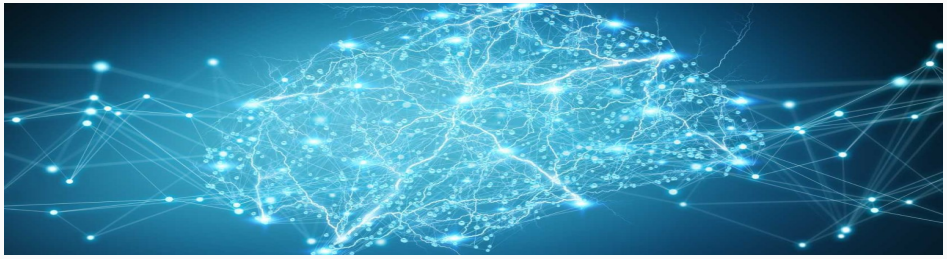


Lecture slides - Week 2

MLP and Activation Functions

Dr. Aamir Akbar

Director of both AWKUM AI Lab and AWKUM Robotics, Final Year Projects (FYPs) coordinator, and lecturer at the department of Computer Science
Abdul Wali Khan University, Mardan (AWKUM)

Contents

Multi-Layer Perceptron (MLP)

What is a MLP? How is it different than FNN and SLP? i

Feedforward Neural Network (FNN) generally refers to any neural network where the connections between the nodes do not form a cycle (or there is no feedback loop). In other words, the information flows in one direction, from the input nodes (or neurons) through the hidden layers (if any) to the output nodes. FNNs can have one or more hidden layers, but they do not have feedback connections.

Multi-layer Perceptron (MLP) refers to a type of feedforward neural network with at least one hidden layer of nodes between the input and output layers. Each node (or neuron) in the hidden layer is a perceptron, which is a basic computational unit that calculates a weighted sum of its inputs and applies an activation function to produce the output.

To sum up, while all MLPs are feedforward neural networks, not all feedforward neural networks are MLPs.

What is a MLP? How is it different than FNN and SLP? ii

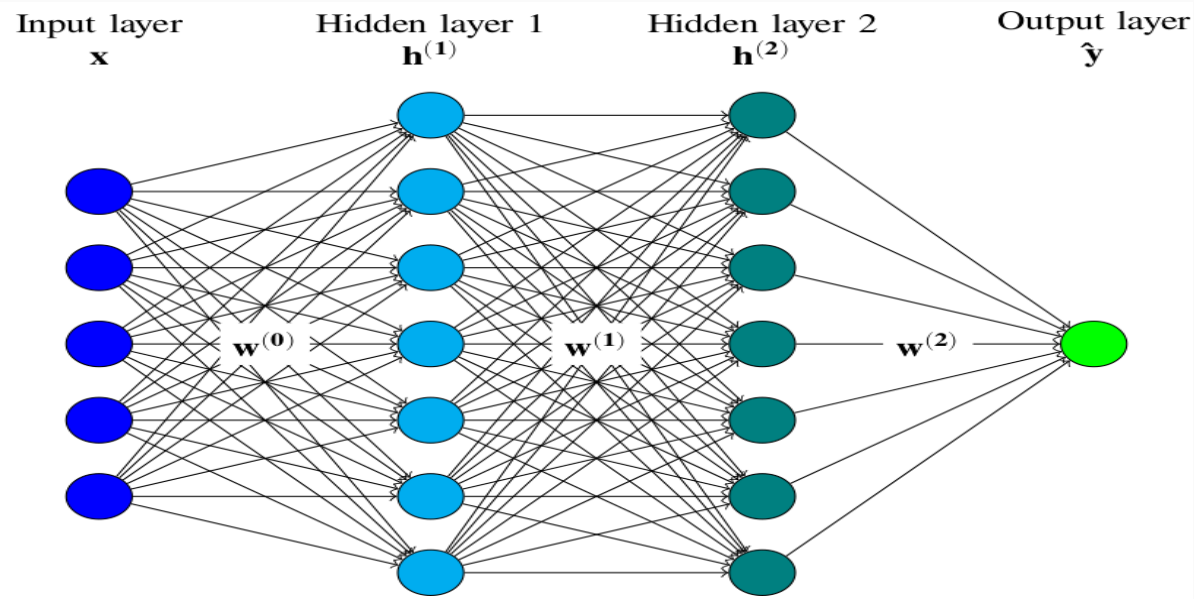
A single-layer perceptron (SLP), or a single-layer neural network, refers to a neural network with only one layer, or with more than one neuron in a single layer.

1. Each neuron in the single layer directly connects to the input features.
2. The neurons calculate a weighted sum of the input features and apply an activation function to produce the output.
3. There are no hidden layers between the input and output layers.

SLPs are limited in their ability to model complex relationships in data and can only learn linear decision boundaries. However, they are still useful for simple classification tasks.

Mathematical Background of MLP

MLP: Mathematical Modeling i



$$h^{(1)} = \sigma(W^{(0)}x + b^{(0)}) \tag{1}$$

$$h^{(2)} = \sigma(W^{(1)}h^{(1)} + b^{(1)}) \tag{2}$$

$$\hat{y} = \sigma_{out}(W^{(2)}h^{(2)} + b^{(2)}) \tag{3}$$

The input layer serves as a `placeholder for input features`, transmitting data to subsequent layers. The computational core of the MLP lies within the hidden layers and the output layer.

MLP: Mathematical Modeling ii

Elementwise operation of the network:

$$\hat{y} = \sigma_{out}\Big(\sum_{i=1}^{d_2} w_i^{(2)}\ \sigma\Big(\sum_{i=1}^{d_1} w_{ji}^{(1)}\ \sigma\Big(\sum_{i=1}^{d} w_{ji}^{(0)} x_i + b_j^{(0)}\Big) + b_j^{(1)}\Big) + b_j^{(2)}\Big) \quad (4)$$

where,

- x_i is the ith input feature
- d is the number of input features
- d_k is the number of neurons in the kth hidden layer
- $w_{ji}^{(k)}$ is the weight connecting neuron i in layer $k-1$ to neuron j in layer k
- $b_j^{(k)}$ is the bias of neuron j in layer k
- σ is the activation function (e.g., sigmoid, ReLU)
- $\hat{y}$ is the predicted value of the output layer

MLP: Mathematical Modeling iii

The Equation 4 represents the Elementwise operation of the MLP Network following these steps:

1. The `input layer` consists of d features, denoted by x_i. This layer merely serve as placeholders for the input features.
2. The `first hidden` layer has d neurons. Each neuron j in the first hidden layer computes a weighted sum of the inputs x_i, adds a bias $b_j^{(0)}$, and applies an activation function σ (e.g., sigmoid, ReLU). The output of neuron j is denoted by $h_j^{(1)}$, whereas the output of the first hidden layer is denoted in Equation 1.
3. The `second hidden` layer has d_1 neurons. Each neuron j in the second hidden layer computes a weighted sum of the outputs from the first hidden layer $h_i^{(1)}$, adds a bias $b_j^{(1)}$, and applies the activation function σ. The output of neuron j is denoted by $h_j^{(2)}$, whereas the output of the second hidden layer is denoted in Equation 2.

MLP: Mathematical Modeling iv

4. The output layer has a single neuron. It computes a weighted sum of the outputs from the second hidden layer $h_i^{(2)}$, adds a bias $b^{(2)}$, and applies the activation function σ_{out}. The output of the network is denoted by $\hat{y}$, as shown in Equations 3 and 4.

Lastly, it should be noted that the activation function σ is applied element-wise to the output of each neuron in layer k.

Activation Function

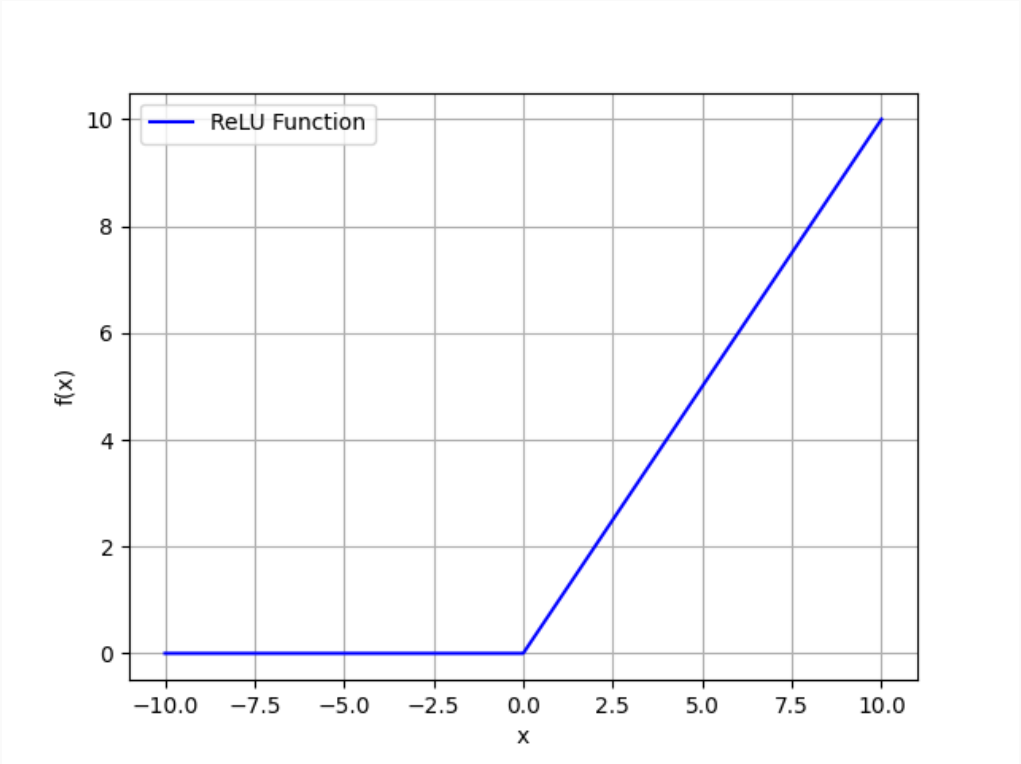
Popular Activation Functions

Several activation functions are commonly used in artificial neural networks (ANNs), each with its own characteristics and suitability for different types of tasks. Some of the most popular activation functions include:

- ReLU (Rectified Linear Unit)
- Sigmoid
- Tanh (Hyperbolic Tangent)

Note: the `choice of activation function` depends on the specific requirements of the problem and the `characteristics of the data`.

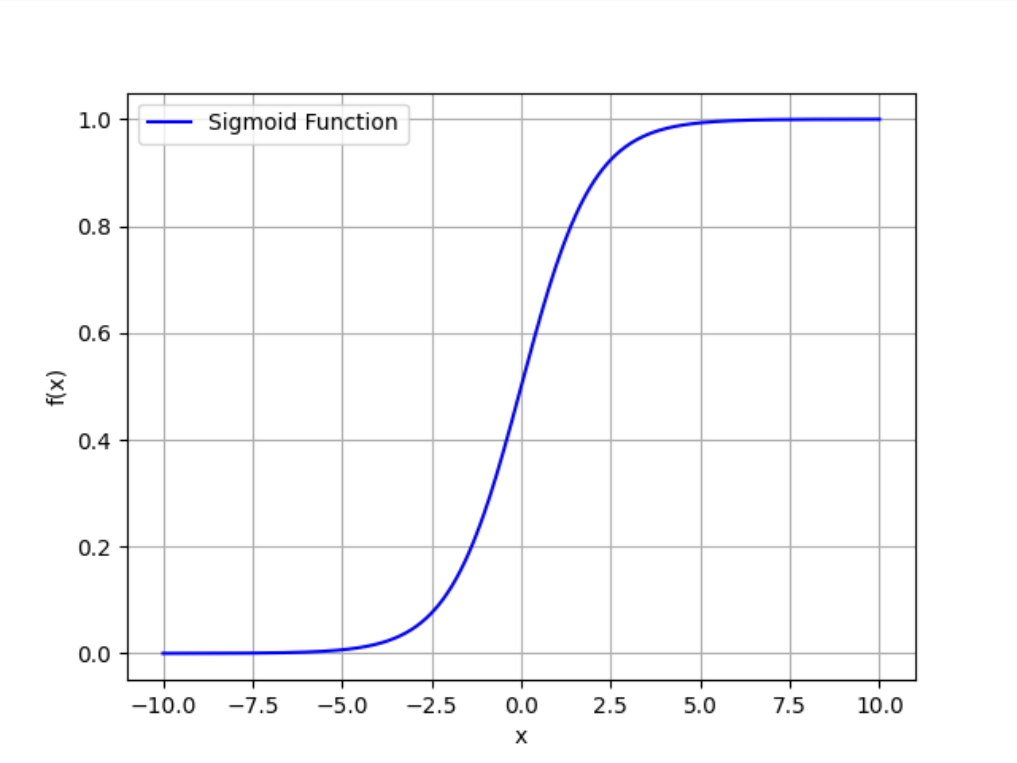
ReLU (Rectified Linear Unit)



$$f(x) = \max(0, x) \tag{5}$$

ReLU is widely used due to its simplicity and effectiveness. It introduces non-linearity to the network and helps avoid the vanishing gradient problem. It's computationally efficient and converges faster compared to sigmoid and tanh.

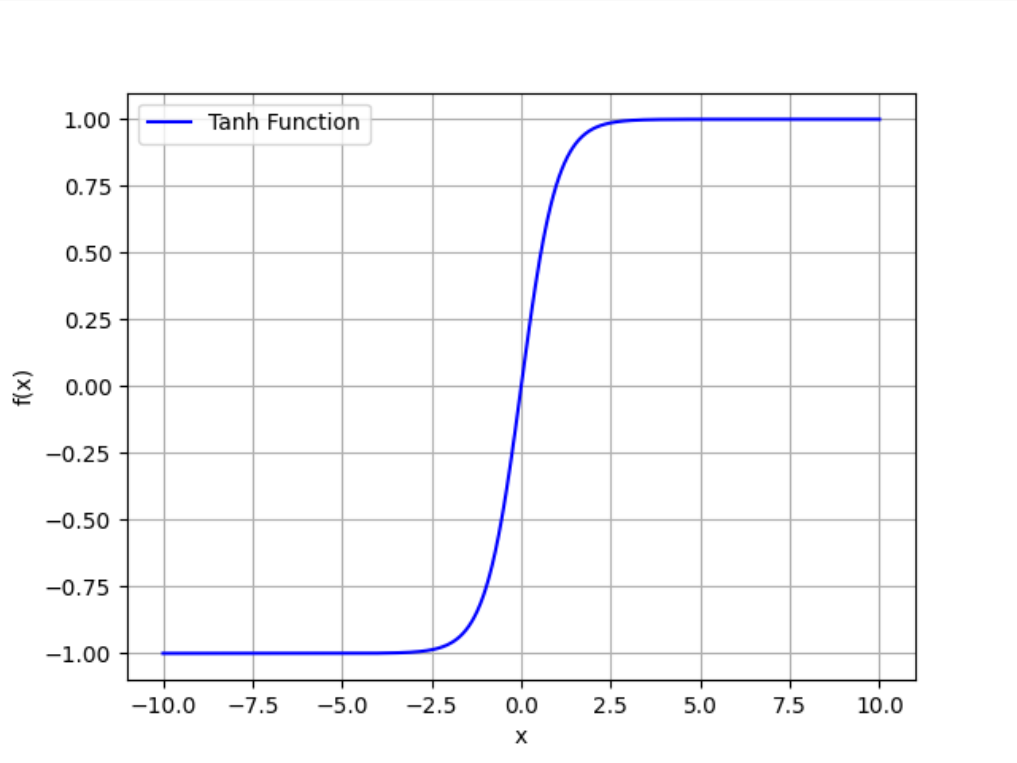
Sigmoid



$$f(x) = \frac{1}{1 + e^{-x}} \tag{6}$$

Sigmoid squashes the input values between 0 and 1. It's often used in binary classification problems where the output needs to be interpreted as probabilities. However, `it suffers from the vanishing gradient problem and is less commonly used in hidden layers of deep neural networks.`

Tanh (Hyperbolic Tangent)



$$f(x) = \frac{e^x - e^{-x}}{e^x + e^{-x}} \tag{7}$$

Tanh squashes the input values between -1 and 1. It's similar to the sigmoid function but ranges from -1 to 1, making it centered around zero. Like sigmoid, it suffers from the vanishing gradient problem, but it's still used in certain contexts.

Where to use which activation function? i

ReLU: ReLU activation in the hidden layers tends to converge faster during training compared to sigmoid and tanh. Also, ReLU avoids the vanishing gradient problem and is computationally efficient, leading to faster training times.

Sigmoid: If the task is binary classification, the network can be trained to output probabilities using the sigmoid function and then apply a threshold (e.g., 0.5) to determine the predicted class. Therefore, sigmoid function is commonly used in the output layer, where the goal is to predict probabilities of belonging to one of the two classes.

Tanh: When used in the output layer, it helps with better converge during training, especially when utilizing gradient descent optimization algorithms. Being centered around zero and having outputs in the range $[-1, 1]$, can improve the stability of the optimization process.

Where to use which activation function? ii

An Optimized and Common Approach:

For binary classification tasks, using ReLU activation in the hidden layers and tanh activation in the output layer, with subsequent conversion to the $[0, 1]$ range using sigmoid, is a common and effective approach in neural network architectures.

ReLU activation in the hidden layers converge faster during training, avoids the vanishing gradient problem and is computationally efficient, leading to faster training times.

Tanh function performs better when utilizing gradient descent optimization algorithms, as it improve the stability of the optimization process.

The sigmoid function is well-suited for tasks where the output values are required as probabilities.

Where to use which activation function? iii

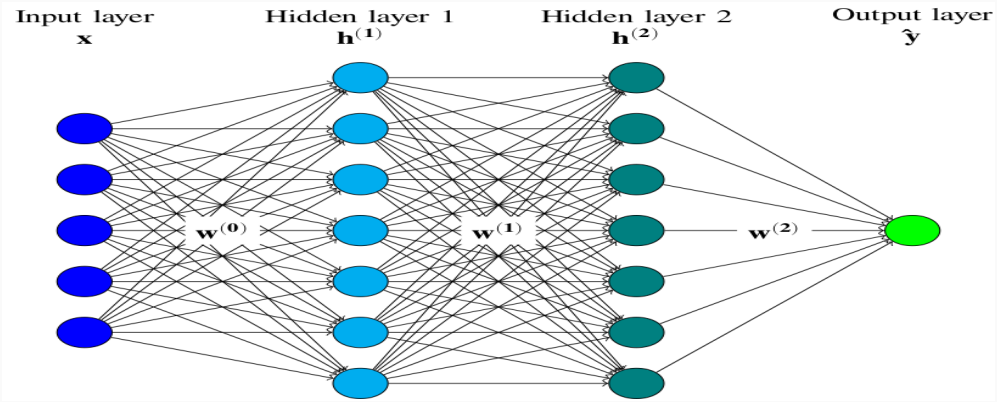
However, it's important to carefully consider the specific characteristics of the problem and the architecture of the neural network to determine the best activation functions and overall design. Experimentation and empirical validation are often necessary to find the optimal configuration for a given task.

Building MLP

Python Implementation using TensorFlow

```python
import tensorflow as tf
from tensorflow.keras.layers import InputLayer
from tensorflow.keras.layers import Dense

# Define the model
class MLP(tf.keras.Model):
    def __init__(self, input_dim, hidden_dim1, hidden_dim2, output_dim):
        super(MLP, self).__init__()
        self.input_layer = InputLayer(input_shape=(input_dim,))
        self.dense1 = Dense(hidden_dim1, activation="relu")
        self.dense2 = Dense(hidden_dim2, activation="relu")
        self.dense3 = Dense(output_dim,  activation="tanh")

    def call(self, inputs):
        x = self.input_layer(inputs)
        x = self.dense1(x)
        x = self.dense2(x)
        return self.dense3(x)

# Create an instance of the model
model = MLP(input_dim=5, hidden_dim1=7, hidden_dim2=7, output_dim=1)

# Print model summary
model.build((None, 5))
model.summary()
```

TensorFlow Model Summary

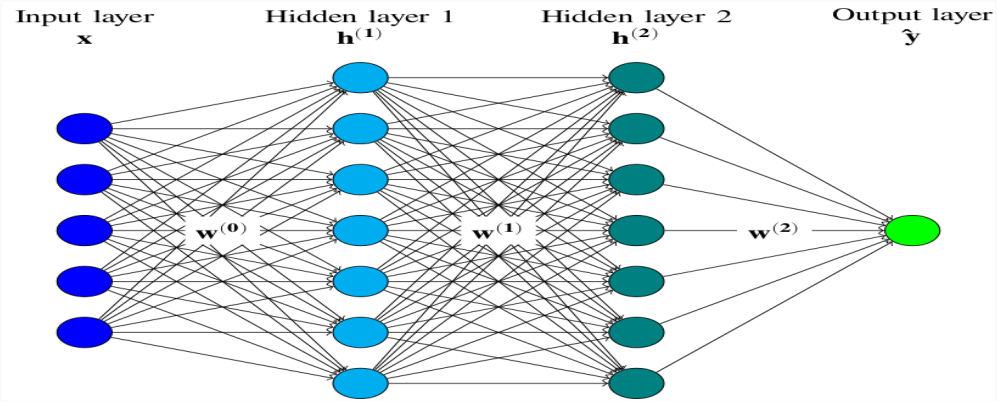



```
Layer (type)                 Output Shape              Param #
=================================================================
input_1 (InputLayer)         [(None, 5)]               0
dense (Dense)                multiple                  42
dense_1 (Dense)              multiple                  56
dense_2 (Dense)              multiple                  8
=================================================================
Total params: 106
Trainable params: 106
Non-trainable params: 0
```

Python Implementation using PyTorch

```python
import torch
import torch.nn as nn

# Define the model
class MLP(nn.Module):
    def __init__(self, input_dim, hidden_dim1, hidden_dim2, output_dim):
        super(MLP, self).__init__()
        self.input_layer = nn.Linear(input_dim, hidden_dim1)
        self.hidden_layer1 = nn.Linear(hidden_dim1, hidden_dim2)
        self.hidden_layer2 = nn.Linear(hidden_dim2, output_dim)
        self.activation = nn.ReLU()

    def forward(self, inputs):
        x = self.input_layer(inputs)
        x = self.activation(x)
        x = self.hidden_layer1(x)
        x = self.activation(x)
        x = self.hidden_layer2(x)
        x = torch.tanh(x)
        return x

# Create an instance of the model
model = MLP(input_dim=5, hidden_dim1=7, hidden_dim2=7, output_dim=1)

# Print model summary
print(model)
```

Pytorch Model Summary




```
MLP(
  (input_layer): Linear(in_features=5, out_features=7,
  bias=True)
  (hidden_layer1): Linear(in_features=7, out_features=7,
  bias=True)
  (hidden_layer2): Linear(in_features=7, out_features=1,
  bias=True)
  (activation): ReLU()
)
```

Resources

Resources

To download the source codes used in the previous slides, follow the link:

▸ Download Source Codes

Import the codes into your preferred development environment, such as Visual Studio Code (VS Code), to practice and explore further.

To learn programming in Python, follow my comprehensive 15-week Programming in Python course at:

▸ Programming in Python