



Deep Learning

# Fundamentals of Neural Network

DEEP LEARNING

## Module No: 02

# Training, Optimization and Regularization of Deep Neural Network





### Forward propagation

- Forward propagation in deep learning refers to the process of passing input data through the neural network to get the output or prediction.
- It involves a series of computations, where the input data is transformed as it passes through the layers of the network.
- Process of passing the input forward through the network, involving weighted sums, biases, and activation functions, is forward propagation.
- The network learns the optimal weights and biases during the training phase to make accurate predictions.



### Simple Analogy

- Preparing a recipe (making predictions)
- Ingredients (Input)
- Ingredients importance or preference (Weights and Biases)
- Mixing Ingredients (Weighted Sums)
- Taste recipe (Activation Function)
- Final Dish as per desire (Output ; 0/1)



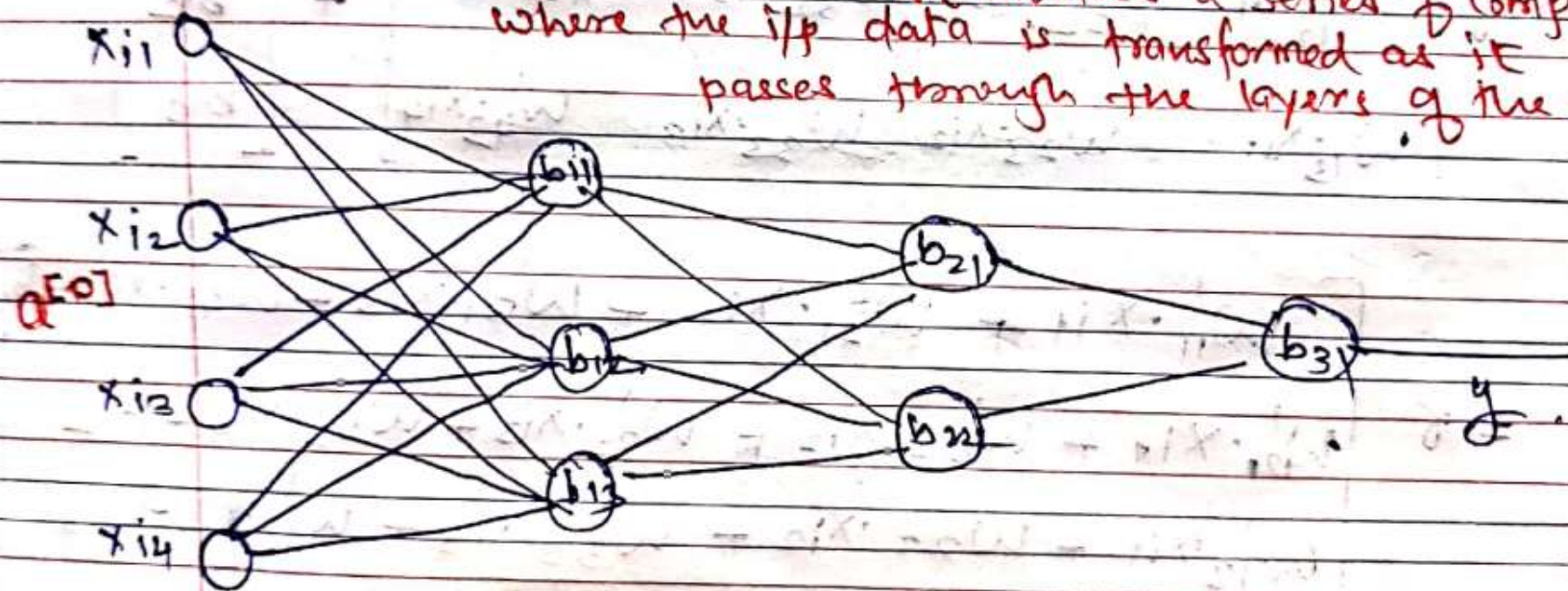
# Deep Learning

forward propagation.

organised way of doing mathematics.

Cgpa	iq	10 <sup>th</sup> Marks	12 Marks	Placed?
7.2	75	69	81	1
8.1	92	50	40	0

Forward propagation in DL refers to the process of passing i/p data through the NN to get the o/p or prediction. \* It involves a series of computations where the i/p data is transformed as it passes through the layers of the network.







# Deep Learning

Weight Matrix for layer 1:

$$\begin{bmatrix} w'_{11} & w'_{12} & w'_{13} \\ w'_{21} & w'_{22} & w'_{23} \\ w'_{31} & w'_{32} & w'_{33} \\ w'_{41} & w'_{42} & w'_{43} \end{bmatrix}$$

$$\text{prediction} = \sigma (w^T \cdot x + b)$$

for layer 1

$$= \sigma \begin{bmatrix} w'_{11} & w'_{21} & w'_{31} & w'_{41} \\ w'_{12} & w'_{22} & w'_{32} & w'_{42} \\ w'_{13} & w'_{23} & w'_{33} & w'_{43} \end{bmatrix} \cdot \begin{bmatrix} x_{i1} \\ x_{i2} \\ x_{i3} \\ x_{i4} \end{bmatrix} + \begin{bmatrix} b_{i1} \\ b_{i2} \\ b_{i3} \end{bmatrix}$$

$3 \times 4$        $4 \times 1$        $3 \times 1$





# Deep Learning

$$= \delta \begin{bmatrix} w'_{11} & w'_{21} & w'_{31} & w'_{41} \\ w'_{12} & w'_{22} & w'_{32} & w'_{42} \\ w'_{13} & w'_{23} & w'_{33} & w'_{43} \end{bmatrix} \cdot \begin{bmatrix} x_{i1} \\ x_{i2} \\ x_{i3} \\ x_{i4} \end{bmatrix} + \begin{bmatrix} b_{11} \\ b_{12} \\ b_{13} \end{bmatrix}$$

$3 \times 4 \quad \quad \quad 4 \times 1 \quad \quad \quad 3 \times 1$

$$= \delta \begin{bmatrix} w'_{11} \cdot x_{i1} & w'_{21} \cdot x_{i2} & w'_{31} \cdot x_{i3} & w'_{41} \cdot x_{i4} \\ w'_{12} \cdot x_{i1} & w'_{22} \cdot x_{i2} & w'_{32} \cdot x_{i3} & w'_{42} \cdot x_{i4} \\ w'_{13} \cdot x_{i1} & w'_{23} \cdot x_{i2} & w'_{33} \cdot x_{i3} & w'_{43} \cdot x_{i4} \end{bmatrix} + \begin{bmatrix} b_{11} \\ b_{12} \\ b_{13} \end{bmatrix}$$

$$= \delta \begin{bmatrix} w'_{11} \cdot x_{i1} + w'_{21} \cdot x_{i2} + w'_{31} \cdot x_{i3} + w'_{41} \cdot x_{i4} + b_{11} \\ w'_{12} \cdot x_{i1} + w'_{22} \cdot x_{i2} + w'_{32} \cdot x_{i3} + w'_{42} \cdot x_{i4} + b_{12} \\ w'_{13} \cdot x_{i1} + w'_{23} \cdot x_{i2} + w'_{33} \cdot x_{i3} + w'_{43} \cdot x_{i4} + b_{13} \end{bmatrix}$$





# Deep Learning

$$= \begin{bmatrix} 0_{11} \\ 0_{12} \\ 0_{13} \end{bmatrix}$$

.....  $q[1]$

Now weight matrix for layer #2

$$\begin{bmatrix} w_{11}^2 & w_{12}^2 \\ w_{21}^2 & w_{22}^2 \\ w_{31}^2 & w_{32}^2 \end{bmatrix}$$

$$\text{prediction for layer 2} = \delta W^T \cdot O + b$$

Scanned by CamScanner

Date:

YOUVA

$$= \delta \begin{bmatrix} w_{11}^2 & w_{21}^2 & w_{31}^2 \\ w_{12}^2 & w_{22}^2 & w_{32}^2 \end{bmatrix} \cdot \begin{bmatrix} 0_{11} \\ 0_{12} \\ 0_{13} \end{bmatrix} + \begin{bmatrix} b_{21} \\ b_{22} \\ \cancel{b_{23}} \end{bmatrix}$$





# Deep Learning

$$= \delta \begin{bmatrix} w_{11}^2 \cdot o_{11} & w_{21}^2 \cdot o_{12} & w_{31}^2 \cdot o_{13} \\ w_{12}^2 \cdot o_{11} & w_{22}^2 \cdot o_{12} & w_{32}^2 \cdot o_{13} \end{bmatrix} + \begin{bmatrix} b_{21} \\ b_{22} \end{bmatrix}$$

$$= \delta \begin{bmatrix} w_{11}^2 \cdot o_{11} + w_{21}^2 \cdot o_{12} + w_{31}^2 \cdot o_{13} + b_{21} \\ w_{12}^2 \cdot o_{11} + w_{22}^2 \cdot o_{12} + w_{32}^2 \cdot o_{13} + b_{22} \end{bmatrix}$$

$$= \begin{bmatrix} o_{21} \\ o_{22} \end{bmatrix} \quad \dots \dots \quad a[2]$$

Now Weight Matrix for layer 3

$$\begin{bmatrix} w_{11}^3 \\ w_{21}^3 \end{bmatrix}$$

prediction for layer 3 =  $\delta W^T \cdot O + b$ .



$$= \delta \begin{bmatrix} w_{11}^3 & w_{21}^3 \end{bmatrix} \begin{bmatrix} 0_{21} \\ 0_{22} \end{bmatrix} + [b_{13}]$$

$$= \delta \begin{bmatrix} w_{11}^3 \cdot 0_{21} & w_{21}^3 \cdot 0_{22} \end{bmatrix} + [b_{13}]$$

$$= \delta \begin{bmatrix} w_{11}^3 \cdot 0_{21} + w_{21}^3 \cdot 0_{22} + b_{13} \end{bmatrix}$$

$$= [0_{31}]$$

$a[3]$

$= y$

We will simplify above

$$a^{[1]} = \delta (a^{[0]} w^{[1]} + b^{[1]})$$

$$a^{[2]} = \delta (a^{[1]} w^{[2]} + b^{[2]})$$

$$a^{[3]} = \delta (a^{[2]} w^{[3]} + b^{[3]}) = y$$





## Loss Function

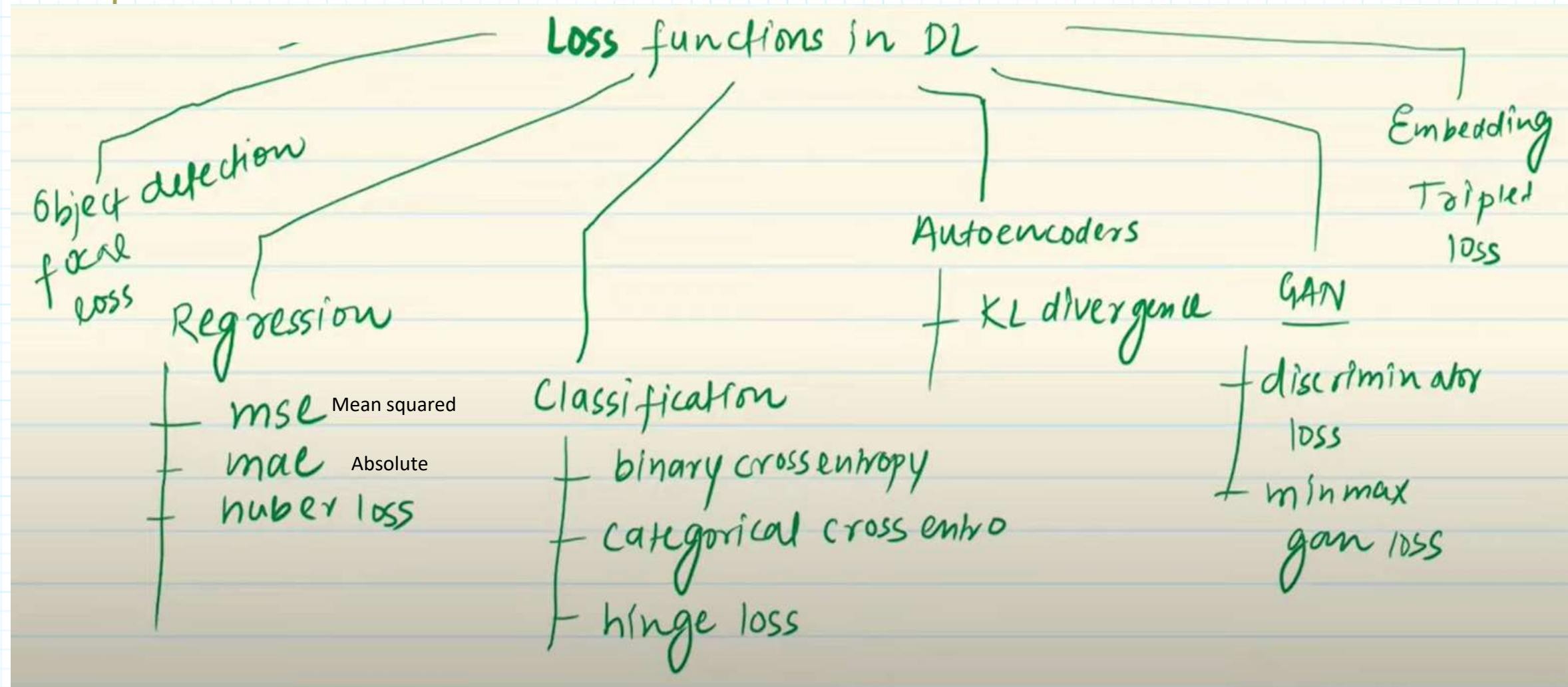
### **Loss function**

- In deep learning, a loss function is a measure of how well a model's predictions match the actual target values.
- The goal during the training of a model is to minimize this loss function.
- It quantifies the difference between predicted values and true values, providing a way to assess how well the model is performing.
- Different types of problems (classification, regression, etc.) and algorithms use different loss functions.





## Loss Function






## Loss Function

### Squared Error Loss (Mean Squared Error - MSE):

- **Use Case:** Typically used for regression problems, where the goal is to predict a continuous variable.
- **Calculation:** It calculates the average of the squared differences between the predicted and actual values.
- **Formula:**

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

$n$  is the number of data points,  $y_i$  is the actual value, and  $\hat{y}_i$  is the predicted value.



# Deep Learning

## Squared Error Loss (Mean Squared Error - MSE)

### Problem:

Suppose you have a dataset with actual values and predicted values as follows:

Actual Values ( $y$ ): [5, 10, 15, 20]

Predicted Values ( $\hat{y}$ ): [7, 9, 14, 18]



**Solution:**

**Calculate Squared Differences:**

$$(y_i - \hat{y}_i)^2$$

$$(5 - 7)^2 = 4$$

$$(10 - 9)^2 = 1$$


$$(15 - 14)^2 = 1$$

$$(20 - 18)^2 = 4$$

**Calculate Mean Squared Error (MSE):**

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

$$MSE = \frac{4+1+1+4}{4} = \frac{10}{4} = 2.5$$



# Deep Learning

## Squared Error Loss (Mean Squared Error - MSE)

Suppose you have a dataset with actual values and predicted values as follows:

- True values:  $[5, 10, 15]$
- Predicted values:  $[7, 8, 13]$

Now, we calculate the squared differences for each pair:

$$(5 - 7)^2 = 4$$

$$(10 - 8)^2 = 4$$

$$(15 - 13)^2 = 4$$

Then, we take the average:

$$\text{MSE} = \frac{1}{3} \times (4 + 4 + 4) = \frac{12}{3} = 4$$



## Cross Entropy Loss (Binary Cross Entropy and Categorical Cross Entropy)

### Binary Cross Entropy Loss:

- **Use Case:** Commonly used for classification problems.
- **Calculation:** For binary classification problems (two classes).
- **Formula:**

$$BCE = -\frac{1}{n} \sum_{i=1}^n [y_i \cdot \log(\hat{y}_i) + (1 - y_i) \cdot \log(1 - \hat{y}_i)]$$

$n$  is the number of data points,  $y_i$  is the true label (0 or 1),  $\hat{y}_i$  is the predicted probability.





## Cross Entropy Loss (Binary Cross Entropy and Categorical Cross Entropy)

### Binary Cross Entropy Loss:

consider a binary classification problem where the true class label (ground truth) is 1, and the model prediction is 0.8.

Here, Binary Cross Entropy (BCE) Loss is calculated as:

$$\text{BCE Loss} = -[y \log(p) + (1 - y) \log(1 - p)]$$

Where:

- $y$  is the true label (1 for the positive class, 0 for the negative class).
- $p$  is the predicted probability of the positive class.



## Cross Entropy Loss (Binary Cross Entropy and Categorical Cross Entropy)

### Binary Cross Entropy Loss:

Let's substitute the values into the formula:

$$\text{BCE Loss} = -[1 \cdot \log(0.8) + (1 - 1) \cdot \log(1 - 0.8)]$$

$$\text{BCE Loss} = -[\log(0.8)]$$

Now, calculate the numerical value:

$$\text{BCE Loss} = -[-0.09691]$$

$$\text{BCE Loss} \approx 0.09691$$



## Cross Entropy Loss (Binary Cross Entropy and Categorical Cross Entropy)

### Exercise:

Suppose the true label (ground truth) is 0 (negative class), and the model predicts a probability of 0.3 for the positive class.

Let's substitute the values into the formula:

$$\text{BCE Loss} = -[0 \cdot \log(0.3) + (1 - 0) \cdot \log(1 - 0.3)]$$

$$\text{BCE Loss} = -[\log(0.7)]$$

Now, calculate the numerical value:

$$\text{BCE Loss} = -[-0.1549]$$

$$\text{BCE Loss} \approx 0.1549$$





## Cross Entropy Loss (Binary Cross Entropy and Categorical Cross Entropy)

### Categorical Cross Entropy Loss:

- **Use Case:** Commonly used for classification problems.
- **Calculation:** For multi-class classification problems (more than two classes).
- **Formula**

$$CCE = -\frac{1}{n} \sum_{i=1}^n \sum_{j=1}^m y_{i,j} \cdot \log(\hat{y}_{i,j})$$

$n$  is the number of data points,  $m$  is the number of classes,  $y_{i,j}$  is the true label,  $\hat{y}_{i,j}$  is the predicted probability for class  $j$ .



## Cross Entropy Loss (Binary Cross Entropy and Categorical Cross Entropy)

### **Categorical Cross Entropy Loss:**

consider a scenario where we have a classification problem with three classes (C1, C2, and C3). We have a set of true class probabilities and predicted class probabilities for each instance. The Categorical Cross-Entropy Loss is commonly used in such multi-class classification problems.

Suppose we have three instances, and the true class probabilities and predicted class probabilities are as follows:



## Cross Entropy Loss (Binary Cross Entropy and Categorical Cross Entropy)

Instance 1:

- True class probabilities:  $[1, 0, 0]$  (belongs to C1)
- Predicted class probabilities:  $[0.8, 0.15, 0.05]$

Instance 2:

- True class probabilities:  $[0, 1, 0]$  (belongs to C2)
- Predicted class probabilities:  $[0.2, 0.7, 0.1]$

Instance 3:

- True class probabilities:  $[0, 0, 1]$  (belongs to C3)
- Predicted class probabilities:  $[0.05, 0.1, 0.85]$





## Cross Entropy Loss (Binary Cross Entropy and Categorical Cross Entropy)

The Categorical Cross-Entropy Loss for each instance is calculated as:

$$L_i = - \sum_j y_{ij} \cdot \log(\hat{y}_{ij})$$

Where:

- $y_{ij}$  is the true probability of class  $j$  for instance  $i$ .
- $\hat{y}_{ij}$  is the predicted probability of class  $j$  for instance  $i$ .

Let's calculate the loss for each instance:



## Cross Entropy Loss (Binary Cross Entropy and Categorical Cross Entropy)

**Instance 1:**

$$L_1 = -(1 \cdot \log(0.8) + 0 \cdot \log(0.15) + 0 \cdot \log(0.05))$$

**Instance 2:**

$$L_2 = -(0 \cdot \log(0.2) + 1 \cdot \log(0.7) + 0 \cdot \log(0.1))$$

**Instance 3:**

$$L_3 = -(0 \cdot \log(0.05) + 0 \cdot \log(0.1) + 1 \cdot \log(0.85))$$

Now, we can calculate the average loss over all instances:

$$\text{Average Loss} = \frac{L_1 + L_2 + L_3}{3}$$



## Cross Entropy Loss (Binary Cross Entropy and Categorical Cross Entropy)

**Instance 1:**

$$L_1 = -(1 \cdot \log(0.8) + 0 \cdot \log(0.15) + 0 \cdot \log(0.05))$$

**Instance 2:**

$$L_2 = -(0 \cdot \log(0.2) + 1 \cdot \log(0.7) + 0 \cdot \log(0.1))$$

**Instance 3:**

$$L_3 = -(0 \cdot \log(0.05) + 0 \cdot \log(0.1) + 1 \cdot \log(0.85))$$

Now, we can calculate the average loss over all instances:

$$\text{Average Loss} = \frac{L_1 + L_2 + L_3}{3}$$

This average loss gives us an indication of how well the predicted probabilities match the true class probabilities across all instances.

Lower values indicate better performance.





# What is activation function?

- The activation function decides whether a neuron should be activated or not by calculating the weighted sum and further adding bias to it. The purpose of the activation function is to introduce non-linearity into the output of a neuron.
- In artificial neural networks, an activation function is one that outputs a smaller value for tiny inputs and a higher value if its inputs are greater than a threshold. An activation function "fires" if the inputs are big enough; otherwise, nothing happens.
- An activation function, then, is a gate that verifies how an incoming value is higher than a threshold value.
- The activation function is a fundamental component of neural networks that introduces non-linearity, enabling them to learn complex relationships, adapt to various data patterns, and make sophisticated decisions.

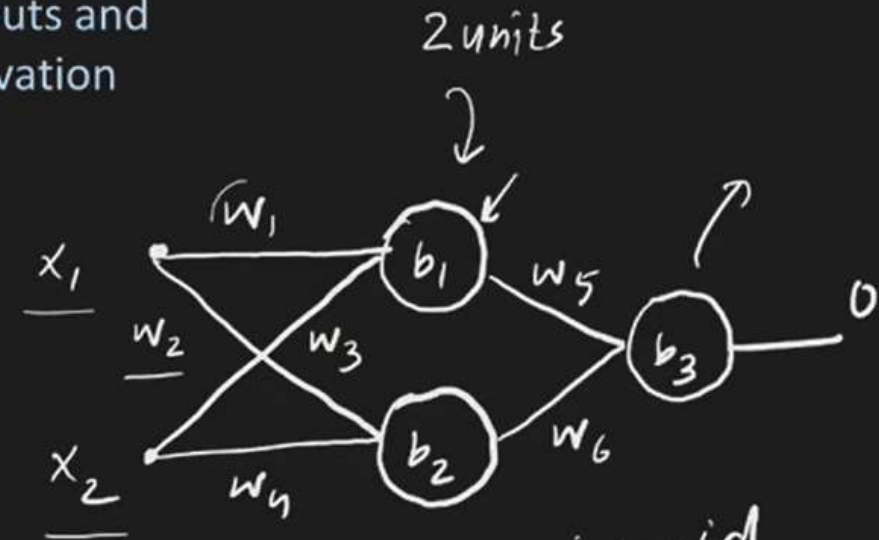
# What are Activation Functions?

31 May 2022 14:49

In artificial neural networks, each neuron forms a weighted sum of its inputs and passes the resulting scalar value through a function referred to as an activation function or transfer function. If a neuron has  $n$  inputs then the output or activation of a neuron is

$$a = g(w_1x_1 + w_2x_2 + w_3x_3 + \dots w_nx_n + b)$$

This function  $g$  is referred to as the activation function.



$$g(w_1x_1 + w_2x_2 + b_1)$$

activation

sigmoid  
relu  
softmax

+ Text

✓ RAM  
Disk

```
from tensorflow.keras.layers import Dense  
from tensorflow.keras.layers import Dropout  
from tensorflow.keras.optimizers import Adam
```

```
model = Sequential()
```

```
model.add(Dense(128, input_dim=2, activation="linear"))
```

```
model.add(Dense(128, activation="linear"))
```

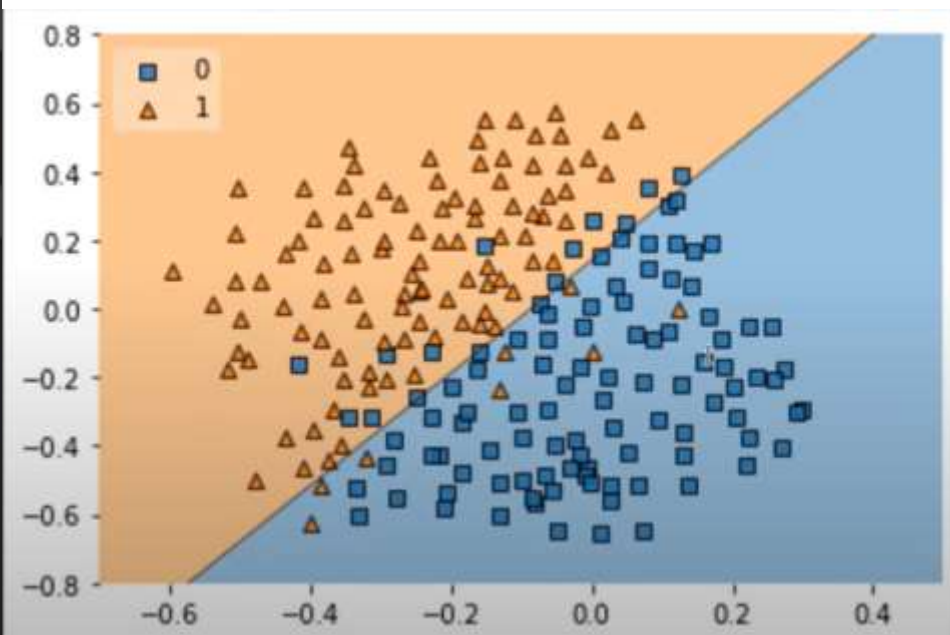
```
model.add(Dense(1, activation="sigmoid"))
```

```
adam = Adam(learning_rate=0.01)
```

```
model.compile(loss='binary_crossentropy', optimizer=adam, metrics=['accuracy'])
```

```
history = model.fit(X, y, epochs=500, validation_split = 0.2, verbose=0)
```

```
from mlxtend.plotting import plot_decision_regions
```





```
model = Sequential()
```

```
model.add(Dense(128, input_dim=2, activation="relu"))
```

```
model.add(Dense(128, activation="relu"))
```

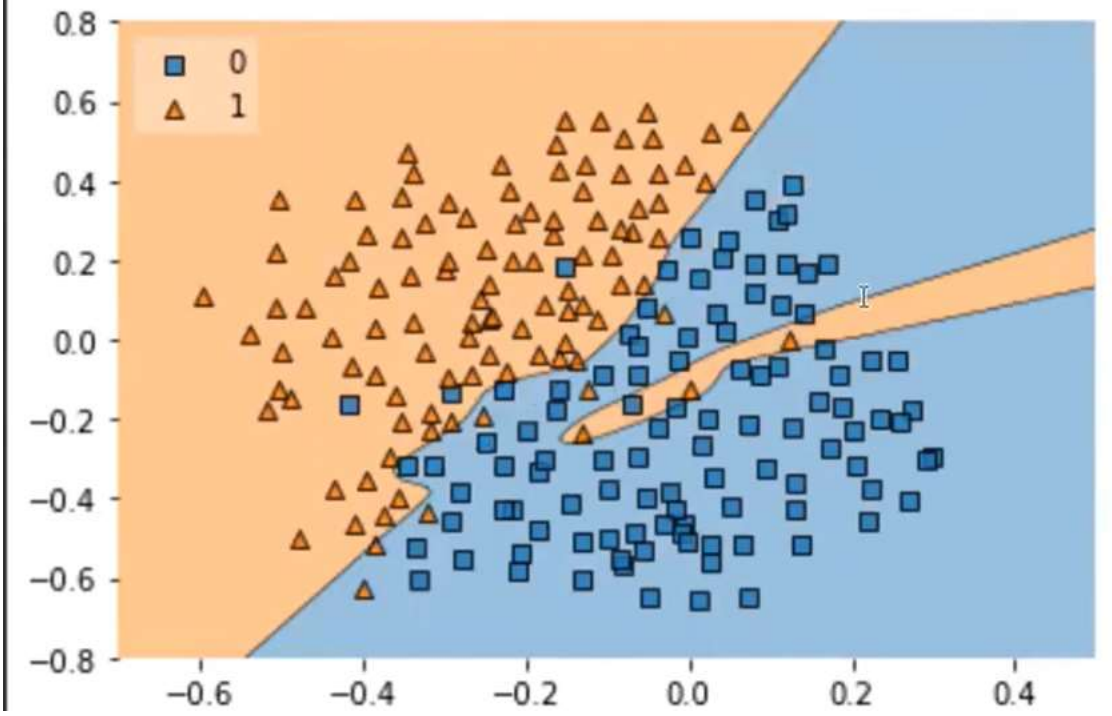
```
model.add(Dense(1, activation="sigmoid"))
```

```
adam = Adam(learning_rate=0.01)
```

```
model.compile(loss='binary_crossentropy', optimizer=adam, metrics=['accuracy'])
```

```
history = model.fit(X, y, epochs=500, validation_split = 0.2, verbose=0)
```

```
from mlxtend.plotting import plot_decision_regions
```





# Why there is a need of activation function?

### Introducing Non-linearity:

- Without activation functions, the entire neural network would behave like a linear model.
- The stacking of multiple linear operations would result in a linear combination, limiting the network's ability to learn and represent complex, non-linear patterns in the data.

### Capturing Complex Relationships:

- Many real-world problems involve intricate and non-linear relationships.
- Activation functions allow the neural network to model and capture these complex patterns, making it more powerful in representing diverse data.

### Enabling Neural Network to Learn:

- The non-linear transformations introduced by activation functions enable the network to learn and adapt to intricate patterns in the input data during the training process. This is crucial for the network to generalize well to unseen data.



# Why there is a need of activation function?

### Thresholding and Output Scaling:

- Activation functions often introduce thresholding effects, where the neuron activates or not based on certain conditions.
- This helps in decision-making and provides a level of abstraction. Additionally, activation functions like sigmoid and softmax scale the output to represent probabilities in classification tasks.

### Avoiding Vanishing or Exploding Gradients:

- Activation functions play a role in mitigating issues like vanishing or exploding gradients during backpropagation, especially in deep neural networks.
- Well-designed activation functions help in the stable training of deep networks.



# Why there is a need of activation function?

### Introducing Sparsity:

- Some activation functions, like ReLU (Rectified Linear Unit) and its variants, introduce sparsity in the network by setting negative values to zero.
- This can be beneficial in certain scenarios.

### Facilitating Backpropagation:

- Activation functions provide derivatives or gradients that are essential for the backpropagation algorithm, which is used to update the weights of the network during training.
- This enables the network to learn and improve its performance over time.





# Types of activation function

In a perceptron or a neural network, activation functions play a crucial role by introducing non-linearity to the model.

Here are some common types of activation functions used in perceptrons:

1. Linear activation function
2. Logistic activation function
3. Tanh activation function
4. Softmax activation function
5. ReLU activation function
6. Leaky ReLU activation function



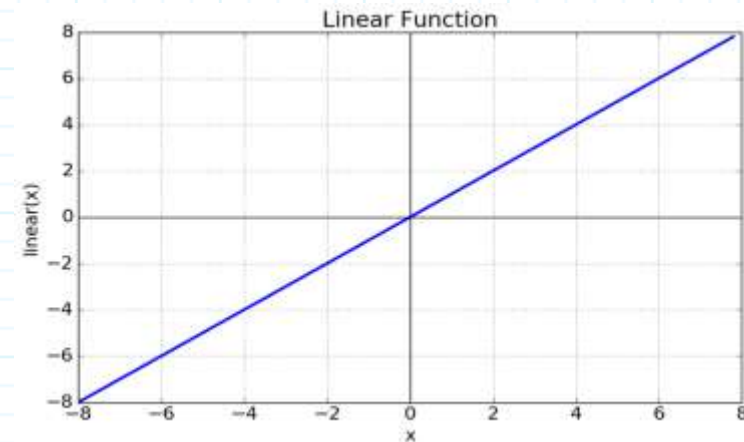
## Types of activation function

### 1. Linear Function:

- **Description:** The linear activation function, also known as the identity activation function, is a straightforward and simple function. It is defined as:

- **Mathematical Form:**

$$f(x) = x$$



- **Advantages:**

- **Simplicity**

- **Ease of Interpretation:**

- Direct proportionality between input and output.
- Straightforward interpretation.

- **Compatibility with Linear Models** (Well-suited for tasks with linear relationships)



# Types of activation function

### Disadvantages:

- **Limited Expressiveness:**
  - Inability to model complex, non-linear relationships.
  - Stacking linear layers results in a linear model.
- **Vanishing Gradient Problem:**
  - Prone to vanishing gradients, especially in deep networks.
  - May lead to slow learning.
- **Not Suitable for Classification Problems:**
  - Challenging for binary classification tasks.
  - Output not squashed into a specific range.
- **Not Used in Hidden Layers of Deep Networks:**
  - Rarely used in deep networks' hidden layers.
  - Non-linear activations preferred.

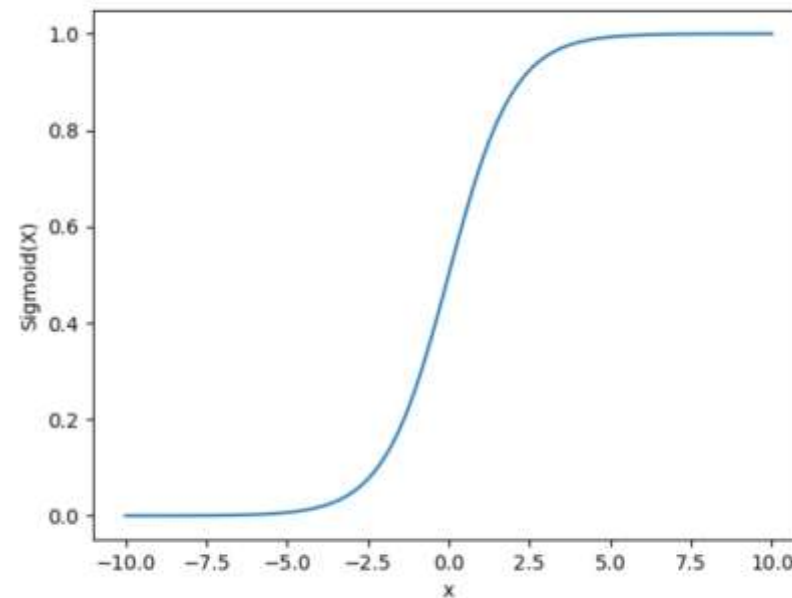


## Types of activation function

### 2. Logistic Activation Function :

- is also commonly referred to as the Sigmoid Activation Function.
- **Description:** The sigmoid (logistic) function squashes input values to the range (0, 1). It is commonly used in the output layer of binary classification models.
- **Mathematical Form:**

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







# Types of activation function

- It is a function which is plotted as 'S' shaped graph.
- **Nature** : Non-linear. Notice that X values lies between -2 to 2, Y values are very steep. This means, small changes in x would also bring about large changes in the value of Y.
- **Value Range** : 0 to 1
- **Uses** : Usually used in output layer of a binary classification, where result is either 0 or 1, as value for sigmoid function lies between 0 and 1 only so, result can be predicted easily to be 1 if value is greater than 0.5 and 0 otherwise.



## Types of activation function

### Question:

Calculate the output of a neuron with the given parameters using the sigmoid activation function:

- $x_1 = 0.7$
- $x_2 = -0.3$
- $w_1 = 0.5$
- $w_2 = -1$
- $b = 0.2$



## Types of activation function

Given  $x_1 = 0.7$ ,  $x_2 = -0.3$ ,  $w_1 = 0.5$ ,  $w_2 = -1$ , and  $b = 0.2$ , the weighted sum  $z$  is calculated as:

$$z = (0.5 \cdot 0.7) + ((-1) \cdot (-0.3)) + 0.2$$

Simplifying this:

$$z = 0.35 + 0.3 + 0.2 = 0.85$$

Now, applying the sigmoid activation function:

$$\sigma(0.85) = \frac{1}{1+e^{-0.85}}$$

Calculating this:

$$\sigma(0.85) \approx \frac{1}{1+e^{-0.85}} \approx \frac{1}{1+0.427} \approx \frac{1}{1.427} \approx 0.7$$

So, the output of the neuron for the given inputs, weights, and bias is approximately 0.7.



## Types of activation function

### Problem:

Calculate the output of a neuron with the given data using the sigmoid activation function:

- $x_1 = -0.6$
- $x_2 = 0.4$
- $w_1 = 1.2$
- $w_2 = -0.8$
- $b = 0.5$





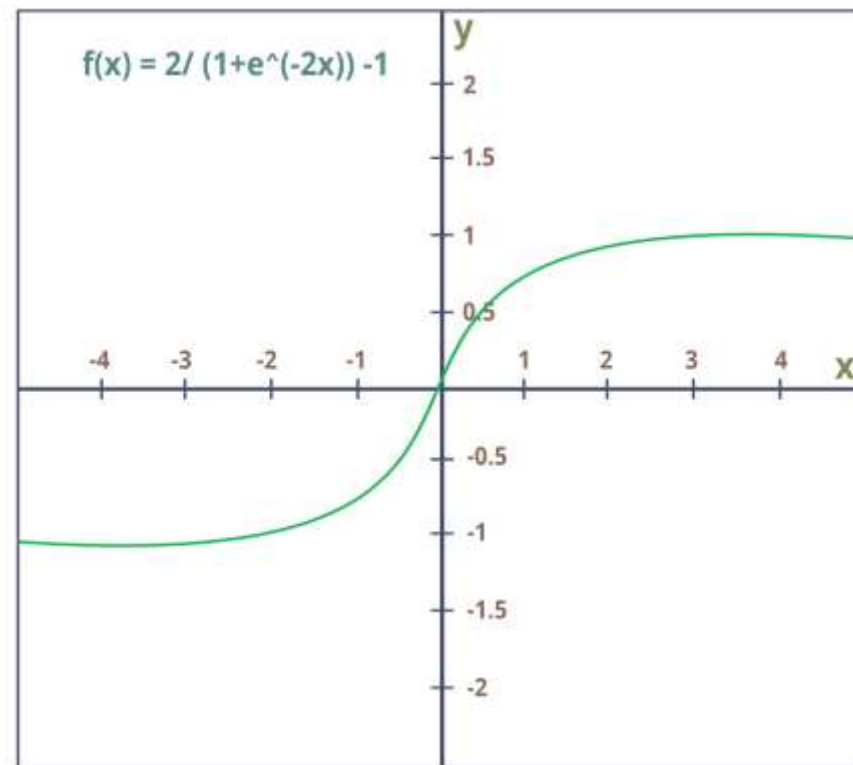
## Types of activation function

### 3. Tanh (Hyperbolic Tangent) Function:

- **Description:** Similar to the sigmoid function, the tanh function maps input values to the range  $(-1, 1)$ . It is often used in hidden layers of neural networks.
- **Mathematical Form:**

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

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





# Types of activation function

- The activation that works almost always better than sigmoid function is Tanh function also known as Tangent Hyperbolic function. It is actually mathematically shifted version of the sigmoid function. Both are similar and can be derived from each other.
- **Value Range** : -1 to +1
- **Nature** :- non-linear
- **Uses** :- Usually used in hidden layers of a neural network as its values lie between -1 to 1 hence the mean for the hidden layer comes out to be 0 or very close to it, hence helps in centering the data by bringing mean close to 0. This makes learning for the next layer much easier.



## Types of activation function

### Problem:

Calculate the output of a neuron with the given data using the hyperbolic tangent (tanh) activation function:

- $x_1 = 0.8$
- $x_2 = -0.2$
- $w_1 = 0.6$
- $w_2 = 1.1$
- $b = -0.4$



## Types of activation function

**Solution:**

1. Calculate the weighted sum ( $z$ ):

$$z = (0.6 \cdot 0.8) + (1.1 \cdot -0.2) - 0.4$$

$$z = 0.48 - 0.22 - 0.4$$

$$z = -0.14 - 0.4$$

$$z = -0.54$$

2. Apply the hyperbolic tangent ( $\tanh$ ) activation function:

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

Substitute  $z = -0.54$  into the  $\tanh$  function:

$$\tanh(-0.54) = \frac{e^{-0.54} - e^{0.54}}{e^{-0.54} + e^{0.54}}$$

Calculating the numerator and denominator separately:

$$\tanh(-0.54) = \frac{0.5806 - 1.7147}{0.5806 + 1.7147}$$

$$\tanh(-0.54) = \frac{-1.1341}{2.2953}$$

The final result is:

$$\tanh(-0.54) \approx -0.4929$$





## Types of activation function

### Problem:

Calculate the output of a neuron using the hyperbolic tangent (tanh) activation function for the following data:

$$x_1 = -0.6, \quad x_2 = 0.2, \quad w_1 = 0.8, \quad w_2 = -1.3, \quad b = 0.5$$



## Types of activation function

1. Calculate the weighted sum ( $z$ ):

$$z = (-0.6 \cdot 0.8) + (0.2 \cdot -1.3) + 0.5$$

$$z = -0.48 - 0.26 + 0.5$$

$$z = -0.74 + 0.5$$

$$z = -0.24$$

2. Apply the hyperbolic tangent ( $\tanh$ ) activation function:

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

Substitute  $z = -0.24$  into the  $\tanh$  function:

$$\tanh(-0.24) = \frac{e^{-0.24} - e^{0.24}}{e^{-0.24} + e^{0.24}}$$

Calculating the numerator and denominator separately:

$$\tanh(-0.24) = \frac{0.7878 - 1.2689}{0.7878 + 1.2689}$$

$$\tanh(-0.24) = \frac{-0.4811}{2.0567}$$

The final result is:

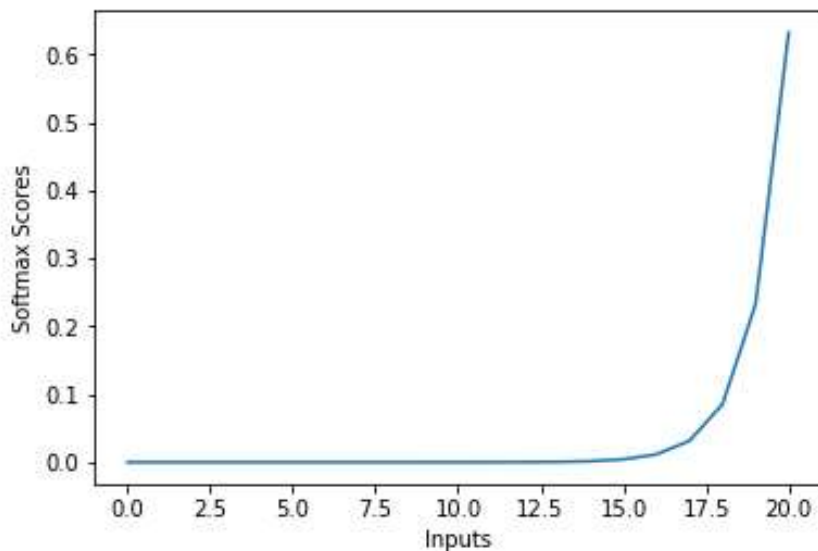
$$\tanh(-0.24) \approx -0.2345$$



## Types of activation function

### 4. Softmax Function:

- **Description:** Often used in the output layer of a neural network for multi-class classification problems. It transforms the raw output scores (logits) into a probability distribution over multiple classes. The Softmax function is particularly useful when dealing with problems where an input can belong to one of several exclusive classes.
- **Mathematical Form:**



For a given input vector  $\mathbf{z}$  with  $K$  elements (where  $K$  is the number of classes), the Softmax function is defined as follows:

$$\text{Softmax}(\mathbf{z})_i = \frac{e^{z_i}}{\sum_{j=1}^K e^{z_j}}, \quad \text{for } i = 1, 2, \dots, K$$

Here:

- $\text{Softmax}(\mathbf{z})_i$  is the  $i$ -th element of the Softmax output.
- $e^{z_i}$  is the exponential of the  $i$ -th element of the input vector.
- The denominator is the sum of the exponentials of all elements in the input vector.



# Types of activation function

- The softmax function is also a type of sigmoid function but is handy when we are trying to handle multi- class classification problems.
- Nature :- non-linear
- Uses :- Usually used when trying to handle multiple classes. the softmax function was commonly found in the output layer of image classification problems. The softmax function would squeeze the outputs for each class between 0 and 1 and would also divide by the sum of the outputs.
- If your output is for binary classification then, sigmoid function is very natural choice for output layer.
- If your output is for multi-class classification then, Softmax is very useful to predict the probabilities of each classes.





## Types of activation function

### Problem:

Calculate the output probabilities of a neural network using the Softmax activation function for the following data:

$$x_1 = 1.2, \quad x_2 = -0.5, \quad w_1 = 1.5, \quad w_2 = -2.3, \quad w_3 = 1.3, \quad b = 0.5$$



## Types of activation function

1. Calculate the weighted sum:

$$z_1 = (1.2 \times 1.5) + (-0.5 \times -2.3) + 0.5$$

$$z_2 = (1.2 \times 2.3) + (-0.5 \times 3.1) + 0.5$$

$$z_3 = (1.2 \times -1.3) + (-0.5 \times 2.1) + 0.5$$

Simplifying these equations:

$$z_1 = 1.8 + 1.15 + 0.5 \approx 3.45$$

$$z_2 = 2.76 + 1.55 + 0.5 \approx 4.81$$

$$z_3 = -1.56 - 1.05 + 0.5 \approx -2.11$$



## Types of activation function

### 2. Apply the Softmax function:

- Calculate the unnormalized exponentials:

$$e^{3.45} \approx 31.91, \quad e^{4.81} \approx 121.46, \quad e^{-2.11} \approx 0.123$$

- Calculate the sum of the exponentials:

$$\text{Sum} = 31.91 + 121.46 + 0.123 \approx 153.493$$

- Calculate the probabilities:

$$P(\text{class}_1) = \frac{31.91}{153.493} \approx 0.208$$

$$P(\text{class}_2) = \frac{121.46}{153.493} \approx 0.791$$

$$P(\text{class}_3) = \frac{0.123}{153.493} \approx 0.001$$

### 3. Final Output:

- The calculated probabilities for each class using the Softmax function are approximately:

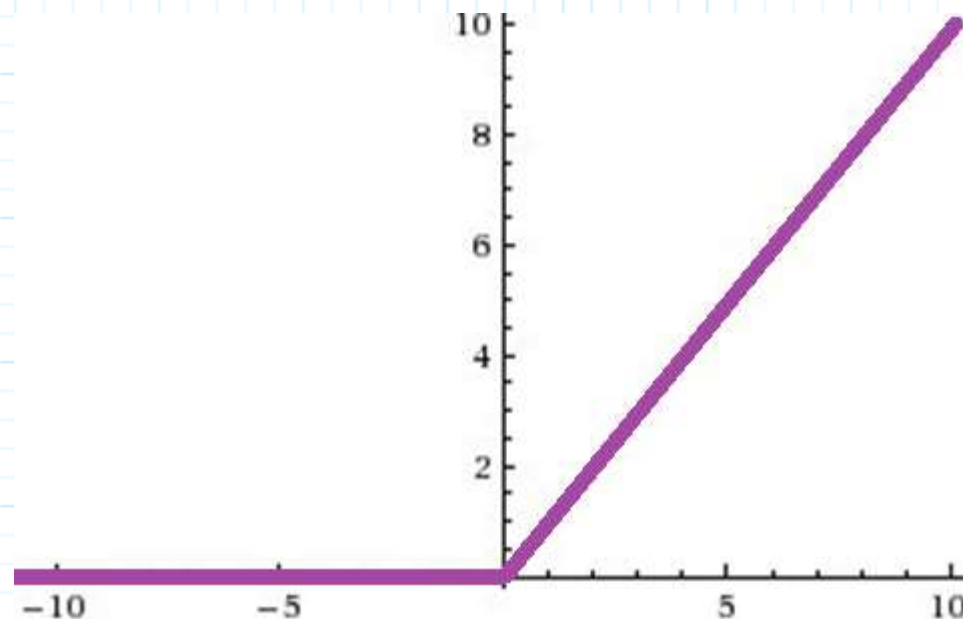
$$P(\text{class}_1) \approx 0.208, \quad P(\text{class}_2) \approx 0.791, \quad P(\text{class}_3) \approx 0.001$$



## Types of activation function

### 5. Rectified Linear Unit (ReLU):

- **Description:** ReLU is a popular activation function that outputs the input for positive values and zero for negative values. It introduces non-linearity and is computationally efficient.
- **Mathematical Form:**  $f(x) = \max(0, x)$





# Types of activation function

- It Stands for Rectified linear unit. It is the most widely used activation function. Chiefly implemented in hidden layers of Neural network.
- **Value Range** :-  $[0, \infty)$
- **Nature** :- non-linear, which means we can easily backpropagate the errors and have multiple layers of neurons being activated by the ReLU function.
- **Uses** :- ReLu is less computationally expensive than tanh and sigmoid because it involves simpler mathematical operations. At a time only a few neurons are activated making the network sparse making it efficient and easy for computation.





## Types of activation function

### Problem:

Calculate the output of a neuron using the Rectified Linear Unit (ReLU) activation function for the following data:

$$x_1 = 0.9, \quad x_2 = -0.5, \quad w_1 = 0.6, \quad w_2 = 0.8, \quad b = -0.2$$



## Types of activation function

1. Calculate the weighted sum ( $z$ ):

$$z = (0.9 \cdot 0.6) + (-0.5 \cdot 0.8) - 0.2$$

$$z = 0.54 - 0.4 - 0.2$$

$$z = -0.06$$

2. Apply the ReLU activation function:

$$\text{ReLU}(z) = \max(0, z)$$

Substitute  $z = -0.06$  into the ReLU function:

$$\text{ReLU}(-0.06) = \max(0, -0.06)$$

Since  $-0.06$  is less than zero, the ReLU function returns zero:

$$\text{ReLU}(-0.06) = 0$$

3. Final Output:

- The calculated weighted sum ( $z$ ) is approximately -0.06.
- The output after applying the ReLU function is 0.



## Types of activation function

### Problem:

Calculate the output of a neuron using the Rectified Linear Unit (ReLU) activation function for the following data:

$$x_1 = 0.6, \quad x_2 = 0.3, \quad w_1 = 0.8, \quad w_2 = -0.5, \quad b = 0.2$$



## Types of activation function

1. Calculate the weighted sum ( $z$ ):

$$z = (0.6 \cdot 0.8) + (0.3 \cdot (-0.5)) + 0.2$$

$$z = 0.48 - 0.15 + 0.2$$

$$z = 0.53$$

2. Apply the ReLU activation function:

$$\text{ReLU}(z) = \max(0, z)$$

Substitute  $z = 0.53$  into the ReLU function:

$$\text{ReLU}(0.53) = \max(0, 0.53)$$

Since 0.53 is greater than zero, the ReLU function returns the input value:

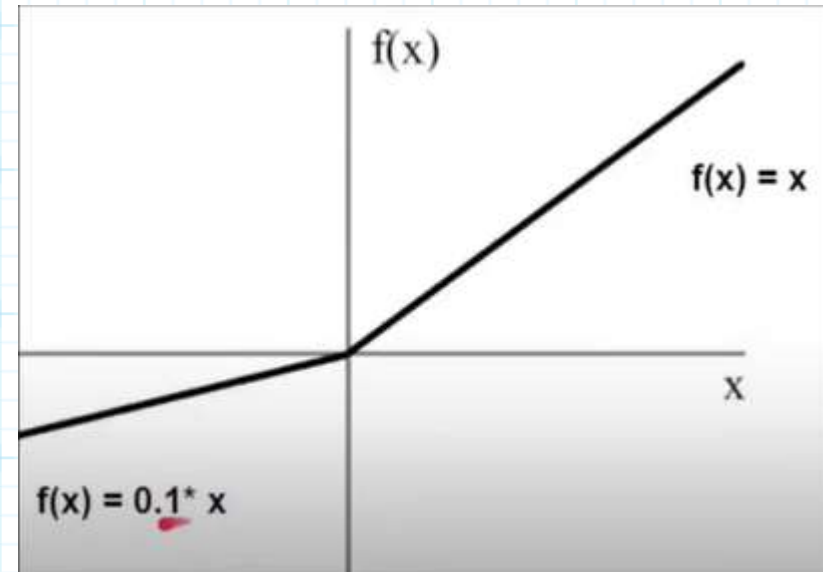
$$\text{ReLU}(0.53) = 0.53$$



## Types of activation function

### 5. Leaky ReLU (Rectified Linear Unit) Function:

- **Description:** Leaky ReLU is an activation function used in artificial neural networks to introduce nonlinearity among the outputs between layers of a neural network. This activation function was created to solve the dying ReLU problem using the standard ReLU function that makes the neural network die during training.
- **Mathematical Form:**



$$\text{LeakyReLU}(x) = \max(\alpha * x, x)$$

In the equation, we see that for every parameter  $x$ , the Leaky ReLU function will return the maximum value between  $x$  or  $\alpha * x$ , where  $\alpha$  is a small positive constant.





# Types of activation function

- Using this function, we can convert negative values to make them close to 0 but not actually 0, solving the dying ReLU issue that arises from using the standard ReLU function during neural network training.
- The Leaky ReLU is a popular activation function that is used to address the limitations of the standard ReLU function in deep neural networks by introducing a small negative slope for negative function inputs, which helps neural networks to maintain better information flow both during its training and after.



# Types of activation function

In a perceptron or a neural network, activation functions play a crucial role by introducing non-linearity to the model.

Here are some common types of activation functions used in perceptrons:

1. Linear activation function
2. Logistic activation function
3. Tanh activation function
4. Softmax activation function
5. ReLU activation function
6. Leaky ReLU activation function



# Backpropagation

- Backpropagation is a supervised learning algorithm used to train artificial neural networks.
- In backpropagation the neural networks adjust weights and biases to minimize the error between predicted and actual outputs.
- Backpropagation aims to minimize the difference between predicted and actual outputs.
- It uses the gradient of the error with respect to weights to iteratively adjust weights for better predictions.
- During the forward pass, input data is fed through the neural network to produce predictions.(the process of passing input data through the neural network to get the prediction)
- The loss function quantifies the error between predicted and actual outputs.
- In backpropagation we try to minimize the loss to improve the model's accuracy.
- Iterate through forward and backward passes until the model converges and Set criteria for stopping iterations.



## Gradient

Gradient has multiple definitions:

### Slope

The degree to which something inclines. For example, a mountain road with a 10% gradient rises one foot for every ten feet of horizontal length.

---

### Change rate

The rate at which a physical quantity, such as temperature or pressure, changes over a distance.

---

### Computer definition

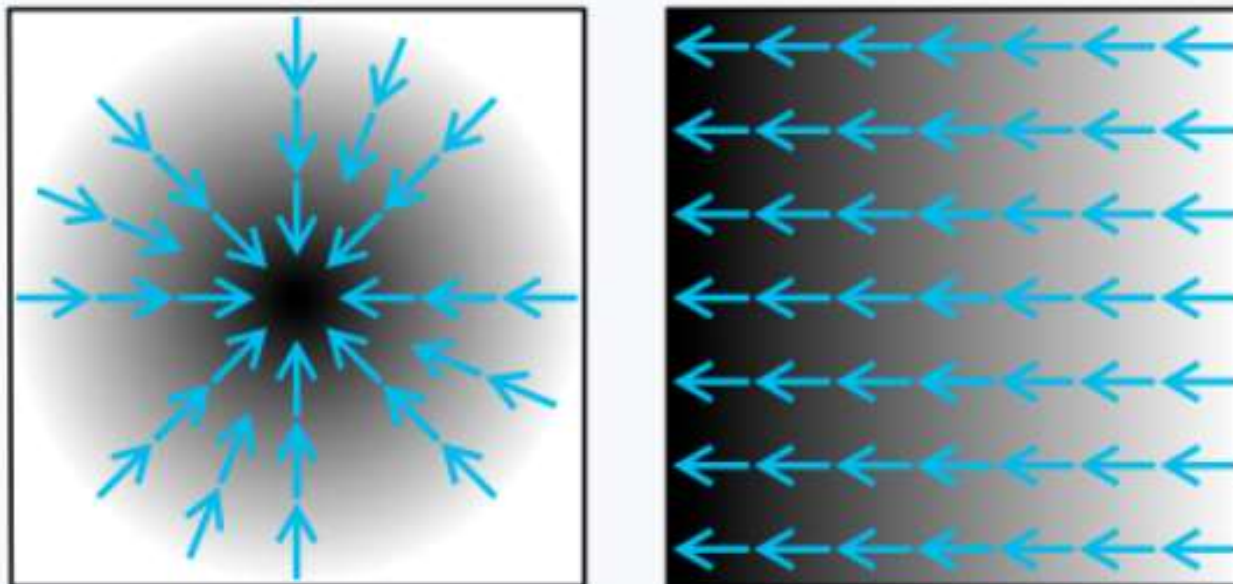
A smooth blending of shades from light to dark or from one color to another. In 2D drawing programs and paint programs, gradients are used to create colorful backgrounds and special effects as well as to simulate lights and shadows.

Gradient can also be defined as a single value that describes if a line is increasing (has positive gradient) or decreasing (has negative gradient).

Gradients can be calculated by dividing the vertical height by the horizontal distance.



## Gradient



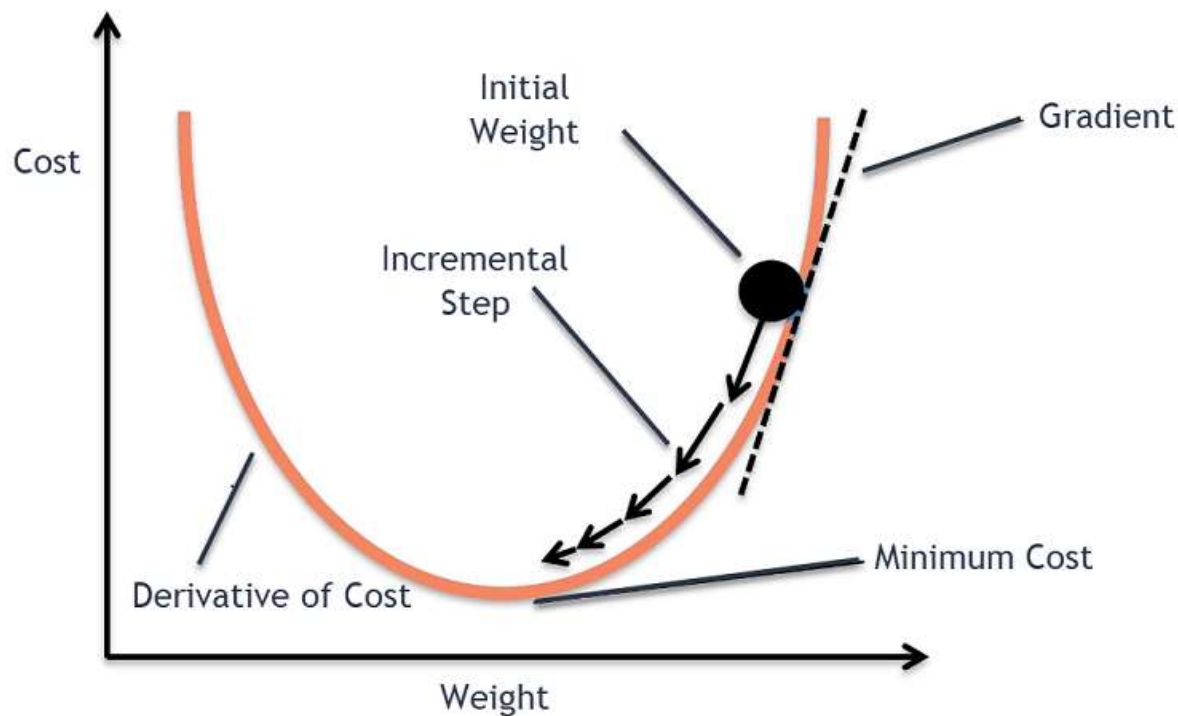
The gradient, represented by the blue arrows, denotes the direction of greatest change of a scalar function. The values of the function are represented in greyscale and increase in value from white (low) to dark (high).





## Gradient Descent

- Gradient Descent is an optimization algorithm used in machine learning and deep learning for training models/Neural networks and finding the optimal parameters that minimize a cost function.





# Gradient Descent

- Step-by-step explanation of how Gradient Descent works:
  - **Step:1 Initialization:** It begins by initializing the model parameters randomly or with some predetermined values. These parameters could be the weights and biases of a neural network, for example.
  - **Step:2 Compute Gradient:** At each iteration, the algorithm computes the gradient of the cost function with respect to each parameter. The gradient represents the direction of the steepest ascent of the function.
  - **Step:3 Update Parameters:** Once the gradient is computed, the algorithm updates the parameters by moving them in the opposite direction of the gradient. This is done to minimize the cost function. The update rule typically involves multiplying the gradient by a learning rate parameter and subtracting the result from the current parameter values.
  - **Step:4 Iterate:** Steps 2 and 3 are repeated iteratively until a stopping criterion is met. This could be a predefined number of iterations or until the change in the cost function falls below a certain threshold.



# Gradient Descent

- Example: Finding the Lowest Point in a Valley

Imagine you are blindfolded and placed somewhere in a valley. Your goal is to find the lowest point in the valley without being able to see the terrain(area of land). Here's how you might proceed:

- **Initial Position:** You start at a random location in the valley.
- **Objective:** Your objective is to descend to the lowest point in the valley.
- **Sense of Touch:** You can feel the slope of the ground beneath your feet, giving you an indication of the direction of descent.
- **Movement:** You take a step in the direction of the steepest slope, relying on your sense of touch to guide you downhill.
- **Repetition:** You repeat this process, continuously adjusting your direction based on the slope of the terrain.
- **Convergence:** Eventually, you reach the lowest point in the valley, indicating convergence to the optimal solution.



## Gradient Descent

- In this analogy:
  - The blindfolded person represents the optimization algorithm.
  - The sense of touch represents the gradient, providing information about the direction of descent.
  - Moving downhill corresponds to updating the parameters in the direction that minimizes the objective function.
  - This example illustrates how Gradient Descent works by iteratively adjusting parameters to minimize a cost function, much like finding the lowest point in a valley by descending along the steepest slope.





# Gradient Descent

There are three different variants of Gradient Descent

1. Batch Gradient Descent
2. Stochastic Gradient Descent
3. Mini-batch Gradient Descent





# Batch Gradient Descent

- Batch Gradient Descent (BGD) is a variant of the Gradient Descent optimization algorithm used to minimize a cost function in machine learning and deep learning models.
- It's called "batch" because it processes the entire training dataset in each iteration to update the model parameters.
  - Optimization Algorithm in Machine/Deep Learning
  - Variant of Gradient Descent
  - Minimize a cost function in ML/DL models
  - Processes entire training dataset in each iteration



# Batch Gradient Descent

- **Working Principle**
  - **Initialization**: Start with initial guess for model parameters
  - **Compute Gradient**: Calculate gradient of cost function with respect to parameters using entire training dataset
  - **Update Parameters**: Adjust parameters using gradients and learning rate
  - **Repeat** until convergence criteria met
- **Key Aspects**
  - Uses entire dataset in each iteration so it is computationally expensive for large datasets
  - Accurate estimate of gradient
  - Often converges to global minimum in most of the problems



# Stochastic Gradient Descent

- SGD is a variant of the gradient descent optimization algorithm widely used in machine learning and deep learning.
- Unlike batch gradient descent, which computes the gradient using the entire dataset, SGD updates the model parameters using a single training example at a time.



# Stochastic Gradient Descent

### Working Principle:

- Initialization: Start with initial guess for model parameters.
- For each epoch:
  - Shuffle the training dataset.
  - Iterate through each training example:
    - Compute the gradient of the cost function with respect to the current training example.
    - Update the model parameters using the computed gradient and a predefined learning rate.
- Repeat the process for a fixed number of epochs or until convergence criteria are met.



# Mini-batch Gradient Descent

- MBGD strikes a balance between the efficiency of stochastic gradient descent (SGD) and the stability of batch gradient descent by updating the model parameters using a small subset of the training data at each iteration.
- Instead of using the entire training dataset (as in batch gradient descent) or just one example (as in SGD), MBGD divides the dataset into small batches and updates the parameters based on the average gradient computed over each batch.





# Mini-batch Gradient Descent

### Working Principle:

- Initialization: Start with an initial guess for the model parameters.
- Divide the training dataset into mini-batches of equal size (e.g., 32, 64, or 128 examples per batch).
- For each epoch:
  - Shuffle the training dataset to introduce randomness and prevent the model from getting stuck in local minima.
  - Iterate through each mini-batch:
    - Compute the gradient of the cost function with respect to the mini-batch.
    - Update the model parameters using the computed gradient and a predefined learning rate.
- Repeat the process for a fixed number of epochs or until convergence criteria are met.



# Mini-batch Gradient Descent

### Advantages:

- Offers a good compromise between the efficiency of SGD and the stability of batch gradient descent.
- Well-suited for training on large datasets that do not fit into memory.

### Limitation:

- Requires tuning of hyperparameters such as the learning rate and batch size.



# Advanced Optimizers

### Optimizers :

- Optimizers adjust model parameters iteratively during training to minimize a loss function, enabling neural networks to learn from data.
- Choosing an appropriate optimizer for a deep learning model is important as it can greatly impact its performance. Optimization algorithms have different strengths and weaknesses and are better suited for certain problems and architectures.
- **Some advanced optimizers used in neural networks:**
  1. SGD with Momentum
  2. Nesterov Accelerated Gradient (NAG)
  3. AdaGrad (Adaptive Gradient)
  4. Gradient Descent with RMSprop(Root Mean Squared Propagation)
  5. Adam (Adaptive Moment Estimation)



# SGD with Momentum

- Momentum optimization is a popular variant of the gradient descent optimization algorithm commonly used to train neural networks. It addresses some of the limitations of basic gradient descent, particularly slow convergence in the presence of flat or small gradients and oscillations in the optimization process.
- In momentum optimization, instead of updating the weights based solely on the current gradient, it also considers the accumulation of past gradients to determine the direction of the update.
- This is achieved by introducing a new parameter called the momentum parameter, denoted by  $\beta$ , which is typically set to a value between 0 and 1.



## SGD with Momentum

The update rule for momentum optimization can be expressed as follows:

$$v_{t+1} = \beta \cdot v_t + \alpha \cdot \nabla J(\theta)$$

$$\theta_{t+1} = \theta_t - v_{t+1}$$

where:

- $v_t$  is the momentum term at iteration  $t$ ,
- $\alpha$  is the learning rate,
- $\nabla J(\theta)$  is the gradient of the loss function  $J$  with respect to the parameters  $\theta$ , and
- $\theta_t$  and  $\theta_{t+1}$  are the parameters at iterations  $t$  and  $t + 1$ , respectively.



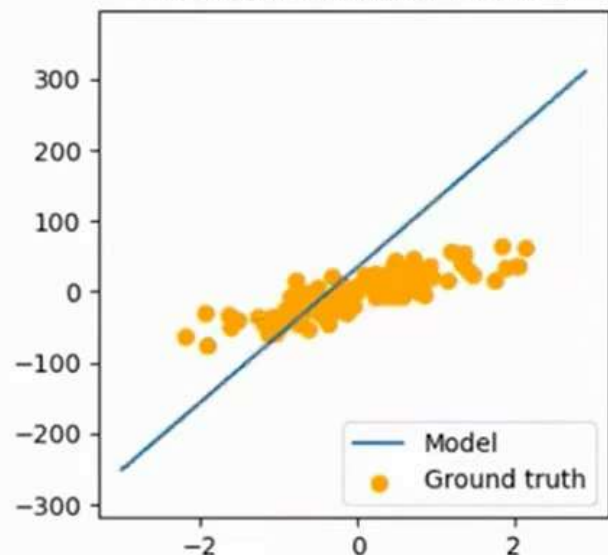


# SGD with Momentum

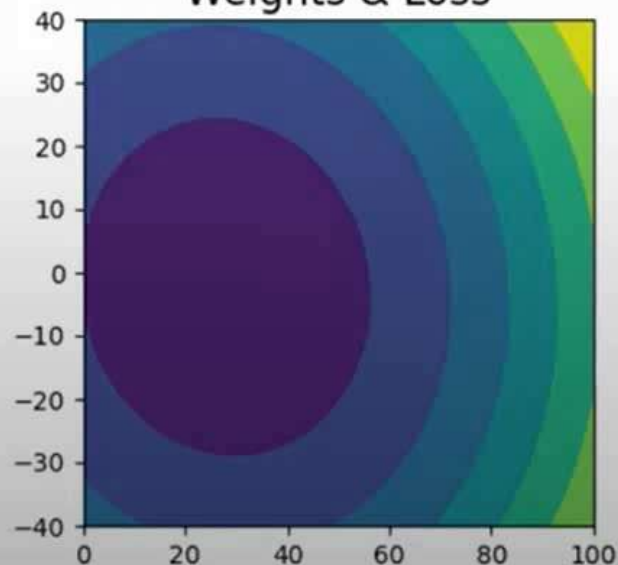
- The momentum term  $V_t$  is an exponentially weighted moving average of past gradients (Exponentially Weighted Moving Average is a method for smoothing time-series data by assigning exponentially decreasing weights to older observations. It is widely used for trend analysis, noise reduction, and forecasting in various fields.). It accelerates the updates in directions where the gradients point consistently over time and dampens oscillations in directions where the gradients change direction frequently.
- By incorporating momentum, the optimizer gains inertia, enabling it to continue moving in the same direction for a longer time and traverse through regions of flat or small gradients more efficiently. This leads to faster convergence and reduced oscillations during training.
- In short momentum optimization accelerates gradient descent by introducing a momentum term that accumulates past gradients, helping the optimizer navigate through complex optimization landscapes more effectively. It is widely used in practice due to its ability to speed up training and improve convergence for deep learning models.

# Batch Gradient Descent epoch number: = 0

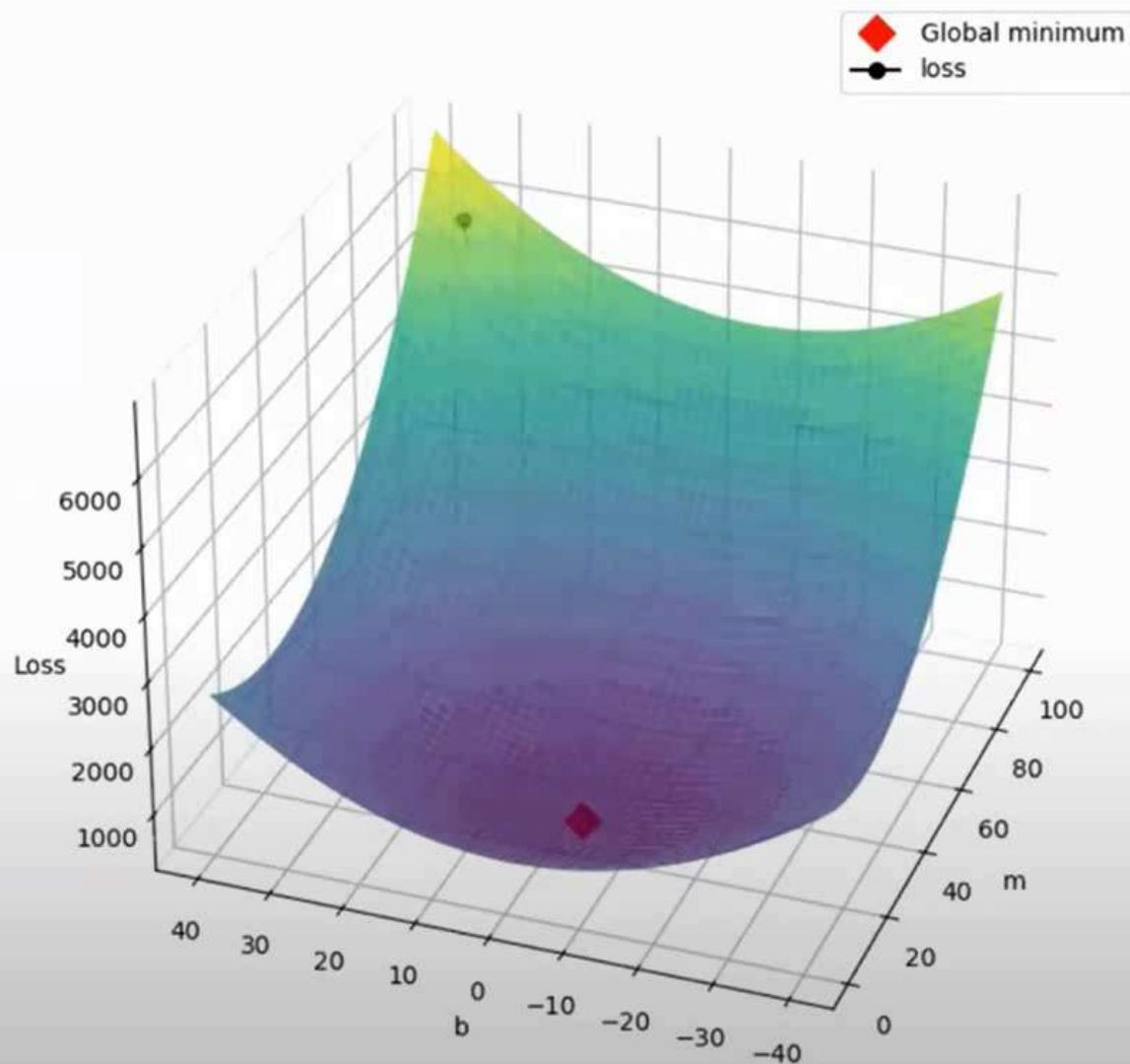
Ground truth & Model



Weights & Loss

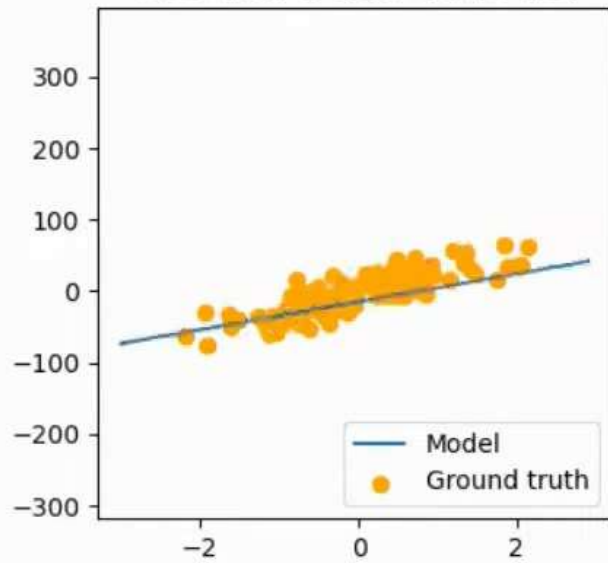


contour plot

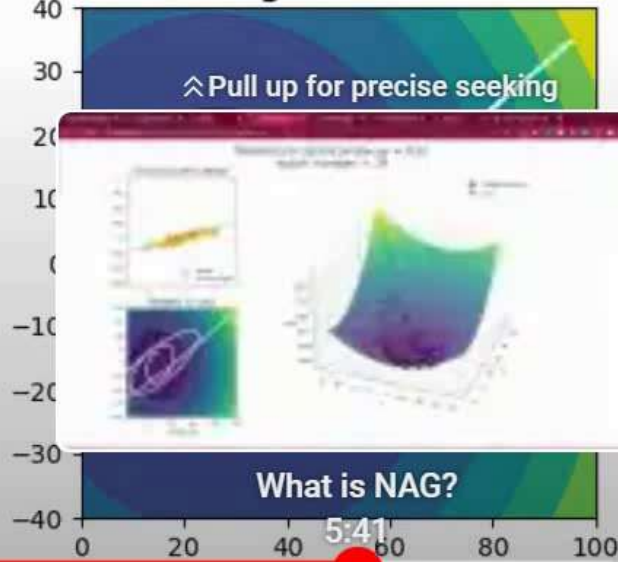


# Momentum Optimizer(decay = 0.8) epoch number: = 3

Ground truth & Model

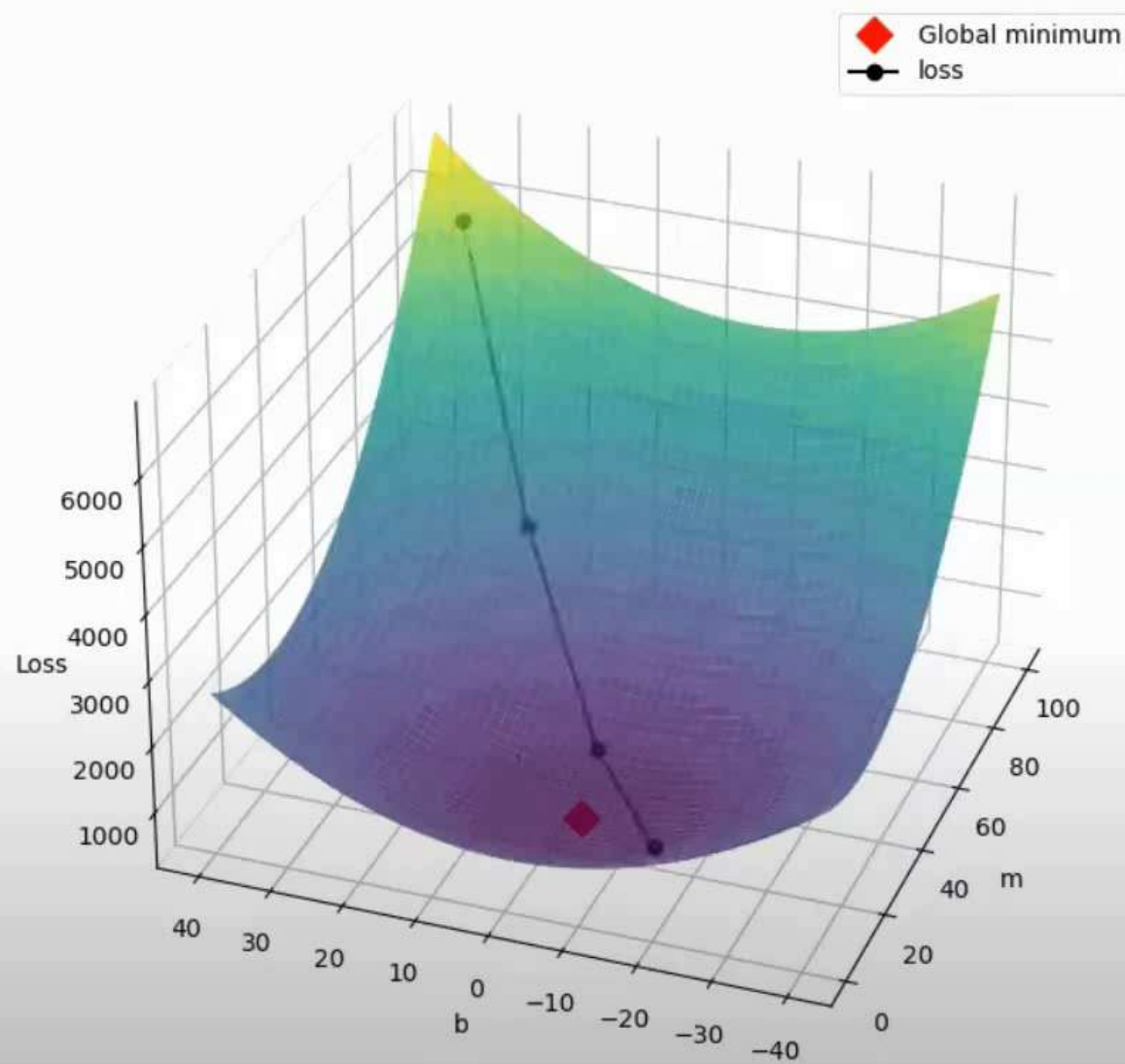


Weights & Loss



What is NAG?

5:41





# Nesterov Accelerated Gradient (NAG)

- Nesterov Accelerated Gradient (NAG) is an optimization algorithm that builds upon the momentum optimization method. It aims to improve upon the original momentum approach by addressing the issue of momentum overshooting, which can occur when the current gradient update is combined with the accumulated momentum term.
- In Nesterov Accelerated Gradient, instead of evaluating the gradient at the current position of the parameters, it evaluates the gradient at an adjusted position that takes into account the momentum term. This adjustment is made to anticipate the future position of the parameters based on the momentum.



# Nesterov Accelerated Gradient (NAG)

Nesterov Accelerated Gradient can be expressed as follows:

$$v_{t+1} = \beta \cdot v_t + \alpha \cdot \nabla J(\theta - \beta \cdot v_t)$$

$$\theta_{t+1} = \theta_t - v_{t+1}$$

where:

- $v_t$  is the momentum term at iteration  $t$ ,
- $\alpha$  is the learning rate,
- $\nabla J(\theta - \beta \cdot v_t)$  is the gradient of the loss function  $J$  evaluated at the adjusted position  $\theta - \beta \cdot v_t$ ,
- $\theta_t$  and  $\theta_{t+1}$  are the parameters at iterations  $t$  and  $t + 1$ , respectively, and
- $\beta$  is the momentum parameter, typically set to a value between 0 and 1.





# Nesterov Accelerated Gradient (NAG)

- The key difference between Nesterov Accelerated Gradient and traditional momentum optimization is that the gradient is evaluated at the "lookahead" position ( $\theta - \beta \cdot v_t$ ) which anticipates the future position of the parameters before updating them with the momentum term. This allows NAG to correct the momentum overshooting problem by incorporating a more accurate gradient estimate.
- By incorporating Nesterov momentum, the optimizer can adjust the momentum term more effectively and reduce oscillations, leading to faster convergence and improved optimization performance compared to traditional momentum optimization.



# Deep Learning

