intelligence computationnelle

Speciality : *Computer Systems Engineering*

Neural Network

Carried out by

- Rouabah Oualid

2023/2024

• Table of Contents

• Introduction

In the realm of artificial intelligence, mastering the intricacies of neural networks involves a multifaceted approach. This encompasses the creation of a multilayer perceptron neural network, a sophisticated architecture capable of handling complex tasks by leveraging interconnected layers of nodes. Central to this understanding is the exploration of various activation functions, pivotal elements that govern the activation of neurons within the network. Implementing both forward and backward propagation mechanisms becomes paramount, as these processes orchestrate the flow of information during prediction and learning phases. Forward propagation involves passing input through the network to produce an output, while backward propagation, also known as backpropagation, is the engine behind learning. Through meticulous adjustment of internal parameters during backpropagation, the neural network refines its understanding of patterns and relationships within the data, ultimately embodying the essence of how neural networks learn and train. Mastering these components is pivotal for unlocking the true potential of neural networks in solving intricate problems across diverse domains.

In this report we will do a simple neural network to understand how it works and basic principles.

• Expected goals

The development of a multilayer perceptron neural network capable of distinguishing between two items *Vis* and *Clou*.

• Implementation of homework

• Explanation of architecture used

Our model's architecture comprises 3 layers. We employ the sigmoid function due to its smooth curve, ensuring it consistently yields positive values.

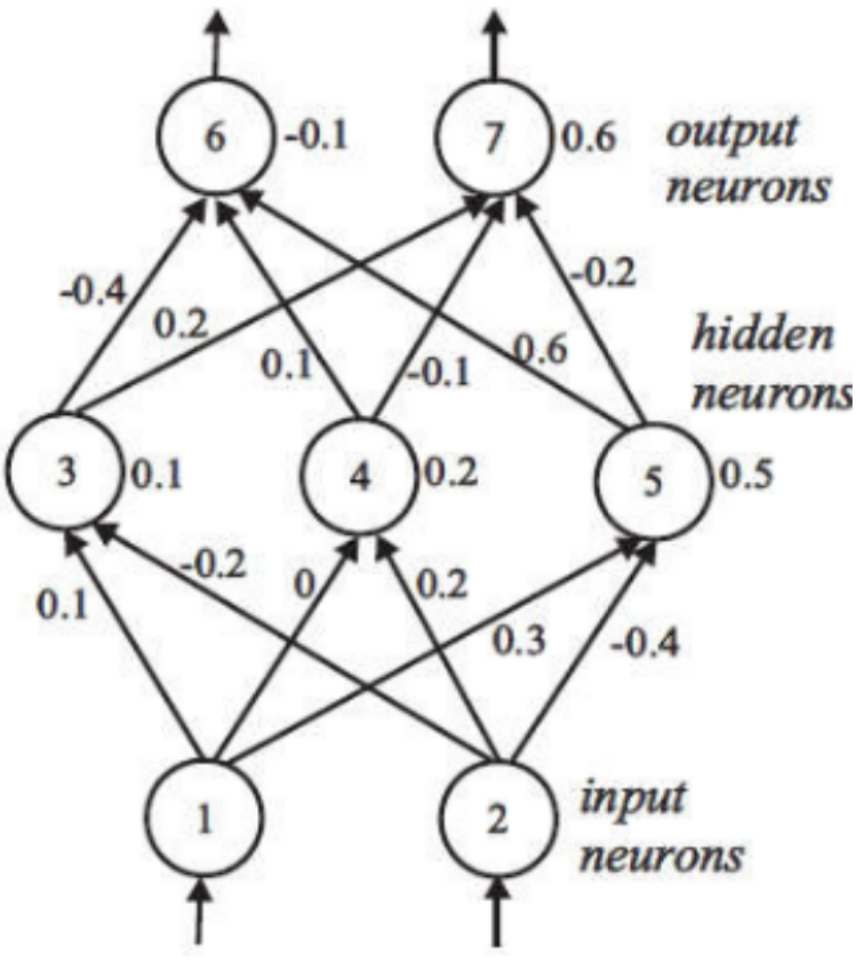


FIGURE 1 – initial neural network

- **Used functions**

Here the used functions in this TP and the implementation with python :

1. **Sigmoid** : We use the Sigmoid function as activation function in our work. Is represented as :

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

This function is commonly used in neural networks. **Implementation** :

```
import numpy as np
def activation_function(x):
    return 1 / (1 + np.exp(-x)) #OR math.exp(-x)
```

OR :

```
import scipy.special
def activation_function(x):
    return scipy.special.expit(x)
```

2. **Forward propagation** : in forward propagation we use this function to calculate the output in each node

$$\text{Out} = \sigma(\sum(\mathbf{W}_{\text{input}} \cdot \mathbf{Input}))$$

Implementation :

```
samples = selected_samples[0]
label = selected_samples[1]
inputs = numpy.array(samples, ndmin=2).T
targets = numpy.array(label, ndmin=2).T
hidden_inputs = numpy.dot(self.weight_input_hidden, inputs)
hidden_outputs =self.activation_function_sigmoid(hidden_inputs)
final_inputs = numpy.dot(self.weight_hidden_output, hidden_outputs)
final_outputs = self.activation_function_sigmoid(final_inputs)
```

3. **Backward propagation(Gradient descent)** : is specifically used for the backpropagation algorithm, which updates the weights of the network to minimize error :

$$\frac{\partial \text{Error}}{\partial \text{Weight}} = \frac{\partial \text{Error}}{\partial \text{Out}} \times \frac{\partial \text{Out}}{\partial \text{In}} \times \frac{\partial \text{In}}{\partial \text{Weight}}$$

$$\boldsymbol{\Delta}_{\text{output}} = \mathbf{y}_{\text{longed}} - \mathbf{y}_{\text{got}}$$

$$\boldsymbol{\Delta}_{\text{i}} = \mathbf{a}_{\text{i}} \times (1 - \mathbf{a}_{\text{i}}) \times \sum \mathbf{W}_{\text{i,j}} \cdot \boldsymbol{\Delta}_{\text{j}}$$

Implementation :

```
output_errors = targets - final_outputs
hidden_errors = numpy.dot(self.weight_hidden_output.T, output_errors)*
 → hidden_outputs * (1.0 - hidden_outputs))
```

4. **Update the weights** :

$$\mathbf{W}_{\text{i,j}}(\text{t+1}) = \mathbf{W}_{\text{i,j}}(\text{t}) + \alpha \times (\mathbf{a}_{\text{i}} \times \boldsymbol{\Delta}_{\text{j}})$$

Implementation :

```
self.weight_hidden_output += self.lr * numpy.dot(
                (output_errors),
                numpy.transpose(hidden_outputs),
            )


self.weight_input_hidden += self.lr * numpy.dot(
                (hidden_errors)),
                numpy.transpose(inputs),
            )
```

• Script Overview

The script defines a neural network class, initializes it with parameters, trains it on a dataset, and evaluates its performance before and after training. It also visualizes the error rate during training and the distribution of data after training and prediction. The overall goal is to showcase the implementation of a simple neural network for a binary classification task.

• Dataset used

In our code we use a 3D NumPy array for representing a collection of samples for training a neural network. Let's break down the structure of the dataset :

```
dataset = numpy.array(
    [
        [[0.6, 0.1], [1, 0]],
        [[0.2, 0.3], [0, 1]],
        [[0.4, 0.4], [0, 1]],
        [[0.4, 0.2], [0, 1]],
        [[0.5, 0.3], [1, 0]],
        [[0.1, 0.2], [0, 1]],
        [[0.8, 0.7], [1, 0]],
    ],
    dtype=float,
)
```

1. Each entry in the dataset is a sample, represented as a list of two lists.
   - The first inner list [0.6, 0.1] represents the features of the sample, possibly corresponding to cloud and visibility values.
   - The second inner list [1, 0] represents the label or target for the binary classification task. In this case, it's a one-hot encoded label, where [1, 0] indicate "Vis", and [0, 1] indicates the "Clou".
2. The entire dataset is a 3D NumPy array, where each element is a 2D array representing a sample.
3. The dtype=float specifies that the elements in the dataset are of type float.

- **Neural Network Class Definition**

```
class neuralNetwork:

    # initialise the neural network
 →  def __init__(self, layers_number, initial_weight, alpha, epoch,

 →  batch_testing=1):

        self.input_node_number = layers_number[0]

        self.hidden_node_number = layers_number[1]

        self.output_node_number = layers_number[2]

        self.iteration = epoch

        self.lr = alpha

        self.batch_testing = batch_testing
 →      self.weight_input_hidden = numpy.array(initial_weight[0],

 →  dtype=float)
 →      self.weight_hidden_output = numpy.array(initial_weight[1],

 →  dtype=float)
 →      self.activation_function_sigmoid = lambda x:

 →  activation_function(x)

        self.error_by_training = []
```

Then initial it like this :

```
neurel_network = neuralNetwork(

    layers,

    initial_weight,

    learning_rate,

    iteration,

    batch_testing=50

)
```

with :

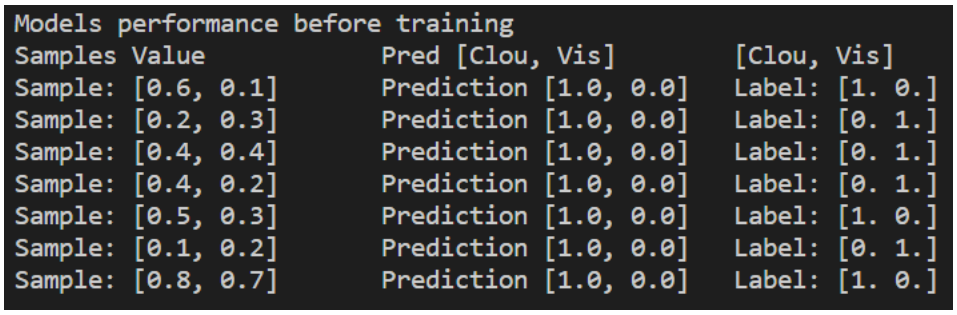
```
learning_rate = 0.01
iteration = 20000
# number of input, hidden and output nodes
input_nodes = 2
hidden_nodes = 3
output_nodes = 2
layers = [input_nodes, hidden_nodes, output_nodes]
# initial weight
initial_weight = [
    [[0.1, -0.2], [0, 0.2], [0.3, -0.4]],
    [[-0.4, 0.1, 0.6], [0.2, -0.1, -0.2]],
]
```

- **Testing dataset**

The **testing** function evaluates the neural network's performance on a dataset by comparing its predictions to the actual labels. It calculates the total error for all samples in the dataset.

```
def testing(self, dataset):
    result = []
    for sample, label in dataset:
        prediction = self.query(sample)
        error = label - [prediction[0][0], prediction[1][0]]
        result.append(sum(numpy.absolute(error)))
    return sum(result)
```

For our dataset before training we get :



```
Models performance before training
Samples Value               Pred [Clou, Vis]        [Clou, Vis]
Sample: [0.6, 0.1]          Prediction [1.0, 0.0]   Label: [1. 0.]
Sample: [0.2, 0.3]          Prediction [1.0, 0.0]   Label: [0. 1.]
Sample: [0.4, 0.4]          Prediction [1.0, 0.0]   Label: [0. 1.]
Sample: [0.4, 0.2]          Prediction [1.0, 0.0]   Label: [0. 1.]
Sample: [0.5, 0.3]          Prediction [1.0, 0.0]   Label: [1. 0.]
Sample: [0.1, 0.2]          Prediction [1.0, 0.0]   Label: [0. 1.]
Sample: [0.8, 0.7]          Prediction [1.0, 0.0]   Label: [1. 0.]
```

FIGURE 2 – Result before training

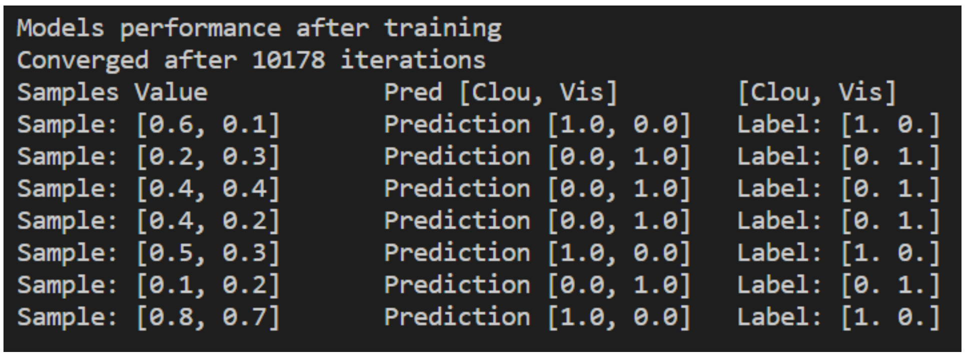
As we see, there is many error in our Neural network se we should train it.

- **Training the Neural Network**

The **train** function is responsible for training the neural network using the provided dataset. It performs forward and backward passes, updating weights based on the error in predictions. The training loop runs for the specified number of epochs (self.iteration). In each epoch, a random sample is selected from the dataset, and the network's weights are updated accordingly. The self.batch_testing parameter controls how often the error is recorded for analysis. The function also checks for convergence, and if the accuracy reaches 100%, it prints a message and breaks out of the loop.

```
def train(self, dataset):
    # Reset error list
    self.error_by_training = []
    # Training loop
    count = 0
    for epoch in range(self.iteration):
        # Randomly select a sample from the dataset
        dataset_length = len(dataset)
        selected_samples = dataset[randint(0, dataset_length - 1)]
        samples = selected_samples[0]
        label = selected_samples[1]
        # Convert samples and labels to numpy arrays
        inputs = numpy.array(samples, ndmin=2).T
        targets = numpy.array(label, ndmin=2).T
        # Forward pass
        hidden_inputs = numpy.dot(self.weight_input_hidden, inputs)
        hidden_outputs = self.activation_function_sigmoid(hidden_inputs)
→       final_inputs = numpy.dot(self.weight_hidden_output,
→   hidden_outputs)
        final_outputs = self.activation_function_step(final_inputs)
        # Backward pass

        output_errors = targets - final_outputs
→       hidden_errors = numpy.dot(self.weight_hidden_output.T,

→   output_errors)

        # Update weights

        self.weight_hidden_output += self.lr * numpy.dot(

            (output_errors),

            numpy.transpose(hidden_outputs),

        )

        self.weight_input_hidden += self.lr * numpy.dot(

            (hidden_errors * hidden_outputs * (1.0 - hidden_outputs)),

            numpy.transpose(inputs),

        )

        # Update counter and error list

        count += 1

        if count > self.batch_testing:

            count = count % self.batch_testing

            self.error_by_training.append(self.testing(dataset))

        # Check for convergence

        accuracy = self.calculate_accuracy(dataset)

        if accuracy == 100.0:

            print(f"Converged after {epoch + 1} iterations")

            break
```

after training our model for the dataset and testing it we have improved results :



```
Models performance after training
Converged after 10178 iterations
Samples Value            Pred [Clou, Vis]        [Clou, Vis]
Sample: [0.6, 0.1]       Prediction [1.0, 0.0]   Label: [1. 0.]
Sample: [0.2, 0.3]       Prediction [0.0, 1.0]   Label: [0. 1.]
Sample: [0.4, 0.4]       Prediction [0.0, 1.0]   Label: [0. 1.]
Sample: [0.4, 0.2]       Prediction [0.0, 1.0]   Label: [0. 1.]
Sample: [0.5, 0.3]       Prediction [1.0, 0.0]   Label: [1. 0.]
Sample: [0.1, 0.2]       Prediction [0.0, 1.0]   Label: [0. 1.]
Sample: [0.8, 0.7]       Prediction [1.0, 0.0]   Label: [1. 0.]
```

FIGURE 3 – Result after training

We got this after 10178 iterations as mentioned in Figure 3.

- **Querying the Neural Network**

The **query** function takes an input and performs a forward pass through the neural network to make a prediction. It converts the input to a *numpy array*, calculates signals through the hidden layer, and finally produces the output.

```python
def query(self, inputs_list):
    # convert inputs list to 2d array
    inputs = numpy.array(inputs_list, ndmin=2).T
    # calculate signals into hidden layer
    hidden_inputs = numpy.dot(self.weight_input_hidden, inputs)
    # calculate the signals emerging from hidden layer
    hidden_outputs = self.activation_function_sigmoid(hidden_inputs)
    # calculate signals into final output layer
    final_inputs = numpy.dot(self.weight_hidden_output, hidden_outputs)
    # calculate the signals emerging from final output layer
    final_outputs = self.activation_function_sigmoid(final_inputs)
    final_outputs = np.round(final_outputs, 0)
    return final_outputs
```

- **Calculating Accuracy**

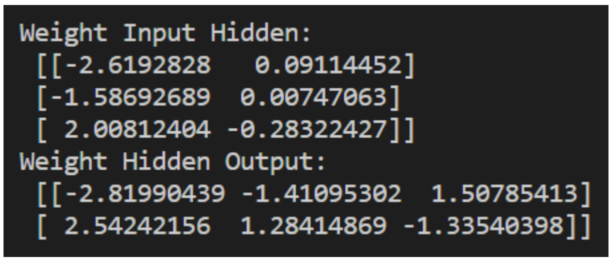
The **calculate_accuracy** function computes the accuracy of the neural network on a given dataset, it evaluates the accuracy of the neural network by comparing its predictions to the true labels in the given dataset. The final accuracy is expressed as a percentage.

```python
def calculate_accuracy(self, dataset):
    # Calculate the accuracy of the network on the given dataset
    correct_predictions = 0
    total_samples = len(dataset)
    for sample, label in dataset:
        prediction = self.query(sample)
        rounded_prediction = np.round(prediction).astype(int).flatten()
        if np.array_equal(rounded_prediction, label.astype(int)):
            correct_predictions += 1
    accuracy = (correct_predictions / total_samples) * 100
    return accuracy
```

For our model after training it we have the accuracy 100%.

- **Update weights**

After training and testing our dataset with the accuracy in 100% we get new weights like this :



```
Weight Input Hidden:
 [[-2.6192828   0.09114452]
 [-1.58692689  0.00747063]
 [ 2.00812404 -0.28322427]]
Weight Hidden Output:
 [[-2.81990439 -1.41095302  1.50785413]
 [ 2.54242156  1.28414869 -1.33540398]]
```

FIGURE 4 – New weights

Our neural network will be :

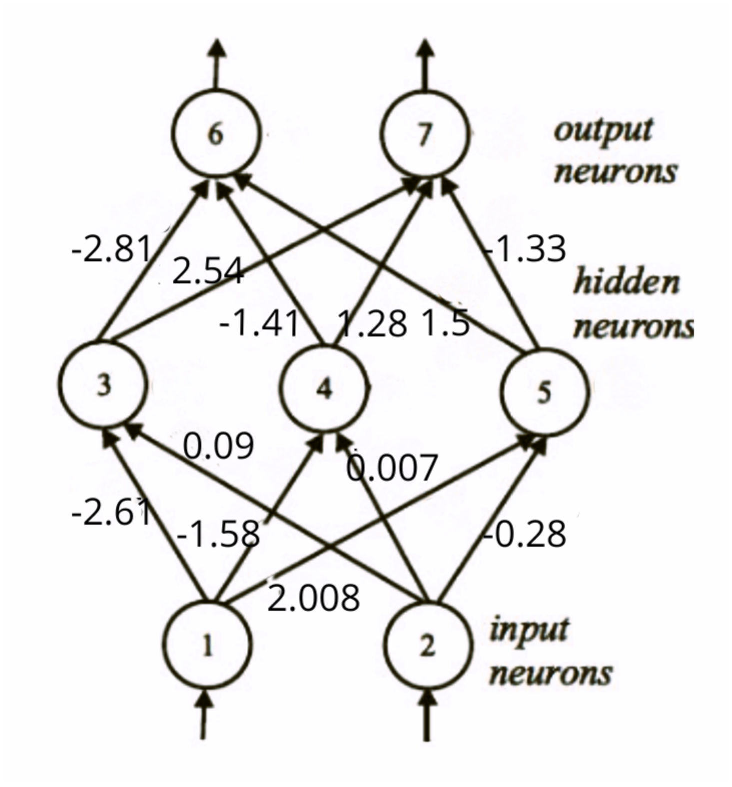


FIGURE 5 – New neural network

- **Prediction data**

With using those new weights after training our neural network with previous dataset, we gonna use them to predict new data and do classification to it, if it's a "Clou" or "Vis".

N 1	N 2	Class
0.3	0.9	?
0.7	0.8	?
0.6	0.8	?
0.2	0.1	?

TABLE 1 – Data to predict

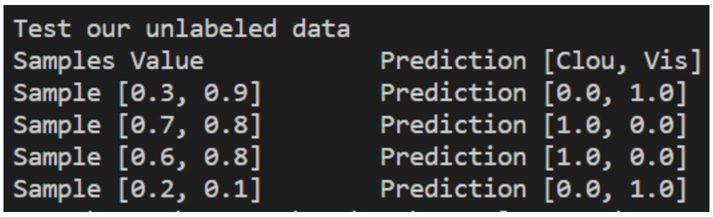
The representation for this data is :

```
prediction_dataset = numpy.array(
    [
        [0.3, 0.9],
        [0.7, 0.8],
        [0.6, 0.8],
        [0.2, 0.1],
    ],
)
```

Then we will query the data :

```
print("Samples Value\t", "\t Prediction [Clou, Vis]")
for sample in prediction_dataset:
    result = list(neurel_network.query(sample))
    print(
        "Sample",
        [sample[0], sample[1]],
        "\t Prediction",
        [result[0][0], result[1][0]]
    )
```

Result :



```
Test our unlabeled data
Samples Value             Prediction [Clou, Vis]
Sample [0.3, 0.9]         Prediction [0.0, 1.0]
Sample [0.7, 0.8]         Prediction [1.0, 0.0]
Sample [0.6, 0.8]         Prediction [1.0, 0.0]
Sample [0.2, 0.1]         Prediction [0.0, 1.0]
```

FIGURE 6 – Result of testing unlabeled data

The prediction is :

N 1	N 2	Class
0.3	0.9	Vis
0.7	0.8	Clou
0.6	0.8	Clou
0.2	0.1	Vis

TABLE 2 – Prediction

- **Plotting**

•Plotting Error Rate During Training

To plot the number of errors of this is model we made it with this function :

```
length_error = len(neural_network.error_by_training)
iterations = np.array(range(length_error)) * neural_network.batch_testing
fig, ax = plt.subplots()
ax.plot(iterations, neural_network.error_by_training)
ax.grid(True, linestyle='-.')
ax.tick_params(labelcolor='r', labelsize='medium', width=3)
plt.show()
```

Here's a breakdown of what's happening :

1. `length_error = len(neurel_network.error_by_training` : Obtains the length of the error list from the neural network object, representing training errors.
2. `iterations = np.array(range(length_error)) * neurel_network.batch_testing` : Creates an array of iteration numbers based on the length of training errors and the batch size for testing.
3. `fig, ax = plt.subplots()` : Sets up a plot with a single subplot.
4. `ax.plot(iterations, neurel_network.error_by_training)` : Plots the training error of the neural network over iterations.
5. `ax.grid(True, linestyle="-.")` : Adds a grid to the plot with a dash-dot style.
6. `ax.tick_params(labelcolor="r", labelsize="medium", width=3)` : Customizes tick parameters, setting label color to red, size to medium, and width to 3.
7. `plt.show()` : Displays the plot.

Results :

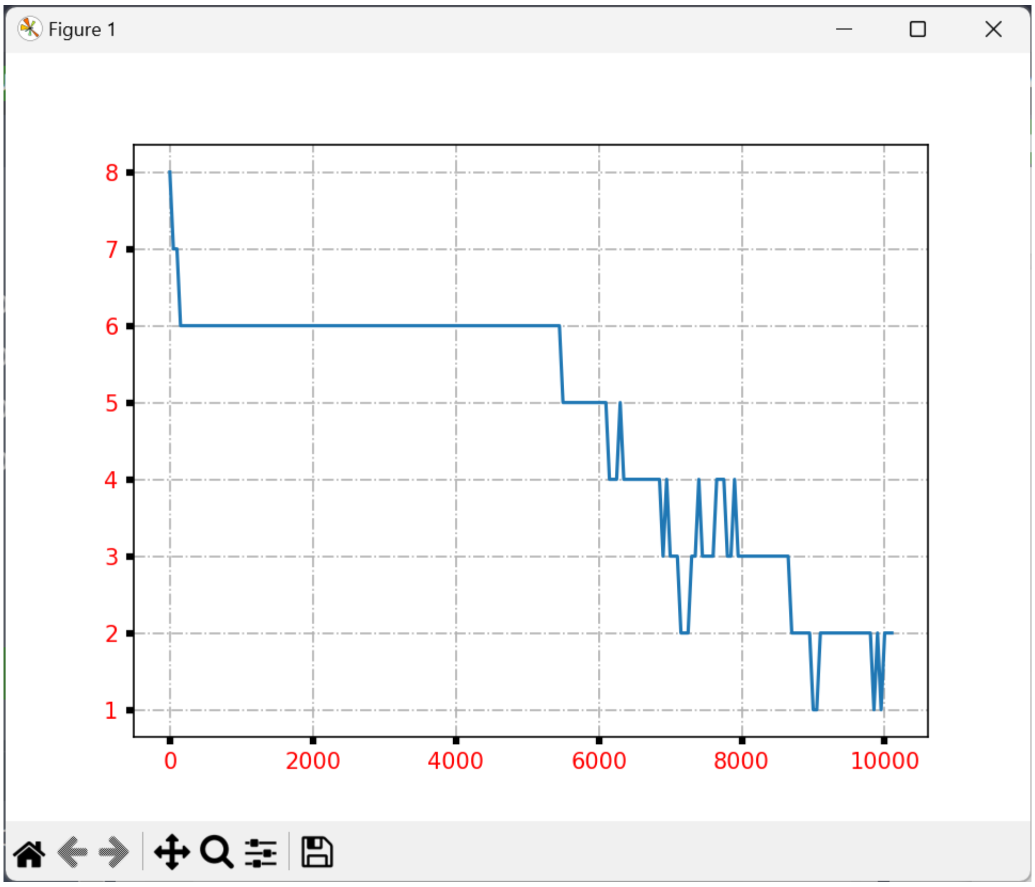


FIGURE 7 – Plotting Error rate

•Plotting Data After Training and Prediction

To plot the dataset and the data that we predict it we gonna use this function :

```python
my_dataset = numpy.array([...])
get_x = lambda x: [i[0] for i in x]
get_y = lambda x: [i[1] for i in x]
area1, area2, area3 = [], [], []
for sample, label in my_dataset:
    if label[0] == 1:
        area1.append(sample)
    elif label[0] == 0:
        area2.append(sample)
    else:
        area3.append(sample)
plt.scatter(get_x(area1), get_y(area1), label="Vis", marker="+")
plt.scatter(get_x(area2), get_y(area2), label="Clou", marker="o")
plt.scatter(get_x(area3), get_y(area3), label="none", marker="*")
plt.legend(["Vis", "Clou", "none"], loc="upper left")
plt.show()
```

Here's a breakdown of what's happening :

1. `my_dataset = numpy.array(...)` : is a NumPy array containing samples and their corresponding labels.
2. `get_x = lambda x: [i[0] for i in x]` and `get_y = lambda x: [i[1] for i in x]` : Define lambda functions to extract x and y coordinates from samples.
3. `for sample, label in my_dataset: ...` : Categorizes samples into areas based on labels using a for loop.
4. `plt.scatter(get_x(area1), get_y(area1), label="Vis", marker="+"), etc.` : Creates scatter plots for each area with different markers.
5. `plt.legend(["Vis", "Clou", "none"], loc="upper left")` : Adds a legend to the plot with area labels.
6. `plt.show()` : Displays the plot.

Result :

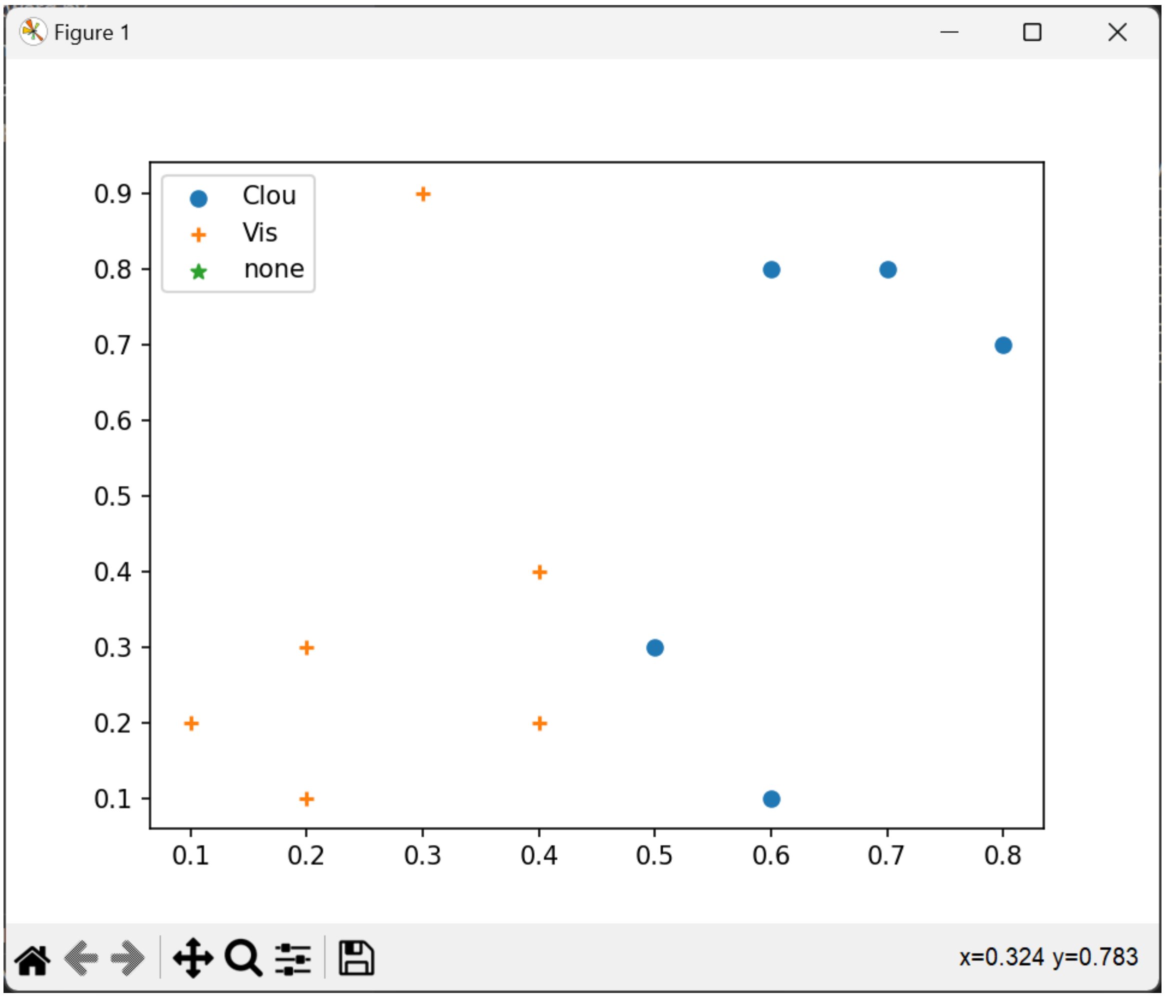


FIGURE 8 – Plotting data

- ## Further

Modifying a neural network can involve various changes to its architecture, hyperparameters, or training process. Here are some common modifications you can make to a neural network :

1. **Change the Number of Nodes or Layers :**
   - **Increase Nodes :** Adding more nodes to a layer or more layers can increase the model's capacity to learn complex patterns.
   - **Decrease Nodes :** Reducing nodes can simplify the model, potentially preventing overfitting and reducing computational complexity.
2. **Adjust Learning Rate :**
   - **Increase Learning Rate :** A higher learning rate can lead to faster convergence but may risk overshooting the optimal weights.
   - **Decrease Learning Rate :** A lower learning rate can enhance stability during trai-

ning and fine-tuning.

3. **Apply Regularization Techniques :**
   - **L1 or L2 Regularization :** Penalize large weights to prevent overfitting.
   - **Dropout :** Randomly deactivate nodes during training to enhance generalization.
4. **Modify Activation Functions :**
   - **ReLU (Rectified Linear Unit) :** Commonly used for hidden layers.
   - **Sigmoid or Softmax :** Used for the output layer, especially in binary or multiclass classification.
   - **Tanh :** Used for hidden layers in certain cases.
5. **Change Batch Size :**
   - **Increase Batch Size :** Can lead to faster training but requires more memory.
   - **Decrease Batch Size :** Can improve convergence, especially with limited data.
6. **Experiment with Optimizers :**
   - **Adam, SGD, RMSprop :** Different optimization algorithms may affect convergence speed and final performance.
7. **Modify Initialization Schemes :**
   - **Xavier/Glorot Initialization :** Suitable for deep networks.
   - **He Initialization :** Especially useful for ReLU activations.
8. **Introduce Skip Connections (Residual Networks) :**
   - **Residual Networks (ResNets) :** Connections that bypass one or more layers, aiding in the training of very deep networks.
9. **Use Different Architectures :**
   - **Convolutional Neural Networks (CNNs) :** Suitable for image data.
   - **Recurrent Neural Networks (RNNs) :** Suitable for sequential data.
   - **Transformer Architectures :** Effective for sequence-to-sequence tasks.
10. **Early Stopping :**
    - Monitor the validation loss and stop training when it stops improving to prevent overfitting.
11. **Data Augmentation :**
    - Introduce variations to training data, especially useful in computer vision tasks.
12. **Feature Scaling :**
    - Ensure consistent scaling of input features, especially for algorithms sensitive to scale.

• Conclusion

In summary, the implemented neural network exhibited promising learning capabilities during the experiment. Trained on a small labeled dataset, the model demonstrated improved accuracy, effectively adapting its weights through gradient descent. Visualization tools such as Matplotlib were employed to monitor the training process, showcasing a reduction in error over iterations. Despite its success on the provided dataset, there is room for enhancement through hyperparameter tuning, regularization, and exploring more complex architectures. This foundational script provides a valuable starting point for future experiments and applications in the dynamic field of neural networks.