DEEP LEARNING

BY PRITESH JHA

INTRO TO DEEP LEARNING

What is Deep Learning?

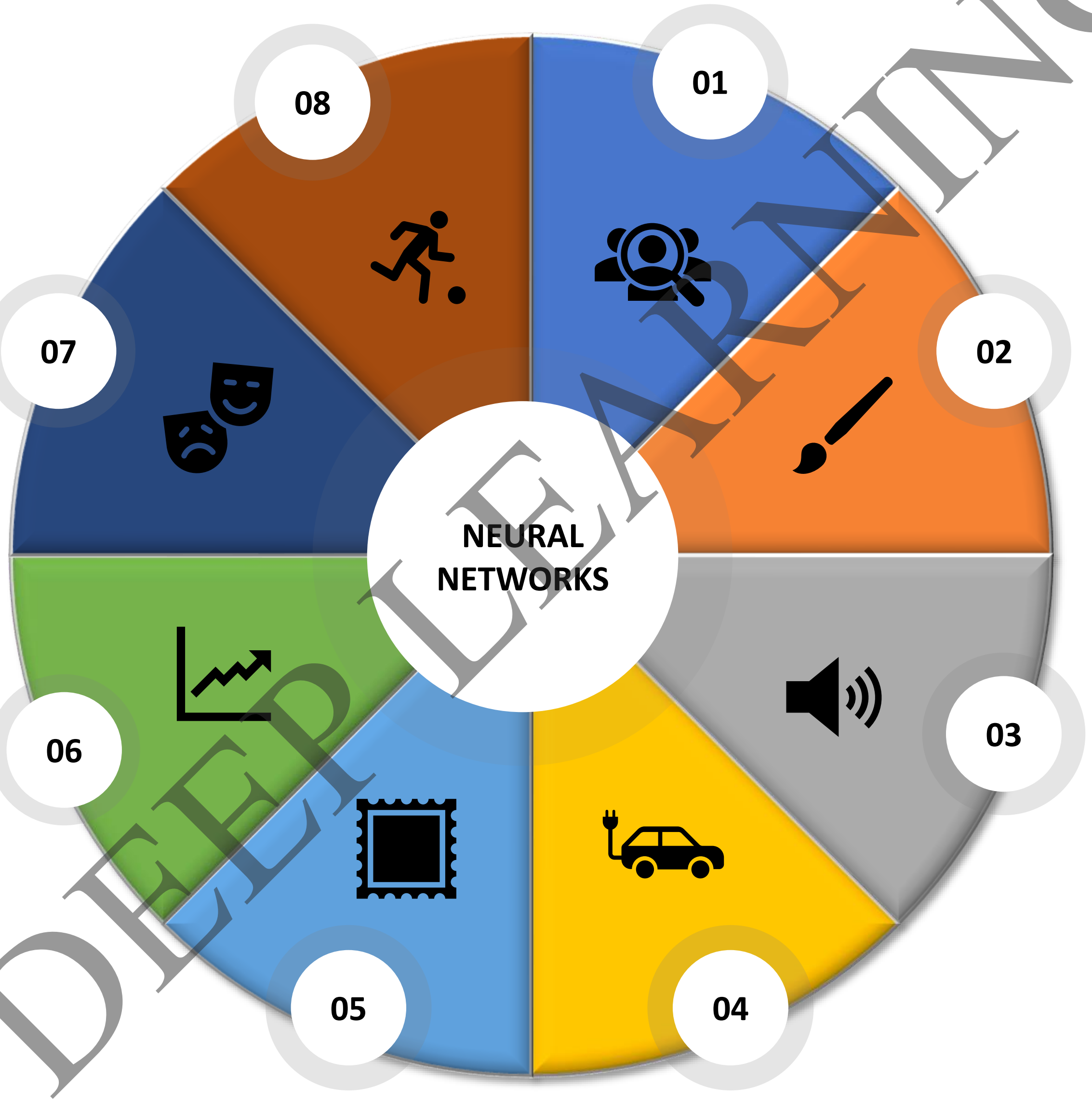
Deep Learning is a subset of artificial intelligence (AI) that mimics the workings of the human brain in processing data and creating patterns for use in decision making.
It is built around neural networks, which are algorithms modeled loosely after the human brain. These neural networks consist of layers of nodes, or "neurons," each layer designed to perform specific tasks and capable of learning from vast amounts of data.

How is Deep Learning different from Machine Learning?

Machine Learning	Deep Learning
• ML models are based on different techniques (Logistic Regression, SVM, Tree based models, etc)	• The core of DL models are Neural networks (Only the architectures may vary)
• Typically requires less data to train the model	• Requires large amount of data to be effective
• Can be less computationally intensive compared to deep learning	• Requires significant computational power (often GPUs) due to the complexity of the neural networks
• Models (especially simpler ones) tend to be more interpretable	• Models are generally considered "black boxes" because of their complexity

INTRO TO DEEP LEARNING

Applications of Deep Learning



- 🟦 Computer Vision
- 🟧 NLP
- ⬜ Speech Analytics
- 🟨 Self Driving Cars
- 🟦 Robotics
- 🟩 Regression & Classification
- 🟦 Segmention & Product Recommendation
- 🟫 Sports Analytics

- PRITESH JHA

RECAP

What do we need to know before we start learning Neural Networks?

- Python - Numpy, Pandas
- Maths - Derivatives (Good to know)
- Linear Regression (Theory)
- Logistic Regression (Theory)

RECAP

Linear Regression

The equation for Simple Linear Regression is :

$$y = a*X + b$$

where
Y = dependant variable
X = independent variable
a = weight
b = bias

Sample data

X1	X2	X3	y
20	30	10	58
30	10	25	67
21	15	31	78

Assuming the data is in the above format, the equation becomes :

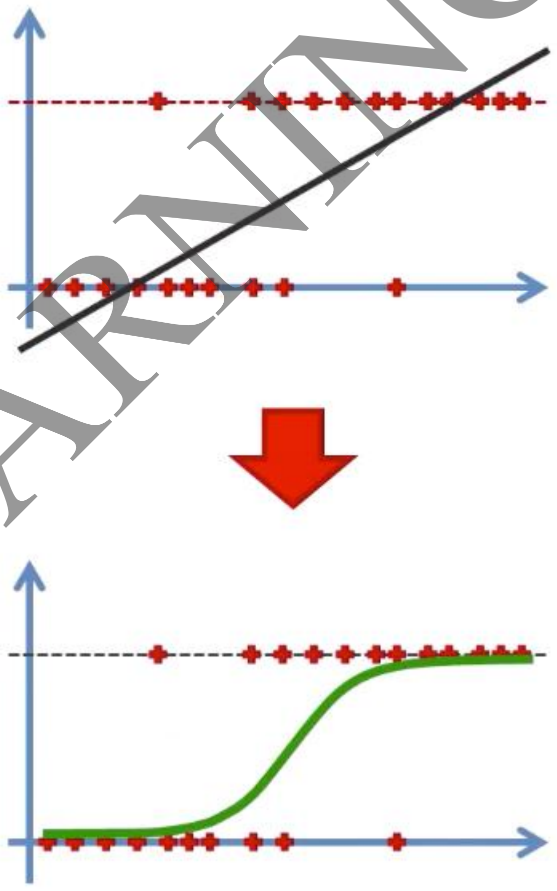
$$y = a1*X1 + a2*X2 + a3*X3 + b$$

RECAP

Logistic Regression

The equation for Linear Regression is :

$$f(z): \quad y = a*X + b$$



Applying Sigmoid Function to f(z) :

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

RECAP

Derivatives

Sigmoid Function: $\sigma(z) = \dfrac{1}{1+e^{-z}}$

Quotient rule: $\dfrac{d}{dx}\cdot\dfrac{f(x)}{g(x)} = \dfrac{f'(x)g(x) - f(x)g'(x)}{g(x)^2}$

Taking derivative of the Sigmoid Function using Quotient Rule

$$\sigma'(z) = \frac{0-(-e^{-z})}{(1+e^{-z})^2} = \frac{e^{-z}}{(1+e^{-z})^2} = \frac{1+e^{-z}-1}{(1+e^{-z})^2} = \frac{\cancel{1+e^{-z}}}{(1+\cancel{e^{-z})}^2} - \frac{1}{(1+e^{-z})^2}$$

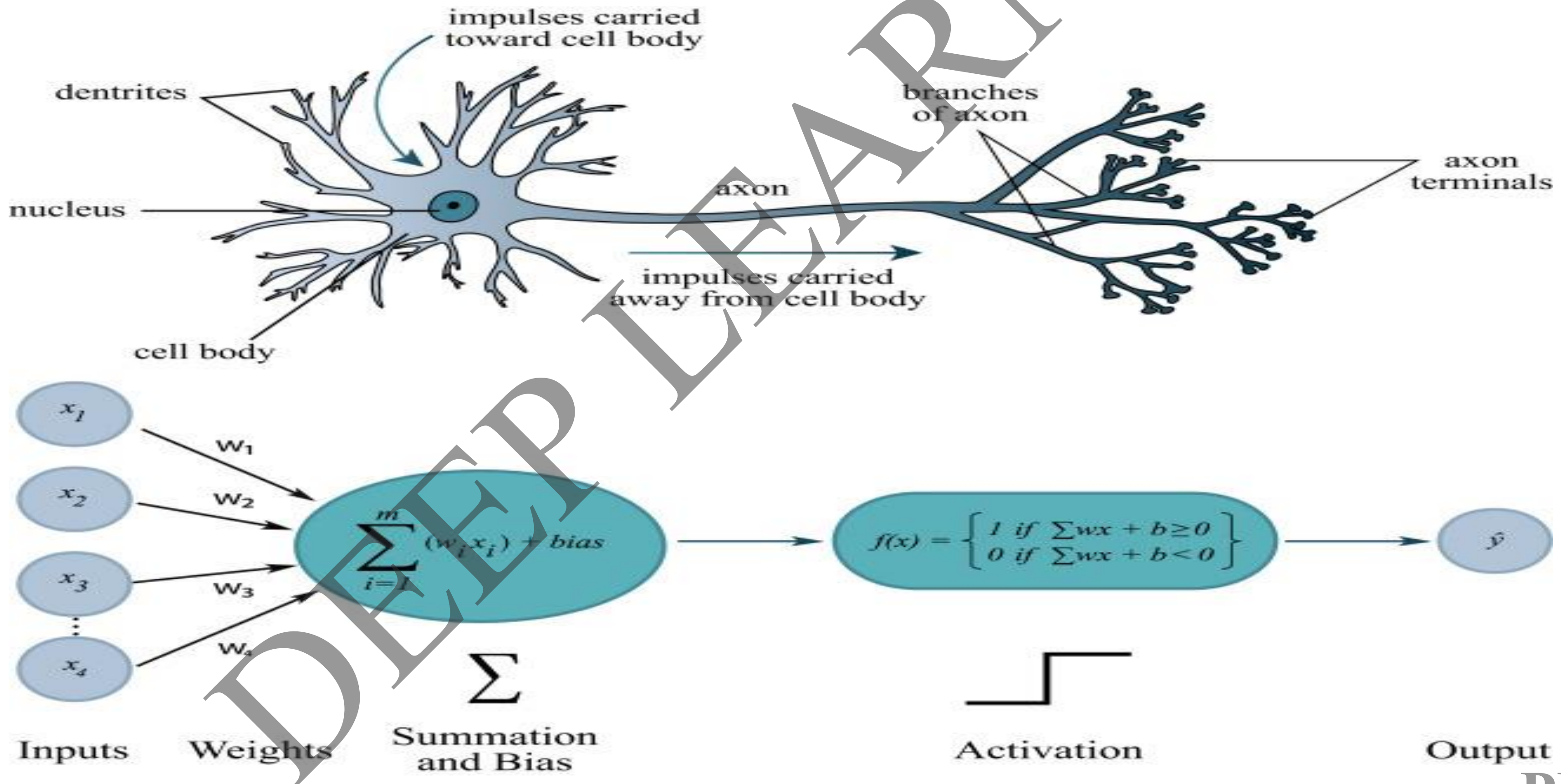
$$= \frac{1}{1+e^{-z}} - \frac{1}{(1+e^{-z})^2}$$

$$= \underbrace{\frac{1}{1+e^{-z}}}_{\sigma(z)}\Bigg(1 - \underbrace{\frac{1}{1+e^{-z}}}_{\sigma(z)}\Bigg) \longrightarrow \boxed{\sigma'(z) = \sigma(z)(1-\sigma(z))}$$

NEURAL NETWORKS

Biological Reference

Neurons process the information from different sources in the body, processes the data, and pass on the “refined” information to the next neuron / node to comprehend a much complex task and generate the required action.

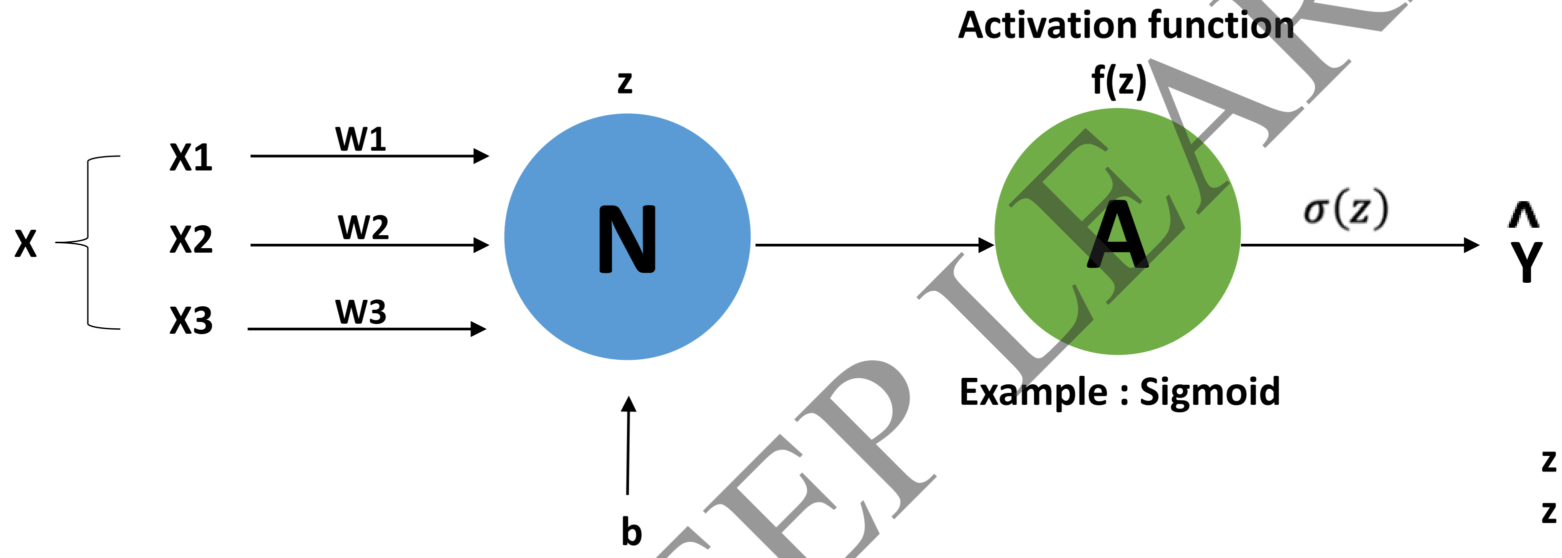

impulses carried toward cell body
dentrites
nucleus
cell body
axon
branches of axon
axon terminals
impulses carried away from cell body
x_1
x_2
x_3
x_4
W_1
W_2
W_3
W_4
$\sum_{i=1}^{m}(w_i x_i) + bias$
$f(x) = \begin{cases} 1 \text{ if } \sum wx + b \geq 0 \\ 0 \text{ if } \sum wx + b < 0 \end{cases}$
$\hat{y}$
Σ
Inputs
Weights
Summation and Bias
Activation
Output

NEURAL NETWORKS

Fundamentals of Neural Networks

The basic architecture of a “unit” neural network consist of 3 stages :

- Input Layer
- Hidden Layers and Activation Function
- Output Layer



z : y = w*X + b

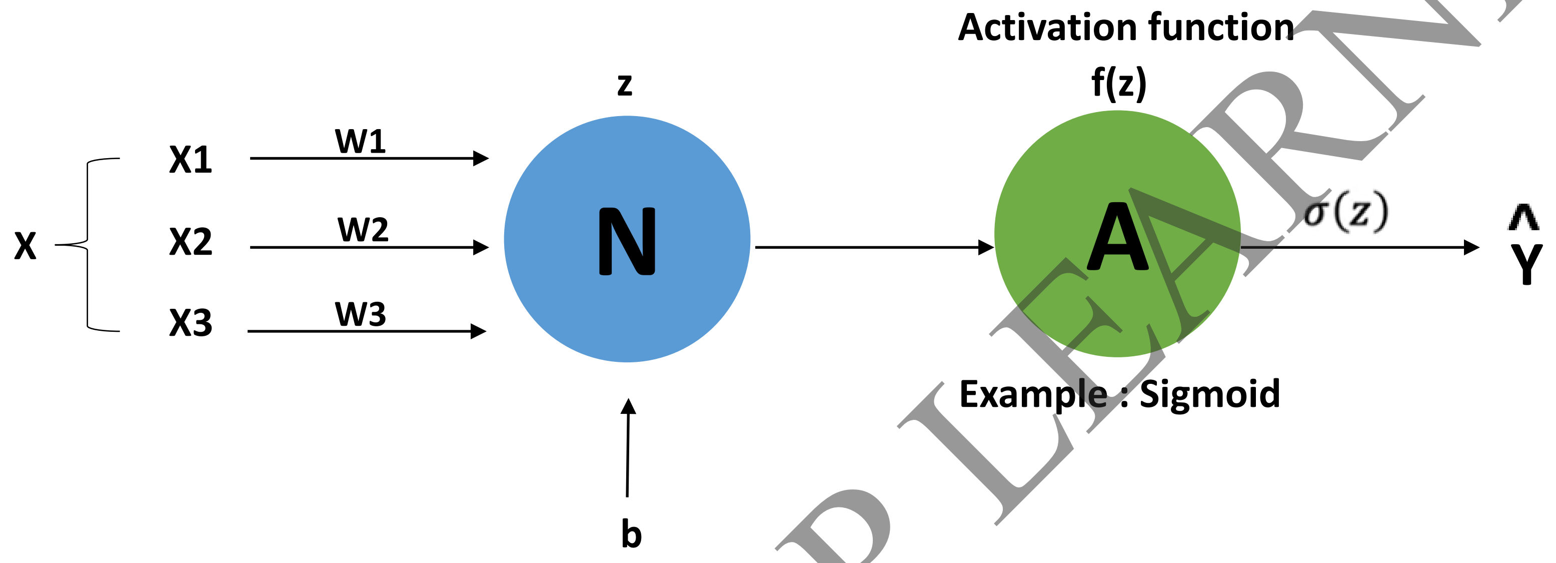
z : y = w1*X1 + w1*X1 + w1*X1 + b

Note : This unit neuron architecture will give similar results to Logistic Regression as the working principle is same.
As there is a single node only, the number of hidden layers = 1 and number of nodes = 1.

NEURAL NETWORKS – WORKING OF A NEURON

Working of a Neuron

Let us assign some input values to the node, and calculate the output to understand how the computation works at neuron level.



X1 = 0.9 W1 = 2
X2 = 0.2 W2 = 3
X3 = 0.3 W3 = -1
B = 0.5

z = w1*x1 + w2*x2 + w3*x3 + b

z = 0.9*2 + 0.2*3 + 0.3*(-1) + 0.5

z = 2.6

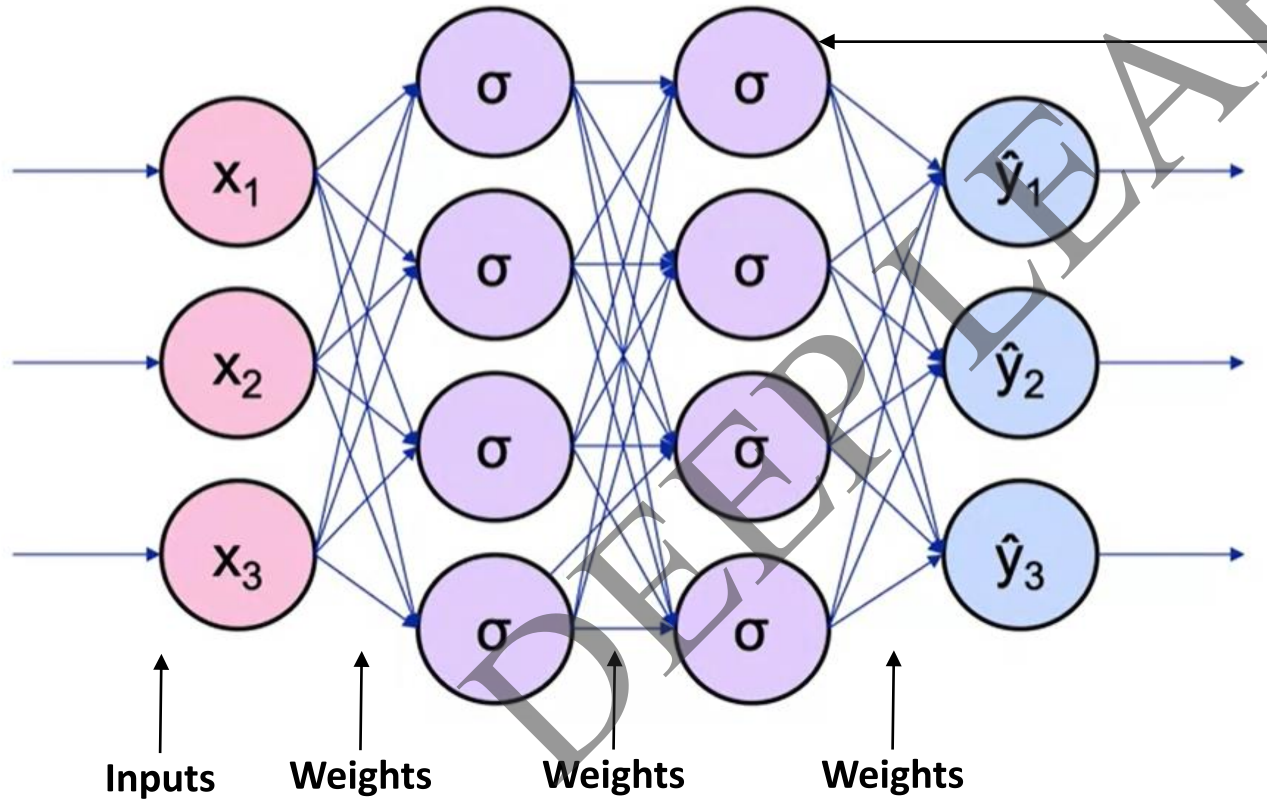
f(z) = f(2.6) = 1 / (1 + exp(-2.6))

f(z) = 0.93

NEURAL NETWORKS – FEED FORWARD NETWORK

Multi-layer Perceptron

Let us build an NN architecture of a multiclass classification model where
input features = 3,
hidden layers = 2 with nodes = 4 in each layer and
number of target classes = 3



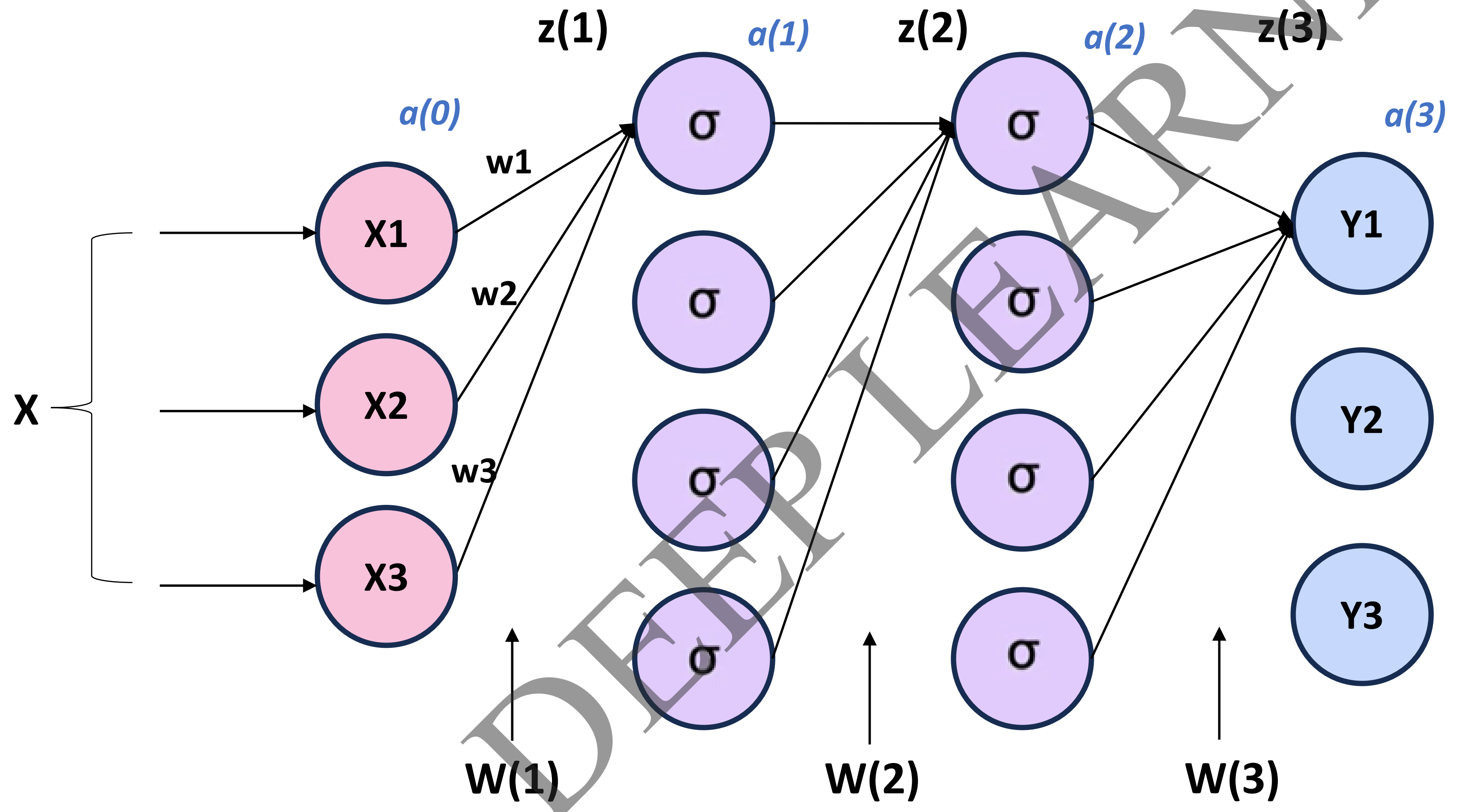
Both z and f(z) calculated at each of these neurons

Let us analyse the end to end operation

NEURAL NETWORKS – FEED FORWARD NETWORK

input features = 3,
hidden layers = 2 with nodes = 4 in each layer and
number of target classes = 3

Multi-layer Perceptron



X : input dataset
X1, X2, X3 : Input variables

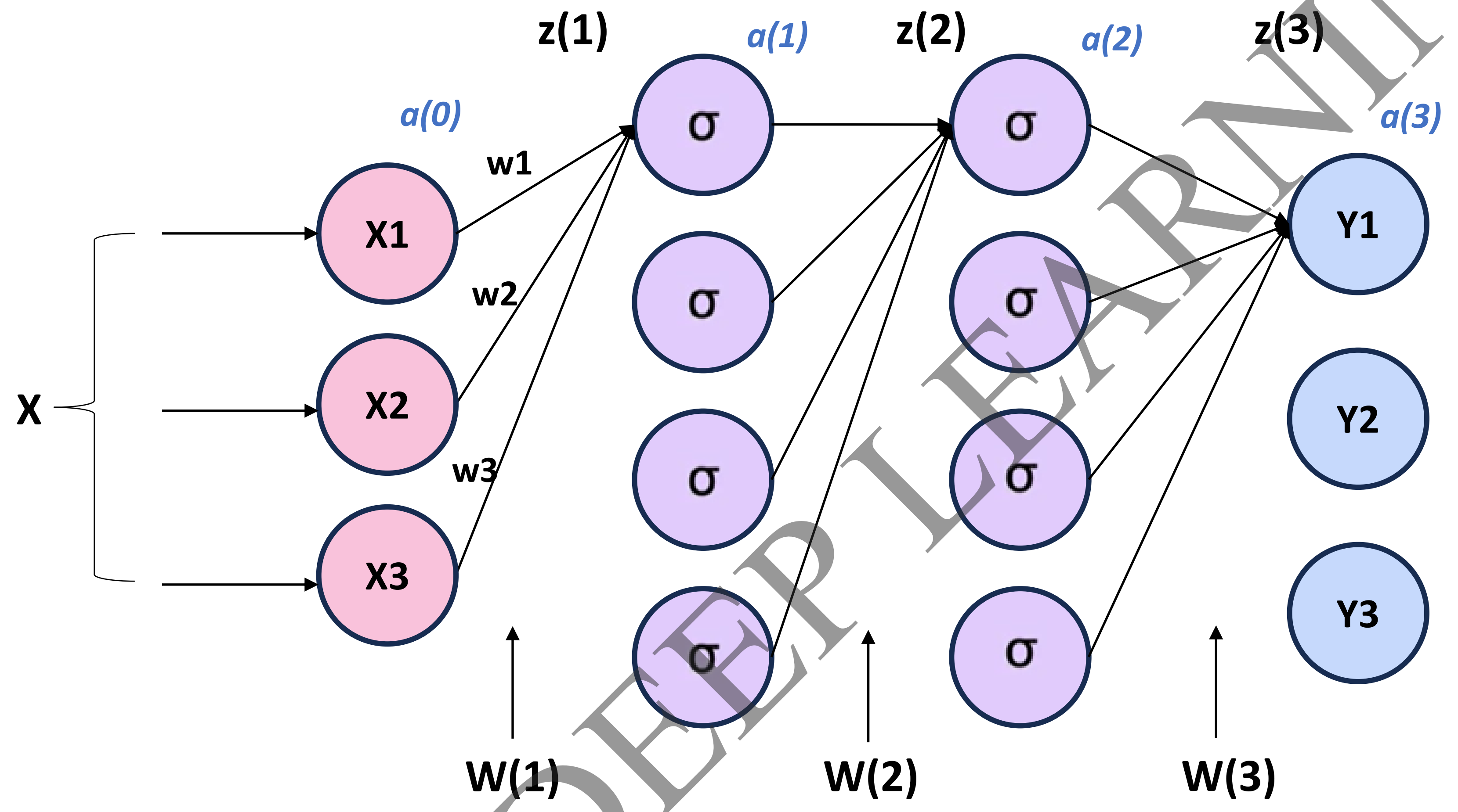
a : hidden / activation layer

W : weights matrix
W1, w2, w3 : individual weights

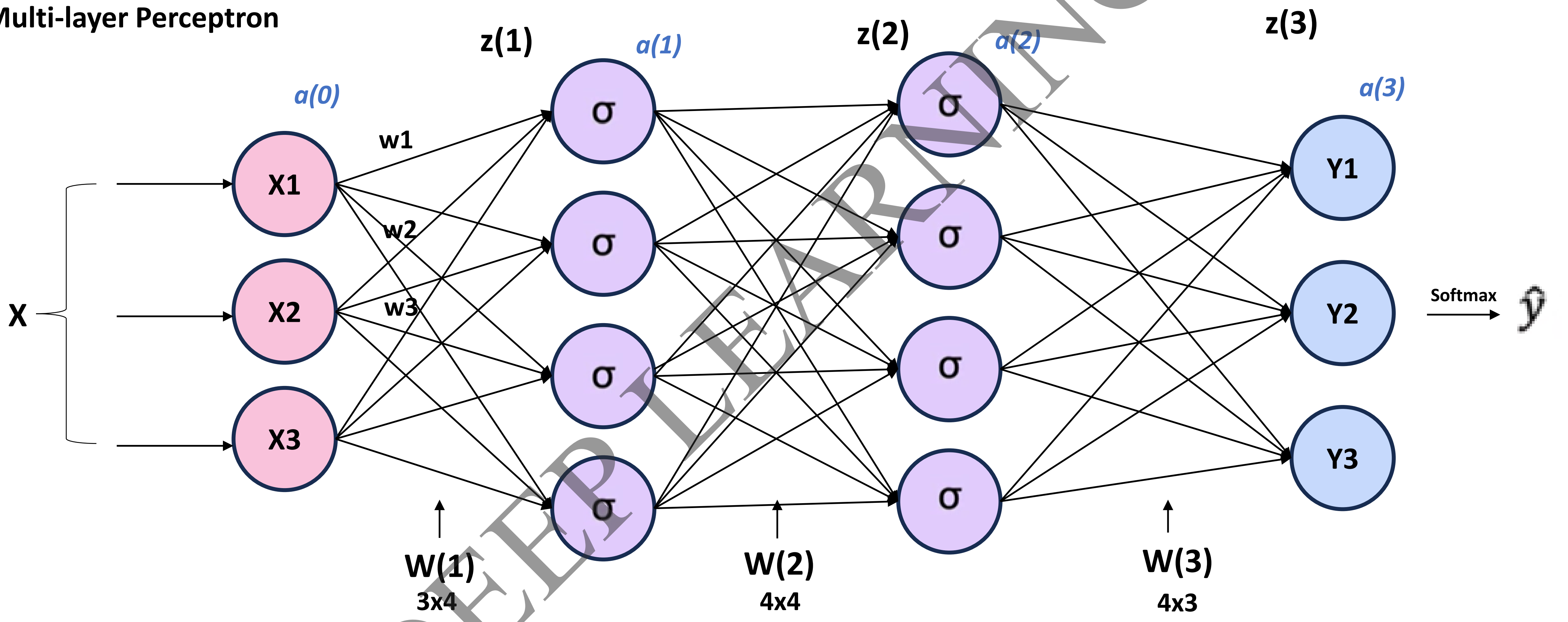
z : y = w*X + b

NEURAL NETWORKS – FEED FORWARD NETWORK

Multi-layer Perceptron



NEURAL NETWORKS – FEED FORWARD NETWORK



$z^{(1)} = xW^{(1)}$
$a^{(1)} = \sigma(z^{(1)})$

$\longrightarrow$

$z^{(2)} = a^{(1)}W^{(2)}$
$a^{(2)} = \sigma(z^{(2)})$

$\longrightarrow$

$z^{(3)} = a^{(2)}W^{(3)}$
$\hat{y} = softmax(z^{(3)})$

NEURAL NETWORKS – OPTIMIZATION

What does “Optimization” refer to in Neural Nets?

- **Initialization** - In a feed forward network, once the weights are assigned, the function f(z) is calculated and the output moves in a forward direction from one layer to another.
  The weights may or may not be the best combination for the node calculations, i.e. The weights “initialized” may not result in the best possible model at the final layer.
  We find the ‘*cost function*’ or ‘*loss function*’ or ‘*error*’ in the model to evaluate the model performance.

- **Weights second assignment** - Hence, after the first feed-forward network is built, the weights are then both increased & decreased to check how the model performance gets updated.

- **Re-Iterate** - Then we re-iterate the process in the same direction (either only increase or only decrease the weight) as long as the performance is improving.
  Each of the iterations are referred to as “steps”.

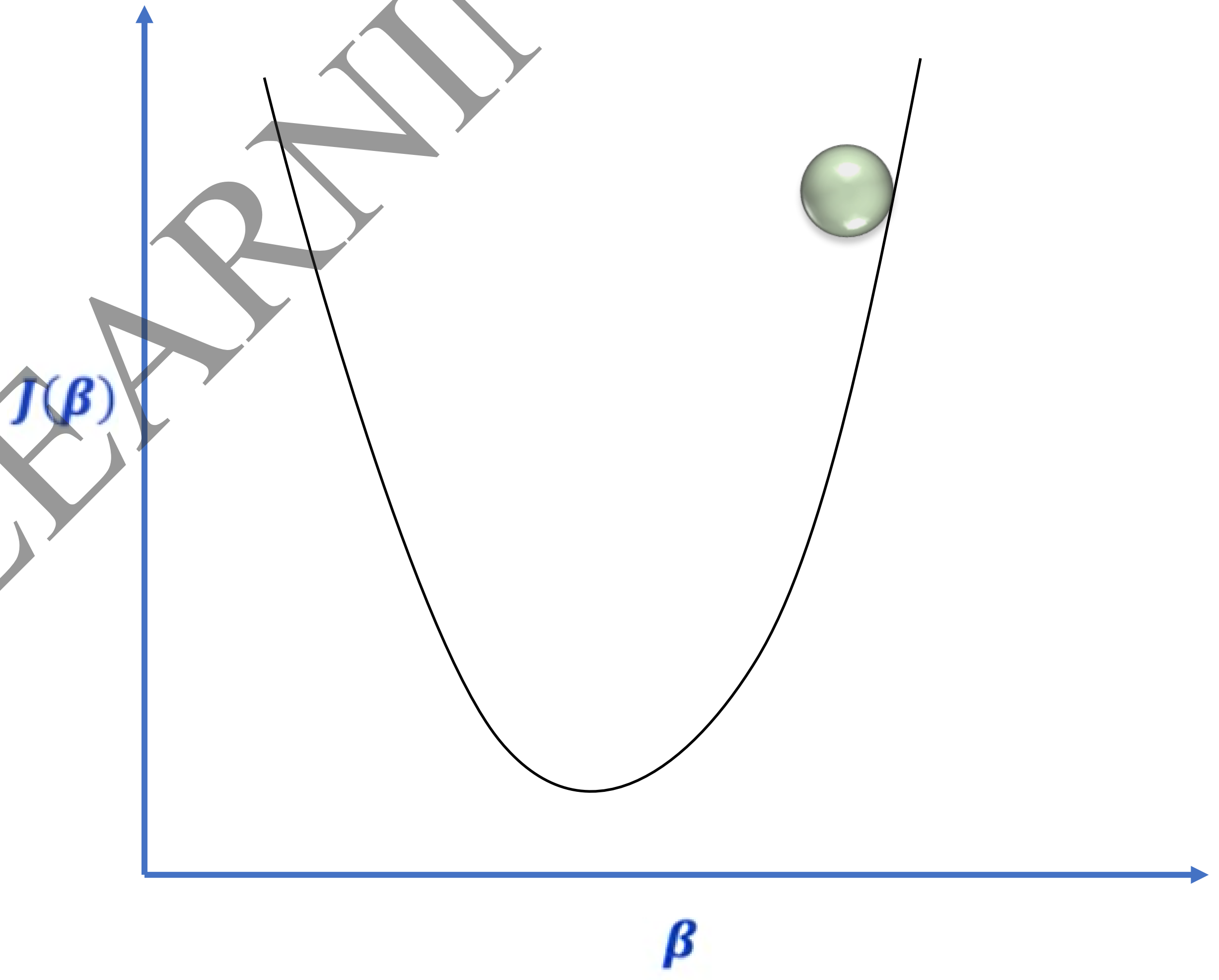
- **Peak Performance** - After a certain point, the model performance starts to decrease.
  This point of peak performance is called ‘**Gradient Descent’**.

Optimization refers to the method of finding the Gradient Descent in a neural network.

NEURAL NETWORKS – GRADIENT DESCENT

Working of Gradient Descent :

Step 1 : Initialization
Random weights are assigned to train the model.
Then, Cost function **J(β)** is calculated.

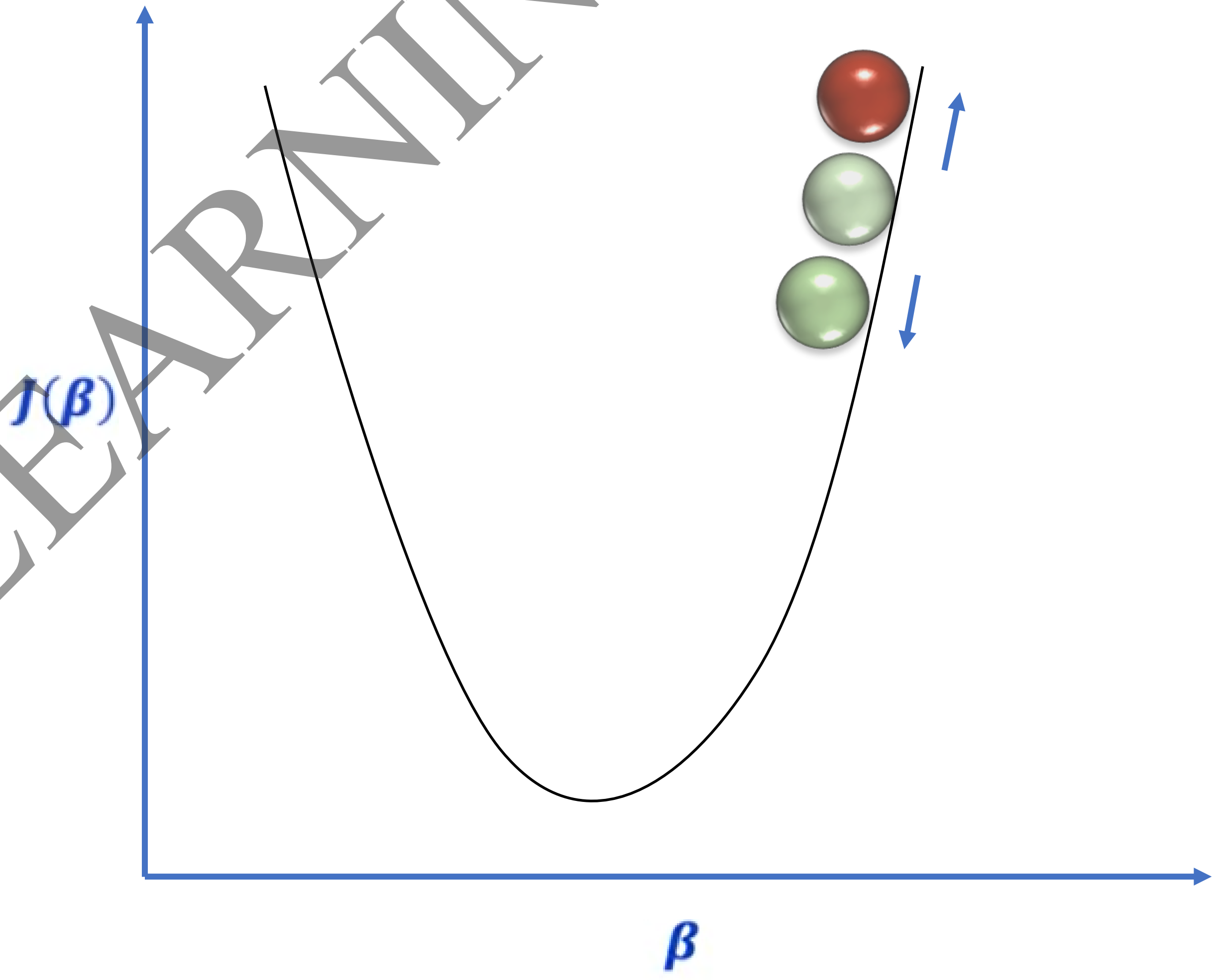


NEURAL NETWORKS – GRADIENT DESCENT

Working of Gradient Descent :

Step 1 : Initialization
Random weights are assigned to train the model.
Then, Cost function **J(β)** is calculated.

Step 2 : Second Assignment
Weights are both increased and decreased to train the model.
Then, Cost function **J(β)** is calculated for both.
Assuming if reducing the weights reduces the cost function, then we move the further iterations in that direction



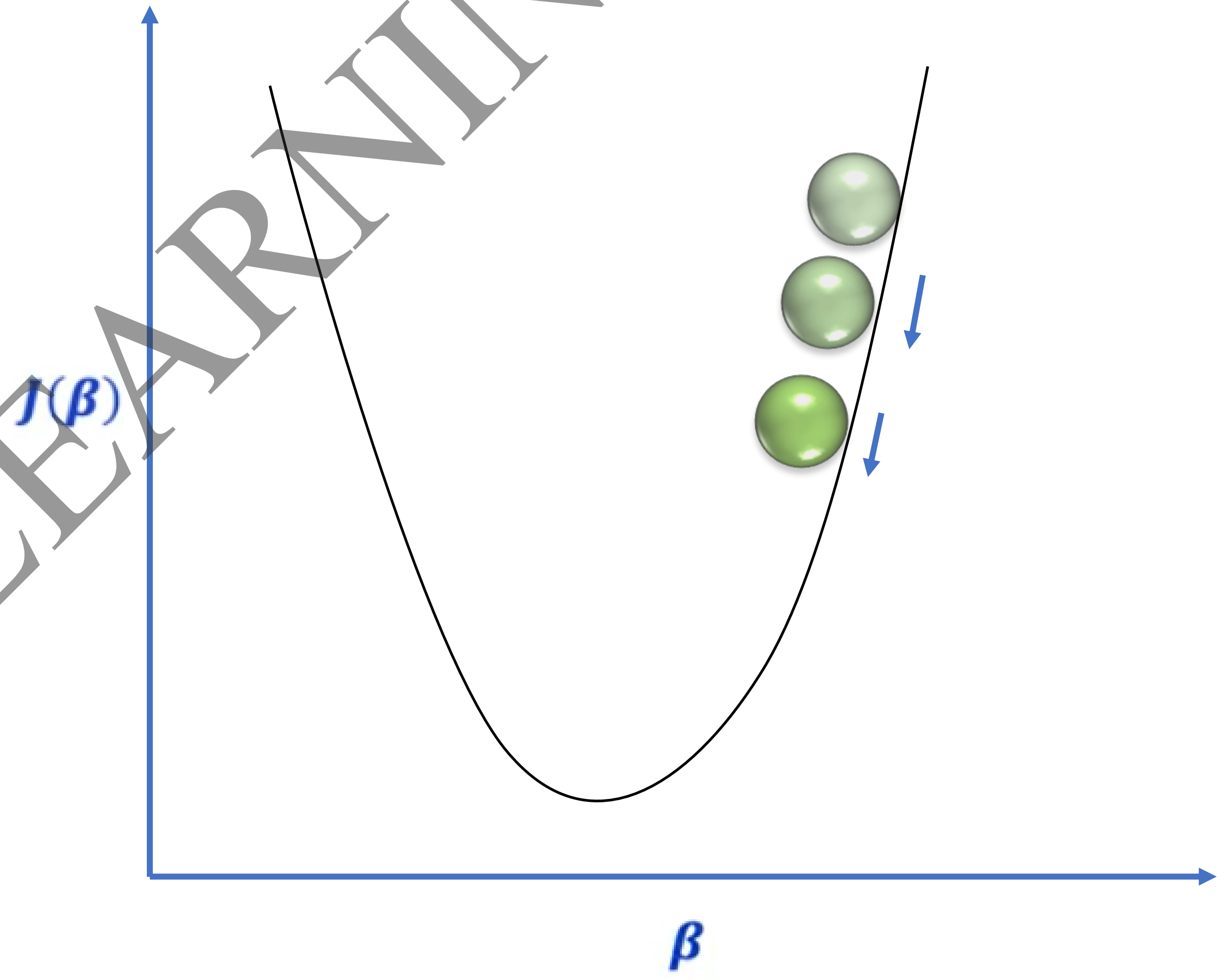
NEURAL NETWORKS – GRADIENT DESCENT

Working of Gradient Descent :

Step 1 : Initialization
Random weights are assigned to train the model.
Then, Cost function **J(β)** is calculated.

Step 2 : Second Assignment
Weights are both increased and decreased to train the model.
Then, Cost function **J(β)** is calculated for both.
Assuming if reducing the weights reduces the cost function, then we move the further iterations in that direction

Step 3 : Re-iteration
We keep iterating in the same direction and Cost function **J(β)** is calculated at each 'step' to validate thar loss is getting reduced.



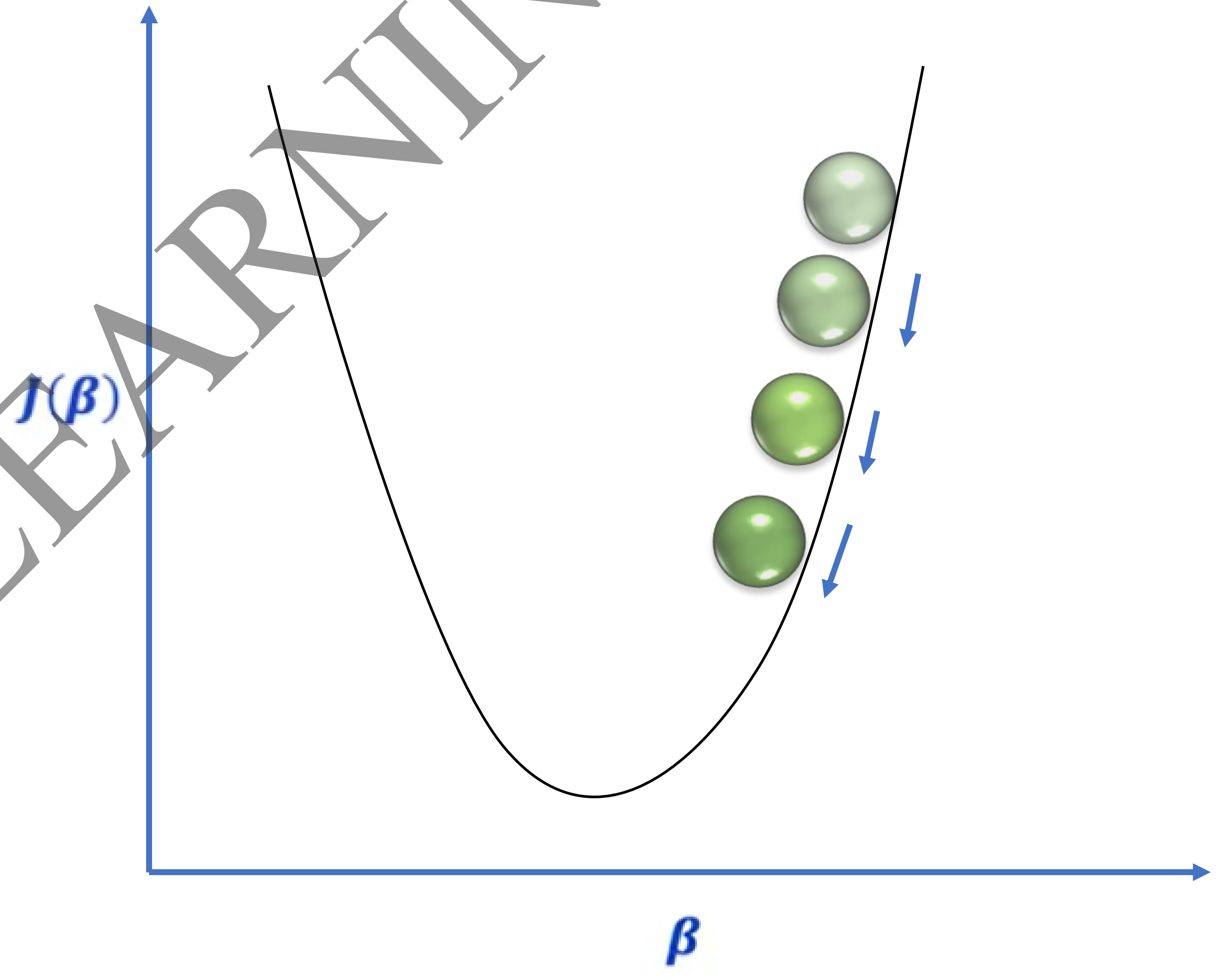
NEURAL NETWORKS – GRADIENT DESCENT

Working of Gradient Descent :

Step 1 : Initialization
Random weights are assigned to train the model.
Then, Cost function **J(β)** is calculated.

Step 2 : Second Assignment
Weights are both increased and decreased to train the model.
Then, Cost function **J(β)** is calculated for both.
Assuming if reducing the weights reduces the cost function, then we move the further iterations in that direction

Step 3 : Re-iteration
We keep iterating in the same direction and Cost function **J(β)** is calculated at each 'step' to validate thar loss is getting reduced.



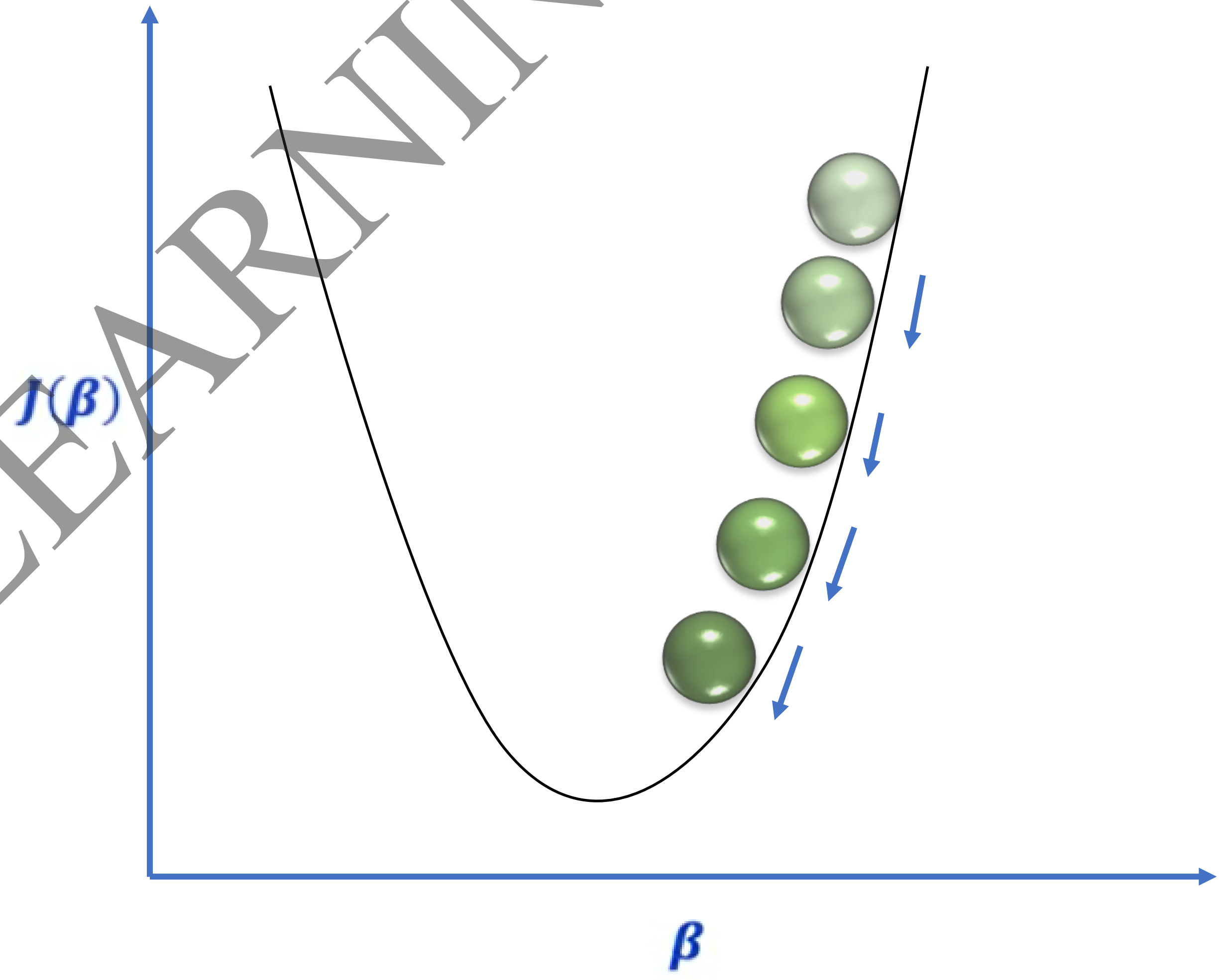
NEURAL NETWORKS – GRADIENT DESCENT

Working of Gradient Descent :

Step 1 : Initialization
Random weights are assigned to train the model.
Then, Cost function **J(β)** is calculated.

Step 2 : Second Assignment
Weights are both increased and decreased to train the model.
Then, Cost function **J(β)** is calculated for both.
Assuming if reducing the weights reduces the cost function, then we move the further iterations in that direction

Step 3 : Re-iteration
We keep iterating in the same direction and Cost function **J(β)** is calculated at each 'step' to validate thar loss is getting reduced.



NEURAL NETWORKS – GRADIENT DESCENT

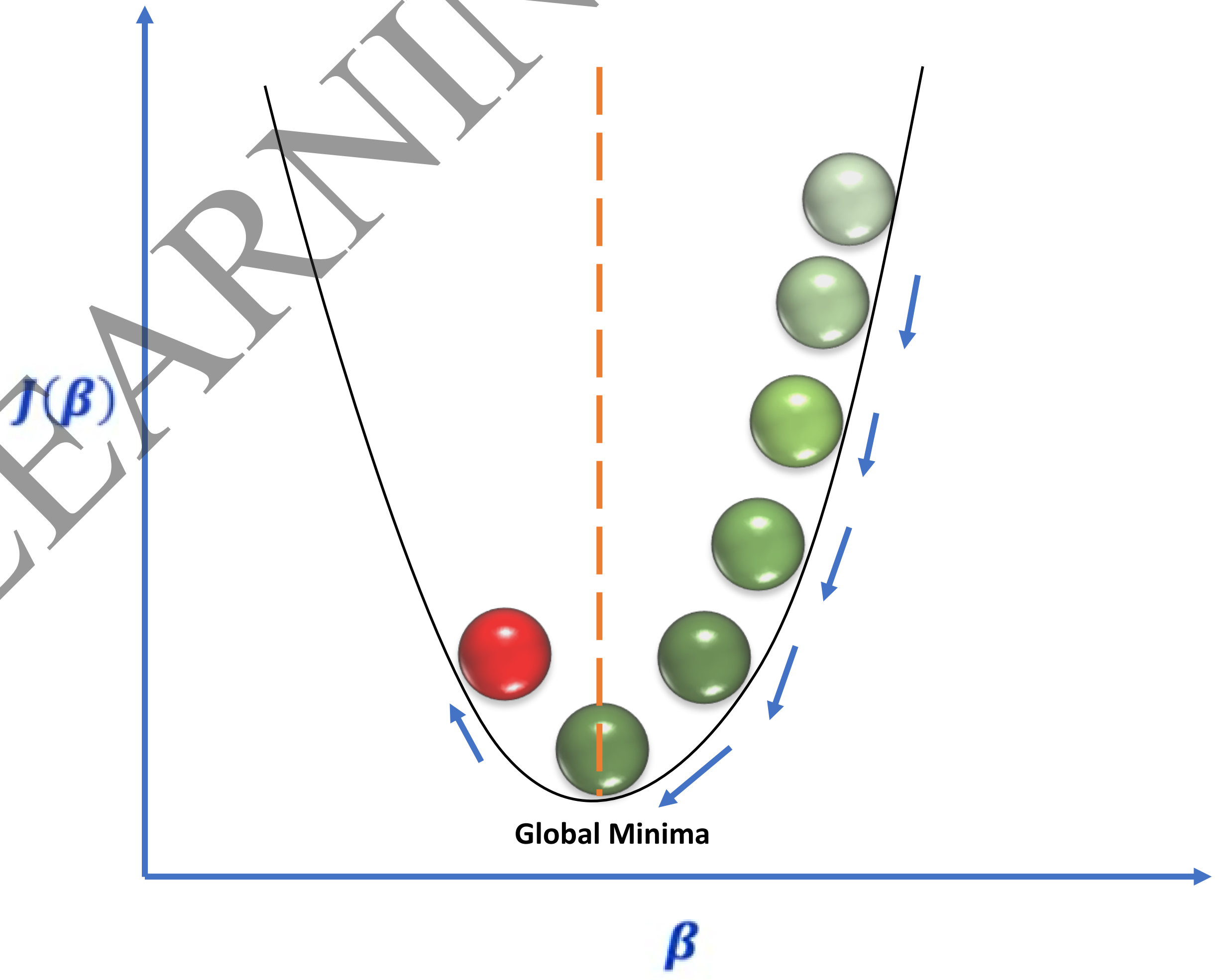
Working of Gradient Descent :

Step 1 : Initialization
Random weights are assigned to train the model.
Then, Cost function **J(β)** is calculated.

Step 2 : Second Assignment
Weights are both increased and decreased to train the model.
Then, Cost function **J(β)** is calculated for both.
Assuming if reducing the weights reduces the cost function, then we move the further iterations in that direction

Step 3 : Re-iteration
We keep iterating in the same direction and Cost function **J(β)** is calculated at each 'step' to validate thar loss is getting reduced.

Step 4 : Peak Performance / Gradient Descent
Once the global minima is reached (***J(β) is minimum***), we can say that the Gradient Descent is achieved and the model is optimized.



NEURAL NETWORKS – GRADIENT DESCENT

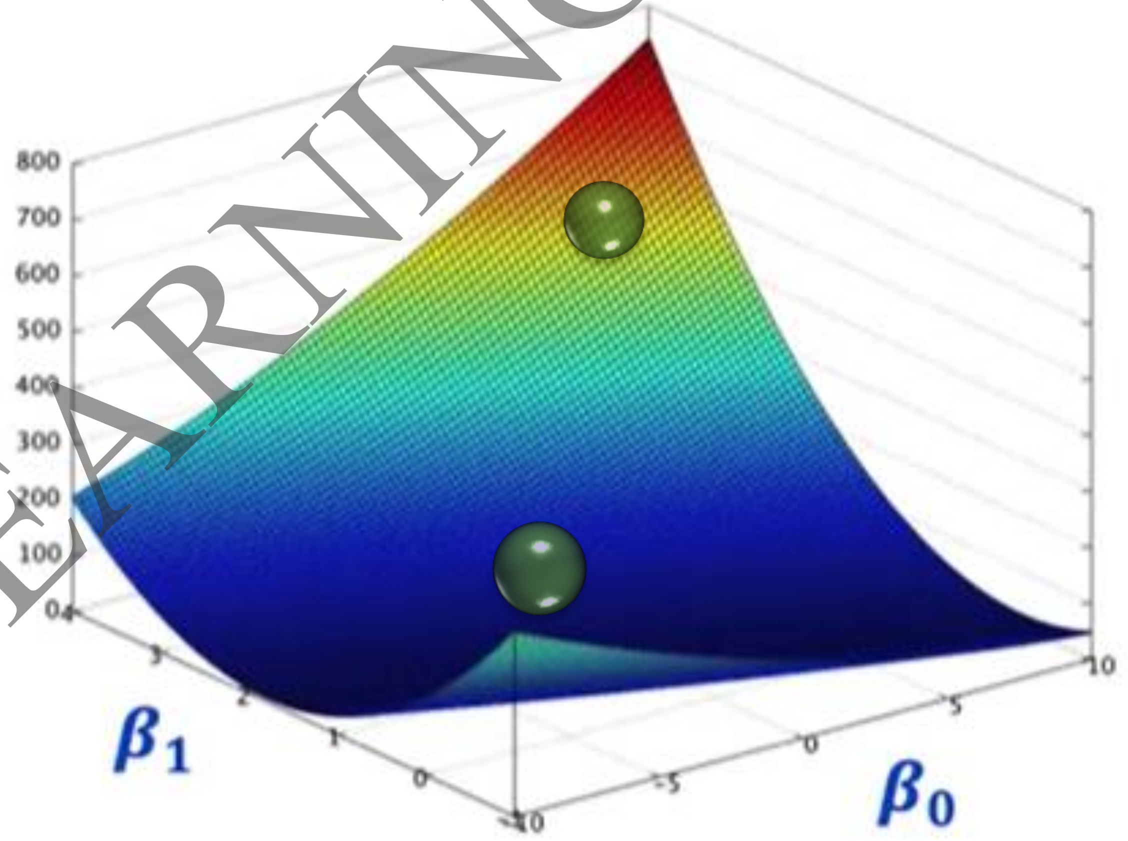
Working of Gradient Descent with (β0, β1):

Step 1 : Initialization
Random weights are assigned to train the model.
Then, Cost function **J(β0, β1)** is calculated.

Step 2 : Second Assignment
Weights are both increased and decreased to train the model.
Then, Cost function **J(β0, β1)** is calculated for both.
Assuming if reducing the weights reduces the cost function, then we move the further iterations in that direction

Step 3 : Re-iteration
We keep iterating in the same direction and Cost function **J(β0, β1)** is calculated at each 'step' to validate thar loss is getting reduced.

Step 4 : Peak Performance / Gradient Descent
Once the global minima is reached (<u>**J(β0, β1)** *is minimum*</u>), we can say that the Gradient Descent is achieved and the model is optimized.



- PRITESH JHA

NEURAL NETWORKS – GRADIENT DESCENT

Types of Gradient Descent

There are 3 types of Gradient Descent –

- Full Batch Gradient Descent
- Stochastic Gradient Descent
- Mini Batch Gradient Descent

The purpose of having different types of GD is only to have a system with balanced <u>space and time complexity</u>.

Full batch Gradient Descent

- Uses all the data (without sampling) to run each step.
  This makes the space complexity very high.

- Since it uses all the data at once, the number of steps / iterations required are significantly lower.
  This makes the time complexity low.

- Path to gradient descent is more direct.

NEURAL NETWORKS – GRADIENT DESCENT

Stochastic Gradient Descent

- Uses only one data point to run each step.
  This makes the space complexity very low.

- Since it uses only one data point at once, the number of steps / iterations required are significantly high.
  This makes the time complexity high / the system is very slow.

- Path to gradient descent is very complex. Prone to noise in the data.

Mini batch Gradient Descent

- Uses sampling technique to run each step in small batches of the data.
  This makes the space complexity comparatively lower than Full batch GD.

- Since it uses the data in small batches, the number of steps / iterations required are still higher than Full Batch GD but much faster than Stochastic GD.
  This makes the time complexity moderate.

- Due to having a moderate space as well as time complexity, this is often considered as the best of both worlds.

NEURAL NETWORKS – BACK PROPOGATION

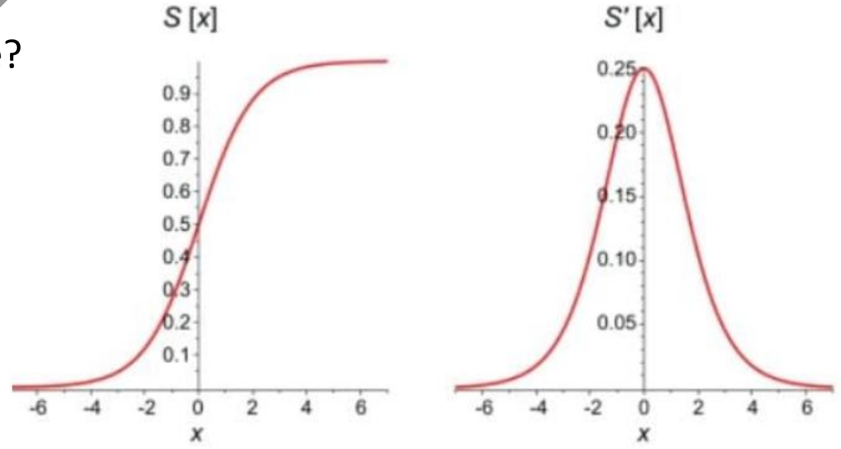
We have studied how to train Neural Networks.

- Put in the input variables.
- Initialize Weights and get output
- Calculate Loss Function. Adjust weights (increase or decrease) and repeat to achieve the Gradient Descent.

But how do we actually change the weights? How do we know how much to change?

$$W_{new} = W_{old} - \eta * \frac{\partial C}{\partial w}$$

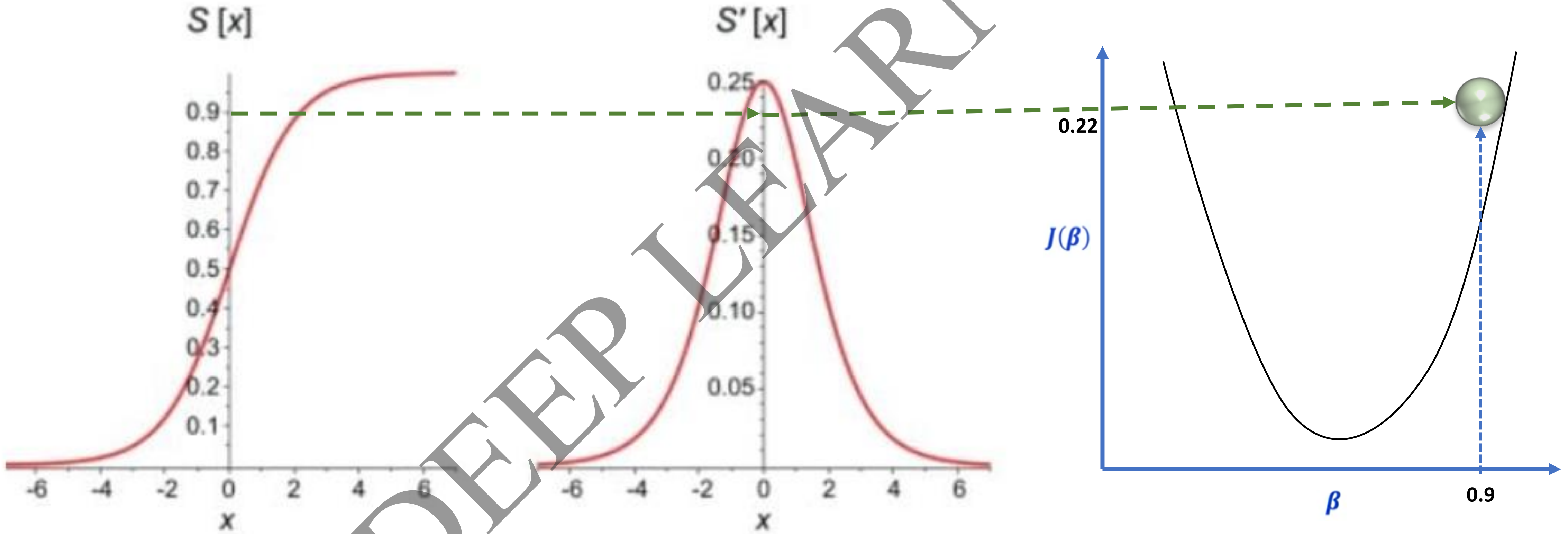
where
η = learning rate (usually very small like 0.1)
C = cost function



NEURAL NETWORKS – BACK PROPOGATION

Step 1 : For weight = 0.9, we build Initial Feed forward network and calculate the loss function using derivatives.

The next weight is going to be 0.9 – 0.22 = 0.68



Sigmoid Activation Function

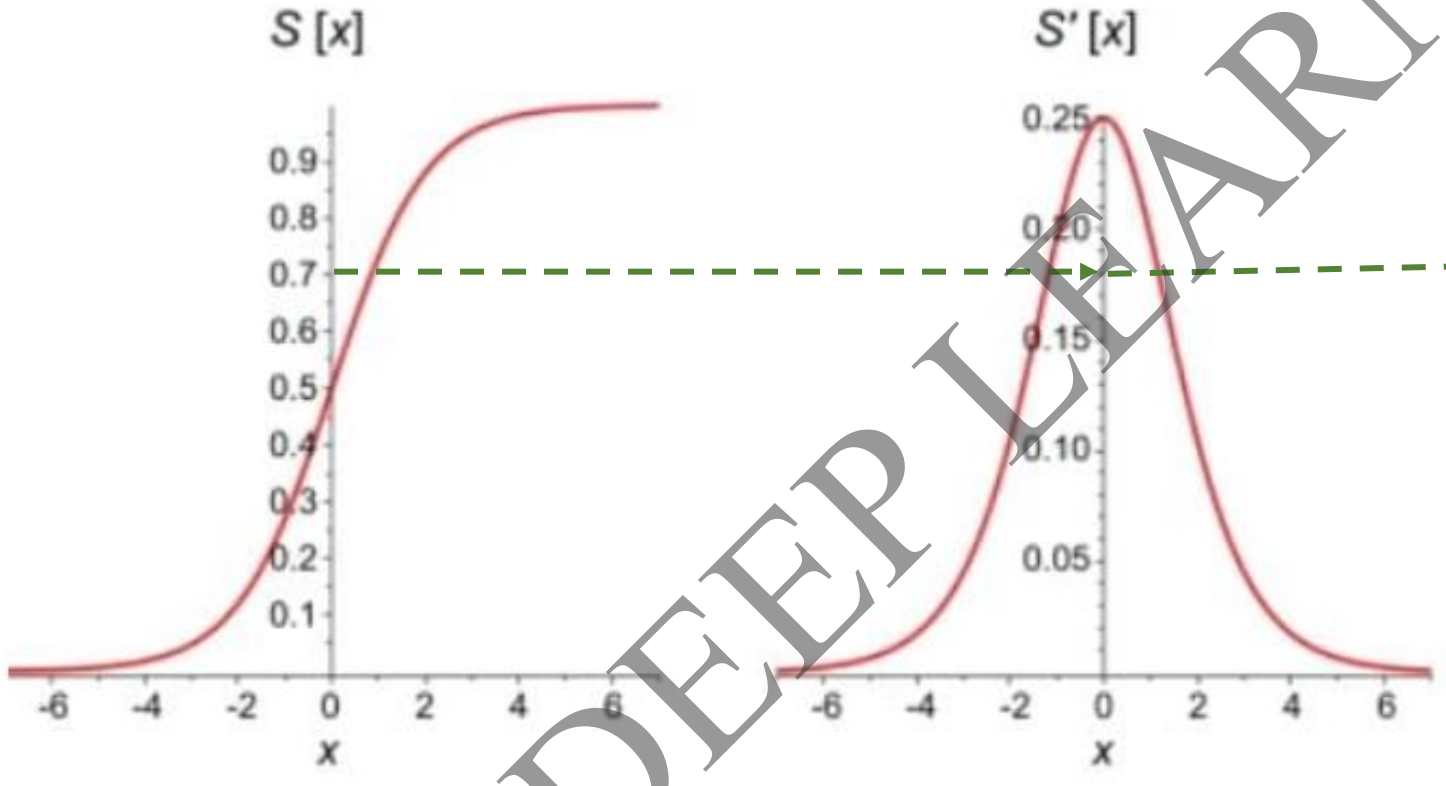
$\sigma'(z) = \sigma(z)(1 - \sigma(z))$

Loss Function Plot

NEURAL NETWORKS – BACK PROPOGATION

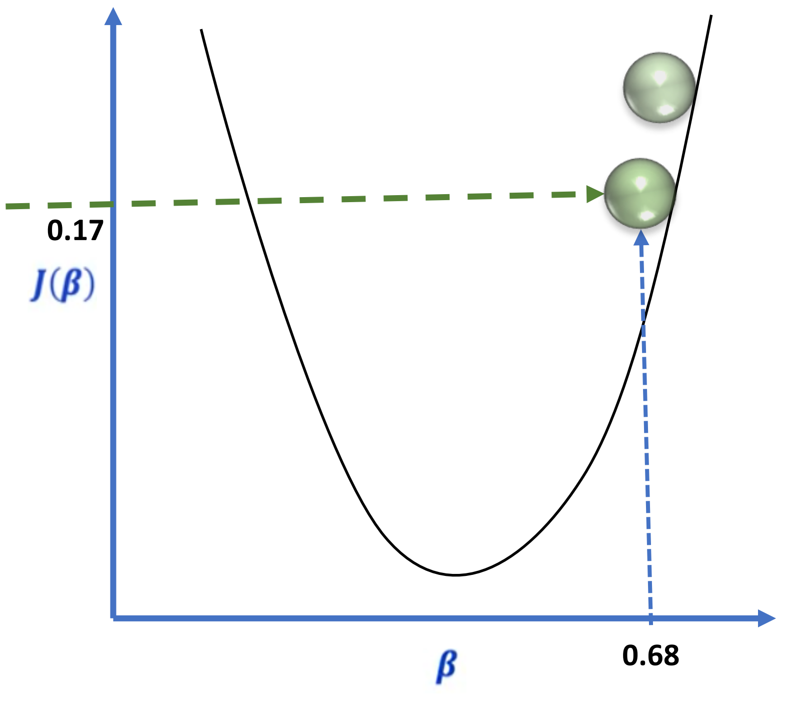
Step 2 : Calculate the new weights using the following formula : $W_{new} = W_{old} - \eta * \frac{\partial C}{\partial w}$

The updated value comes at 0.68 – 0.17 = 0.51



Sigmoid Activation Function

$\sigma'(z) = \sigma(z)(1 - \sigma(z))$



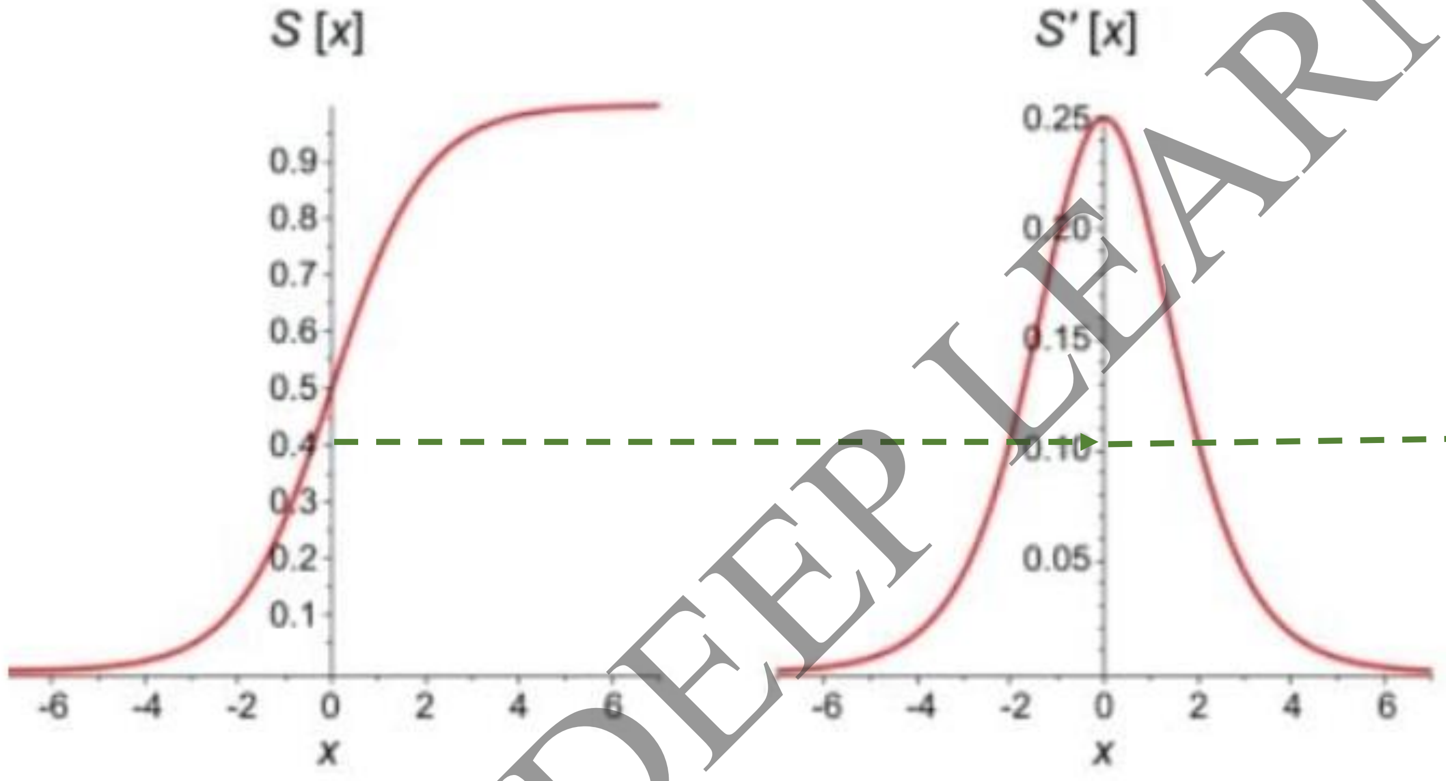
Loss Function Plot

NEURAL NETWORKS – BACK PROPOGATION

Step 3 : Calculate the new weights using the following formula : $W_{new} = W_{old} - \eta * \frac{\partial C}{\partial w}$

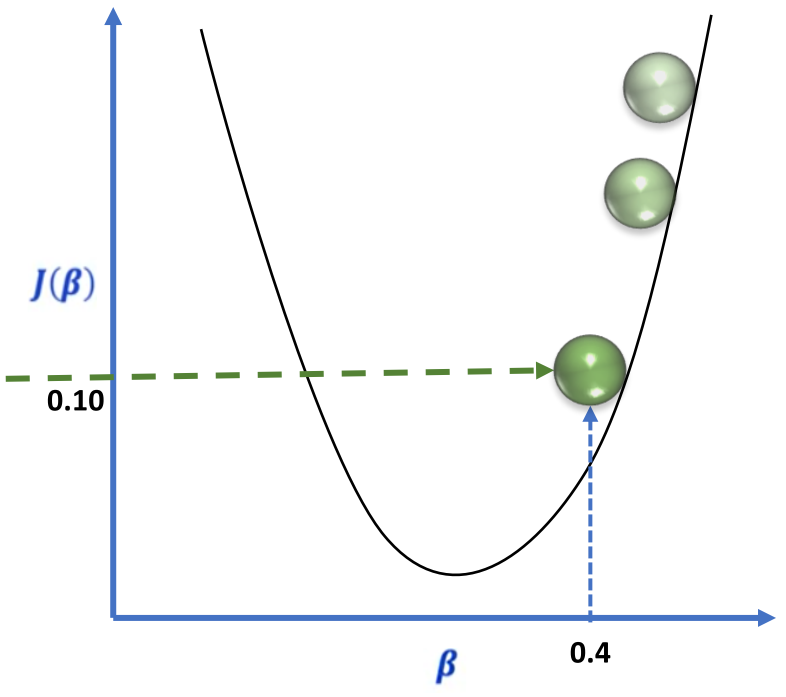
Let us say that after a few iterations, the updated value comes at 0.4

The next value would be 0.4 – 0.1 = 0.3



Sigmoid Activation Function

$$\sigma'(z) = \sigma(z)(1 - \sigma(z))$$



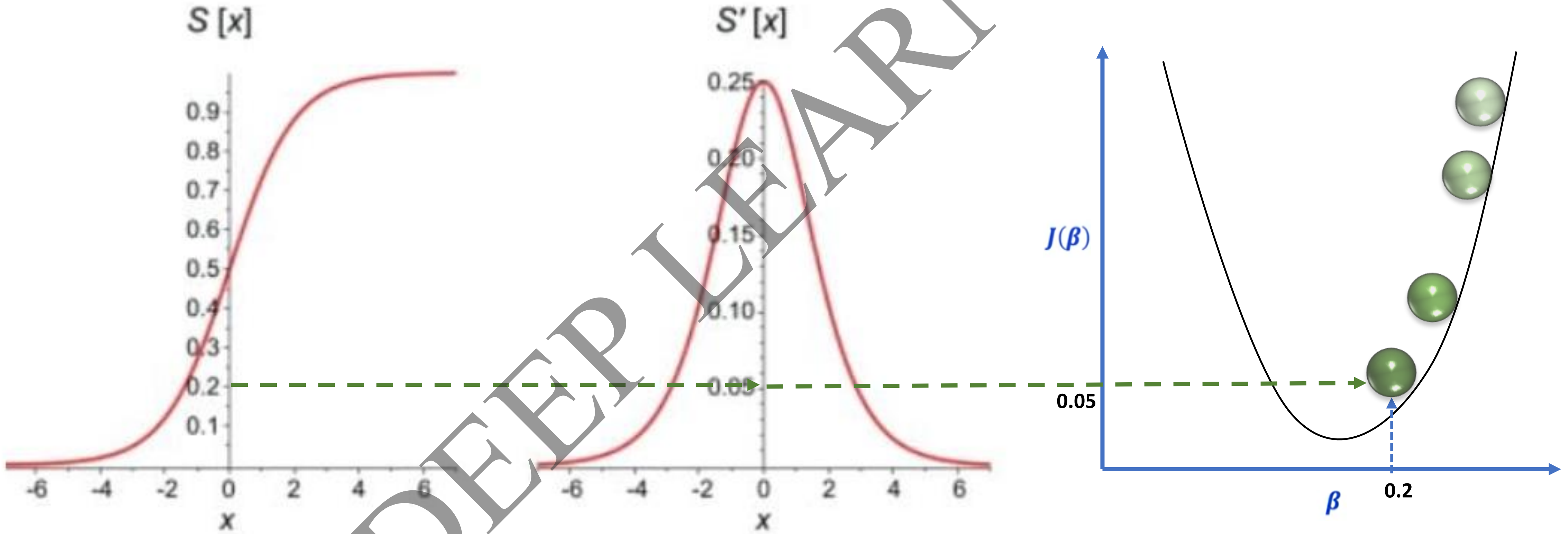
Loss Function Plot

Note : Values are assumed for representation purpose only

NEURAL NETWORKS – BACK PROPOGATION

Step 4 : Calculate the new weights using the following formula : $W_{new} = W_{old} - \eta * \frac{\partial C}{\partial w}$

Let us say that the updated value after few iterations comes at 0.2



Sigmoid Activation Function

$\sigma'(z) = \sigma(z)(1 - \sigma(z))$

Loss Function Plot

Note : Values are assumed for representation purpose only

NEURAL NETWORKS – BACK PROPOGATION

Once the weights start to fall even below, the partial derivative of sigmoid, becomes very close to **zero.**

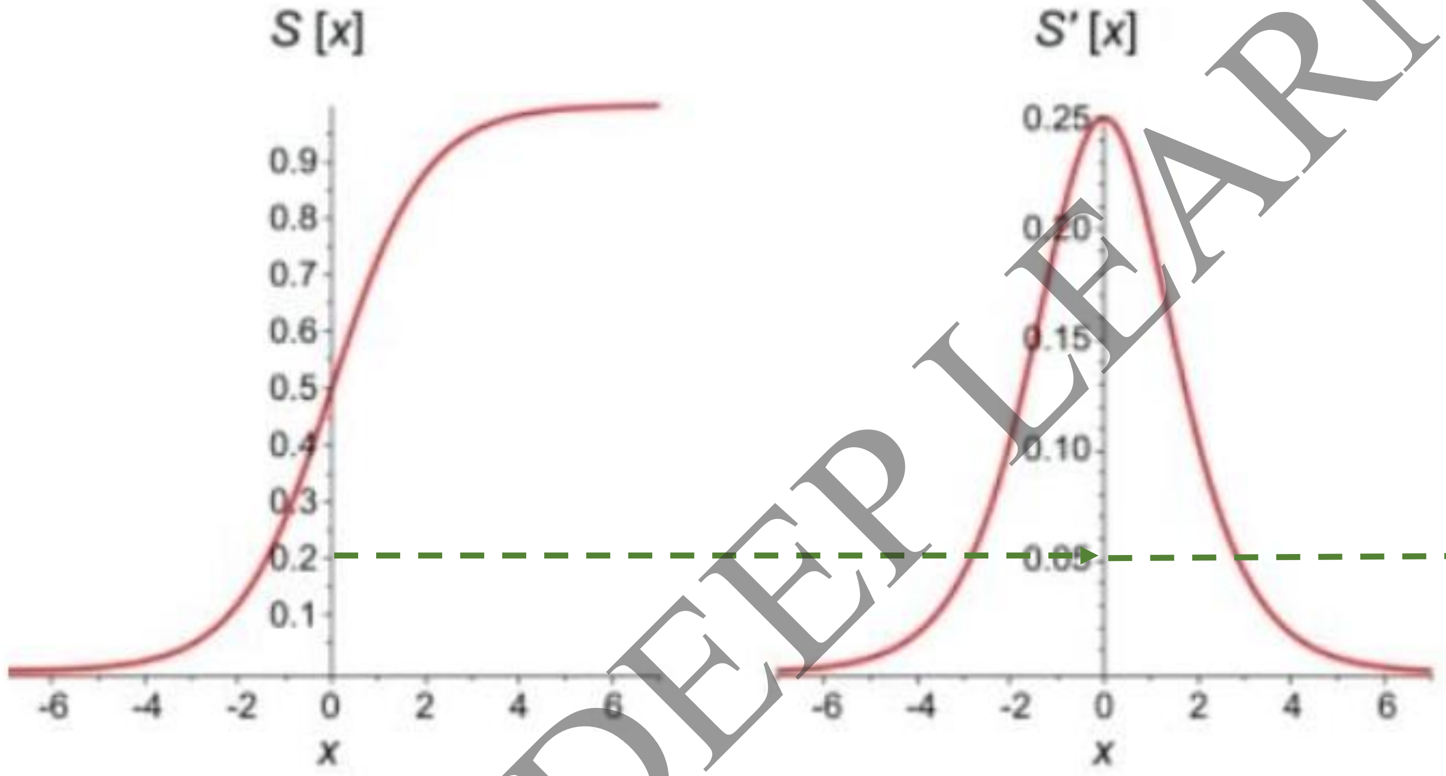
As the weights start to reduce, there may come a moment when the derivative becomes zero.
Due to this, the weights will not be reduced any further and the global minima will not be met.
This problem is called as "**Vanishing gradient**" problem.

$$W_{new} = W_{old} - \eta * \frac{\partial C}{\partial w}$$

$$W_{new} = W_{old} - 0$$

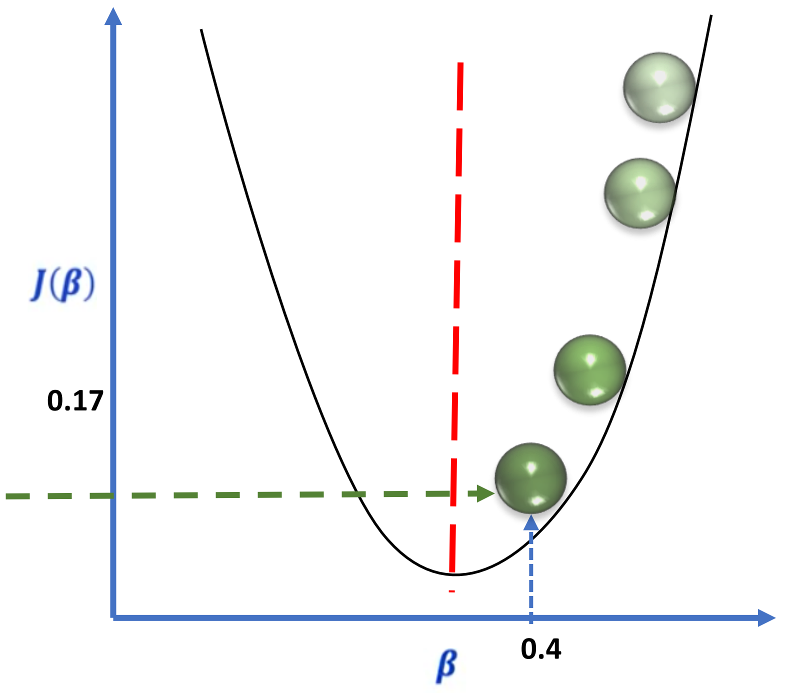
$$W_{new} = W_{old}$$

Vanishing Gradient Problem



Sigmoid Activation Function

$$\sigma'(z) = \sigma(z)(1 - \sigma(z))$$

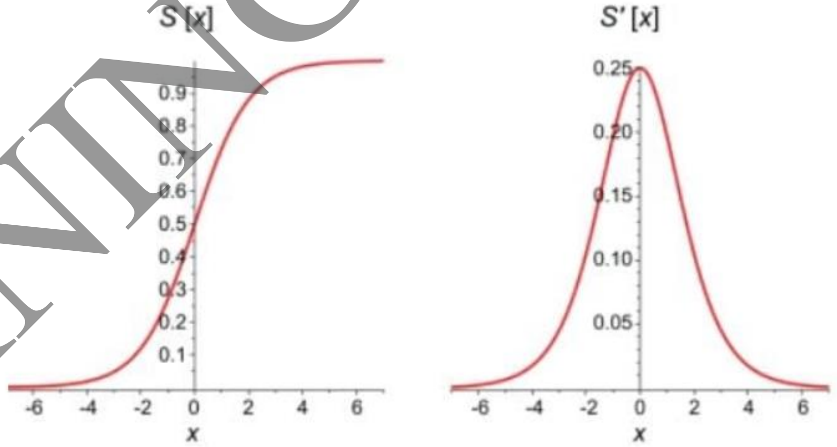


Loss Function Plot

NEURAL NETWORKS – BACK PROPOGATION

$$W_{new} = W_{old} - \eta * \frac{\partial C}{\partial w}$$

where
η = learning rate (usually very small like 0.1)
C = cost function



With the chain rule of partial derivatives, we can represent gradient of the loss function as a product of gradients of all the activation functions of the nodes with respect to their weight. Therefore, the updated weights of nodes in the network depend on the gradients of the activation functions of each node.

For the nodes with sigmoid activation functions, we know that the partial derivative of the sigmoid function reaches a maximum value of 0.25.

When there are more layers in the network, the value of the product of derivative decreases until at some point the partial derivative of the loss function approaches a value close to zero, and the partial derivative vanishes. This is called as **Vanishing gradient** problem.

NEURAL NETWORKS – Optimizers

Optimizers

We have considered the different optimization methods such as Full Batch GD, Stochastic GD and Mini Batch GD.
However, all of them have used the same formula to update the new weights as follows :

$$W_{new} = W_{old} - \eta * \frac{\partial C}{\partial w} \qquad W := W - \alpha \cdot \nabla J$$

There are some more methods to find the new weights. These methods are called as **Optimizers**

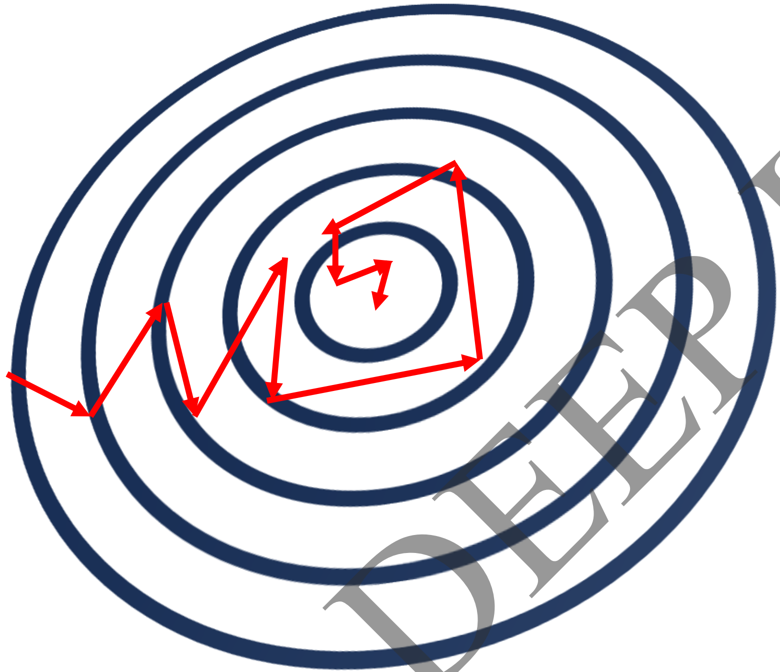
NEURAL NETWORKS – Optimizers

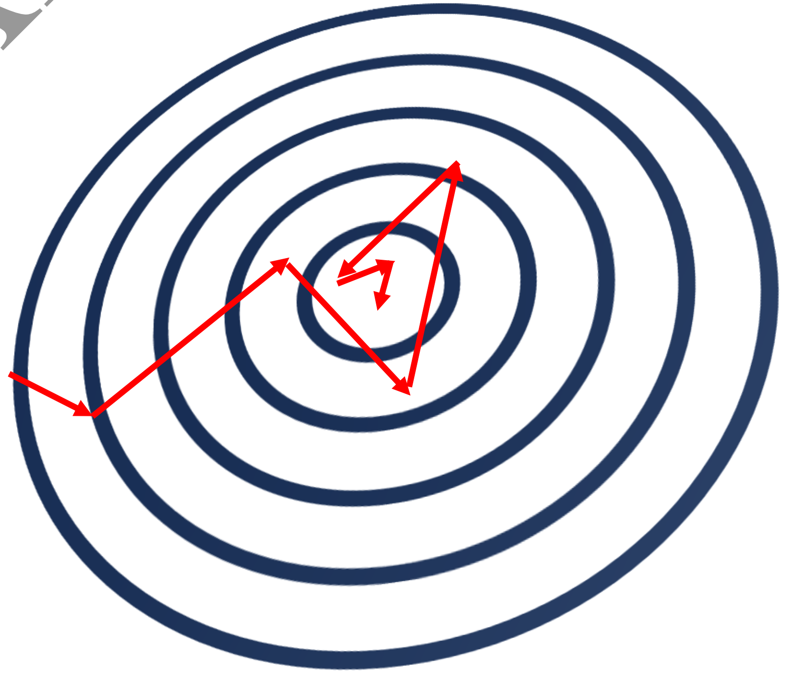
Momentum

The idea here is to keep a "running average" of the step directions, smoothing out the variation at each points.

$$v_t \coloneqq \eta \cdot v_{t-1} - \alpha \cdot \nabla J$$
$$W \coloneqq W - v_t$$

Here η is referred to as momentum. It is generally < 1.
This method is quite prone to the "**overshooting**" problem.


Gradient Descent


Gradient Descent with Momentum

NEURAL NETWORKS – Optimizers

AdaGrad

Idea: scale the update for each weight separately.

1. Update frequently-updated weights less.
2. Keep running sum of previous updates.
3. Divide new updates by factor of previous sum.

With starting point $G_i(0) = 0$:

$$G_i(t) = G_i(t-1) + (\frac{\partial L}{\partial w_i}(t))^2$$

G will continue to increase

$$W := W - \frac{\eta}{\sqrt{G_t} + \epsilon} \cdot \nabla J$$

This leads to smaller updates each iteration

RMSProp

Quite similar to AdaGrad.

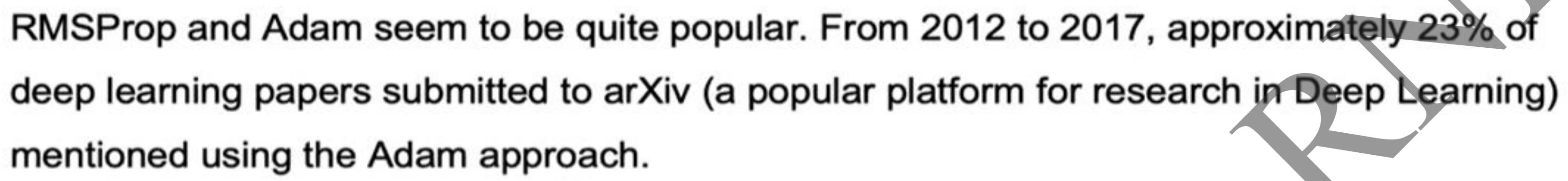
– Rather than using the sum of previous gradients, decay older gradients more than more recent ones.

– More adaptive to recent updates.

NEURAL NETWORKS – Optimizers

Adam (Adaptive Moment Estimation)

The idea is to utilize both momentum and RMSProp concepts



RMSProp and Adam seem to be quite popular. From 2012 to 2017, approximately 23% of deep learning papers submitted to arXiv (a popular platform for research in Deep Learning) mentioned using the Adam approach.

NEURAL NETWORKS – Optimizers

	Momentum	AdaGrad	RMSProp	ADAM
Concept	Incorporates the 'momentum' of its updates to keep on moving in the direction of steepest descent. It helps to accelerate the gradient descent algorithm by considering the past gradients to smooth out the update. It does this by adding a fraction of the direction of the previous step to the current step, effectively increasing the speed of convergence.	Adapts the learning rate for each parameter, giving low learning rates to parameters with frequently occurring features and higher learning rates to parameters with infrequent features. It achieves this by accumulating the square of the gradients in the denominator; as the accumulated value grows, the learning rate shrinks.	Tries to resolve AdaGrad's radically diminishing learning rates by using a moving average of squared gradients. It adjusts the learning rate for each weight based on the mean of recent magnitudes of the gradients for that weight.	Combines ideas from both Momentum and RMSProp. Besides storing an exponentially decaying average of past squared gradients like RMSProp, Adam also keeps an exponentially decaying average of past gradients, similar to momentum. It calculates adaptive learning rates for each parameter.
Advantage	Helps to prevent oscillations and speeds up convergence, particularly in areas of the objective function that have a shallow gradient.	Very effective for sparse data (lots of zeros in data), as it adapts the learning rate to the frequency of parameters.	Solves the diminishing learning rate problem of AdaGrad, making it suitable for both non-stationary and stationary problems.	Requires less memory and is computationally efficient. It is well suited for problems that are large in terms of data/parameters or problems with noisy/spare gradients. Adam generally works well in practice and outperforms other adaptive learning-method algorithms.
Disadvantage	Might overshoot the minimum because of the momentum it accumulates.	The continuously accumulating squared gradients in the denominator can cause the learning rate to shrink and become infinitesimally small, which essentially stops the network from learning further.	Still requires manual tuning of the learning rate.	The adaptive learning rate can sometimes lead to overshooting in the initial stages of the training because of the high moments. Also, it might not converge to the optimal solution under certain conditions as theoretically expected, requiring more tuning of hyperparameters like the initial learning rate.

NEURAL NETWORKS – ACTIVATION FUNCTION

What are Activation functions?

Activation functions are the mathematical function that is applied to the linear output of the neurons which will be able to define the use of our neural network application.
For example, Sigmoid activation function can be used to classify the output into 0 and 1.

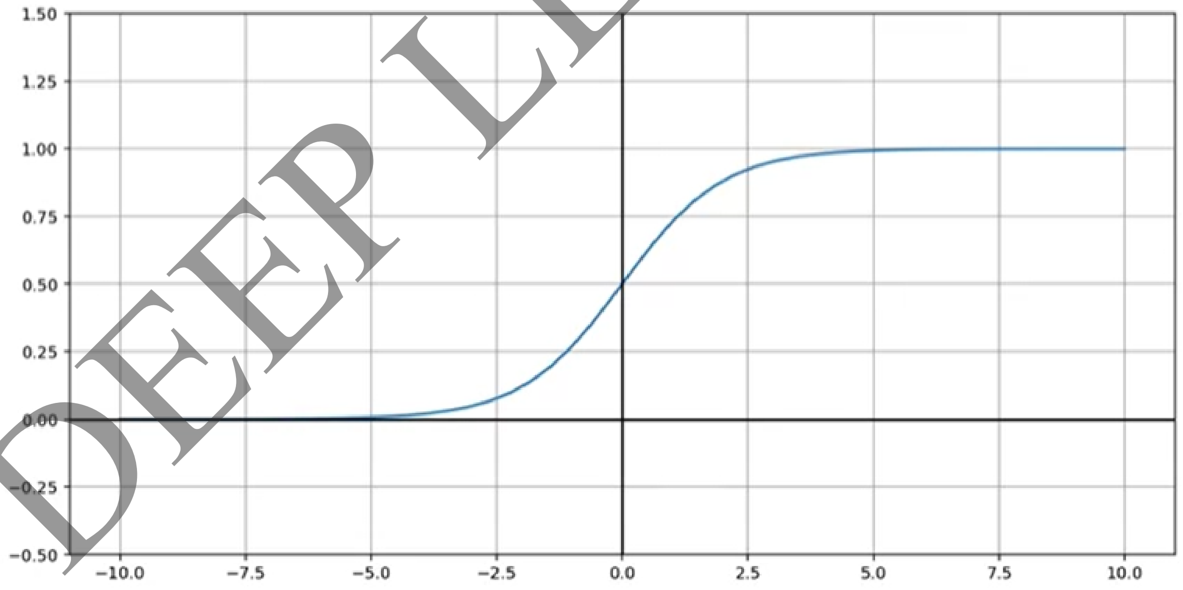
Different activation functions are used based on the type of problem statement.

Sigmoid activation function

This is called the "sigmoid" function: $\sigma(z) = \frac{1}{1+e^{-z}}$

Useful when outcomes should be in (0,1)

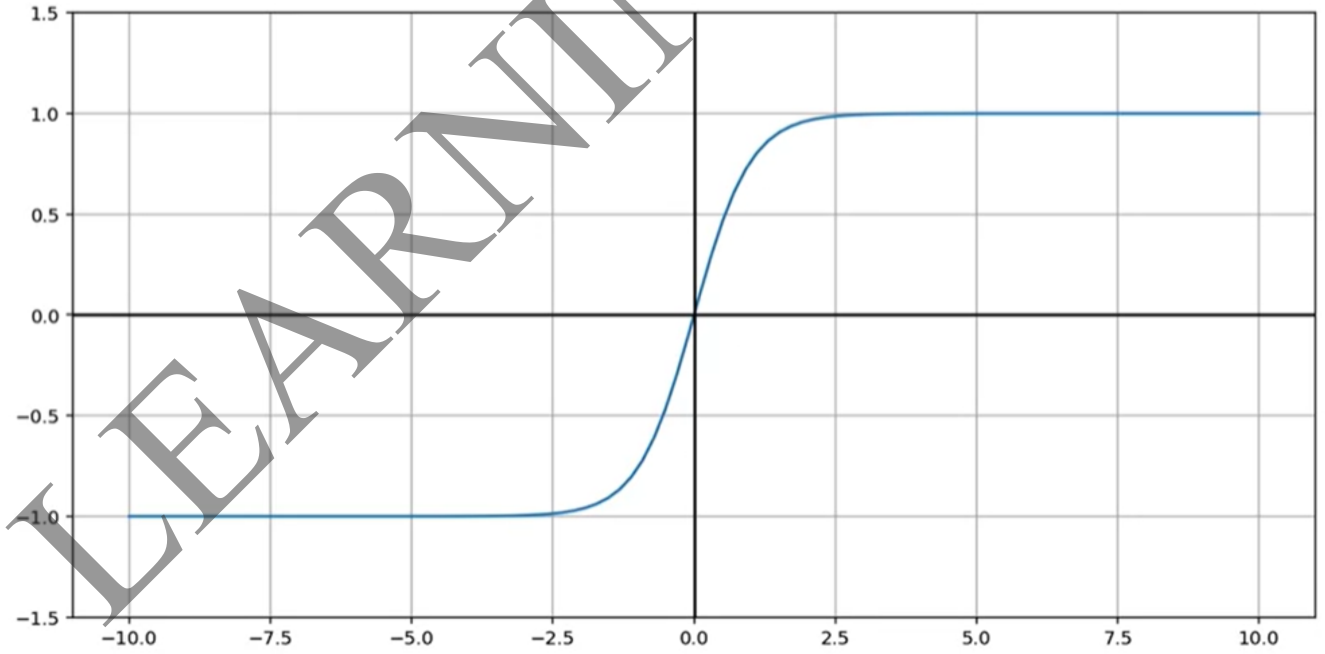
Not immune to Vanishing Gradient Problem



NEURAL NETWORKS – ACTIVATION FUNCTION

Hyperbolic Tangent activation function

$$tanh(z) = \frac{\sinh(z)}{\cosh(z)} = \frac{e^{2x} - 1}{e^{2x} + 1}$$



Useful when outcomes should be in (-1,1)

Not immune to Vanishing Gradient Problem

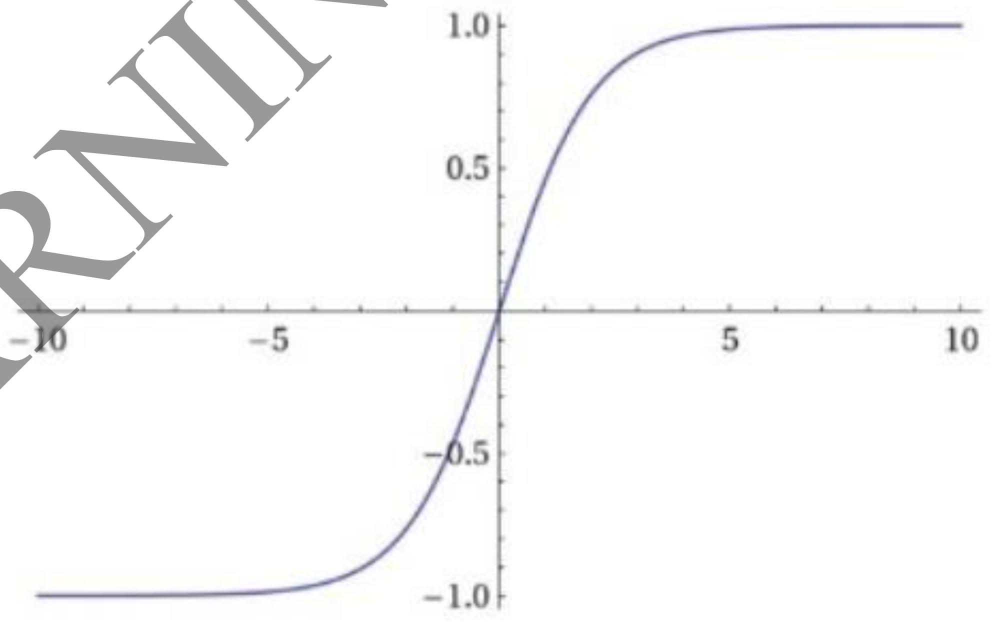
$$tanh(0) = 0$$
$$tanh(\infty) = 1$$
$$tanh(-\infty) = -1$$

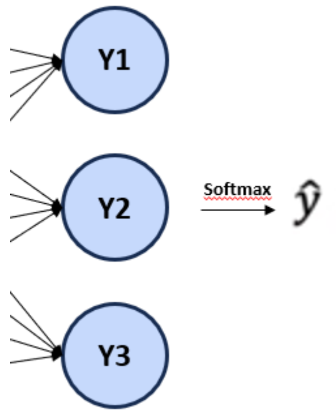
NEURAL NETWORKS – ACTIVATION FUNCTION

Softmax activation function

Softmax activation function is a popular choice for multi-class target problems.
The formula is as follows :
$$\frac{e^{z_i}}{\sum_{j=1}^{K} e^{z_j}}$$

- Probability of each class is calculated using the above formula
- Range of each probability is going to be in the range (0,1)
- The sum of all probabilities is going to be 1





$$\frac{e^{y1}}{(e^{y1}+e^{y2}+e^{y3})}$$

$$\frac{e^{y2}}{(e^{y1}+e^{y2}+e^{y3})}$$

$$\frac{e^{y3}}{(e^{y1}+e^{y2}+e^{y3})}$$

P(y1) = 0.45

P(y2) = 0.30

P(y3) = 0.25

Note : Values are assumed for representation purpose only

NEURAL NETWORKS – ACTIVATION FUNCTION

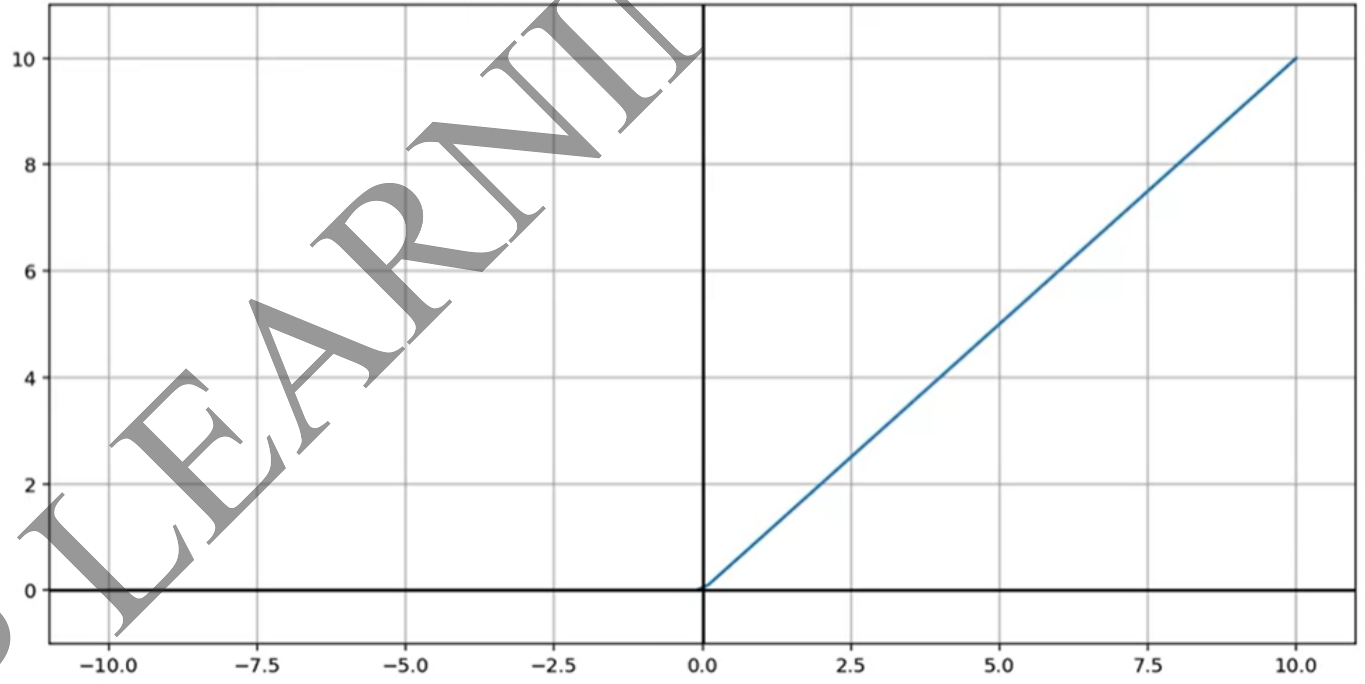
ReLU activation function

$$ReLU(z) = \begin{cases} 0, & z < 0 \\ z, & z \geq 0 \end{cases}$$

$$= \max(0, z)$$

Useful to capture large values but does not allow negative outcomes.

Immune to Vanishing Gradient Problem



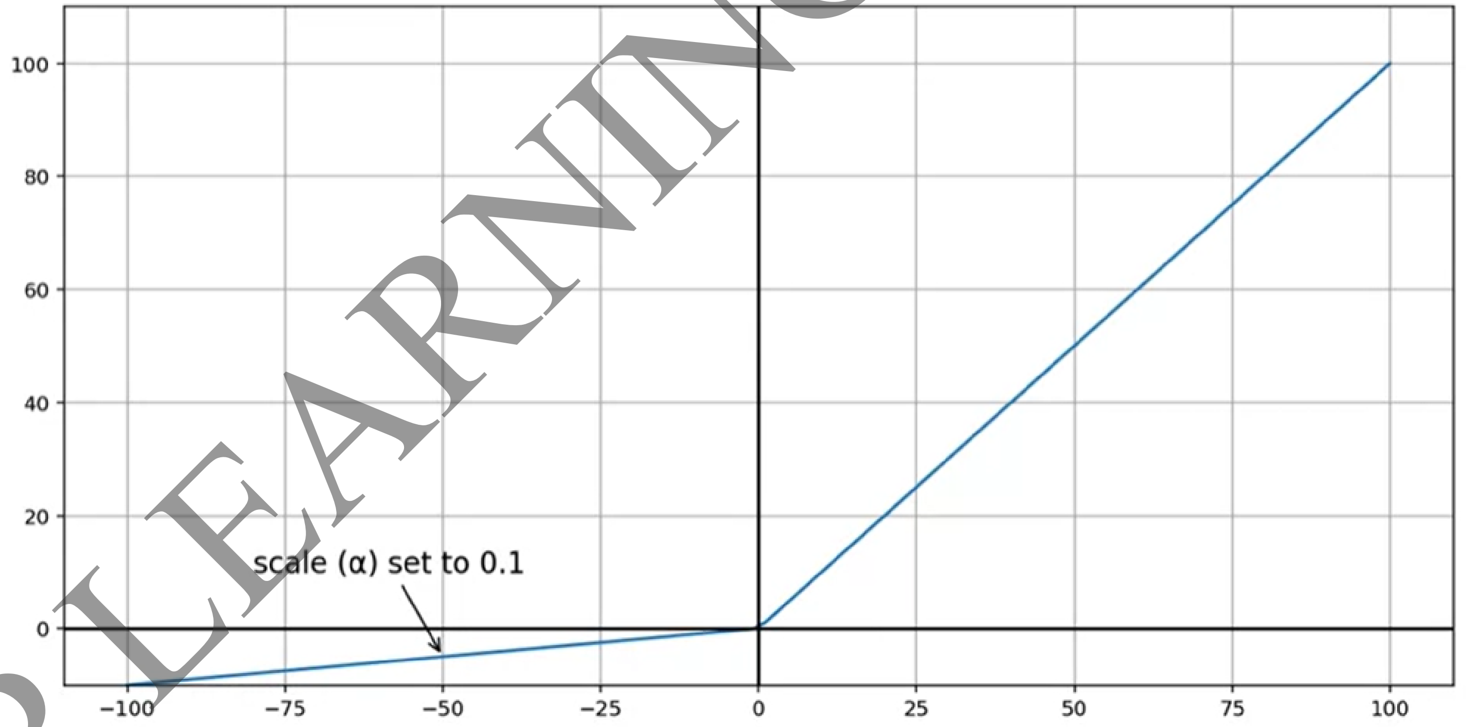
$$ReLU(0) = 0$$
$$ReLU(z) = z\ ; \quad \text{for } (z \gg 0)$$
$$ReLU(-z) = 0$$

NEURAL NETWORKS – ACTIVATION FUNCTION

Leaky ReLU activation function

$$LReLU(z) = \begin{cases} \alpha z, & z < 0 \\ z, & z \geq 0 \end{cases}$$

$$= \max(\alpha z, z)\,; \quad \text{for } (\alpha < 1)$$



Useful to capture large values but allows negative outcomes as well (unlike ReLU).

Immune to Vanishing Gradient Problem

$$LReLU(0) = 0$$
$$LReLU(z) = z\,; \quad \text{for } (z \gg 0)$$
$$LReLU(-z) = -\alpha z$$

NEURAL NETWORKS – ACTIVATION FUNCTION

Summarization Table

Target Type	Target Class	Activation Function	Range
Categorical	2 classes (Binary)	**Sigmoid**	(0,1)
Categorical	3 classes	**Hyperbolic Tangent (tan h)**	(-1,0,1)
Categorical	>=3 classes (Multi-class)	**SoftMax**	n - classes
Continuous	Finite +ve numbers	**ReLU**	(0, n) where -ve values are replaced by 0
Continuous	Both +ve and -ve numbers	**Leaky ReLU**	(-α*n, n) where α is less than 0

NEURAL NETWORKS – Regularization

“Regularization is any modification we make to a learning algorithm that is intended to reduce its generalization error but not its training error.” – Goodfellow et al. (2016)

Different Regularization Methods

- Regularization penalty in Cost function
- Dropout
- Early Stopping

NEURAL NETWORKS – Regularization

Regularization Penalty in Cost Function

This is similar to regularization in Ridge Regression for numeric variables

$$J = \frac{1}{2n}\sum_{i=1}^{n}(\hat{y}_i - y_i)^2 + \lambda\sum_{j=1}^{m}W_i^2$$

For continuous variables, we use **L1-L2** regularization.

For categorical variables, we use **categorical cross entropy**.

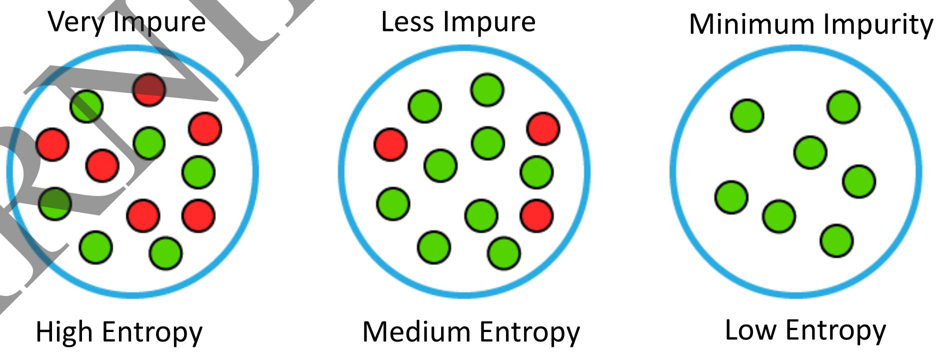
Let's Learn How To Build A DT using Entropy & Information Gain

Entropy :

Entropy is a measure of disorder or impurity in a node. i.e. measures homogeneity of a node

A **high value of Entropy** means that the **randomness** in the system is **high** and thus making accurate predictions is **tough**.

A **low value of Entropy** means that the **randomness** in the system is **low** and thus making accurate predictions is **easier**.



$$E = -\sum_{i=1}^{N} p_i log_2 p_i$$

N -> number of classes

Pi -> probability of the ith class

Suppose you have marbles of three colors; red, purple, and yellow
If we have **one red, three purple**, and **four yellow** observations in our set, our equation becomes:

$$E = -(p_r log_2 p_r + p_p log_2 p_p + p_y log_2 p_y)$$

where pr, pp and py are the probabilities of choosing a red, purple and yellow. So,
pr=1/8
pp=3/8
py=4/8
Our equation now becomes:
E=−((1/8)*log(1/8)+(3/8)*log(3/8)+(4/8)*log(4/8))
Our entropy would be: **0.97**

- PRITESH JHA

Let's Learn How To Build A DT using Entropy & Information Gain

What happens to Entropy when all observations belong to the same class say red?

$$E = -(p_r log_2 p_r + p_p log_2 p_p + p_y log_2 p_y)$$

$$E = -((8/8)log(8/8)+(0)log(0)+(0)log(0))$$

$$E = -(1 log_2 1) = 0$$

Such dataset has no impurity.
This implies that such a dataset would not be useful for learning.

What happens to Entropy if we have two classes, half made up of yellow and the other half being purple?

$$E = -(p_r log_2 p_r + p_p log_2 p_p + p_y log_2 p_y)$$

$$E = -((4/8)log(4/8)+(4/8)log(4/8)+(0)log(0))$$

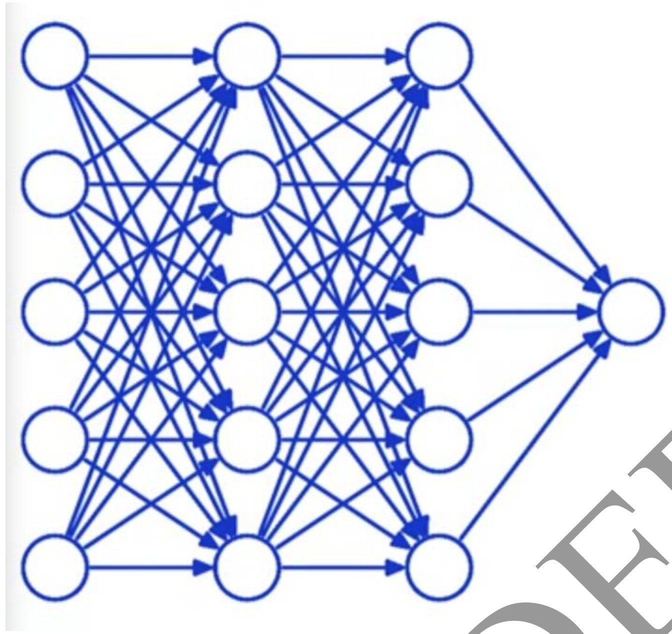
$$E = -((0.5 log_2 0.5) + (0.5 log_2 0.5)) = 1$$

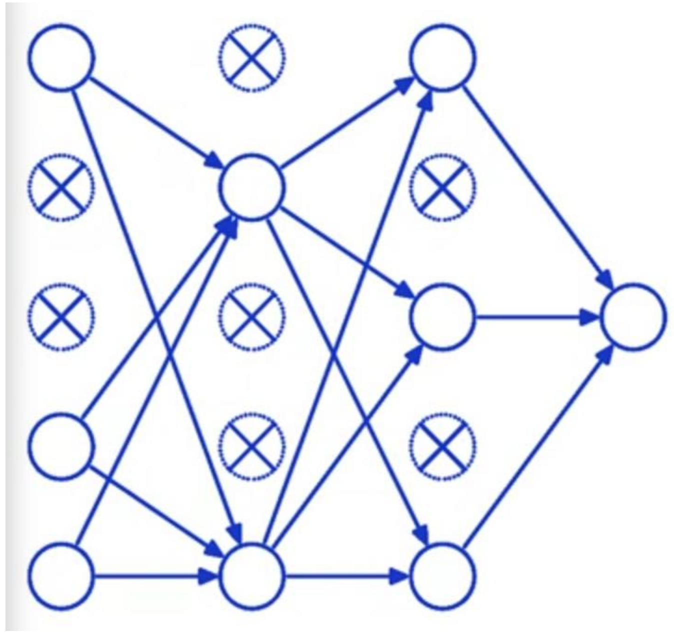
Such dataset has high impurity.
This implies that such a dataset is useful for learning.

NEURAL NETWORKS – Regularization

Dropout

Dropout mechanism is used to rescale the weights of neurons and dropping some nodes based on the activity status.
In simple words, we drop the nodes which do not create any meaningful output (or the output wont change even if they are removed).


Standard Neural Network

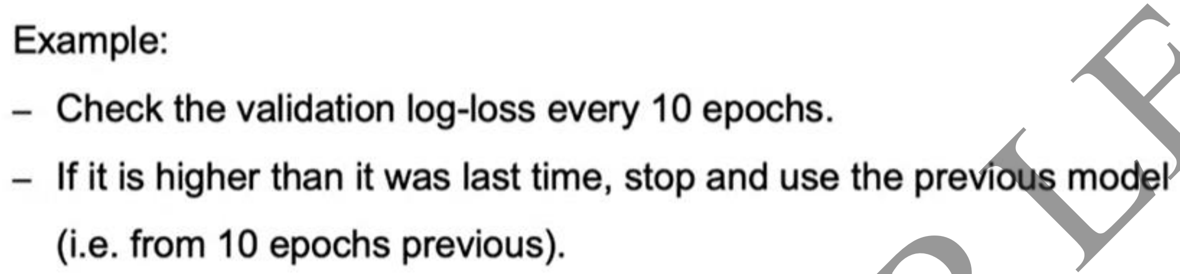

After applying Dropout

NEURAL NETWORKS – Regularization

Early Stopping

This refers to choosing some rules, after which the model stops training.
For example, we can set the early stopping criteria as Accuracy = 0.9 and this will be used at each iteration of the training network.

Once the criteria has been met, the model will stop training.


Example:
- Check the validation log-loss every 10 epochs.
- If it is higher than it was last time, stop and use the previous model (i.e. from 10 epochs previous).

NEURAL NETWORKS – Loss Funtions

Summarization Table

Target Type	**Target Class**	**Loss Function**
Categorical	Binary	**Binary Cross-Entropy**
Categorical	Multi-Class	**Categorical Cross-Entropy**
Continuous	Continuous (Both +ve and -ve)	**MSE / MAE**

END OF NEURAL NETWORKS

COMPUTER VISION

What is Computer Vision?

In the context of deep learning, computer vision is a field that focuses on enabling machines to <u>*interpret*</u> and understand the visual data in a manner <u>*similar to human vision*</u>.

It involves the development of deep learning models that can process, analyze, and make decisions based on visual data including images and videos.

Deep learning has revolutionized computer vision by providing models, particularly **Convolutional Neural Networks (CNNs)**, that are capable of achieving high levels of accuracy in these tasks, often surpassing human performance in specific domains.

CONVOLUTION NEURAL NETWORKS

Key Aspects of Computer Vision in Deep Learning

•**Image Classification**: Determining the category or class of an object in an image. For example, identifying whether a photograph contains a cat or a dog.

•**Object Detection**: Identifying objects within an image and determining their boundaries or locations, usually represented by bounding boxes.

•**Image Generation**: Generating new images that are similar to a given set of images, using techniques like Generative Adversarial Networks (GANs).

•**Facial Recognition**: Identifying or verifying a person's identity from a digital image or video frame.

CONVOLUTION NEURAL NETWORKS

Industrial Applications of Computer Vision using Deep Learning

- •**Automotive**: Autonomous driving systems use computer vision to understand the environment around the vehicle, including detecting pedestrians, cars, and road signs.
- •**Healthcare**: Medical image analysis, such as diagnosing diseases from X-rays, MRIs, or CT scans.
- •**Retail**: Automated checkout systems, customer behavior analysis, and inventory management through product recognition.
- •**Security**: Surveillance systems that can detect suspicious activities or perform facial recognition for identity verification.
- •**Agriculture**: Monitoring crop health and automating tasks like harvesting through drone or satellite imagery.

CONVOLUTION NEURAL NETWORKS

Before we study CNN, there are some concepts that are crucial for the CNN architectures.

- **Kernels**
- **Padding**
- **Stride**
- **Depth**
- **Pooling**

CONVOLUTION NEURAL NETWORKS

Kernels

A kernel is a grid of weights “overlaid” on image, centered on one pixel.

- Each weight is multiplied by pixel underneath it
- Output over the centered pixel is : $\sum_{p=1}^{P} W_p \cdot pixel_p$

Kernels are used with image data for the following tasks :
- Edge Detection
- Blur
- Sharpen

CONVOLUTION NEURAL NETWORKS

Kernels

Let us consider a greyscale image with the shape of (5x5)

Try to plot the matrix and find out the number it draws as an image

A Kernel of (3x3) is created, and overlaid on the image matrix

The dot product is calculated at each overlay and a new (3x3) matrix is created

-1	1	-1
-1	1	-1
-1	1	-1

255	255	10	255	255
255	10	10	255	255
255	255	10	255	255
255	255	10	255	255
255	10	10	10	255

Note : The kernel overlay will always start from top left of the image and eventually move towards right

CONVOLUTION NEURAL NETWORKS

Kernels

Let us consider a greyscale image with the shape of (5x5)

Try to plot the matrix and find out the number it draws as an image

A Kernel of (3x3) is created, and overlaid on the image matrix

The dot product is calculated at each overlay and a new (3x3) matrix is created

-1 255	1 255	-1 10	255	255
-1 255	1 10	-1 10	255	255
-1 255	1 255	-1 10	255	255
255	255	10	255	255
255	10	10	10	255

Output = (-1*255) + (1*255) + (-1*10) +
(-1*255) + (1*10) + (-1*10) +
(-1*255) + (1*255) + (-1*10)

= -255 + 255 – 10
-255 + 10 – 10
-255 + 255 – 10

= **-275**

Note : The kernel overlay will always start from top left of the image and eventually move towards right

CONVOLUTION NEURAL NETWORKS

Kernels

Let us consider a greyscale image with the shape of (5x5)

Try to plot the matrix and find out the number it draws as an image

A Kernel of (3x3) is created, and overlaid on the image matrix

The dot product is calculated at each overlay and a new (3x3) matrix is created

Output = (-1*255) + (1*10) + (-1*255) +
(-1*10) + (1*10) + (-1*255) +
(-1*255) + (1*10) + (-1*255)

= -255 + 10 – 255
-10 + 10 – 255
-255 + 10 – 255

= **-1255**

255	-1 255	1 10	-1 255	255
255	-1 10	1 10	-1 255	255
255	-1 255	1 10	-1 255	255
255	255	10	255	255
255	10	10	10	255

Note : The kernel overlay will always start from top left of the image and eventually move towards right

CONVOLUTION NEURAL NETWORKS

Kernels

Let us consider a greyscale image with the shape of (5x5)

Try to plot the matrix and find out the number it draws as an image

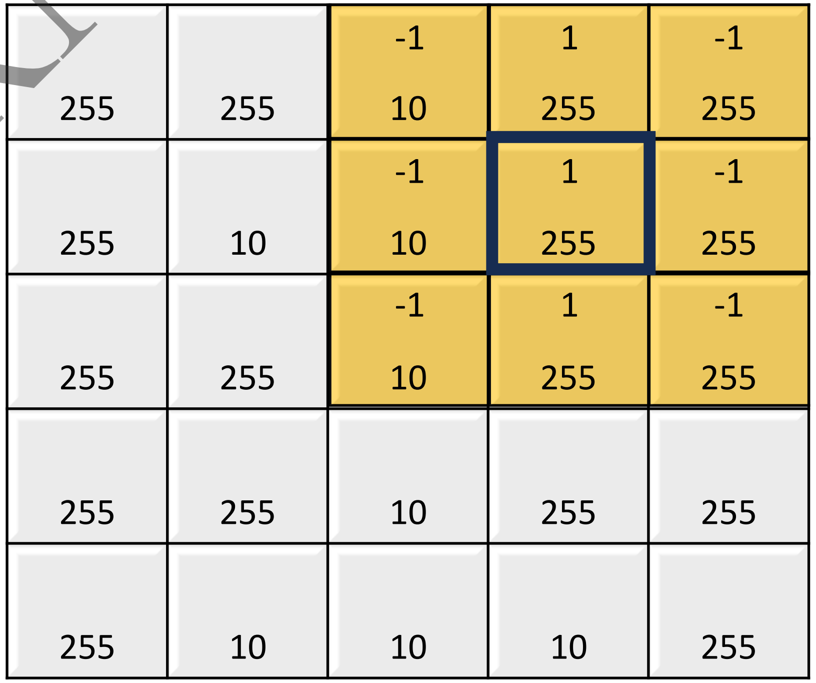
A Kernel of (3x3) is created, and overlaid on the image matrix

The dot product is calculated at each overlay and a new (3x3) matrix is created

Output = (-1*10) + (1*255) + (-1*255) +
(-1*10) + (1*255) + (-1*255) +
(-1*10) + (1*255) + (-1*255)

= -10 + 255 – 255
-10 + 255 – 255
-10 + 255 – 255

= **-30**



255	255	-1 10	1 255	-1 255
255	10	-1 10	1 255	-1 255
255	255	-1 10	1 255	-1 255
255	255	10	255	255
255	10	10	10	255

Note : The kernel overlay will always start from top left of the image and eventually move towards right

CONVOLUTION NEURAL NETWORKS

Kernels

Let us consider a greyscale image with the shape of (5x5)

Try to plot the matrix and find out the number it draws as an image

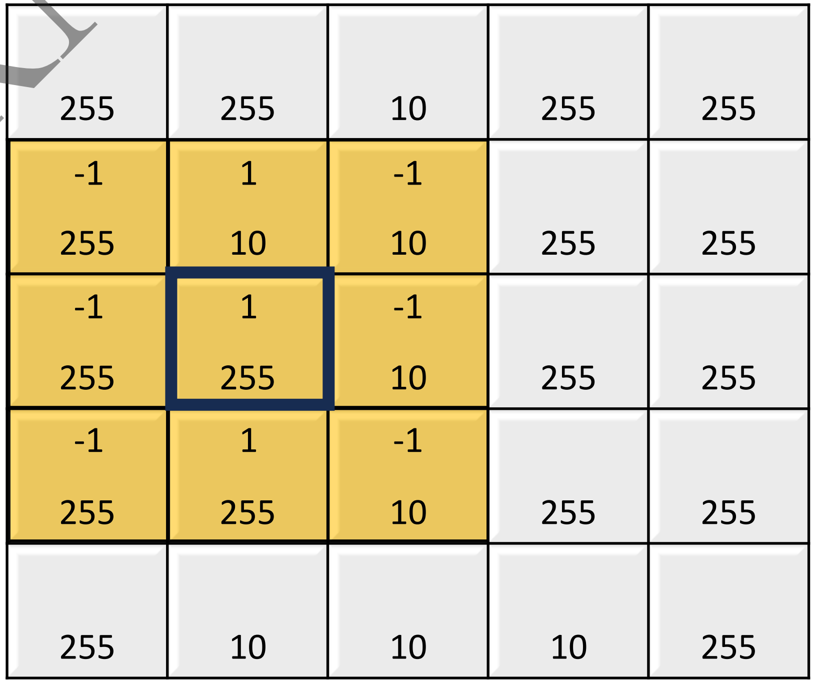
A Kernel of (3x3) is created, and overlaid on the image matrix

The dot product is calculated at each overlay and a new (3x3) matrix is created

Output = (-1*255) + (1*10) + (-1*10) +
(-1*255) + (1*255) + (-1*10) +
(-1*255) + (1*255) + (-1*10)

= -255 + 10 – 10
-255 + 255 – 10
-255 + 255 – 10

= **-275**



255	255	10	255	255
-1 255	1 10	-1 10	255	255
-1 255	1 255	-1 10	255	255
-1 255	1 255	-1 10	255	255
255	10	10	10	255

Note : The kernel overlay will always start from top left of the image and eventually move towards right

CONVOLUTION NEURAL NETWORKS

Kernels

Let us consider a greyscale image with the shape of (5x5)

Try to plot the matrix and find out the number it draws as an image

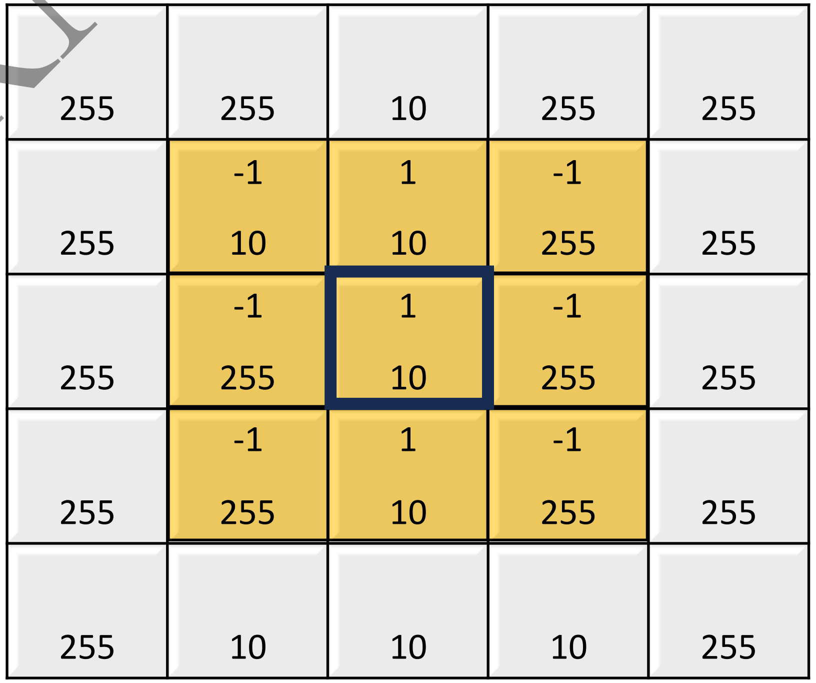
A Kernel of (3x3) is created, and overlaid on the image matrix

The dot product is calculated at each overlay and a new (3x3) matrix is created

Output = (-1*10) + (1*10) + (-1*255) +
(-1*255) + (1*10) + (-1*255) +
(-1*255) + (1*10) + (-1*255)

= -10 + 10 – 255
-255 + 10 – 255
-255 + 10 – 255

= **-1255**



255	255	10	255	255
255	-1 10	1 10	-1 255	255
255	-1 255	1 10	-1 255	255
255	-1 255	1 10	-1 255	255
255	10	10	10	255

Note : The kernel overlay will always start from top left of the image and eventually move towards right

CONVOLUTION NEURAL NETWORKS

Kernels

Let us consider a greyscale image with the shape of (5x5)

Try to plot the matrix and find out the number it draws as an image

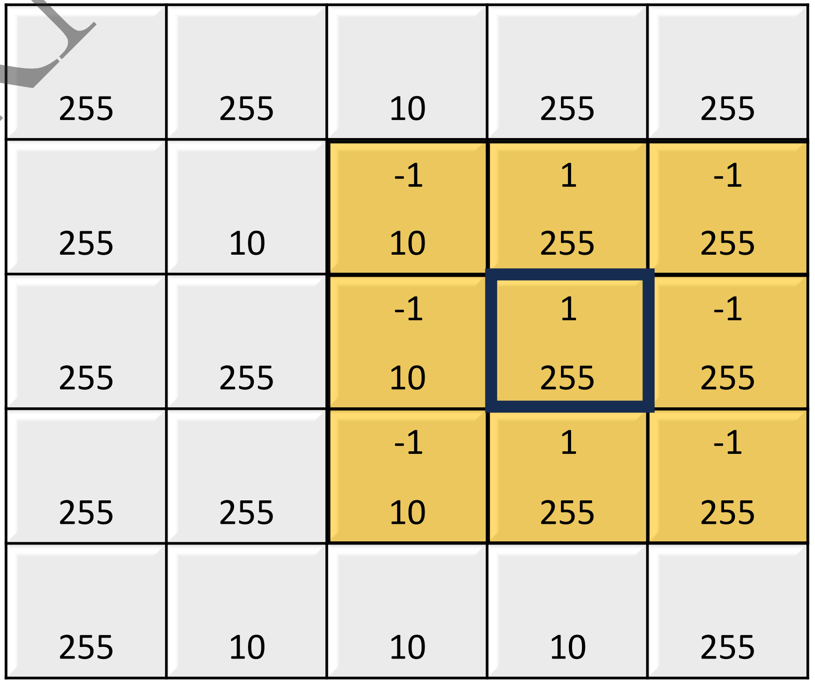
A Kernel of (3x3) is created, and overlaid on the image matrix

The dot product is calculated at each overlay and a new (3x3) matrix is created

Output = (-1*10) + (1*255) + (-1*255) +
(-1*10) + (1*255) + (-1*255) +
(-1*10) + (1*255) + (-1*255)

= -10 + 255 – 255
-10 + 255 – 255
-10 + 255 – 255

= **-30**



255	255	10	255	255
255	10	-1 10	1 255	-1 255
255	255	-1 10	1 255	-1 255
255	255	-1 10	1 255	-1 255
255	10	10	10	255

Note : The kernel overlay will always start from top left of the image and eventually move towards right

CONVOLUTION NEURAL NETWORKS

Kernels

Let us consider a greyscale image with the shape of (5x5)

Try to plot the matrix and find out the number it draws as an image

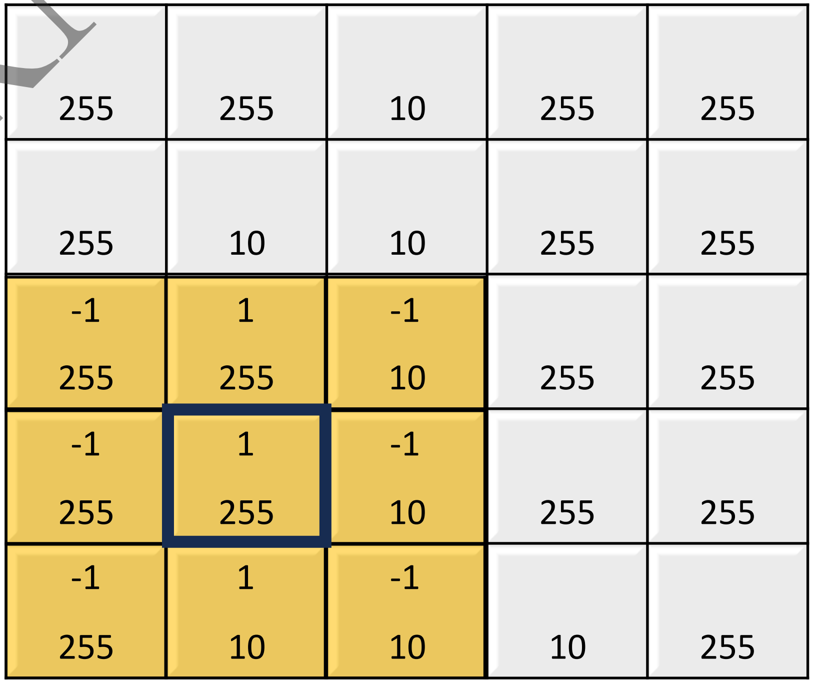
A Kernel of (3x3) is created, and overlaid on the image matrix

The dot product is calculated at each overlay and a new (3x3) matrix is created

Output = (-1*255) + (1*255) + (-1*10) +
(-1*255) + (1*255) + (-1*10) +
(-1*255) + (1*10) + (-1*10)

= -255 + 255 – 10
-255 + 255 – 10
-255 + 10 – 10

= **-275**



255	255	10	255	255
255	10	10	255	255
-1 255	1 255	-1 10	255	255
-1 255	1 255	-1 10	255	255
-1 255	1 10	-1 10	10	255

Note : The kernel overlay will always start from top left of the image and eventually move towards right

CONVOLUTION NEURAL NETWORKS

Kernels

Let us consider a greyscale image with the shape of (5x5)

Try to plot the matrix and find out the number it draws as an image

A Kernel of (3x3) is created, and overlaid on the image matrix

The dot product is calculated at each overlay and a new (3x3) matrix is created

255	255	10	255	255
255	10	10	255	255
255	-1 255	1 10	-1 255	255
255	-1 255	1 10	-1 255	255
255	-1 10	1 10	-1 10	255

Output = (-1*255) + (1*10) + (-1*255) +
(-1*255) + (1*10) + (-1*255) +
(-1*10) + (1*10) + (-1*10)

= -255 + 10 – 255
-255 + 10 – 255
-10 + 10 – 10

= **-1010**

Note : The kernel overlay will always start from top left of the image and eventually move towards right

CONVOLUTION NEURAL NETWORKS

Kernels

Let us consider a greyscale image with the shape of (5x5)

Try to plot the matrix and find out the number it draws as an image

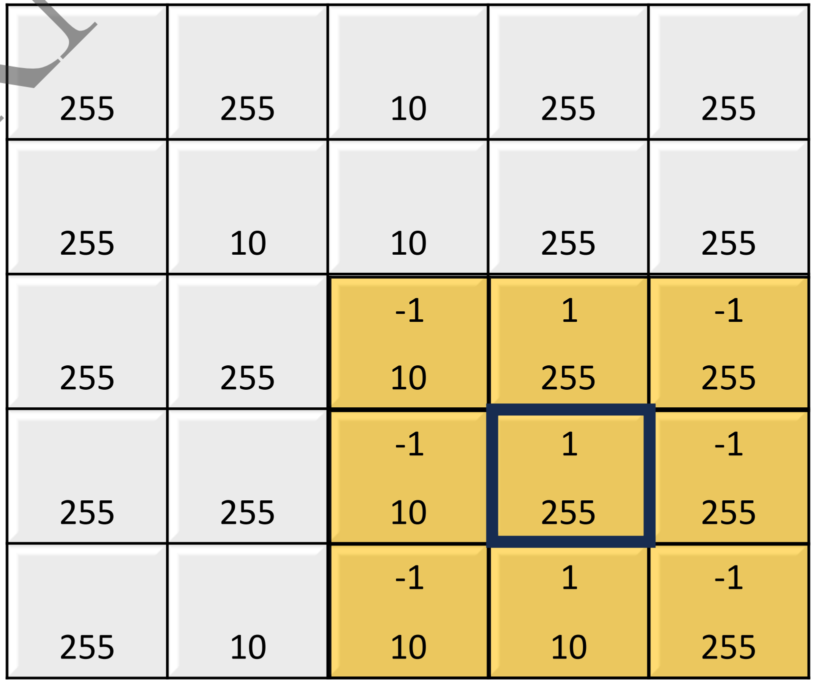
A Kernel of (3x3) is created, and overlaid on the image matrix

The dot product is calculated at each overlay and a new (3x3) matrix is created

Output = (-1*10) + (1*255) + (-1*255) +
(-1*10) + (1*255) + (-1*255) +
(-1*10) + (1*10) + (-1*255)

= -10 + 255 – 255
-10 + 255 – 255
-10 + 10 – 255

= **-275**



255	255	10	255	255
255	10	10	255	255
255	255	-1 10	1 255	-1 255
255	255	-1 10	1 255	-1 255
255	10	-1 10	1 10	-1 255

Note : The kernel overlay will always start from top left of the image and eventually move towards right

CONVOLUTION NEURAL NETWORKS

Kernels

255	255	10	255	255
255	10	10	255	255
255	255	10	255	255
255	255	10	255	255
255	10	10	10	255

Image Matrix

dot product

-1	1	-1
-1	1	-1
-1	1	-1

Vertical Line Detector

=

-275	-1225	-30
-275	-1225	-30
-275	-1010	-275

Convolution Output

CONVOLUTION NEURAL NETWORKS

Some other common kernels are as follows

-1	1	-1
-1	1	-1
-1	1	-1

Vertical Line Detector

-1	-1	-1
1	1	1
-1	-1	-1

Horizontal Line Detector

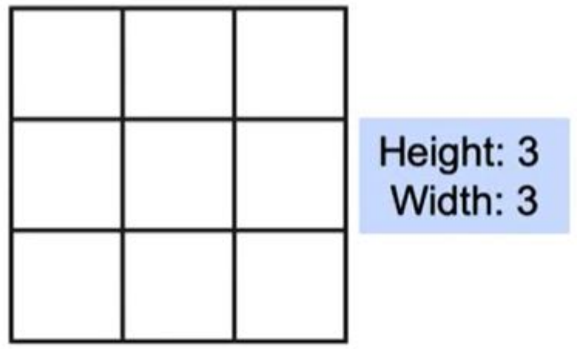
-1	-1	-1
-1	1	1
-1	1	1

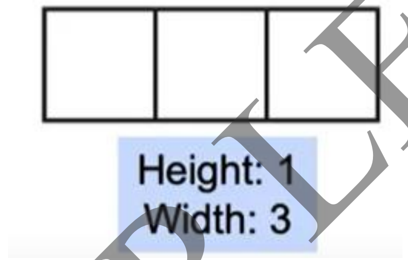
Corner Detector

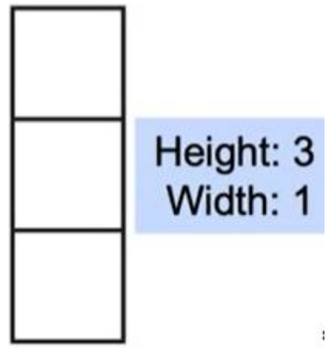
CONVOLUTION NEURAL NETWORKS

Grid Size

- Typically use odd numbers like 3x3 or 5x5 to have a “centre” of the kernel
- Kernel does not have to be a square
- Using kernels directly (like our previous example) will lead to “edge effect”.
  Pixels at the edge of the image will never be able to overlay with the centre of the kernel



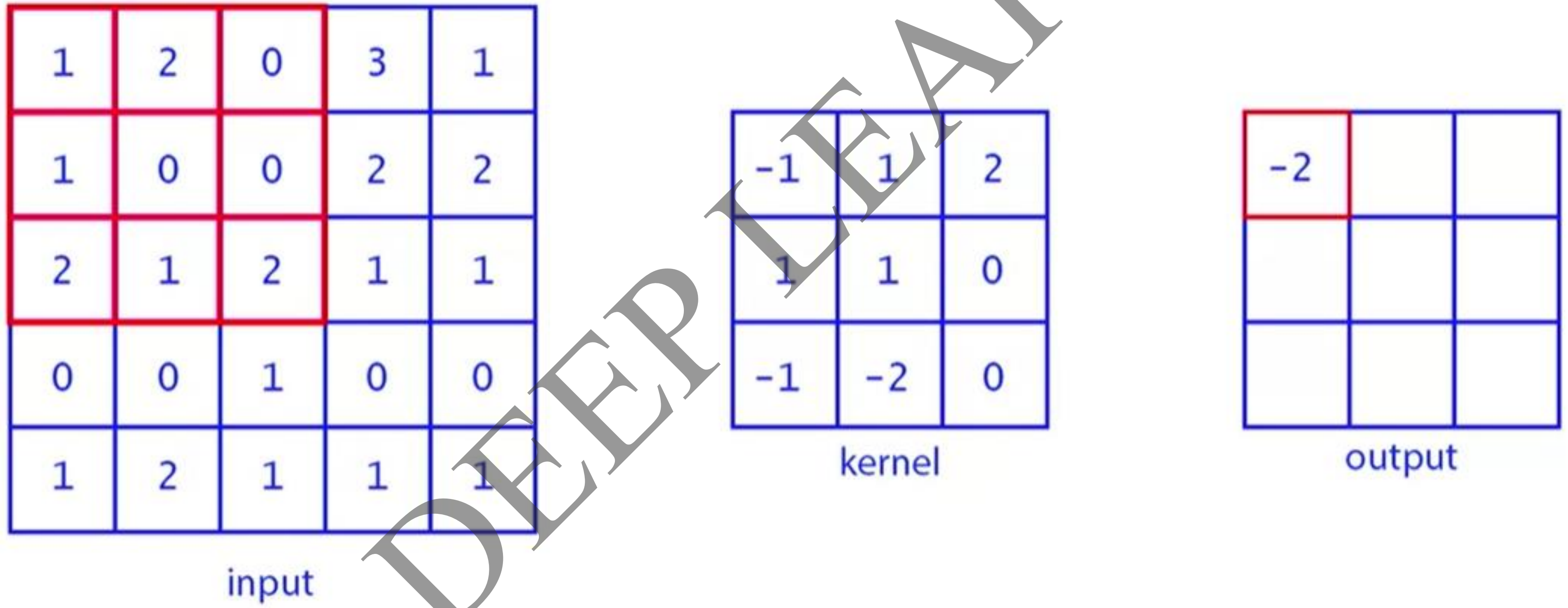




CONVOLUTION NEURAL NETWORKS

Padding

- Padding adds extra pixels around the frame, so that the pixels from the edges of the frame can become centre picels as the kernel moves around the image.
- Added pixels are typically value zero; also called as “zero-padding”.

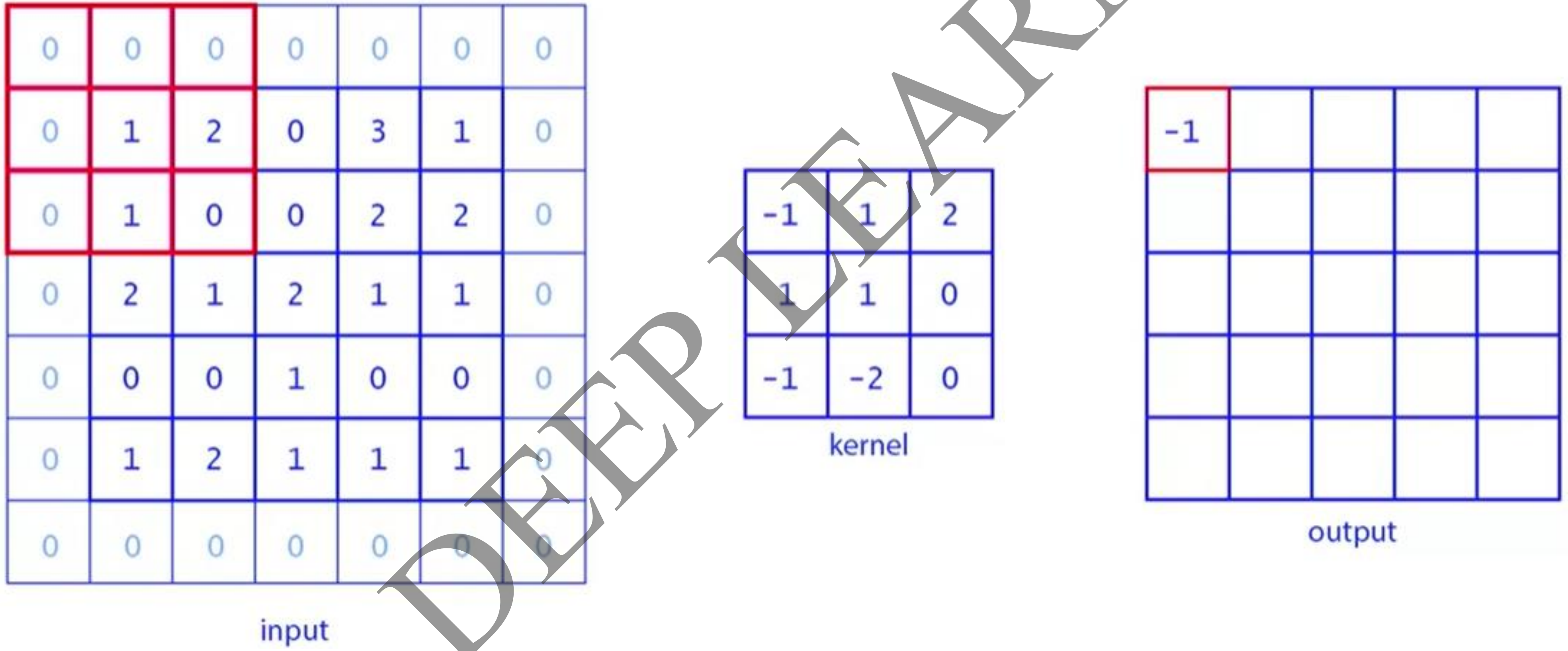


Without Padding

CONVOLUTION NEURAL NETWORKS

Padding

- Padding adds extra pixels around the frame, so that the pixels from the edges of the frame can become centre picels as the kernel moves around the image.
- Added pixels are typically value zero; also called as “zero-padding”.



With Padding

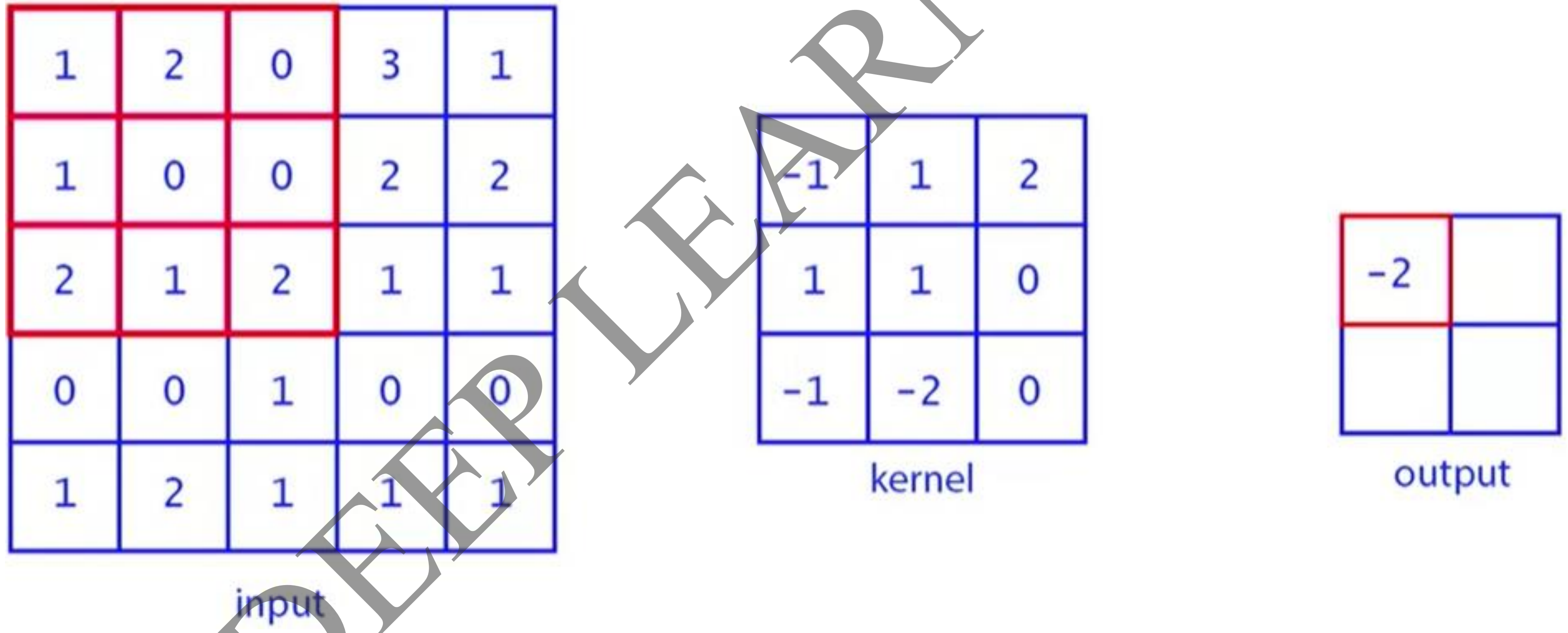
CONVOLUTION NEURAL NETWORKS

Stride

- The “step size” as the kernel moves across image
- Usually stride = 1 for most cases
- Can be different vertically and horizontally, but usually kept as same
- When the stride is greater than 1, it scales down the output dimension

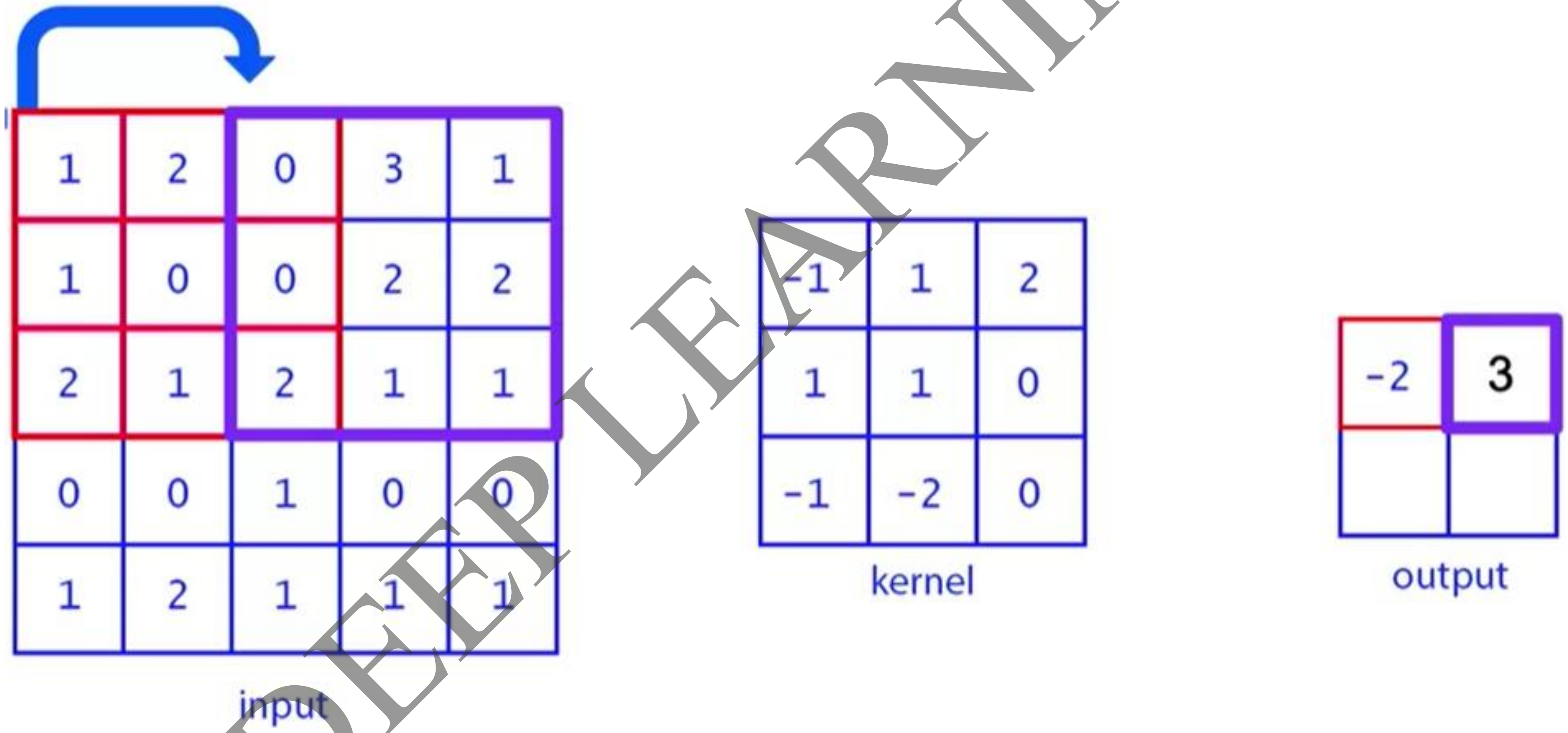
CONVOLUTION NEURAL NETWORKS

Stride Without Padding



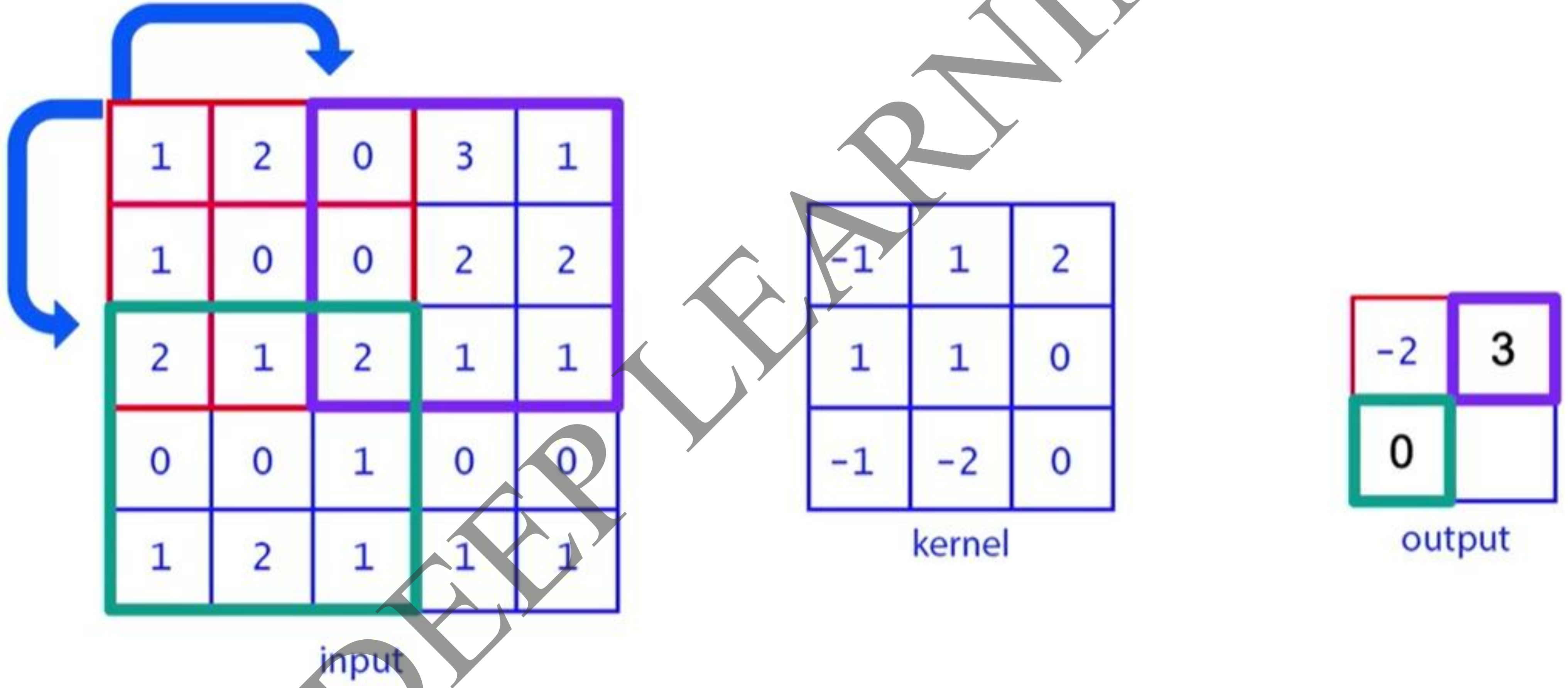
CONVOLUTION NEURAL NETWORKS

Stride Without Padding



CONVOLUTION NEURAL NETWORKS

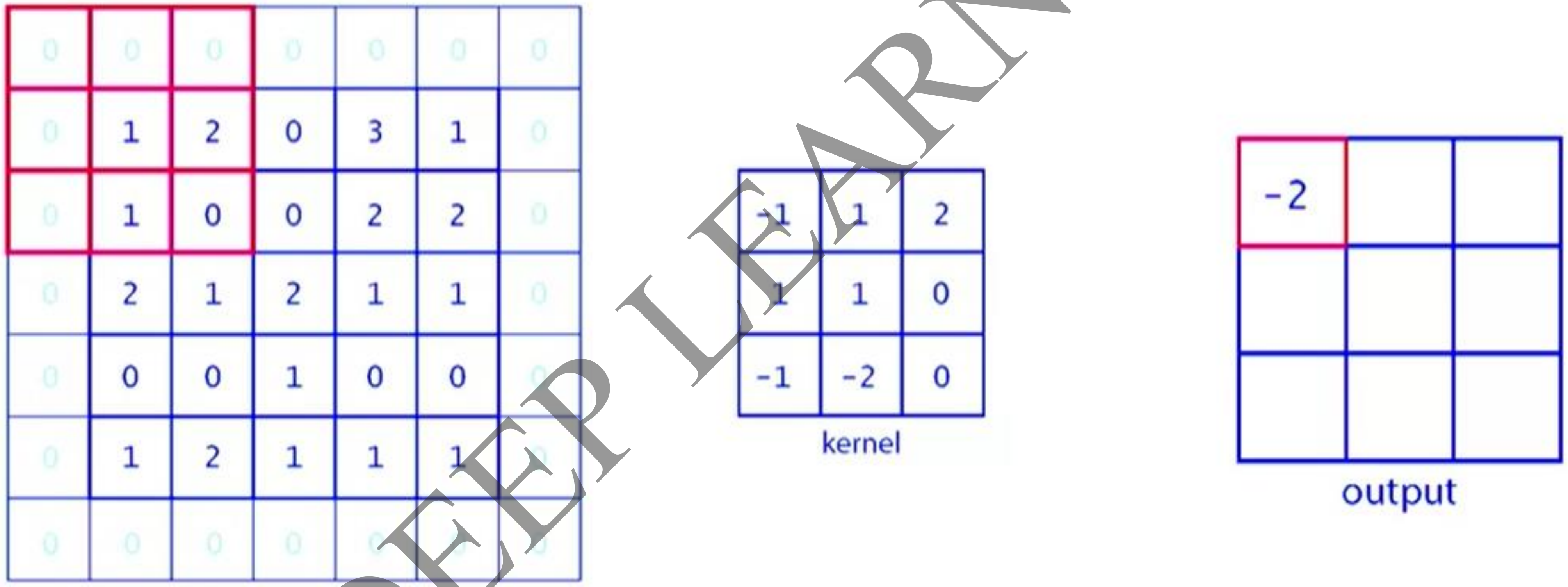
Stride Without Padding



- PRITESH JHA

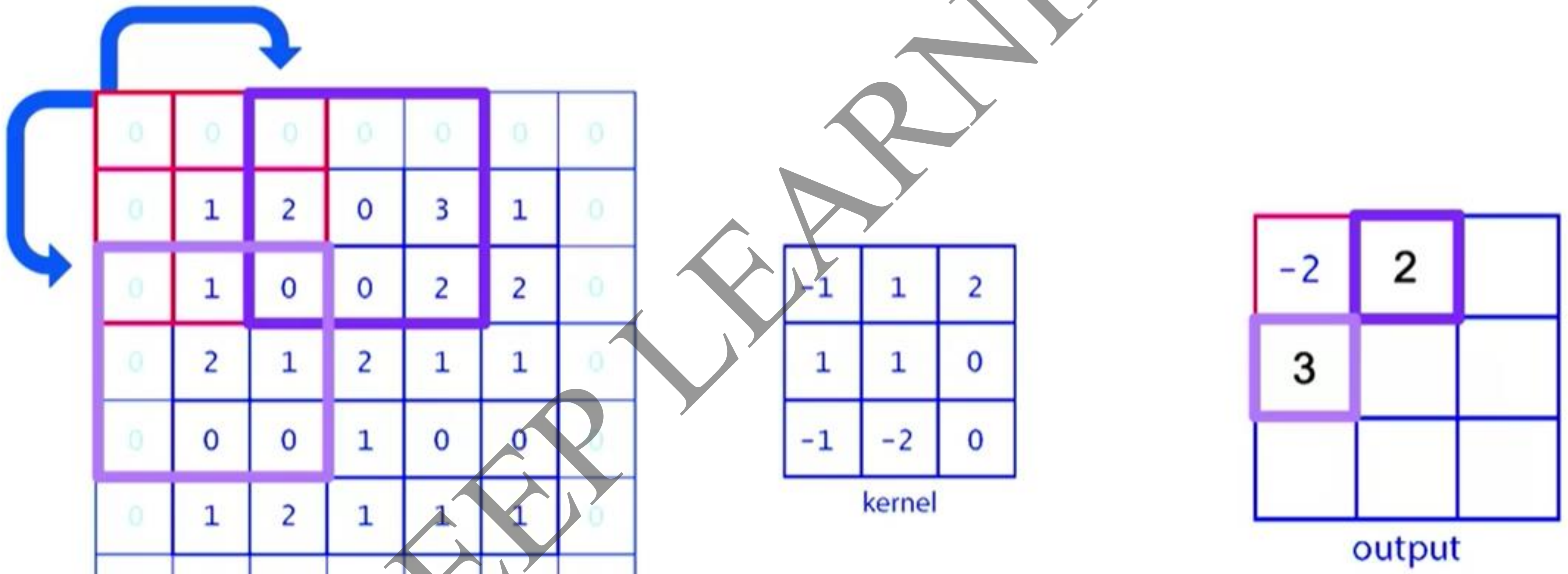
CONVOLUTION NEURAL NETWORKS

Stride with Padding



CONVOLUTION NEURAL NETWORKS

Stride with Padding



CONVOLUTION NEURAL NETWORKS

Depth

In images, we often have multiple values at “each” pixel location. These different numbers referred as “**channels**”.

RGB – 3 channels (Red Green Blue)
CMYK – 4 channels (Cyan Magenta Yellow Black)

The number of channels in the image are called as “**Depth”.**

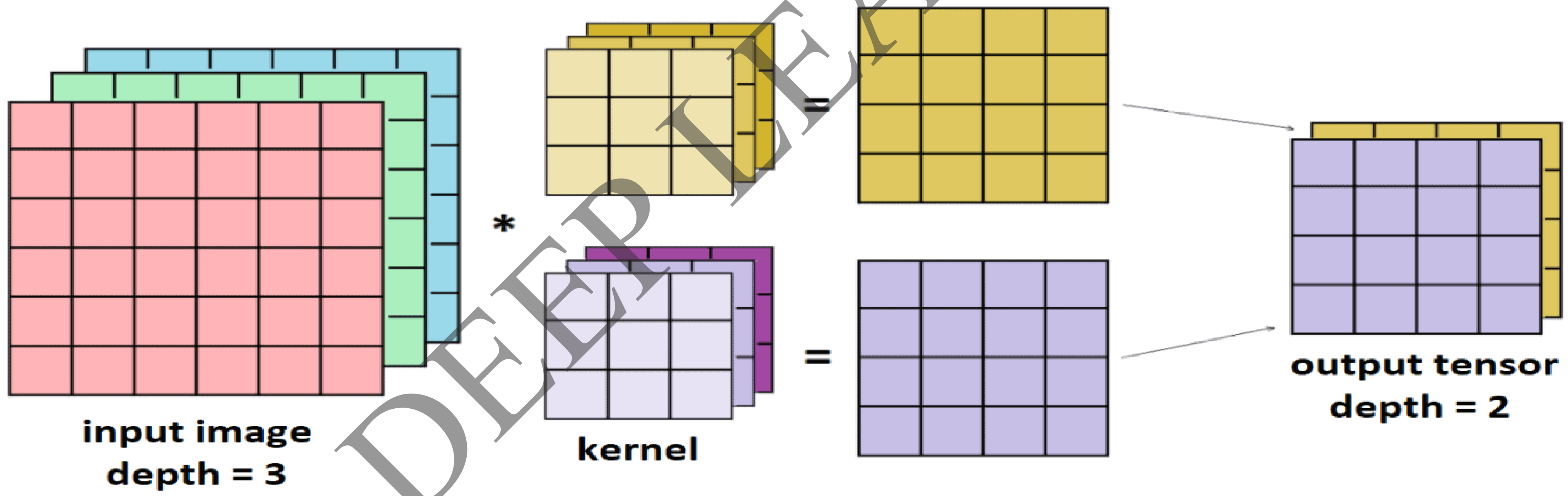
For example,
A 5x5 kernel on RGB image will have 5 x 5 x 3 = 75 weights

CONVOLUTION NEURAL NETWORKS

Depth

The output layer also has a depth.

- The networks typically train on multiple kernels
- Each kernel gives a single number output at each pixel location
- So, if there are 2 kernels used in a layer, the output will have depth = 2



CONVOLUTION NEURAL NETWORKS

Pooling

Idea : Reduce the image size by mapping a patch of pixels to a single value.
Basically, it reduces the dimensions of an image.

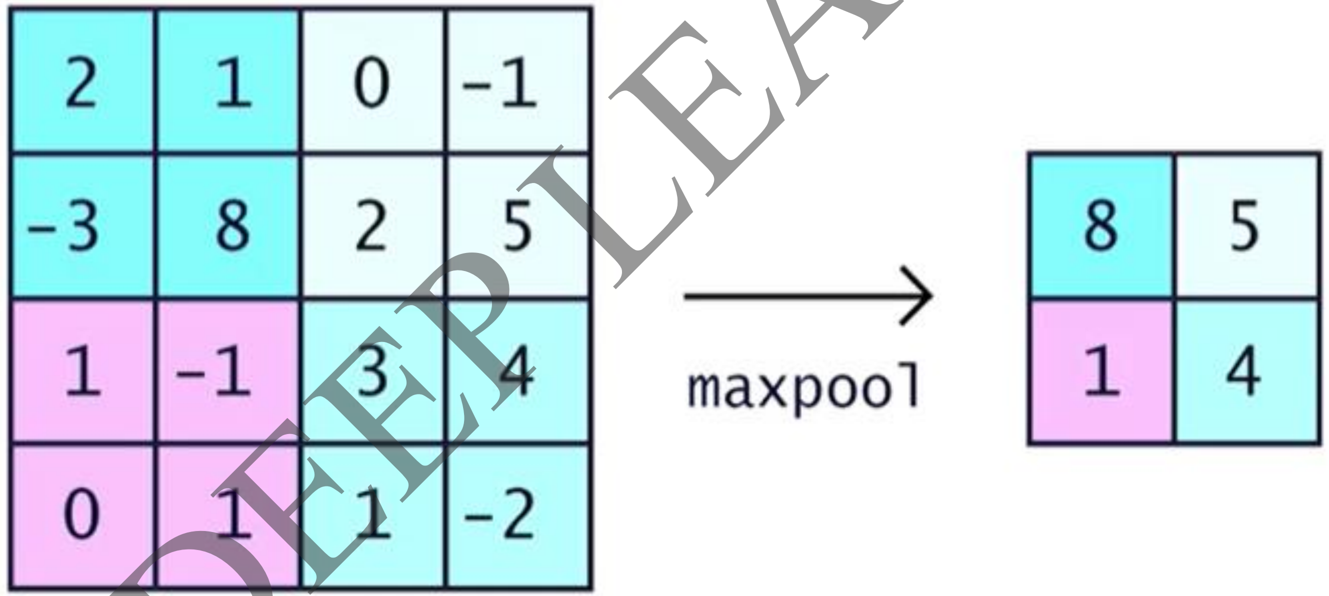
There are different types of pooling :

- Max Pooling (most common)
- Average Pooling

CONVOLUTION NEURAL NETWORKS

Max Pooling

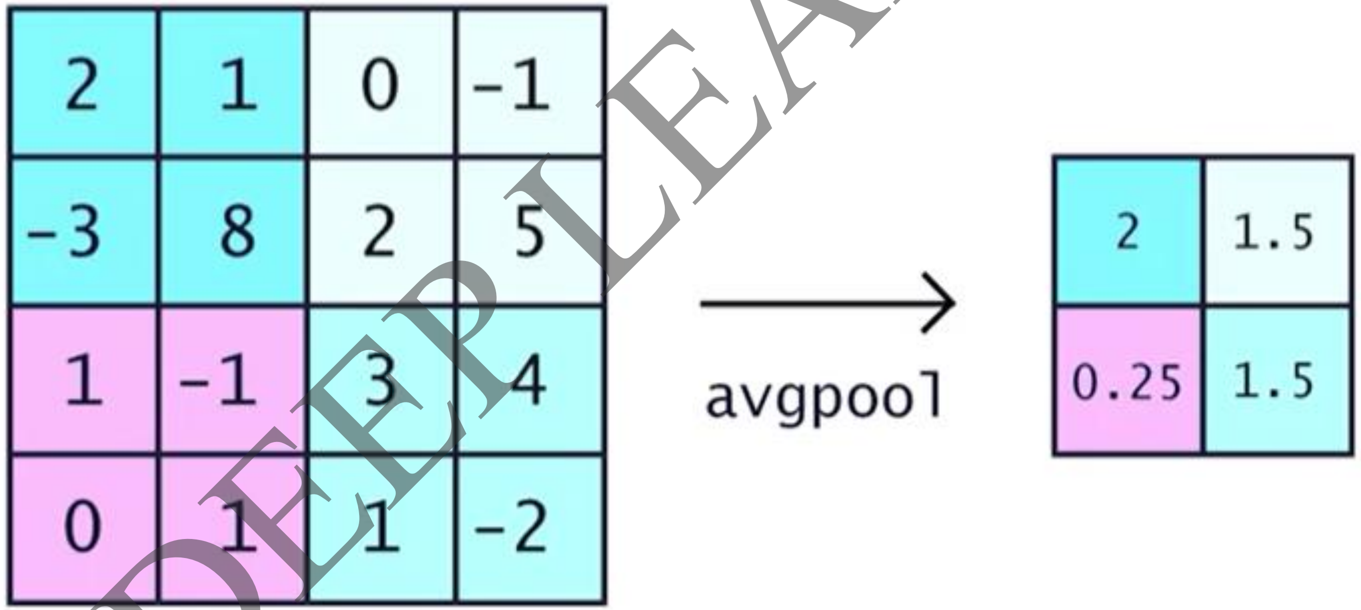
- An Image matrix is represented by multiple patches clubbed together
- The **max** of each patch is taken, and a new image matrix is formed



CONVOLUTION NEURAL NETWORKS

Average Pooling

- An Image matrix is represented by multiple patches clubbed together
- The **average** of each patch is taken, and a new image matrix is formed



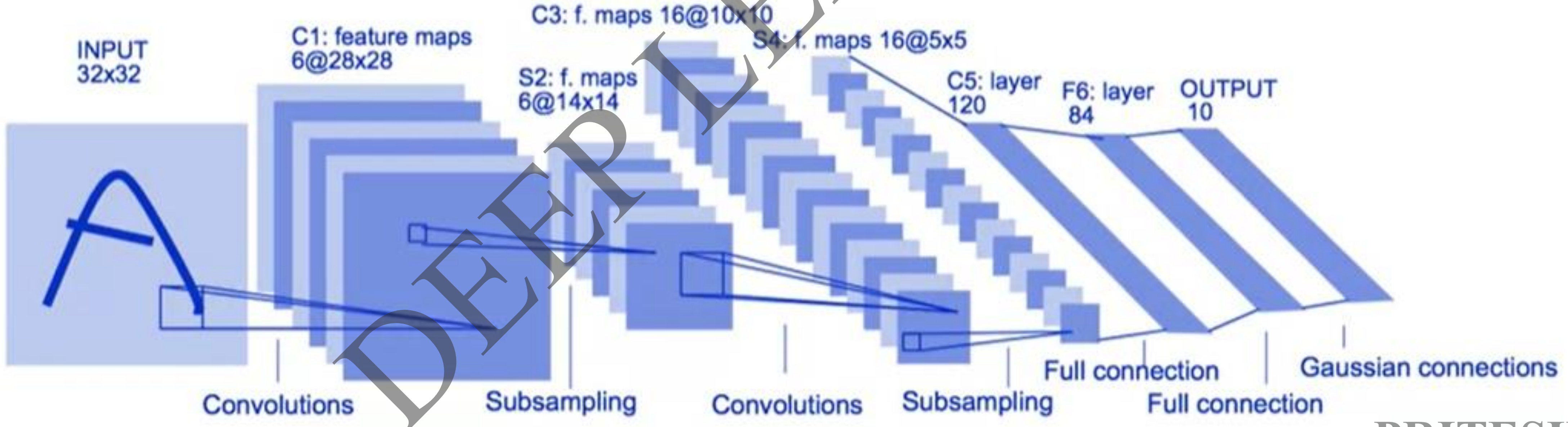
CONVOLUTION NEURAL NETWORKS - ARCHITECTURES

LeNet-5 Architecture

Created by Yann LeCun in the 1990s

Used on the MNIST data set.

Novel Idea: Use convolutions to efficiently learn features on data set.



CONVOLUTION NEURAL NETWORKS - ARCHITECTURES

Features of LeNet-5

•Every convolutional layer includes three parts: convolution, pooling, and nonlinear activation functions

•Using convolution to extract spatial features (Convolution was called receptive fields originally)

•**The average pooling layer** is used for subsampling.

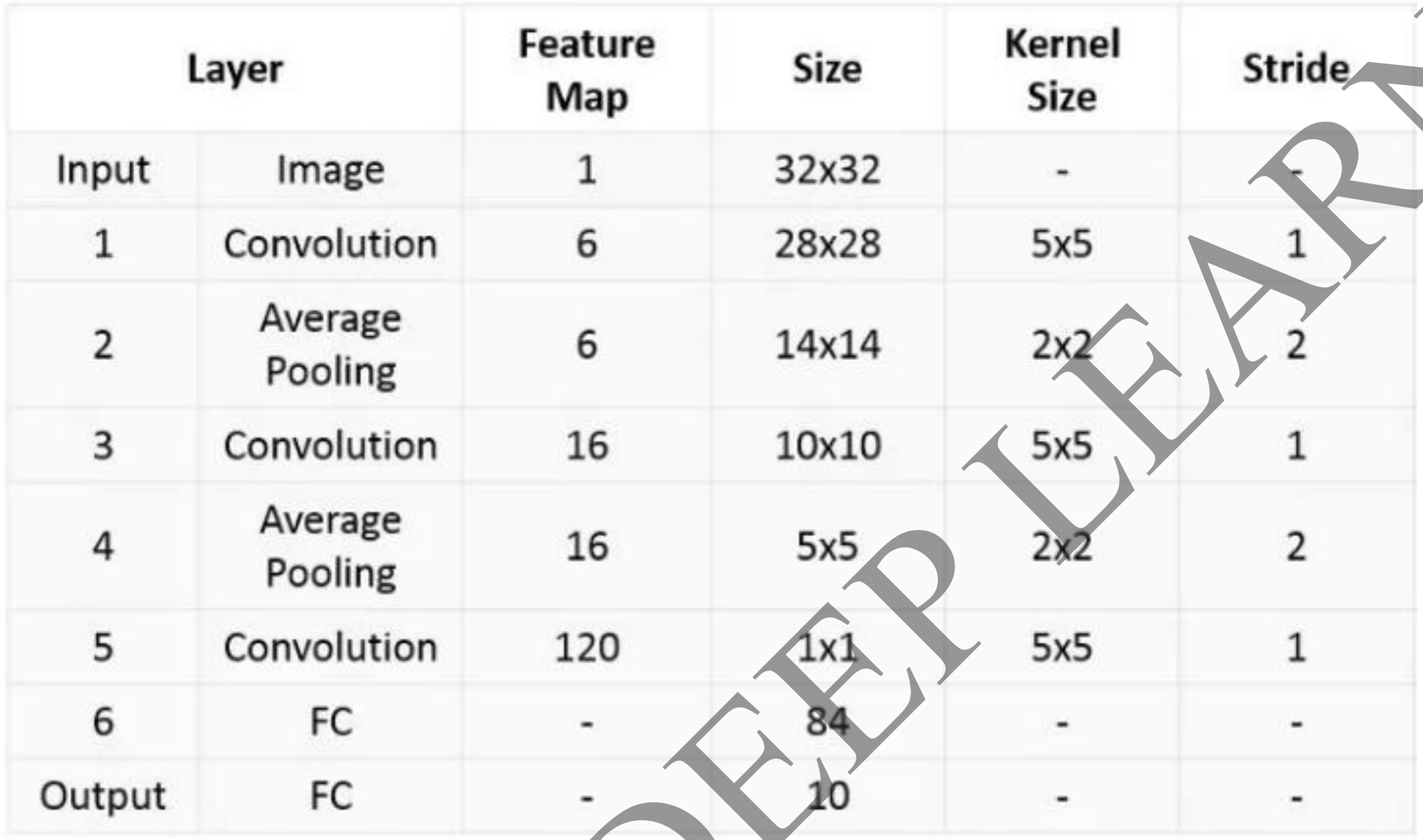
•'**tanh**' was originally used used as the activation function
Post 2012, **'ReLU'** has become the top choice.

•Using **Multi-Layered Perceptron** or **Fully Connected Layers** as the last classifier

•The sparse connection between layers reduces the complexity of computation

CONVOLUTION NEURAL NETWORKS - ARCHITECTURES

Summary of LeNet-5 Architecture

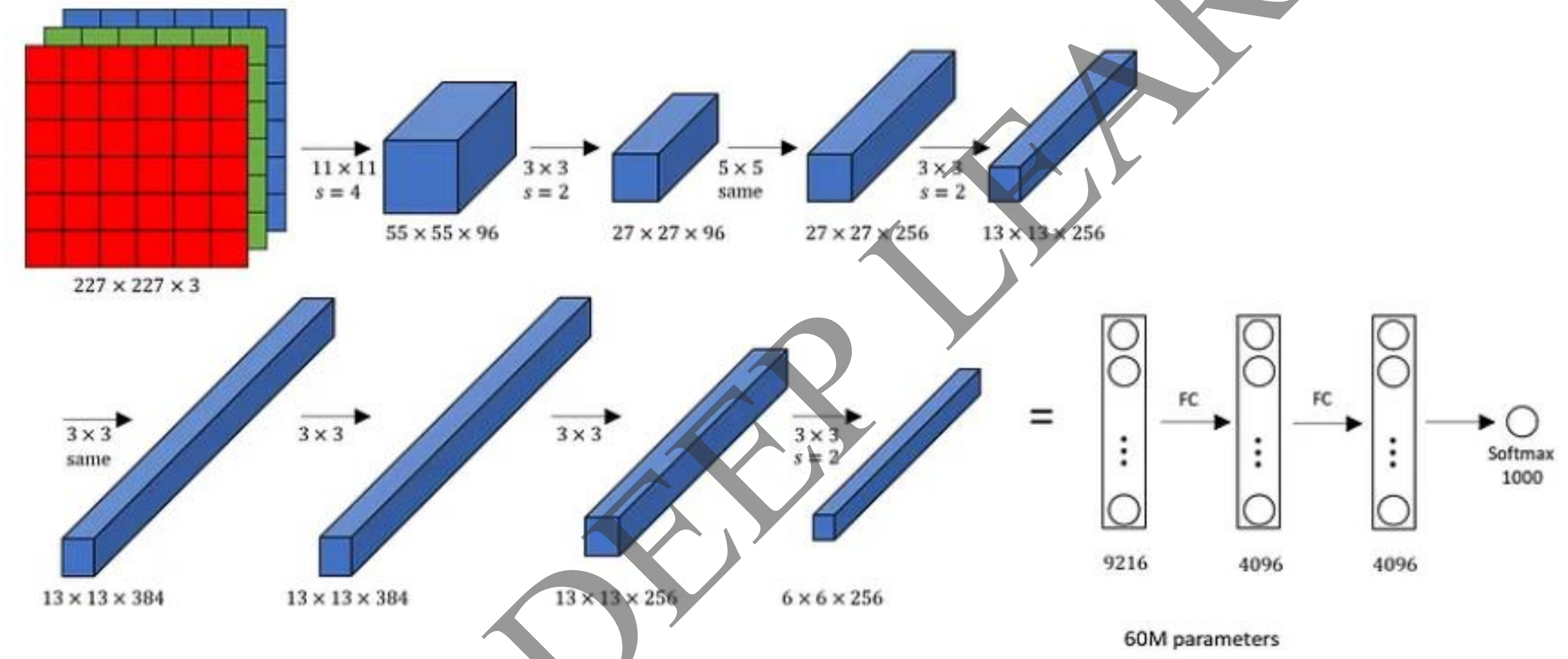


Layer		Feature Map	Size	Kernel Size	Stride
Input	Image	1	32x32	-	-
1	Convolution	6	28x28	5x5	1
2	Average Pooling	6	14x14	2x2	2
3	Convolution	16	10x10	5x5	1
4	Average Pooling	16	5x5	2x2	2
5	Convolution	120	1x1	5x5	1
6	FC	-	84	-	-
Output	FC	-	10	-	-

CONVOLUTION NEURAL NETWORKS - ARCHITECTURES

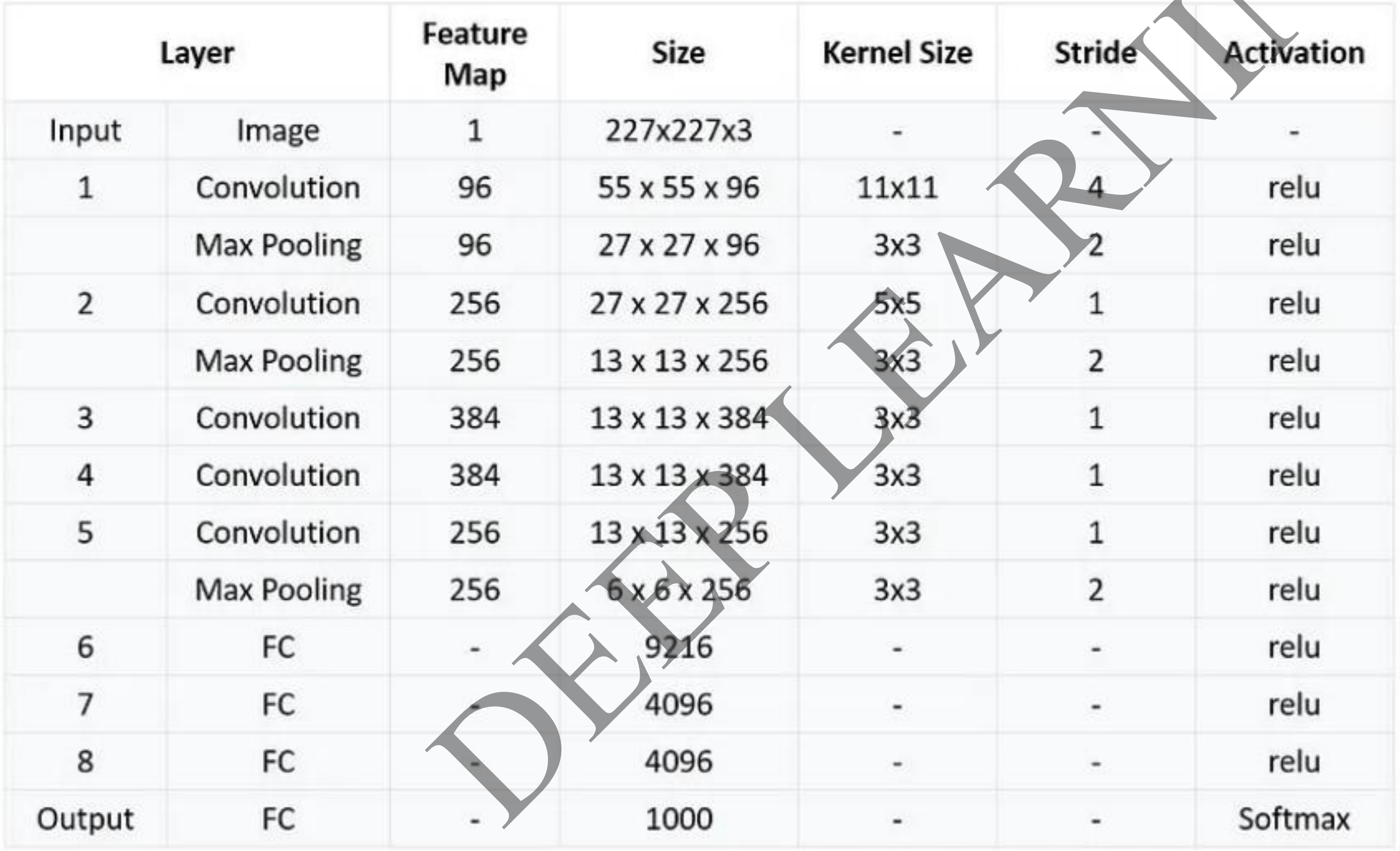
AlexNet Architecture

AlexNet is a groundbreaking CNN architecture that was developed by Alex Krizhevsky, Ilya Sutskever, and Geoffrey Hinton. It significantly outperformed other models in the ImageNet Large Scale Visual Recognition Challenge **(ILSVRC)** in 2012. AlexNet is larger and deeper than LeNet-5 and was the first to use ReLU (Rectified Linear Units) for the non-linear parts.



CONVOLUTION NEURAL NETWORKS - ARCHITECTURES

AlexNet Architecture

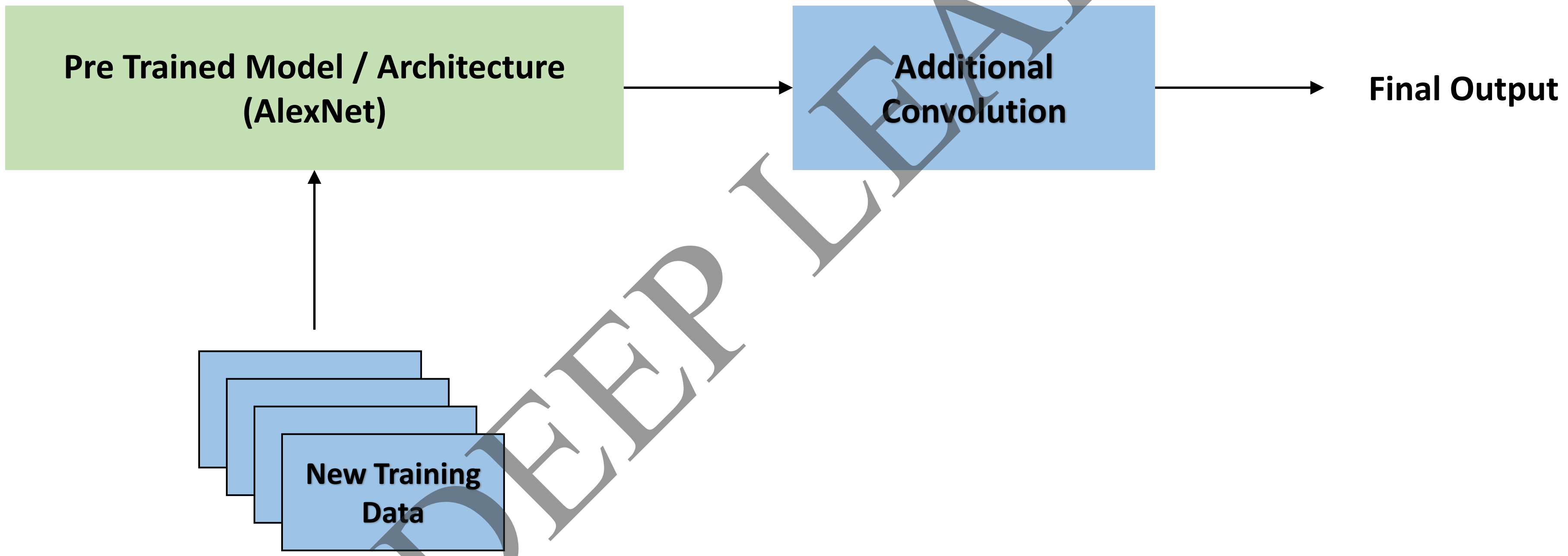


Layer		Feature Map	Size	Kernel Size	Stride	Activation
Input	Image	1	227x227x3	-	-	-
1	Convolution	96	55 x 55 x 96	11x11	4	relu
	Max Pooling	96	27 x 27 x 96	3x3	2	relu
2	Convolution	256	27 x 27 x 256	5x5	1	relu
	Max Pooling	256	13 x 13 x 256	3x3	2	relu
3	Convolution	384	13 x 13 x 384	3x3	1	relu
4	Convolution	384	13 x 13 x 384	3x3	1	relu
5	Convolution	256	13 x 13 x 256	3x3	1	relu
	Max Pooling	256	6 x 6 x 256	3x3	2	relu
6	FC	-	9216	-	-	relu
7	FC	-	4096	-	-	relu
8	FC	-	4096	-	-	relu
Output	FC	-	1000	-	-	Softmax

CONVOLUTION NEURAL NETWORKS

Transfer Learning

Existing architectures or models (that have already been trained on existing or generic data) can be used to retrain an additional layer of new data to get fine tuned results based on a specific requirement. This method of using existing model to train new data is called as "Transfer Learning"



END OF COMPUTER VISION

- PRITESH JHA

NATURAL LANGUAGE PROCESSING

What is NLP?

Natural Language Processing (NLP) is a field at the intersection of computer science, artificial intelligence, and linguistics.

It involves the development of algorithms and systems that enable computers to understand, interpret, and generate human language in a valuable and meaningful way.

NLP tasks often involve a deep understanding of both the language (syntax, semantics, grammar, etc.) and the context. Early approaches to NLP were based on sets of hand-written rules, but current state-of-the-art methods use machine learning, especially deep learning, to process and analyze large amounts of natural language data.

As a field, NLP is evolving rapidly with advancements like transformers and large language models (like GPT-3 and BERT), which have significantly improved the ability of machines to understand and generate human-like text.

NATURAL LANGUAGE PROCESSING

Applications of NLP

- **Language Translation**: Tools like Google Translate use NLP to convert text or speech from one language to another.
- **Speech Recognition**: Systems like Apple's Siri or Amazon's Alexa understand and respond to voice commands thanks to NLP.
- **Sentiment Analysis**: This involves analyzing text from reviews, social media, etc., to determine the sentiment behind it, like whether the text is positive, negative, or neutral.
- **Chatbots and Virtual Assistants**: NLP enables chatbots and virtual assistants to interact naturally with humans.
- **Information Extraction**: This involves extracting useful information from large amounts of text, like names, dates, and places from news articles.
- **Text Summarization**: Automatically generating a concise and coherent summary of a longer document.
- **Topic Detection and Modeling**: Identifying the main topics or themes in a large collection of documents.

NLP tasks using NLTK

NLTK Package in Python

The Natural Language Toolkit (NLTK) is a powerful Python library used for working with human language data (text) in the field of Natural Language Processing (NLP). It provides easy-to-use interfaces to over 50 corpora and lexical resources, along with a suite of text processing libraries for classification, tokenization, stemming, tagging, parsing, and semantic reasoning.

When to Use NLTK:

- Text Preprocessing: Tokenization, stemming, lemmatization.
- Building Basic NLP Models: Like a simple text classifier.
- Academic Purposes and Learning: NLTK's simplicity and extensive documentation make it great for education.
- Linguistic Research: With its vast array of corpora and lexical resources.

NLP tasks using NLTK

Corpora / Corpus

A corpus (plural: corpora) is a large and structured set of texts used in linguistic research and natural language processing.

Corpora in NLTK include a diverse range of texts such as literary works, chat transcripts, news articles, and more. These collections are used for linguistic analysis and algorithm training for various NLP tasks like sentiment analysis, topic modeling, and language modeling.

Tokenization

Tokenization is the process of dividing text into a sequence of tokens, which can be thought of as pieces or units of the text. In most cases, these tokens are words, but they can also be phrases, symbols, or other meaningful elements depending on the granularity of the tokenization process.

- **Word Tokenization:** The most common form, where text is split into individual words. It deals with the complexity of word boundaries, which can vary across different languages.

- **Sentence Tokenization:** Splitting a text into individual sentences, useful for tasks that require understanding the structure of the text, such as summarization or translation.

- **Subword Tokenization:** Breaking words into smaller units (like syllables or parts of words). This is particularly useful in languages where compound words are common, or for processing morphologically rich languages.

NLP tasks using NLTK

Stemming

- **Definition:** Stemming is the process of reducing words to their word stem or root form. The main idea is to remove suffixes from words, hoping to achieve this goal.
- **Method:** It typically uses algorithmic approaches, such as stripping suffixes ("ing", "ly", "es", "s" etc.) from words. For example, "running", "runner", and "ran" may all be reduced to the stem "run".
- **Speed and Simplicity:** Stemming algorithms are generally faster as they simply chop off the end of a word using heuristics, without any understanding of the context in which a word is used.
- **Accuracy:** This method can be less accurate, sometimes resulting in words that are not actual words (e.g., "argue", "argued", "arguing", all reduce to "argu").
- **Use Cases:** It's useful in search engines for indexing words. It's less accurate but faster than lemmatization.

NLP tasks using NLTK

Lemmatization

- **Definition:** Lemmatization is the process of reducing words to their base or dictionary form, known as the lemma. It involves a deeper linguistic understanding of the word.
- **Method:** It considers the context and converts the word to its meaningful base form. For instance, "is", "am", and "are" are all lemmatized to "be".
- **Contextual Accuracy:** Lemmatization is typically more accurate as it uses more informed analysis to create groups of words with similar meaning based on the context of the word. It often involves morphological analysis and requires additional linguistic knowledge, like part-of-speech and word sense disambiguation.
- **Speed:** It is generally slower than stemming as it involves a more complex algorithmic process to find the correct base form of the word.
- **Use Cases:** It's particularly useful in natural language understanding (NLU) tasks like language translation, sentiment analysis, and document summarization.

NLP tasks using NLTK

Stopwords

Stopwords are words that are filtered out before or after processing of natural language data (text).

They are typically the most common words in a language and are often considered as noise in various text processing applications because they carry less meaningful information about the content of the text.

Example : the, is, that, for, they, are, etc.

NLP tasks using NLTK

N-grams

N-grams are a concept used in natural language processing (NLP) and computational linguistics, referring to a contiguous sequence of n items from a given sample of text or speech.
An n-gram of size 1 is referred to as a "unigram"; size 2 is a "bigram"; size 3 is a "trigram". Beyond that, they might be called four-grams, five-grams, etc.
For example, in the sentence "The quick brown fox", "quick brown" is a bigram, and "The quick brown" is a trigram.

Language Modeling: N-grams are used in language modeling, where the probability of a word is approximated by the context provided by the preceding n-1 words.

Feature Extraction: In text mining, n-grams are used to create features for models. For instance, in spam detection systems or sentiment analysis, different n-grams can be features that are input to a classifier.

Importance in NLP: N-grams capture the language structure from the statistical point of view, i.e., considering the probability of each word (or letter) depends on the previous word (or letter).
They are used in various applications like text prediction, spell checking, and speech recognition.

Limitations:

Sparsity: As the value of n increases, the frequency of such n-grams in the corpus decreases, leading to sparsity issues.

Computational Cost: Larger n-grams (like 5-grams or 6-grams) may lead to a large number of features, which can be computationally expensive to process.

Context Limitation: N-grams do not account for long-range dependencies in text as they are limited to contiguous sequences of words.

NATURAL LANGUAGE PROCESSING

Recurrent Neural Networks

In CNN architectures, we reshape all the data into the same shapes before feeding it into the model.
This is possible with image data.

For text data, each message / statement / document can be of a different length.

Thus, the concept of "recurrence" is used in Natural Language Processing.
In simple words, the data is actually utilized in a way that –
"Each word is trained on the word prior to it".

Thus all the data points can have different length

NATURAL LANGUAGE PROCESSING

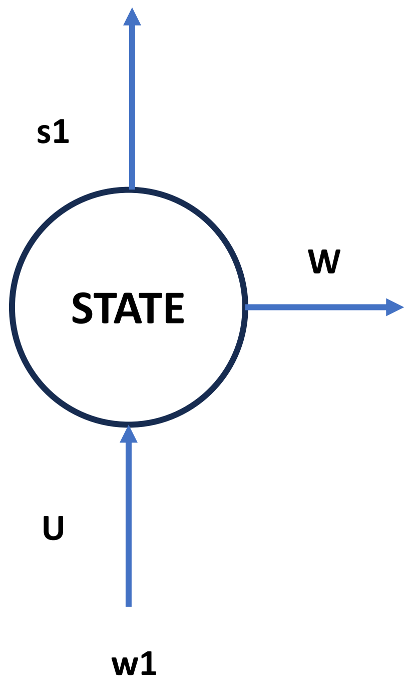
Recurrent Neural Networks

In RNN, the network outputs two things –

- Prediction : What would be predicted if the sequence ended with that word
- State : Summary of everything that has happened in the past

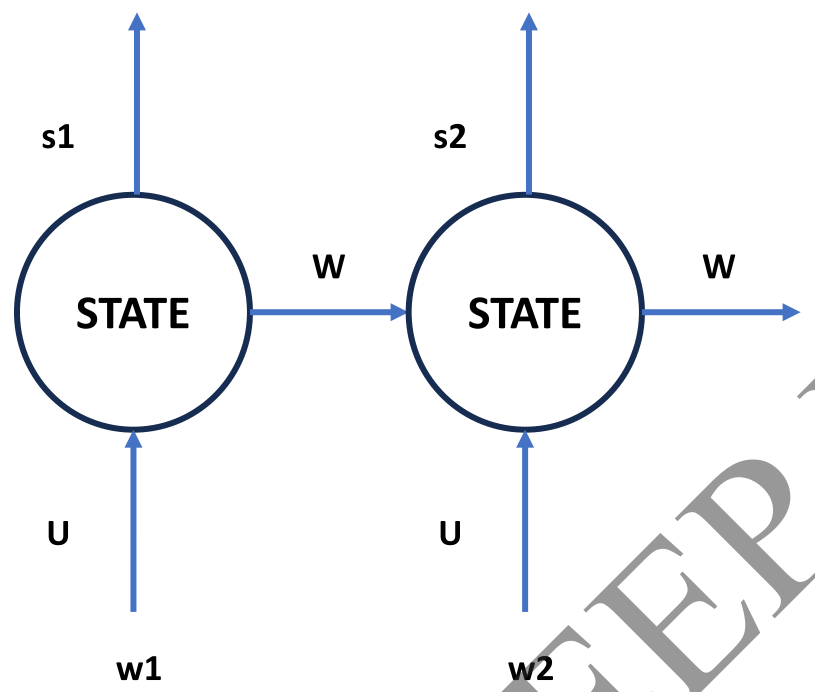
NATURAL LANGUAGE PROCESSING

Recurrent Neural Networks



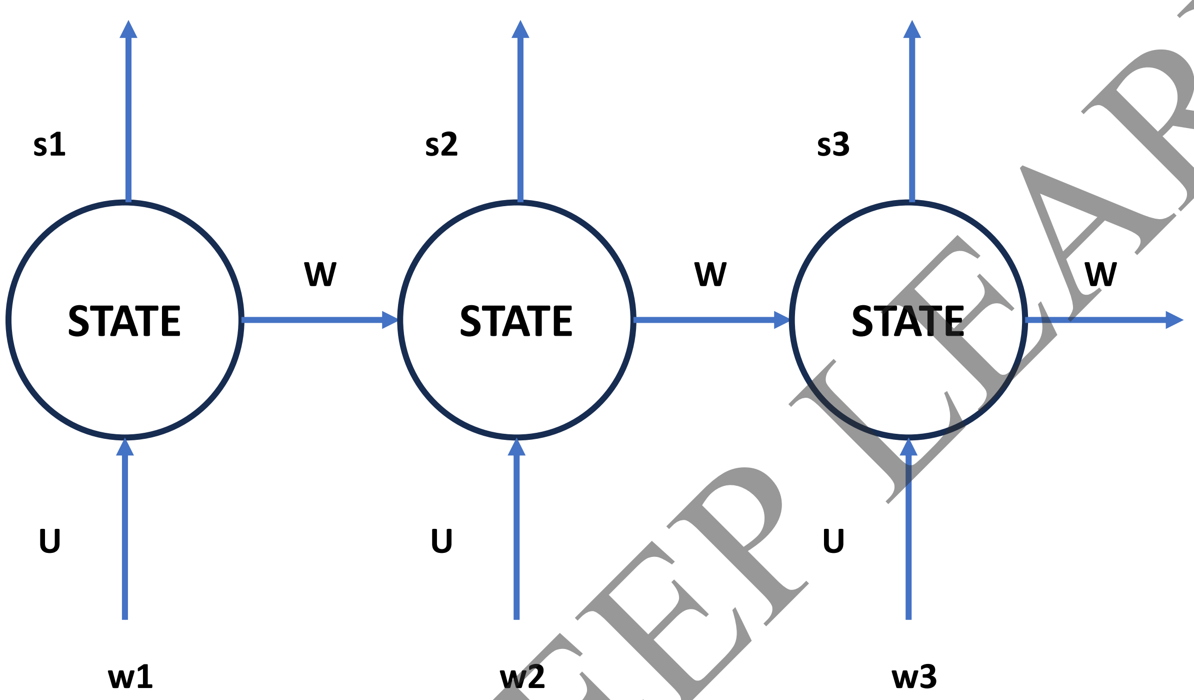
NATURAL LANGUAGE PROCESSING

Recurrent Neural Networks



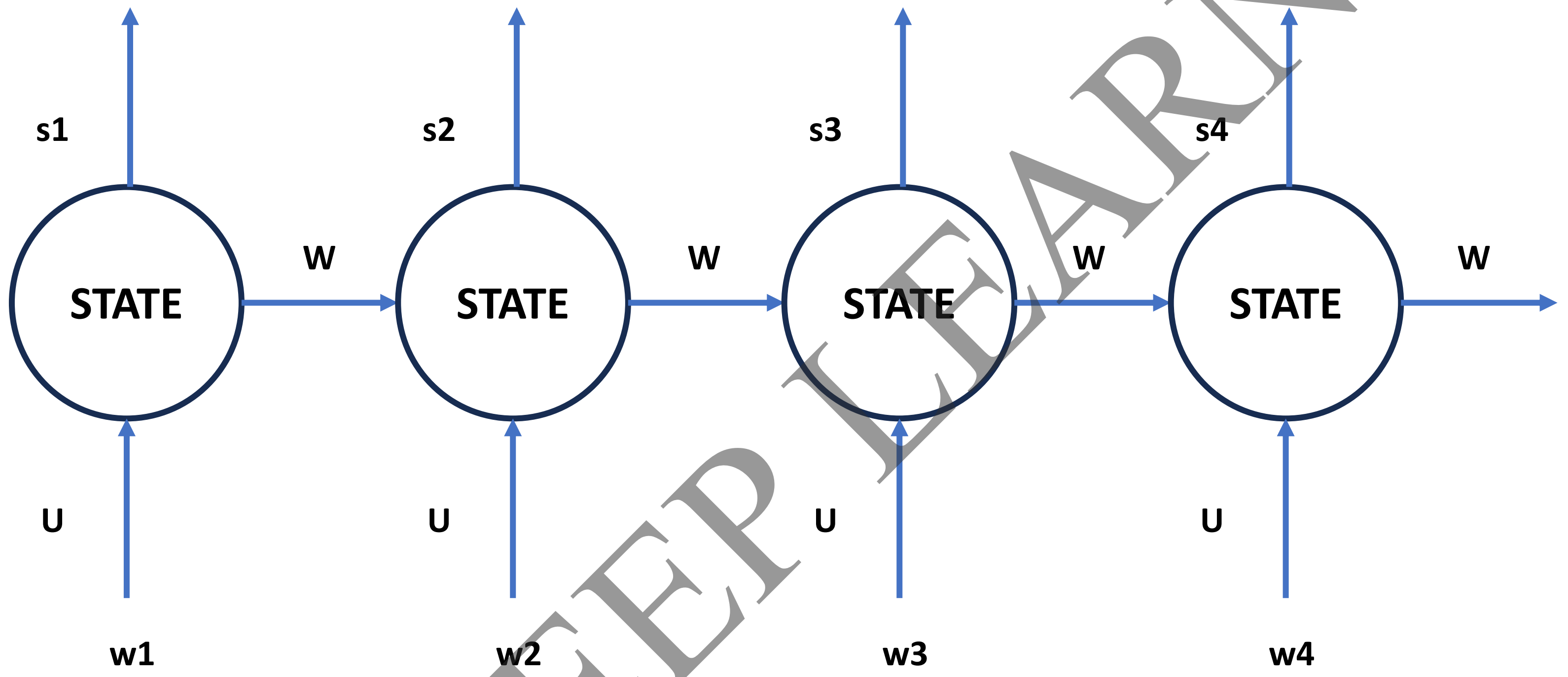
NATURAL LANGUAGE PROCESSING

Recurrent Neural Networks



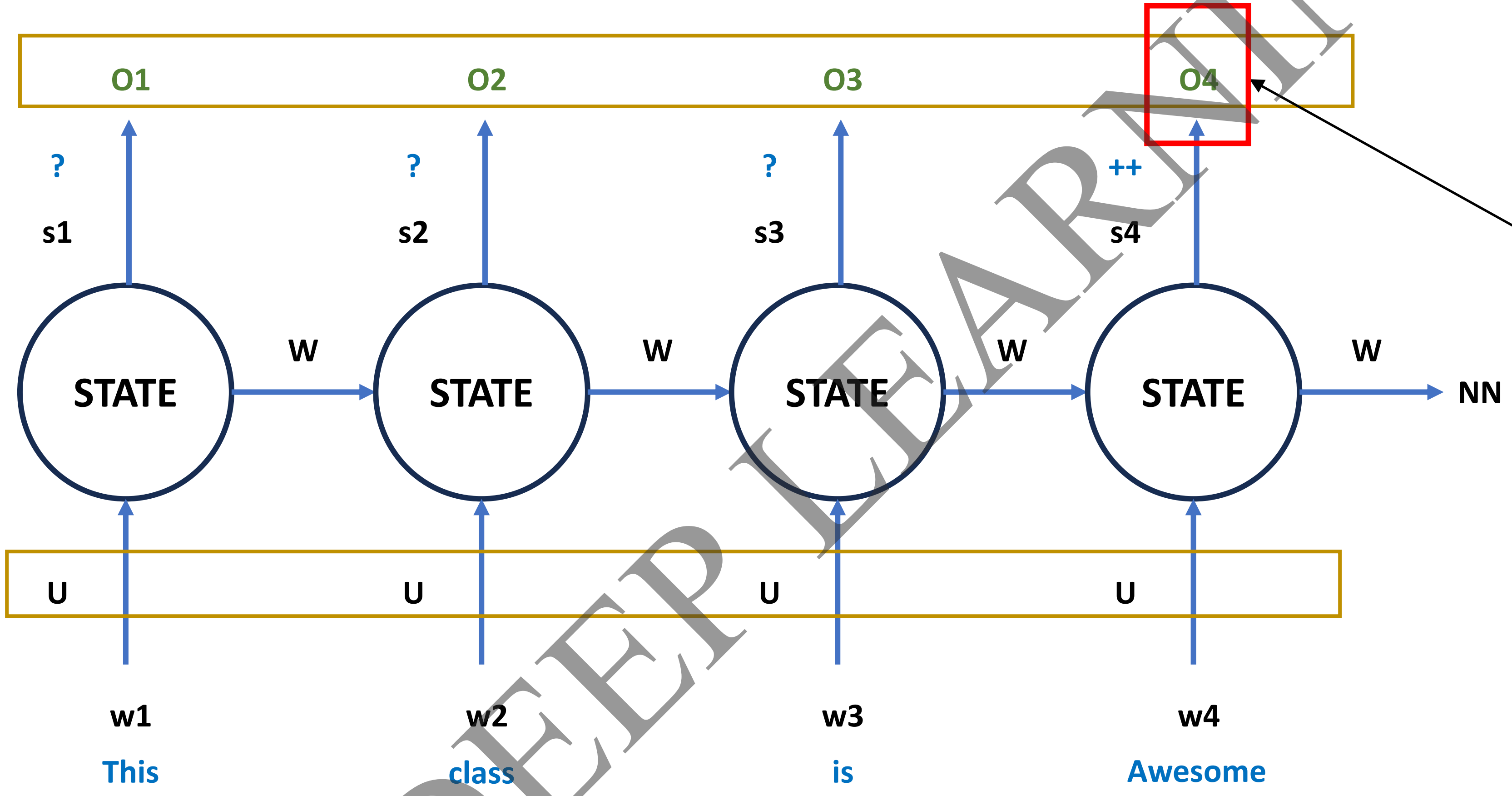
NATURAL LANGUAGE PROCESSING

Recurrent Neural Networks



NATURAL LANGUAGE PROCESSING

Recurrent Neural Networks



Since, each word / input is trained based on previous word and previous state,

O4 is the only output that will be fed into the Neural Network for training (as Dense layer in Keras).

These are called as **"Kernels"**

RECURRENT NEURAL NETWORKS

Mathematical Equation

w_i is the word at position i

s_i is the state at position i

o_i is the output at position i

$s_i = f(Uw_i + Ws_{i-1})$ (Core RNN)

$o_i = softmax(Vs_i)$ (subsequent dense layer)

– current state = function1(old state, current input).
– current output = function2(current state).

RECURRENT NEURAL NETWORKS

Matrix Calculations

r = dimension of input vector

s = dimension of hidden state

t = dimension of output vector (after dense layer)

U is a s × r matrix

W is a s × s matrix

V is a t × s matrix

Note: The weight matrices **U, V, W** are the same across all positions.

RECURRENT NEURAL NETWORKS

Points to remember when building a RNN architecture

- We only train on the "final" output and ignore the intermediate outputs
- Slight variation called Backpropagation Through Time (BPTT) is used to update weights while training RNNs
- Sensitive to length of sequence (due to vanishing gradient problem)
  Hence, we set a maximum length of our sequences.
- If the sequence length is small, we pad it.
  If the sequence length is large, we truncate.

RECURRENT NEURAL NETWORKS

Limitations of RNN

- As only the final output is being utilised during state transition, any information from distant past is lost in the memory

(This drawback is covered under LSTM technique)

RECURRENT NEURAL NETWORKS

Code Structure of RNN using Keras

- Sequential
- Embedding(input_dim, output_dim)
- SimpleRNN(units, activation*)
- compile(optimizer, loss, metrics)
- model.summary()

Long-Short Term Memory

What is LSTM?

- Long Short-Term Memory (LSTM) is a type of Recurrent Neural Network (RNN) architecture used in the field of deep learning. LSTM has been widely used for sequence prediction problems and has gained fame for its performance in applications like language modeling, machine translation, and speech recognition.

Core Concepts of LSTM:

- **Handling Long-Term Dependencies:** Traditional RNNs struggle with long-term dependencies due to the vanishing gradient problem, where the gradients used in the training process become extremely small, causing the earlier layers to learn very slowly. LSTM addresses this by introducing a memory cell that can maintain information in memory for long periods of time.

- **LSTM Units:** An LSTM network is composed of LSTM units. Each unit has three gates and a cell state:

    - Forget Gate: Decides what information should be thrown away from the cell state.
    - Input Gate: Updates the cell state with new information.
    - Output Gate: Determines what the next hidden state should be, which is used in the output of the LSTM unit.

- **Cell State:** The cell state runs straight down the entire chain, with only minor linear interactions. It’s very easy for information to just flow along it unchanged.

Long-Short Term Memory

Workflow of LSTM:

- **First Step (Forget Gate):** It decides what information is discarded from the cell state. This decision is made by a sigmoid layer.

- **Second Step (Input Gate):** It updates the cell state. First, a sigmoid layer decides which values the cell state should update, and then a tanh layer creates a vector of new candidate values that could be added to the state.

- **Third Step (Cell State Update):** The old cell state is updated into the new cell state. The previous steps already decided what to do, and this step actually carries out the process.

- **Fourth Step (Output Gate):** Based on the cell state, the final decision is made about what to output. This output is based on a filtered version of the cell state.

Long-Short Term Memory

Code Structure of LSTM using Keras

- Sequential
- Embedding(input_dim, output_dim)
- LSTM(units, activation*, recurrent_activation*)
- compile(optimizer, loss, metrics)
- model.summary()

ADDITIONAL ARCHITECTURES

Code Structure of LSTM using Keras

- Gated Recurrent Units
- Seq2Seq Models
- Bidirectional RNN (BiRNN)
- Echo State Networks

- Transformers
    - BERT
        - RoBERTa
        - ALBERT

END OF N.L.P