EC3357:Machine Learning

Lecture 5B: Neural Network Representation

Creating our own simple neural network:

- Let's say we want our neural network to be able to return outputs according to the rules of the AND truth table.
- The beginning of the program just defines libraries and the values of the parameters and creates a list which contains the values of the weights that will be modified (those are generated randomly).

```
import random
lr = 1 #learning rate
bias = 1 #value of bias
weights=[random.random(),random.random(),random.random()]
#weights generated in a list (3 weights in total for 2 neurons and the bias)
```

- Create a function which defines the work of the output neuron. It takes 3 parameters (the 2 values of the neurons and the expected output). “outputP” is the variable corresponding to the output given by the Perceptron. Calculate the error, used to modify the weights of every connections to the output neuron right after.

```
def Perceptron(input1, input2, output):
    outputP = input1 * weights[0] + input2 * weights[1] + bias *
weights[2]
    if outputP > 0:  # activation function (here Heaviside)
        outputP = 1
    else:
        outputP = 0
    error = output - outputP
    weights[0] += error * input1 * lr
    weights[1] += error * input2 * lr
    weights[2] += error * bias * lr
```

- The following part is the learning phase. The number of iteration is chosen according to the precision we want. However, be aware that too much iterations could lead the network to over-fitting, which causes it to focus too much on the treated examples, so it couldn’t get a right output on case it didn’t see during its learning phase.

```
# Train the perceptron
for i in range(50):
    Perceptron(1, 1, 1)  # True or true
    Perceptron(1, 0, 0)  # True or false
    Perceptron(0, 1, 0)  # False or true
    Perceptron(0, 0, 0)  # False or false
```

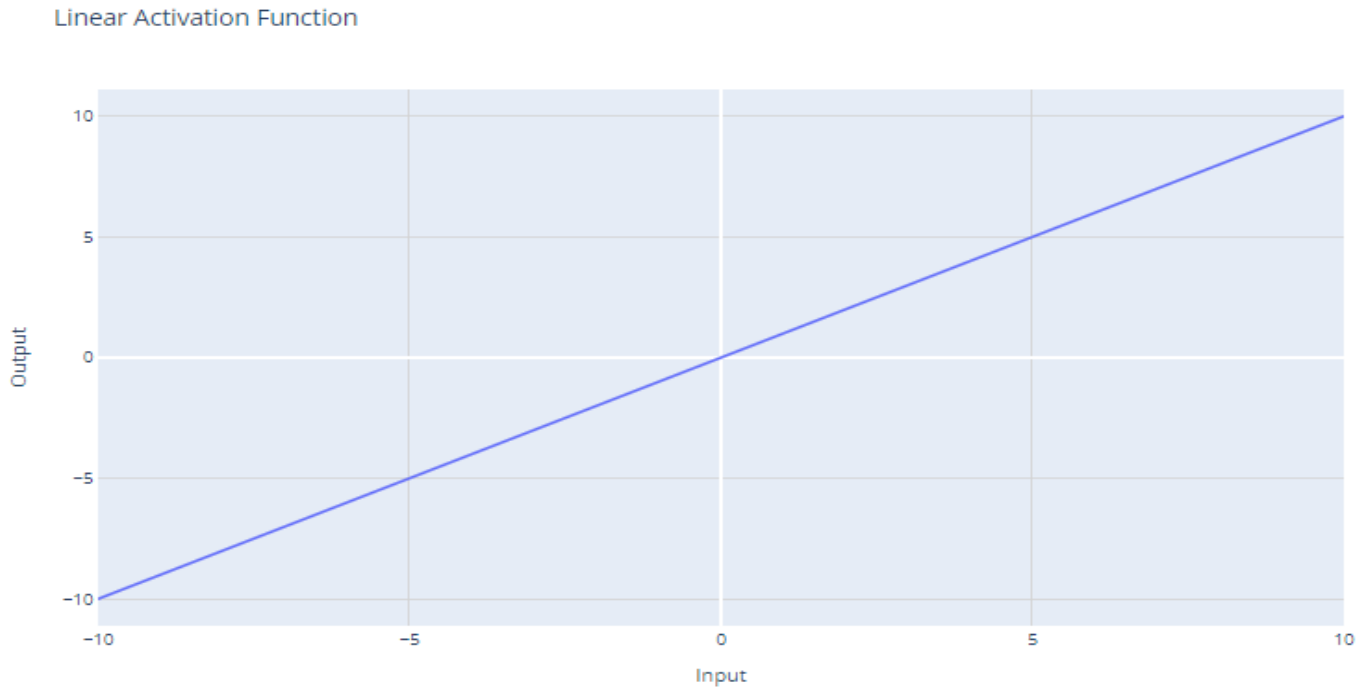
- Finally, we can ask the user to enter himself the values to check if the Perceptron is working. This is the testing phase.

```
# Take user input
x = int(input("Enter the first binary input (0 or 1): "))
y = int(input("Enter the second binary input (0 or 1): "))

# Calculate the perceptron output
outputP = x * weights[0] + y * weights[1] + bias * weights[2]

if outputP > 0:  # activation function
    outputP = 1
else:
    outputP = 0

print(x, "and", y, "is:", outputP)
```

Activation Function

- Activation functions are an integral building block of neural networks that enable them to learn complex patterns in data. They transform the input signal of a node in a neural network into an output signal that is then passed on to the next layer.
- There is a different kind of the activation function, but primarily either linear or non-linear sets of functions. Some of the commonly used sets of activation functions are the Binary, linear, and Tanh hyperbolic sigmoidal activation functions.

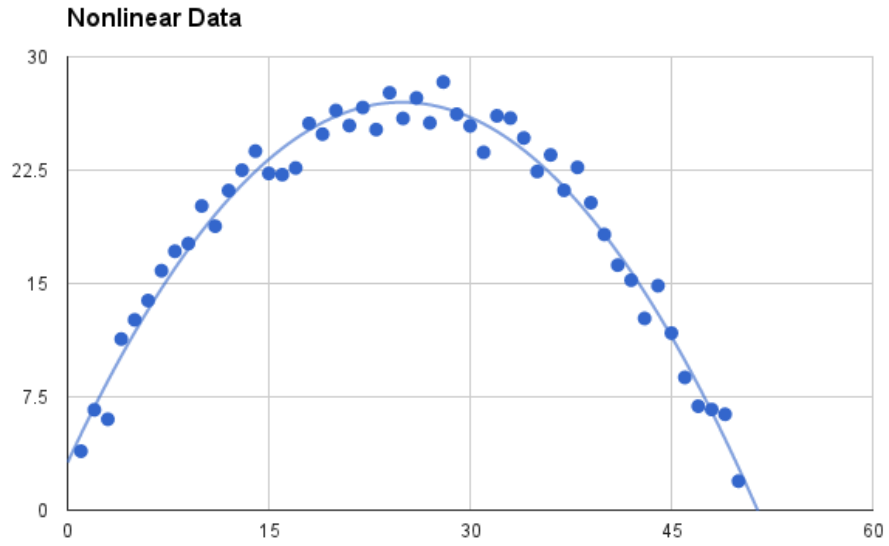
Linear or Identity Activation Func' '



- **Equation :** f(x) = x
- **Range :** (-infinity to infinity)
- Mainly used for regression
- It doesn’t help with the complexity or various parameters of usual data that is fed to the neural networks.

Non-linear Activation Function

- The Nonlinear Activation Functions are the most used activation functions. Nonlinearity helps to makes the graph look something like this:

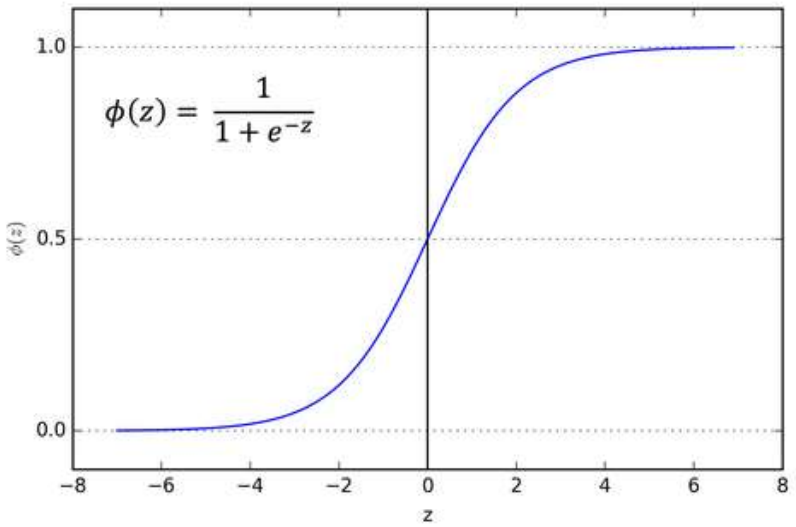


- It makes it easy for the model to generalize or adapt with variety of data and to differentiate between the output.

- The main terminologies needed to understand for nonlinear functions are:
  - Derivative or Differential: Change in y-axis w.r.t. change in x-axis. It is also known as slope.
  - Monotonic function: A function which is either entirely non-increasing or non-decreasing.
- The Nonlinear Activation Functions are mainly divided on the basis of their range or curves.

1. Sigmoid or Logistic Activation Function

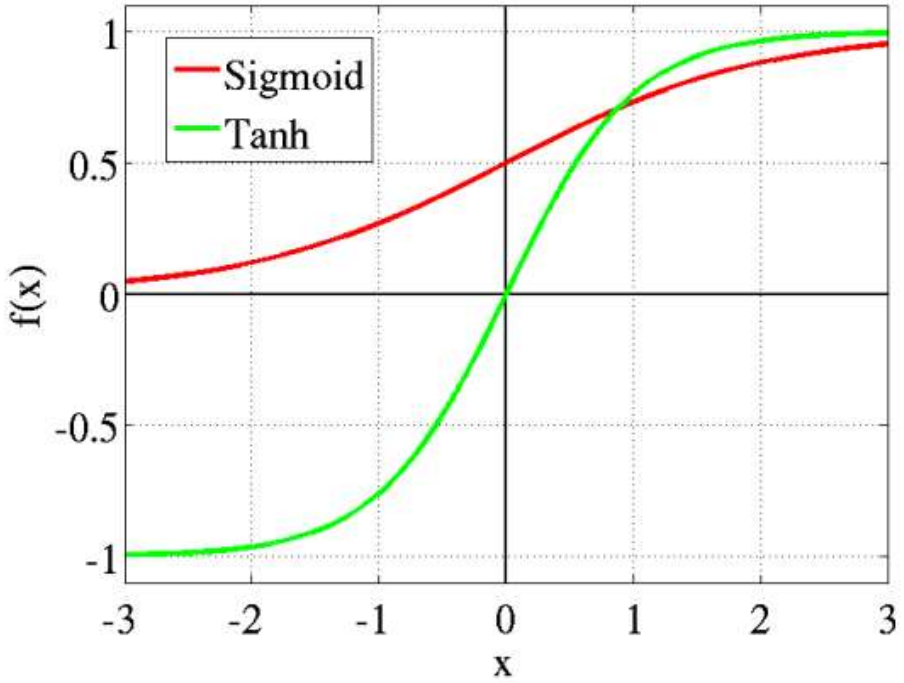
The Sigmoid Function curve looks like a S-shape.



- The main reason why we use sigmoid function is because it exists between **(0 to 1).** Therefore, it is especially used for models where we have to **predict the probability** as an output. Since probability of anything exists only between the range of **0 and 1,** sigmoid is the right choice.

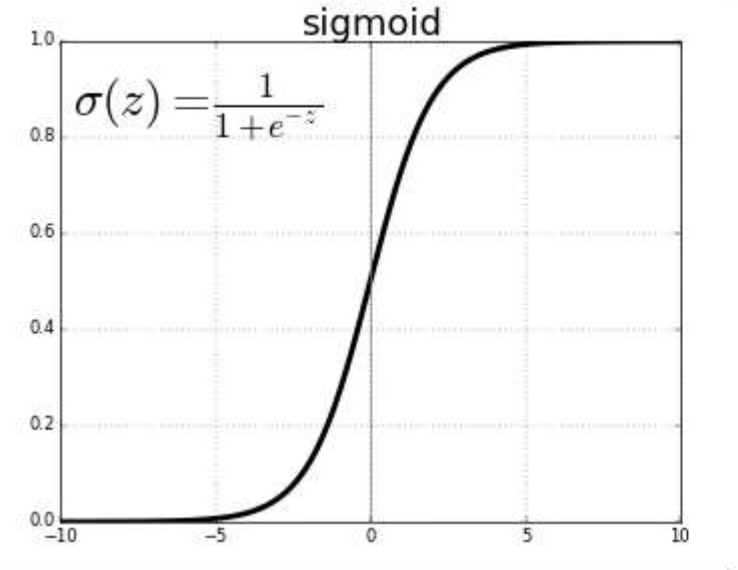
2. Tanh or hyperbolic tangent Activation Function

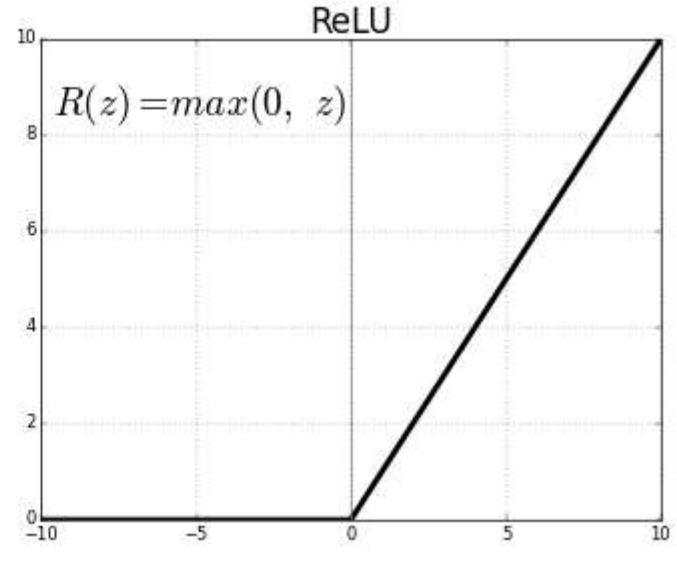
- tanh is also like logistic sigmoid but better. The range of the tanh function is from (-1 to 1). This means that it can deal with negative values more effectively than the sigmoid function, which has sigmoidal (s - shaped).



3.ReLU (Rectified Linear Unit) Activation Function

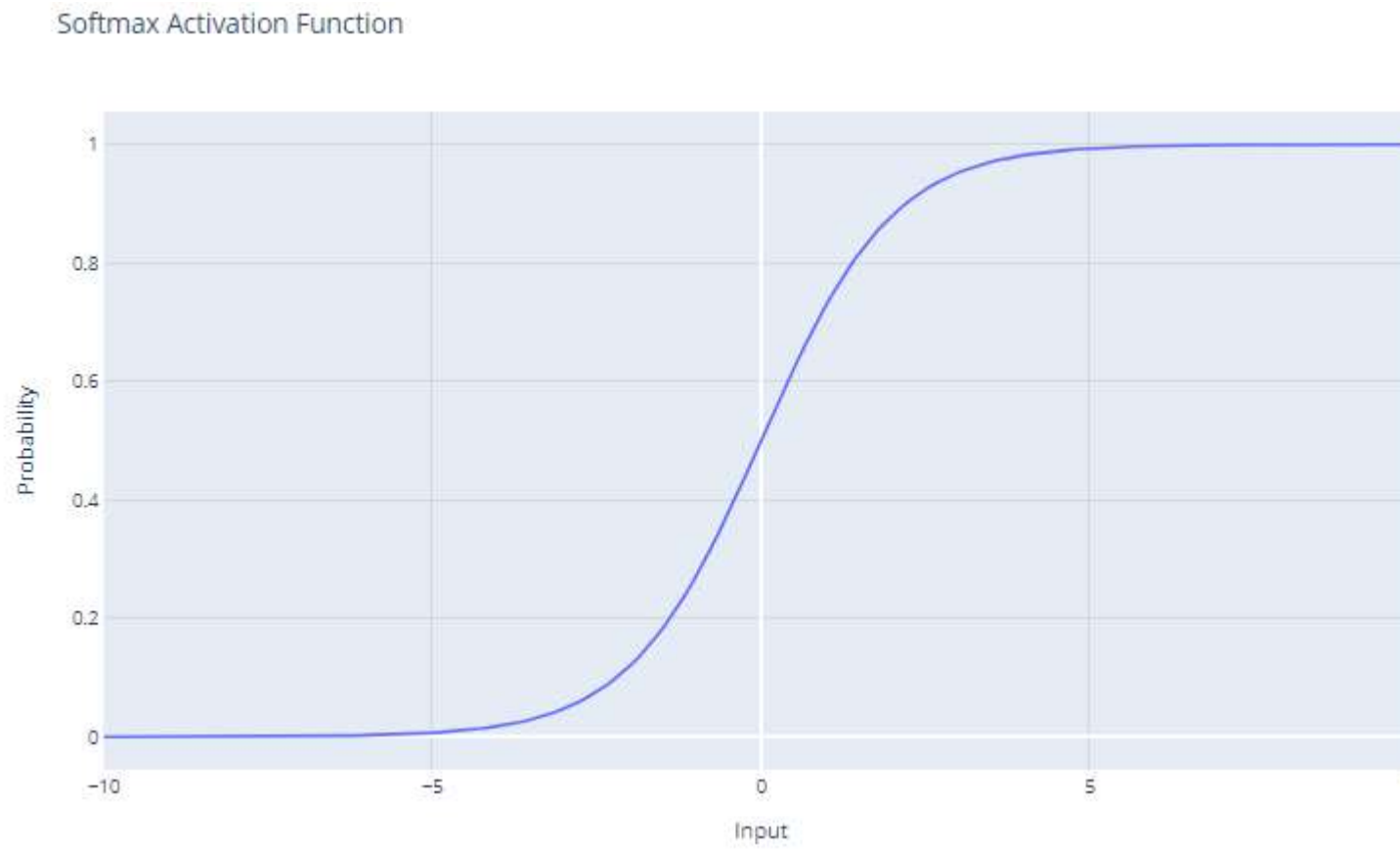
- The ReLU is the most used activation function in the world right now. Since, it is used in almost all the convolutional neural networks or





Softmax activation

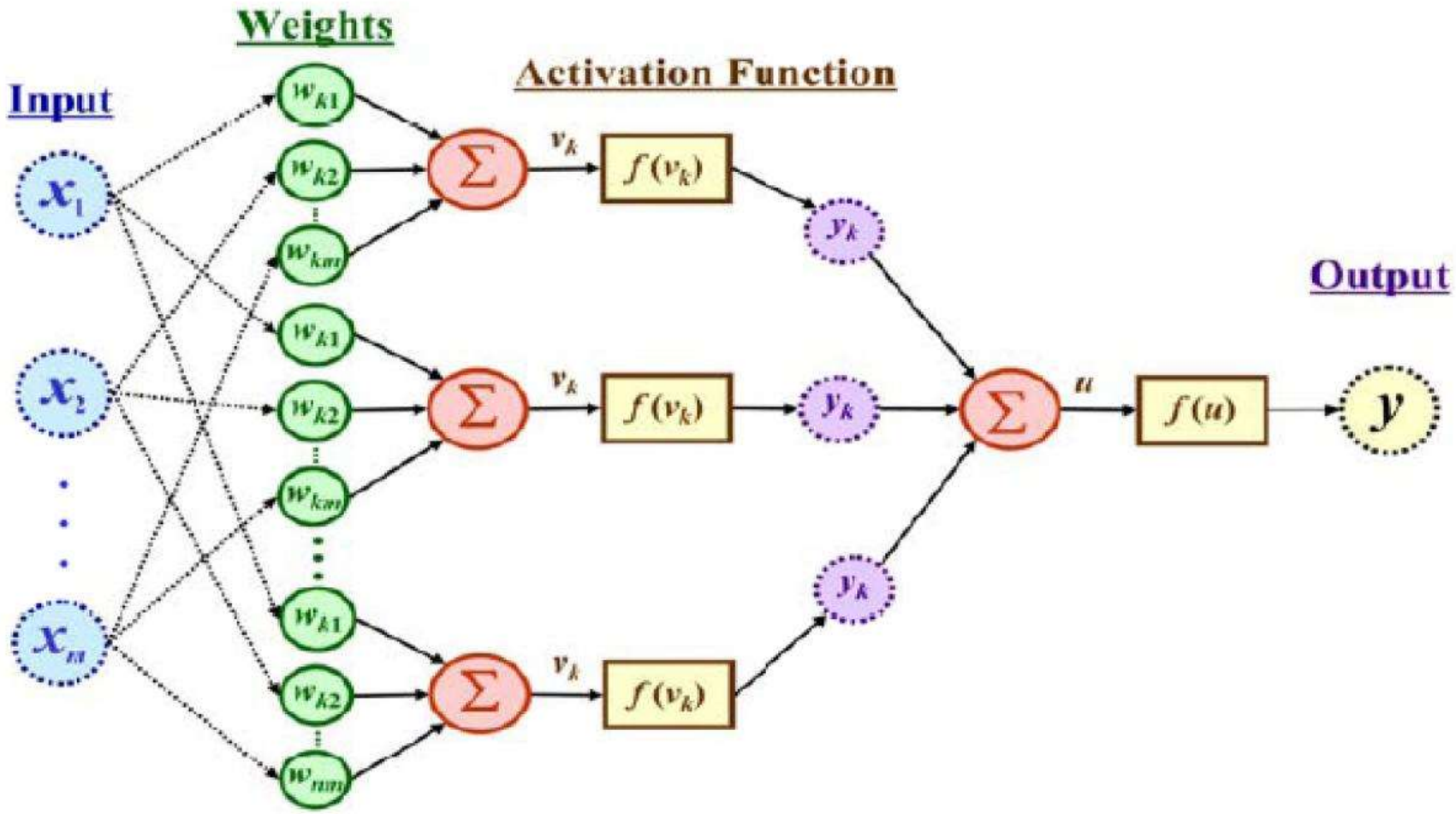
- The softmax activation function, also known as the normalized exponential function, is particularly useful within the context of multi-class classification pr



Choosing the Right Activation Function

- The choice of activation function depends on the type of problem you are trying to solve. Here are some guidelines:
  - For binary classification:
    - Use the sigmoid activation function in the output layer. It will squash outputs between 0 and 1, representing probabilities for the two classes.
  - For multi-class classification:
    - Use the softmax activation function in the output layer. It will output probability distributions over all classes.
  - If unsure:
    - Use the ReLU activation function in the hidden layers. ReLU is the most common default activation function and usually a good choice.

The output is a function of the input, that is affected by the weights, and the transfer functions



Neural Network Classifier

- **Input: Classification data**

  It contains classification attribute

  Data is divided, as in any classification problem.

  [Training data and Testing data]

- **<u>All data must be normalized.</u>**

  (i.e. all values of attributes in the database are changed to contain values in the internal [0,1] or [-1,1])

  Neural Network can work with data in the range of (0,1) or (-1,1)

- **Two basic normalization techniques**

  [1] Max-Min normalization

  [2] Decimal Scaling normalization

Data Normalization

[1] Max- Min normalization formula is as follows:

$$v' = \frac{v - \min\ A}{\max\ A - \min\ A}(new\_\max\ A - new\_\min\ A) + new\_\min\ A$$

**[minA, maxA , the minimun and maximum values of the attribute A
max-min normalization maps a value v of A to v' in the range
{new_minA, new_maxA}]**

Example of Max-Min Normalization

Max- Min normalization formula

$$v' = \frac{v - \min A}{\max A - \min A}(new\_\max A - new\_\min A) + new\_\min A$$

Example: We want to normalize data to range of the interval [0,1].
We put: new_max A= 1, new_minA =0.

Say, max A was 100 and min A was 20 (That means maximum and minimum values for the attribute).

Now, if v = 40 (If for this particular pattern , attribute value is 40), v' will be calculated as , v' = (40-20) x (1-0) / (100-20) + 0
=> v' = 20 x 1/80
=> v' = 0.4

Decimal Scaling Normalization

[2]Decimal Scaling Normalization

Normalization by decimal scaling normalizes by moving the decimal point of values of attribute A.

$$v' = \frac{v}{10^j}$$

Here j is the smallest integer such that max|v'|<1.

Example :

A – values range from -986 to 917. Max |v| = 986.

v = -986 normalize to v' = -986/1000 = -0.986

Mean Squared Error (MSE)

```
def mean_squared_error(predictions, targets):
    return ((predictions - targets) ** 2).mean()

# Example usage:
predictions = model.predict(X)
mse = mean_squared_error(predictions, y_true)
print(f"Mean Squared Error: {mse}")
```

Cross-Entropy Loss:

```
import numpy as np

def cross_entropy_loss(predictions, targets):
    epsilon = 1e-15
    predictions = np.clip(predictions, epsilon, 1 - epsilon)
    return -np.mean(targets * np.log(predictions) + (1 -
targets) * np.log(1 - predictions))

# Example usage:
predictions = model.predict(X)
ce_loss = cross_entropy_loss(predictions, y_true)
print(f"Cross-Entropy Loss: {ce_loss}")
```

Mean Absolute Error (MAE):

```
def mean_absolute_error(predictions, targets):
    return np.abs(predictions - targets).mean()

# Example usage:
predictions = model.predict(X)
mae = mean_absolute_error(predictions, y_true)
print(f"Mean Absolute Error: {mae}")
```

Conclusion

- It is important to understand how to calculate the errors and how to interpret them.
- By doing so, we improve the accuracy and performance of our neural network and ensure that it is generalizing well to unseen data.