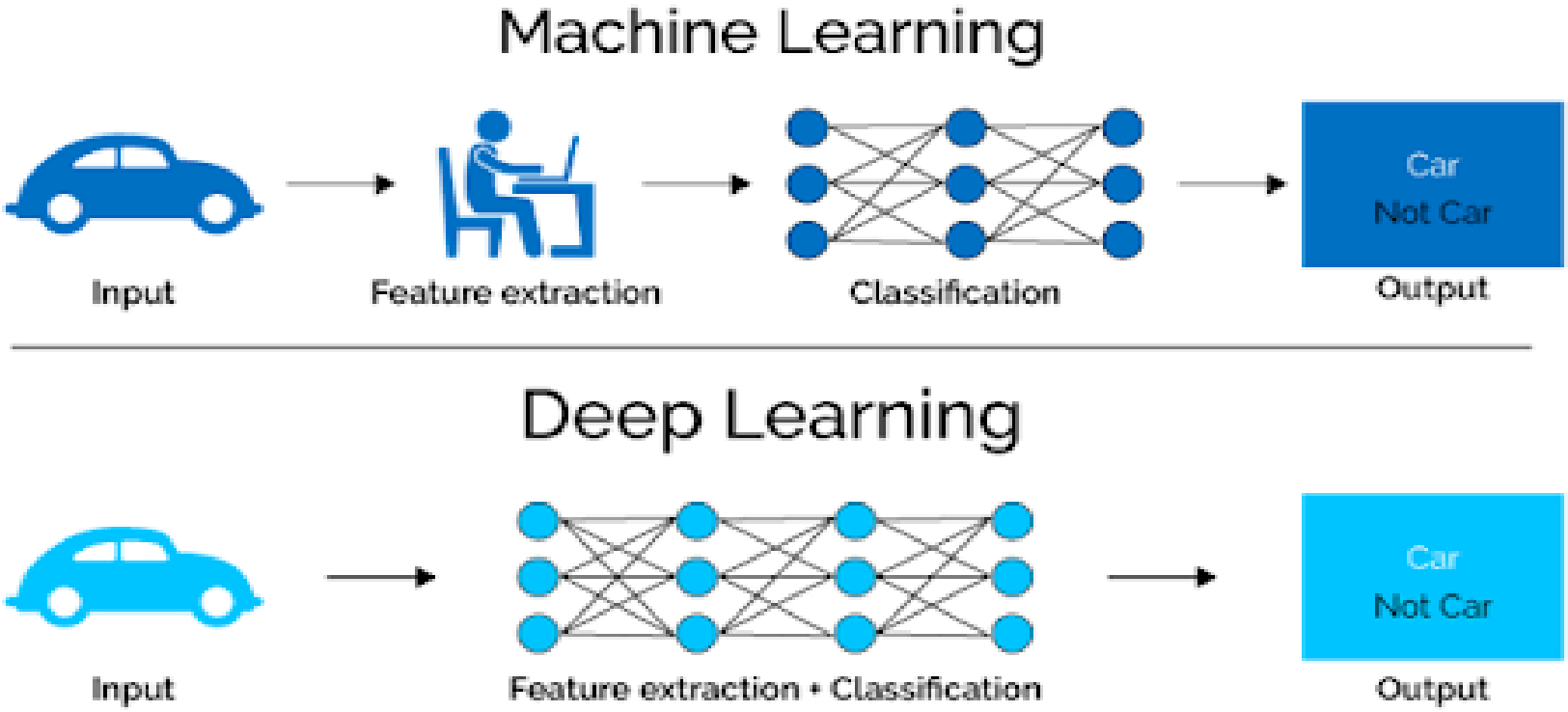
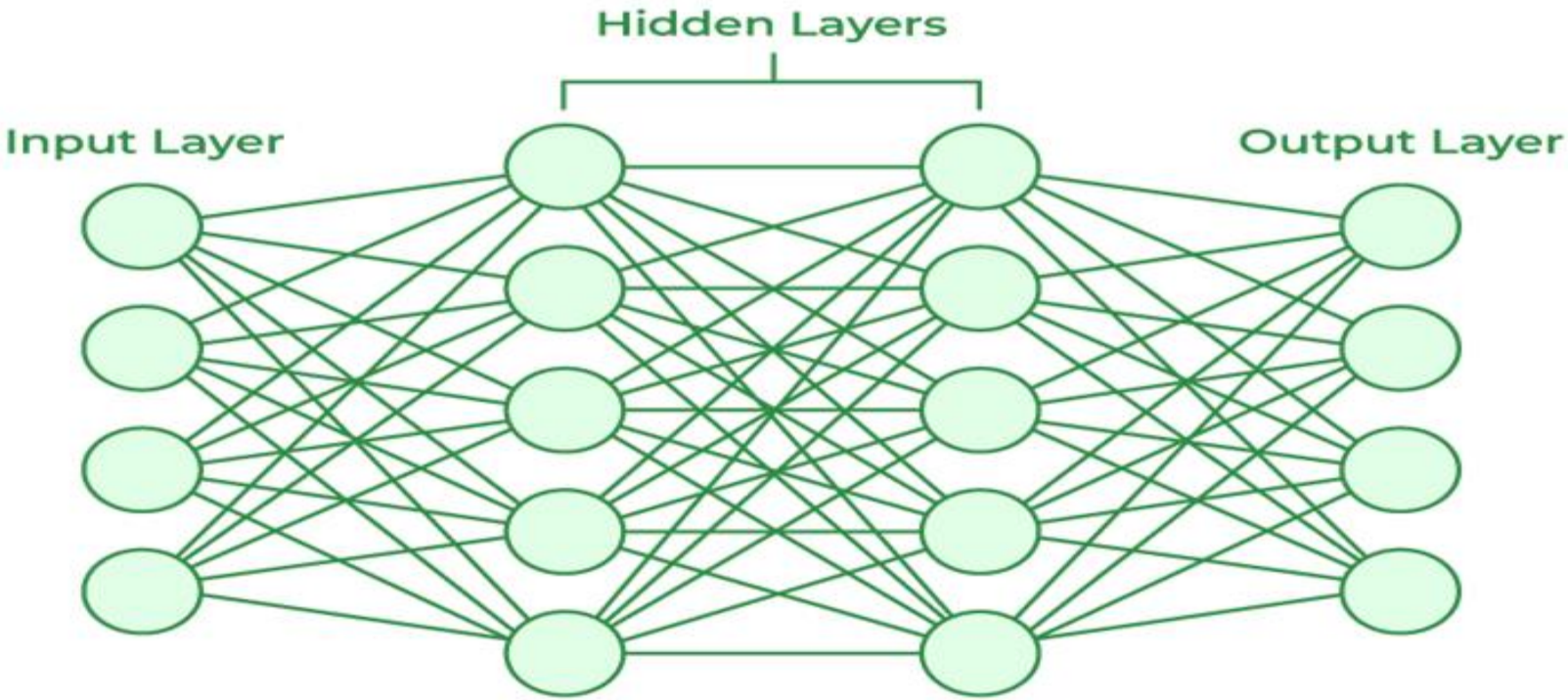

Adobe Stock | #201339012

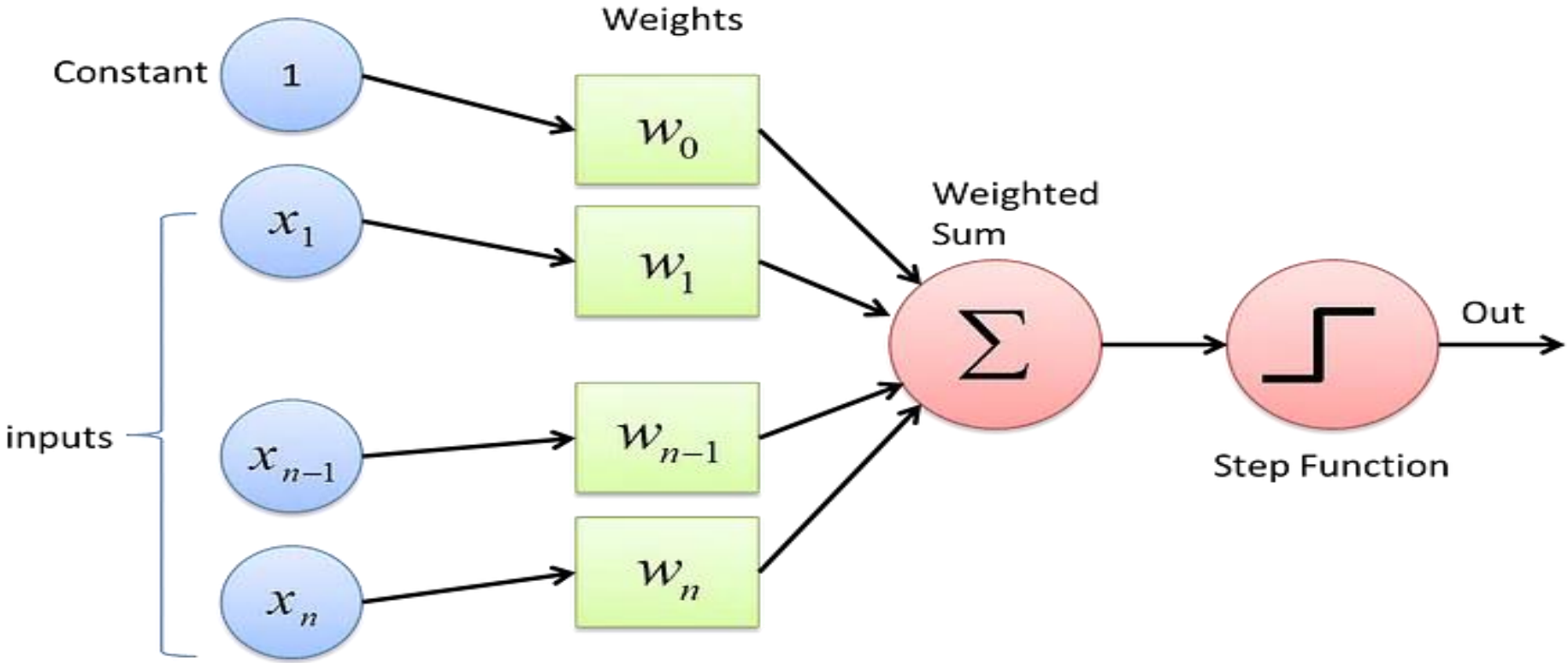
# Introduction to deep learning



# Artificial Neural Network



# What is perceptron ?



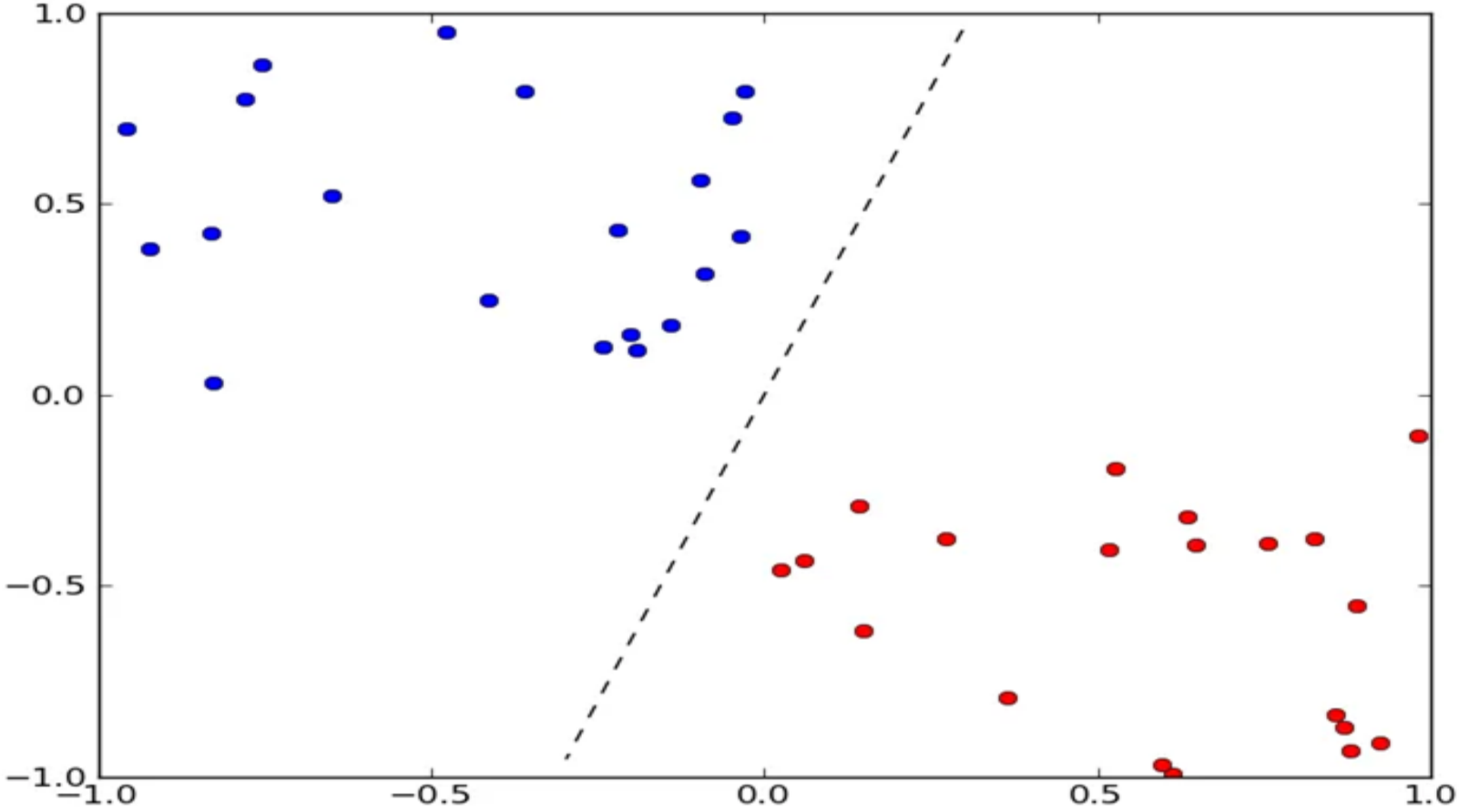
# Functionality of the Perceptron

- It is designed to mimic human brain
- If the weighted sum is greater than a threshold (usually 0), it outputs one class (e.g., 1)
- If it's less than or equal to that threshold, it outputs the other class (e.g., 0).

# Limitations of Perceptron

1. They can only solve **linearly separable** problems (problems where a single straight line can separate the classes).

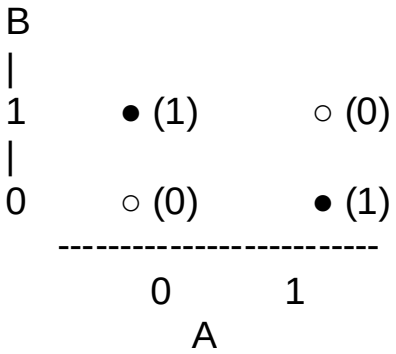
1. They cannot handle complex problems like XOR (exclusive OR), where the classes aren’t linearly separable.


1.0
0.5
0.0
−0.5
−1.0
−1.0
−0.5
0.0
0.5
1.0

# Problem with XOR

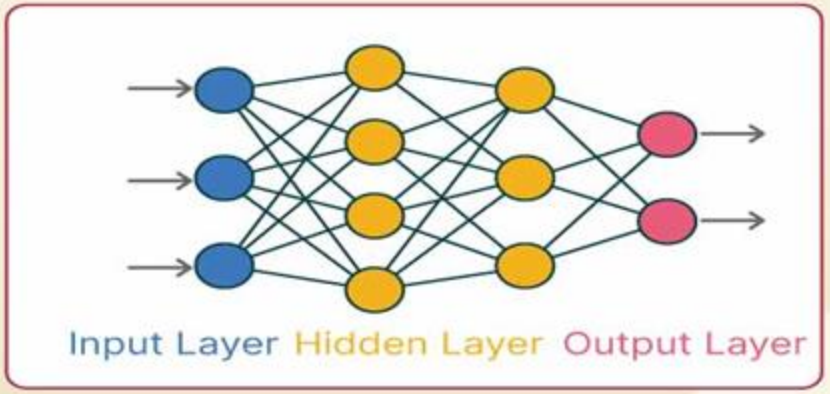
The XOR (Exclusive OR) truth table for two binary inputs, $A$ and $B$, looks like this:

| A | B | Output |
|---|---|---|
| 0 | 0 | 0 |
| 0 | 1 | 1 |
| 1 | 0 | 1 |
| 1 | 1 | 0 |


B
1
0
● (1)
○ (0)
○ (0)
● (1)
0
1
A

# Solution

## Multilayer Perceptron (MLP) Neural Networks




# Working of MLP

- **Input Layer** - Receives the input features and passes them to the first hidden layer.
- **Hidden Layers -** Each neuron in these layers computes a weighted sum of its inputs, adds a bias, and applies an activation function (like ReLU or Sigmoid) to introduce non-linearity, enabling the model to capture complex patterns.

- **Output layer** - After passing through all hidden layers, the output layer produces the final result. In classification, this output is often interpreted as class probabilities.

- **Training with Backpropagation**: During training, the model calculates the difference (error) between the predicted and actual outputs. Backpropagation, combined with an optimization algorithm like gradient descent, adjusts the weights and biases to minimize this error, iteratively refining the model's predictions.

# Why activation functions are needed ?

- Activation functions are essential in neural networks because they introduce non linearities,enabling network to learn complex patterns.
- Without them, the network would only be able to model linear relationships, limiting its ability to solve intricate problems, especially in high-dimensional data. Here’s a breakdown of why activation functions are important:

# Properties of an ideal activation function

- Non-linear - to add more complexity
- Differentiable - to calculate gradients
- Zero centred
- non - saturating - to avoid vanishing and exploding gradients

# Types of activation functions

- Sigmoid function
- Tanh function
- Softmax function
- Relu function
- Relu variants

# Sigmoid function

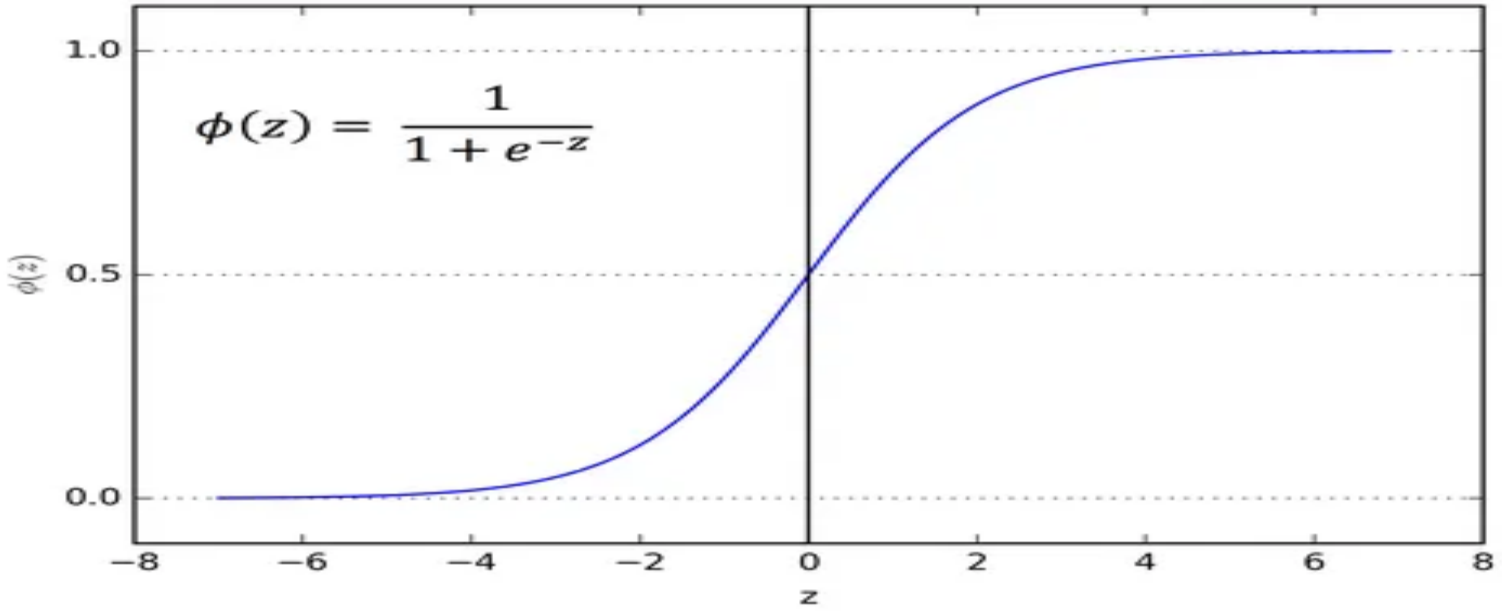



Fig: Sigmoid Function

- The main reason of using sigmoid because it exists between (0,1) used to predict probabilities for classification tasks.

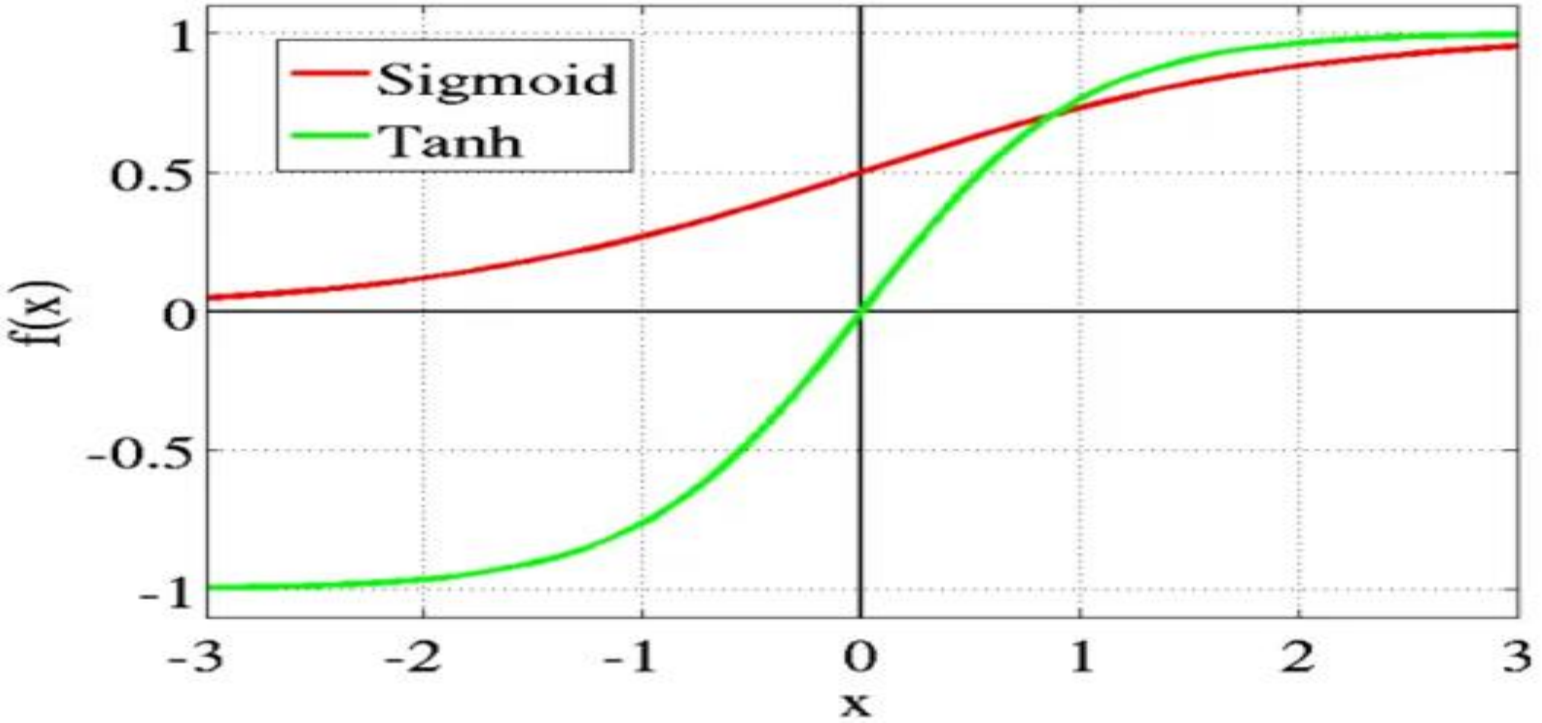
- Sigmoid function is differentiable so gradients can be computed

- Problem with sigmoid activation is saturation which leads to vanishing gradients

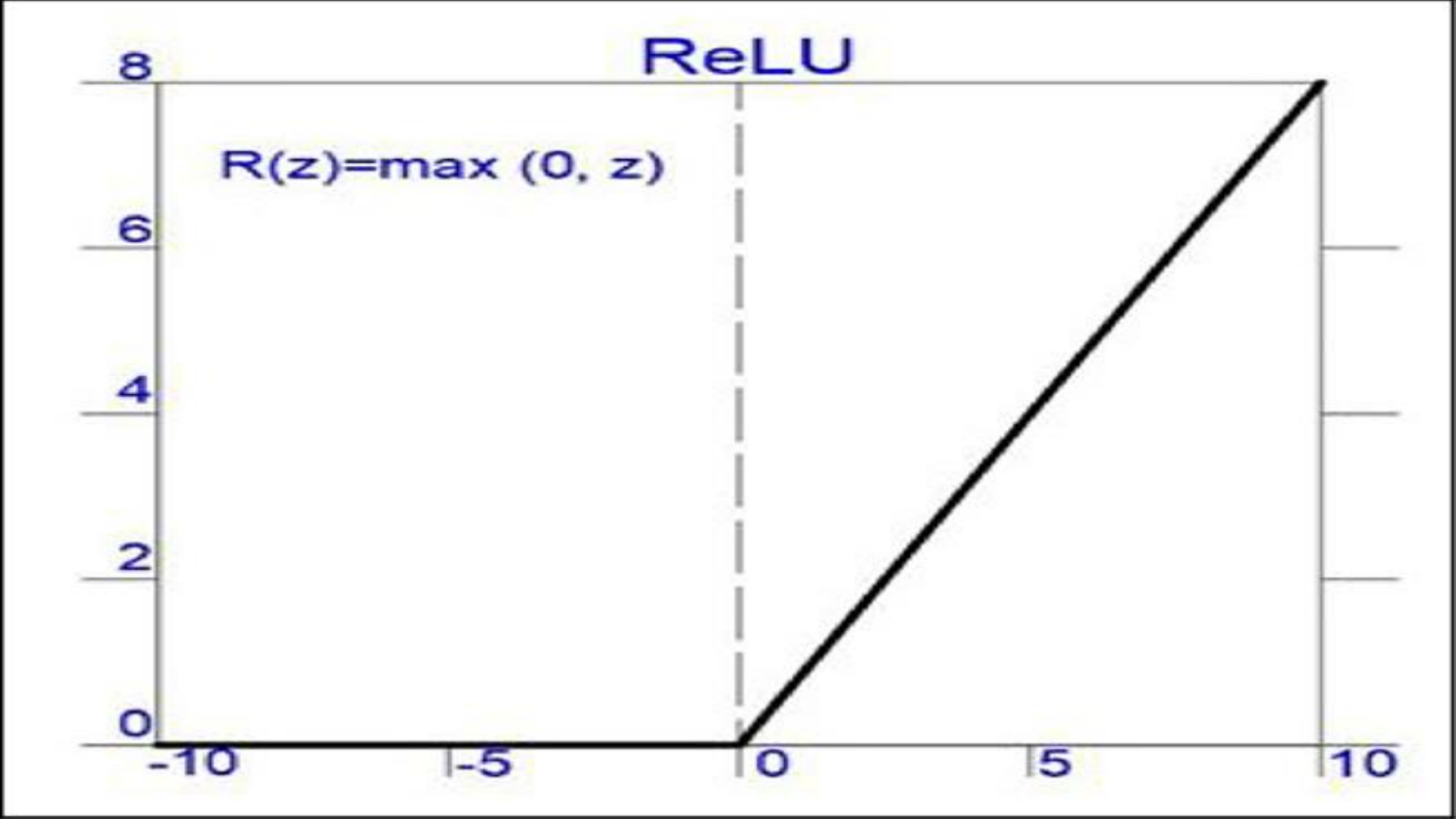
# Perceptron vs Logistic regression

- Perceptron with sigmoid activation will be same as logistic regression.

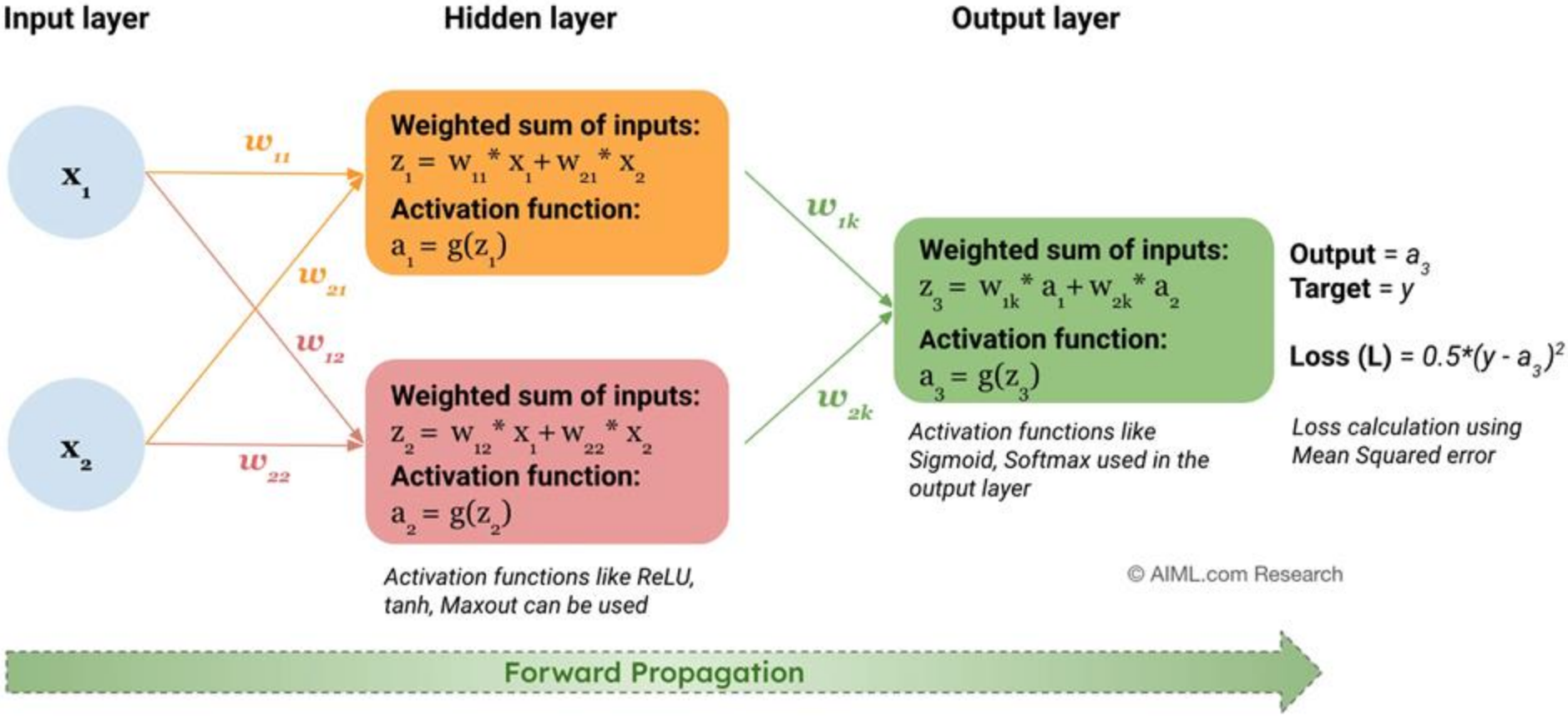
# Tanh function



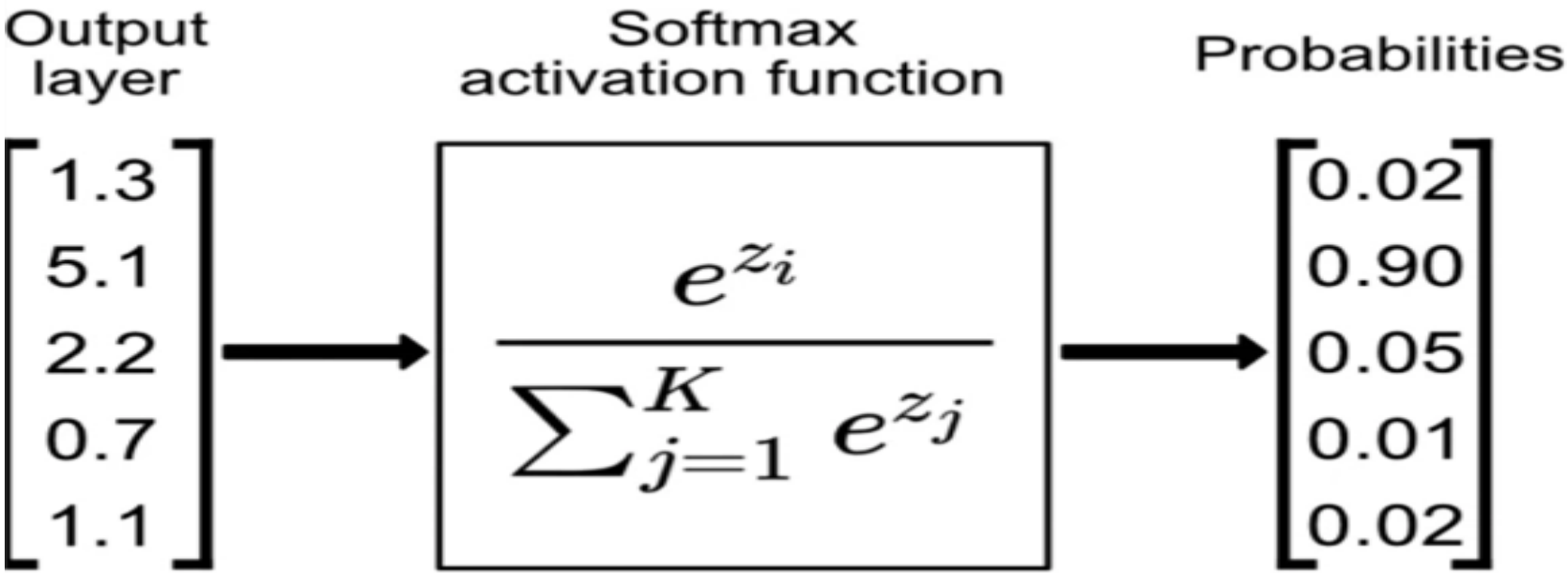
- Output - (-1,1)
- Zero centred - Means mean close to zero and leads to faster convergence
- Stronger gradients compared to sigmoid in mid range values
- Same problem of vanishing gradient similar to sigmoid.


ReLU
R(z)=max (0, z)
8
6
4
2
0
-10
-5
0
5
10

- Non-linearity
- Efficient computation than sigmoid and tanh
- Removes vanishing gradient problem
- Sparse activation function
- Improves training performance and stability


Input layer
Hidden layer
Output layer
$x_1$
$x_2$
$w_{11}$
$w_{21}$
$w_{12}$
$w_{22}$
Weighted sum of inputs:
$z_1 = w_{11} * x_1 + w_{21} * x_2$
Activation function:
$a_1 = g(z_1)$
Weighted sum of inputs:
$z_2 = w_{12} * x_1 + w_{22} * x_2$
Activation function:
$a_2 = g(z_2)$
Activation functions like ReLU, tanh, Maxout can be used
$w_{1k}$
$w_{2k}$
Weighted sum of inputs:
$z_3 = w_{1k} * a_1 + w_{2k} * a_2$
Activation function:
$a_3 = g(z_3)$
Activation functions like Sigmoid, Softmax used in the output layer
Output = $a_3$
Target = $y$
Loss (L) = $0.5*(y - a_3)^2$
Loss calculation using Mean Squared error
© AIML.com Research
Forward Propagation

# Softmax activation function



- Softmax activation function converts logits into probabilities for multi-class classification by exponentiating each logit and normalizing by their sum
- Outputs from each class are from 0 to 1 summing them to 1
- Provides stable training for multi-class tasks by heavily penalizing confident but incorrect predictions.
- These properties make softmax ideal for converting raw scores into interpretable class probabilities.

# Conclusion

Sigmoid/Softmax activation function - Output layers
Relu - Hidden layers
Tanh - Hidden layers but RNN and LSTM

# Why Loss function is needed.

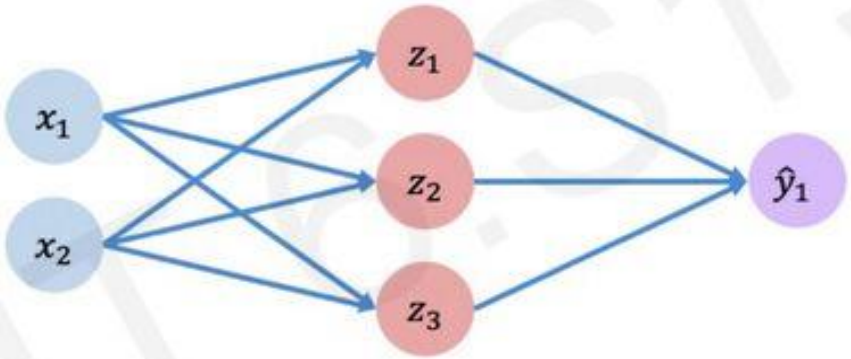
- We can’t improve if we can’t measure
- Example:- total marks = 100 , obtained marks =60 diff/loss = 40.
- Similarly loss in deep learning - (true output-measured o/p)


Classification
Log Loss
Focal Loss
KL Divergence/ Relative Entropy
Exponential Loss
Hinge Loss
Regression
Mean Square Error/ Quadratic Loss
Mean Absolute Error
Huber Loss/ Smooth Mean Absolute Error
Log cosh Loss
Quantile Loss

# Binary Cross Entropy Loss

***Cross entropy loss*** *can be used with models that output a probability between 0 and 1*

$$X = \begin{bmatrix} 4, & 5 \\ 2, & 1 \\ 5, & 8 \\ \vdots & \vdots \end{bmatrix}$$

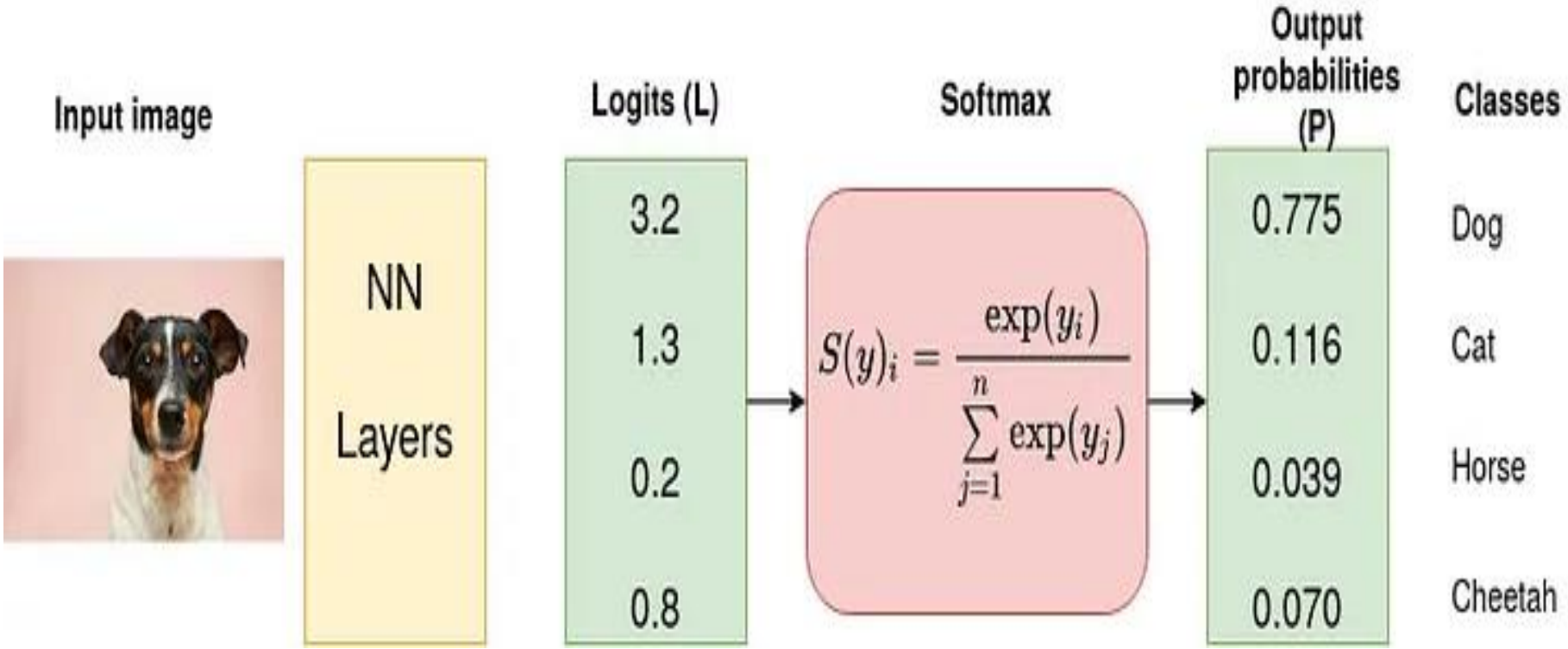



$f(x)$ $y$

$$\begin{bmatrix} 0.1 \\ 0.8 \\ 0.6 \\ \vdots \end{bmatrix} \begin{matrix} \times \\ \times \\ \checkmark \\ \end{matrix} \begin{bmatrix} 1 \\ 0 \\ 1 \\ \vdots \end{bmatrix}$$

$$J(W) = -\frac{1}{n}\sum_{i=1}^{n} \underbrace{y^{(i)}}_{\text{Actual}} \underbrace{\log\left(f\left(x^{(i)}; W\right)\right)}_{\text{Predicted}} + \underbrace{(1 - y^{(i)})}_{\text{Actual}} \underbrace{\log\left(1 - f\left(x^{(i)}; W\right)\right)}_{\text{Predicted}}$$

```
loss = tf.reduce_mean( tf.nn.softmax_cross_entropy_with_logits(y, predicted) )
```


Input image
NN
Layers
Logits (L)
3.2
1.3
0.2
0.8
Softmax
$S(y)_i = \frac{\exp(y_i)}{\sum_{j=1}^{n} \exp(y_j)}$
Output probabilities (P)
0.775
0.116
0.039
0.070
Classes
Dog
Cat
Horse
Cheetah

# Mean square error

$$\mathrm{MSE} = \frac{1}{n}\sum_{i=1}^{n}(y_i - \hat{y}_i)^2$$

where:

- $y_i$ represents the actual value.
- $\hat{y}_i$ represents the predicted value.
- $n$ is the number of observations.

- Non - negative and differentiable
- Sensitive to outliers
- Heavily penalized large errors.
- Scale dependent (higher values leads to higher mse)

- Does not handle nonlinear errors well.

# Mean Absolute error

$$MAPE = \frac{1}{n}\sum_{i=1}^{n}\frac{|A_i - F_i|}{A_i}$$

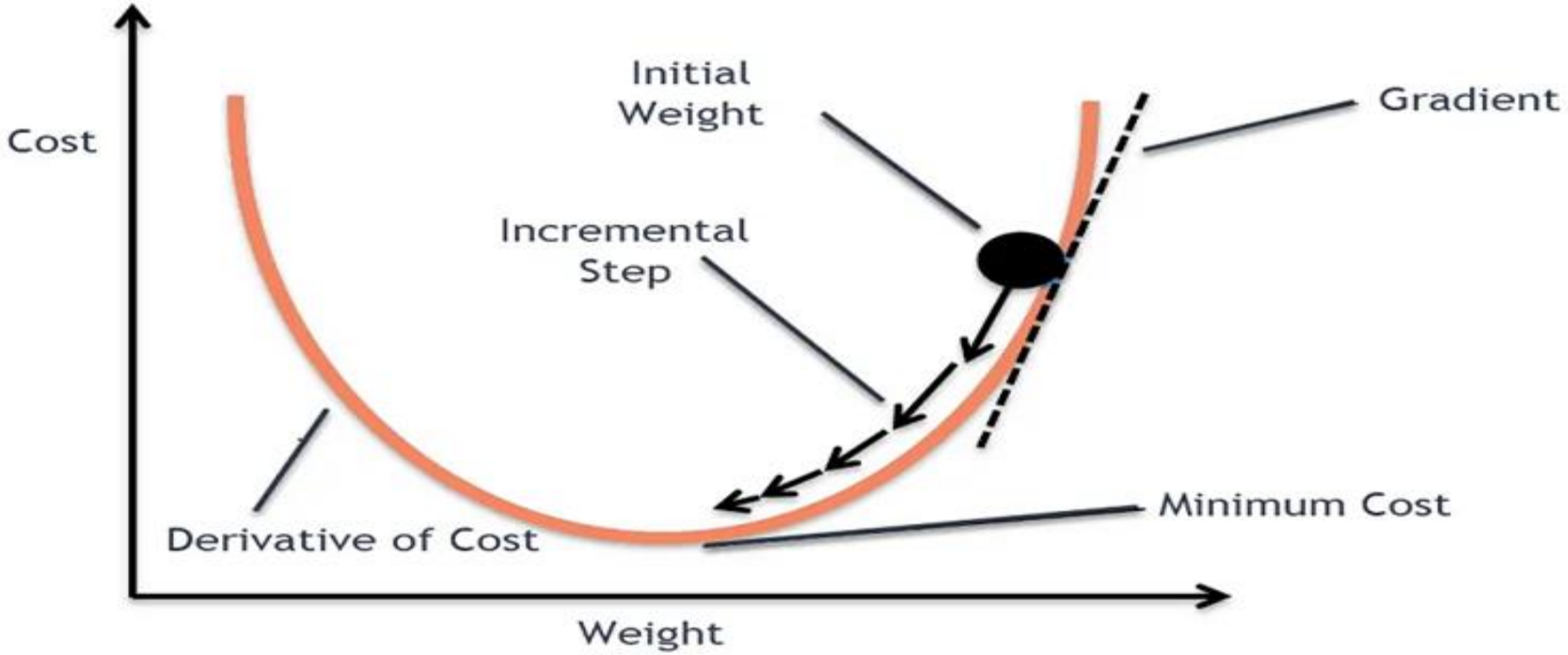
$A_i$ is the actual value

$F_i$ is the forecast value

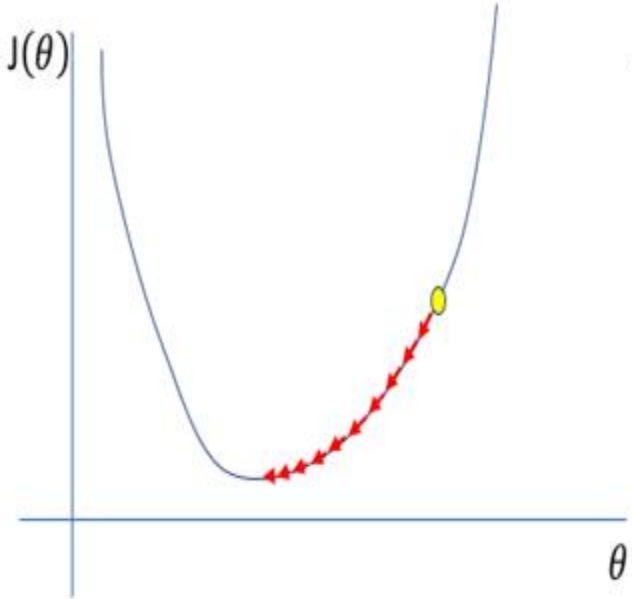
n is total number of observations

- Interpretability
- Sensitivity to magnitude of errors
- Robustness to outliers
- Not differentiable.
- No directional bias
- Linearity

# Gradient descent(Optimization algorithm)

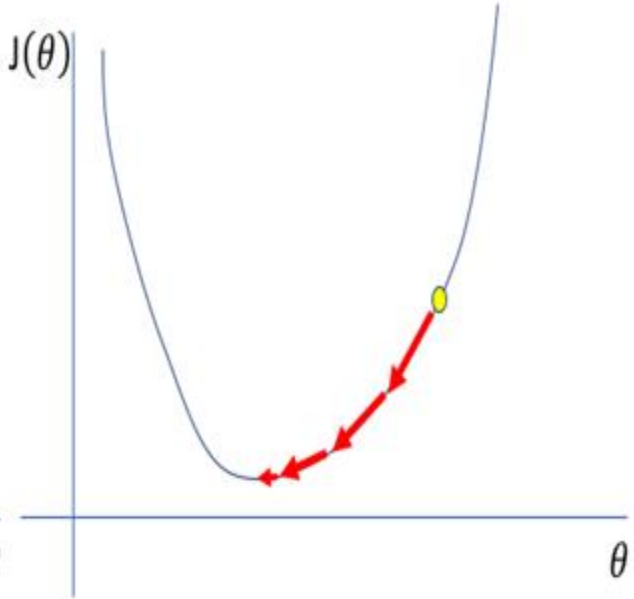


## Too low



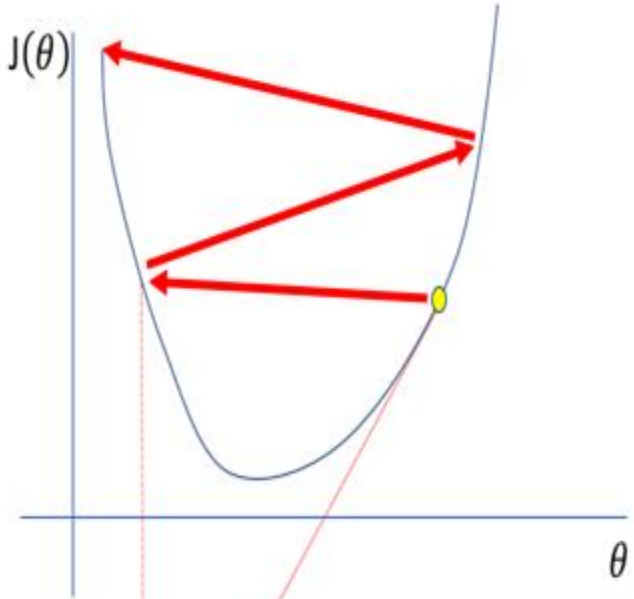

A small learning rate requires many updates before reaching the minimum point

## Just right




The optimal learning rate swiftly reaches the minimum point

## Too high




Too large of a learning rate causes drastic updates which lead to divergent behaviors

# Type of gradient descent algorithms

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

# Batch gradient descent

- **Process the entire dataset** - Calculates the gradient of the loss function using the entire training dataset.
- **Stable convergence** - Batch gradient descent converges smoothly towards the global minimum.
- **Slower and memory intensive**
- **Updates are consistent and repeatable, as each iteration uses the full dataset.**
- **Limited for Large-Scale or Online Learning**

# Stochastic gradient descent

- Start with randomly initialized model params.
- Select random sample.
- Compute gradient of loss function w.r.t model params using only the selected sample.
- Update the gradient in the direction to minimize the negative loss.

# Key characteristics of SGD

- Speed
- Noisy updates
- Choosing an appropriate learning rate

# Mini batch gradient descent

- It is a variant of gradient descent that balances between the extremes of **stochastic gradient descent (SGD)** and **batch gradient descent** by dividing the dataset into small, manageable chunks, or "mini-batches."

- It allows for faster convergence and more stable updates by taking advantage of both computational efficiency of batch processing and reduced variance of updates seen in SGD.

# How it works

**Divide the dataset** - Split the dataset into mini batches typically ranging from 32 to 512 examples (depending on the dataset size and computational resources).

**Update parameters** - For each mini-batch, compute the gradient of the loss function and update the model parameters based on this mini-batch gradient.

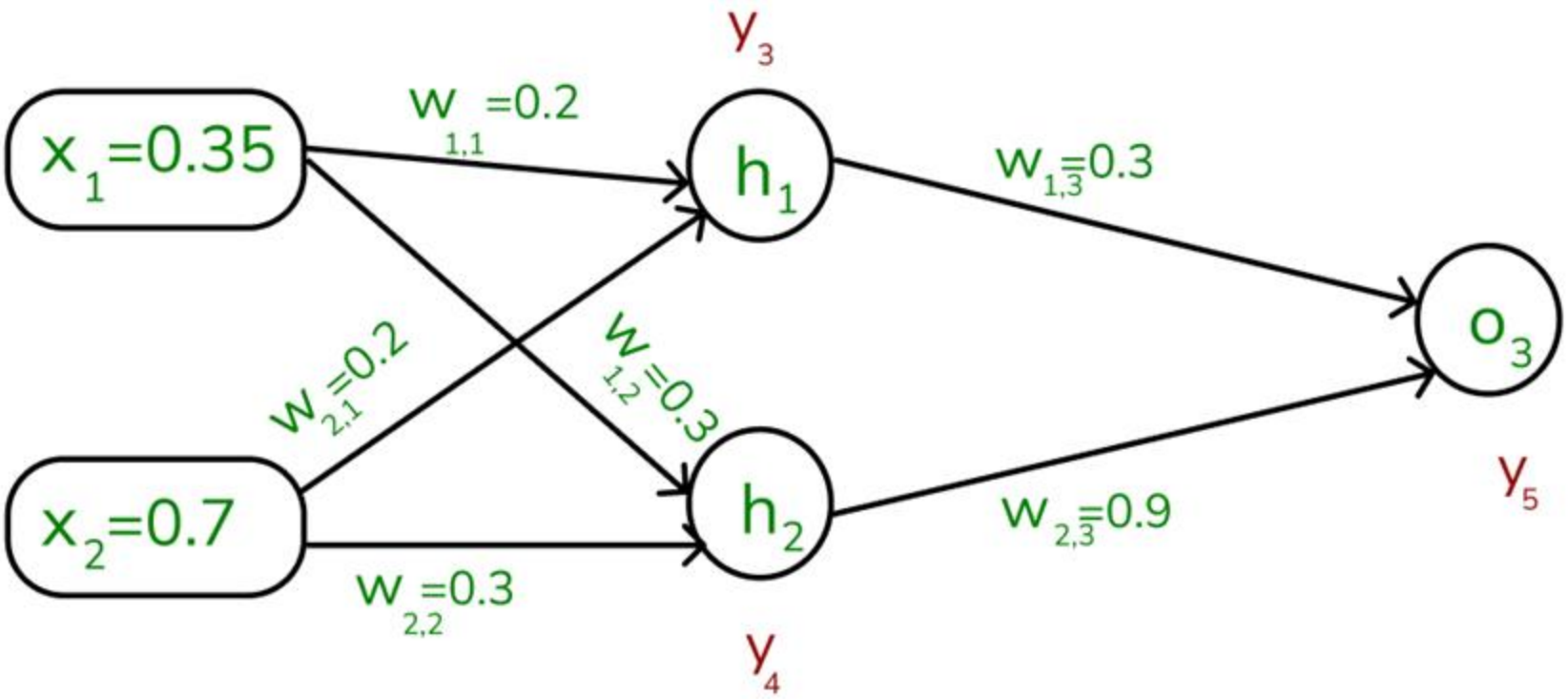
**Iterate**: Repeat the process for all mini-batches and continue iterating across the dataset multiple times (epochs) until convergence.

# Practical considerations

- Choosing mini batch size  of (2,4,8,16,32,64,128…) as they are optimized for memory operations in GPUs

- **Learning rate adjustment** -  As training progresses, it may be necessary to adjust the learning rate or use learning rate schedules to avoid overshooting the minimum.

# Forward propagation



- **Input Layer**: Inputs are fed into the network's first layer, representing the features of the data.

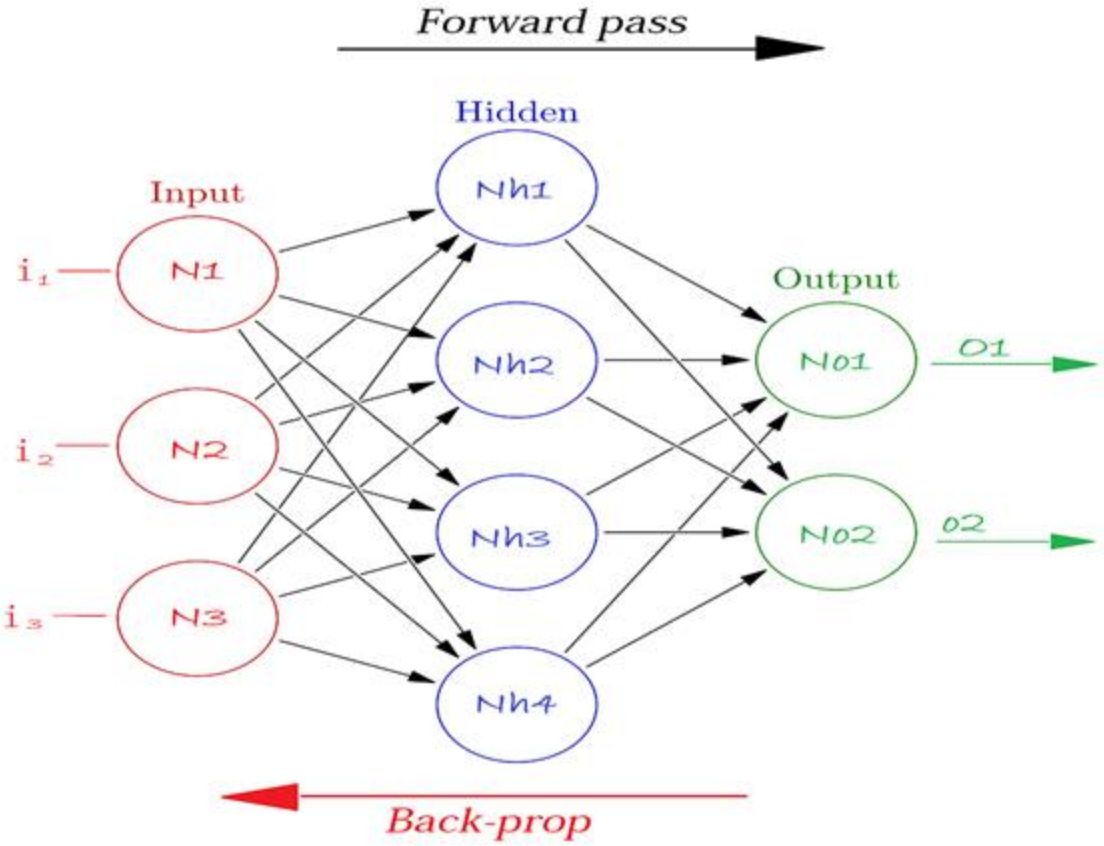
- **Weighted Sum**: Each input is multiplied by weights, and biases are added to form a weighted sum for each neuron.

- **Activation Function**: The weighted sum is passed through an activation function, adding non-linearity to the output.

- **Layer-by-Layer Propagation**: Outputs from one layer become inputs to the next, continuing through all hidden layers.

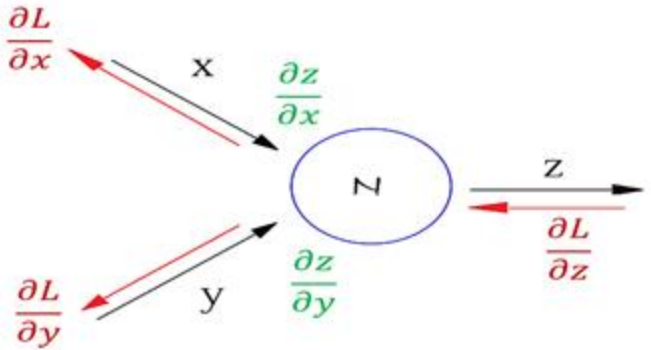
- **Output Layer**: The final layer produces the network's prediction, completing the forward pass.

# Back propagation




## Chain rule

$$\frac{\partial L}{\partial x} = \frac{\partial L}{\partial z} * \frac{\partial z}{\partial x}$$




$$\frac{\partial L}{\partial y} = \frac{\partial L}{\partial z} * \frac{\partial z}{\partial y}$$

- **Error Calculation**: Calculate the error or "loss" at the output layer by comparing the predicted values with the actual values (using a loss function, e.g., mean squared error or cross-entropy).

- **Gradient Computation**: Compute the gradient of the loss function with respect to each weight by applying the chain rule. This step involves propagating the error backward from the output layer to each preceding layer.

- **Weight Adjustment**: Use these gradients to update the weights in the network. The amount of adjustment is controlled by the learning rate, which determines the step size for each update.

- **Iterative Process**: Repeat the forward pass and backpropagation for multiple epochs (training iterations) until the loss converges to a minimum or reaches a satisfactory level.

- **Optimization**: This process helps the neural network learn by minimizing the error, refining the weights to improve predictions over time and enable accurate learning of patterns in data.

# Why the Vanishing Gradient Problem Happens

- In a neural network, backpropagation computes the gradient of the loss function with respect to each weight by applying the chain rule across layers. If a network has many layers, this chain rule can cause the gradients to become very small or event halts the learning process for these layers.

# Reasons for vanishing gradient

- **Activation function** - Sigmoid and tanh activation functions are particularly prone to this issue because their gradients are always between 0 and 1. Overall gradient will be shrinked.

- **Weight initialization** - If weights are initialized too small, gradients become even smaller after each layer, leading to a rapid reduction in gradient values across layers.

# Solutions to vanishing gradient problem

- RELU Activation function and its variants
- Weight initialisation techniques - Xavier/He
- Batch Normalisation
- ResNets(Residual connections)

# Exploding gradient

- The *exploding gradient* problem occurs during the training of deep neural networks when the gradients—the values used to update the model's weights—become excessively large. This can happen in networks with many layers (deep networks) where repeated multiplications in backpropagation lead to values that increase exponentially.

- As a result, the gradients "explode," causing the model weights to update in very large steps, which can make the training process unstable and lead to issues like:

  The model parameters growing too large and destabilizing,

  Loss values becoming NaN (not a number), and

  The network failing to converge to a useful solution.

# Solution for exploding gradient

- **Gradient clipping** - It is achieved by limiting the maximum value of the gradient.

- LSTM other architectures are also implemented.

# Weight initialisation

- Weight initialization in neural networks is essential because it influences how the network learns during training, impacting convergence speed and overall performance. Proper initialization can help avoid issues like vanishing or exploding gradients, making it easier for the network to learn patterns from data.

# Weight initialisation methods

1. **Zero Initialization -** Sets all weights to zero.

- **Problem**: Causes neurons in each layer to learn the same features, resulting in a lack of diversity. The network cannot break symmetry, leading to poor learning.

**2. Random Initialization -** Initializes weights randomly, usually using small values from a uniform or normal distribution

- Helps break symmetry, allowing neurons to learn different features.
- **Problem**: In deep networks, random initialization can lead to vanishing or exploding gradients.

## 3. Xavier (Glorot) Initialization

- Sets the weights to values drawn from a distribution with a variance of $\frac{2}{n_{in}+n_{out}}$, where $n_{in}$ and $n_{out}$ are the numbers of input and output units for a given layer.
- Works well with sigmoid and tanh activation functions, helping keep gradients stable.
- **Formula:**
  - For normal distribution: $W \sim \mathcal{N}(0, \frac{2}{n_{in}+n_{out}})$
  - For uniform distribution: $W \sim U\left[-\sqrt{\frac{6}{n_{in}+n_{out}}}, \sqrt{\frac{6}{n_{in}+n_{out}}}\right]$

## 4. He Initialization

- Sets the weights based on the variance of $\frac{2}{n_{in}}$, where $n_{in}$ is the number of input units.
- Works well with ReLU and its variants because these activations tend to have non-saturating gradients, so the increased variance can help.
- **Formula**:
  - For normal distribution: $W \sim \mathcal{N}(0, \frac{2}{n_{in}})$
  - For uniform distribution: $W \sim U\left[-\sqrt{\frac{6}{n_{in}}}, \sqrt{\frac{6}{n_{in}}}\right]$

## Choosing the Right Initialization:

- **Sigmoid or tanh activations**: Try Xavier initialization to maintain balanced gradients.
- **ReLU or its variants**: He initialization often works well.
- **SELUs**: LeCun initialization is effective.

Correct weight initialization can reduce the likelihood of vanishing or exploding gradients, improve convergence speed, and lead to better final performance.

# Batch Normalization

- Batch Normalization (BatchNorm) is a technique used in deep learning to stabilize and accelerate the training of neural networks.

- it helps improve model performance by normalizing the inputs of each layer, which mitigates the internal covariate shift.

# Internal covariate shift

- Internal covariate shift - A phenomenon where the distribution of layer inputs changes during training. Here's a closer look at how it works and its benefits.

## How Batch Normalization Works

1. **Normalization**: For each mini-batch, BatchNorm calculates the mean and variance of the layer's inputs. It then normalizes the inputs by subtracting the batch mean and dividing by the batch standard deviation.

$$\hat{x} = \frac{x - \mu_{\text{batch}}}{\sqrt{\sigma^2_{\text{batch}} + \epsilon}}$$

Here:

- $x$ is the input to the layer.
- $\mu_{\text{batch}}$ is the mean of the inputs in the batch.
- $\sigma^2_{\text{batch}}$ is the variance of the inputs in the batch.
- $\epsilon$ is a small constant added for numerical stability.

2. **Scaling and Shifting**: BatchNorm then applies a scaling factor $\gamma$ and a shifting factor $\beta$, both learnable parameters, to allow the network to restore the ability to represent the identity function if needed:

$$y = \gamma \hat{x} + \beta$$

This transformation helps the model retain representational power, even after normalization.

## Benefits of Batch Normalization

1. **Improves Training Speed**: By reducing internal covariate shift, BatchNorm allows the model to converge faster, enabling the use of higher learning rates.
2. **Regularization Effect**: It acts as a form of regularization by adding noise to the model during training (due to batch-based statistics), which can reduce the need for dropout and other forms of regularization.
3. **Stabilizes Training**: BatchNorm makes training less sensitive to the initial choice of hyperparameters and helps avoid issues such as exploding and vanishing gradients.
4. **Increased Model Performance**: In practice, BatchNorm can lead to higher accuracy and improved model generalization on validation and test sets.

## When and Where to Apply Batch Normalization

- **CNNs**: Typically applied after the convolutional layer and before the activation function.
- **RNNs**: BatchNorm can be applied between time steps, although other techniques like Layer Normalization may work better in recurrent architectures.
- **Fully Connected Networks**: Can be applied after the linear layer and before the activation function.

## Key Points to Consider

- **Batch Size Sensitivity**: Performance of BatchNorm can degrade with very small batch sizes since batch statistics may become less representative.
- **Training vs. Inference**: During training, BatchNorm uses mini-batch statistics, while in inference it uses running averages of the mean and variance calculated during training.

# Effect of hyper parameters

- **Learning rate -**

  Controls the step size at each iteration while moving toward a minimum of the loss function.

  Impact: A high learning rate may lead to convergence to a suboptimal solution or oscillations, while a low learning rate may slow down training.

- **Batch Size -**

Refers to the number of training samples used in one forward/backward pass.

Impact: Larger batch sizes generally allow for more stable gradient estimates, but require more memory. Smaller batches can lead to noisy updates but may generalize better.

- **Number of epochs -**

Specifies the number of complete passes through the training dataset.

Impact: Too few epochs can lead to underfitting, while too many may cause overfitting, where the model memorizes the training data but doesn’t generalize well.

- ## Optimizer Choice (SGD,ADAM)-

  Determines the algorithm for updating model parameters.

  Impact: Different optimizers (SGD, Adam, RMSprop) have various strengths depending on the problem; for example, Adam generally converges faster, while SGD with momentum can provide better generalization.

- ## Weight initialisation-

  How initial values of model weights are set (e.g., Xavier, He initialization).

  Impact: Good initialization can help avoid vanishing or exploding gradients, leading to faster and more stable training.

- **Dropout Rate** -

  Randomly sets a fraction of input units to zero during training to prevent overfitting.

  Impact: Higher dropout rates can improve generalization by preventing the model from relying on specific neurons, but can make training slower and less stable.

- Regularization parameters
- Learning rate scheduler
- Early stopping
- Activation function

# Why we need optimizers

- Optimizers are essential in training machine learning models because they adjust the model's parameters (e.g., weights and biases) to minimize the loss function.

- **Efficient parameter adjustment** - Goal of training is to find the best parameters that minimize loss.
- **Handling large and complex models** - Optimizers are designed to work with these complex models, iteratively improving parameters to achieve high accuracy without manually intervening in the learning process.
- **Avoiding local minima** - Some optimizers, like stochastic gradient descent (SGD) with momentum, can help the model escape local minima—points where the loss is low but not the lowest possible—by adding randomness or momentum to the updates, allowing the model to find better solutions.

**Faster Convergence**: Optimizers like Adam and RMSProp adapt the learning rate, helping models converge more quickly and reliably.

**Handling Complex Loss Landscapes**: Optimizers are designed to navigate the non-convex, often complicated surfaces of loss functions, ensuring stable training.

# Motivation behind momentum

In standard SGD, each update to the model's weights is directly proportional to the gradient of the loss with respect to the weights. However, this can lead to:

- Oscillations: When the loss surface is steep in some directions and flat in others, gradients may fluctuate wildly along steep directions, causing the weights to oscillate and make slower progress.

- Slow convergence on flat surfaces: When the gradients are small or zero in flat regions, SGD might get "stuck" or move very slowly, delaying the model's progress toward convergence.

# Optimizers

- SGD With momentum
- NAG (Nesterov accelerated gradient)
- Adagrad(Adaptive gradient)
- RMS prop(Root Means Square Propagation)
- Adam(Adaptive Momentum Estimation)

# SGD with momentum

**Description**: Adds a "momentum" term to SGD to help accelerate learning by smoothing out oscillations along the path to the optimal point. It accumulates a fraction of the past gradients in the current update.

**Pros**: Faster convergence, helps to avoid getting stuck in local minima.

**Cons**: Requires careful tuning of momentum hyperparameter.

# Nesterov Accelerated Gradient

**Description**: A variation of momentum-based SGD that calculates the gradient at an approximated future position rather than the current position. This provides a more accurate update direction.

**Pros**: Faster and often achieves better performance than standard momentum.

**Cons**: Slightly more complex implementation.

# Adagrad(Adaptive gradient)

**Description**: Adagrad adapts the learning rate for each parameter based on the cumulative sum of past gradients. Parameters with frequent updates have a slower learning rate, while less-updated parameters keep a higher learning rate.

**Pros**: Works well for sparse data, no need to manually tune learning rates.

**Cons**: Accumulated gradients can cause the learning rate to decay too fast, which may stop learning early.

# RMS Prop(Root Mean Square Propagation)

**Description**: RMSprop addresses Adagrad’s rapid learning rate decay by maintaining a moving average of squared gradients to control the learning rate. This allows the model to continue learning at a steady rate.

**Pros**: Effective for non-stationary objectives and avoids rapid learning rate decay.

**Cons**: Requires tuning of decay rate hyperparameter.

# Adam(Adaptive moment estimation)

**Description**: Combines RMSprop and momentum by computing adaptive learning rates for each parameter. Adam maintains an exponentially decaying average of past gradients and squared gradients.

**Pros**: Often considered the default optimizer, adaptive learning rate and momentum handling make it efficient and widely applicable.

**Cons**: Can sometimes lead to suboptimal convergence; requires more memory.


The End