

B.Sc. IT (Hons) Software Development

Faculty of Information & Communication Technology (ICT)

**Study Unit** – Business Intelligence

**Code** –CIS3187

Assignment

Shaizel Victoria Bezzina (0343703L)

GitHub repository

A close up of a signature

Description automatically generatedA close-up of a text

Description automatically generatedA document with text on it

Description automatically generated

Shaizel Victoria Bezzina

Business Intelligence Assignment Shaizel Victoria Bezzina (0343703L)

CIS3187

09/01/2024

**Source code**

# Artificial Neural Network for Iris Dataset Classification  
  
import pandas as pd  
import numpy as np  
from sklearn.model\_selection import train\_test\_split  
from sklearn.preprocessing import StandardScaler  
from sklearn.datasets import load\_iris  
import matplotlib.pyplot as plt  
  
# Loading the iris dataset  
iris\_df = load\_iris()  
data = pd.DataFrame(data= np.c\_[iris\_df['data'], iris\_df['target']], columns= iris\_df['feature\_names'] + ['target']) #concatinating the feature data and the target data for data handling and analysis tasks  
  
# one-hot encoding the target variable  
target\_one\_hot = pd.get\_dummies(data['target']).values  
  
# Splitting the dataset into training set 80% = 120 sets and 20% = 30 sets  
train\_data, test\_data, train\_target, test\_target = train\_test\_split(data.iloc[:, :4], target\_one\_hot, test\_size = 0.2, random\_state = 42)  
  
# Standarizing the features  
scaler = StandardScaler()  
train\_data = scaler.fit\_transform(train\_data)  
test\_data = scaler.transform(test\_data)  
  
# Stating the parameters  
input\_neurons = 4  
hidden\_neurons = 4  
output\_neurons = 3  
learning\_rate = 0.2  
error\_threshold = 0.2  
  
# Sigmoid Activation function (will be used for backward and forward propogation)  
def sigmoid(x):  
 return 1 / (1 + np.exp(-x))  
  
# Derivative of the Sigmoid Activation function (will be used for backward and forward propogation)  
def sigmoid\_derivative(x):  
 return x \* (1 - x)  
  
# Initialising the weight and the biases  
np.random.seed(42)  
weights\_input\_hidden = np.random.rand(input\_neurons, hidden\_neurons)  
weights\_hidden\_output = np.random.rand(hidden\_neurons, output\_neurons)  
bias\_hidden = np.zeros((1, hidden\_neurons))  
bias\_output = np.zeros((1, output\_neurons))  
  
# Training the Neural Network  
epochs = 10000  
mse\_history = []  
bad\_epochs = []  
  
  
for epoch in range(epochs):  
 # Forward Propagation  
 hidden\_layer\_input = np.dot(train\_data, weights\_input\_hidden) + bias\_hidden  
 hidden\_layer\_output = sigmoid(hidden\_layer\_input)  
  
 output\_layer\_input = np.dot(hidden\_layer\_output, weights\_hidden\_output) + bias\_output  
 predicted\_output = sigmoid(output\_layer\_input)  
  
 # Backwards propagation  
 output\_error = train\_target - predicted\_output  
 output\_delta = output\_error \* sigmoid\_derivative(predicted\_output)  
  
 hidden\_layer\_error = output\_delta.dot(weights\_hidden\_output.T)  
 hidden\_layer\_delta = hidden\_layer\_error \* sigmoid\_derivative(hidden\_layer\_output)  
  
 # Updating the Weights and Biases  
 weights\_hidden\_output += hidden\_layer\_output.T.dot(output\_delta) \* learning\_rate  
 weights\_input\_hidden += train\_data.T.dot(hidden\_layer\_delta) \* learning\_rate  
 bias\_output += np.sum(output\_delta, axis=0, keepdims=True) \* learning\_rate  
 bias\_hidden += np.sum(hidden\_layer\_delta, axis=0, keepdims=True) \* learning\_rate  
  
 # Calculating the Mean Squared Error  
 mse = np.mean(np.square(output\_error))  
 mse\_history.append(mse)  
  
 # Counting mis-classifications (bad epochs)  
 bad\_epochs\_count = np.sum(np.argmax(predicted\_output, axis=1) != np.argmax(train\_target, axis=1))  
 bad\_epochs.append(bad\_epochs\_count)  
  
 # Printing the error every 1000 epochs  
 if epoch % 1000 == 0:  
 print(f"Epoch {epoch}, Mean Squared Error: {mse}, Bad Epochs: {bad\_epochs\_count}")  
  
 # Stopping the training if error is below the threshold  
 if mse < error\_threshold:  
 print(f"Training complete!")  
 print(f"Mean Squared Error: {mse}")  
 break  
  
# Plotting the mean squared error's history  
plt.figure(figsize=(12, 5))  
  
plt.subplot(1, 2, 1)  
plt.plot(mse\_history)  
plt.title('Mean Squared Error During Training')  
plt.xlabel('Epochs (in thousands)')  
plt.ylabel('Mean Squared Error')  
  
plt.subplot(1, 2, 2)  
plt.plot(bad\_epochs, color='purple')  
plt.title('Bad Epochs During Training')  
plt.xlabel('Epochs (in thousands)')  
plt.ylabel('Bad Epochs')  
  
plt.show()  
  
# Testing the Neural Network  
hidden\_layer\_input\_test = np.dot(test\_data, weights\_input\_hidden) + bias\_hidden  
hidden\_layer\_output\_test = sigmoid(hidden\_layer\_input\_test)  
  
output\_layer\_input\_test = np.dot(hidden\_layer\_output\_test, weights\_hidden\_output) + bias\_output  
predicted\_output\_test = sigmoid(output\_layer\_input\_test)  
  
# Convertinf one-hot encoded predictions to the class labels  
predicted\_labels = np.argmax(predicted\_output\_test, axis=1)  
actual\_labels = np.argmax(test\_target, axis=1)  
  
# Calculating accuracy  
accuracy = np.mean(predicted\_labels == actual\_labels)  
print(f"Test Accuracy: {accuracy}")

**Dataset**

The dataset used is the Iris dataset. This dataset was loaded using the **load\_iris** function from **sklearn.datasets.** The dataset is used to train and test the neural network. The feature data and the target data are concatenated into a Pandas DataFrame, which is used for any arithmetic calculations. The target variable is one-hot encoded.

The Iris dataset can be found here;

<https://www.wikiwand.com/en/Iris_flower_data_set>

**Source Code:**

from sklearn.datasets import load\_iris

**# one-hot encoding the target variable**  
target\_one\_hot = pd.get\_dummies(data['target']).values

**# Loading the iris dataset**iris\_df = load\_iris()  
data = pd.DataFrame(data= np.c\_[iris\_df['data'], iris\_df['target']], columns= iris\_df['feature\_names'] + ['target']) #concatinating the feature data and the target data for data handling and analysis tasks

**# Standarizing the features**scaler = StandardScaler()

**Training and testing the neural network**

The Iris Dataset is randomly split into an 80%-20% ratio for training and testing, respectively. The features are standardized using the **‘scaler = StandardScaler()’.** It includes a learning rate of 0.2 and an error threshold of 0.2 as specified.

**Source code:**

**# Stating the parameters**  
learning\_rate = 0.2  
error\_threshold = 0.2

**# Splitting the dataset into training set 80% = 120 sets and 20% = 30 sets**  
train\_data, test\_data, train\_target, test\_target = train\_test\_split(data.iloc[:, :4], target\_one\_hot, test\_size = 0.2, random\_state = 42)

**# Standarizing the features**  
train\_data = scaler.fit\_transform(train\_data)  
test\_data = scaler.transform(test\_data)

**Neural network architecture**

The neural network architecture includes 4 input neurons, 4 hidden neurons, and 3 output neurons. The Sigmoid activation function is also employed for both the forward propagation and the backwards propagation steps as specified.

**Source code:**

**# Stating the parameters**input\_neurons = 4  
hidden\_neurons = 4  
output\_neurons = 3

**# Sigmoid Activation function (will be used for backward and forward propagation)**  
def sigmoid(x):  
 return 1 / (1 + np.exp(-x))  
 **# Derivative of the Sigmoid Activation function (will be used for backward and forward propagation)**  
def sigmoid\_derivative(x):  
 return x \* (1 - x)

**Training process**

The neural network is trained for a maximum of 10,000 epochs. The loss function used is the **Mean Squared Error**, which is used to monitor the training process every 1000 epochs. The training stops either when the maximum number of epochs are reached or when the Mean Squared Error falls below the specified error threshold.

**# Calculating the Mean Squared Error**  
 mse = np.mean(np.square(output\_error))  
 mse\_history.append(mse)  
  
 **#Printing the error every 1000 epochs**  
 if epoch % 1000 == 0:  
 print(f"Epoch {epoch}, Mean Squared Error: {mse}")  
  
 **# Stopping the training if error is below the threshold**  
 if mse < error\_threshold:  
 print(f"Training complete!")  
 print(f"Mean Squared Error: {mse}")  
 break  
  
**# Plotting the mean squared error's history**  
plt.plot(mse\_history)  
plt.title('Mean Squared Error During Training')  
plt.xlabel('Epochs (in thousands)')  
plt.ylabel('Mean Squared Error')  
plt.show()

**Convergence plot**

In order to visualize the convergence of the neural network, the Mean Squared Error during training is plotted.

**# Plotting the mean squared error's history**  
plt.plot(mse\_history)  
plt.title('Mean Squared Error During Training')  
plt.xlabel('Epochs (in thousands)')  
plt.ylabel('Mean Squared Error')  
plt.show()

A screenshot of a graph

Description automatically generated

**Testing results**

The neural network is tested on 20% of the dataset. The predictions are compared with the actual labels, and the accuracy of the test is calculated.

**# Testing the Neural Network**  
hidden\_layer\_input\_test = np.dot(test\_data, weights\_input\_hidden) + bias\_hidden  
hidden\_layer\_output\_test = sigmoid(hidden\_layer\_input\_test)  
  
output\_layer\_input\_test = np.dot(hidden\_layer\_output\_test, weights\_hidden\_output) + bias\_output  
predicted\_output\_test = sigmoid(output\_layer\_input\_test)

A black background with numbers

Description automatically generated

**Problems encountered**

One problem that I encountered was related to visualizing bad epochs accurately. Ensuring the plot reflected the network's learning progress required careful consideration of the data tracking mechanisms.

**Choices made**

Weight initialization was done randomly using a seed for reproducibility. This choice was made to avoid symmetry issues.

A graph with a plot of the Mean Squared Error over epochs is generated, in order to understand the network’s learning process. This choice helps in determining the effectiveness of the chosen parameters.