Neural Networks

and

Deep Learning

Topics:

- Mcculloch-Pitts Neurons
- Fully Connected Neural Networks

Questions:

- Mcculloch-Pitts Neural Networks
- AdaLine and MAdaLine Neural Networks
- Auto-Encoders for Classification
- Multi-Layer Perceptron Neural Networks

Question 1. Mcculloch-Pitts Neural Networks

In this question, we aim to first become familiar with **deterministic finite automaton (DFA)** and then design a **neural network** for it.

Deterministic Finite Automaton (DFA)

In simple terms, a deterministic finite automaton (DFA) can be thought of as a black box that receives input and announces it in the output if it detects a specific pattern in the inputs. It uses a set of states to store the observed patterns.

Example. Consider a deterministic finite automaton that can recognize the pattern "100" at least once in the alphabet $\{0, 1\}$. After observing the first "100" it remains in the accepting state. The state diagram of the deterministic finite automaton is shown in **Figure 1**.

- The number inside each circle represents the state number.
- The numbers on the edges are the inputs that cause the current state to transition to the next state.
- The process starts from state number zero.
- If the inputs are exhausted and we are in a state that is double-lined (state three), the desired input pattern has been detected by the machine (it is accepted).

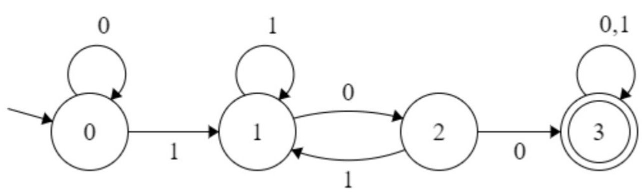


Figure 1. A DFA State Diagram

Consider the input $\mathbf{011001}$.

1. Initially, in state zero with input 0 ($\mathbf{011001}$), we stay in **state zero**.
2. Then, in state zero with the next input 1 ($\mathbf{011001}$), we move to **state one**.
3. Now, in state one with input 1 ($\mathbf{011001}$), we return to **state one**.
4. In state one with input 0 ($\mathbf{011001}$), we move to **state two**.
5. In state two with input 0 ($\mathbf{011001}$), we move to **state three**.
6. In state three with input 1 ($\mathbf{011001}$), we return to **state three**.

Since the inputs are exhausted and we have remained in **state three**, the input string is accepted, and the pattern has been recognized by the deterministic finite automaton.

Now we can draw the state transition table for the deterministic finite automaton as **Table 1**.

Now we are going to simulate the given DFA using an extended McCulloch-Pitts neuron. The **present state** and **input of the DFA** will be considered as the input to the neural network, while the **next state** and whether the state is **accepting** ($acceptance = 1$, $non-acceptance = 0$) will be considered as the output of the neural network. (Three input neurons and three output neurons.)

Note that the state numbers, inputs, and whether the states are accepting or not are all **binary**. Also, the timing order of operations in this question is not important, so there is no need to consider delays for operations.

Present State	DFA Input	Next State	Acceptance
0	0	0	0
0	1	1	0
1	0	2	0
1	1	1	0
2	0	3	1
2	1	1	0
3	0	3	1
3	1	3	1

Table 1. State Transition Table of the Deterministic Finite Automaton.

Now we simplify the state transition table by converting the inputs and outputs to binary form, as shown in **Table 2**.

Present State [1]	Present State [0]	DFA Input	Next State [1]	Next State [0]	Acceptance
0	0	0	0	0	0
0	0	1	0	1	0
0	1	0	1	0	0
0	1	1	0	1	0
1	0	0	1	1	1
1	0	1	0	1	0
1	1	0	1	1	1
1	1	1	1	1	1

Table 2. Binary Form of State Transition Table of the Deterministic Finite Automaton.

To facilitate the design of the desired network using McCulloch-Pitts networks, we will derive the **logical equation** for each of the outputs in terms of the inputs.

$$\boldsymbol{NS}\,[\mathbf{1}] = PS[1].PS[0] + PS[1].\overline{DFA} + PS[0].\overline{DFA}$$

$$\boldsymbol{NS}\,[\mathbf{0}] = PS[1] + DFA$$

$$\boldsymbol{ACC} = PS[1].PS[0] + PS[1].\overline{DFA}$$

Figure 2 shows the McCulloch-Pitts neural network corresponding to the logical gates we need. In these networks, the threshold is set to **2** ($\theta = 2$).

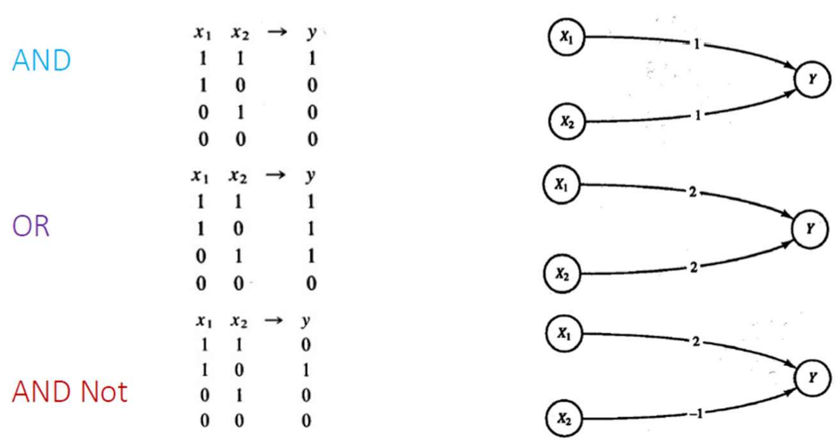
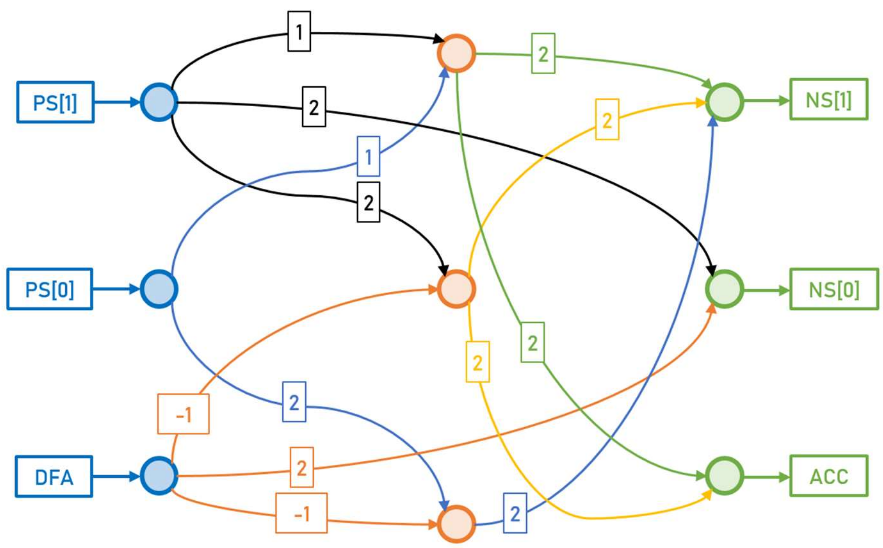
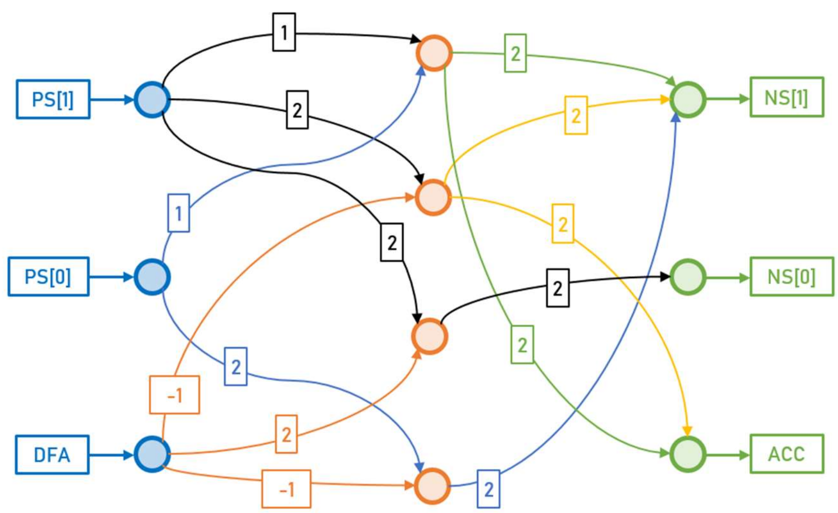


Figure 2. Required Logic Functions by McCulloch-Pitts Neural Network (θ=2)

The neural network designed using the McCulloch-Pitts network with $threshold = \theta \ = \ 2$ is shown in **Figure 3**.



a. Recurrent Neural Network (RNN)



b. Feed-Forward Neural Network

Figure 3. Designed Neural Networks using McCulloch-Pitts Neural Network (θ=2)

Using **Python**, we will implement the designed (Feed-Forward) network and display the output for all states for all inputs as shown in **Figure 4**. (**File Name:** DFA.py)

	Present State	DFA Input	Next State	Acceptance
0	0	0	0	0
1	0	1	1	0
2	1	0	2	0
3	1	1	1	0
4	2	0	3	1
5	2	1	1	0
6	3	0	3	1
7	3	1	3	1

Figure 4. The Output for All States for All Inputs (**Python** Code Result)

Question 2. AdaLine and MAdaLine Neural Networks

In this question, we will become familiar with the **AdaLine** and **MadaLine** networks.

2.1. AdaLine Neural Network

Suppose our data in two dimensions is defined as follows: (x, y)

- $\boldsymbol{x}$: a normally distributed variable with mean μ_x and standard deviation σ_x
- $\boldsymbol{y}$: a normally distributed variable with mean μ_y and standard deviation σ_y

Now we consider two groups as follows:

Group one: Contains 100 data points, where the variable x has a mean of 0 and a standard deviation of 0.1, and the variable y also has a mean of 0 and a standard deviation of 0.4.

$$x \sim \mathcal{N}(0, 0.1)\,, \qquad y \sim \mathcal{N}(0, 0.4)$$

Group two: Contains 100 data points, where the variable x has a mean of 1 and a standard deviation of 0.2, and the variable y also has a mean of 1 and a standard deviation of 0.2.

$$x \sim \mathcal{N}(1, 0.2)\,, \qquad y \sim \mathcal{N}(1, 0.2)$$

The data for both groups is shown in **Figure 5**.

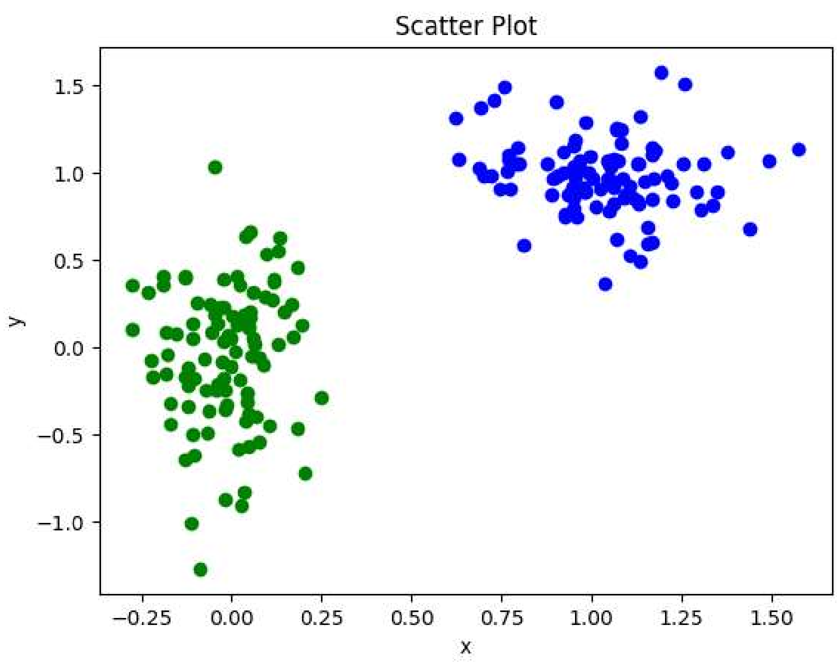


Figure 5. The data for both groups

Now, following the **AdaLine learning** algorithm, we first initialize the weights, bias, and learning rate with random and small values, and then update the weights and biases until the cost function decreases below a predetermined threshold. The changes in the cost function based on the samples are shown in **Figure 6**.

As you can see, the cost function has decreased as learning progresses. The method of separating the two classes using the AdaLine network is shown in **Figure 7**. (accuracy = 100%)

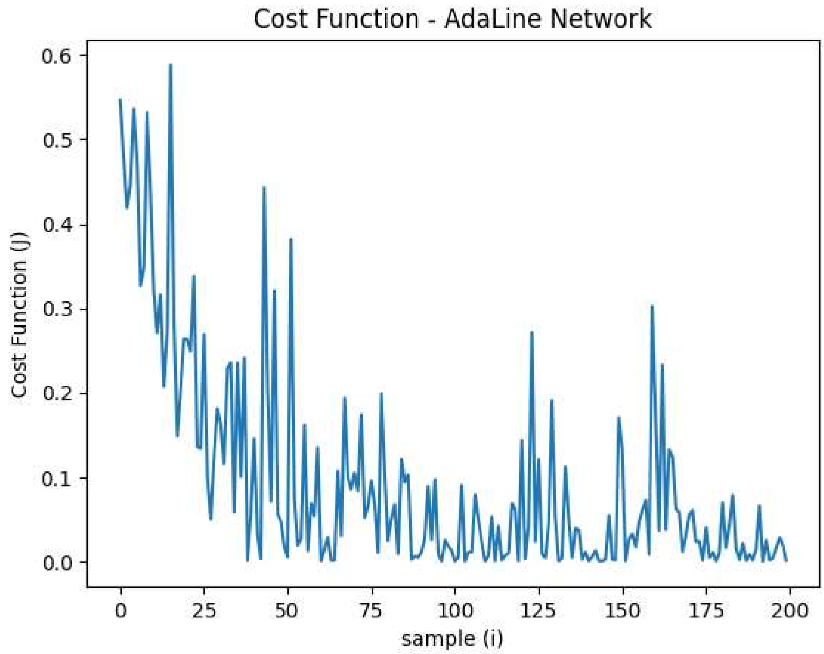


Figure 6. The AdaLine Network Cost Function

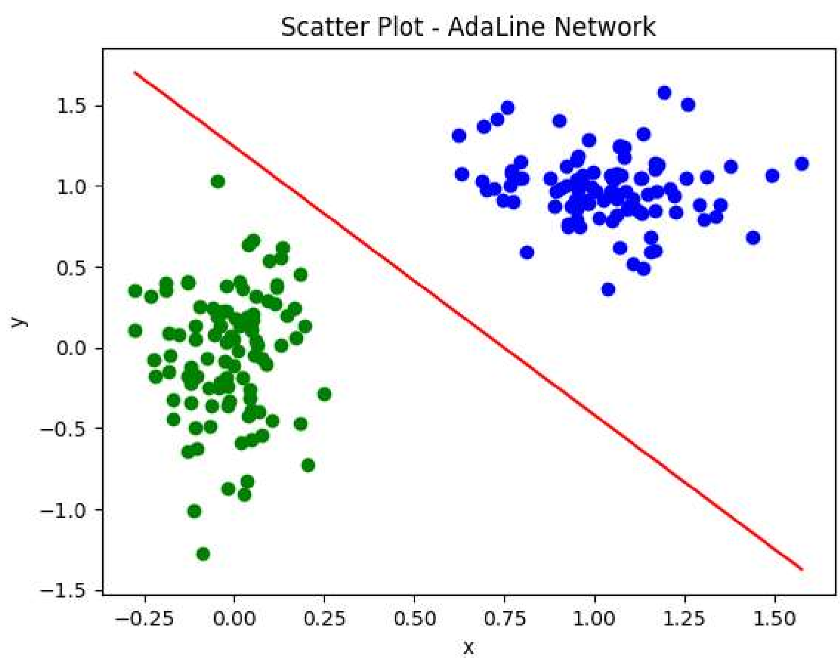


Figure 7. AdaLine Network and the train data

Now suppose we define two new data groups as follows:

Group one: Contains 100 data points, where the variable x has a mean of 0 and a standard deviation of 0.4, and the variable y also has a mean of 0 and a standard deviation of 0.4.

$$x \sim \mathcal{N}(0, 0.4)\,, \qquad y \sim \mathcal{N}(0, 0.4)$$

Group two: Contains 100 data points, where the variable x has a mean of 1 and a standard deviation of 0.3, and the variable y also has a mean of 1 and a standard deviation of 0.3.

$$x \sim \mathcal{N}(1, 0.3)\,, \qquad y \sim \mathcal{N}(1, 0.3)$$

We want to test the designed AdaLine network on the new data whose variance has increased. The result is shown in **Figure 8**. As expected, the separation for the new data also has suitable accuracy. (accuracy = 96%)

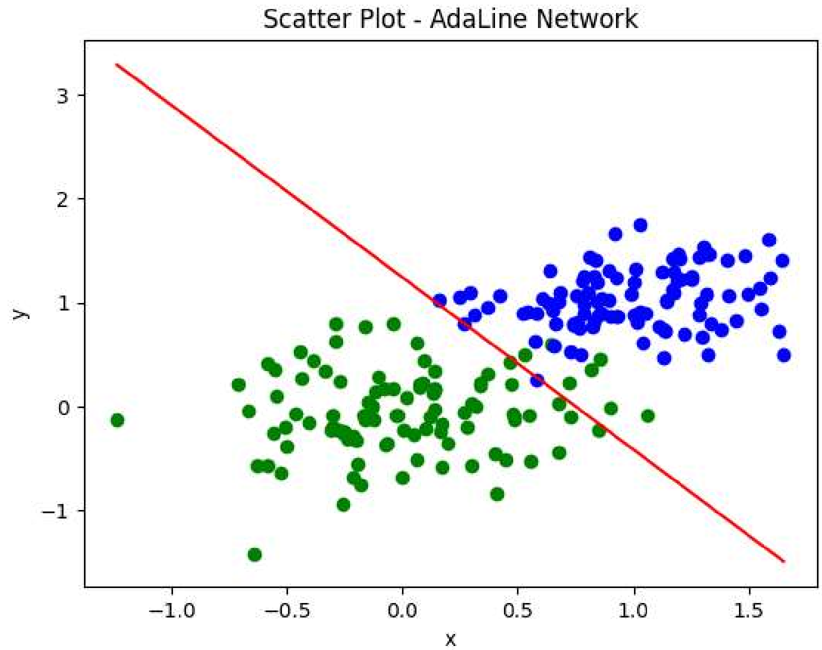


Figure 8. AdaLine Network and another data

2.2. MAdaLine Neural Network

In this question, we aim to classify the given data (MadaLine.csv), which consists of two classes. The scatter plot of this data is shown in **Figure 9**.

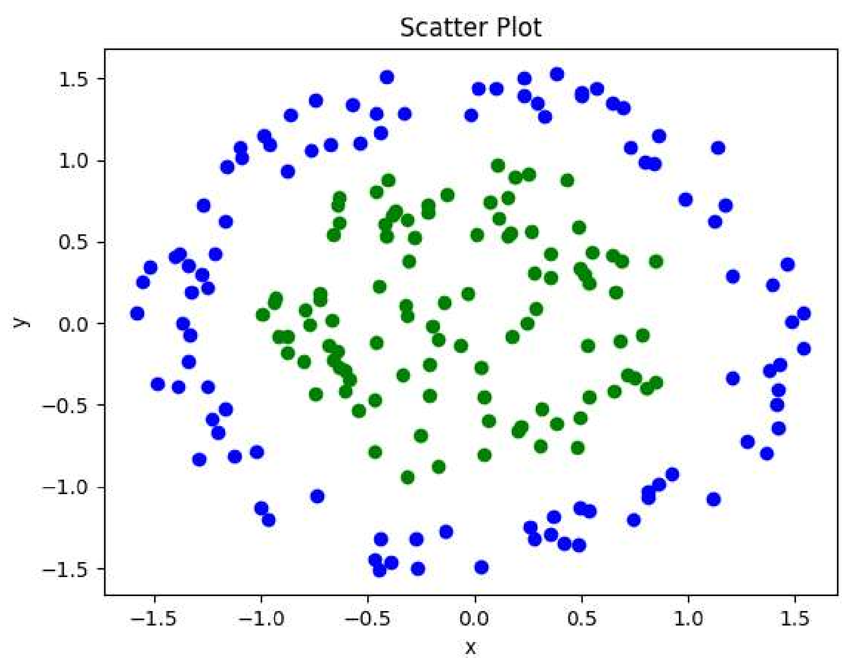


Figure 9. MadaLine.csv data scatter plot

The MAdaLine network design has two algorithms (MR-I and MR-II). We will use the MR-I algorithm for designing the network related to our problem. This algorithm is shown in **Figure 10**.

Training Algorithm for* MADALINE *(MRI). The activation function for units Z_1, Z_2, and Y is

$$f(x) = \begin{cases} 1 & \text{if } x \geq 0; \\ -1 & \text{if } x < 0. \end{cases}$$

Step 0. Initialize weights:
Weights v_1 and v_2 and the bias b_3 are set as described; small random values are usually used for ADALINE weights.
Set the learning rate α as in the ADALINE training algorithm (a small value).

Step 1. While stopping condition is false, do Steps 2–8.

Step 2. For each bipolar training pair, **s**:t, do Steps 3–7.

Step 3. Set activations of input units:

$$x_i = s_i.$$

Step 4. Compute net input to each hidden ADALINE unit:

$$z\_in_1 = b_1 + x_1 w_{11} + x_2 w_{21},$$

$$z\_in_2 = b_2 + x_1 w_{12} + x_2 w_{22}.$$

Step 5. Determine output of each hidden ADALINE unit:

$$z_1 = f(z\_in_1),$$

$$z_2 = f(z\_in_2).$$

Step 6. Determine output of net:

$$y\_in = b_3 + z_1 v_1 + z_2 v_2;$$

$$y = f(y\_in).$$

Step 7. Determine error and update weights:
If $t = y$, no weight updates are performed.
Otherwise:
If $t = 1$, then update weights on Z_J, the unit whose net input is closest to 0,

$$b_J(\text{new}) = b_J(\text{old}) + \alpha(1 - z\_in_J),$$

$$w_{iJ}(\text{new}) = w_{iJ}(\text{old}) + \alpha(1 - z\_in_J)x_i;$$

If $t = -1$, then update weights on all units Z_k that have positive net input,

$$b_k(\text{new}) = b_k(\text{old}) + \alpha(-1 - z\_in_k),$$

$$w_{ik}(\text{new}) = w_{ik}(\text{old}) + \alpha(-1 - z\_in_k)x_i.$$

Figure 10. MadaLine MR-I Algorithm

To design the network, we consider three cases where the number of neurons in the hidden layer (the number of AdaLine algorithm lines) is **3**, **5**, or **10**.

We will use **90%** of the data for **training** and keep the remaining **10%** for **testing** the designed network at the end.

Figure 11 shows the accuracy and the decreasing cost function graph per iteration for each of the three mentioned cases of the number of neurons in the hidden layer.

As observed, **increasing the number of neurons** in the hidden layer (or, in other words, increasing the number of AdaLines and enlarging the polygon separating the two classes) results in **higher network accuracy**. The **cost function** also exhibits a **more pronounced downward trend, requiring fewer iterations to reach a stable state**.

```
Accuracy (m = 3) = 80.0%
Accuracy (m = 5) = 95.0%
Accuracy (m = 10) = 100.0%
```

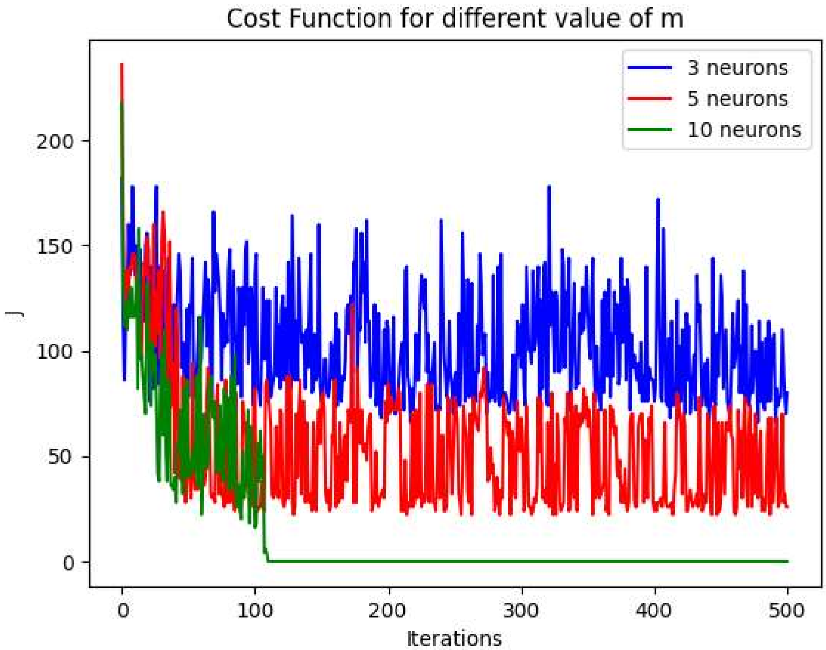


Figure 11. MadaLine MR-I Algorithm

Question 3. Auto-Encoders for Classification

In this question, I want to solve a classification problem using an **Auto-Encoder**. For a better understanding of Auto-Encoders, it is recommended to read the attached paper (liu2017.pdf). The goal of this exercise is to familiarize yourself with the '**keras' package** and work with the **MNIST dataset**.

3.1. Introduction to MNIST Dataset

In this section, the goal is to get familiar with and work with the dataset. You can add the dataset using the '**torchvision' package** as shown below. Alternatively, you can use the attached file (mnist.npz).

Figure 12 shows 10 random images from the dataset along with their labels.

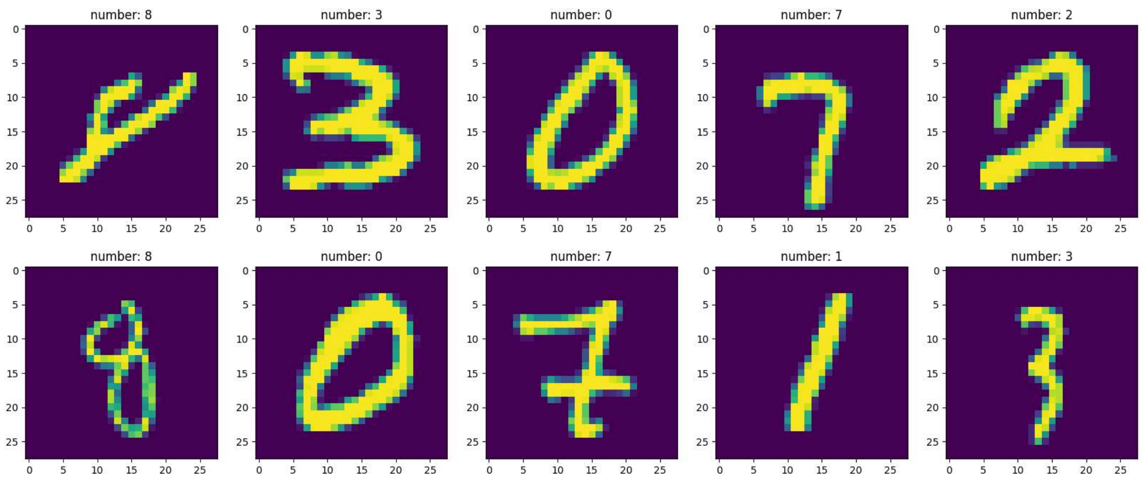


Figure 12. Some Random Images from MNIST Dataset

Figure 13 shows the number of data points for each label in the training dataset.

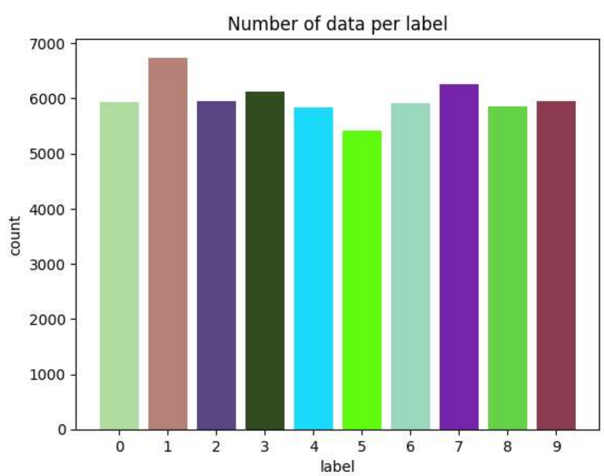


Figure 13. Number of Data for each Label (Training set)

3.2. Auto-Encoder Network

In this section, we want to design the desired network. For this purpose, we will design two separate parts for the network:

1. the **encoder**
2. the **decoder**.

Figure 14 shows the architecture of both parts.

Layer (type)	Output Shape	Param #
input_layer (InputLayer)	(None, 784)	0
encoder (Functional)	(None, 30)	575,930
decoder (Functional)	(None, 784)	576,684

Layer (type)	Output Shape	Param #
input_layer (InputLayer)	(None, 784)	0
flatten (Flatten)	(None, 784)	0
dense (Dense)	(None, 500)	392,500
dense_1 (Dense)	(None, 300)	150,300
dense_2 (Dense)	(None, 100)	30,100
dense_3 (Dense)	(None, 30)	3,030

Layer (type)	Output Shape	Param #
input_layer_1 (InputLayer)	(None, 30)	0
dense_4 (Dense)	(None, 100)	3,100
dense_5 (Dense)	(None, 300)	30,300
dense_6 (Dense)	(None, 500)	150,500
dense_7 (Dense)	(None, 784)	392,784

Figure 14. Architecture of Encoder and Decoder Networks

Now, we will train the network. For this purpose, we will use the 'adam' **optimizer** and the 'binary_crossentropy' **loss function**.

Figure 15 shows the loss function plot against the number of epochs.

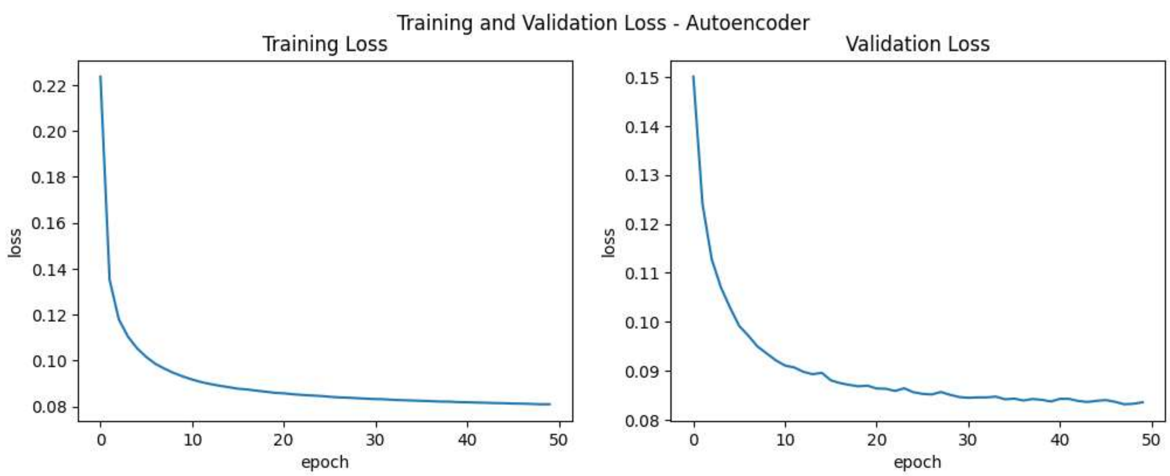


Figure 15. Auto-Encoder Loss Function During Training Process

Figure 16 compares the original images with the decoded ones (the network's output).

3.3. Classification

In this section, I want to use a 30-dimensional feature space (the output of the encoder) to build a simple classifier with two hidden layers. To do this, I will separate the **encoder part** after training the autoencoder and use its outputs to train the **classifier**.

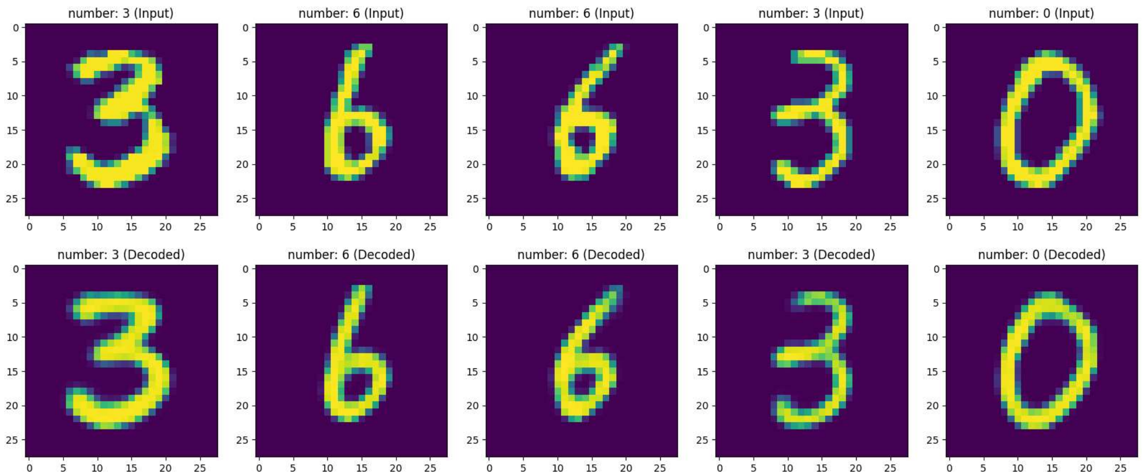


Figure 16. Comparison of the Original and the Decoded Images (the Network's Output)

Figure 17 shows the architecture of the classifier network.

Layer (type)	Output Shape	Param #
input_layer_3 (InputLayer)	(None, 30)	0
dense_11 (Dense)	(None, 60)	1,860
dense_12 (Dense)	(None, 30)	1,830
dense_13 (Dense)	(None, 10)	310

Figure 17. Architecture of the Classifier Network

Now, I will train the classifier network. **Figure 18** shows the loss plot, and **Figure 19** shows the accuracy plot against the number of epochs for both the training and validation datasets.

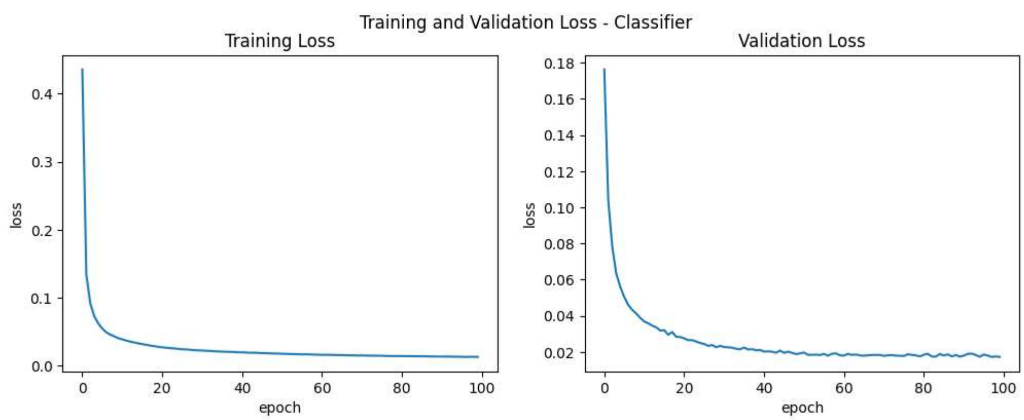


Figure 18. Classifier Loss Function During Training Process

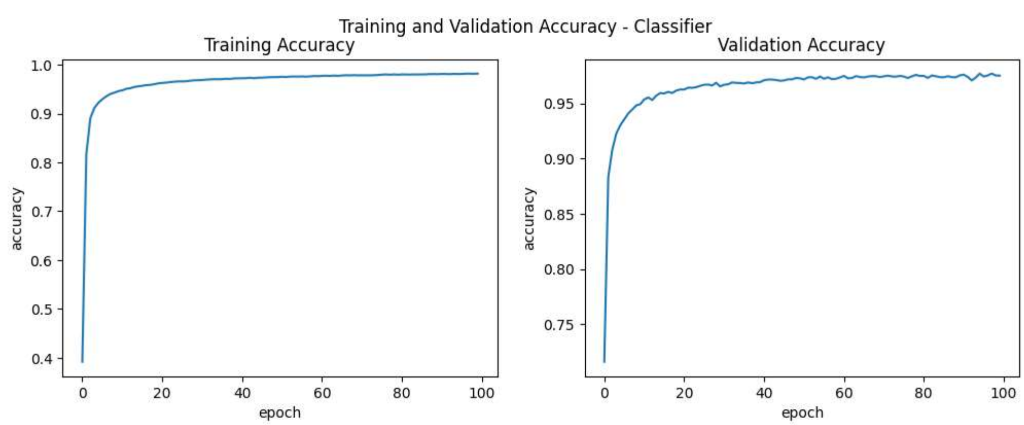


Figure 19. Classifier Accuracy During Training Process

Finally, we connect the encoder network to the classifier network.

Figure 20 shows the architecture of the final network along with the accuracy on the training and test datasets.

```
Train Accuracy: 98.16%
Test Accuracy: 97.5%
```

Layer (type)	Output Shape	Param #
input_layer (InputLayer)	(None, 784)	0
encoder (Functional)	(None, 30)	575,930
classifier (Functional)	(None, 10)	4,000

Figure 20. Architecture of the Final Network and the Accuracy on the Training and Test Datasets

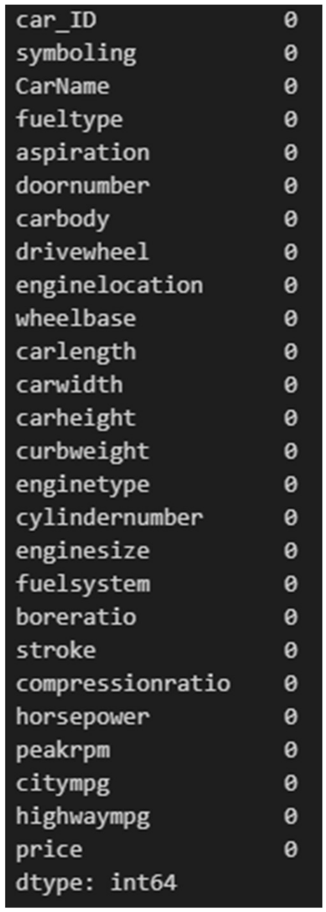
Question 4. Multi-Layer Perceptron Neural Networks

In this question, we have a dataset for predicting prices (CarPrice_Assignment.csv). First, we will work with the data and get familiar with feature engineering. Then, we will use a **Multi-Layer Perceptron (MLP)** network to predict prices and compare the predictions with the actual prices.

The aim of this question is to become familiar with **Multi-Layer Perceptron (MLP)** and the 'TensorFlow' and 'keras' libraries.

First, we will read the data using 'pandas' package.

Figure 21 shows the total number of NaN values in each feature.



```
car_ID              0
symboling           0
CarName             0
fueltype            0
aspiration          0
doornumber          0
carbody             0
drivewheel          0
enginelocation      0
wheelbase           0
carlength           0
carwidth            0
carheight           0
curbweight          0
enginetype          0
cylindernumber      0
enginesize          0
fuelsystem          0
boreratio           0
stroke              0
compressionratio    0
horsepower          0
peakrpm             0
citympg             0
highwaympg          0
price               0
dtype: int64
```

Figure 21. Total Number of NaN Values in each Feature

4.1. Introduction to the Dataset and Preprocessing

In this section, we will preprocess the data to prepare it for network training. The preprocessing steps are as follows:

1. **Extract Company Name:** Separate the company name from the CarName column and store it in a new column named CompanyName. (I will also correct any incorrectly entered company names.)
2. **Drop Columns:** Remove the columns **CarName**, **car_ID**, and **symbolling**.
3. **Convert Categorical Data:** Convert descriptive data to numeric data using pd.get_dummies().

Next, we use the correlation matrix to identify the feature that has the highest correlation with the price (target). **Figure 22** shows the distribution of price and enginesize (which has the highest correlation with the price).

Finally, we will split the data into two parts: 85% for training and 15% for testing. This will prepare the data for training the network.

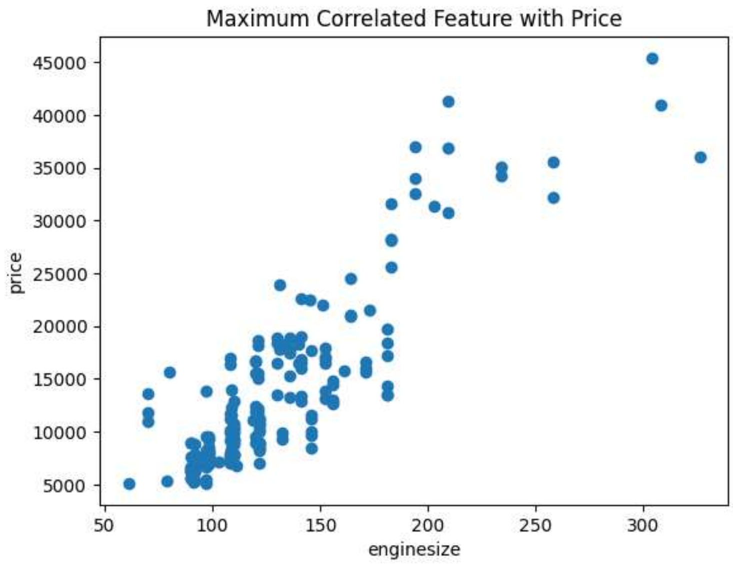


Figure 22. Distribution of 'price' and 'enginesize'

4.2. Multi-Layer Perceptron

In this section, we will first design a network with 3 hidden layers for predicting car prices. **Figure 23** shows the architecture of the network.

Layer (type)	Output Shape	Param #
input_layer (InputLayer)	(None, 75, 1)	0
flatten (Flatten)	(None, 75)	0
dense (Dense)	(None, 200)	15,200
dense_1 (Dense)	(None, 150)	30,150
dense_2 (Dense)	(None, 100)	15,100
dense_3 (Dense)	(None, 1)	101

Figure 23. MadaLine MR-I Algorithm

I will train the network using the 'MeanAbsolutePercentageError' loss function and the 'adamw' optimizer.

Figure 24 shows the **loss function** and **R2-score** metrics for the training and test datasets over the course of training (as a function of epochs).

Figure 25 shows the scatter plot of predicted prices versus actual prices. As observed, the car prices are predicted with good accuracy, achieving an error of approximately 10%.

The comparison of predicted prices with actual prices for 5 random cars from the test dataset is shown in **Figure 26**. As observed, the predicted prices are quite close to the actual prices.

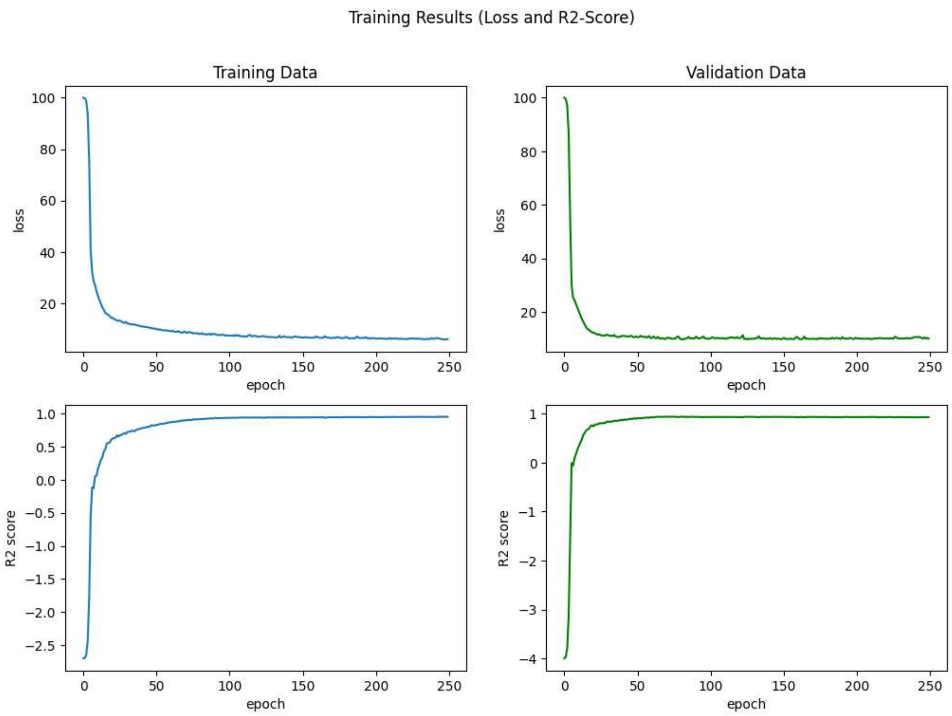


Figure 24. Loss Function And R2-Score Metrics during Training Process

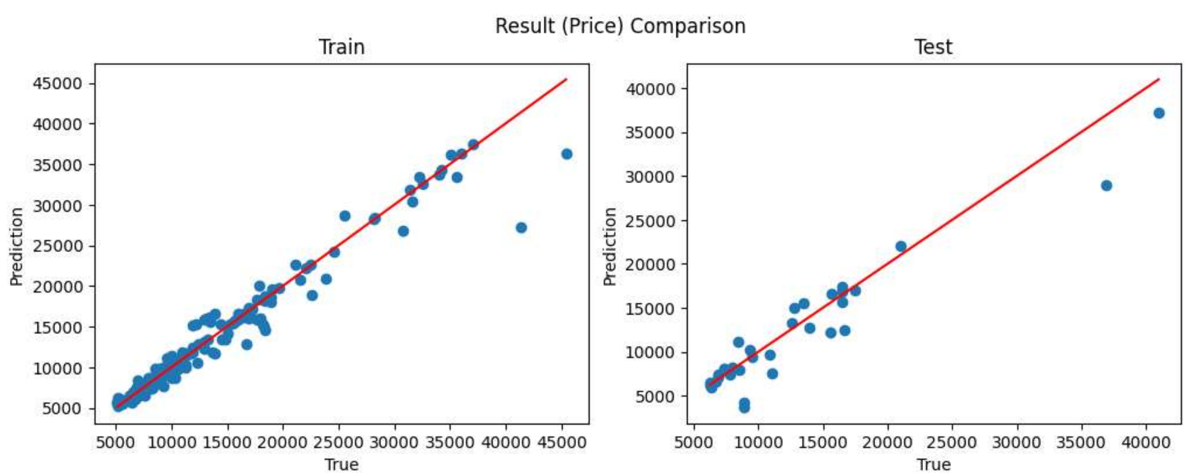


Figure 25. MadaLine MR-I Algorithm

```
Real Price for Car Number 10: [16430.]
Predicted Price for Car Number 10: [16880.1]


Real Price for Car Number 9: [16630.]
Predicted Price for Car Number 9: [12518.669]


Real Price for Car Number 24: [6938.]
Predicted Price for Car Number 24: [7441.733]


Real Price for Car Number 23: [15690.]
Predicted Price for Car Number 23: [16636.943]


Real Price for Car Number 20: [16515.]
Predicted Price for Car Number 20: [15613.918]
```

Figure 26. MadaLine MR-I Algorithm