * \log(\hat{y}_{ij})$$

i = no. of rows

j = no of categories

y_i = is one hot encoded target vector

→ Categorical Cross Entropy

- It is used in Multiclass classification
- In case of a multiclass classification the predicted probability is compared to the target/actual, where class is assigned a unique integer value (0, 1, 3, ..., t), assuming data has t unique class. It calculates a score that provides the negative average difference b/w

the actual & predicted probabilities for all classes.

- for multiclass cross entropy, actual targets (y) are one-hot encoded. For a 3-class classification

[[0,0,1], [1,0,0], [0,1,0]]

$$L = -\sum_{j=1}^{t} y_{ij} * \log(\hat{y}_{ij})$$

where i = no. of rows
j = no of categories
y_i = is one hot encoded target vector.

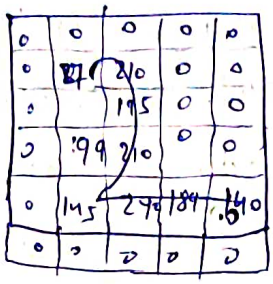
CNN (Convolutional Neural Network)

- CNN image classifications take an input image, process it & classify it under certain categories (E.g., Dog, Cat, Tiger, Lion).
- Computers see an input images as assay of pixels & it depends on the image resolution. Based on the image resolution, it will see h x W x d (h = height, W = width, d = dimensions). E.g. An image of 6 x 6 x 3 assay of matrix of RGB (3 refers to RGB values) & an image of 4 x 4 x 1 assay of matrix of grayscale image.
- When Deep learning CNN models are applied to train & test, each input image will pass through a series of convolution layers with filters (Kernels), Pooling, Flatten layer, fully connected layers (FC). ~~Softmax~~ and apply softmax function to classify an object with probalistic values b/w 0 & 1. For multiclass classification, we can use loss as ^ Categorical cross entropy, sparse. Categorical cross entropy.
  - Categorical cross entropy → used for label encoded data
  - ! hot encoded data

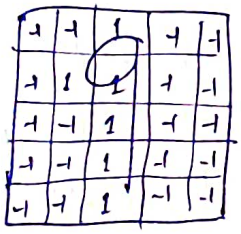
Convolution

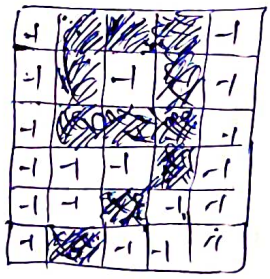
- Let us suppose this is the input matrix of 5 x 5 & a filter of matrix 3 x 3, for those who don't know what a filter is a set of weight applied on an image or a matrix to obtain the required feature.

- In Convolutional layer, the image pixels (image matrix) are multiplied with filter matrix to generate a feature matrix.
- Consider the handwritten digit 2 & its pixel representation.



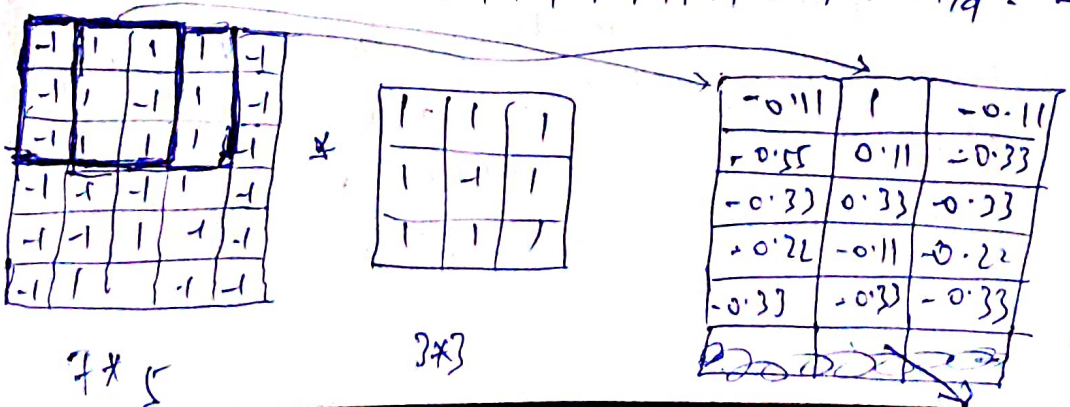
- for sake of simplicity of mathematical operations. consider image matrix of 9 as follows





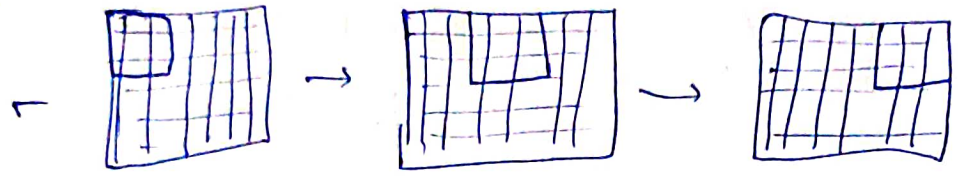
- Convolution operation is applied using a filter. Let's assume that a 3*3 filter is used for this operation. In convolution layer, the image pixels (image matrix) are multiplied with filter (kernels) matrix to generate feature map.

$-1+1+1-1-1-1-1+1+1 = -1 = -1/9 = -0.11$

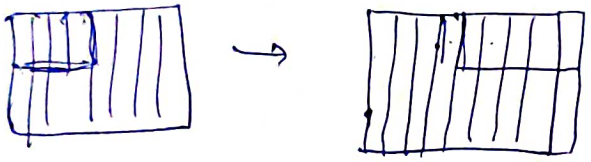


img := (ih ; iw)
k = (kh, kw)
fm = (ih - kh + 1, iw - kw + 1)

Featured map



Stride = 1 (1 cell movement)



- Stride = 2 (2 cell movement)

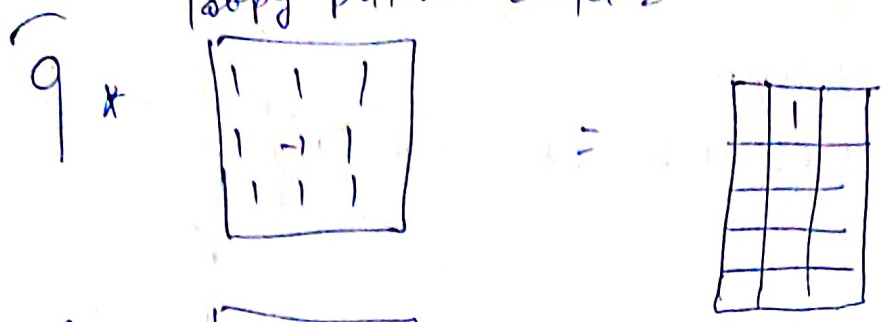
↓ for Stride = 2

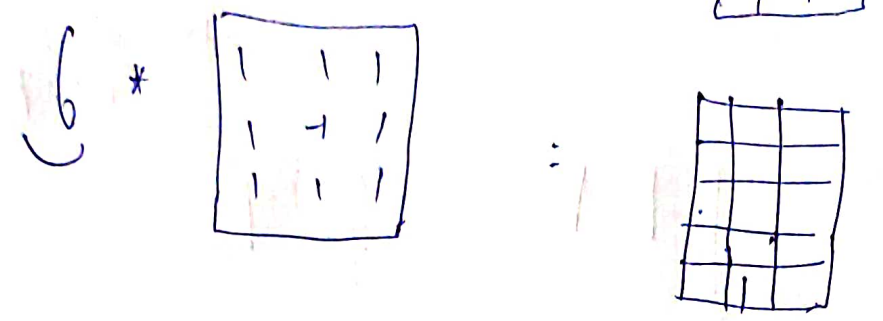
img := ih, iw
k := kh, kw
s := s
fm := [(ih - kh)/2 + 1; (iw - kh)/2 + 1]

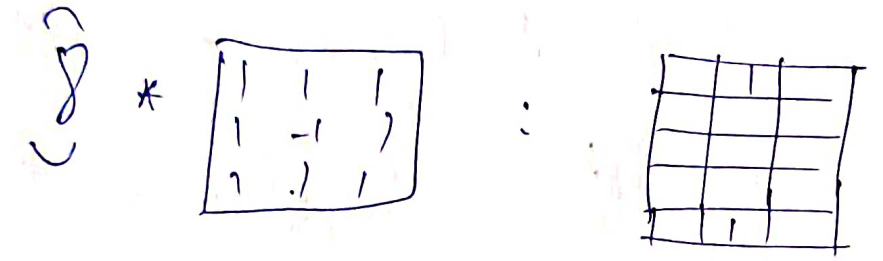
- This is how result of convolution process detects different patterns in the images using filters.

→

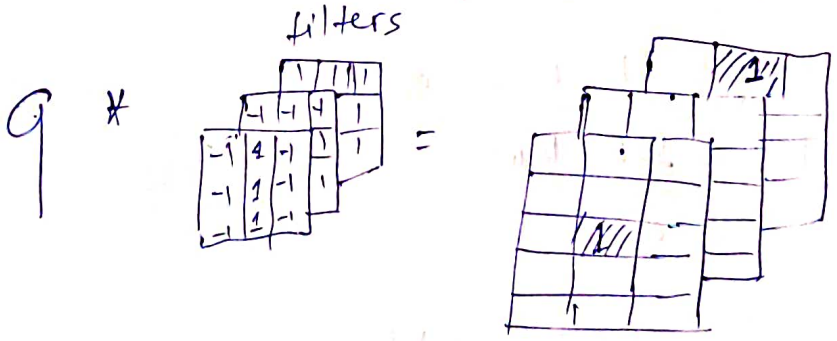
loopy pattern detectors

- 

- 

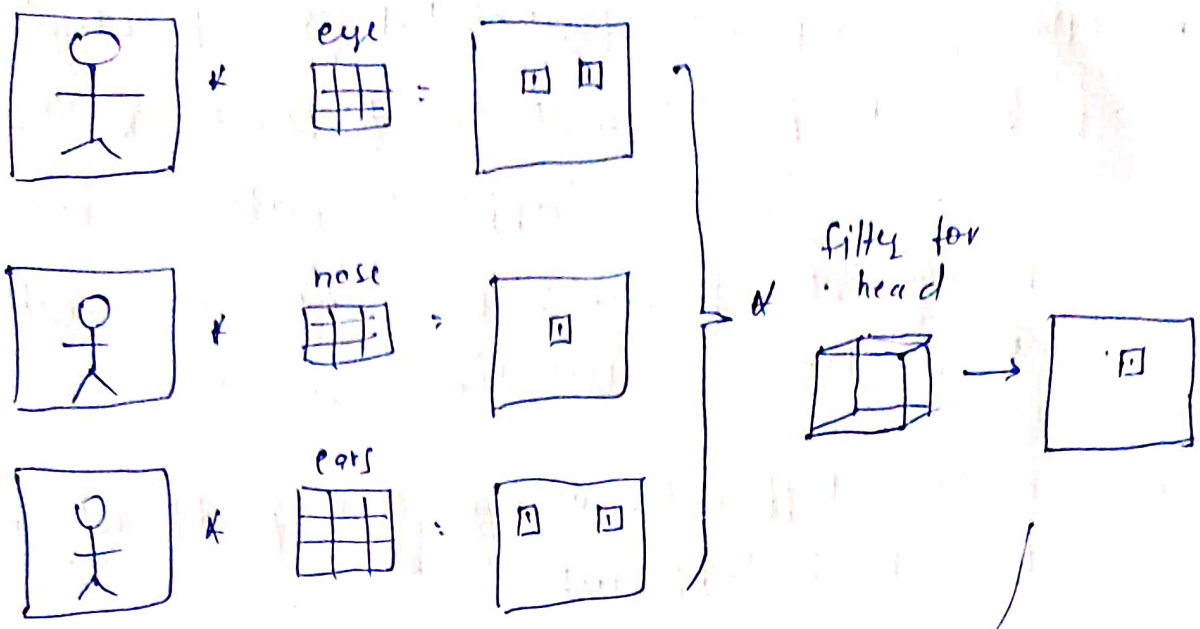
- 

- 8 filters are used in each convolution layer to detect different features

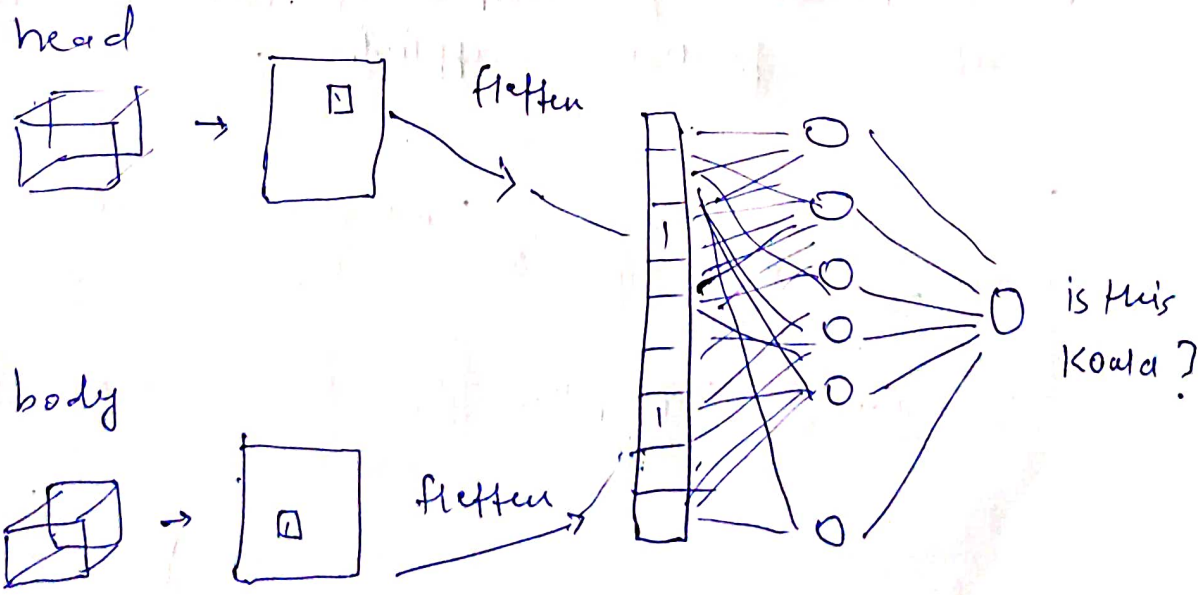


feature maps

→



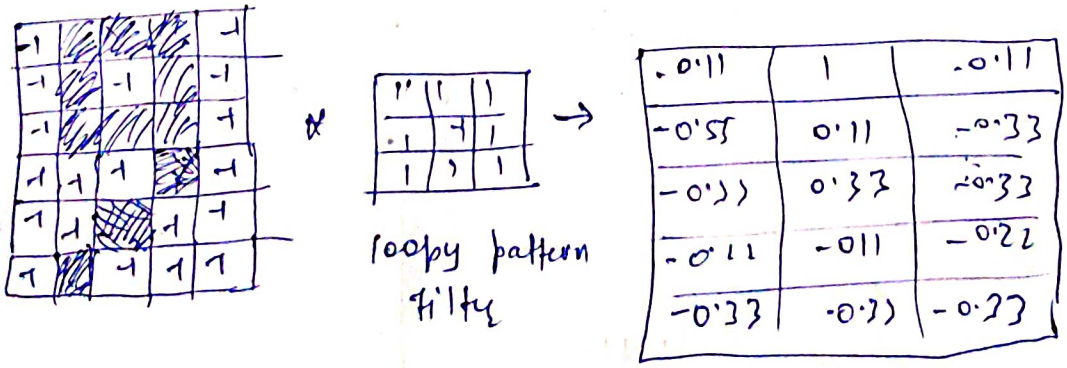
- In case of above example, filters might end up detecting eyes, nose, ears, Then feature map matrix obtained is again subjected to another convolution possibly to detect head of animal.

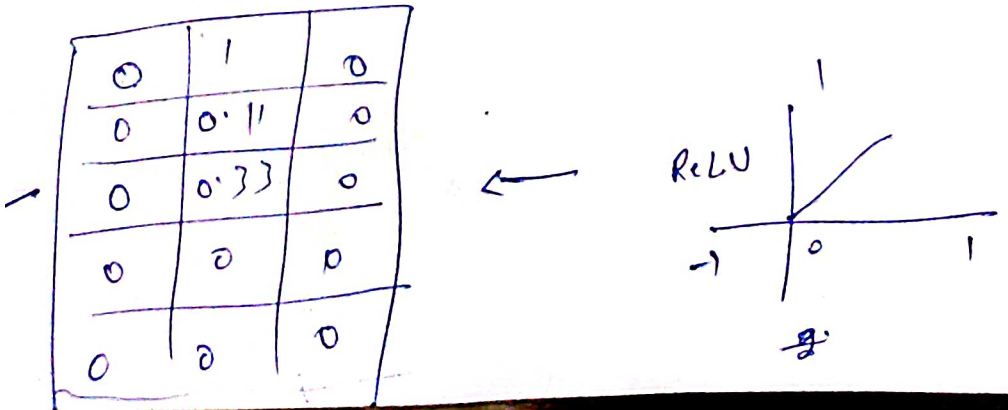


- After two repeated convolutional operation, the feature map obtained is flattend & then a Dense layer is applied with sigmoid (binary classification) or softmax (multi classification) activation function.
- with different images the flattend matrix will be different.

- Convolutional layer is responsible for feature extraction
- Dense output layer is responsible for classification

* Activation function is applied to the feature map.
  - on hidden layer (output of convolution layer) activation function is applied.





* Pooling

- pooling layer is used to reduce the dimension
- on the feature map obtained after convolutional layer, MaxPooling layer is applied to select the maximum value from that window of given stride.

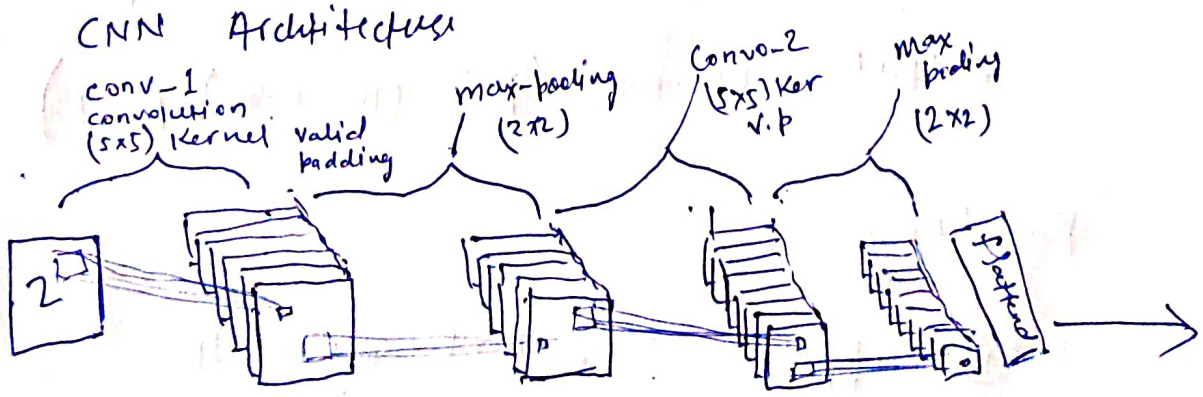
max pooling

0.33	0.33
0.33	0.33
0	0

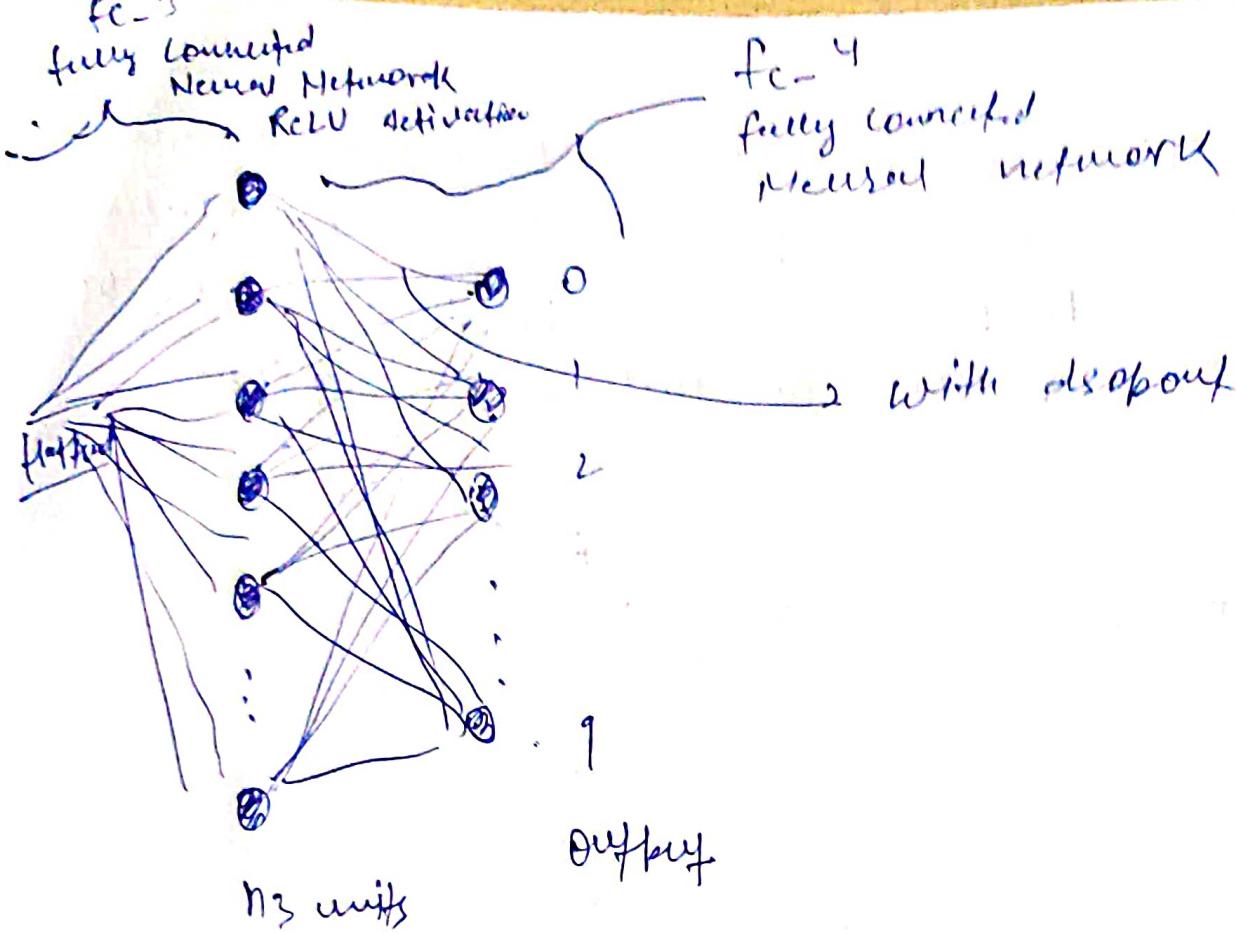
pooling size = (2, 2)
stride = 1

- changing the position of 9 to back to 1 cell, this type of positional movement is known as translational.
- Max pooling along with convolution helps detect position invasient feature detection.
- pooling also prevents overfitting as these are less parameters.

* CNN Architecture



input (28×28×1) | n_1 channel (24×24×n_1) | n_1 channel (12×12×n_1) | n_2 channel (8×8×n_2) | n_2 channel (4×4×n_2)



* Stride

- Stride is number of pixels shifts over the input matrix. When the stride is 1 then we move filters to 1 pixel at a time. When the stride is 2 then we move the filters to 2 pixel at a time & so on.

Input (m, n), filter (n, n) Stride $= p$

output $= (m-n)/p+1, (m-n)/p+1)$

* Padding

It is applied so that even the corner pixels get their due weightage.