INNOMATICS
RESEARCH LABS

Deep Learning

Artificial Neural Network

Introduction

INNOMATICS RESEARCH LABS

What are Neural Networks?

- Neural networks are a new method of programming computers.
- They are exceptionally good at performing pattern recognition and other tasks that are very difficult to program using conventional techniques.
- Programs that employ neural nets are also capable of learning on their own and adapting to changing conditions.
- An NN is configured for a specific application, such as pattern recognition or data classification, through a learning process. Learning in biological systems involves adjustments to the synaptic connections that exist between the neurons. This is true of NNs as well.

INNOMATICS RESEARCH LABS

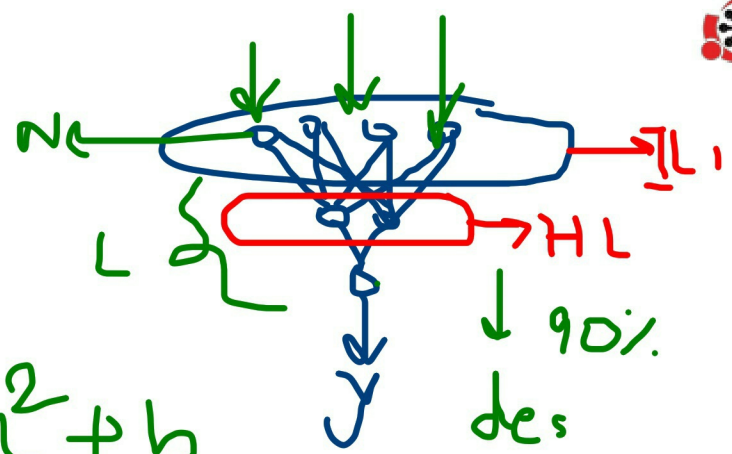
Is Deep learning Worth the Hype ?

- Wide range of application



- Automated Feature Selection
- Architecture Innovation and Flexibility
- Scalable and Robust
- Understand the underlying distribution of the data

Building Components



Layer:
IP: Input Layer
HL: Hidden Layer
OL: Output Layer

$f(x) = wx^2 + b$

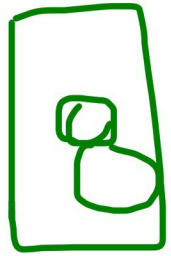
Layers

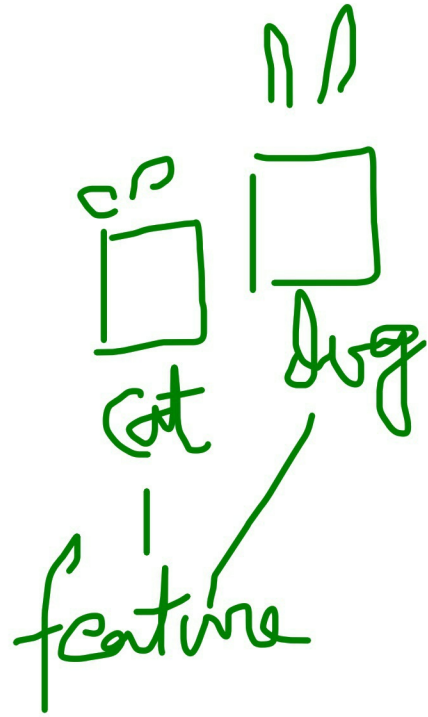
Nodes

Activation Functions

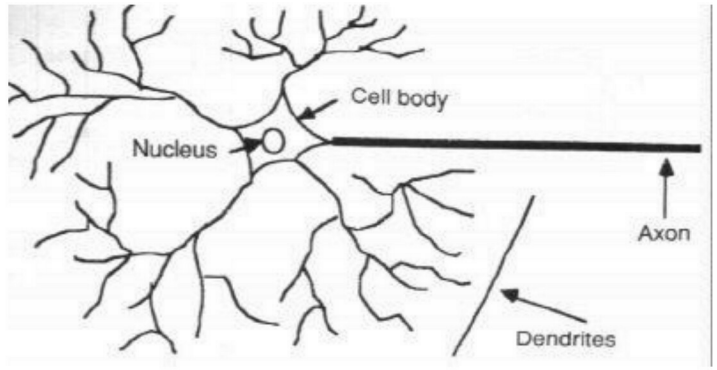
Optimizers

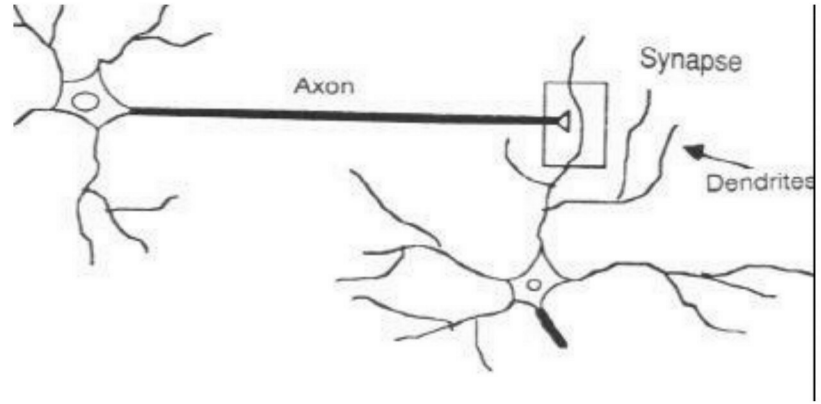
Back Propagation

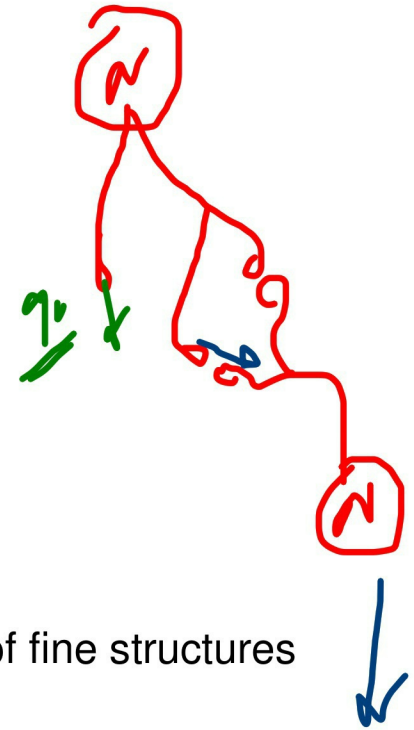




How the Human Brain learns

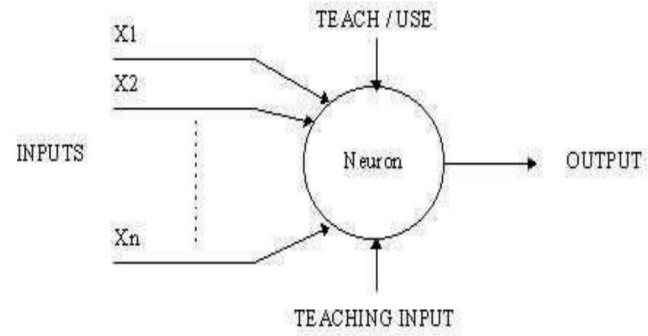


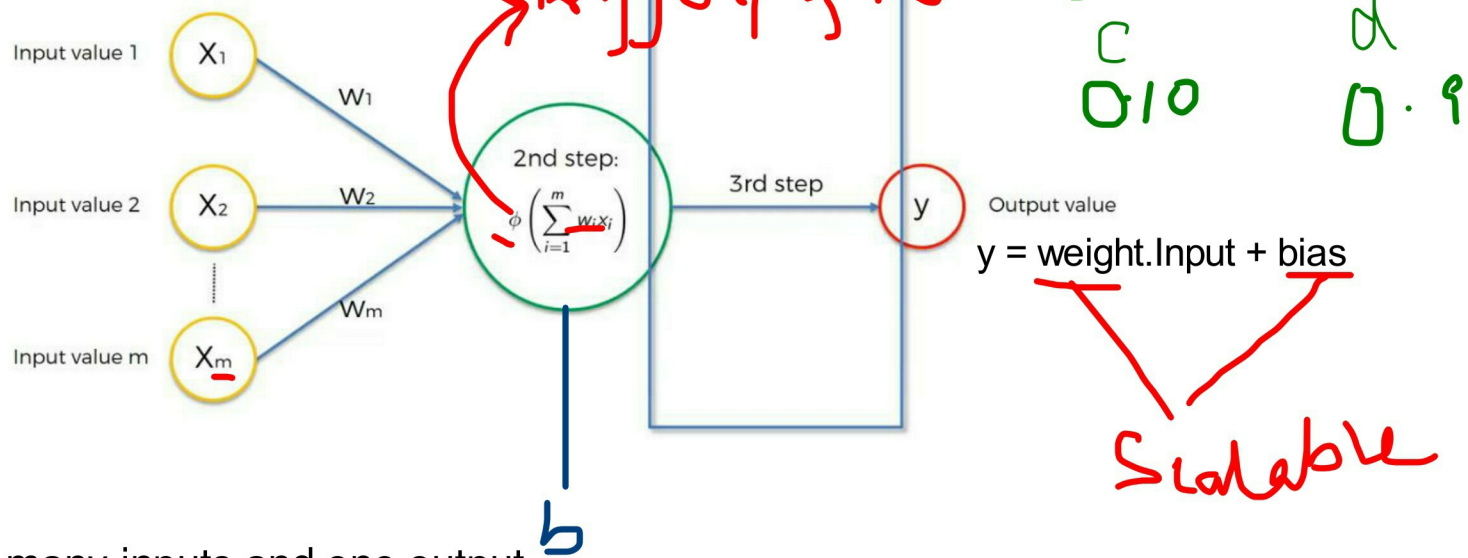




- In the human brain, a typical neuron collects signals from others through a host of fine structures called *dendrites*.
- The neuron sends out spikes of electrical activity through a long, thin stand known as an *axon*, which splits into thousands of branches.
- At the end of each branch, a structure called a *synapse* converts the activity from the axon into electrical effects that inhibit or excite activity in the connected neurons.

A Simple Neuron





An artificial neuron is a device with many inputs and one output.

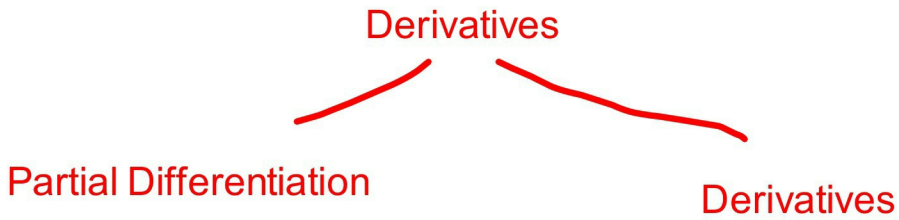
The neuron has two modes of operation;

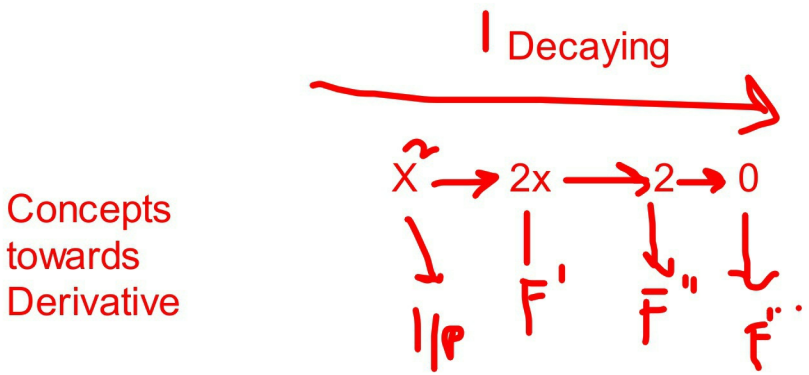
- ✓ the training mode
- ✓ the using mode.

$\hat{y}$ = AF(Sum(weight* input + bias))

- ❖ In the training mode, the neuron can be trained to fire (or not), for particular input patterns.
- ❖ In the using mode, when a taught input pattern is detected at the input, its associated output becomes the current output. If the input pattern does not belong in the taught list of input patterns, the firing rule is used to determine whether to fire or not.

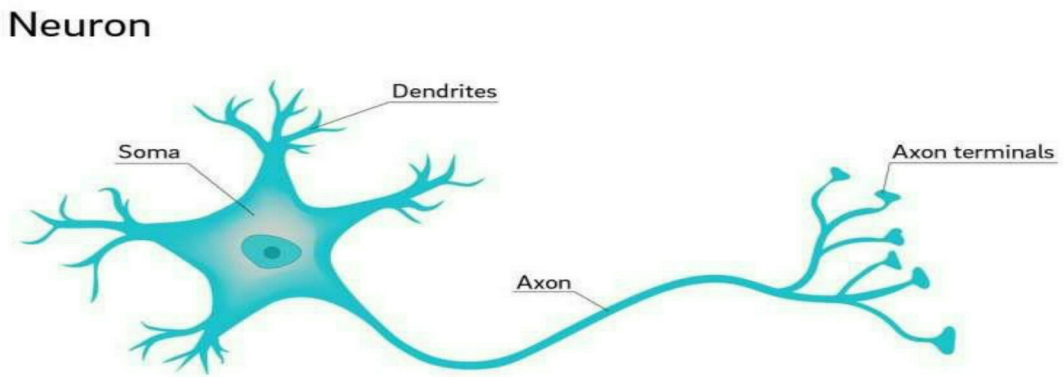
N = Integer Value
(Neg/Positive)





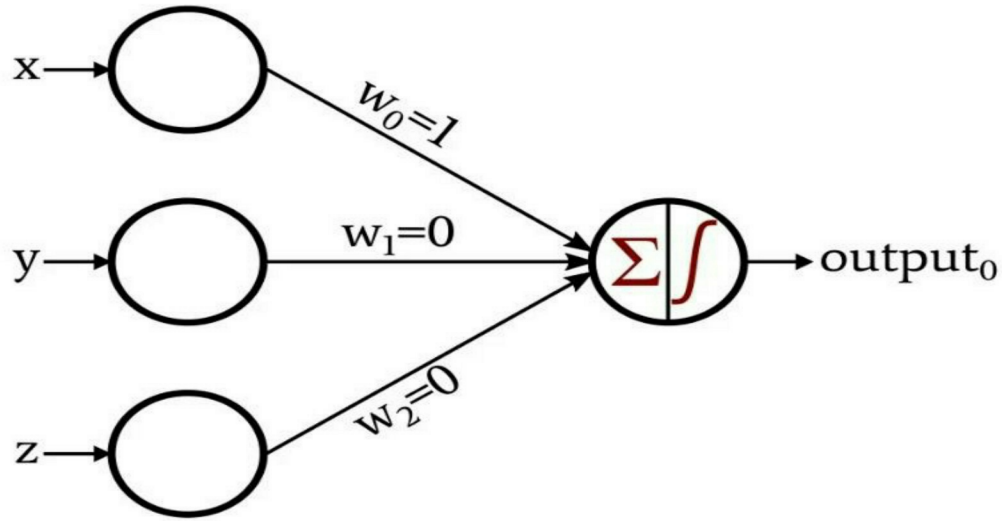
What is a neuron?

- The **neuron** is the basic working unit of the brain, a specialized cell designed to transmit information to other nerve cells, muscle, or gland cells. **Neurons** are cells within the nervous system that transmit information to other nerve cells, muscle, or gland cells



perceptron

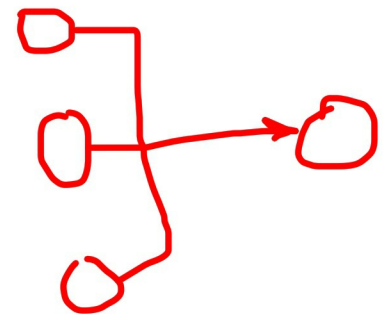
A **perceptron** is a neural network unit (an artificial neuron) that does certain computations to detect features or business intelligence in the input data



Activation function

Transformation from Linear to non linear fucntion

- Activation functions are mathematical equations that determine the output of a neural network. The function is attached to each neuron in the network, and determines whether it should be activated (“fired”) or not, based on whether each neuron’s input is relevant for the model’s prediction.
  - What is the role of Activation function?

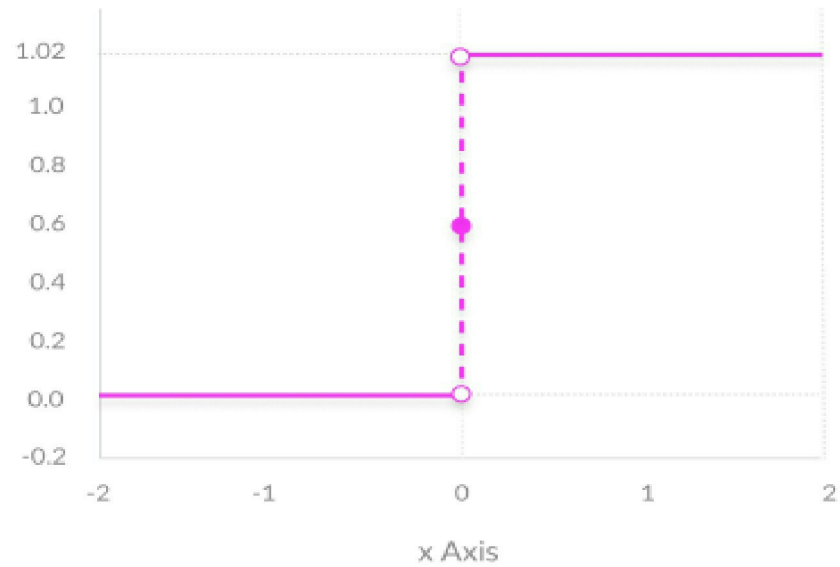


process


Take input and multiply by the neuron's weight.
Add bias*
Feed the result, x, to the activation function: f(x)
Take the output and transmit to the next layer of neurons.
Tn

Types of activation functions

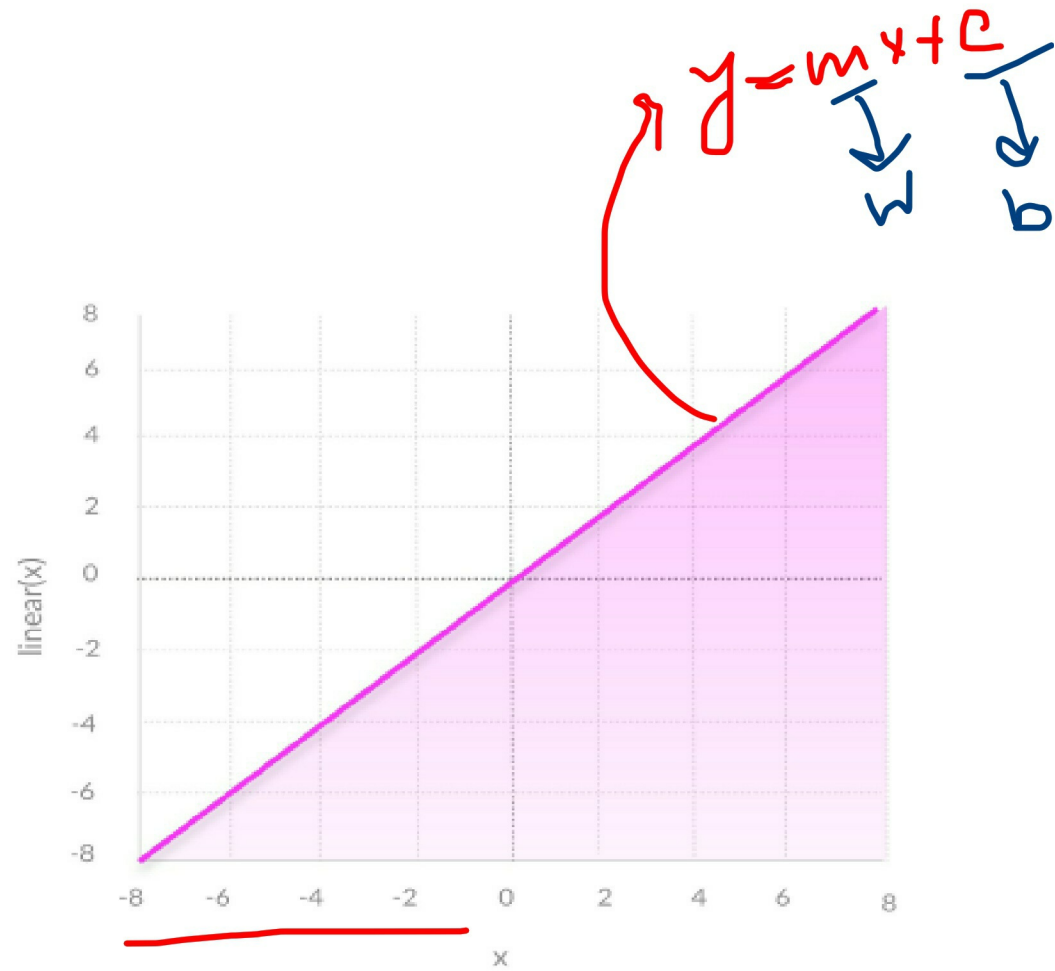
- Binary Step Function



Linear activation function

- no range
- Linear models

0 - 1



Sigmoid function

$\sigma(x)$

known as logistic function / Squashing Fucntion

- Non Linear AF
- mostly used in feed forward neural network

$w \cdot x + b$

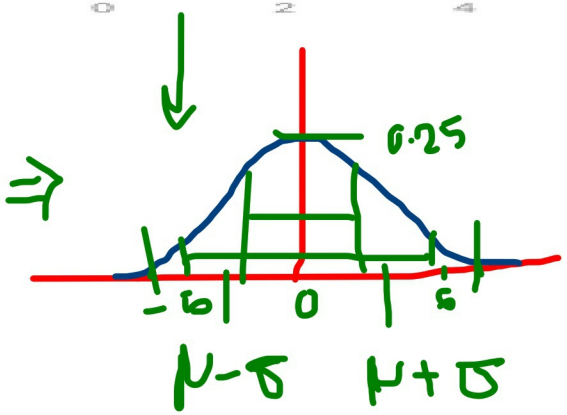
$$f(x) = \frac{1}{1 + e^{-x}}$$

Range: 0 - 1

Standard Normal Distribution = Mean $\mu = 0$

$\sigma^2 = 1$

$$\frac{d\sigma(x)}{dx} \Rightarrow$$



$$f(z) = \frac{1}{1 + e^{(1-2)}}$$

Sigmoid Types

- Hard Sigmoid → Solution to complex/ time consuming sigmoid, Can be implemented in H/W or S/W related solution
- Sigmoid Weighted Linear Units
- Derivative of Sigmoid weighted Linear Units → Derived/motivated from Gradient Descent

$$f(x) = \max(0, \min(1, \frac{x+1}{2}))$$

$$z \cdot \alpha(z)$$

weight

It can only be used in Hidden layers and in Reinforcement learning

$$\alpha(z) \cdot (1 + z(1 - \alpha(z)))$$

It can be used in modifying/ upgrading weights

$$\sum w \cdot 1/p + b$$

Decaying Problem

$W_{old} \rightarrow 0.1$

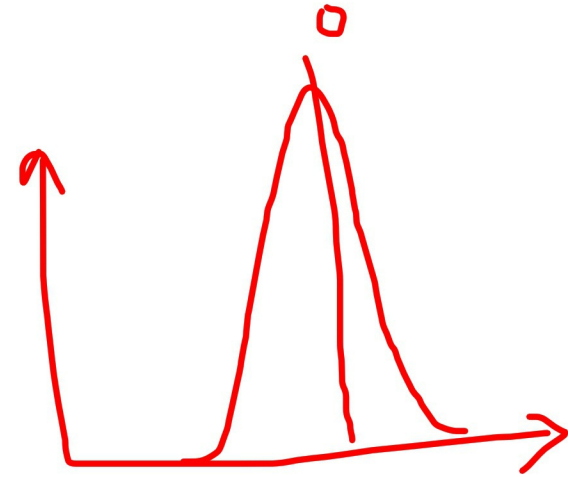
$W_N = \frac{0.01}{} \times 6 + 0.1$

TanH / Hyperbolic Tangent

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

-Gives better performance in training and specifically in multilayer network

Zero centered activation function
Provide strong decision (positive/negative/netrual scenarios)



Range: -1 to 1

Widely In Natural Language processing, NLU, NLG & Sentiment Analysis, Speech Recognition

Hard TanH

$$f(x) = \begin{cases} -1 & x < -1 \\ x & -1 = x \leq 1 \\ 1 & x > 1 \end{cases}$$

Speech Recognition

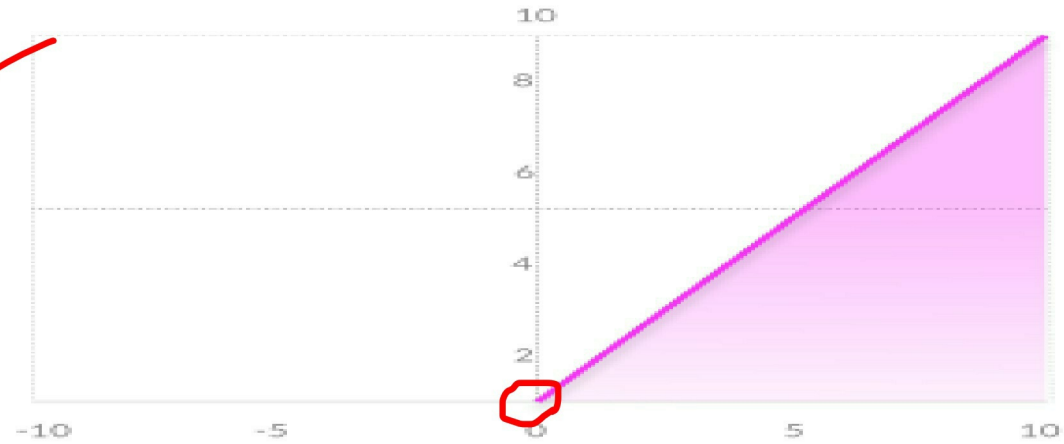
Accuracy Improvement

ReLU (Rectified Linear Unit)

67

1024

2

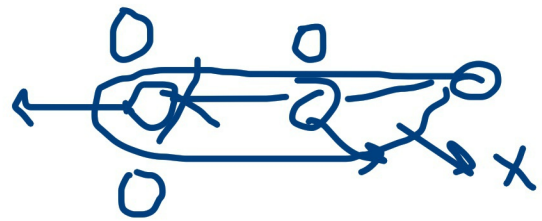


$$f(x) = \max(0, x)$$

$$\Rightarrow \begin{cases} x & x \geq 0 \\ 0 & x < 0 \end{cases}$$

Disadv:

- Easily Overfit
- Gradient Die



Adv:

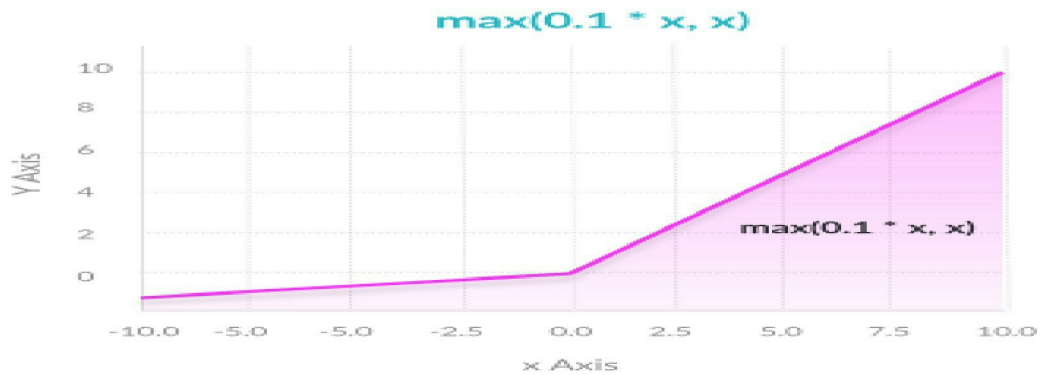
- It introduces sparsity in hidden layers

squeeze the value frm 0 - max

Leaky ReLU

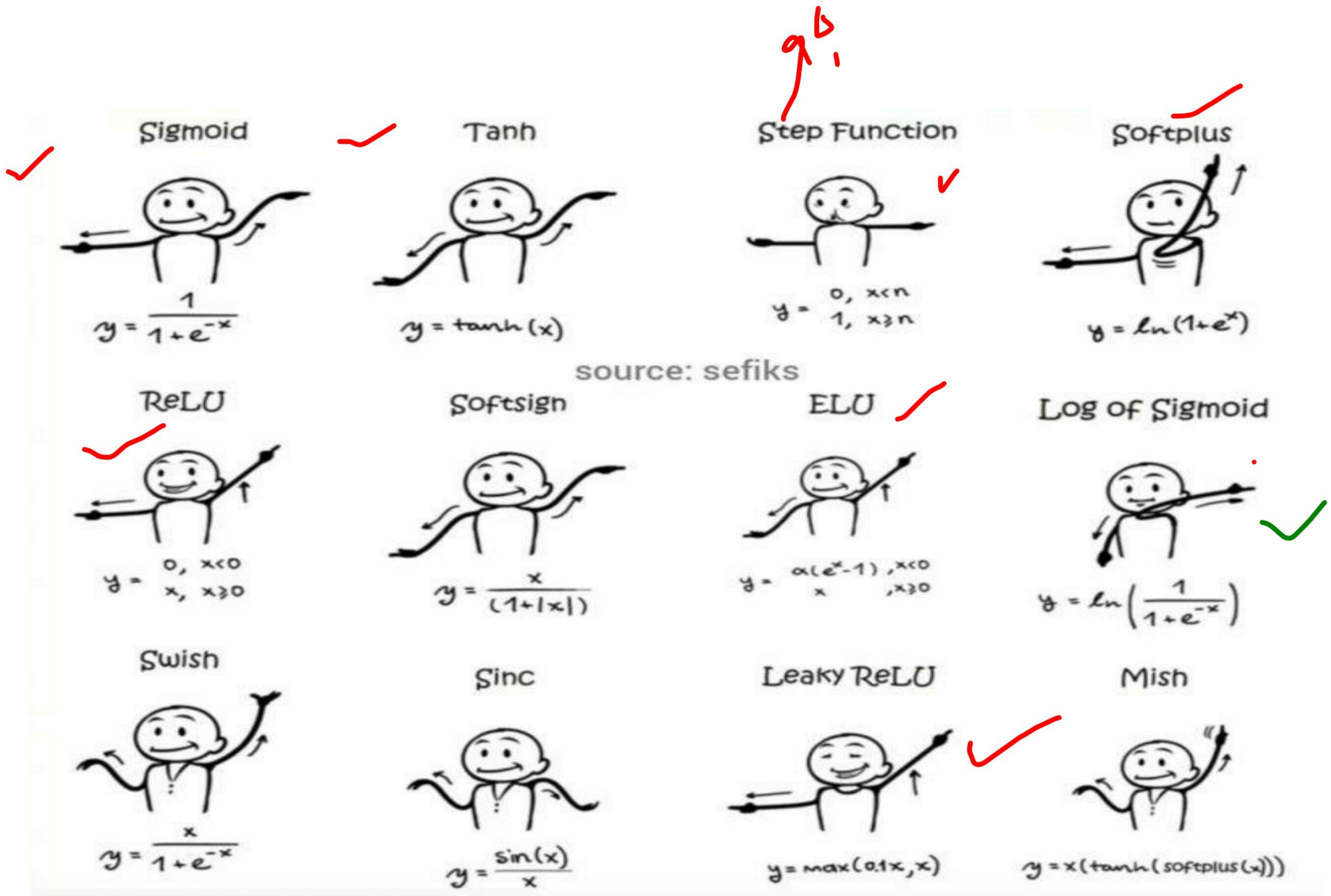
learning Constamt 0.01

$$f(x) = \alpha x + x$$



$$\begin{cases} x & x \geq 0 \\ \alpha x & x \leq 0 \end{cases}$$

Widely used in Speech Recognition & Classification


Sigmoid
$y = \frac{1}{1+e^{-x}}$
Tanh
$y = \tanh(x)$
Step Function
$y = \begin{cases} 0, & x<n \\ 1, & x \geq n \end{cases}$
Softplus
$y = \ln(1+e^{x})$
source: sefiks
ReLU
$y = \begin{cases} 0, & x<0 \\ x, & x \geq 0 \end{cases}$
Softsign
$y = \frac{x}{(1+|x|)}$
ELU
$y = \begin{cases} \alpha(e^{x}-1), & x<0 \\ x, & x \geq 0 \end{cases}$
Log of Sigmoid
$y = \ln\left(\frac{1}{1+e^{-x}}\right)$
Swish
$y = \frac{x}{1+e^{-x}}$
Sinc
$y = \frac{\sin(x)}{x}$
Leaky ReLU
$y = \max(0.1x, x)$
Mish
$y = x(\tanh(\text{softplus}(x)))$

Softplus Gradient

$$f(x) = \log(1 + e^x)$$

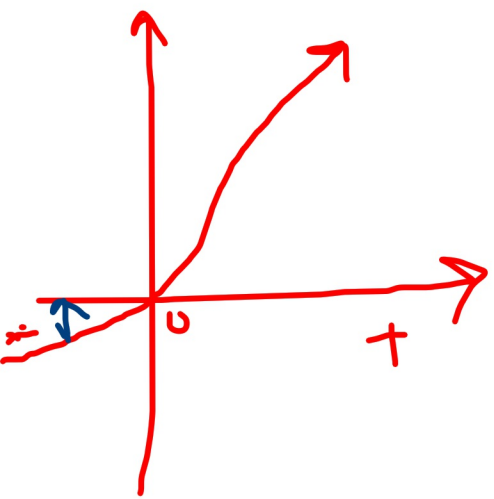
non Zero AF

Smoothing our learning curve
It works with lesser epochs

Widely in Speech Recognition

also known as dominant AF

ELU = Exponential Linear units

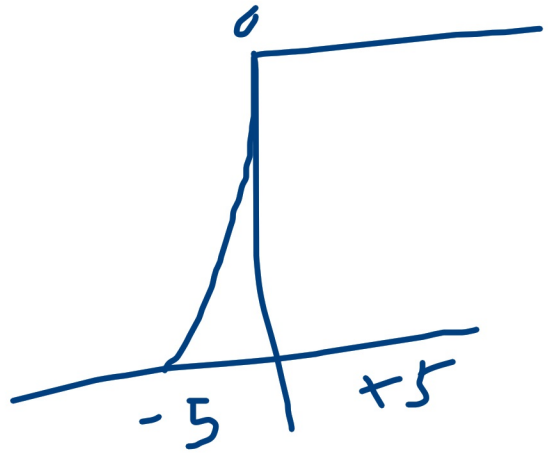


$$f(x) = \begin{cases} x & x > 0 \\ \alpha(e^{(x)} - 1) & x \leq 0 \end{cases}$$

Stops from Neuro Die/ Gradient Die

Learnable constant

(negate cons)



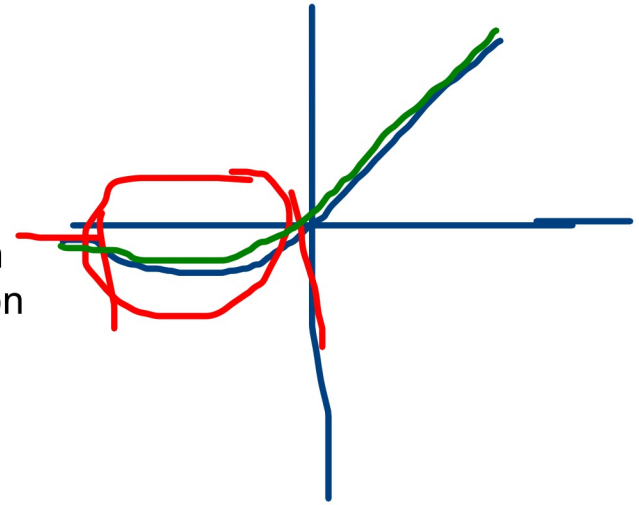
$$f(x) = \begin{cases} 1 & x > 0 \\ f(x) + \alpha & x \leq 0 \end{cases}$$

Swish - Designed by Google

$$f(x) = y = x * \text{sigmoid}(x)$$

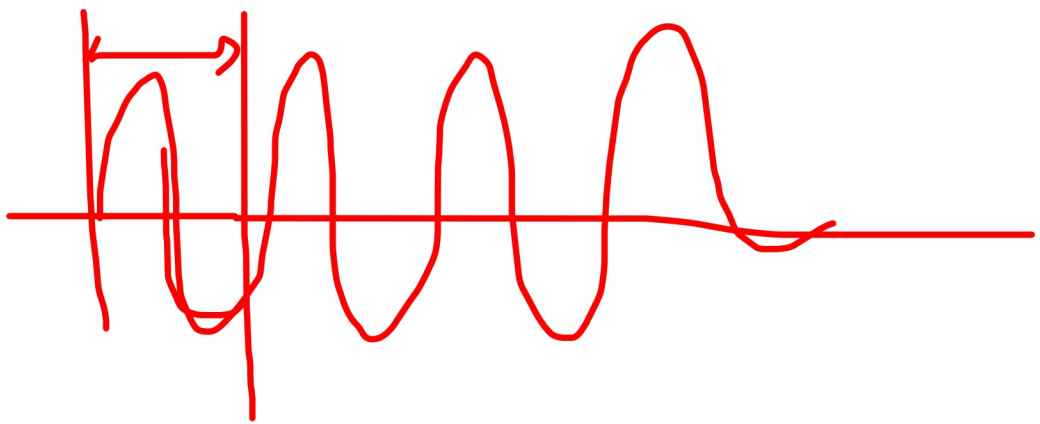
Layer greater than 40

Widely used in LSTM (solution of RNN)



Sinc Fucntion - Mostly used in IOT based devices / Computations

$$y = \frac{\sin(x)}{x} \quad \text{where } x \neq 0$$



Used by NASA

It can learn quicky with less amount of epoch

Mish

$$f(x) = \log(1+e^x)$$

$$Y = X(\tanh(\text{Softplus}(x))$$

Used by Google
Performs a bit
better than
Swish

SoftSign
Function

$$y = \frac{x}{1+|x|}$$

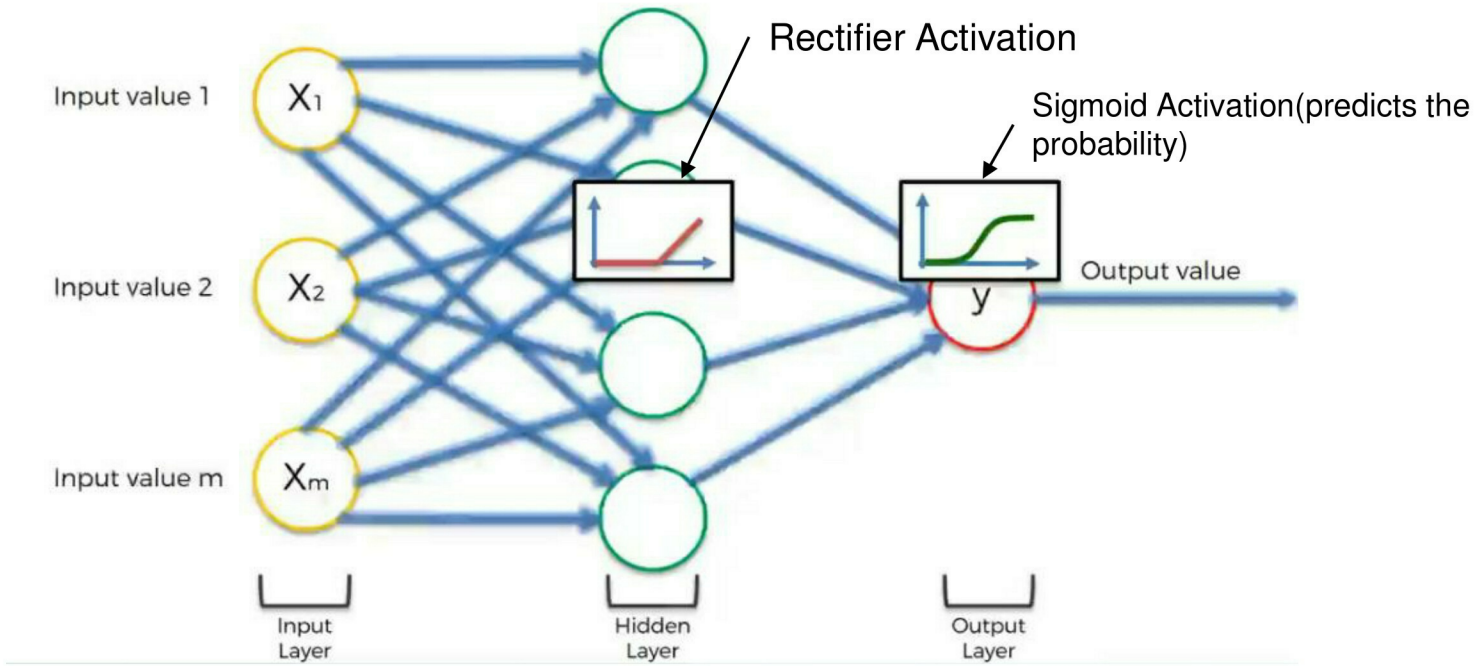
Range : -1 to 1

Activation functions

- Decides whether a neuron should be activated or not.
- Whether the information that the neuron is receiving is relevant for the given information or should it be ignored.

$$Y = \text{Activation}(\Sigma(weight * input) + bias)$$

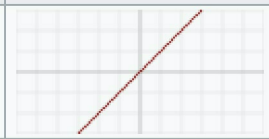
- The activation function is the non linear transformation that we do over the input signal. This transformed output is then sent to the next layer of neurons as input.



Different activation functions:

INNOMATICS RESEARCH LABS

6

Name	Plot	Equation
Identity		$f(x) = x$
Binary step		$f(x) = \begin{cases} 0 & \text{for } x < 0 \\ 1 & \text{for } x \geq 0 \end{cases}$
Logistic (a.k.a. Sigmoid or Soft step)		$f(x) = \sigma(x) = \frac{1}{1+e^{-x}}$ [1]
TanH		$f(x) = \tanh(x) = \frac{(e^{x} - e^{-x})}{(e^{x} + e^{-x})}$
Rectified linear unit (ReLU)[10]		$f(x) = \begin{cases} 0 & \text{for } x < 0 \\ x & \text{for } x \geq 0 \end{cases}$
Leaky rectified linear unit (Leaky ReLU)[11]		$f(x) = \begin{cases} 0.01x & \text{for } x < 0 \\ x & \text{for } x \geq 0 \end{cases}$

Name	Equation
Softmax	$f_i(\vec{x}) = \frac{e^{x_i}}{\sum_{j=1}^{J} e^{x_j}}$ for $i = 1, \ldots, J$
Maxout[23]	$f(\vec{x}) = \max_i x_i$

$f(x)$

$\downarrow$

$\sum(W \cdot I + b)$

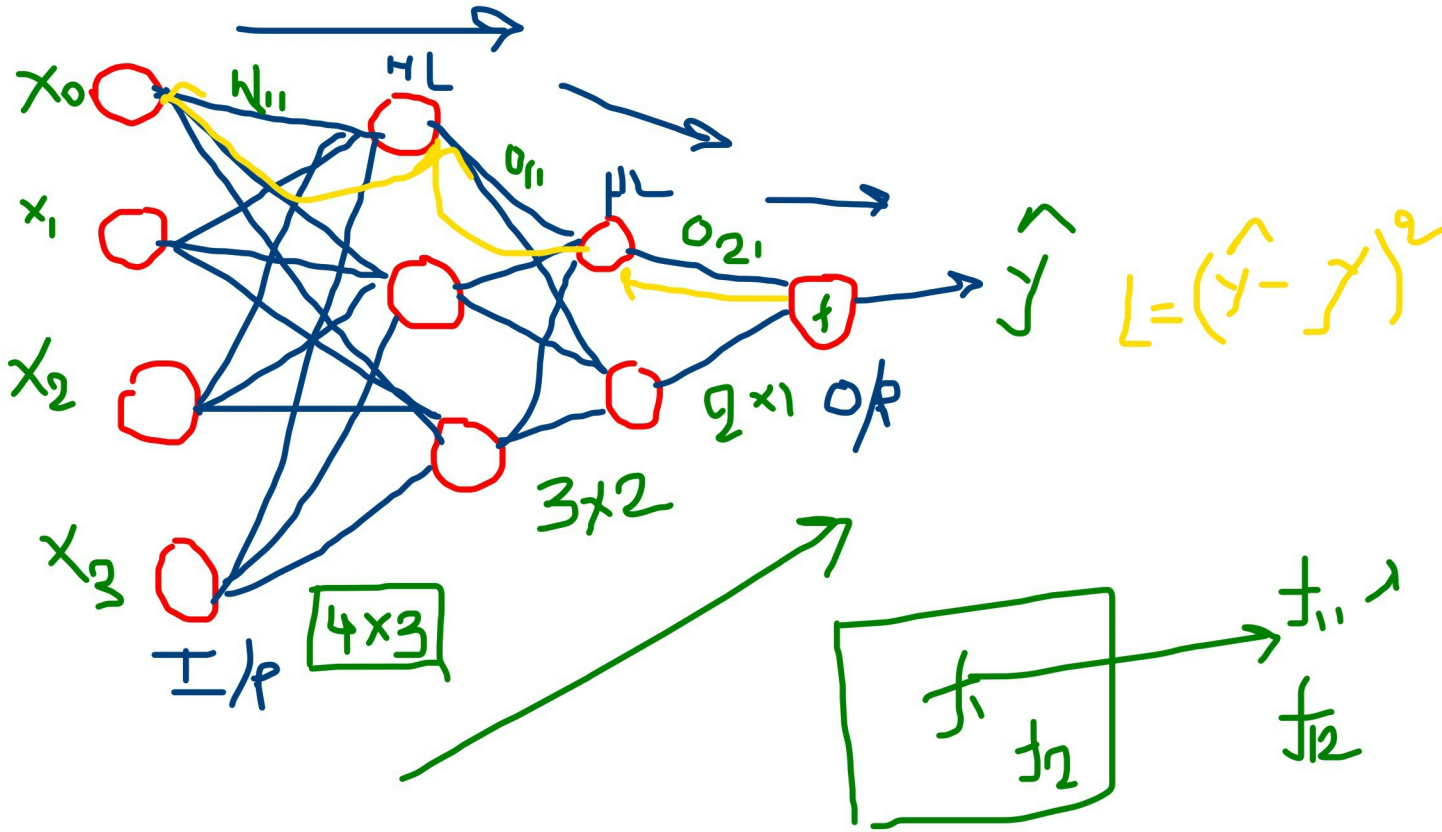
Choosing the right Activation Function

INNOMATICS RESEARCH LABS

- Sigmoid functions and their combinations generally work better in the case of classifiers
- Sigmoid and tanh functions are sometimes avoided due to the vanishing gradient problem
- ReLU function is a general activation function and is used in most cases these days
- If we encounter a case of dead neurons in our networks the leaky ReLU function is the best choice
- Always keep in mind that ReLU function should only be used in the hidden layers
- As a rule of thumb, you can begin with using ReLU function and then move over to other activation functions in case ReLU doesn't provide with optimum results

FFNN

- Input layer
- Hidden layer
- Output layer
- Importance of hidden layer



Handson

```
In [1]: import numpy as np

        u = np.array([3,4])
        v = np.array([30,40])

In [2]: print(u, v)

        [3 4] [30 40]

In [3]: # Length / magnitude of a vector

        print(np.linalg.norm(u))
        print(np.linalg.norm(v))

        5.0
        50.0

In [4]: def direction(x):
            return x/np.linalg.norm(x)

In [5]: w = direction(u)
        z = direction(v)
        print(w)
        print(z)

        [0.6 0.8]
```

```
In [15]: def dot_product(X,y):
             result = 0
             for i in range (len(X)):
                 result = result + X[i]*y[i]
             return(result)

In [14]: X = [3,5]
         y = [8,2]

         print(dot_product(X,y))  # should get the same result as geometric dot product.
                                  # instead of finding the theta, simply multiply the magnitudes of the respective dimensions

         34

In [ ]: X = np.array([0, 1, 2, 3])
        y = np.array([-1, 0.2, 0.9, 2.1])
        A = np.vstack([X, np.ones(len(X))]).T
        print(A)
        m, c = np.linalg.lstsq(A, y)[0]
        print(m)  # recall m is tan(@) and very close to 1 indicating the theta to be 45 degrees

In [ ]: # 3x+4y = 6
        # 5x+7y = 8
        # 8x+9y = 3
```

```
In [ ]: # Dot product / projection of one vector on other, influence of one over the other
        import math

        def geometric_dot_product(x,y, theta):
            x_norm = np.linalg.norm(x)
            y_norm = np.linalg.norm(y)
            return x_norm * y_norm * math.cos(math.radians(theta))

In [ ]:

        theta = 45   # reduce theta, dot product will increase
        X = [3,5]
        y = [8,2]
        x_norm = np.linalg.norm(X)
        y_norm = np.linalg.norm(y)
        print(x_norm*y_norm)
        print(geometric_dot_product(X, y, theta))

In [ ]: def dot_product(X,y):
            result = 0
            for i in range (len(X)):
                result = result + X[i]*y[i]
            return(result)

In [ ]: X = [3,5]
```

```
In [18]: A = np.vstack([X, np.ones(len(X))])
         print(A)

         [[0. 1. 2. 3.]
          [1. 1. 1. 1.]]

In [19]: # 3x+4y = 6
         # 5x+7y = 8
         # 8x+9y = 3

         a = np.array([[3,4], [5, 7]])
         b= np.array([[6], [8]])
         val = np.linalg.solve(a,b)
         c = np.linalg.lstsq(a,b)[0]
         print(c)
         print(val)
         print(3*c[0]+4*c[1])

         [[10.]
          [-6.]]
         [[10.]
          [-6.]]
         [6.]
         C:\Users\R1007398\AppData\Local\Continuum\anaconda3\lib\site-packages\ipykernel_launcher.py:8: FutureWarning: `rcond`
         will change to the default of machine precision times ``max(M, N)`` where M and N are the input matrix dimensions.
         To use the future default and silence this warning we advise to pass `rcond=None`, to keep using the old, explicitly
```

```
print(A)
```

```
[[0. 1. 2. 3.]
 [1. 1. 1. 1.]]
```

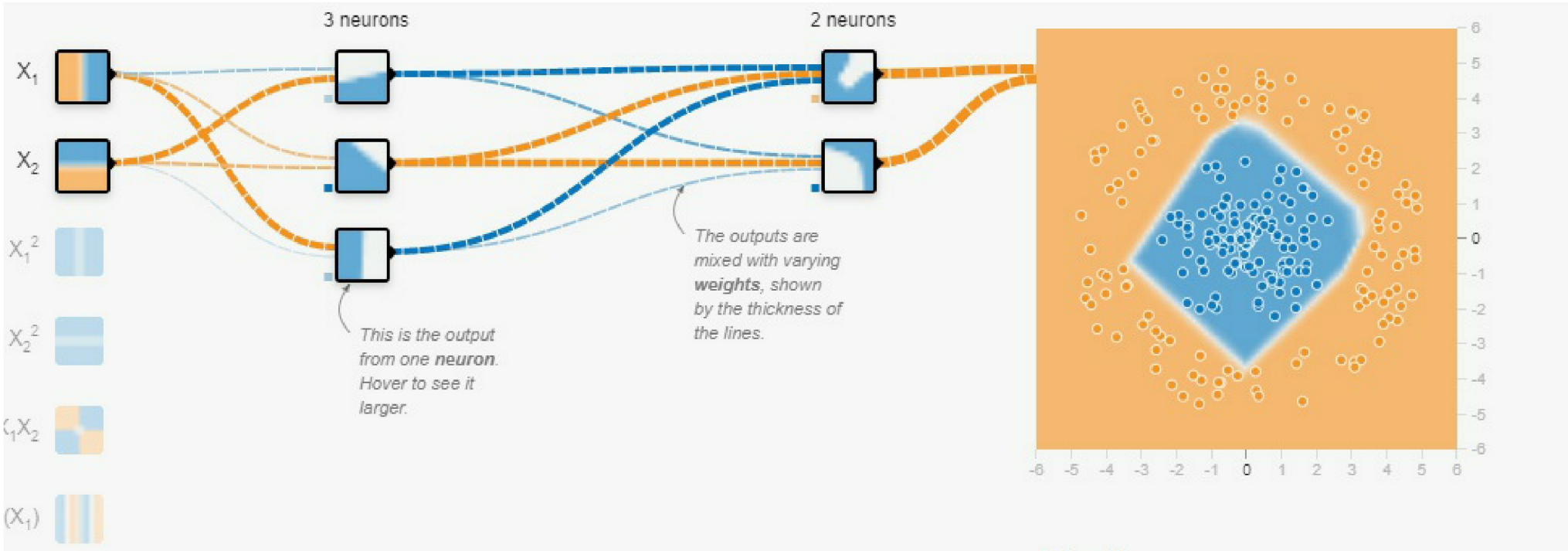
In [22]:

```
# 3x+4y = 6
# 5x+7y = 8
# 8x+9y = 3

a = np.array([[3,4], [5, 7], [8, 9]])
b= np.array([[6], [8], [3]])
# val = np.linalg.solve(a,b)
c = np.linalg.lstsq(a,b)[0]
print(c)

print(3*c[0]+4*c[1])
```

```
[[-5.17687075]
 [ 4.95238095]]
[4.27891156]
C:\Users\R1007398\AppData\Local\Continuum\anaconda3\lib\site-packages\ipykernel_launcher.py:8: FutureWarni
will change to the default of machine precision times ``max(M, N)`` where M and N are the input matrix dim
To use the future default and silence this warning we advise to pass `rcond=None`, to keep using the old,
d=-1`.
```

In [ ]:

```
import matplotlib.pyplot as plt
```


3 neurons
2 neurons
X1
X2
X1^2
X2^2
X1X2
sin(X1)
This is the output from one neuron. Hover to see it larger.
The outputs are mixed with varying weights, shown by the thickness of the lines.
6
5
4
3
2
1
0
-1
-2
-3
-4
-5
-6
-6 -5 -4 -3 -2 -1 0 1 2 3 4 5 6

Training the NN

epochs

- Given the x(i) and y (i)
- Think of what NN design and hyper parameter design might work
- Form a neural network
  - F(x)
- Compute weights as estimating the y for all samples
- Compute the loss
- Tweak w to reduce the loss
- Repeat the last three steps

Error and loss function

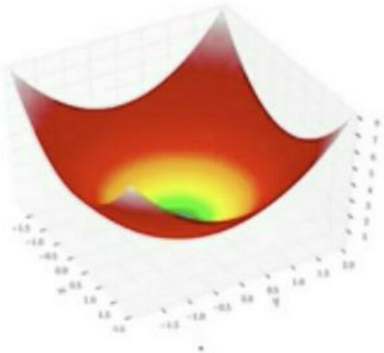
$L = (\hat{y} - y)^2$

y = Actual

$\hat{y}$ = Predicted

- In general error/loss for a NN is difference between actual and predicted.
- The goal is to minimize the loss
- Loss function is a function that is used to calculate the error
- You have to choose the loss function based on the problem you have at hand.
- Loss functions are different for classification and regression

Optimization



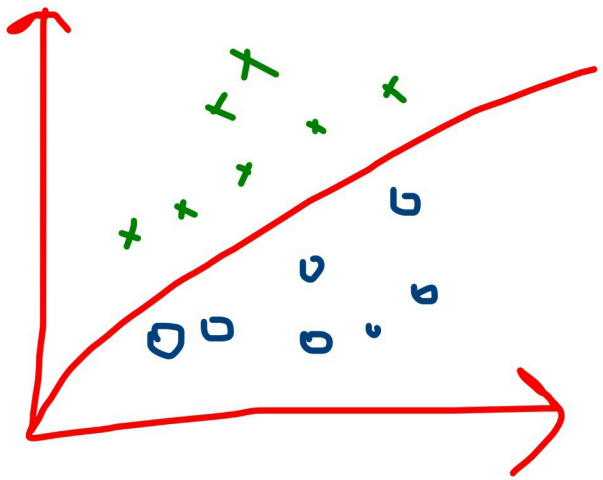
Optimization

In optimization, the main aim is to find weights that reduce loss

Gradient Descent

- Gradient is calculated by optimizing function
- Gradient is the change is loss with change in weights
- The weights are modified according to the calculated gradient
- Same process is repeated until minima is reached

Gradient Descent



$$Y = mx + C$$

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

$$SSE = Sum((mx + C) - y)^2$$

$$MSE = \frac{Sum((mx + C) - y)^2}{n}$$

Partial Derivation

$$\frac{\partial L}{\partial m} \qquad \frac{\partial L}{\partial C}$$

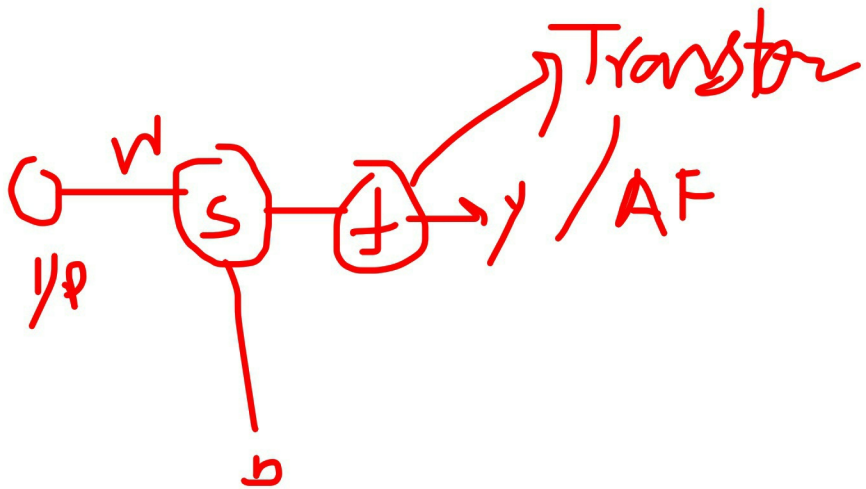
Emperical Equation

Eq1: $m*sum(X^2) + sum(X)C = xy$

Eq2: $m*sum(x) + nC = y$

Eg:

X	Y	XY	X^2
1	1	1	1
3	3	9	9
5	4	20	25
Σ= 9	8	30	35



EQ1: $35M + 9C = 30$

EQ2: $9M + 3C = 8$

Using Simaltaneous Equation

Multiplying Eq2 with 3 =

$27M + 9C = 24$

$35M + 9C = 30$

$27M + 9C = 24$

$8M = 6$

$M = 6/8 = 3/4$

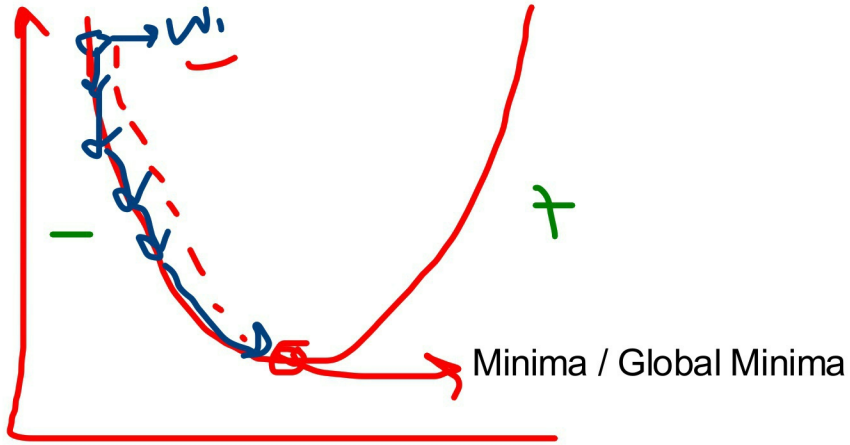
Putting all the values of M in Eq2

$9*3/4 + 3C = 8$

$C = 9/4$

$Y = 3/4 * Input + 9/4$

$Y = AF(SUM(3/4 *input + 9/4))$



Loss ↓ reduced

W = 1
b = 1
⇒ 1 x 1 + 1 = 2L

Weights Adjustment:

Wnew = Wold - Leanring Rate * $\frac{\partial MSE}{\partial W_{old}}$

LR : 0.01 - 0.03

Wnew = Wold - Lerning Rate * $\frac{\partial MSE}{\partial W_{old}}$

bias = bias - learning Rate * ∂ MSE / ∂b

Eg: Area | Bedroom | Price

600	2	40L
500	1	34L

Price = W1 * Area + W2 * Bedroom + bias = 603

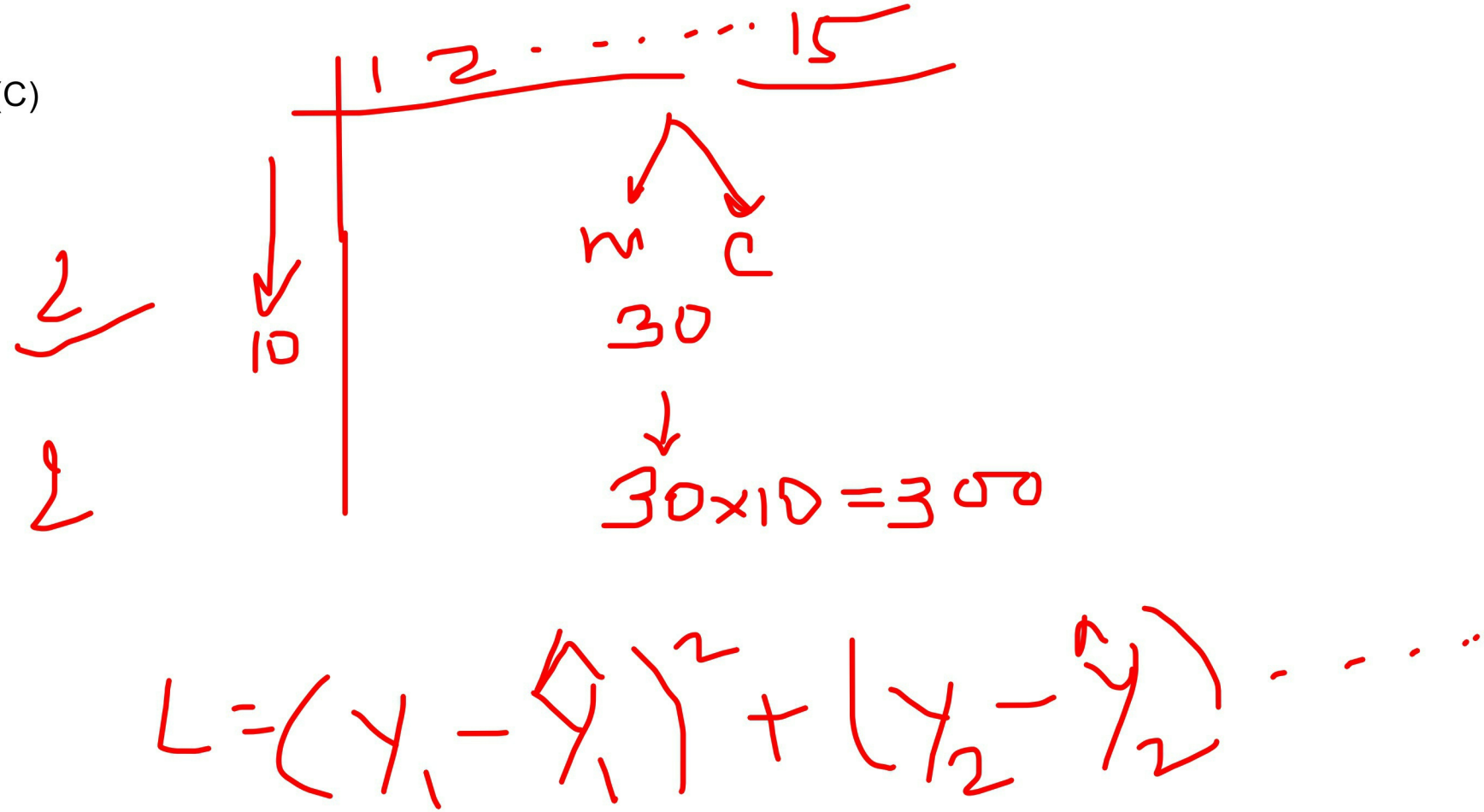
Error : (PriceA - PriceP)^2

$(140L - 603)^2 = 116$

Total Error = T1 + T2 / 2

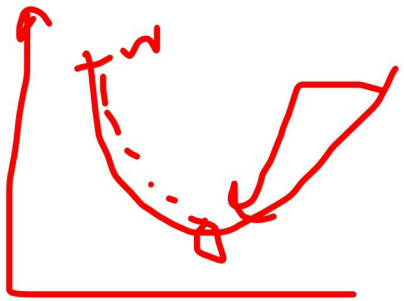
1 Derviating with respect to slope (m)
Derivating With respect to constant(C)

For each parameter we come to number of derivation = 2



$$L=(y_1-\hat{y}_1)^2+(y_2-\hat{y}_2)\cdots$$

Batch Gradient Descent



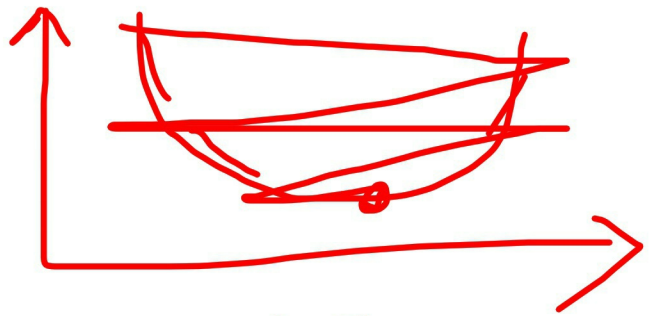
[100]

20 20

~ GD

epoch – less

Stochastic Gradient Descent



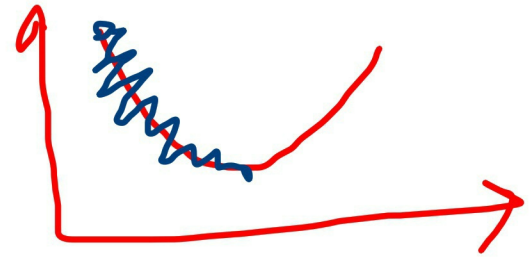
[3/4]

bias

Time to update weights increases

epoch – high

Mini batch



[40]

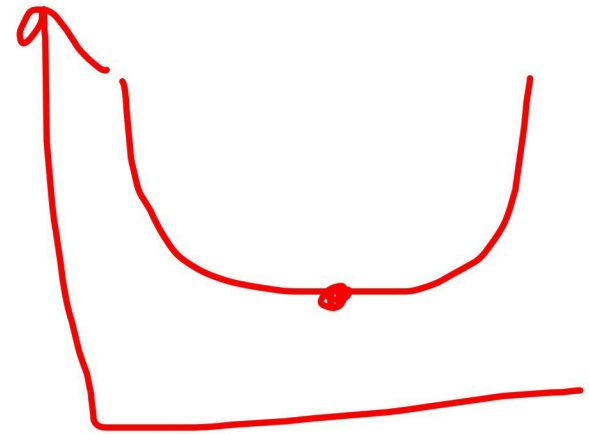
Time to update weights reduces but impact as batch size increases & Computation of GD also reduces

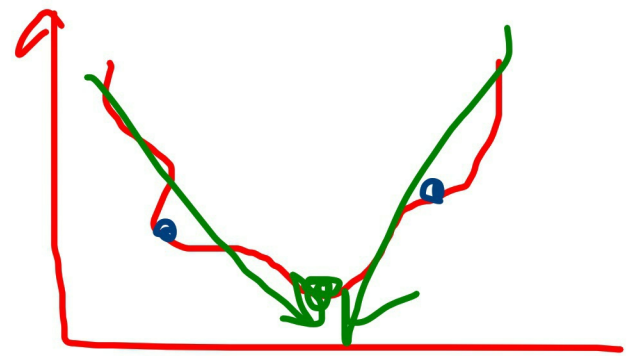
epoch – high

Momentum

Gradient descent variations

- Batch Gradient descent
- Stochastic gradient descent
- Mini-batch gradient descent





Back Propogation in NN

- Backprop is used while training feed forward NN
- It helps in efficiently calculating the gradient of the loss function with respect to weights
- It helps in minimizing loss by updating weights

Learning Rate and Momentum

- LR is a hyper parameter to what extent newly acquired weights overrides the existing weights , lies between 0 to 1

helps to reach global minima faster

Small learning rate gives more epochs and make smaller change in the weights to each update.

large learning rate makes rapid change hence it be less epochs

- Momentum is used to decide the weight on nodes from previous iterations. It helps in improving training speed and also avoiding local minimas*

Momentum =

Exponential Moving Average of weights with every interation

Early stopping and model saving

- We need to monitor the loss on the validation set while training the NN and if we feel there is no change , then we should stop training.

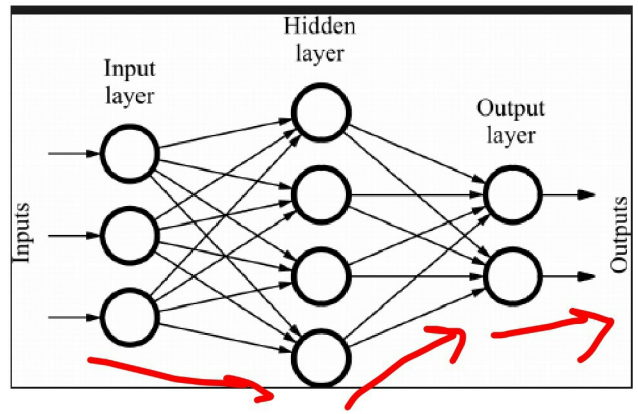
- Save all the necessary details to reuse the model
- Save the weights, biases and also the model architecture.

Different types of Neural Networks

INNOMATICS RESEARCH LABS

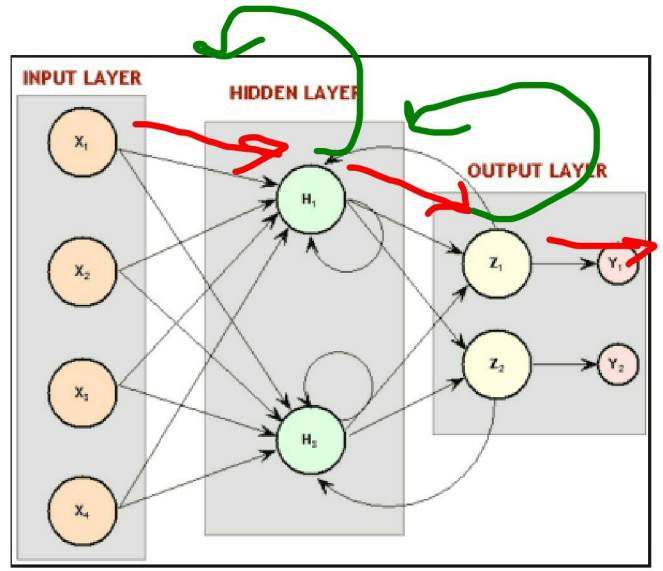
Feed-forward networks

- Feed-forward NNs allow signals to travel one way only; from input to output. There is no feedback (loops) i.e. the output of any layer does not affect that same layer.
- Feed-forward NNs tend to be straight forward networks that associate inputs with outputs. They are extensively used in pattern recognition.
- This type of organization is also referred to as bottom-up or top-down.



Feedback networks

- Feedback networks can have signals traveling in both directions by introducing loops in the network.
- Feedback networks are dynamic; their 'state' is changing continuously until they reach an equilibrium point.
- They remain at the equilibrium point until the input changes and a new equilibrium needs to be found.
- Feedback architectures are also referred to as interactive or recurrent, although the latter term is often used to denote feedback connections in single-layer organizations



A simple Neural Network

INNOMATICS RESEARCH LABS

Network Layers:
Input Layer - The activity of the input units represents the raw information that is fed into the network.

Hidden Layer - The activity of each hidden unit is determined by the activities of the input units and the weights on the connections between the input and the hidden units.
The weights between the input and hidden units determine when each hidden unit is active, and so by modifying these weights, a hidden unit can choose what it represents.

Output Layer - The behavior of the output units depends on the activity of the hidden units and the weights between the hidden and output units.

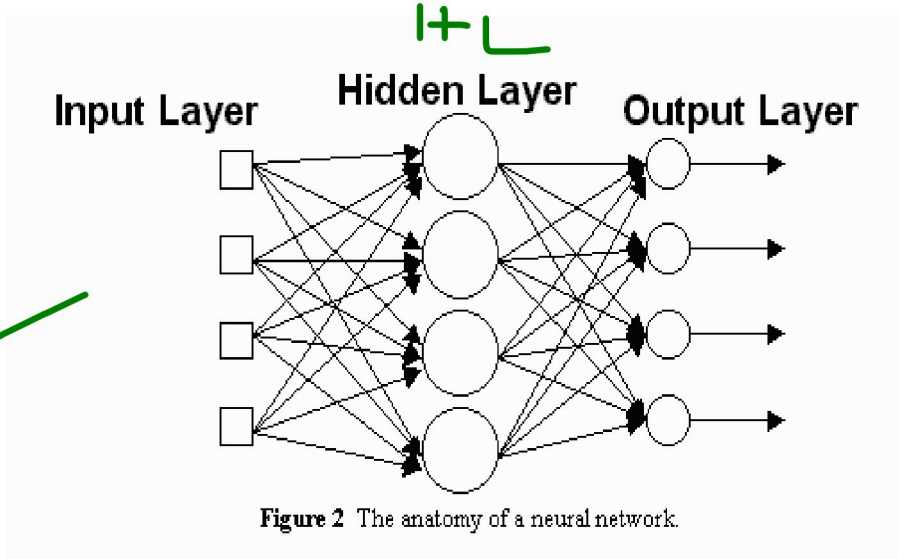
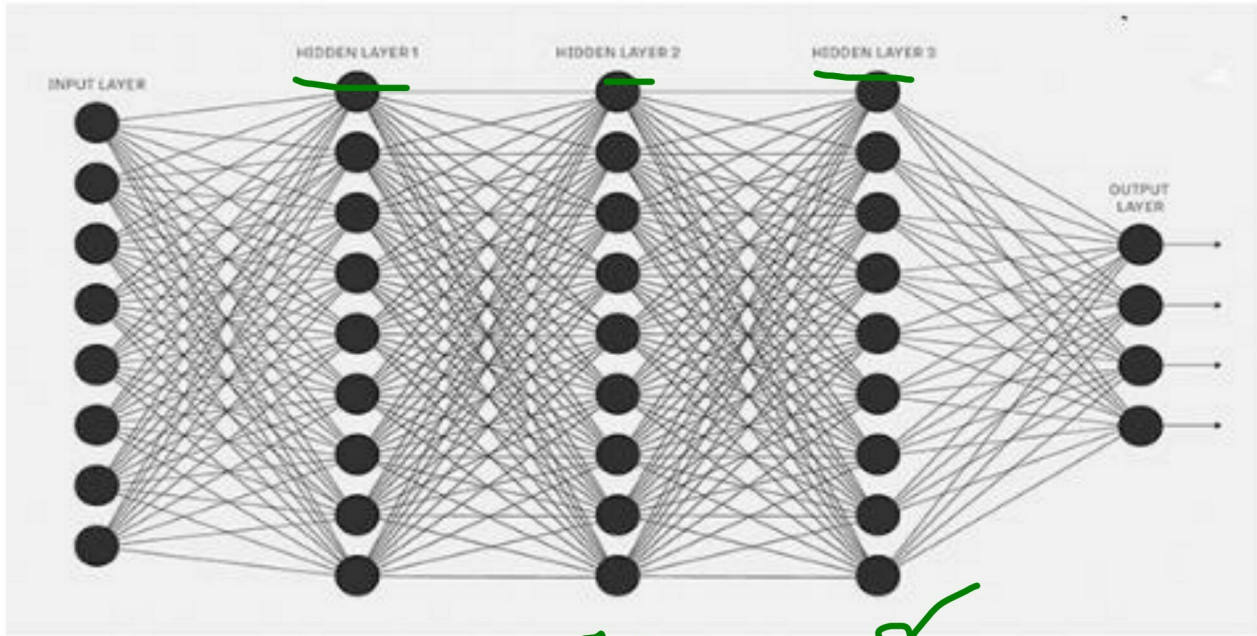


Figure 2 The anatomy of a neural network.

I/P — HL1 — HL2 — O/P

Neural Network with multiple hidden layers



OR

operation

Network Structure

INNOMATICS RESEARCH LABS

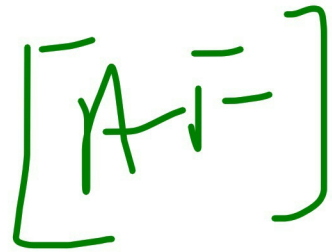
The number of layers and of neurons depend on the specific task. In practice this issue is solved by trial and error.

Two types of adaptive algorithms can be used:

- ✓ start from a large network and successively remove some neurons and links until network performance degrades.
- ✓ begin with a small network and introduce new neurons until performance is satisfactory.

Network Parameters

- ✓ How are the weights initialized?
- ✓ How many hidden layers and how many neurons?
- ✓ How many examples in the training set?



Weights:

In general, initial weights are randomly chosen, with typical values between -1.0 and 1.0 or -0.5 and 0.5.

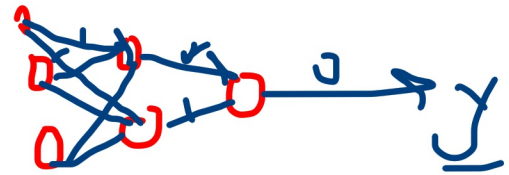
There are two types of NNs

- Fixed Networks – where the weights are fixed
- Adaptive Networks – where the weights are changed to reduce prediction error.

Size of Training Data:

The number of training examples should be at least five to ten times the number of weights of the network.

Training of NN



- The most basic method of training a neural network is trial and error.
- If the network isn't behaving the way it should, change the weighting of a random link by a random amount. If the accuracy of the network declines, undo the change and make a different one.
- It takes time, but the trial and error method does produce results.

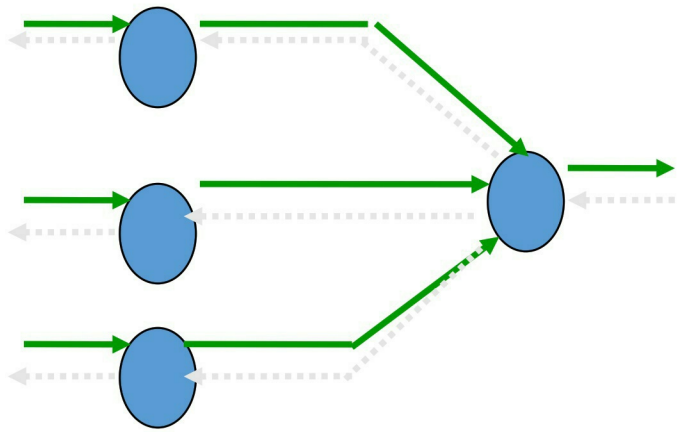
Backprop algorithm:

- The Backprop algorithm searches for weight values that minimize the total error of the network over the set of training examples (training set).
- Backprop consists of the repeated application of the following two passes:
  - **Forward pass:** in this step the network is activated on one example and the error of (each neuron of) the output layer is computed.
  - **Backward pass:** in this step the network error is used for updating the weights. Starting at the output layer, the error is propagated backwards through the network, layer by layer. This is done by recursively computing the local gradient of each neuron.

Back Propagation

INNOMATICS RESEARCH LABS

- Back-propagation training algorithm



Network activation
Forward Step

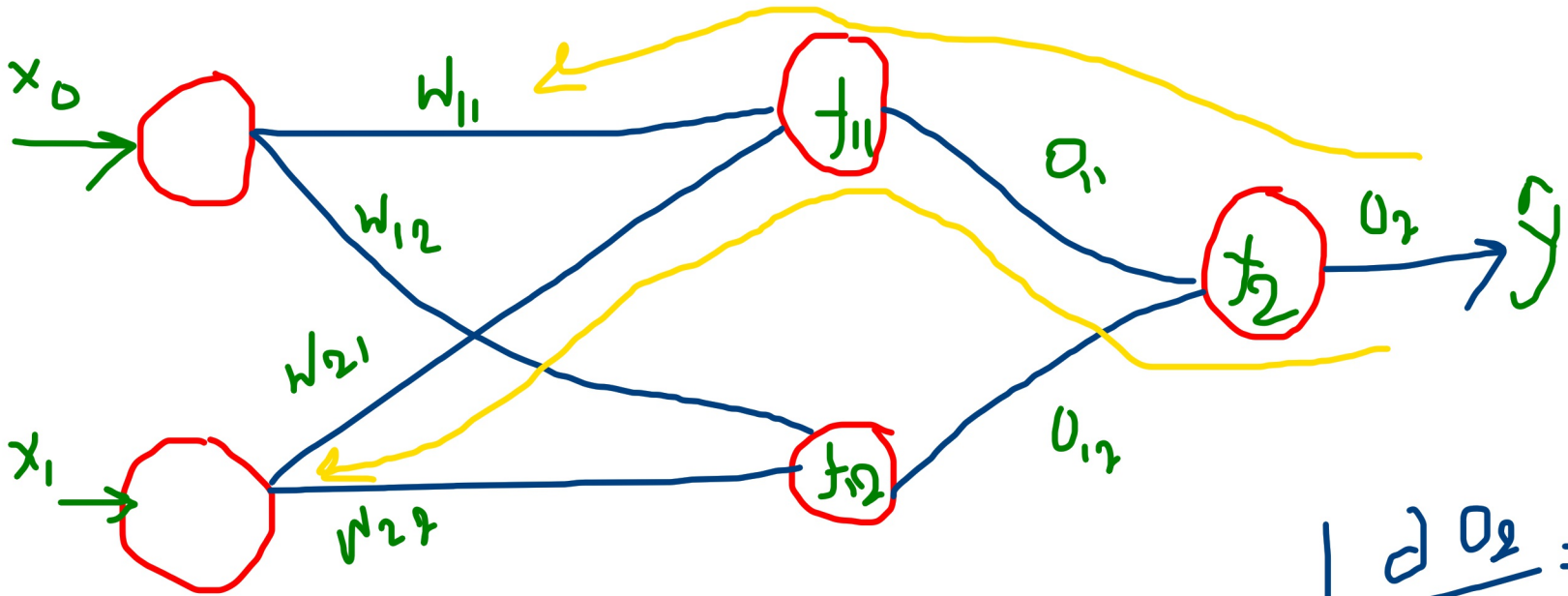
Error propagation
Backward Step

- Back-propagation adjusts the weights of the NN in order to minimize the network total mean squared error.

Chain Rule - Back propagation

Decaying

Expands



$$MSE = \frac{1}{n}(\hat{y} - y)^2$$

$$\frac{\partial o_2}{\partial w_{11}} = \frac{\partial o_2}{\partial o_{11}} \cdot \frac{\partial o_{11}}{\partial w_{11}}$$

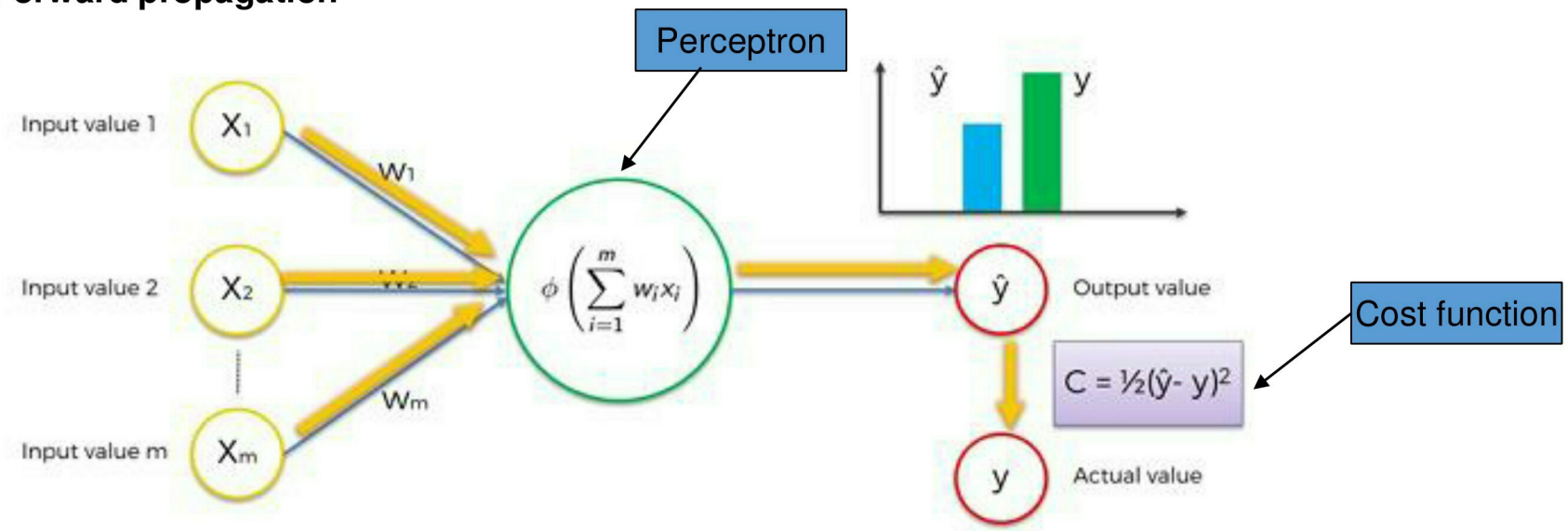
$$\frac{\partial o_2}{\partial w_{21}} = \frac{\partial o_2}{\partial o_{11}} \cdot \frac{\partial o_{11}}{\partial w_{21}}$$

Disadvantage of Neural Network

INNOMATICS RESEARCH LABS

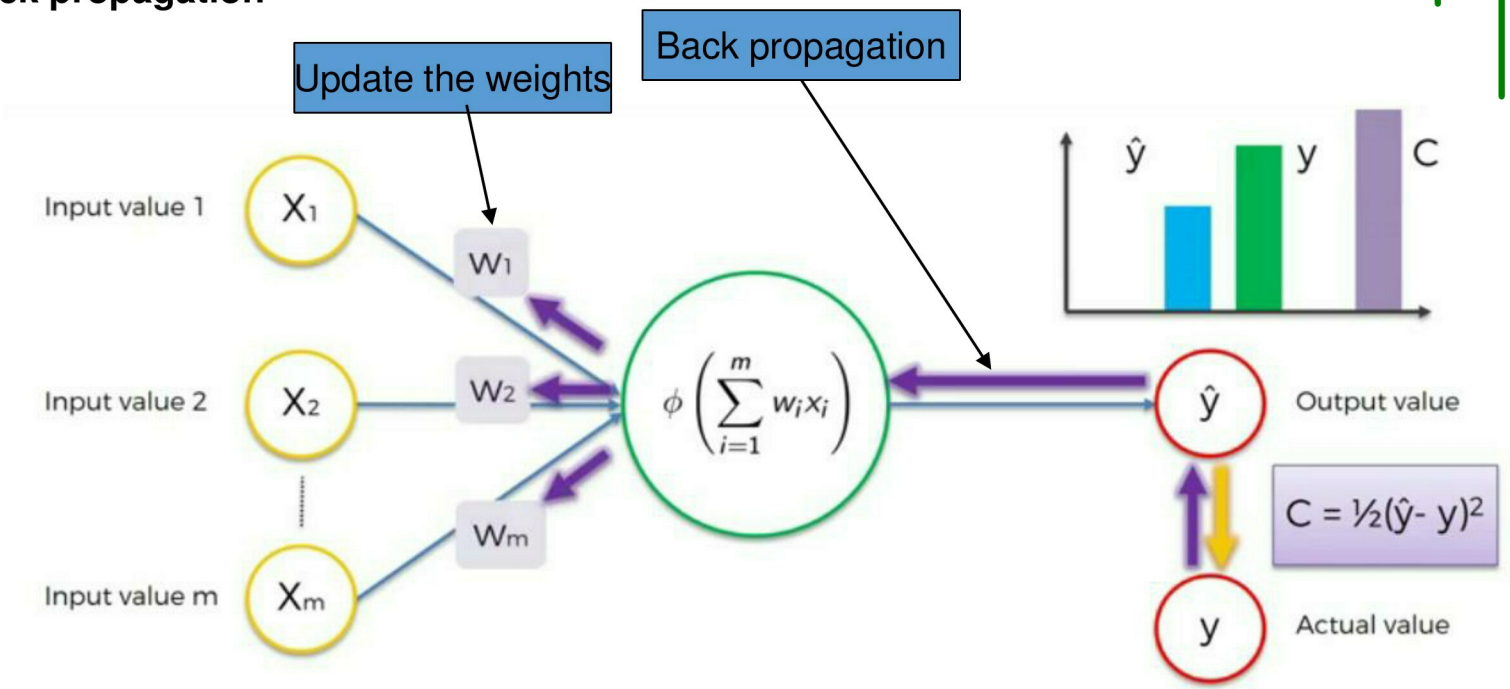
- The individual relations between the input variables and the output variables are not developed by engineering judgment so that the model tends to be a black box or input/output table without analytical basis.
- The sample size has to be large.
- Requires lot of trial and error so training can be time consuming.

Forward propagation

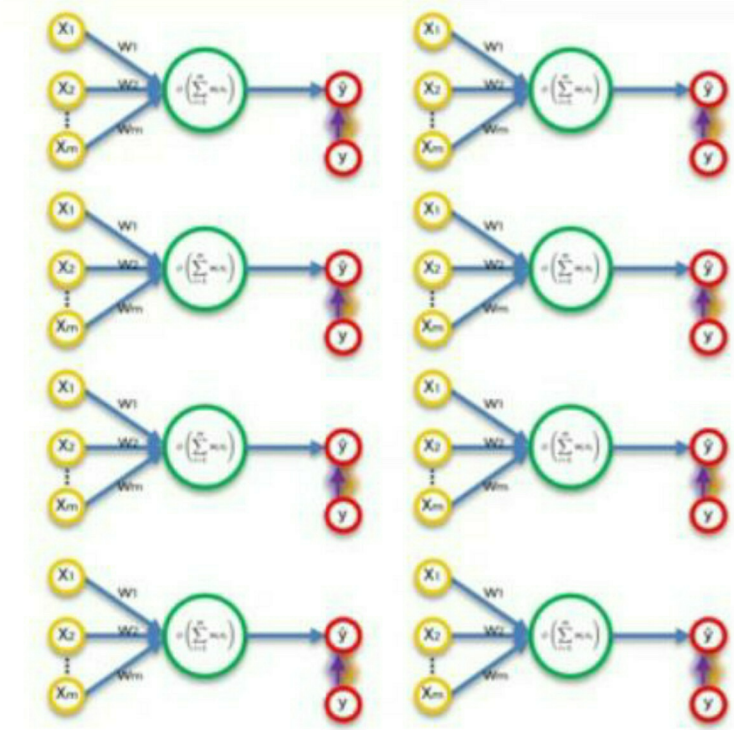


1.Feed features to the input layer
2.Calculate the weights of o each neuron
3.Predict the output
4.Compare the actual and predicted output

Back propagation



1. Calculate the cost function for predicted output in the forward propagation
2. Back propagate(Gradient descent) and adjust the weights of the layers so that cost function is minimised

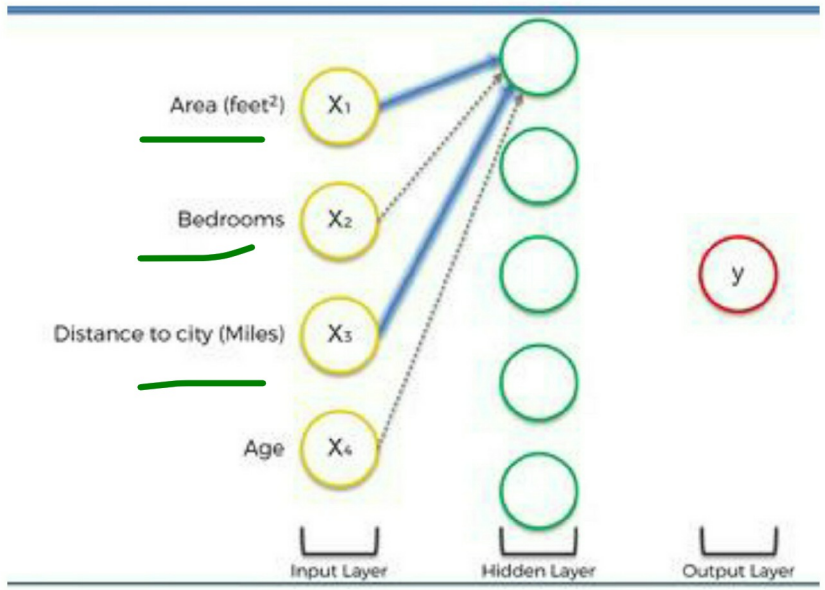


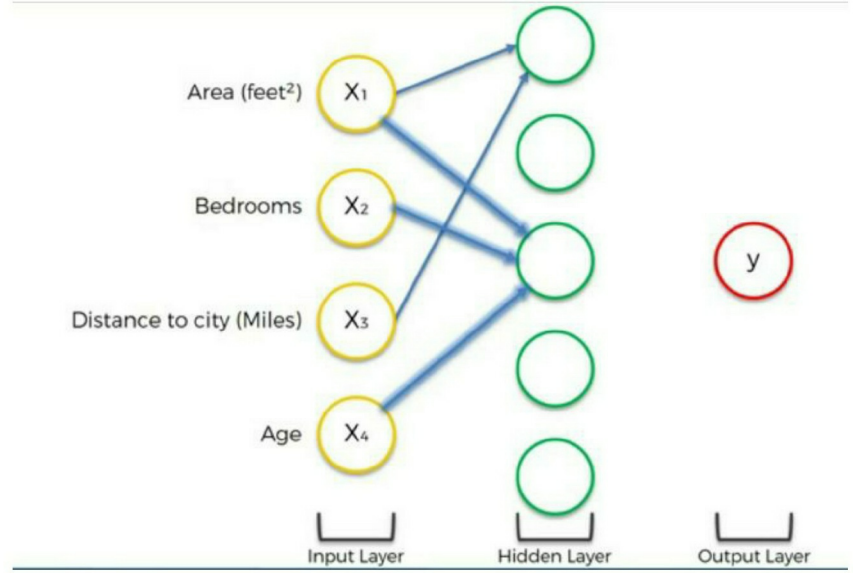
Row ID	Study Hrs	Sleep Hrs	Quiz	Exam
1	12	6	78%	93%
2	22	6.5	24%	68%
3	115	4	100%	95%
4	31	9	67%	75%
5	0	10	58%	51%
6	5	8	78%	60%
7	92	6	82%	89%
8	57	8	91%	97%

$$C = \sum \tfrac{1}{2}(\hat{y} - y)^2$$

- Predict the output for all observations
- Calculate the cost function
- Adjust the weight so that cost function is minimised
- Again predict the output with new adjusted weights and repeat the process untill the cost function is minimised

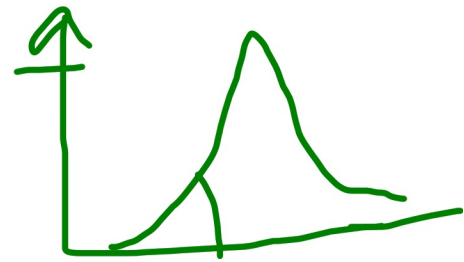
Age → 58 ft

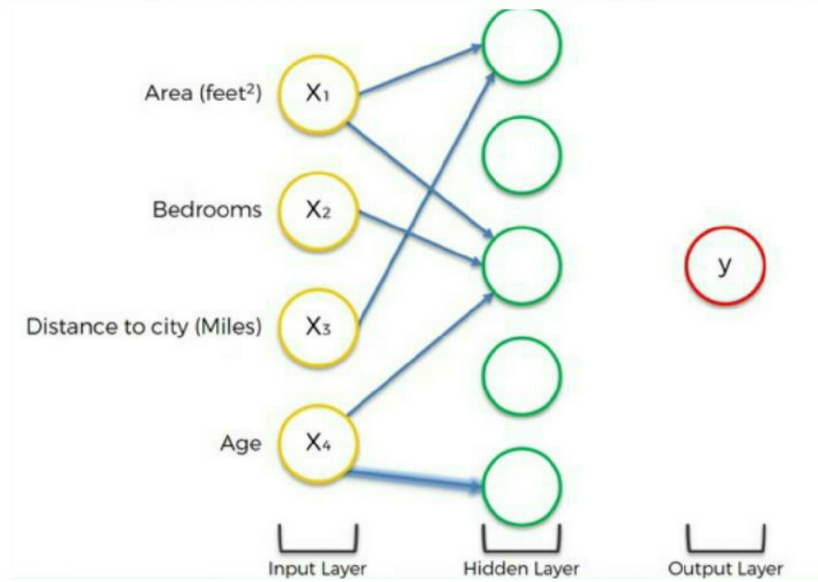


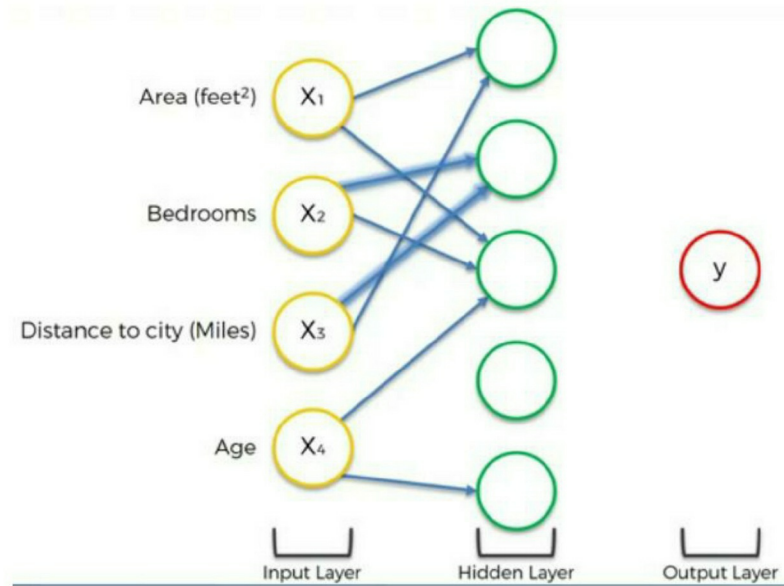


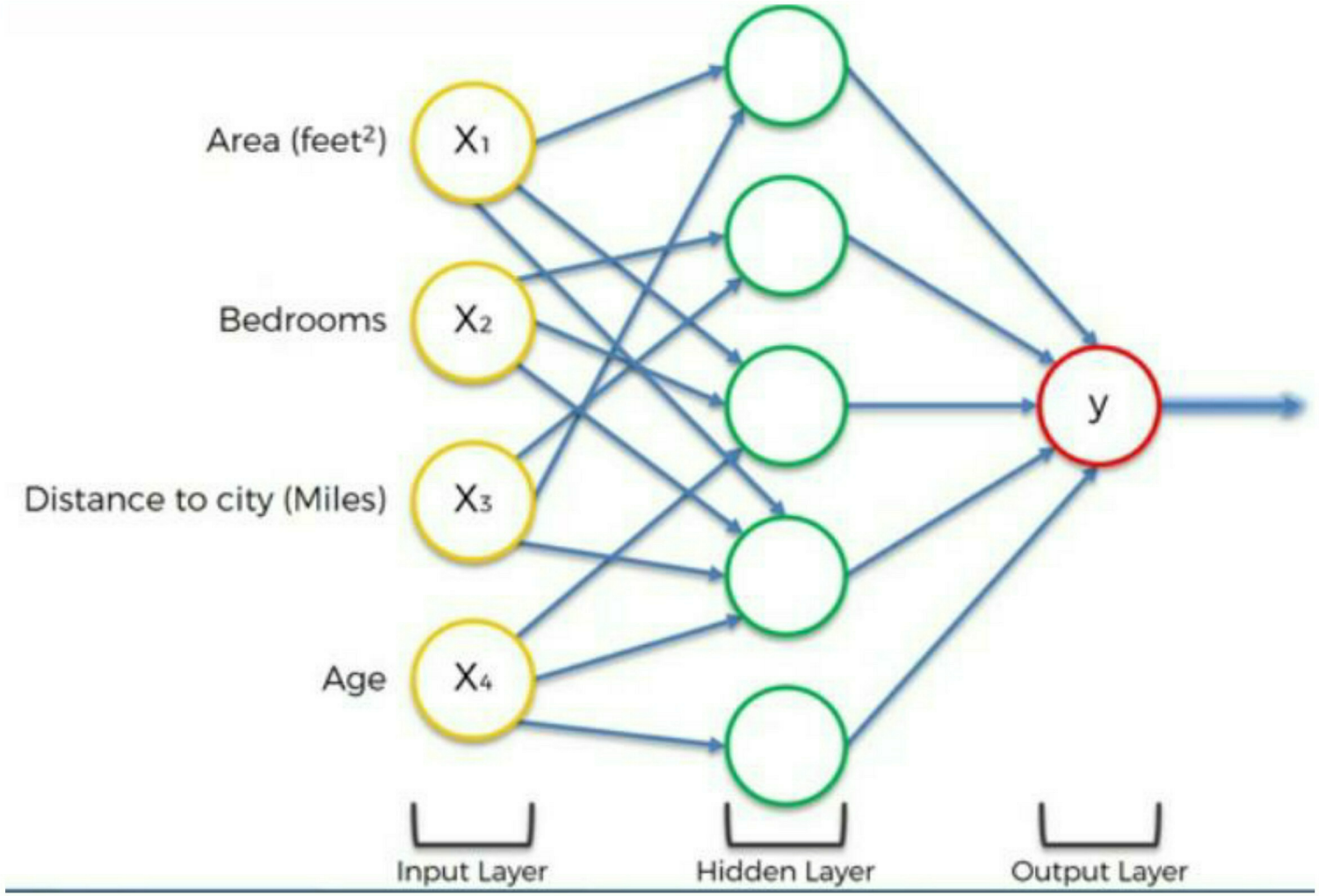
item

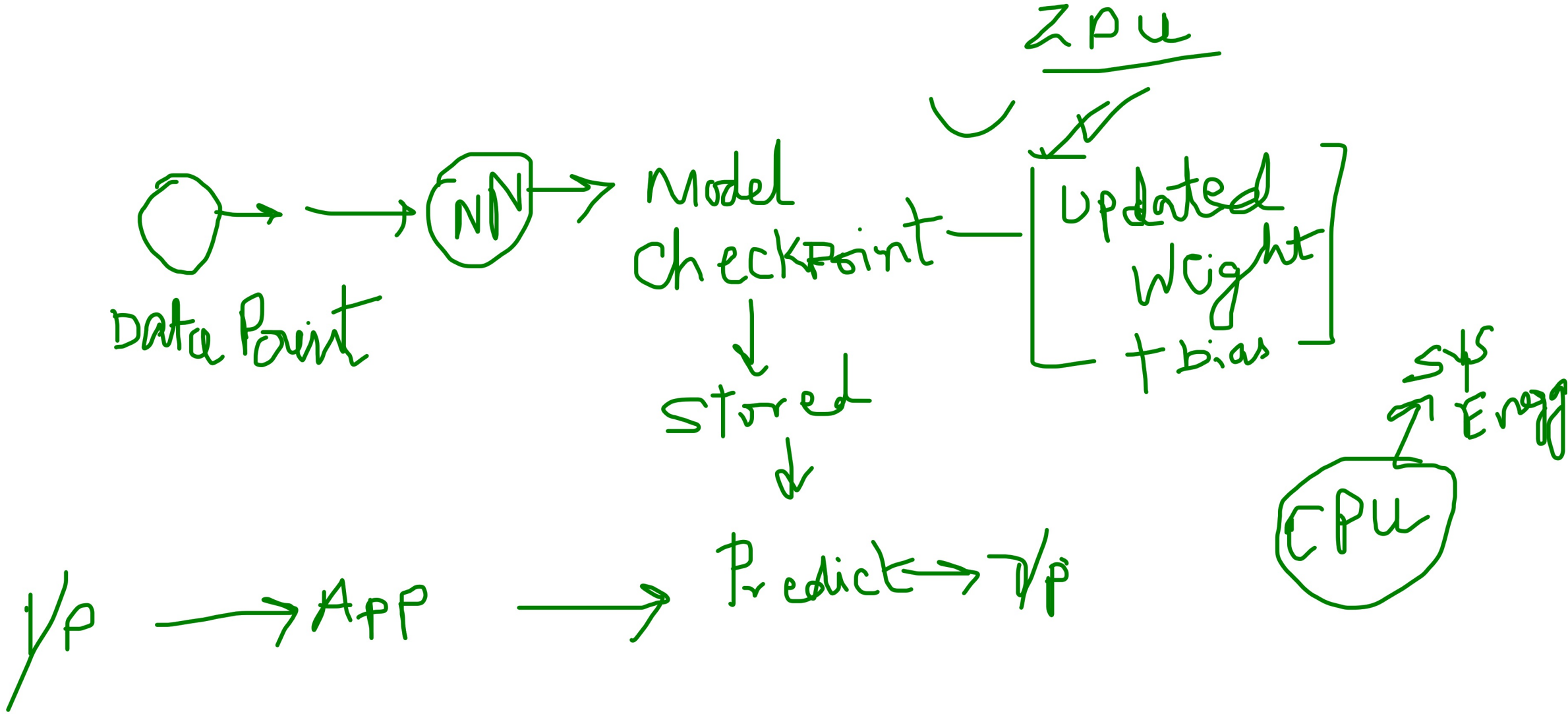
Rice





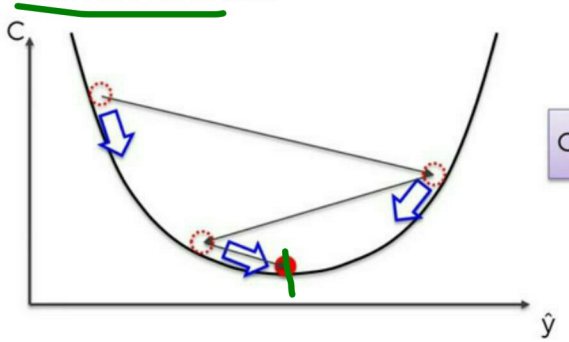



Area (feet²)
X1
Bedrooms
X2
Distance to city (Miles)
X3
Age
X4
y
Input Layer
Hidden Layer
Output Layer


ZPU
Data Point
NN
Model
CheckPoint
Updated
Weight
+ bias
Stored
Predict
I/P
App
O/P
CPU
sys
Engg

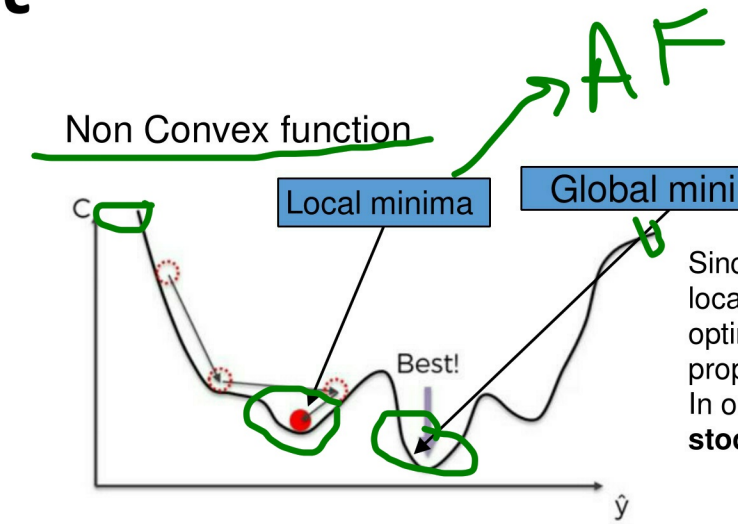
Gradient Descent

Convex function

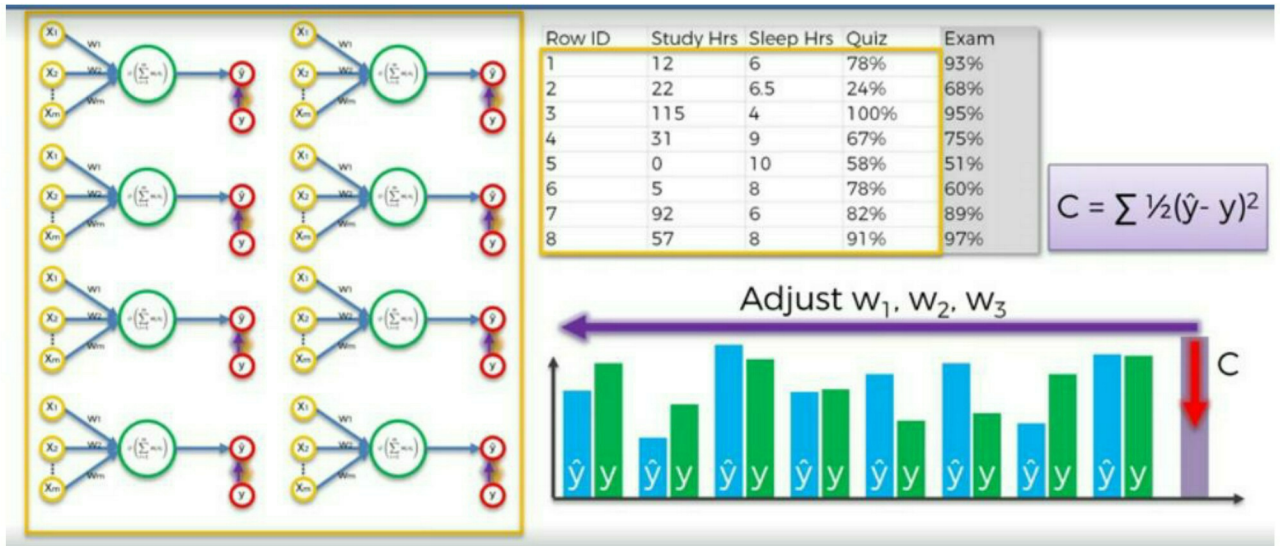


$C = \frac{1}{2}(\hat{y} - y)^2$

Non Convex function

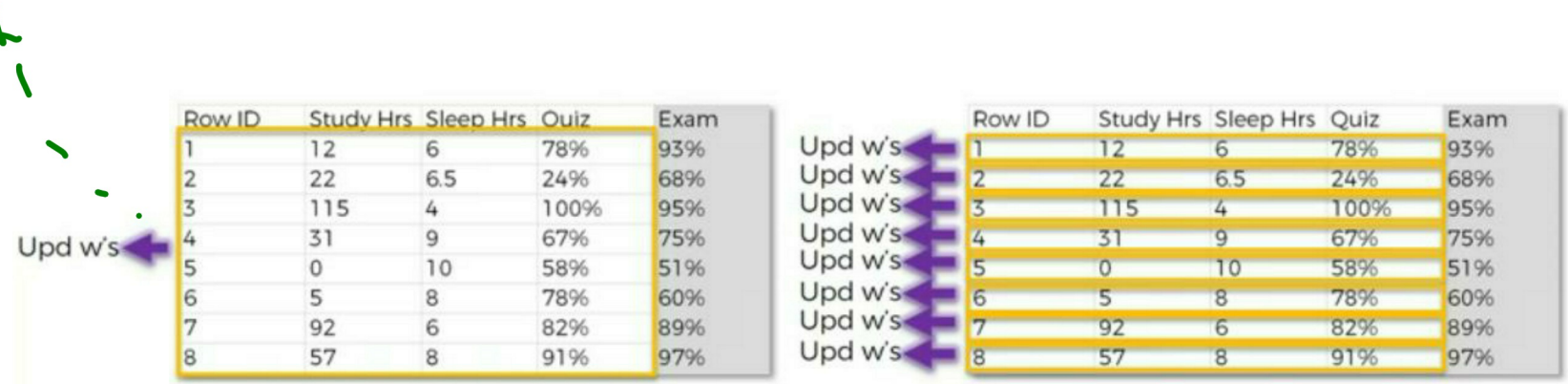


Since we ended in finding local minima we wont get the optimised weights in the back propagation.
In order to Avoid this we use **stochastic gradient descent**



We will calculate the cost function for all observations and update the weights to minimise the cost function .
This is called **Gradient descent /Batch Gradient descent**

Stochastic Gradient Descent



Entire Dataset

Batch Gradient Descent

Learning more dataset

Stochastic Gradient Descent

Random selection and updating weights

fast to update but more biased learning

Stochastic	Batch
Local minima problem is can be resolved	May end up in local minima
Fast	Slow
Less deterministic	More deterministic

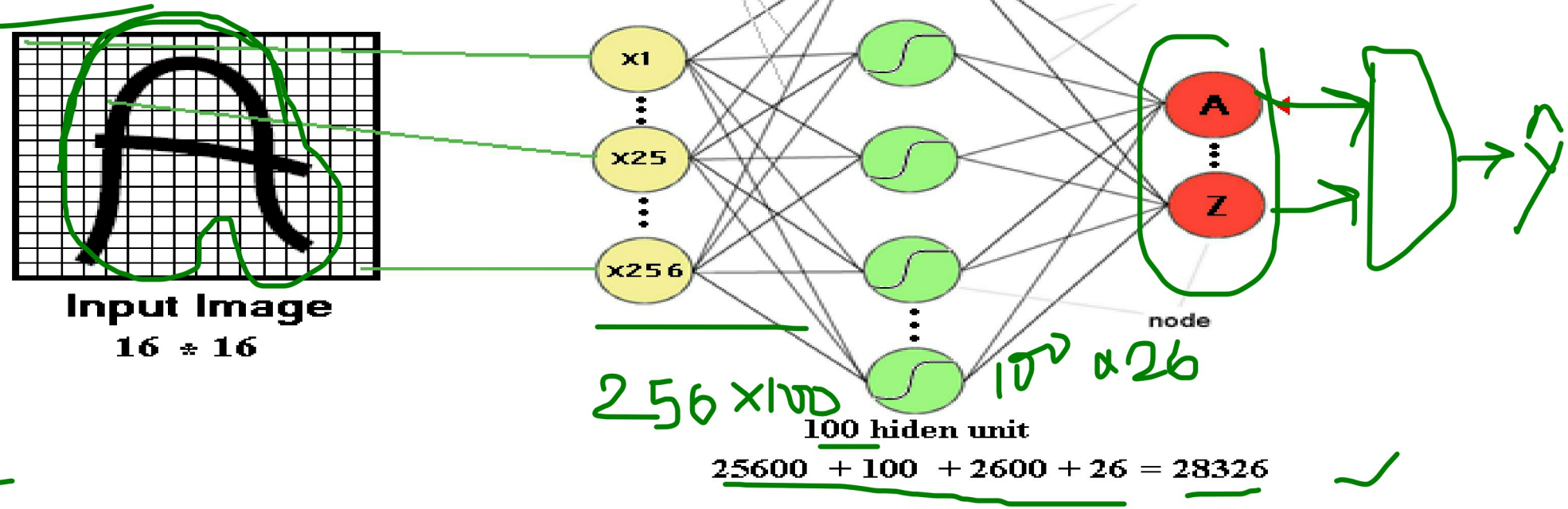
We will calculate the cost function for each observations and update the weights to minimise the cost function .
This is called **Stochastic Gradient descent**

Mini Batch Gradient Descent

	Row ID	Study Hrs	Sleep Hrs	Quiz	Exam
Upd w's	1	12	6	78%	93%
	2	22	6.5	24%	68%
	3	115	4	100%	95%
	4	31	9	67%	75%
Upd w's	5	0	10	58%	51%
	6	5	8	78%	60%
Upd w's	7	92	6	82%	89%
	8	57	8	91%	97%

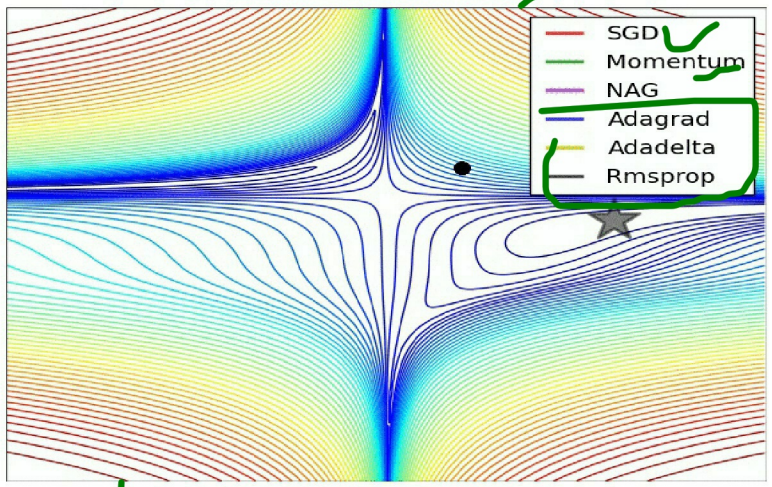
- We will first divide the data into mini batches (5 or 8 observations per batch)
- We will calculate the cost function for each mini observations and update the weights to minimise the cost function .
- This is called **Mini batch Gradient descent**

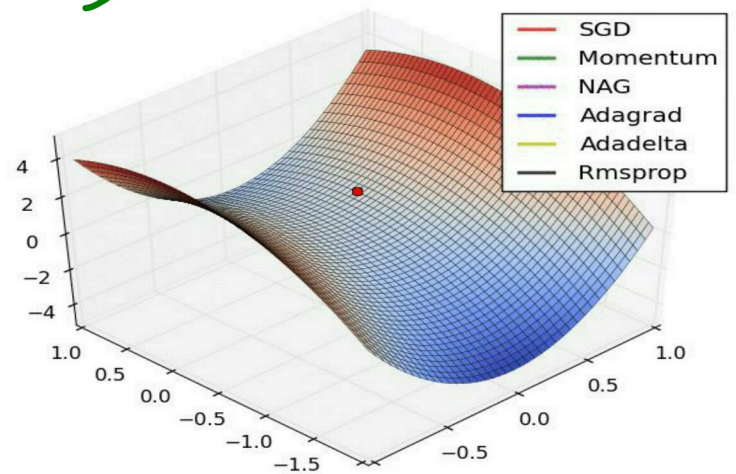
Number of Weights

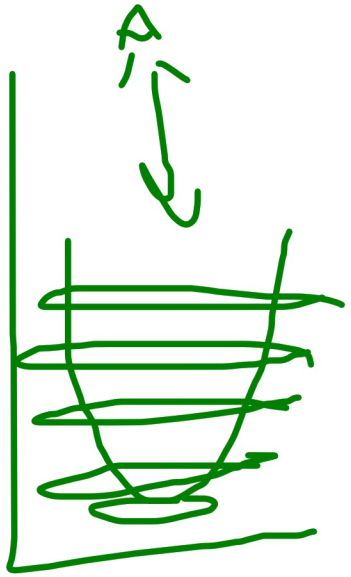


- 256 Inputs + 100 Hidden layers[connected to 256 inputs] -------Input side[25600+100(bias weights)]
  26 outputs + 100 Hidden layers[connected to 26 outputs]---- Output side[2600+26(bias weighs)]
  Total 28236 weights are calculated

Learning rate visualization







Animations that may help your intuitions about the learning process dynamics. **Left:** Contours of a loss surface and time evolution of different optimization algorithms. Notice the "overshooting" behavior of momentum-based methods, which make the optimization look like a ball rolling down the hill. **Right:** A visualization of a saddle point in the optimization landscape, where the curvature along different dimension has different signs (one dimension curves up and another down). Notice that SGD has a very hard time breaking symmetry and gets stuck on the top. Conversely, algorithms such as RMSprop will see very low gradients in the saddle direction. Due to the denominator term in the RMSprop update, this will increase the effective learning rate along this direction, helping RMSProp proceed.

Images and text source: cs231n stanford

Training ANN

INNOMATICS RESEARCH LABS

STEP 1: Randomly initialise the weights to small numbers close to 0 (but not 0).

STEP 2: Input the first observation of your dataset in the input layer, each feature in one input node.

STEP 3: Forward-Propagation: from left to right, the neurons are activated in a way that the impact of each neuron's activation is limited by the weights. Propagate the activations until getting the predicted result y.

STEP 4: Compare the predicted result to the actual result. Measure the generated error.

STEP 5: Back-Propagation: from right to left, the error is back-propagated. Update the weights according to how much they are responsible for the error. The learning rate decides by how much we update the weights.

STEP 6: Repeat Steps 1 to 5 and update the weights after each observation (Reinforcement Learning). Or:
Repeat Steps 1 to 5 but update the weights only after a batch of observations (Batch Learning).

STEP 7: When the whole training set passed through the ANN, that makes an epoch. Redo more epochs.

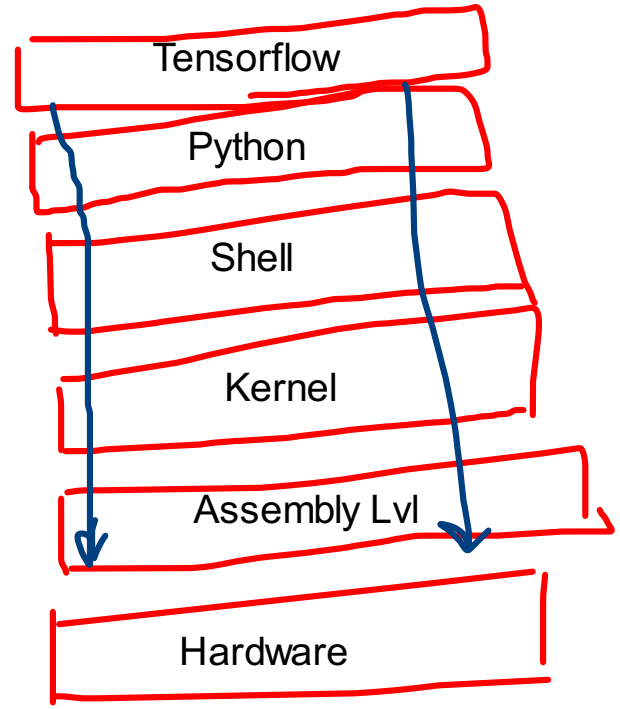
Top Deep Learning Frameworks

- **TensorFlow**- Google's open-source platform TensorFlow is perhaps the most popular tool for **Machine Learning** and **Deep Learning**.

- **PyTorch**- PyTorch is an open-source **Deep Learning framework** developed by Facebook.

- Keras - Tool can run on top of TensorFlow, Theano, Microsoft Cognitive Toolkit, and PlaidML

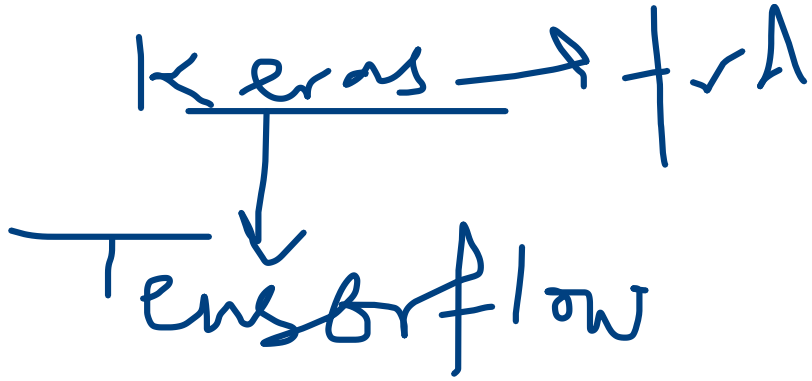
TensorFlow Overview



TensorFlow Architecture

Tensorflow architecture works in three parts:

- ✓ Preprocessing the data
- ✓ Build the model
- ✓ Train and estimate the model



It is called Tensorflow because it takes input as a multi-dimensional array, also known as **tensors**.

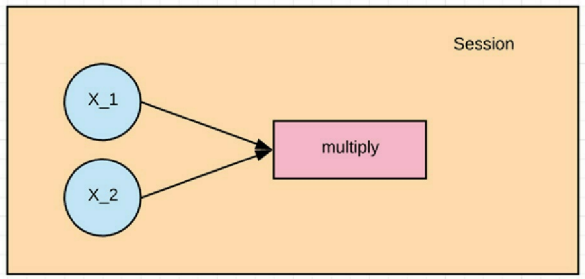
Can construct a sort of **flowchart** of operations (called a Graph) that you want to perform on that input

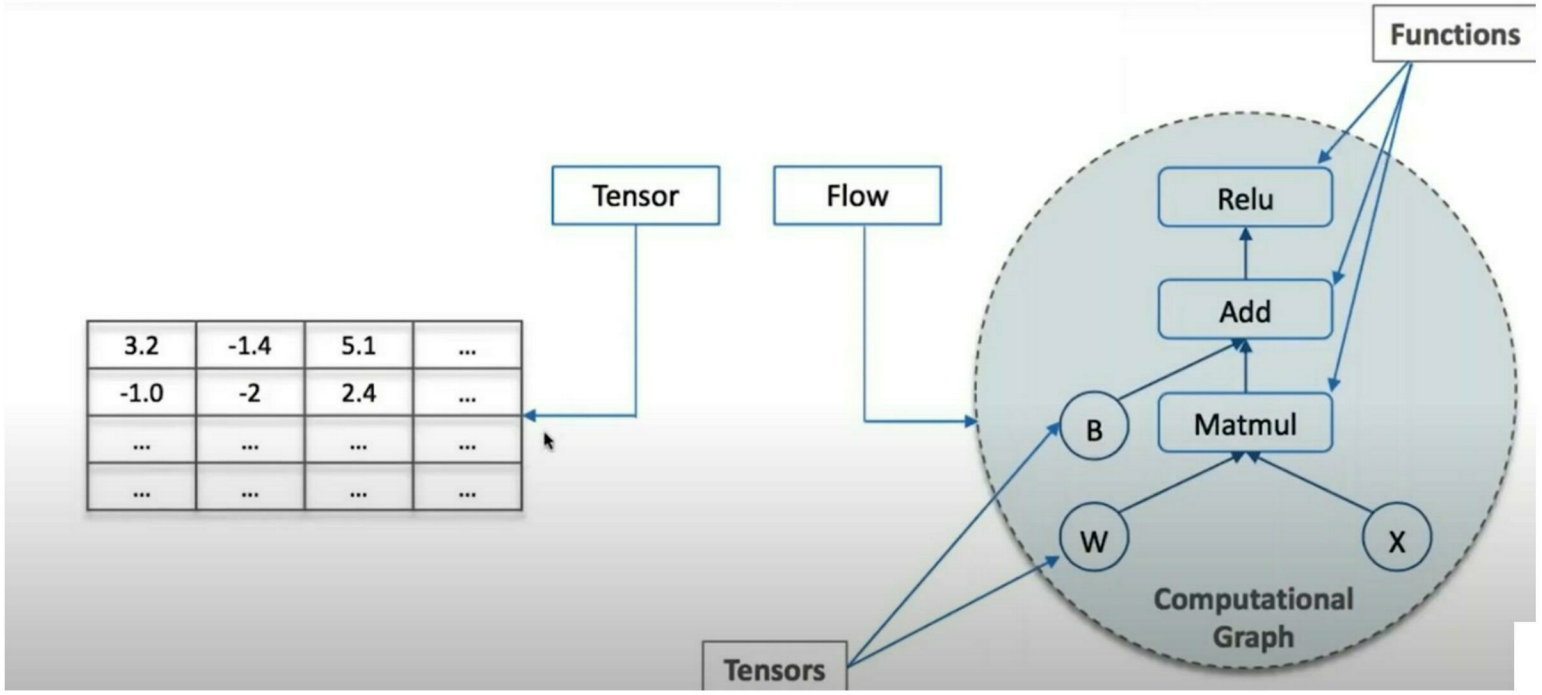
Components of TensorFlow

Generalization of vector and matrix to a higher dimension.

Tensor -input data or the result of a computation.

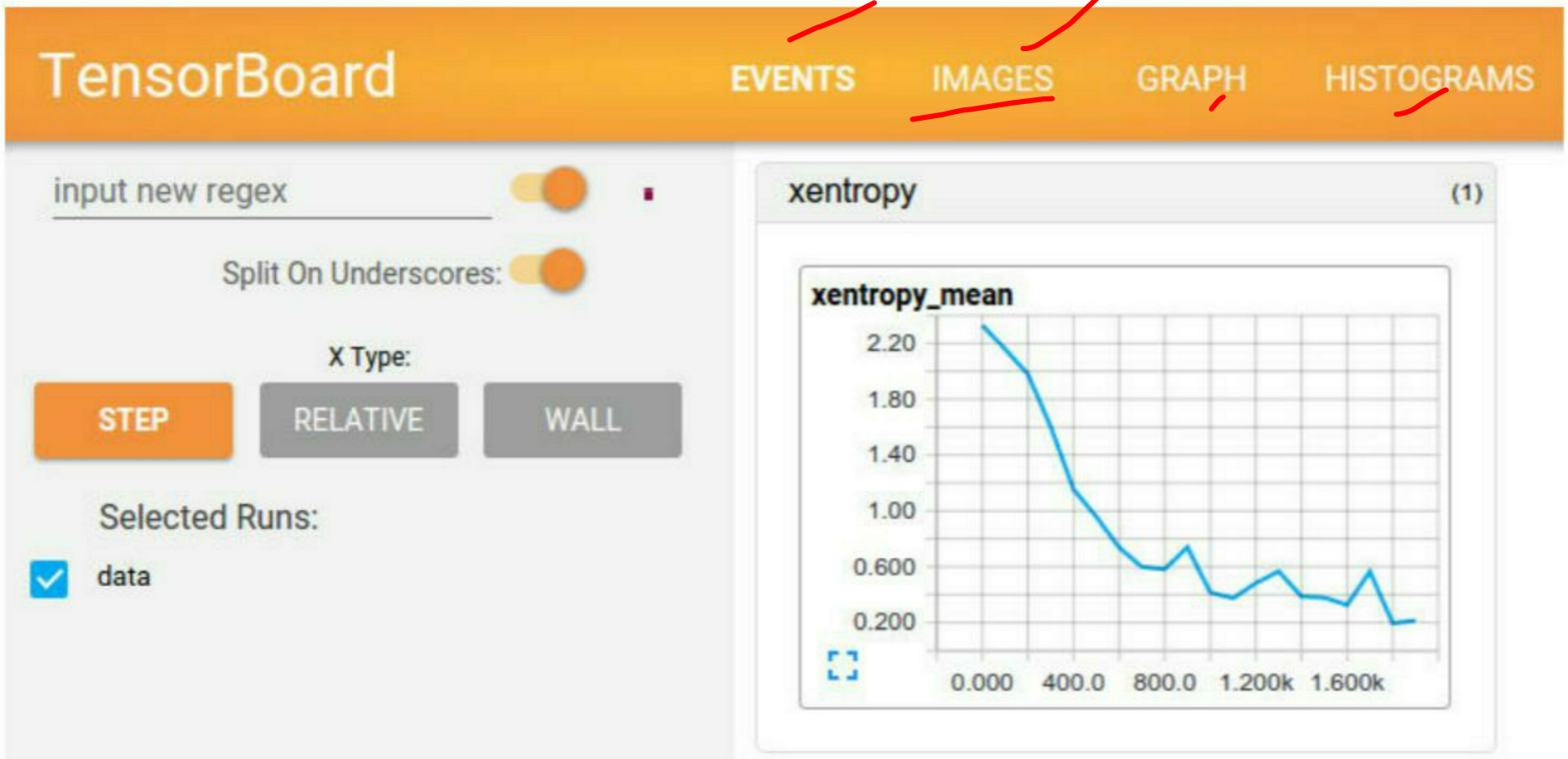
Graphs- The graph gathers and describes all the series computations done during the training.





Tensor board

Tensor board Tensor Board is a suite of web applications for inspecting and understanding your TensorFlow runs and graphs. It can output the graph of the calculation and draw a chart of summary values which you log in the code.



TensorFlow Overview

Highlights of the Important changes from TF 1.4

Eager Execution	TF 2.0 Default execution mode is based on Eager Execution which eliminates manual compilation
Keras Fully Integrated	Keras API is fully integrated with TF 2.0 , to leverage quick model building and implementations. Includes support for building Data Pipelines , Estimators and Eager Execution
Consolidation of Modules	Consolidation of APIs , removal redundant ones , made more productive , easy to use and Faster implementation
Data Pipelines	Data API helps us to build complex input pipelines from various sources. Helps us to handle large volumes of data , formats etc.

Core Modules and API of TensorFlow 2.0

Core TensorFlow Modules and APIs for End to End Deep learning Model building Support

Keras	Data	Accelerators	TF Hub	TF Functions
• Easy to use TensorFlow API for quick prototyping of a Deep learning Model. • With Latest integration and modification can be used for Production as well	• Data API helps users to build Data Pipeline for training a Deep learning model	• Build a distribution strategy for Model training using TensorFlow Accelerators • Train on multiple GPUs	• Leverage transfer learning using TF Hub , a library for reusable Machine Learning modules.	• Construct TensorFlow Graph using Functions Module

Data Pipelines using TensorFlow 2.0

- Build Data pipeline using tf.data
- Design Complex input data pipeline and incorporate transformation Functions as part of it

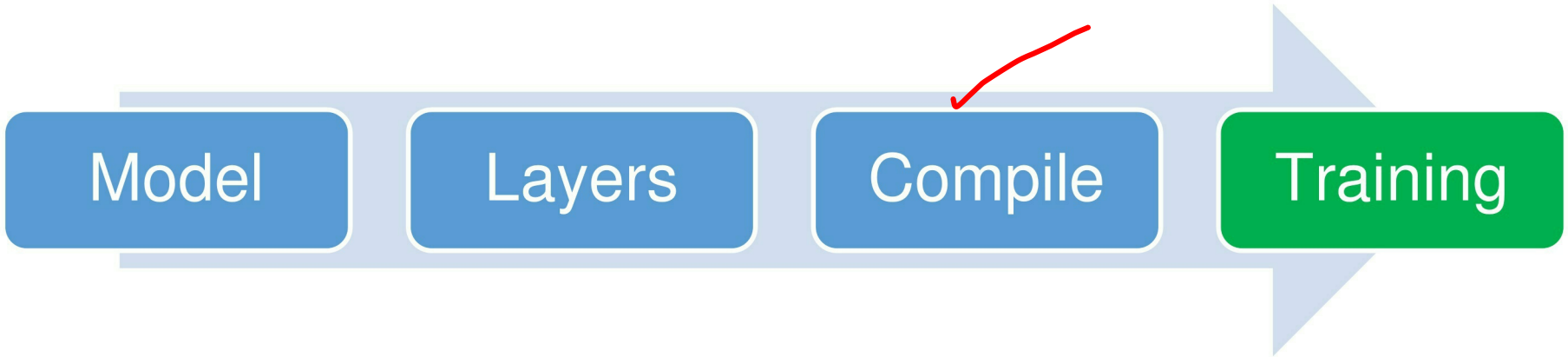
- Data Source

A data source constructs a Dataset from data stored in memory or in one or more files.

- API introduces a tf.data.Dataset abstraction that represents a sequence of elements, in which each element consists of one or more components.

- Transformed Data is used to train the model by sending the data in batches

TensorFlow Keras



- Construct Overall Model structure
- A Framework For the model

- Configure layers based on the Deep learning model type
- Convolutions Layers , Recurrent Layers , Dense Layers etc.

- Assign a Optimizer and Loss functions
- Compile the model

- Train the model based on the number of Epoch Provided

In [2]:
```
import numpy  as np
import math
```

In [3]:
```
def gradient_descent(x,y):
    w_curr = b_curr = 0
    itr = 100
    n =len(x)
    learning_rate = 0.08

    for i in range(itr):
        y_pred = w_curr * x + b_curr
        cost = (1/n) * sum([ val**2 for val in (y-y_pred)])
        adj_w = -(2/n)*sum(x*(y-y_pred))
        adj_b = -(2/n)*sum(y-y_pred)
        w_curr = w_curr - learning_rate* adj_w
        b_curr = b_curr - learning_rate* adj_b
        print("Weights {}, Bias {}, cost {}".format(w_curr, b_curr,cost, i))
```

$W_n = W_0 \cdot \eta\left(\frac{dMSE}{dx}\right)$

In [4]:
```
x= np.array([1,2,3,4,5])
y= np.array([5,6,7,8,9])
```

In [5]:
```
gradient_descent(x,y)
```

Weights 3.68, Bias 1.12, cost 51.0

In [ ]:
```
import tensorflow.compat.v1 as tf
tf.disable_v2_behavior()
x = tf.placeholder("float", None)
y = x * 2
print(type(y))
```

In [31]:
```
val = tf.sparse.SparseTensor(indices=[[0, 0], [1, 2]], values=[1, 2], dense_shape=[3, 4])
```

In [32]:
```
print(val.indices)
```

```
tf.Tensor(
[[0 0]
 [1 2]], shape=(2, 2), dtype=int64)
```

[0,0] = 1

[1,2] = 2

3 x 4

In [ ]:

In [ ]:

Rank of Variable

In [18]:
```
rank1_tensor = tf.Variable([["Test"], ['ok']], [['Innomatics'],['Research']]], tf.string)
rank2_tensor = tf.ragged.constant(np.array([["test", "ok", "tind"], ["test", "yes", 'tind'], [2.3,6.0, 'tind']]), tf.string)
print(rank1_tensor)
print(rank2_tensor)
```

```
<tf.Variable 'Variable:0' shape=(2, 2, 1) dtype=string, numpy=
array([[[b'Test'],
        [b'ok']],

       [[b'Innomatics'],
        [b'Research']]], dtype=object)>
<tf.RaggedTensor [[b'test', b'ok', b'tind'], [b'test', b'yes', b'tind'], [b'2.3', b'6.0', b'tind']]>
```

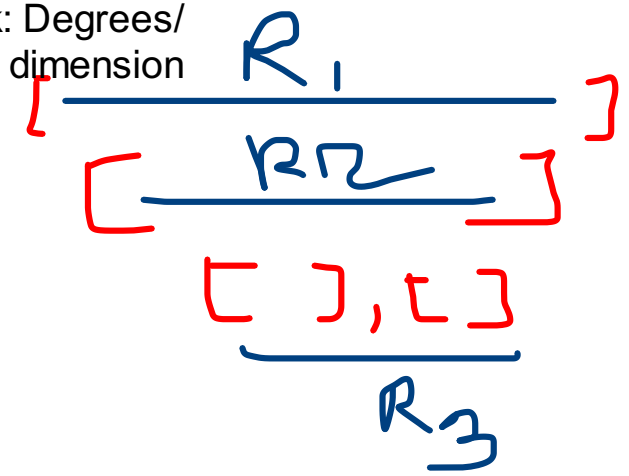
In [20]:
```
tf.rank(rank2_tensor)
```

Out[20]:
```
<tf.Tensor: shape=(), dtype=int32, numpy=2>
```

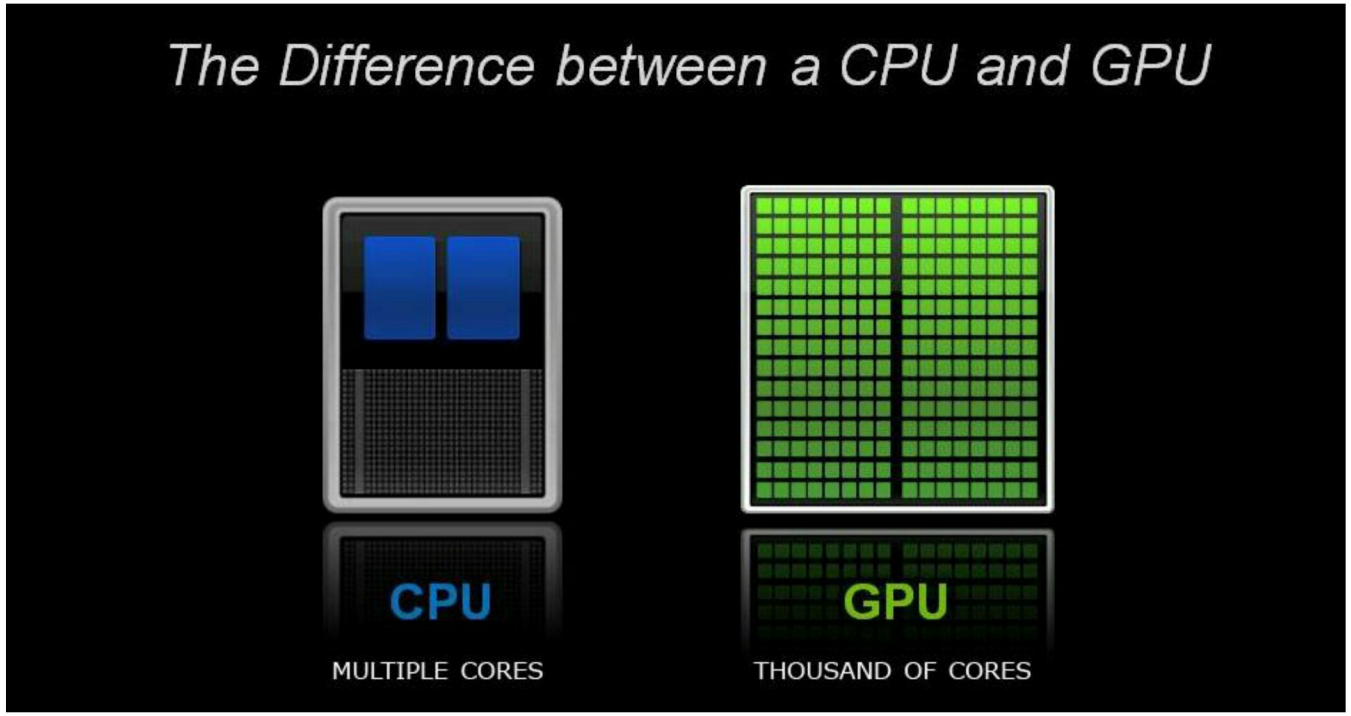
Types of data Structure in Tensorflow

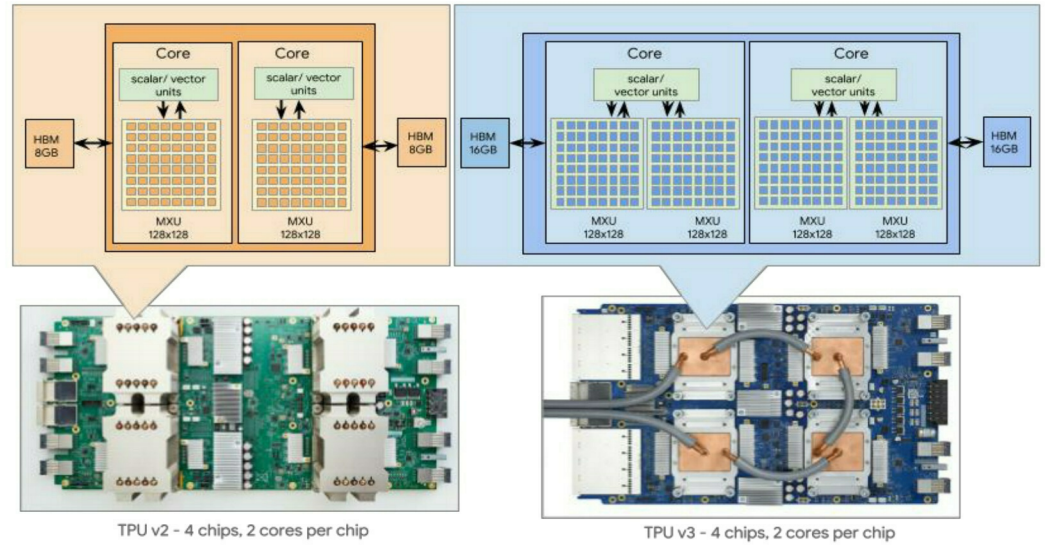
1. Variable → mutable
2. Constant → immutable
3. Placeholder → a space with one memory unit
4. Sparse Tensor → Combining different set of tensor

Rank: Degrees/ no of dimension



CPU vs GPU vs TPU

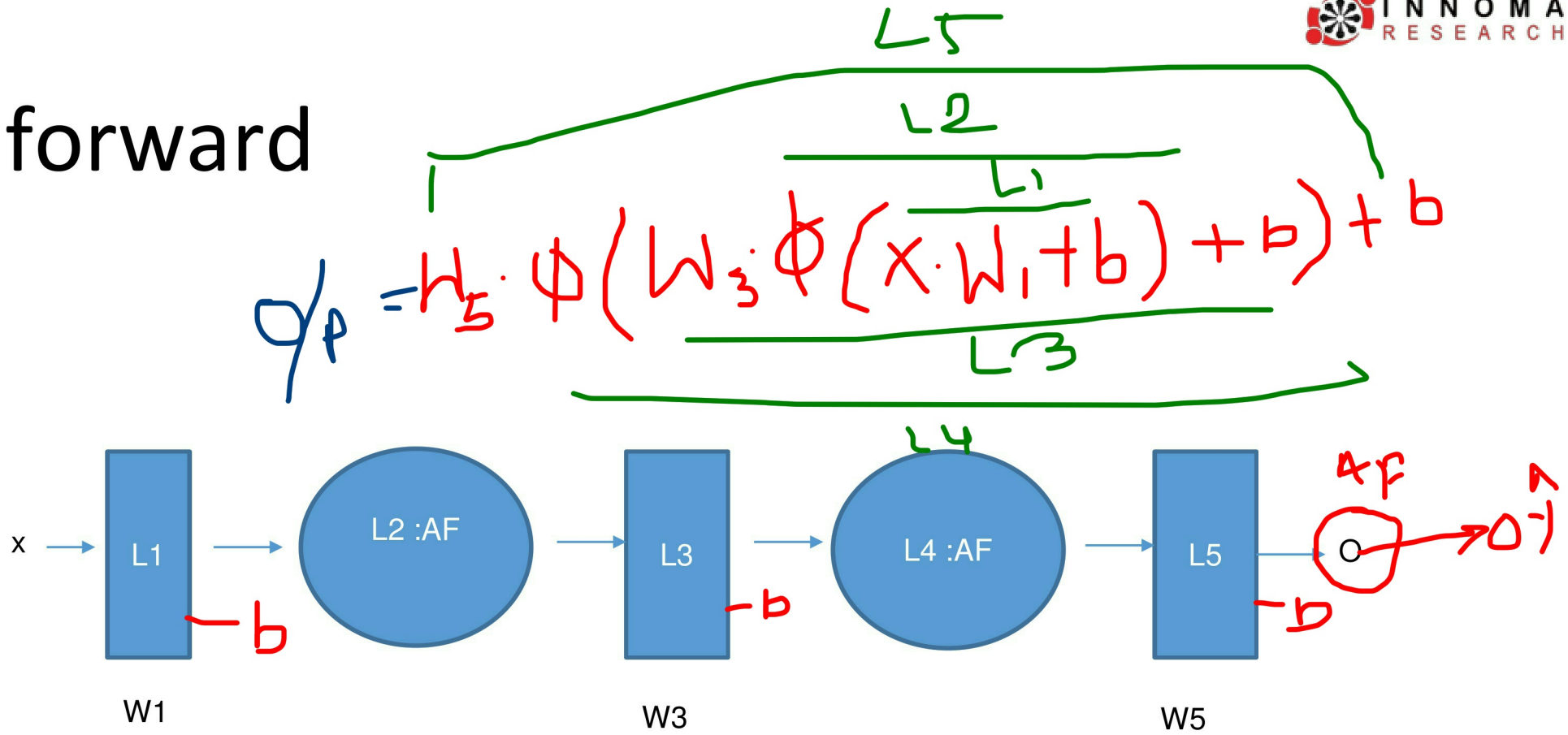




Agenda:

- Feed forward
- Back propagation
- Fully connected layer – forward pass
- Fully connected layer – backward pass
- Activation functions
- Activation functions in practice
- Softmax
- Cross entropy
- Hands on MNIST- with all building blocks explained in detail

Feed forward

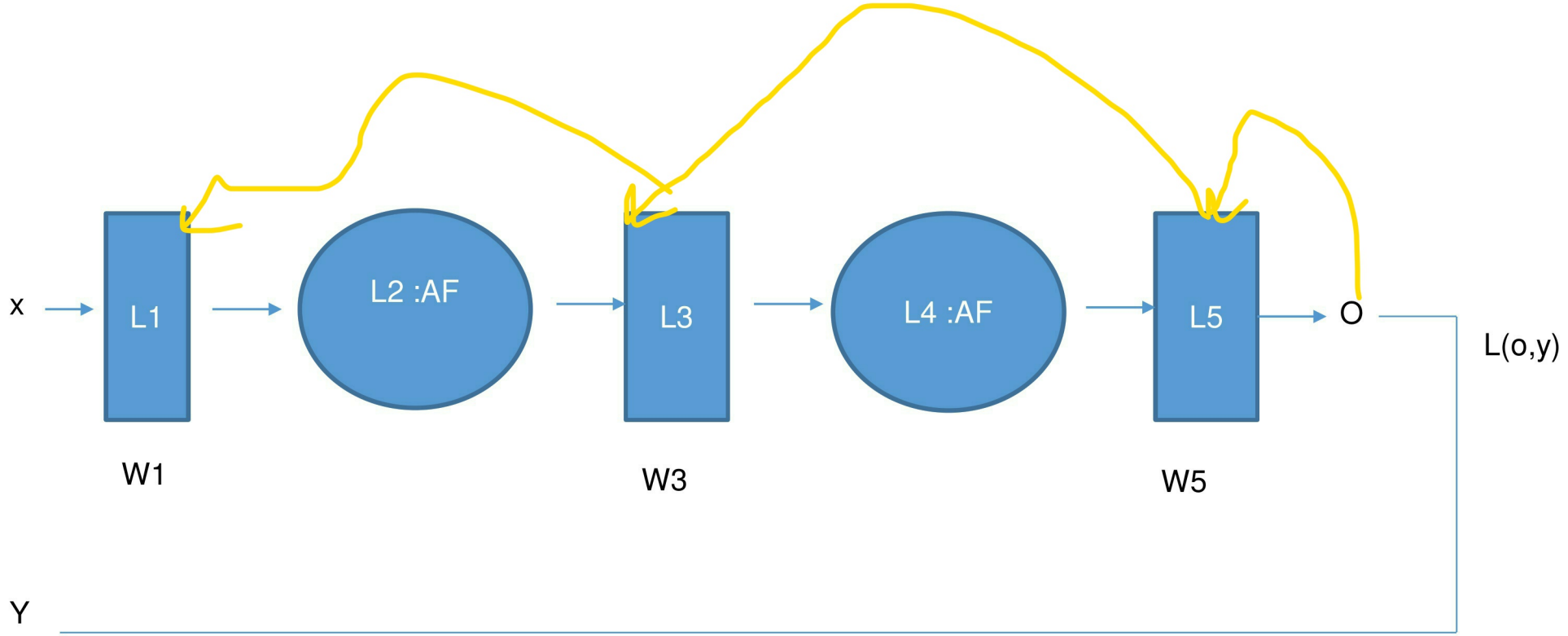


X- input, L1 – Layer 1, Layer 2- Activation function outputs of all layers are a1,a2,a3 and a4

Feed forward

- $a_1 = F(x, W1), x \in \mathbb{R}$
- $a2 = G(a_1)$
- $a_3 = H(a_2, W_3),$
- $a_4 = J(a_3)$
- $o = K(a_4, W_5) = \mathrm{K}\left(\mathrm{J}\left(\mathrm{H}\left(\mathrm{G}\left(\mathrm{F}(\mathrm{x}, W_1)\right), W_3\right)\right), W_5\right) \in \mathbb{R}$

Back propagation



FCN- Forward

- Neurons have connections to all activations of the previous layer
- Number of connections addup very quickly due to all combinations
- Forward pass – one matrix multiplication followed by bias offset and activation function

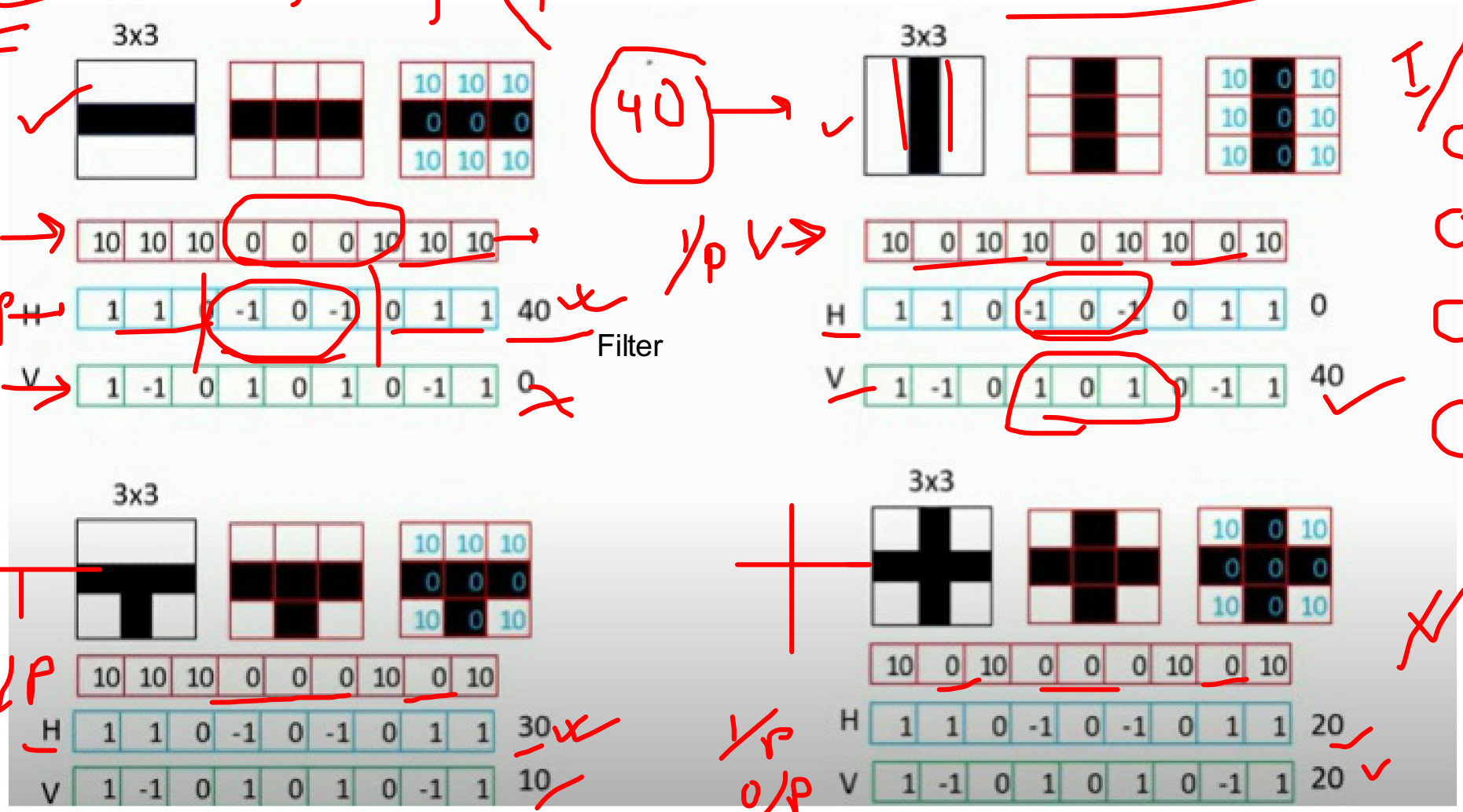
FCL – Backward pass

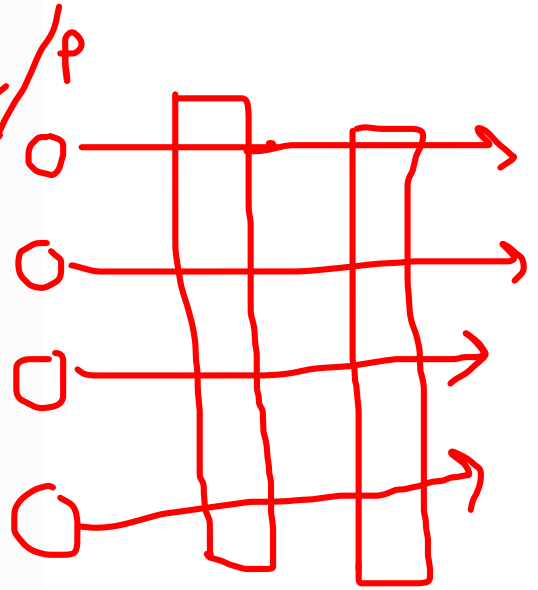
- We have gradients coming from next layer called GradInput
- Size is always equal to the size of output
- We will calculate the gradients which we need to pass to previous layers for continuing the chain rule
- We will also calculate the gradients that we will be using for updating or weights

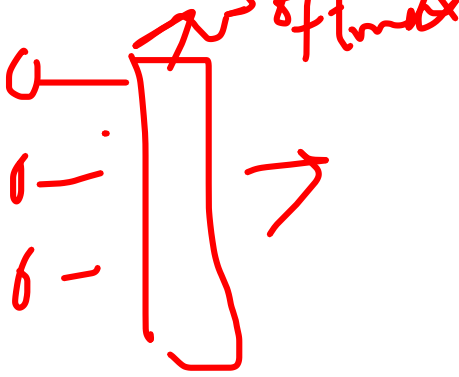
FCL

$f(x) = (I/P \cdot w + b)$

feature







Activation functions

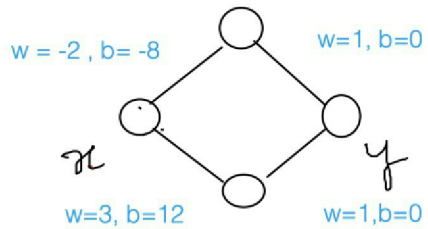
- An activation function takes a single input value and applies a function to it to add non linearity
- Why non-linear? Converts linear into non-linear for having a non-linear decision boundary
- Afs can decide which neuron is switch ON ,it acts as gate

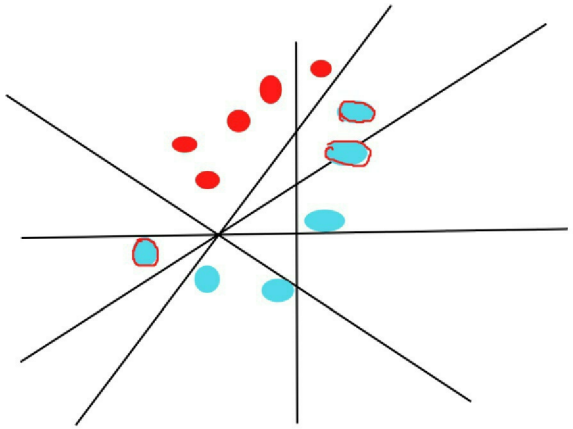
AF in practise

- Use Relu , monitor the dead neurons
- Never use sigmoid
- Try Leaky Relu
- Tanh
  - Not sure if it performs better!

How relu is introducing non linearity?

How relu introduces non-linearity





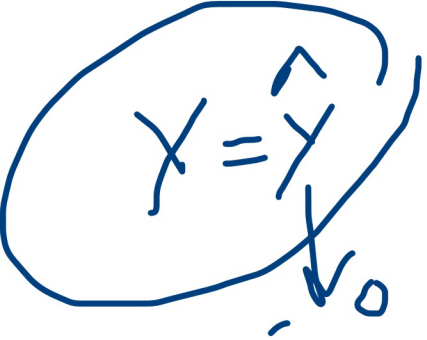
equation for (-2*x))-8,3x+12,x+4

Vanishing gradient

- If we look at the formula for updating the weights of W3 or W1 it's a huge multiplications of Gradient with Jacobians
  - If all values are less than 1 we have vanishing gradient problem
  - If all values are greater than one we have exploding gradient problem

Vanishing gradient

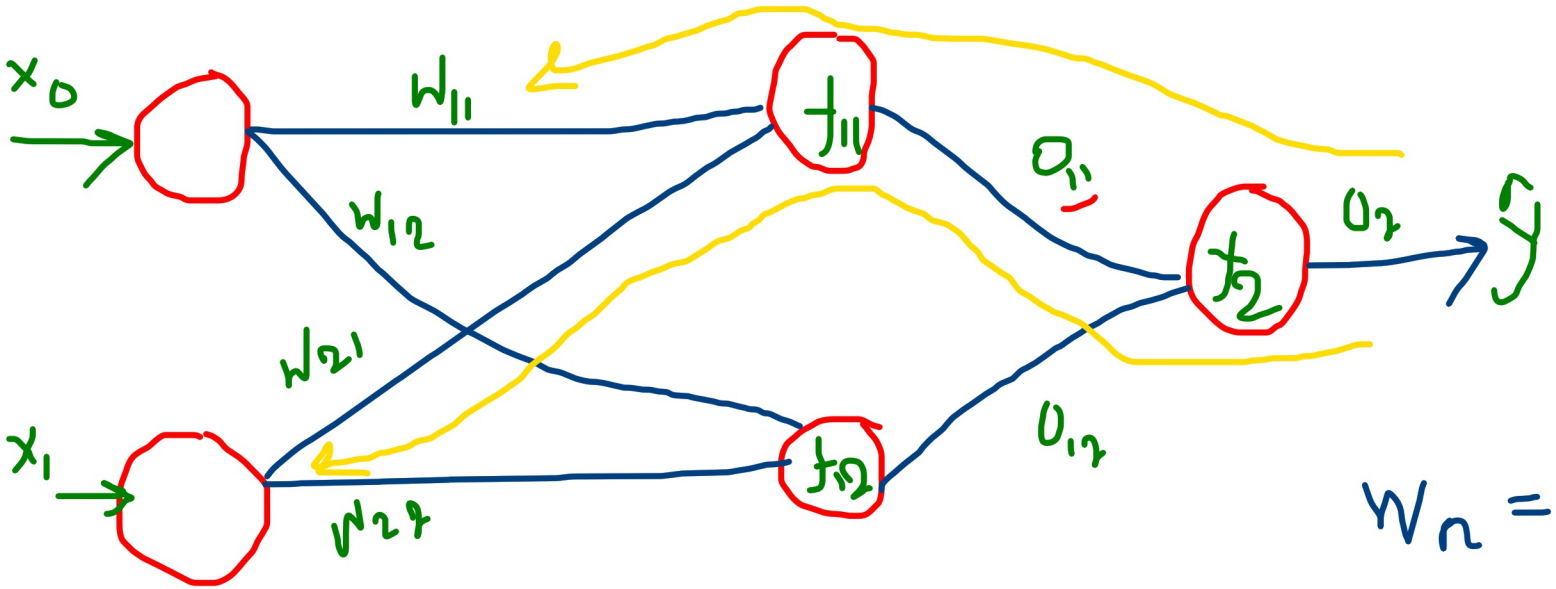
- If we look at the formula for updating the weights of W3 or W1 it's a huge multiplications of Gradient with Jacobians
  - If all values are less than 1 we have vanishing gradient problem
  - If all values are greater than one we have exploding gradient problem



Vanishing Grad - Back propagation

$\sigma(x) = 0 \rightarrow 1$

$\frac{d\sigma(x)}{dx} = 0 \rightarrow 0.25$



$MSE = \frac{1}{n}(\hat{y} - y)^2$

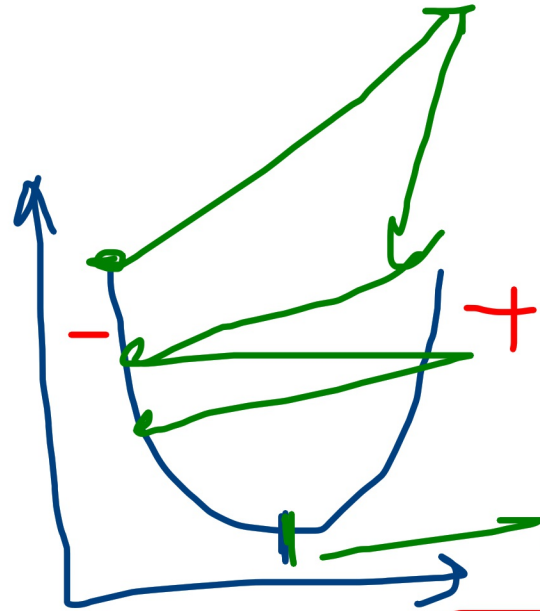
$W_n = W_0 + \eta \frac{dMSE}{dx}$

Small change in W

$\frac{\partial O_2}{\partial W_{11}} = \frac{\partial O_2}{\partial O_{11}} \cdot \frac{\partial O_{11}}{\partial W_{11}}$

$O_{11} = \phi(W \cdot O_{11} + b)$

Exploding Gradient



$W_n = W_0 - \eta \frac{dL}{dx}$

$W_{n_1} = W_n - \eta \frac{dL}{dx}$

$W_0 - \eta \frac{dL}{dx}$

$256 - \eta \times 7000$

W → too low

↓ epoch

W → too high

increasing

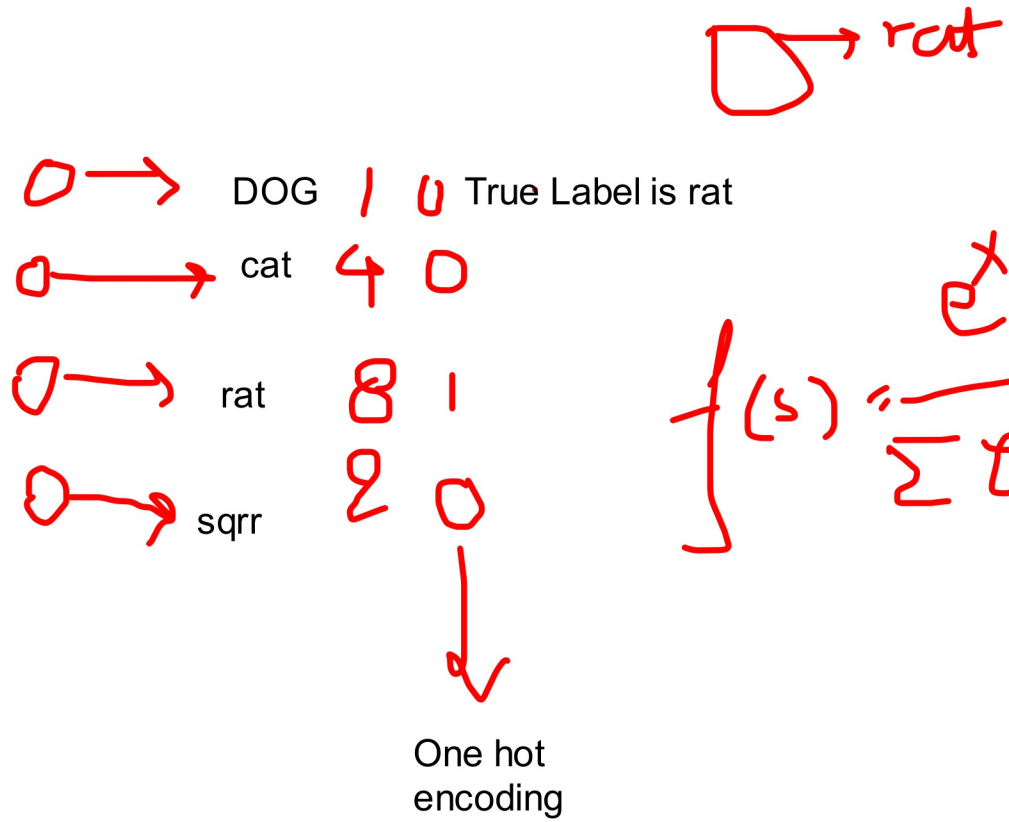
RNN

↓

NLP / NLU / NLG

softmax

- Softmax is multinomial logistic classifier , it can handle multiple classes
- Softmax is typically the last layer of NN
- Softmax itself is an activation function , hence it need not be combined with any activation function



$$f(s)_i = \frac{e^{x_i}}{\sum e^{x_k}}$$

$$D = \frac{e^1}{e^1 + e^4 + e^8 + e^2} = 0.6$$

$$C = \frac{e^4}{e^1 + e^4 + e^8 + e^2} = 0.2$$

✓ = 0.9 →

sq = 0.8

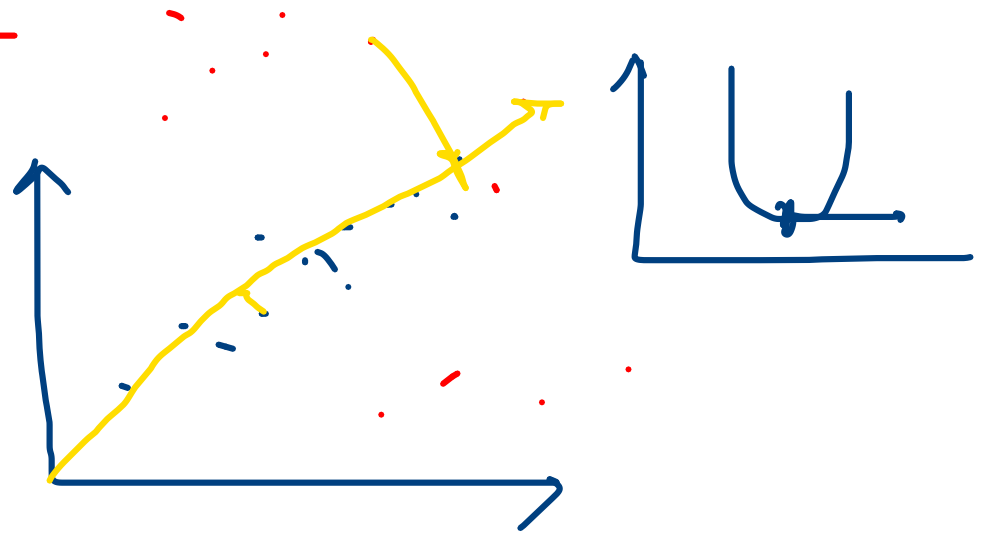
Loss fn

1. Loss = $(\hat{y} - y)^2$ → Cost Function $J = \frac{\sum (\hat{y} - y)^2}{n}$

$y = w \cdot I + b$

$W_0 = 1$ $\Rightarrow \uparrow$

$W_2 = 6.25$



Disadv: Penalizing the loss for outliers

Adv: no local minima

2. Absolute Error loss

$L = |y - \hat{y}| \longrightarrow \frac{\sum |y - \hat{y}|}{n}$

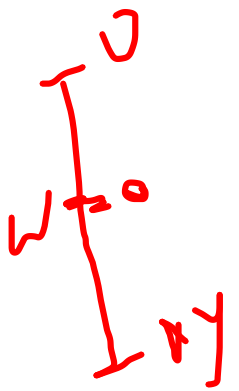
- Computation is too high

Huber Loss Function $= MSE + MAE$

$$Loss = \begin{cases} \frac{1}{2}(y-\hat{y})^2 & \text{Quad} \\ \delta|y-\hat{y}| - \frac{1}{2}\delta^2 & \text{linear} \end{cases}$$

if outliers are present then we use Quadratic Fn else we use linear fn

δ → Hyperparameter



Cross-entropy loss

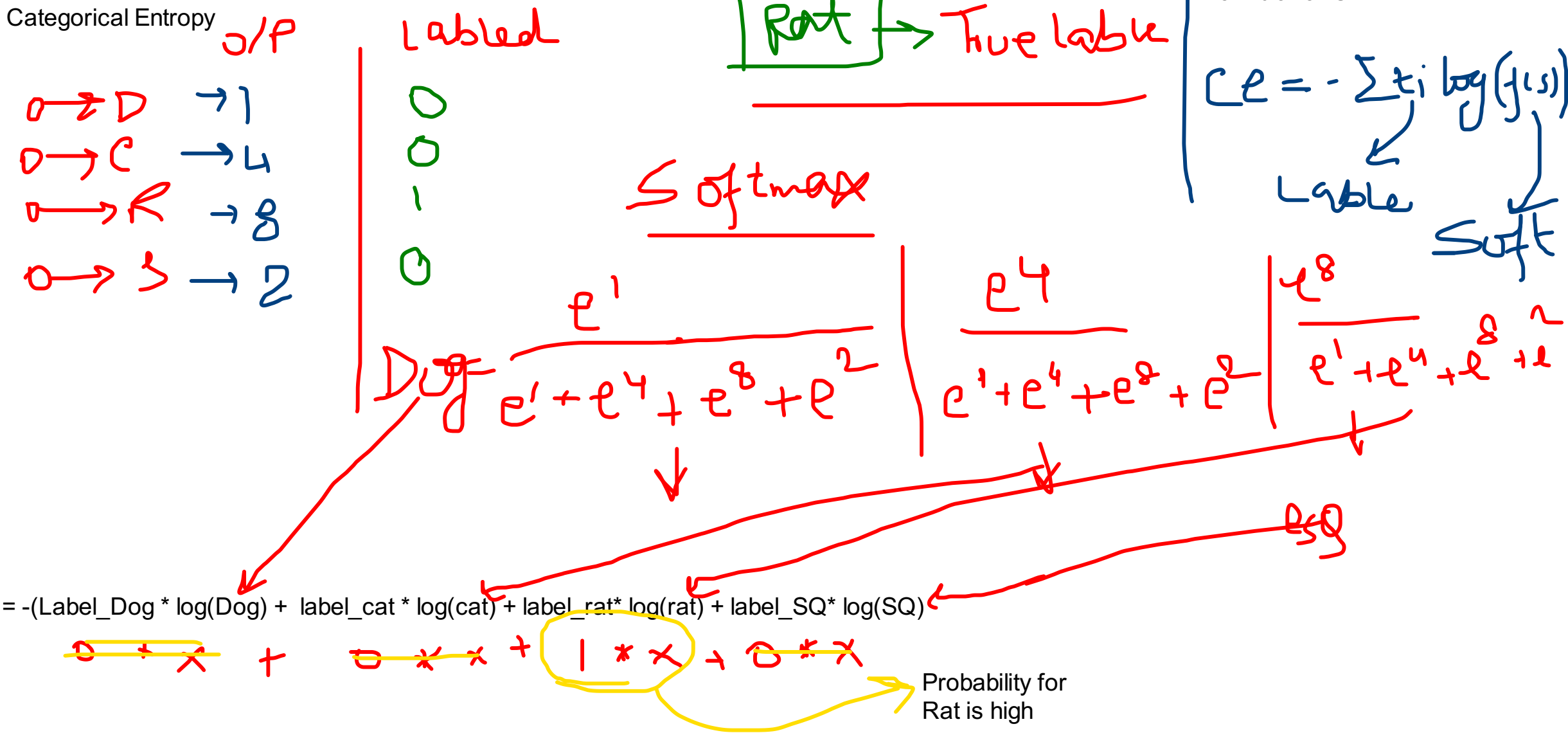
- CE loss also called log loss quantifies the deviation between predicted output and ground truth
- Perfect prediction would have zero loss
- With Gradient Descent , we will try to reduce the error

1. Binary Cross Entropy

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

$$f(x) = \begin{cases} -\log(1-\hat{y}) \\ y = 0 \end{cases} \qquad -\log(\hat{y}) \longrightarrow \therefore y = 1$$

Categorical Entropy



Neural Networks construction

NN Constructed

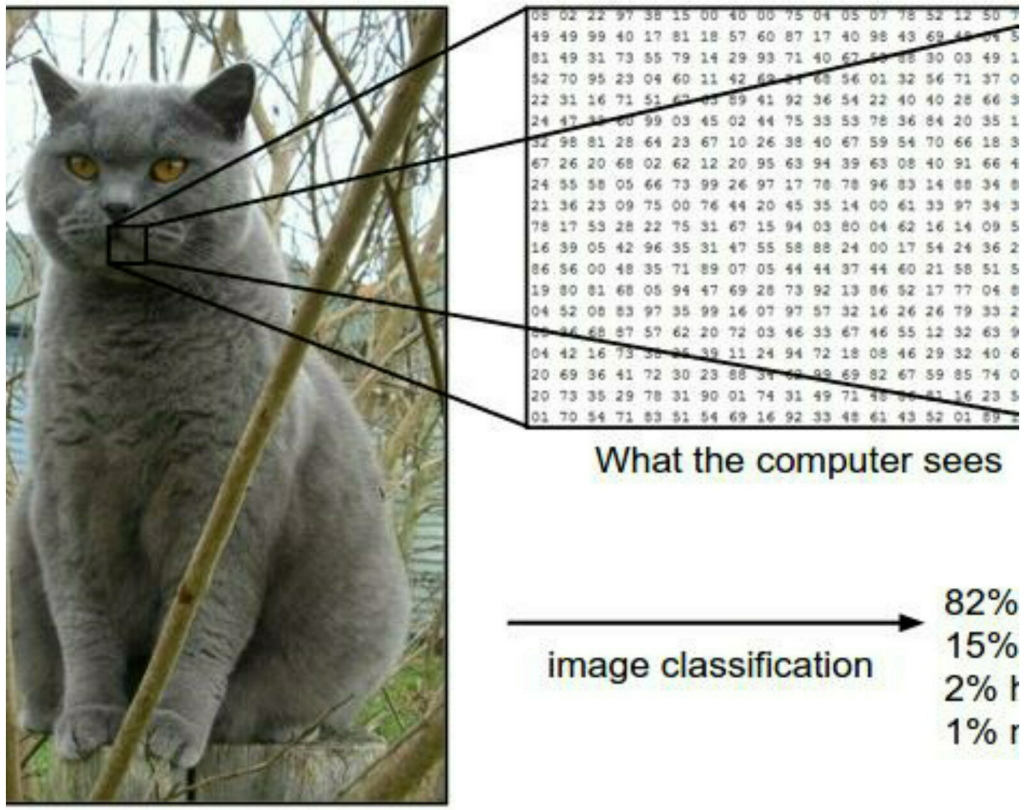
- Data preprocessing
- Data Augementation
- Weight Initialization
- Regularization
  - Implement all these in already build network

Data preprocessing

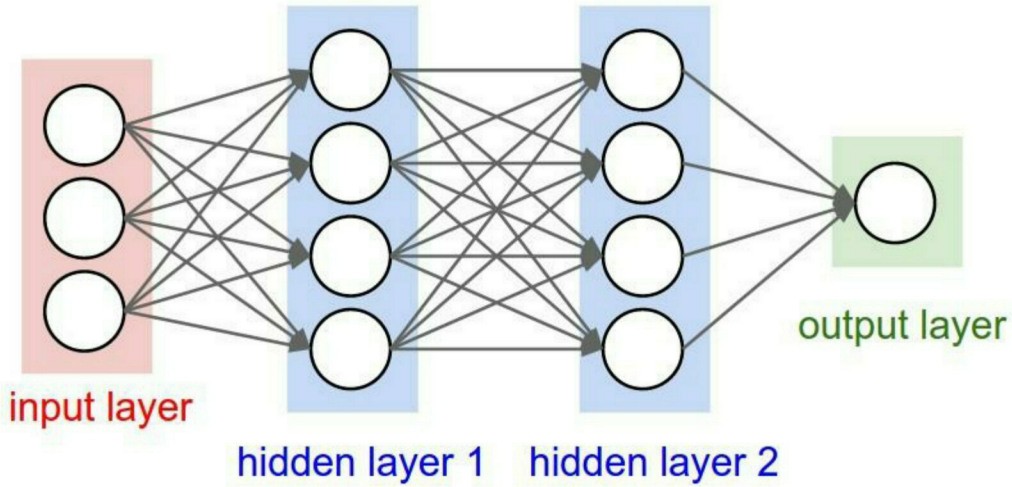
- Data Normalization
  - How to do it?

Data Augmentation

- Rotate
- Grey scale
- Adding random noise
- Horizontal flip
- Color jitter
- Ideal way of doing it , doing all these transformations, since for each epoch do some random transformation for the image



- If we slightly move the image, the pixels are changed and it altogether sees a different image



Weight initialization

- Lets say we have 3 input, 4 hidden and 1 output neurons.
- 3 * 12 etc.,
- And 4 * 4
- What if I initialize with zero?
  - Not a good idea, because output will always move in same direction

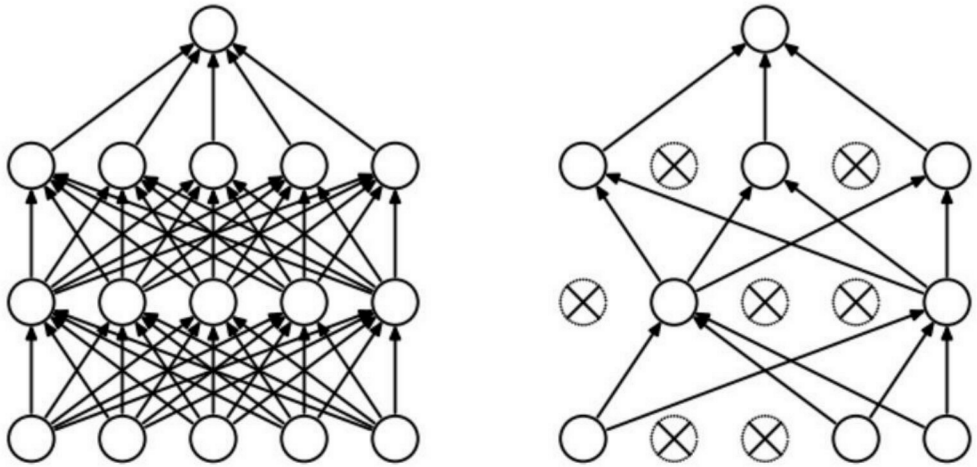
WI

- Simple random numbers (Normal distribution -0.01 to + 0.01)
  - W = np.random.randn(in,out)*0.01
  - B = np.random.randn(out) * 0.01
- Xavier Initialization:
  - W = np.random.randn(in,out)/math.sqrt(in)
  - This is implemented in most of the latest ones
- Idea: Take the random numbers and scale it based on the input neurons

Regularization (batch normalization)

- In weight init, we want our activations should be unit gaussians, hence we initialized the weights similarly
- Usually inserted after all fully connected layers, before the non-linearity
- Affine transformation
- BN:
  - Improves the gradient flow
  - Allows higher learning rates
  - Reduces strong dependency on initialization

Drop out



(a) Standard Neural Net

(b) After applying dropout.

- Dropping out is given as per probability, if p=0.5, 2 are dropped in one layer and 3 in next layer (total 5 in each layer)
- We will see how to do that in coding

Dropout

- It forces the network to have redundant representation
  - Has a ear X
  - Has a tail
  - Is fully X
  - Has claws
  - Mischievous looks X
- Predicting the same as CAT

- Dropout is training a large ensemble of models
- Each binary mask is one model ,gets trained on only one batch
- At test time, all neurons are ON
- We must scale the activations for each neuron so that output at testtime == expected output at traintime
  - What does this mean?

- Implement the same on MNIST Data

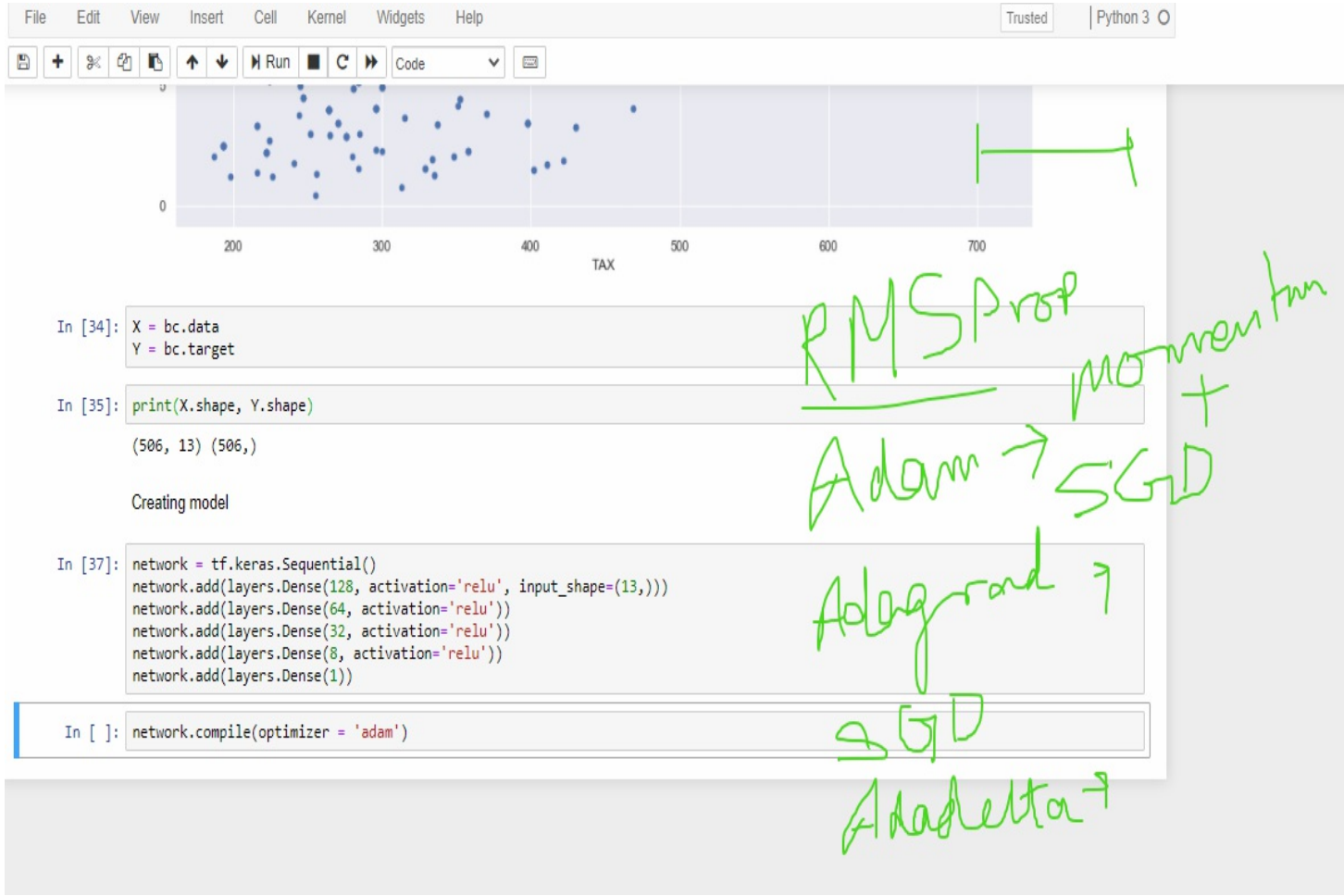
Types of Optimizer

1. RMSprop : The RMSprop optimizer is similar to the gradient descent algorithm with momentum. The RMSprop optimizer restricts the oscillations in the vertical direction. Therefore, we can increase our learning rate and our algorithm could take larger steps in the horizontal direction converging faster.

2.Adam : Adaptive Moment Estimation combines the power of RMSProp (root-mean-square prop) and momentum-based GD. In Adam optimizers, the power of momentum GD to hold the history of updates and the adaptive learning rate provided by RMSProp makes Adam optimizer a powerful method.

3.Adagrad: When dealing with small scale dataset, the hyperparameters are used/needs to be tuned. Till now we are only focusing on how the model parameters are affecting our training, but we haven't talked about the hyper-parameters that are assigned constant value throughout the training. One such important hyper-parameter is learning rate and varying this can change the pace of training.

4.Adadelta: **Adadelta** is a more robust extension of Adagrad that adapts learning rates based on a moving window of gradient updates, instead of accumulating all past gradients. ... Compared to Adagrad, it performs better in large scaled dataset as Adagrad decays the learning rate which slows down the entire training. But it is also applicable for Adagrad datasets as well



INNOMATICS RESEARCH LABS

THANK YOU