Reyhane Shahrokhian 99521361

HomeWork3 of Computational Intelligence Course

Dr. Mozayeni

Q1:

x_1	x_2	w_1	w_2
1	0	0.2	0.9
0	1	0.4	0.7
0	1	0.6	0.5
0	0	0.8	0.3

α = 0.5, $d_j = \sum (w_{ij} - x_i)^2$

For each neighborhood of j that d_j is minimum(the winner):

$w_{ij}(new) = w_{ij}(old) + \alpha(x_i - w_{ij}(old))$

Round1:

$d_1 = (1 - 0.2)^2 + (0 - 0.4)^2 + (0 - 0.6)^2 + (0 - 0.8)^2 = 1.8$

$d_2 = (1 - 0.9)^2 + (0 - 0.7)^2 + (0 - 0.5)^2 + (0 - 0.3)^2 = 0.84$

$w_{12} = 0.9 + 0.5 \times (1 - 0.9) = 0.95$

$w_{22} = 0.7 + 0.5 \times (0 - 0.7) = 0.35$

$w_{32} = 0.5 + 0.5 \times (0 - 0.5) = 0.25$

$w_{42} = 0.3 + 0.5 \times (0 - 0.3) = 0.15$

Round2:

$d_1 = (0 - 0.2)^2 + (1 - 0.4)^2 + (1 - 0.6)^2 + (0 - 0.8)^2 = 1.2$

$d_2 = (0 - 0.95)^2 + (1 - 0.35)^2 + (1 - 0.25)^2 + (0 - 0.15)^2 = 1.91$

$w_{11} = 0.2 + 0.5 \times (0 - 0.2) = 0.1$

$w_{21} = 0.4 + 0.5 \times (1 - 0.4) = 0.7$

$w_{31} = 0.6 + 0.5 \times (1 - 0.6) = 0.8$

$w_{41} = 0.8 + 0.5 \times (0 - 0.8) = 0.4$

This process should be continued for more iterations until the weights converge and the map accurately represents the input data's structure. To find the coverage, the changes in the weights can be monitored, and if they become very small or remain constant over several iterations, the algorithm may have converged.

Q2:

First of all, we should compute the weight matrix based on this formula: $w_{ij} = \sum_{k=1}^{p} x_i^k x_j^k$

4	4	0	0
4	4	0	0
0	0	4	4
0	0	4	4

In Hopfield network, the energy function is: $E =- \frac{1}{2}\sum_{i=1}^{p}\sum_{j=1}^{p} w_{ij}x_i x_j$

$$E_1 =- \frac{1}{2}[(4 \times 1 \times 1) + (4 \times 1 \times 1) + (0 \times 1 \times -1) + (0 \times 1 \times -1)$$
$$+ (4 \times 1 \times 1) + (4 \times 1 \times 1) + (0 \times 1 \times -1) + (0 \times 1 \times -1)$$
$$+ (0 \times -1 \times 1) + (0 \times -1 \times 1) + (4 \times -1 \times -1) + (4 \times -1 \times -1)$$
$$+ (0 \times -1 \times 1) + (0 \times -1 \times 1) + (4 \times -1 \times -1) + (4 \times -1 \times -1) =- 16$$

$$
\begin{aligned}
E_2 =& -\frac{1}{2}[(4 \times -1 \times -1) + (4 \times -1 \times -1) + (0 \times -1 \times 1) + (0 \times -1 \times 1) \\
&+ (4 \times -1 \times -1) + (4 \times -1 \times -1) + (0 \times -1 \times 1) + (0 \times -1 \times 1) \\
&+ (0 \times 1 \times -1) + (0 \times 1 \times -1) + (4 \times 1 \times 1) + (4 \times 1 \times 1) \\
&+ (0 \times 1 \times -1) + (0 \times 1 \times -1) + (4 \times 1 \times 1) + (4 \times 1 \times 1) = -16
\end{aligned}
$$

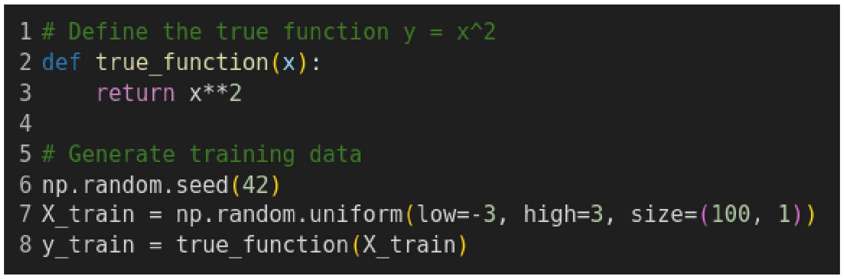
$$
\begin{aligned}
E_3 =& -\frac{1}{2}[(4 \times -1 \times -1) + (4 \times -1 \times -1) + (0 \times -1 \times -1) + (0 \times -1 \times -1) \\
&+ (4 \times -1 \times -1) + (4 \times -1 \times -1) + (0 \times -1 \times -1) + (0 \times -1 \times -1) \\
&+ (0 \times -1 \times -1) + (0 \times -1 \times -1) + (4 \times -1 \times -1) + (4 \times -1 \times -1) \\
&+ (0 \times -1 \times -1) + (0 \times -1 \times -1) + (4 \times -1 \times -1) + (4 \times -1 \times -1) = -16
\end{aligned}
$$

$$
\begin{aligned}
E_4 =& -\frac{1}{2}[(4 \times 1 \times 1) + (4 \times 1 \times 1) + (0 \times 1 \times 1) + (0 \times 1 \times 1) \\
&+ (4 \times 1 \times 1) + (4 \times 1 \times 1) + (0 \times 1 \times 1) + (0 \times 1 \times 1) \\
&+ (0 \times 1 \times 1) + (0 \times 1 \times 1) + (4 \times 1 \times 1) + (4 \times 1 \times 1) \\
&+ (0 \times 1 \times 1) + (0 \times 1 \times 1) + (4 \times 1 \times 1) + (4 \times 1 \times 1) = -16
\end{aligned}
$$

As they have negative Energies, so they can converge and they're solvable.

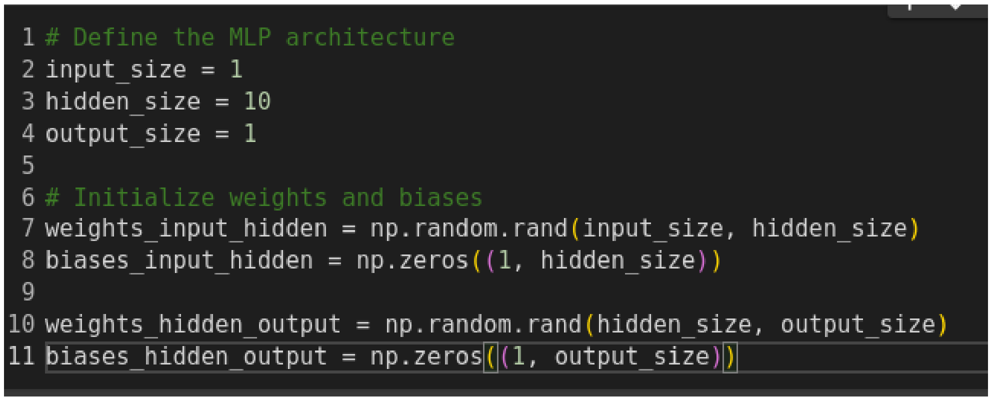
Q3:

In that case, to train the model to learn $y = x^2$, first training data should be generated.



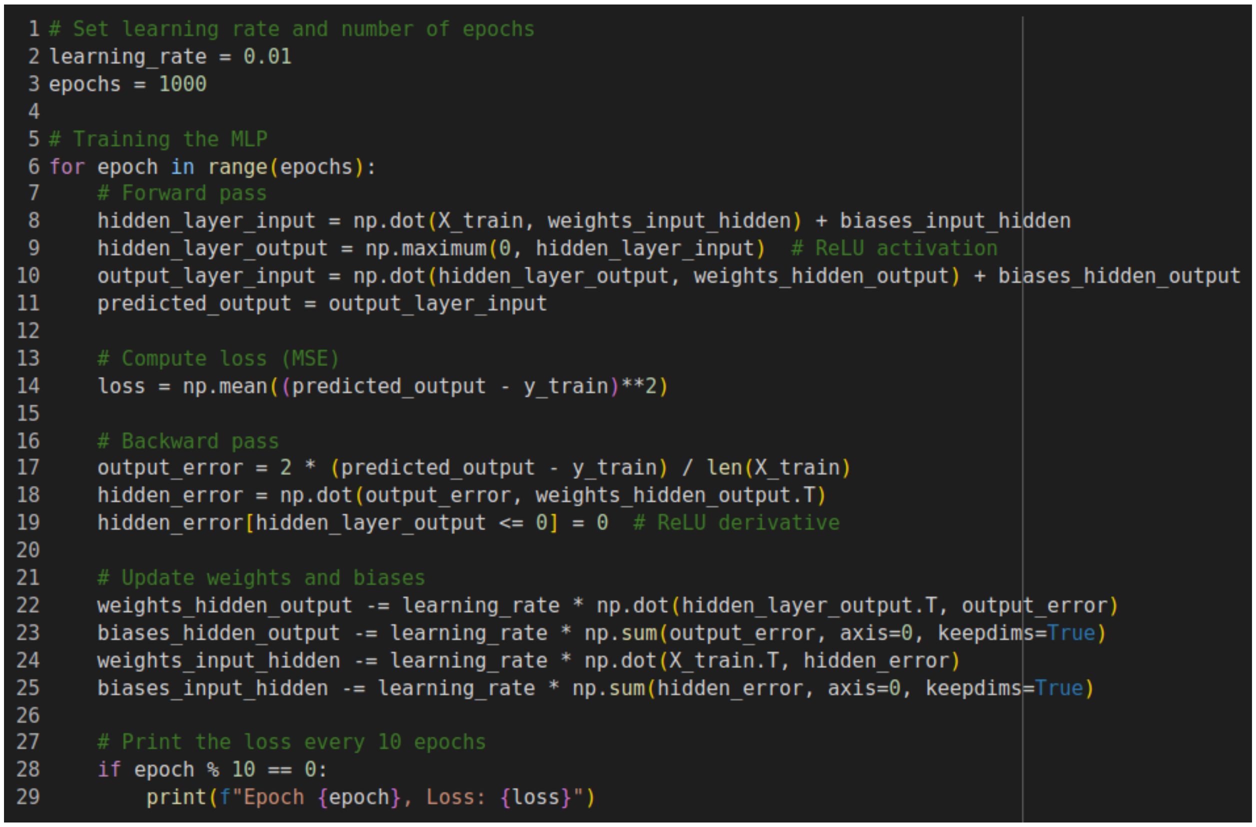
```python
# Define the true function y = x^2
def true_function(x):
    return x**2

# Generate training data
np.random.seed(42)
X_train = np.random.uniform(low=-3, high=3, size=(100, 1))
y_train = true_function(X_train)
```

Then, the weights and bias and also the layer sizes should be initialized.



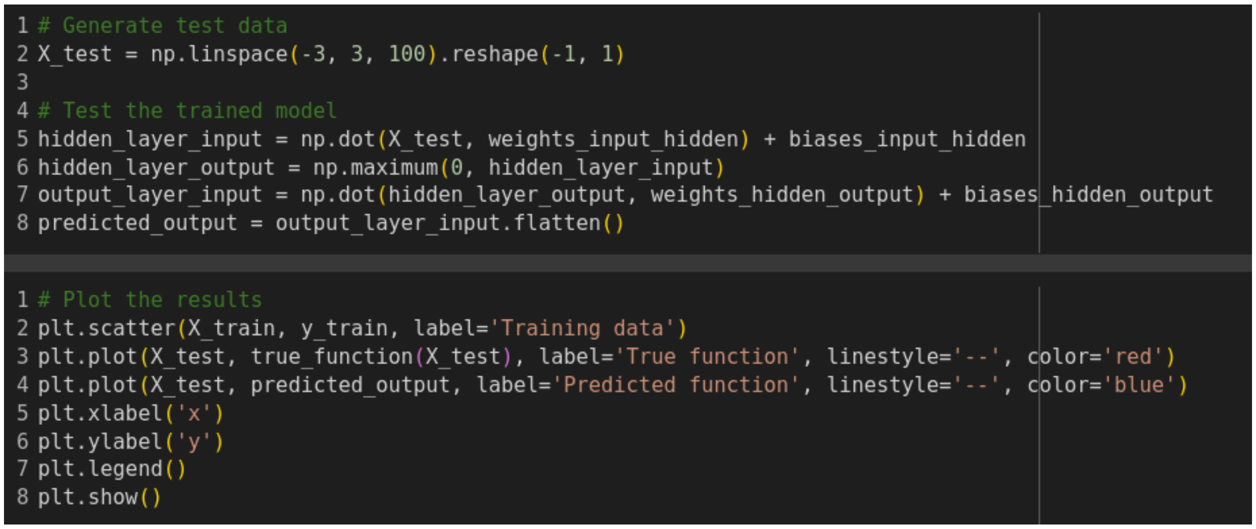
```python
# Define the MLP architecture
input_size = 1
hidden_size = 10
output_size = 1

# Initialize weights and biases
weights_input_hidden = np.random.rand(input_size, hidden_size)
biases_input_hidden = np.zeros((1, hidden_size))

weights_hidden_output = np.random.rand(hidden_size, output_size)
biases_hidden_output = np.zeros((1, output_size))
```

And now go through the training loop that contains forward pass, loss computations, backward pass and weight and bias updating.



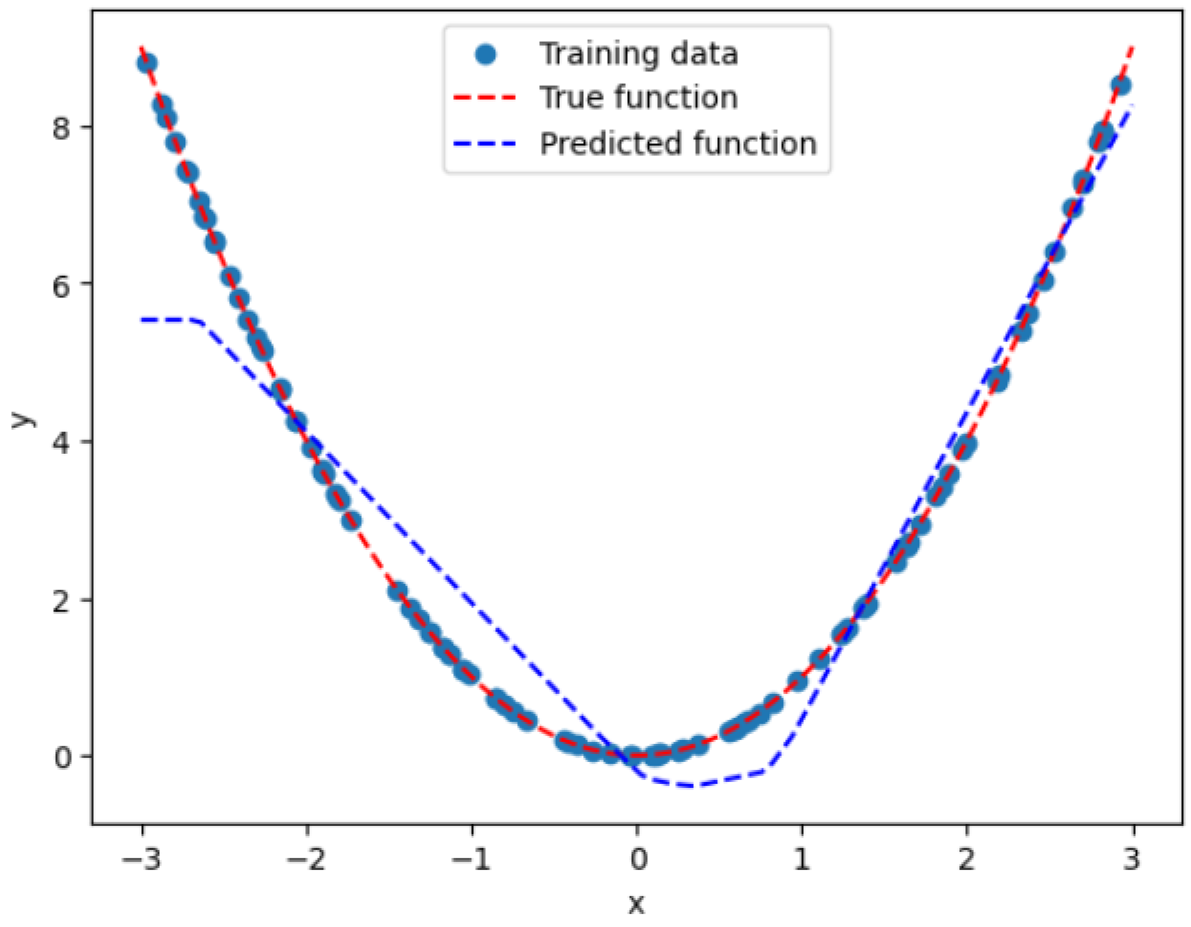
```python
# Set learning rate and number of epochs
learning_rate = 0.01
epochs = 1000

# Training the MLP
for epoch in range(epochs):
    # Forward pass
    hidden_layer_input = np.dot(X_train, weights_input_hidden) + biases_input_hidden
    hidden_layer_output = np.maximum(0, hidden_layer_input)  # ReLU activation
    output_layer_input = np.dot(hidden_layer_output, weights_hidden_output) + biases_hidden_output
    predicted_output = output_layer_input

    # Compute loss (MSE)
    loss = np.mean((predicted_output - y_train)**2)

    # Backward pass
    output_error = 2 * (predicted_output - y_train) / len(X_train)
    hidden_error = np.dot(output_error, weights_hidden_output.T)
    hidden_error[hidden_layer_output <= 0] = 0  # ReLU derivative

    # Update weights and biases
    weights_hidden_output -= learning_rate * np.dot(hidden_layer_output.T, output_error)
    biases_hidden_output -= learning_rate * np.sum(output_error, axis=0, keepdims=True)
    weights_input_hidden -= learning_rate * np.dot(X_train.T, hidden_error)
    biases_input_hidden -= learning_rate * np.sum(hidden_error, axis=0, keepdims=True)

    # Print the loss every 10 epochs
    if epoch % 10 == 0:
        print(f"Epoch {epoch}, Loss: {loss}")
```

When training ends, some test data is generated and then given to the model in order to check the accuracy of that. At the end by plotting the model output and correct output, the accuracy can be compared.



```python
# Generate test data
X_test = np.linspace(-3, 3, 100).reshape(-1, 1)

# Test the trained model
hidden_layer_input = np.dot(X_test, weights_input_hidden) + biases_input_hidden
hidden_layer_output = np.maximum(0, hidden_layer_input)
output_layer_input = np.dot(hidden_layer_output, weights_hidden_output) + biases_hidden_output
predicted_output = output_layer_input.flatten()
```

```python
# Plot the results
plt.scatter(X_train, y_train, label='Training data')
plt.plot(X_test, true_function(X_test), label='True function', linestyle='--', color='red')
plt.plot(X_test, predicted_output, label='Predicted function', linestyle='--', color='blue')
plt.xlabel('x')
plt.ylabel('y')
plt.legend()
plt.show()
```



Q4:

As the output should be 111100, I initialize the weights in a way that the last two rows and columns become negative and other positive and the threshold is set as 0.

1	0	1	1	-1	-1
0	1	0	0	0	-1
1	0	1	1	-1	0
0	1	0	1	-1	-1
0	1	-1	-1	0	-1
-1	-1	1	0	-1	0

Computations:

	x_1	x_2	x_3	x_4	x_5	x_6
Input(t = 0)	0	1	0	0	0	0
Input(t = 1)	1	1	1	1	1	0
Input(t = 2)	1	1	1	1	0	0
Input(t = 3)	1	1	1	1	0	0
$\sum_i x_i w_{ij}$						
	0	1	0	1	1	-1
	2	1	2	1	-1	-2
	3	1	3	2	-1	-1

As we can see, the model could converge at 111100.

Q5:

- Hopfield Network:

  Hopfield networks are a type of recurrent neural network (RNN) that can be used for optimization problems. They are designed to store and retrieve patterns and have been applied to solve combinatorial optimization problems. In the context of TSP, the cities can be represented as nodes, and the network's weights can be adjusted to minimize the total distance traveled. The network can be trained to converge to a state where the order of nodes corresponds to the optimal TSP solution. But it might not guarantee finding the global optimum.
- Multilayer Perceptron (MLP):

  MLPs are feedforward neural networks that can be used for a variety of tasks, including regression and optimization. They can be applied to TSP by treating it as a regression problem where the network outputs the optimal order of cities. The input layer represents the cities, and the output layer produces a permutation of cities.

  However, Training an MLP for TSP can be challenging, and finding an optimal architecture and training strategy may require significant effort. MLPs might struggle with capturing the combinatorial nature of the TSP, and their performance may vary based on problem size and complexity.
- Self-Organizing Map (SOM):

  SOMs are not originally designed for optimization problems, and adapting them for TSP may require additional considerations. They are more commonly used for clustering and visualization tasks. But still we could try implementing it with SOM:

If instead of a grid we declare a *circular array of neurons*, each node will only be conscious of the neurons in front of and behind it. That is, the inner similarity will work just in one dimension. Making this slight modification, the self-organizing map will behave as an elastic ring, getting closer to the cities but trying to minimize the perimeter of it thanks to the neighborhood function. To ensure the convergence of it, we can include a learning rate, α , to control the exploration and exploitation of the algorithm. To obtain high exploration first, and high exploitation after that in the execution, we must include a decay in both the neighborhood function and the learning rate. Decaying the learning rate will ensure less aggressive displacement of the neurons around the model, and decaying the neighborhood will result in a more moderate exploitation of the local minima of each part of the model. Then, our regression can be expressed as:

$$nt + 1 = nt + \alpha t \cdot g(we, ht) \cdot \Delta(e, nt)$$

Where α is the learning rate at a given time, and g is the Gaussian function centered in a winner and with a neighborhood dispersion of h. The decay function consists of simply multiplying the two given discounts, γ , for the learning rate and the neighborhood distance.

$$\alpha t + 1 = \gamma\alpha \cdot \alpha t, ht + 1 = \gamma h \cdot ht$$

Finally, to obtain the route from the SOM, we associate a city with its winner neuron, traverse the ring starting from any point and sort the cities by order of appearance of their winner neuron in the ring. If several cities map to the same neuron, it is because the order of traversing such cities have not been contemplated by the SOM (due to lack of relevance for the final distance or because of not enough precision). In that case, any possible order can be considered for such cities.