The Brain, Artificial Intelligence and Foundations of Deep Learning & GenAI

Prof. E. Adetiba,

(PI, ASPMIR and FEDGEN @ CApIC-ACE, Covenant University)

@

The 3rd Google TensorFlow College Outreach Bootcamp and FEDGEN Mini-Workshop

Date: 11th to 13th December, 2023

Sponsored By

Google

Outline

1.0 The Biological Neurons in the Brain and Artificial Neuron

2.0 Foundations of Deep Learning

2.1 MCP Neuron

2.2 Activation and Loss Functions

2.3 Learning/Training of Artificial Neural Networks

2.4 Layered Architectures and Matrix Notations: *Layered Networks, Multilayer Perceptron, Full Connected – Deep Neural Network(FC-DNN)*

3.0 More on Deep Learning Architectures: CNN and It's Variants

3.1 The Foundations of Generative AI: Sequence Modeling with RNN and It's Variants

3.2 Delay Differential Equation and RNN

4.0 Transformer Architectures; State of the Art in Generative AI:

4.1 Generative AI and Applications for STEM Research

5.0 Beyond Transformer Architectures: State Space Models and Mamba

1.0 The Biological Neurons in the Brain and Artificial Neuron

- A biological neuron(or neuron for short) is a specialized cell in living organisms(including humans) which is the building block of the brain and the entire nervous system.

The brain contains billions of neurons(biological neural network), which transmit information through electrical and chemical signals in living organisms.

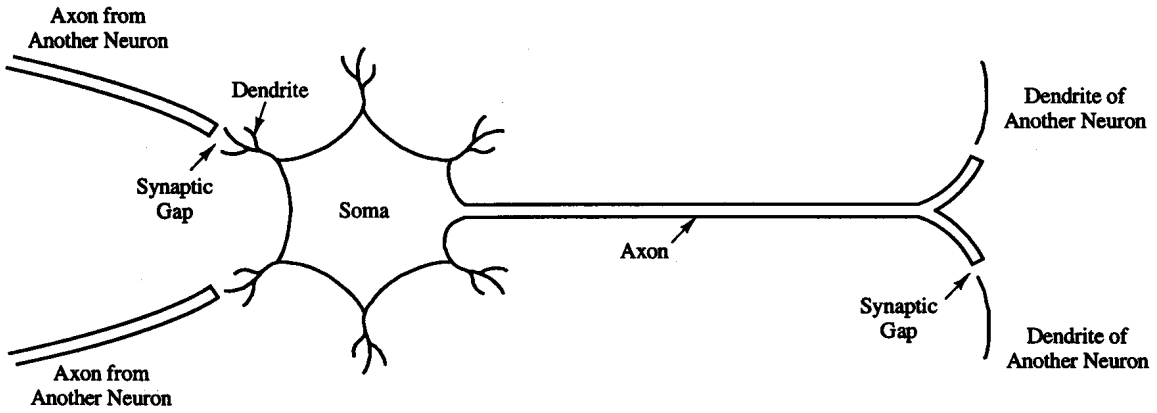
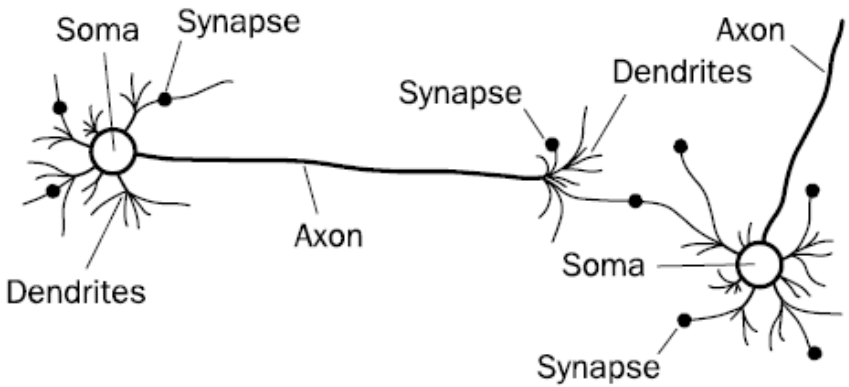
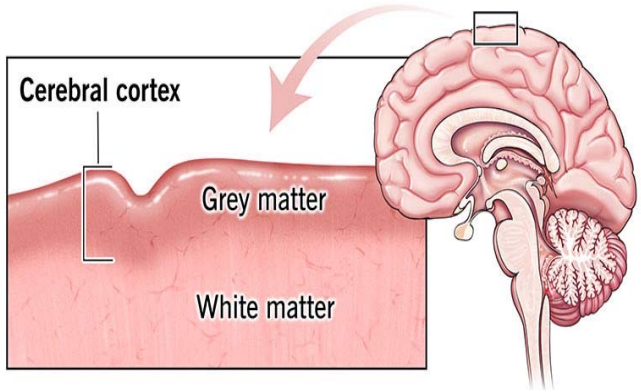


Figure 1.1 Biological Neuron

(Source: Fundamental of Neural Networks: Architectures, Algorithms and Application by Laurene Fausett,)

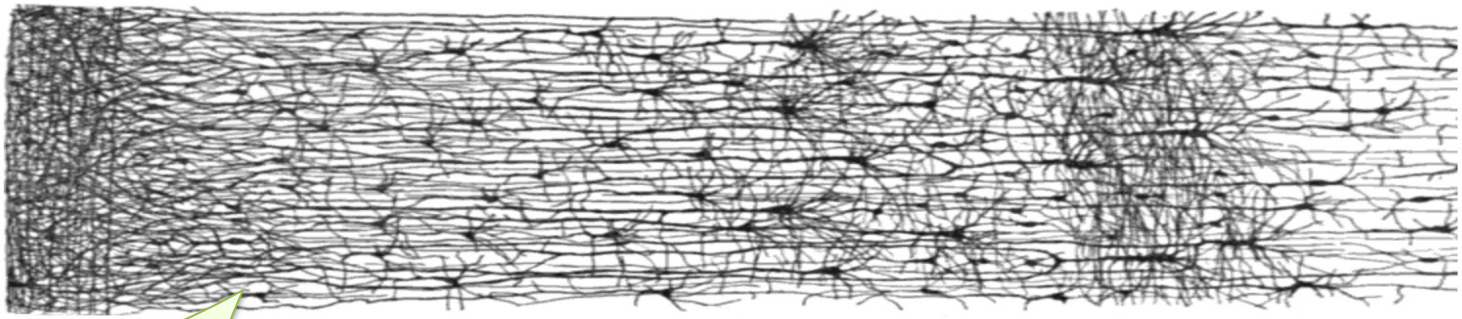


(a): Simple Biological Neural Network



(b): The Human Brain Showing the Cerebral Cortex

(Source:https://my.clevelandclinic.org/health/articles/23073-cerebral-cortex)



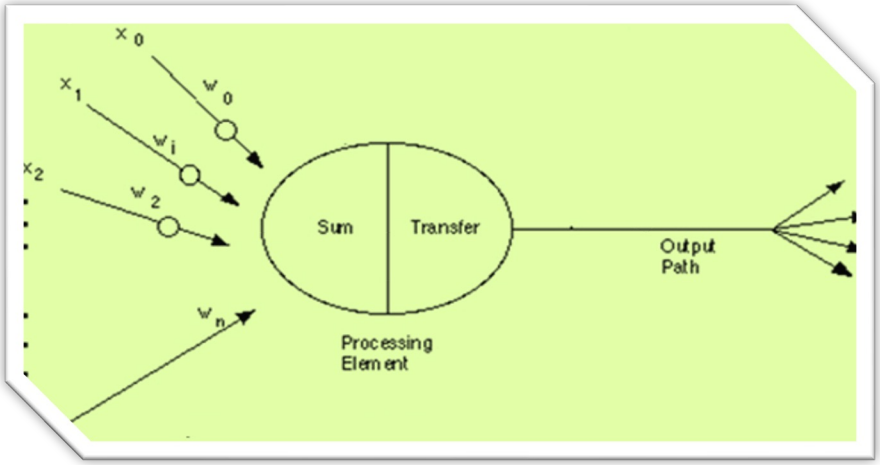
(c): Multiple Layers in Biological Neural Network

(Source: Neural Networks and Deep Learning by Aurélien Géron)

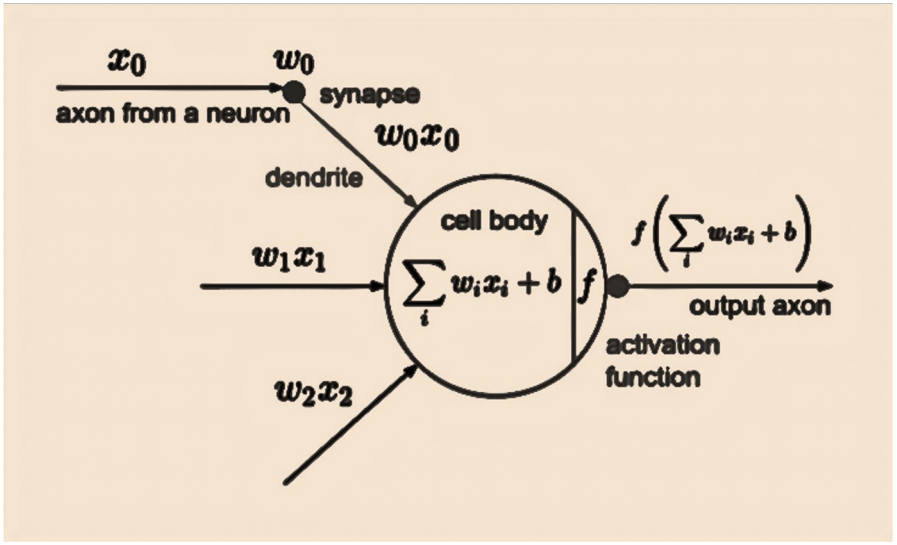
Approx. 14 to 16 billion neurons are found in the cerebral cortex.

Figure 1.2: Biological Neural Network

- **Artificial Neurons** are inspired by **biological neurons** and are simplified mathematical models designed to perform specific tasks.



a



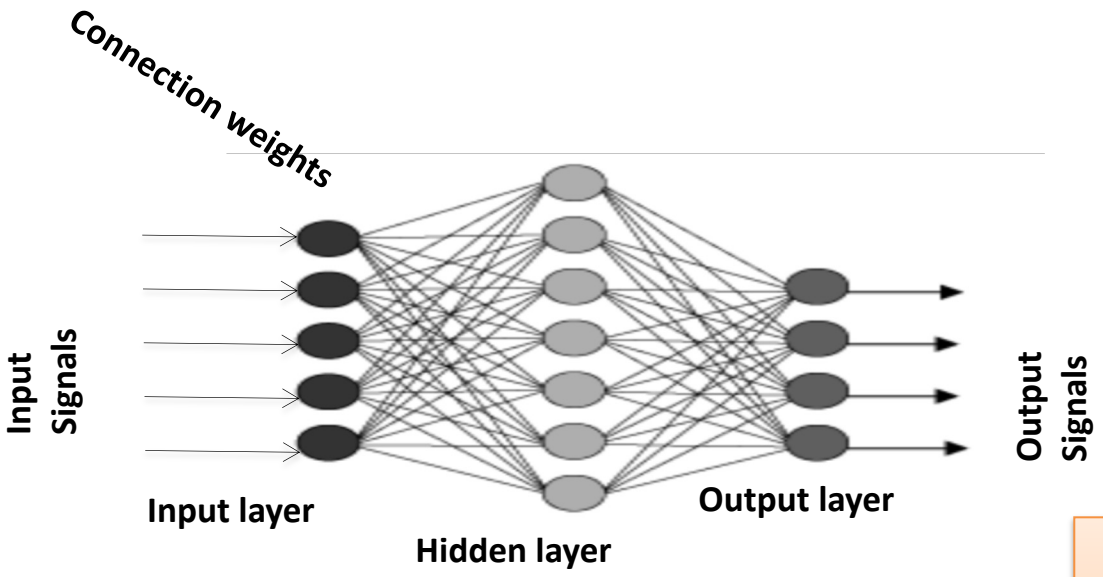
b

Figure 1.3: Artificial Neuron
(Biological Neuron Model)

- An **Artificial Neural Network (ANN)** was developed as a generalization of mathematical models of **neural biology**.

It is an information processing model that was inspired from the understanding of biological nervous system based on the following assumptions.

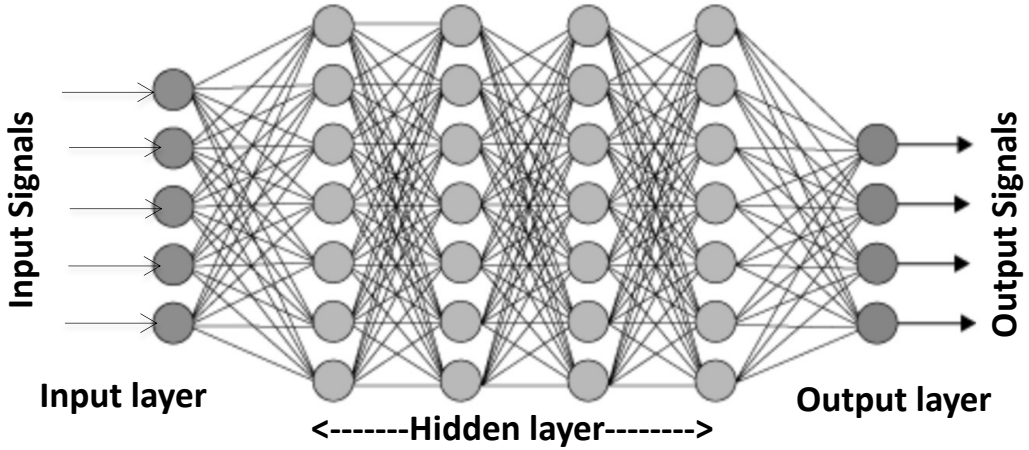
a) **Neurons are simple units in a nervous system at which information processing occurs.**
b) **Incoming information are signals that are passed between neurons through connection links.**
c) **Each connection link has a corresponding weight which multiplies the transmitted signal.**
d) **Each neuron applies an activation function to its net input which is the sum of weighted input signals to determine the output signal.**



a) Shallow Architecture

Exercise

What are the total number of parameterss in these network?



b) Deep Architecture

Figure 1.4: Typical Artificial Neural Network (ANN) Architectures

Table 1: Similarity/Analogy of BNN with ANN

S/N	Biological Neural Network	Artificial Neural Network (ANN)
1	Soma	Neuron
2	Dendrite	Input
3	Axon	Output
4	Synapse	Weight

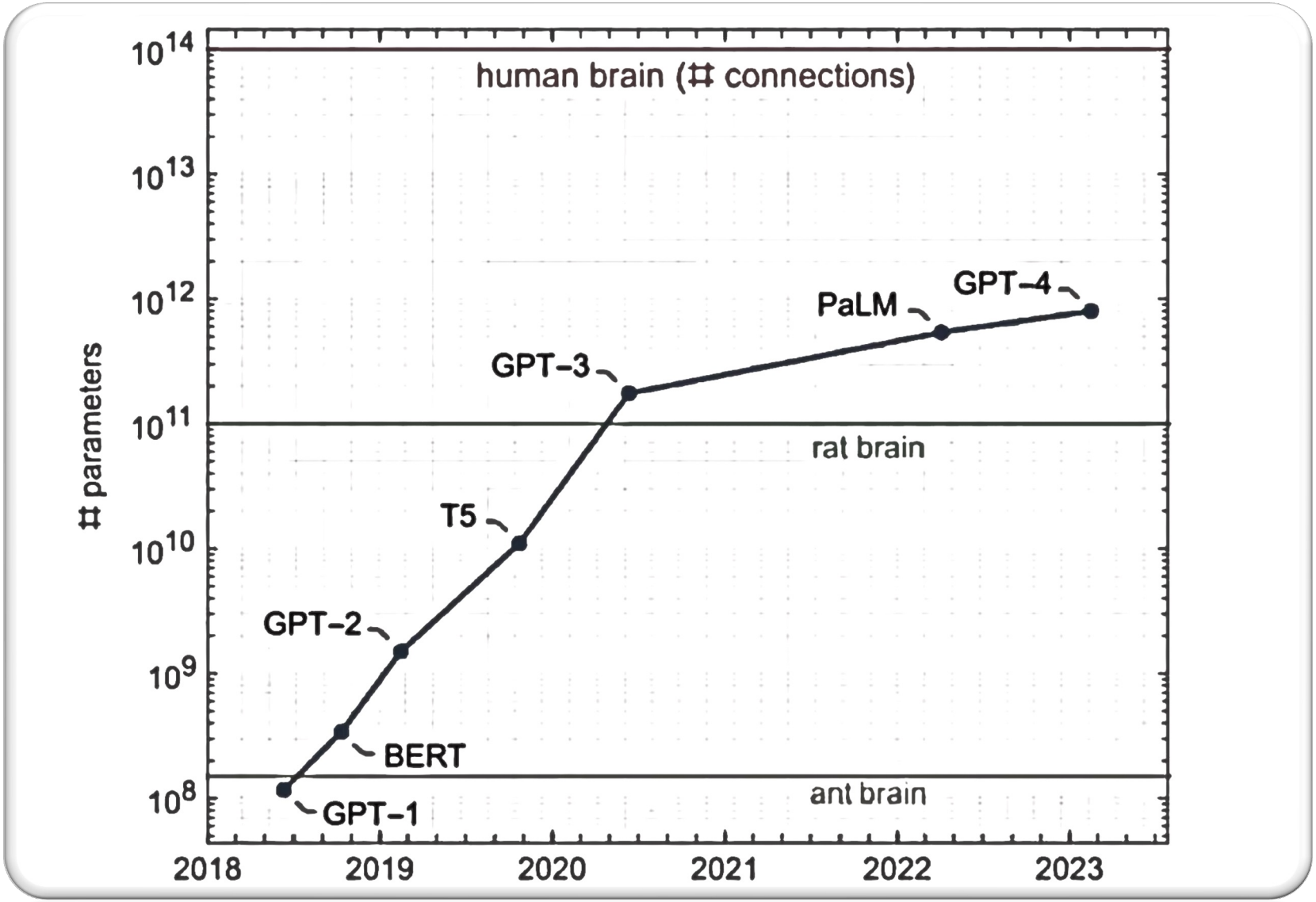


Figure 1.5: Comparison of the Number of Parameters in Modern GenAIs, Ant Brain, Rat Brain and Human Brain

(Source:Mirella Lapata(2023)-)

2.0 Foundations of Deep Learning/GenAI

2.1 The McCulloch-Pitt Neuron

- The McCulloch-Pitt (MCP) neuron is the first formal definition of a synthetic neuron.
  - *It was developed by Warren McCulloch(a Neurophysiologist) and Walter Pitts (a Mathematician) in 1943.*
  - *The model had a huge influence on the thought and studies that led to modern digital computer and AI systems.*
  - *The MCP is a Threshold neuron(i.e. it uses threshold activation function) and hence, the mathematical model is the same as Equation (1.1) i.e. :*

$$f(z) = \begin{cases} 1 & if\ z \geq T \\ 0 & if\ z < T \end{cases} \qquad (1.1)$$

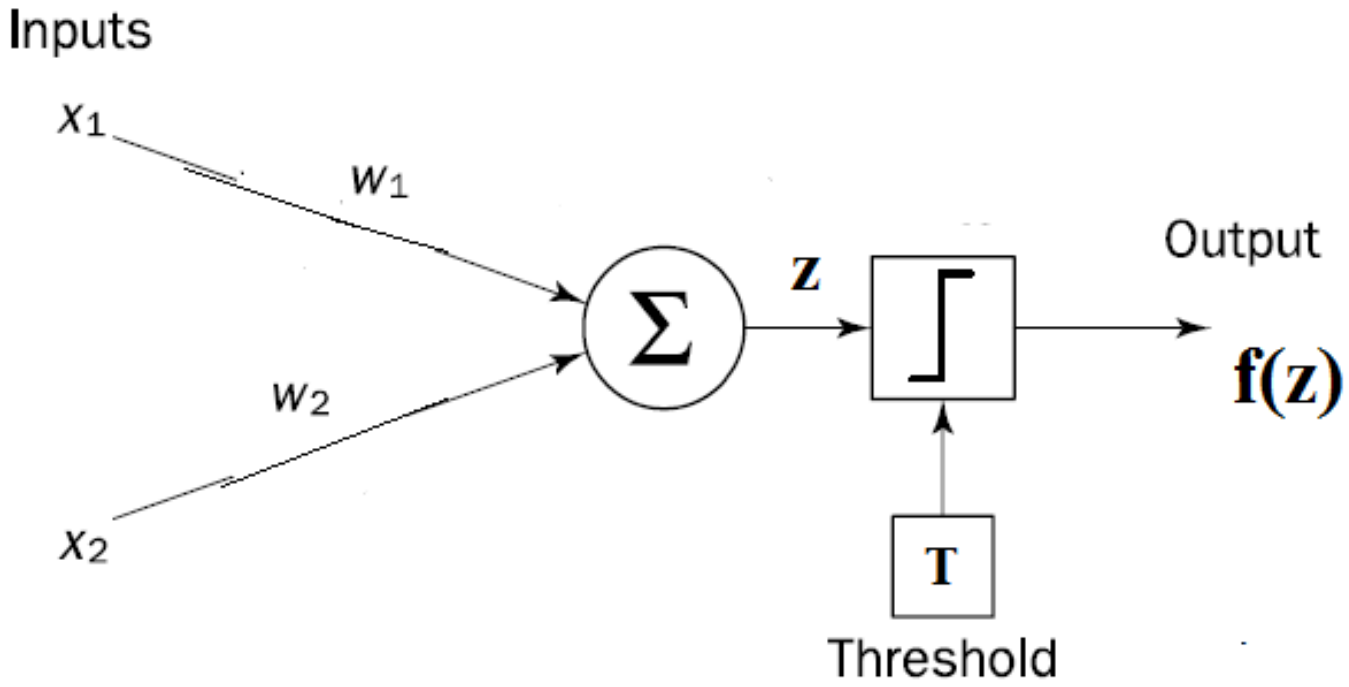


Fig 1.6: Architecture of McCulloch-Pitt Neuron

- Simple logical functions can be implemented directly with a single MCP neuron.

Weights and thresholds can be set to yield neurons which realise the logical functions AND, OR and NOT

"AND" logical function with MCP Neuron

Table 2: "AND" Truth Table

x_1	x_2	x_1 AND x_2
0	0	0
0	1	0
1	0	0
1	1	1

Exercise
Develop a MCP Neuron for the OR logic function.

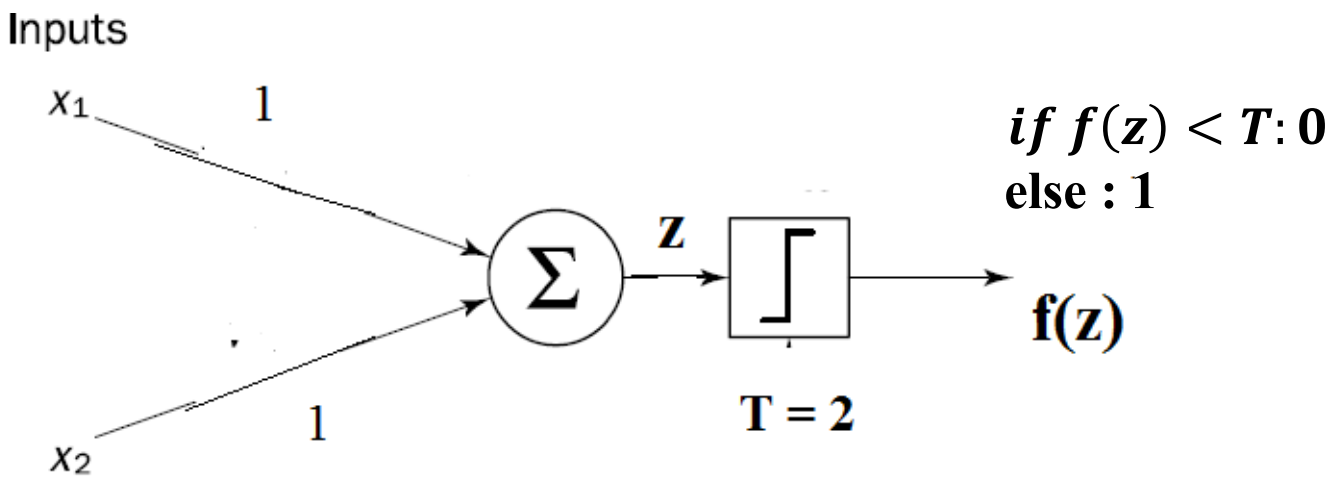


Fig 1.7: Architecture of McCulloch-Pitt Neuron for "AND" Logical Function

2.2 Activation and Loss Functions

2.2.1 Activation Functions

- The basic operation of an artificial neuron involves summing its weighted input signals and applying an activation (or transfer)function to generate an output.

1) Identity(linear) function

This function provides an output which is equal to the neuron weighted input . It is represented mathematically as:

$$f(z) = z \tag{1.2}$$

where z = sum of weighted inputs into the neuron.

2) Step functions

Also called the hard limit function. There are two types of step functions:

a)Binary step function:

Also known as Threshold or Heaviside function. It can be applied for binary classification tasks.

$$f(z) = \begin{cases} 1 & \text{If } z \geq T \\ 0 & \text{If } z < T \end{cases} \qquad (1.3)$$

where z = sum of weighted inputs into the neuron

T = threshold.

b) Bipolar step function

This is also called *sign function*. It is represented mathematically as:

$$f(z) = \begin{cases} 1 & \text{If } z \geq T \\ -1 & \text{If } z < T \end{cases} \qquad (1.4)$$

3) Sigmoid functions

- They are S-shaped curves that are highly useful in multi-layer networks. Some of the available types are:

a)Binary sigmoid function

This is also called **logistic function**. It is a sigmoid function with range from 0 to 1 and used for neural networks in which the desired output are either binary or are in the interval between 0 and 1.

It is represented as:

$$f(z) = \frac{1}{1+\exp(-\sigma z)} \qquad (1.5)$$

where z = *sum of weighted inputs.*

σ = *steepness parameter.*

b) Bipolar sigmoid function

- The range of output for this function is from -1 to 1. It can be used as the activation function when the desired output range is between -1 and 1. It is generically represented as:

$$f(z) = \frac{1-\exp(-\sigma z)}{1+\exp(-\sigma z)} \qquad (1.6)$$

(c) Softmax function

- It is often described as a combination of multiple sigmoids.
- The output is a range of values between 0 and 1, with the sum of the probabilities equal to 1. It is represented as:

$$f(z)_j = \frac{e^{Z_j}}{\sum_{k=1}^{K} e^{z_k}} \quad for\ j = 1,\ldots,K \qquad (1.7)$$

- The function can be used for multiclass classification where it returns the probabilities of each class, with the target class having the highest probability.

Exercise 1

Assuming the sum of weighted inputs of five neurons in the output layer of a multiclass classification task during inference are ***[4.2, 0.8, 0.57, 3.2, 5.4]***.

a) Manually compute the final outputs when the ***softmax function*** is used. Show that the sum of the outputs of the ***softmax operation*** (probability) is 1. What is the class of the input data?

b) In your Notebook on FEDGEN GPU, write a Python code to implement (a) using Numpy/CuPy.

Exercise
Engage Peer Problem Solving/Coding to Complete (a) and (b).

Part Solution

$$f(z_1) = \frac{exp(z_1)}{\sum_{k=1}^{5} exp(z_k)}$$

$$= \frac{exp(4.2)}{exp(4.2) + exp(0.8) + exp(0.57) + exp(3.2) + exp(5.4)}$$

$$= 0.2106$$

3) Rectified Linear Unit functions

- Both binary and bipolar sigmoid functions suffer from the ***vanishing gradients*** problem for large neural networks. This implies that the gradients computed during backpropagation are smaller than 1.

Standard ReLU function *was* developed by Nair and Hinton (2010).

– The most widely used activation function for deep learning applications with state-of-the-art performance. It is represented as:

$$f(z) = \max(0, z) = \begin{cases} z_i \quad if \quad z_i \geq 0 \\ 0 \quad if \quad z_i < 0 \end{cases} \tag{1.8}$$

-It rectifies the values of the inputs that are less than zero by forcing them to zero and if the input is greater than 0, the output is equal to the input.

Table 3: Selected Activation Functions in TensorFlow

Activation Function	TensorFlow Implementation
Binary Sigmoid	tf.keras.activations.sigmoid(z)
Bipolar Sigmoid(Tanh)	tf.keras.activations.tanh(z)
ReLU	tf.keras.activations.relu(z)
Softmax	tf.keras.activations.softmax(z)
Leaky ReLU	tf.keras.layers.LeakyReLU(alpha)

2.2.2 Loss Functions

- In ML, a *Loss Function* (or *Cost Function* or *Error Metric*) is a metric that is used to determine the performance of a model.
  - *It is often computed as the difference between the predicted and the actual values.*

- The different types of loss function are:

1) Distance Based Error

Given a set of input vector **x,** and assuming the actual output is **y,** and the predicted output is $\hat{\mathbf{y}}$, this error is calculated as:

$$\mathcal{E} = y - \hat{y} \tag{1.9}$$

2) L2 Loss or Mean Squared Error(MSE)

L2 loss overcomes the limitations of the distance-based error as it squares the value of error difference and eliminates any kind of negative value. MSE is represented as:

$$MSE = \frac{1}{N}\sum_{i=1}^{N}(\boldsymbol{y} - \hat{\boldsymbol{y}})^2 \qquad (2.0)$$

where N is the total number of observations taken.

3) Root Mean Squared Error (RMSE)

This is the square root of the MSE cost function, computed as follows:

$$RMSE = \sqrt{\frac{1}{N}\sum_{i=1}^{N}(\boldsymbol{y} - \hat{\boldsymbol{y}})^2} \qquad (2.1)$$

4) L1 Loss or Mean Absolute Error (MAE)

Similar to L2 loss or MSE. It computes the absolute difference between the actual and predicted values to avoid negative error and address the drawback of MSE:

$$MAE = \frac{1}{N}\sum_{i=1}^{N}|y - \hat{y}| \tag{2.2}$$

5) Cross Entropy or Log Loss Function

Measures the distance between two probability distributions p and q where p is the actual probability distribution and q is the predicted probability distribution. It is mathematically represented as:

$$H(\boldsymbol{x}) = -\sum_{i=1}^{N} p(\boldsymbol{x})\log q(\boldsymbol{x}) \tag{2.3}$$

The more the cross entropy, the lesser the accuracy of the model and conversely, a model with a cross entropy of 0 is said to be a perfect model.

Exercise 2

This codes illustrate the use of MAE, MSE and Cross Entropy with TensorFlow

```
#Import the required libraries
import numpy
import tensorflow as tf
from tensorflow.keras.losses import mean_absolute_error
from tensorflow.keras.losses import mean_squared_error
from tensorflow.keras.losses import categorical_crossentropy
##Define y_true and y_pred for a regression task with 3 Output Neurons
y_true = [1, 0, 1] #Hypothetical Target Values
y_pred = [0.5, 1, 0.8] #Hypothetical Predicted Values
#Compute and Print the MAE and MSE
print("\nThe MAE is - ", mean_absolute_error(y_true, y_pred).numpy())
print("\nThe MSE is - ", mean_squared_error(y_true, y_pred).numpy())
```

```
##Define y_true and y_pred for a classification task of 3 Output Neurons
#Using one hot encoding vector representation
y_true = [[1, 0, 0], [0, 1, 0], [0, 0, 1]] #Hypothetical Target Values
y_pred = [[0.80, 0.1, 0.1], [0.1, 0.1, 0.8], [0.8, 0.1, 0.1]] #Hypothetical Predicted Values
#Compute and print the categorical-cross-entropy-loss
print("\nThe Categorical Cross Entropy Loss is - ", categorical_crossentropy(y_true, y_pred).numpy())
```

2.3 Learning/Training of ANN

- Learning or the training of an ANN basically consists of the ***modification*** of ***its weights (and bias)*** through the ***application of learning algorithms*** when a group of ***input patterns*** is presented.

- Learning in ANN is an ***optimization problem.*** It refers to the task of either **minimizing** or **maximizing** an appropriate **cost function**. This involves finding **better weights (or parameters)** in order to reach the **lowest cost**.

- The three main categories of learning in ANN are:

i) Supervised learning: *e.g. classification, regression, transcription, translation etc*

ii) Unsupervised learning: *e.g. clustering, dimensionality reduction*

iii) Reinforcement learning: *e.g. games, robotics,*

- *Other emerging sub-categories of learning are: semi-supervised, self-supervised, transfer learning, representative learning, multi-task learning, federated learning etc.*

Table 4: Basic Categories of Learning in ANN

Category	Description	Advantages	Disadvantages
Supervised Learning	It learns from labelled data and with a supervisor or teacher. Error based (i.e. least mean square, gradient descent, backpropagation)	It is easy to implement and produces high accuracy	It requires labelled data can be computationally expensive
Unsupervised Learning	It learns from unlabelled data	It does not require labelled data and can discover hidden patterns	It is difficult to interpret results
Reinforcement Learning	It learns by interacting with an environment	It can solve complex problems and adapt to changes	It is difficult to design reward function and computationally expensive

2.3.1 Gradient Descent Algorithm

- It is the foundation for most supervised learning algorithms in use today.
- It requires the definition of an ***error (or objective) function*** to measure the neurons' errors and finds the weight values that minimizes the error.

$$if \quad E^p = \frac{1}{2}\sum_p \left(d_o^p - y_o^p\right)^2 \tag{2.4}$$

$$E = \sum_p E^p \tag{2.5}$$

where E = *error,* p = *number of patterns,* d_0 = *desired output and* y_0 = *actual output.*

- Taking the differential of the error with respect to the synaptic weight, we have:

$$\frac{\partial E}{\partial w_{oi}} = \frac{\partial E}{\partial y_o} \cdot \frac{\partial y_o}{\partial w_{oi}} \tag{2.6}$$

from (2.4) we have:

$$\frac{\partial E}{\partial y_o} = -(d_o - y_o) \tag{2.7}$$

for identity function for instance:

$$y_o = \sum_i w_{oi} x_i \tag{2.8}$$

$$\therefore \frac{\partial y_o}{\partial w_{oi}} = x_i \tag{2.9}$$

and for gradient descent(i.e. movement along the negative gradient – which is the path of steepest descent), (2.7) becomes:

$$\frac{\partial E}{\partial w_{oi}} = (d_o - y_o)x_i \tag{3.0}$$

which implies:

$$\Delta w_{oi}(t) = \eta(d_o - y_o)x_i \tag{3.1}$$

hence:

$$w_{oi}(t+1) = w_{oi}(t) + \Delta w_{oi}(t) \tag{3.2}$$

and putting (3.1) into (3.2), the updated weight for next iteration is:

$$w_{oi}(t+1) = w_{oi}(t) + \eta(d_o - y_o)x_i \tag{3.3}$$

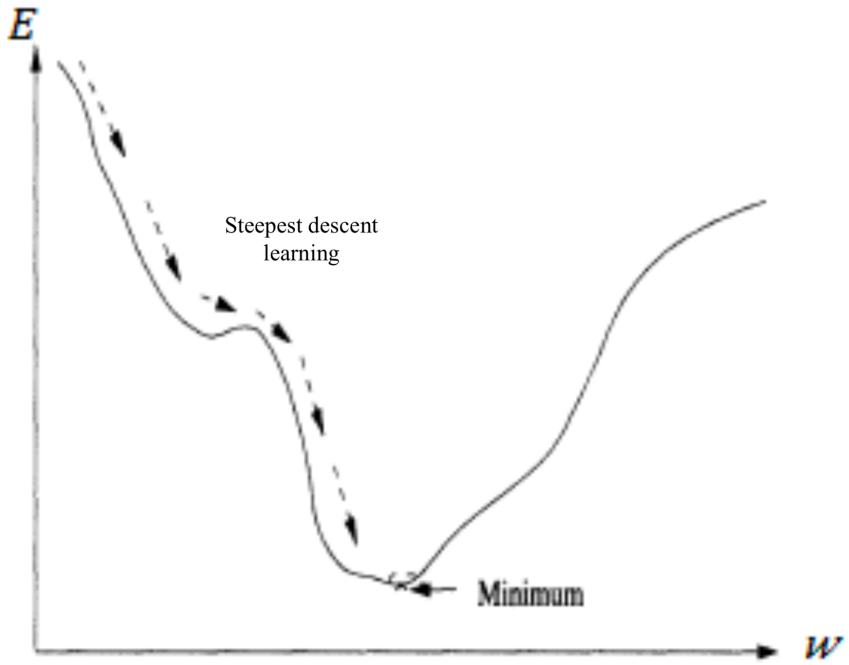
Equation (3.3) represents the gradient descent rule.

where;

$\boldsymbol{\eta = learning\ rate}$ *(it is a major hyperparameter in network training)*

$\boldsymbol{w_{oi}}$ **(t+1) = new weights**

Because is uses the steepest descent algorithm and identity activation function, it is also called the **Widrow-Hoff (Least Mean Square(LMS))** rule.



1.7: Steepest Descent Path

- In **supervised learning**, one set of weight modifications is called an **epoch** and many epochs may be required before the desired accuracy is reached.
- When the entire dataset is used to **compute the error surface**, and the gradient takes the path of steepest descent, we have **Batch Gradient Descent(BGD**).

- BGD works for a **simple quadratic error surface**, but in most practical cases, the error surface may be **a lot more complicated.**

- One of the solutions proposed for the pitfalls of BGD is the ***Stochastic Gradient Descent (SGD).***

- Variants of **SGD** are:
  - ***SGD with Momentum;*** and
  - ***SDG with Nesterov Momentum.***
- Other examples of ML optimization algorithms are:
  - ***AdaGrad;***
  - ***RMSProp;***
  - ***RMSProp with Nesterov Momentum;*** and
  - ***Adam***

- The **Perceptron** was the first **supervised neural model** developed by Frank Rosenblatt(1958).
- The Rosenblatt's perceptron consists of a linear combiner followed by a hard limiter.

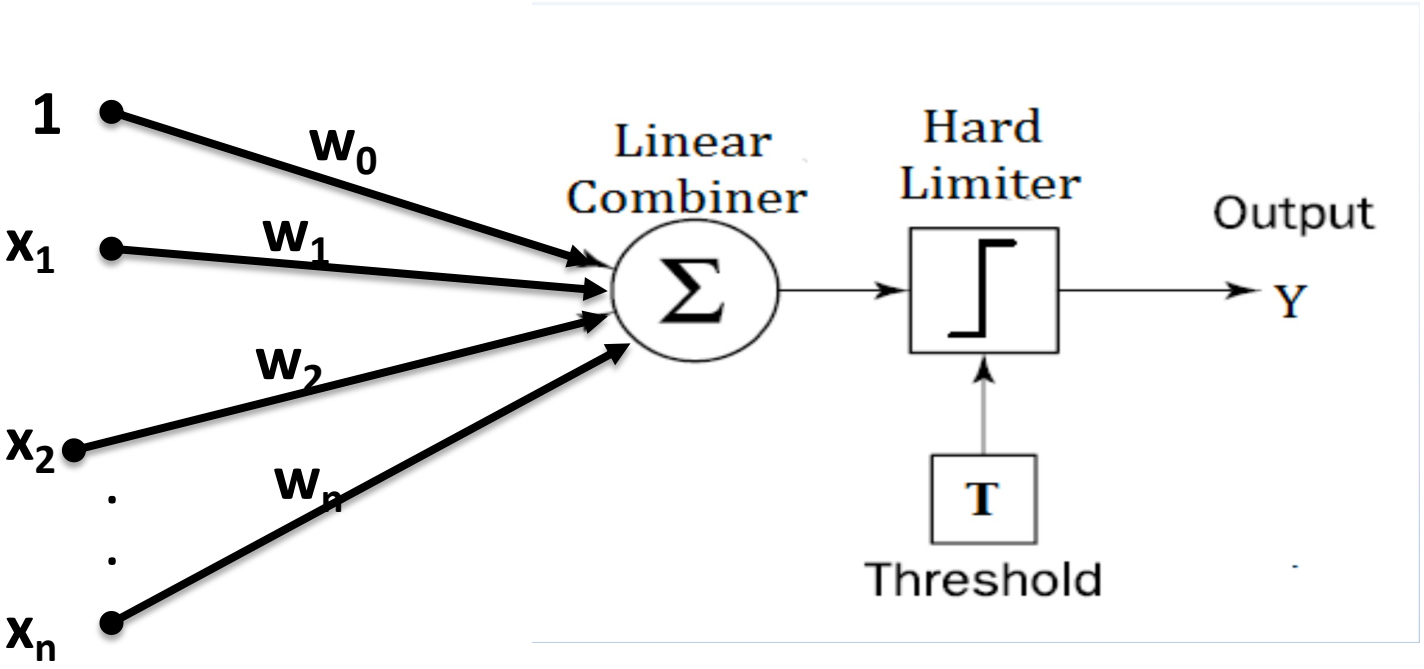


Fig. 1.8: The Frank Rosenblatt Perceptron

- The weighted sum of the inputs (x1, x2, ..., xn) is applied to the **hard limiter(or binary step function)**, which produces an output equal to +1 if the input is positive and 0 if it is negative.
- The weights are adjusted based on the **Perceptron rule,** which is a modified version of the gradient descent.

Exercise 3

Develop a Python Code with Sklearn framework for classification of the first two species of the Iris Dataset.

```
#Import required libraries
from sklearn.datasets import load_iris
from sklearn.linear_model import Perceptron
from sklearn.model_selection import train_test_split
from sklearn.metrics import accuracy_score
from sklearn.metrics import confusion_matrix
#Load the Iris dataset
iris = load_iris()
#View the iris data keys
iris.keys()
```

```
#it is a key of  dictionary iris
iris.target_names
##Convert data and target into a data frame.
#Extract the First 100 Features
X = pd.DataFrame(data = iris.data[:100,:], columns =
iris.feature_names)
X.head(100) #Inspect the features
#Target
y = pd.DataFrame(data=iris.target[:100], columns = ['irisType'])
y.head(100)
# Split the dataset into training and testing sets
X_train, X_test, y_train, y_test = train_test_split(X, y, test_size=0.2,
random_state=42)
```

```
# Create a Perceptron classifier
mdlPercept = Perceptron()
# Train the classifier
mdlPercept.fit(X_train, y_train)
# Make predictions on the test set
y_pred = mdlPercept.predict(X_test)
# Calculate accuracy of the classifier
accuracy = accuracy_score(y_test, y_pred)
print("Accuracy:", accuracy)
#Generate the Confusion Matrix
print(”Perceptron Confusion Matrix:\n",confusion_matrix(y_test, y_pred))
```

2.4 Layered ANN Architectures and Matrix Notations

- Modern **deep neural networks/deep learning** are more powerful versions of shallow layered networks with the power achieved through the combination of the basic components (so far presented) into a comprehensive neural architecture.

2.4.1 Network with Single and Multiple Layers of Neurons

- Neurons in ANN are normally connected to form a single or multiple layers.
- A network with single layer of multiple neurons is shown in Figure 1.8.
- Each of the ***R* inputs** is connected to each of the neurons and the weight matrix **W** has *S* rows. The input vector **x** elements enter the network through the weight matrix **W**:

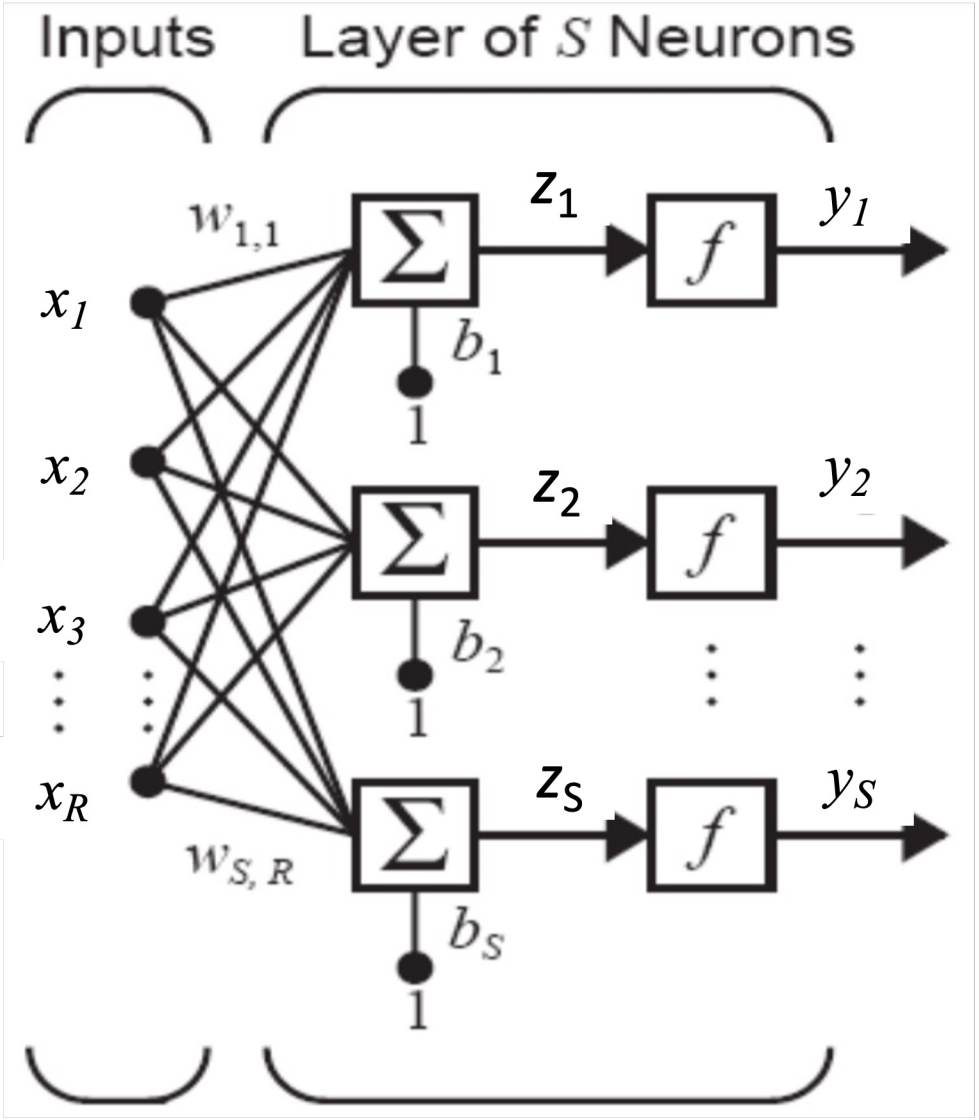


Figure 1.8: A Single Layer of *S* neurons

$$\mathbf{x}=\begin{bmatrix} x_1 \\ x_2 \\ . \\ . \\ . \\ x_R \end{bmatrix}, \quad \mathbf{W} = \begin{bmatrix} w_{1,1} & w_{1,2} & \cdots & w_{1,R} \\ w_{2,1} & w_{2,2} & \cdots & w_{2,R} \\ \vdots & \vdots & & \vdots \\ w_{S,1} & w_{S,2} & \cdots & w_{S,R} \end{bmatrix}, \quad \mathbf{b}=\begin{bmatrix} b_1 \\ b_2 \\ . \\ . \\ . \\ b_S \end{bmatrix}$$

- The output for the **single-layer network** can be represented with **matrix notation** as:

$$\mathbf{z = Wx + b} \tag{3.4}$$

$$\mathbf{y} = \mathbf{f}(\mathbf{z}) \tag{3.5}$$

where,

$$\mathbf{z}=\begin{bmatrix} z_1 \\ z_2 \\ . \\ . \\ . \\ z_S \end{bmatrix}, \qquad \mathbf{y}=\begin{bmatrix} y_1 \\ y_2 \\ . \\ . \\ . \\ y_S \end{bmatrix}$$

- A network of multiple layers(i.e. three), with multiple neurons in each layer is shown in Fig. 1.9.
- Each layer has its own **weight matrix W, bias vector b,** a **net input vector z** and an **output vector y.**
- Different layers can have different numbers of neurons.
- **Output layer** is a layer whose output is the network output.
- **Hidden layers** are other layers between the input neurons and the output layer.

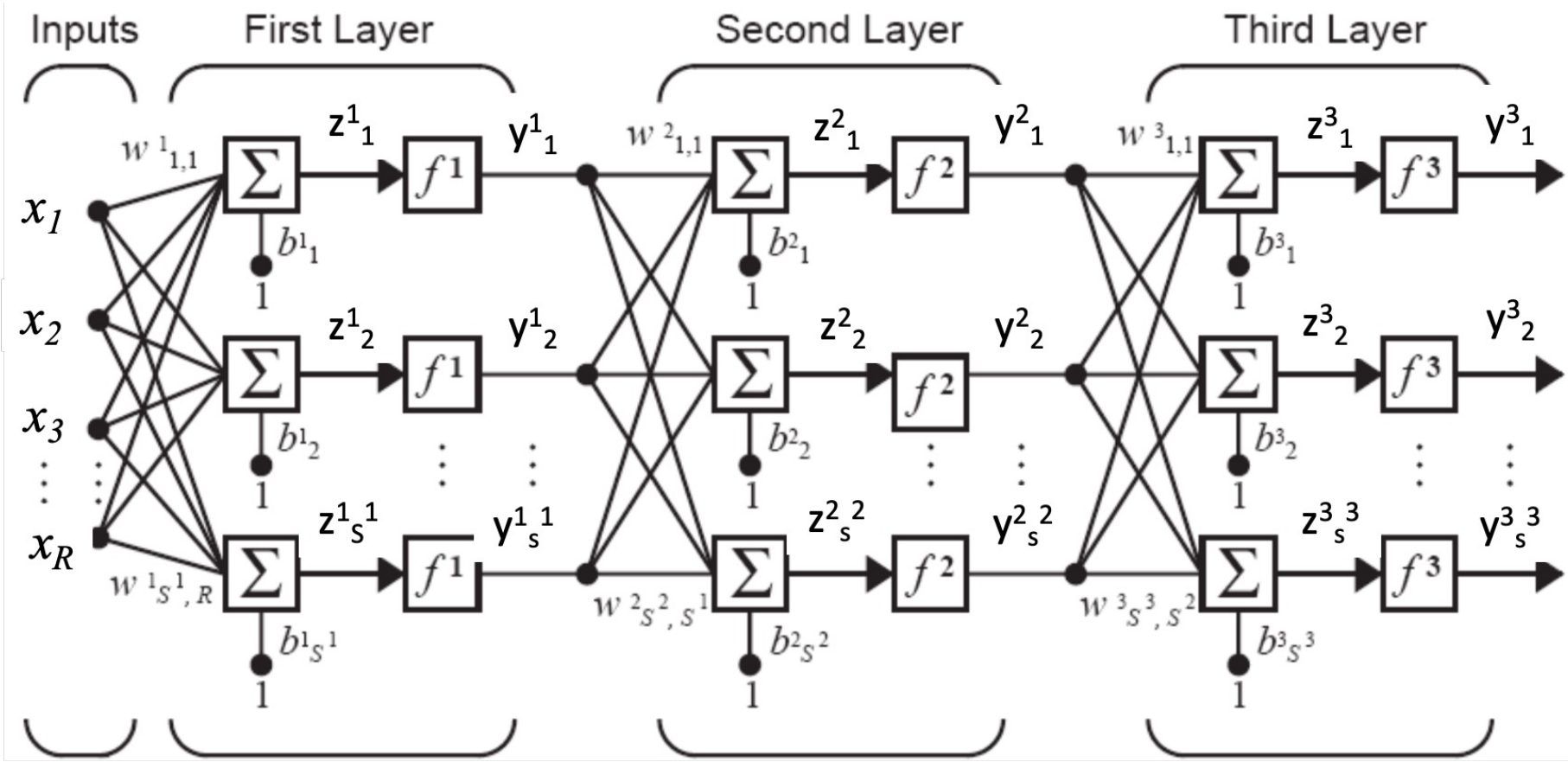


Fig. 1.9: A 3-layer Artificial Neural Network

first layer weight – $\mathbf{W}^1$
second layer weight – $\mathbf{W}^2$
third layer weight – $\mathbf{W}^3$

first layer bias – $\mathbf{b}^1$
second layer bias – $\mathbf{b}^2$
third layer bias – $\mathbf{b}^3$

first layer net input – $\mathbf{z}^1$
second layer net input – $\mathbf{z}^2$
third layer bias net input – $\mathbf{z}^3$

first layer output– $\mathbf{y}^1$
second layer output – $\mathbf{y}^2$
third layer output – $\mathbf{y}^3$

- First layer output:

$$\mathbf{y}^1 = \mathbf{f}^1(\mathbf{W}^1\mathbf{x} + \mathbf{b}^1) \quad (3.6)$$

- Second layer output:

$$\mathbf{y}^2 = \mathbf{f}^2(\mathbf{W}^2\mathbf{y}^1 + \mathbf{b}^2) \quad (3.7)$$

- Third layer output which is the network output is:

$$\mathbf{y}^3 = \mathbf{f}^3(\mathbf{W}^3\mathbf{y}^2 + \mathbf{b}^3) \quad (3.8)$$

Put (3.6) and (3.7) into (3.8)

$$\mathbf{y}^3 = \mathbf{f}^3(\mathbf{W}^3\mathbf{f}^2(\mathbf{W}^2\mathbf{f}^1(\mathbf{W}^1\mathbf{x} + \mathbf{b}^1) + \mathbf{b}^2) + \mathbf{b}^3) \quad (3.9)$$

2.4.2 Multilayer Perceptron and Fully Connected Deep Neural Network

- **Multi-layer Perceptron(MLP)** is a multilayered **supervised feedforward ANN architecture** that maps a set of input dataset $X = x_1, x_2, \ldots, x_n$ onto an output or a target y. It can be used for both classification and regression tasks.

- The *target* could be a scalar or vector $Y = (y_1, y_2, \ldots y_m)$ depending on the tasks, which could be classification or regression.

- MLP consists of three distinct layers, namely: **i) input layer, ii) hidden layer** and **iii) output layer** and they are fully connected.

- The neurons in the layers uses **non-linear activation functions**, except for the **input layer neurons**

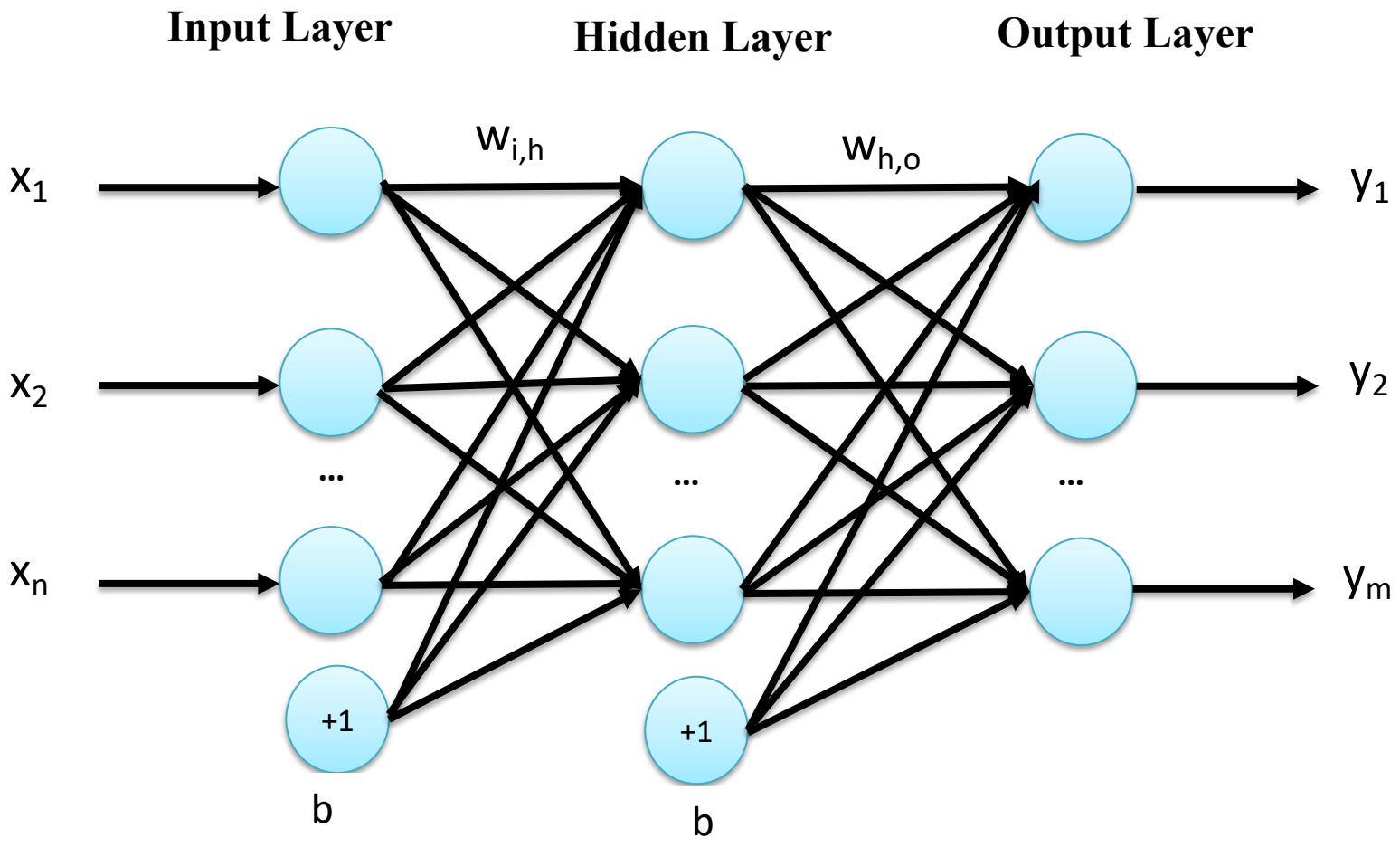


Fig. 2.0: Multilayer Perceptron Architecture with Single Hidden Layer

- Variants of **gradient descent** algorithm such as *SGD*, *SGD with momentum*, *Adam* and etc are normally used to obtain **optimal weights and biases** (parameters) when training MLP with **Backpropagation** algorithm.

- **Sklearn** can be used to implement MLP. It can also be used with the **TensorFlow** framework.

- Some of **hyperparameters** that could be tuned when using Sklearn to build MLP models are described in Table 5.

Table 5: Some Key Sklearn Hyperparameters for MLP

S/N	Hyperparameters	Description	Example
1	**hidden_layer_sizes**	This hyperparameter defines the MLP structure. It accepts a list where each element represents the number of neurons in each of the hidden layers.	**hidden_layer_sizes=(10, 5)** creates a network with two hidden layers with 10 and 5 neurons, respectively.
2	**activation**	This specifies the activation function for the hidden layers. Available Sklearn activation functions in Sklear are: **identity** - no activation; **logistic** - sigmoid function; **tanh** - hyperbolic tangent function; and **relu** - rectified linear unit function.	**activation='relu'**

S/N	Hyperparameters	Description	Example
3	**solver**	This hyperparameter determines the optimization algorithm used to train the neural network. The solvers in Sklearn are: **lbfgs** - limited-memory quasi-Newton code for bound-constrained optimization; **sgd** - stochastic gradient descent, and **adam** - adaptive moment estimation.	**solver='adam'**
4	**max_iter**	This specifies the maximum number of iterations or epochs for training the MLP.	**max_iter=1000**
5	**random_state**	This hyperparameter is for setting the random seed for reproducible results.	**random_state=42**

Exercise 4

Develop a classification model using the Iris dataset in Sklearn and the MLP Classifier.

Solution

```
# Import libraries
from sklearn.neural_network import MLPClassifier
from sklearn.model_selection import train_test_split
from sklearn.datasets import load_iris
from sklearn.metrics import accuracy_score, mean_squared_error, classification_report
import matplotlib.pyplot as plt

# Load the Iris dataset
iris = load_iris()
X = iris.data
y = iris.target

# Split the dataset into training and testing sets
X_train, X_test, y_train, y_test = train_test_split(X, y, test_size=0.2, random_state=42)
```

```
# Create an MLP classifier with specified hyperparameters
mlp_classifier = MLPClassifier(
    hidden_layer_sizes=(10, 5),  # Number of neurons in each hidden layer
    activation='relu',           # Activation function
    solver='adam',               # Solver for weight optimization
    alpha=0.0001,                # L2 penalty (regularization term) parameter
    learning_rate='constant',    # Learning rate schedule
    max_iter=1000,               # Maximum number of iterations
    batch_size='auto',           # Size of mini-batches
    early_stopping=False,        # Whether to use early stopping
    validation_fraction=0.1, # Proportion of training data for validation set
    tol=1e-4                     # Tolerance for optimization
    )
```

```
# Train the classifier on the training set
mlp_classifier.fit(X_train, y_train)

# Make predictions on the test set
y_pred = mlp_classifier.predict(X_test)

# Evaluate the performance of the classifier
accuracy = accuracy_score(y_test, y_pred)
print(f"Accuracy: {accuracy:.2f}")

# Calculate Mean Squared Error
mse = mean_squared_error(y_test, y_pred)
print(f"Mean Squared Error: {mse:.2f}")

# Display classification report
print("Classification Report:")
print(classification_report(y_test, y_pred))
```

- Tradionally, once MLP is configured beyond three hidden layers, it starts overfitting unseen data samples.
- However, the MLP architecture is the precursor of the **Fully Connected Deep Neural Network (FC-DNN)**.

- In **FC-DNN** each neuron in one layer is connected to every neuron in the next layer

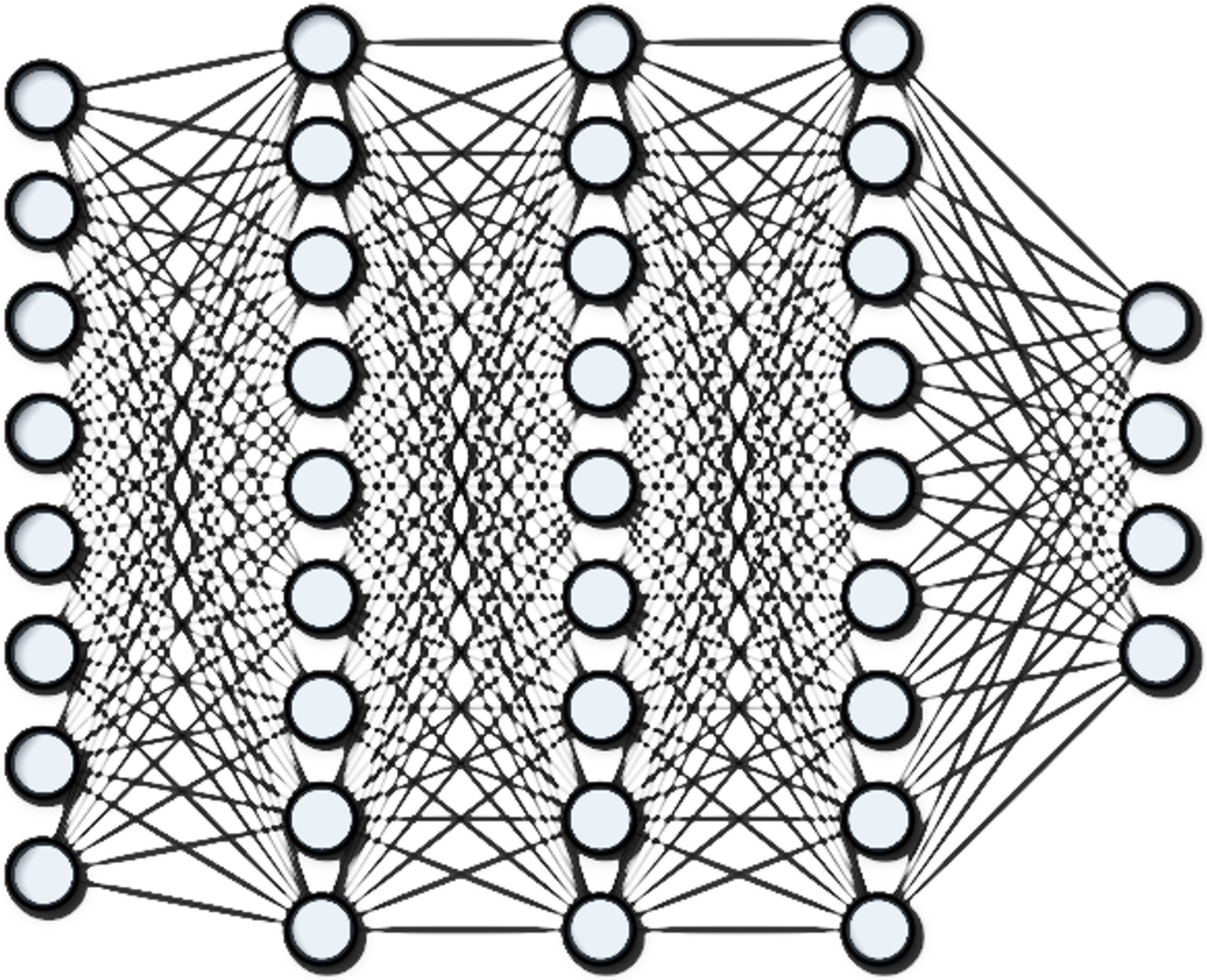


Fig. 2.1: Architecture of a Typical Fully Connected-Deep Neural Network

Exercise 5

Develop a FC-DNN classification model using the Iris dataset in TensorFlow.

Solution

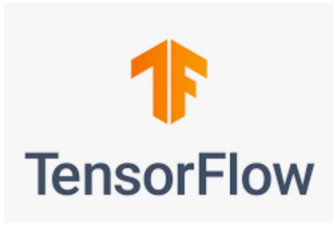
```
#Import all required libraries
from sklearn.datasets import load_iris
from sklearn.model_selection import train_test_split
from tensorflow.keras.models import Sequential
from tensorflow.keras.layers import Dense
from tensorflow.keras.optimizers import Adam

# Load the Iris dataset
iris = load_iris()

# Separate data features and target labels
X = iris.data
y = iris.target

# One-hot encode the target labels
from tensorflow.keras.utils import to_categorical
y = to_categorical(y, num_classes=3)

# Split the data into training and testing sets
X_train, X_test, y_train, y_test = train_test_split(X, y, test_size=0.20, random_state=42)
```

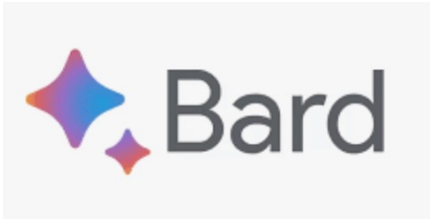
```
# Define the FC-DNN model
model = Sequential()
model.add(Dense(10, activation='relu', input_shape=(4,)))
model.add(Dense(10, activation='relu'))
model.add(Dense(3, activation='softmax'))

# Define the optimizer and compile the model
optimizer = Adam(lr=0.001)
model.compile(loss='categorical_crossentropy', optimizer=optimizer, metrics=['accuracy'])

# Train the model
model.fit(X_train, y_train, epochs=100, batch_size=32)

# Evaluate the model on the test set
loss, accuracy = model.evaluate(X_test, y_test)
print("\n")
print(f"Test loss: {loss:.4f}, Test accuracy: {accuracy*100:.4f}")
```

Next Step

3.0 More on Deep Learning Architectures: *CNN and It's Variants*

Thanks for Listening





jupyternb-gpu.fedgen.net