Heaven's Light is Our Guide

Computer Science and Engineering
Rajshahi University of Engineering and Technology

Course No: CSE 4204

Course Title: Sessional Based on CSE 4203

No of Experiment:03

Name of the Experiment: Design and implementation of Multi-layer Neural Networks algorithm (Back-propagation learning neuralnetworks algorithm)

Course Outcomes:CO1

Learning Domain with Level: Cognitive (Applying, Analyzing,Evaluating and Creating)

Submitted To

Rizoan Toufiq

Assistant Professor

Department of Computer Science and Engineering

Rajshahi University of Engineering and Technology

Submitted By

Name:Rafi Ahammed Songram

Roll:1803095

Department:Computer Science and Engineering

Rajshahi University of Engineering and Technology

Design and implementation of Multi Layer Neural Network Algorithm.

Content Page

Multi-layer Neural Networks algorithm:

A Multi-layer Neural Network, also known as an Artificial Neural Network, is a popular machine learning algorithm used for various tasks, such as image recognition, natural language processing, and predictive analytics. It is a type of feed-forward network that consists of multiple layers of interconnected nodes or neurons. Each layer performs computations on the input data, transforming it into a format that is more suitable for the output layer.

The network's connections between the nodes are weighted, and the weights are adjusted during training using backpropagation to minimize the error between the predicted output and the actual output.

The basic structure of a multi-layer neural network consists of three types of layers: input, hidden, and output layers. The input layer receives the input data, and the output layer produces the final output of the network. The hidden layers perform computations on the input data, transforming it into a format that is more suitable for the output layer.

Each layer is composed of multiple nodes or neurons, which are connected to the nodes in the next layer. The connections between the nodes are weighted, which means that each connection has a certain strength that determines how much it contributes to the output of the next layer.

Steps of Multi-layer Neural Networks algorithm:

Step-1: Initialize weights and threshold. Set all weights and threshold to small random values.
Step-2: Present input and desired output Present input $x_p = x_0$, x_1... and target output $T_p = t_0$ t_1,.... where n is the number of input nodes and m is the number of output nodes. Set w_0 to be $-\theta$, the bias and x_0 to be always 1. For pattern association, X_p and T_p represent the patterns to be associated. For classification, T_p is set to zero except for one element set to 1 that corresponds to the class that X_p is in.

Step-3: Calculate actual output:

$$y_{pj} = f \sum_{i=0}^{n} w_i(t) x_i(t)$$

passes that as input to the next layer. The final layer outputs values O_{pj}

Step-4: Adapt weights:
Start from the output layer, and work backwards.

W_{ij}(t+1) = W_{ij}(t) + n$\delta_{pj} O_{pj}$
W_{ij} represents the weights from node i to node j at time t, n is gain term and δ_{pj} is the error term for node p and j.

For output units:δ_{pj} = k $\emptyset_{pj}$(1-δ_{pj})*(t_{pj} -O_{pj})

For hidden units: δ_{pj} = k $\emptyset_{pj}$(1-δ_{pj}) $\sum \delta_{pk} w_{pk}$

Dataset:

The name of the dataset is **Diabetes** and csv file diabetes.csv

Table 1: Attribute of Dataset

Column	Count	Type
1 pregnencies	768	int64
2.Glucose	768	int64
3.BloodPressure	768	int64
4.SkinThickness	768	int64
5.Insulin	768	int64
6.BMI	768	float64
7.Diabetespedigrefunction	768	float64
8.Age	768	int64
9.Outcome	768	int64

Here are the all attribute of full dataset and we apply on this single layer perceptron algorithm .

Outcome plot of dataset:

Code-snippet

```
ax = sns.countplot(df["diagnosis"],label="Count")
f, ax = plt.subplots(figsize=(2, 2))
sns.countplot(df, x="Outcome", ax=ax)
```

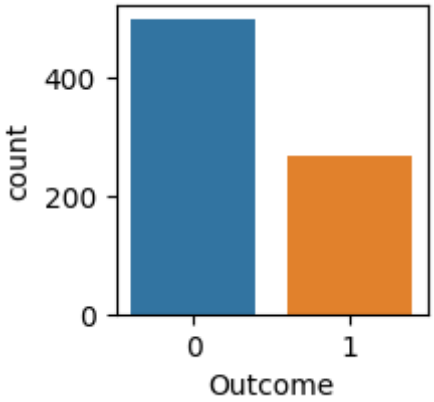

Figure-1: Dataset Outcome Count Plot

Correlation Matrix :

A correlation matrix shows the correlation between different variables in a matrix setting. However, because these matrices have so many numbers on them, they can be difficult to follow. Heatmap coloring of the matrix, where one color indicates a positive correlation, another indicates a negative correlation, and the shade indicates the strength of correlation, can make these matrices easier for the reader to understand.

Code-snippet

```
corr = df.corr(method = 'pearson')
f, ax = plt.subplots(figsize=(5, 5))
cmap = sns.diverging_palette(10, 275, as_cmap=True)
sns.heatmap(corr, cmap=cmap, square=True, linewidths=0.5,
cbar_kws={"shrink": 0.5}, ax=ax)
```

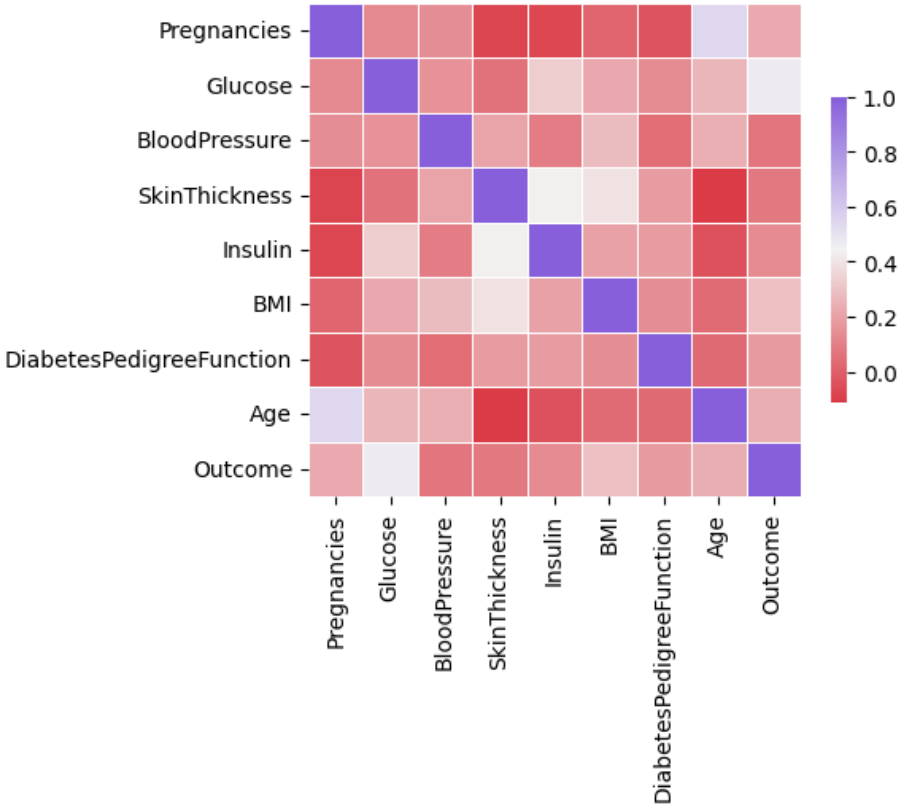

Figure-2: Correlation Heatmap

Split Dataset

Separates the features and target variable, and then splits them into training and testing sets. The training set (Xtrain and Ytrain) is used to train a machine learning model, while the testing set (Xtest and Ytest) is used to evaluate the model's performance.Finally single lair perceptron algorithm will be applied .Here test size = 0.1

```
X_train, X_test,
y_train,y_test = train_test_split(X,y, test_size=0.2)
```

Multi Layer Neural Network Algorithm

No of class size and hidden layers taking as input and using forward and backward propagation output will be calculated.Every iteration self.weight of hidden unit and weight of output unit calculated. sigmoid function was

used to calculated predicted output of hidden unit.
Code Snippet:

```
class MLP:
def __init__(self, num_inputs, num_hidden,
weight_of_hidden_units, weight_of_output_units):
self.num_of_input_units = num_inputs
self.num_of_hidden_units = num_hidden
self.num_of_output_units = 1
if weight_of_hidden_units is None:
self.weight_of_hidden_units = np.random.rand
(self.num_of_input_units, self.num_of_hidden_units)
else:
self.weight_of_hidden_units = weight_of_hidden_units
if weight_of_output_units is None:
self.weight_of_output_units = np.random.rand
(self.num_of_hidden_units, self.num_of_output_units)
else:
self.weight_of_output_units = weight_of_output_units
```

Function
Code Snippet:

```
def sigmoid(self, x):
return 1 / (1 + np.exp(-x))
def fit(self, X, y, learning_rate=0.1, epochs=20):
```

Forward propagation-Hidden Unit
Code Snippet:

```
for epoch in range(epochs):
for idx, x in enumerate(X):
input_of_hidden_units = X.T
net_of_hidden_units = np.dot(
self.weight_of_hidden_units.T, input_of_hidden_units)
output_of_hidden_units = self.sigmoid(net_of_hidden_units)
print("output_of_hidden_units: \n", output_of_hidden_units)
```

```
input_of_output_units = output_of_hidden_units
net_of_output_units = np.dot
(self.weight_of_output_units.T, input_of_output_units)
output_of_output_units = self.sigmoid(net_of_output_units)
```

Backward Propagation-Output Unit

Using the algorithm -folmula is used and finally back propagation tecnique every iteration weight calculated,error calculated of hiddden units and sum and finally result calculated.
Code Snippet:

```
error_of_output_units = np.multiply
((y[idx] - output_of_output_units),
output_of_output_units * (1 - output_of_output_units))
delta_weight_of_output_units = learning_rate * np.dot
(input_of_output_units, error_of_output_units.T)
# print("delta_weight_of_output_units: \n",
delta_weight_of_output_units)
self.weight_of_output_units += delta_weight_of_output_units
# print("weight_of_output_units: \n", self.weight_of_output_units)
error_of_hidden_units = np.multiply(np.dot
(self.weight_of_output_units, error_of_output_units),
output_of_hidden_units * (1 - output_of_hidden_units))
delta_weight_of_hidden_units =
learning_rate * np.dot(input_of_hidden_units,
error_of_hidden_units.T)
self.weight_of_hidden_units += delta_weight_of_hidden_units
print("weight_of_hidden_units: \n", self.weight_of_hidden_units)
```

Model Fit

Finally model was fitted and no of epoch is 20 and accuracy was found of this model on this dataset

Code-Snippet:

```
def predict(self, X):
```

```
pred = np.array([])
for idx, x in enumerate(X):
print(f'x: \n', x)
input_of_hidden_units = x.T
net_of_hidden_units = np.dot
(self.weight_of_hidden_units.T, input_of_hidden_units)
print('NH\n', net_of_hidden_units)
output_of_hidden_units = self.sigmoid(net_of_hidden_units)
print(f'OH: \n', output_of_hidden_units)
input_of_output_units = output_of_hidden_units
net_of_output_units = np.dot
(self.weight_of_output_units.T, input_of_output_units)
output_of_output_units = self.sigmoid(net_of_output_units)
pred = np.append(pred, output_of_output_units)
return pred
if __name__ == '__main__':
X = df.iloc[:, :7]
y = df.iloc[:, 8]
X = np.array(X)
y = np.array(y)
X_train, X_test, y_train, y_test = train_test_split
(X, y, test_size=0.2)
clf = MLPClassifier(hidden_layer_sizes=(30, 30),
max_iter=100, alpha=0.0001, tol=0.000000001)
#clf = MLP(8, 30, None, None)
clf.fit(X_train,Y_train,epochs=20)
```

Evaluate Model and Accuracy

The input layer receives the input data, and the output layer produces the final output of the network. The hidden layers perform computations on the input data, transforming it into a format that is more suitable for the output layer.and the accuracy is- Code-Snippet:

```
loss = clf.loss_curve_
import matplotlib.pyplot as plt
plt.plot(loss)
plt.show()
pred = clf.predict(X_test)
print("pred: ", pred)
print("y_test: ", y_test)
print("Testing accuracy: ", acc)
acc = np.sum(pred == y_test) / len(y_test)
print("Testing accuracy: ", acc)
```

Training Accuracy: 0.7942885011857143
Testing Accuracy : 0.7842857142857143

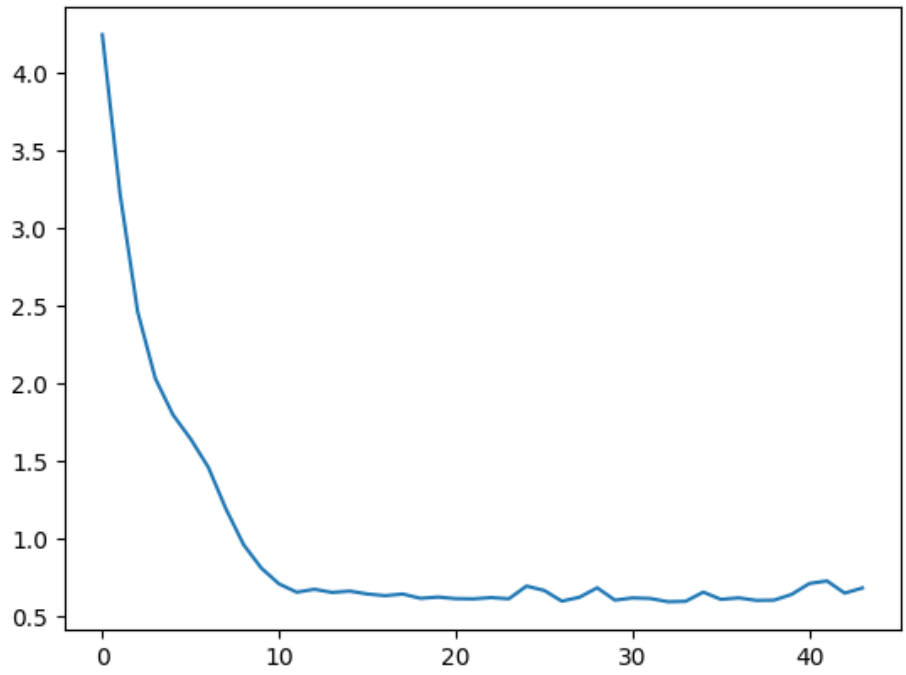

Figure-3:Loss Curve

Evaluation(Confusion) Matrics

confusion matrix visualize and summarize the performance of this trained model performance on trainning and testing and code seems correct and should display a heatmap with annotations for True Positives, True Negatives, False Positives, and False Negatives.
Confusion Matrix:

```
import seaborn as sn
import pandas as pd
from sklearn.metrics import confusion_matrix
import seaborn as sns
cm = confusion_matrix(y_test, pred)
sns.heatmap(cm, annot=True)
plt.show()
plt.figure(figsize = (5,5))
```

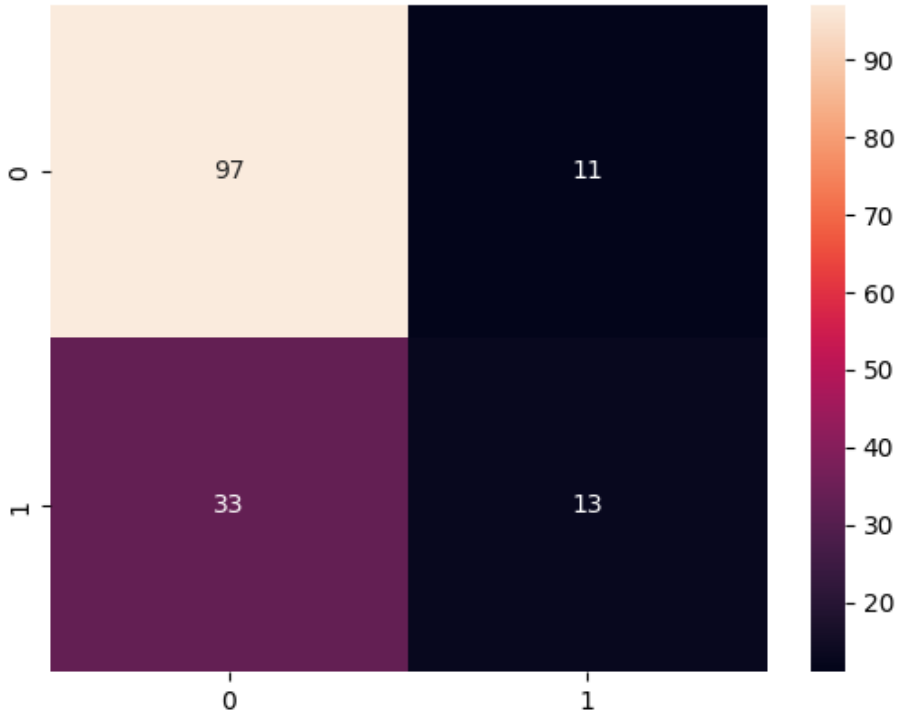

Figure-4:Confusion Matrix

Scatter-Plot:Predicted

```
# plot scatter plot
from sklearn.metrics import confusion_matrix
import seaborn as sns
plt.scatter(X_test[:, 0], X_test[:, 1], c=pred)
plt.title("Predicted")
plt.show()
```

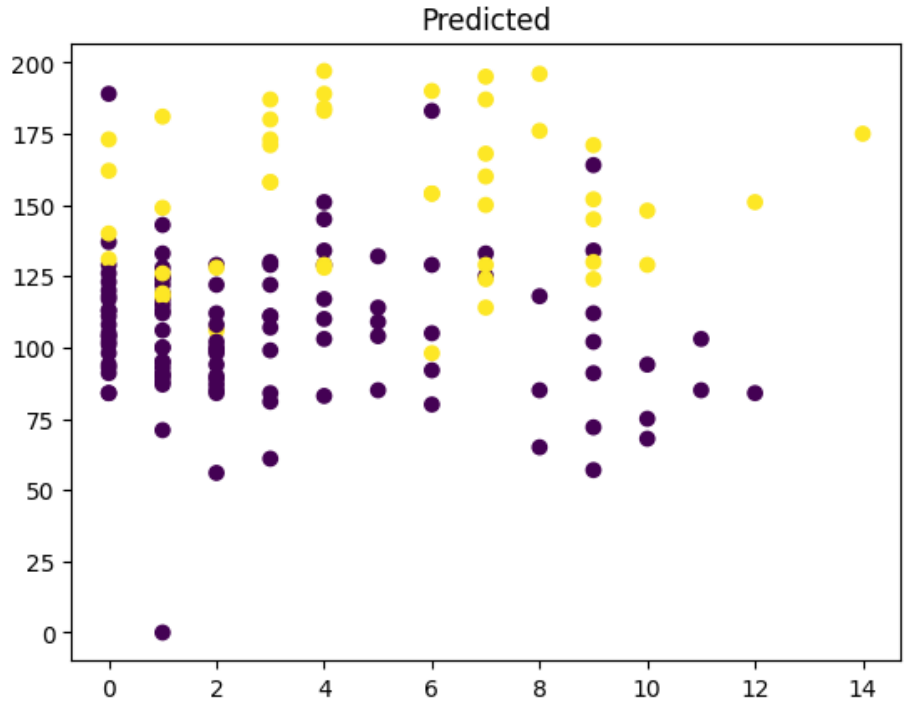

Figure-5:Scatter-Plot:Predicted

Scatter-Plot:Actual

```
# plot scatter plot
from sklearn.metrics import confusion_matrix
import seaborn as sns
plt.scatter(X_test[:, 0], X_test[:, 1], c=y_test)
plt.title("Actual")
plt.show()
```

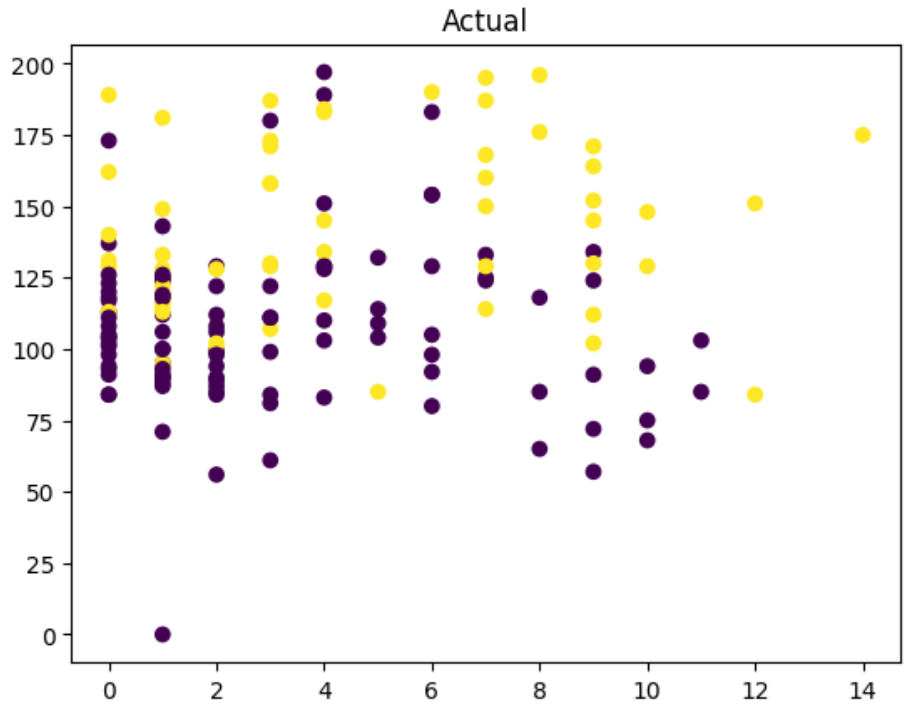

Figure-6:Scatter-Plot:Actual

Limitations:

1.Local minima: It occurs when the algorithm gets stuck in a suboptimal solution due to the presence of multiple minima in the error surface.

2.Underfitting: In multilayer perceptron algorithm, underfitting can occur if the neural network is too shallow or if it has too few hidden units. This means that the model is not able to learn the complex patterns in the data, resulting in poor performance on both the training and test datasets.

3.Overfitting: Multi-layer neural networks are prone to overfitting, which means that the model becomes too complex and performs well on the training data but poorly on the test data. Overfitting occurs when the model captures noise and irrelevant patterns in the data. .

4.Divergency: Divergence is a common problem in multilayer perceptron (MLP) training when the training process fails to converge and instead, the loss function increases indefinitely. This can happen when the learning

rate is too high or the batch size is too large, causing the algorithm to overshoot the optimal solution and continue to diverge from it.

Improvements

Here are some improvements and considerations for enhancing the performance and capabilities of a Multi-layer Neural Network:
1. Dataset is not enough good. Better dataset can give better result.

2.Lowering the gain term

3.Addition of internal nodes.

4.Momentum term and Addition of noise can give better performance.

5.Activation Functions: Modern activation functions such as Rectified Linear Unit (ReLU) variants address the vanishing gradient problem, promoting faster convergence and improved gradient flow.

6.Hyperparameter Tuning: Systematic optimization of hyperparameters, including learning rates and network architecture, contributes to better model performance.

Conclusion

Multilayer neural networks, while powerful for complex tasks, face challenges of computational intensity, potential overfitting, and a demand for large datasets. Their versatility and state-of-the-art performance make them indispensable in various domains. Ongoing efforts to enhance interpretability and efficiency underscore their significance, promising continued advancements in deep learning applications.

References

[1] Neural Computing: An Introduction by R-Beale and T-Jackson

[2] Back Propagation Algorithm: The Best Algorithm Among the Multi-layer Perceptron Algorithm -IJCSNS International Journal of Computer Science and Network Security, VOL.9 No.4, April 2009)