AI1104 – Programming for AI

Final Project Report

Prof Karthik P.N.

(Email)

Done by: Samagalla Sukesh Kumar(ai23betch11024) and

Pakki Sai Kaushik (ai23btech11017)

Task 1: Training Artificial Neural Network

“

The task in this question is to train an artificial neural network, a data set namely data_Q1.csv was given to be trained on. Modules required were **`numpy`**, **`pandas`**, **`pyplot`** from **`matplotlib`**.

The data was converted to a python dataframe with **`pd.read_csv("Yourfile.csv")`** line and stored in a **dataframe** named **`data`**. Later, as asked in the task, each (of 3) columns of the dataframe were converted to numpy arrays by storing only their values with **`.to_numpy()`** method and hence each column of the dataframe was stored in a numoy array. With names **`training_x1`**, **`training_x2`**, **`training_y`** ,each of length 200. And this length was stored in **`N`**.

Then the training dataset was created with name **`training_x`** using the following command given.

```
training_x = np.array([[x1,x2,1] for x1, x2 in zip(training_x1, training_x2)])
```

The basic model of our neural network is

$$X \xrightarrow{W^{(1)}} H \xrightarrow{\sigma} Z \xrightarrow{W^{(2)}} O \xrightarrow{\sigma} \hat{y}.$$

With the following notations:

- ***X*** is the input data (coming from training_data) of size N ×3 .
- $\boldsymbol{W^{(1)}} = [w_{k,l}^{(1)} : k,l \in \{1, 2, 3\}]$ is a weight matrix of size 3 ×3, connecting the input layer and hidden layer.
- $\boldsymbol{W^{(2)}} = [\, w_{k,1}^{(2)} : k \in \{1, 2, 3, 4\}]^T$ is a weight matrix of size 4 × 1 between the hidden layer and the output layer.

- σ is the sigmoid activation function, defined by **$\sigma(x) = 1/(1+\exp(-x))$**.
- $H = X \times W^{(1)}$ is a matrix of size N × 3 obtained at the hidden layer, before the activation function σ is applied.
- Z is a matrix of size N × 4 obtained at the hidden layer after the activation function is applied to each entry of H, followed by appending the bias column at the end.
- $O = Z \times W^{(2)}$ is a matrix of size N ×1 obtained at the output layer before the activation function is applied
- $\hat{y} = \sigma(O)$ is a matrix of size N ×1 containing the predicted values, $\hat{y}$ was stored as **`y_hat`**.

```
np.random.seed(123)
```

This line was added to the code to reproduce same outputs on various runs of our code.

A function `sigmoid(x)` was defined to return σ(x) and another function `grad_sigmoid(x)` was defined to return the `derivative of σ(x)`.

$W^{(1)}$ is represented by weights1. Initialized this 3 ×3 matrix with random floating point numbers lying between [−1, 1]

$W^{(2)}$ is represented by weights2. Initialized this 4 × 1 matrix with random floating point numbers lying between [−1, 1].

For the above task, the code used were:

```
weights1 = 2 * np.random.rand(3, 3) - 1
weights2 = 2 * np.random.rand(4, 1) - 1
```

***Logic**: np.random.rand(a,b) creates random numbers from [0,1] into a matrix of size a x b. mutlipyling this with 2, produces random numbers in range [0,2]. Which after subtracting 1 gives a matrix of size a x b and values in range [-1,1].*

The learning rate(γ) was given to be 0.05 and was stored in

```
learning_rate_gamma = 0.05
```

An empty array **`training_error`** was created to store training errors after each epoch. And `max_epochs = 100` (given).

The basic steps of training the network:

1) **Forward Pass :**

Following the notations, each variable was given it's formula and expression.

`Z = σ(H)` , and of size N x 3 with bais coulumn of all 1's at the end. And this new matrix of size N x 4 was stored in `Z` itself.

For this task, the following code was used:

```
Z = sigmoid(H)
bias_column = np.ones((Z.shape[0], 1))
Z = np.hstack((Z, bias_column))
```

Then, the loss function was calculated and appended to `training_error`. The formula of lass function is :

$$\mathcal{L}(Y, \widehat{y}) := \frac{1}{2N} \sum_{i=1}^{N} (Y_i - \widehat{y}_i)^2,$$

And the code used was :

```
loss = np.mean((training_y - y_hat) ** 2)/2
training_error.append(loss)
```

2) **Back Propagation:**

$$\frac{\partial \mathcal{L}}{\partial W_{k,1}^{(2)}} = \sum_{i=1}^{N} \frac{\partial \mathcal{L}}{\partial \widehat{y}_i} \times \frac{\partial \widehat{y}_i}{\partial O_i} \times \frac{\partial O_i}{\partial W_{k,1}^{(2)}}, \quad k \in \{1, 2, 3, 4\}.$$

The above is partial derivative of error function with respect to $W^{(2)}_{k,1}$.

`gradient2` was a matrix with same shape of `weights2`. The code of that task was:

```
gradient2 = (np.dot(Z.T, (-(training_y.reshape(200,1) - y_hat) *
                    grad_sigmoid(O))))/N
```

Similarly `gradient1` was a matrix with same shape of `weights1`. By partial derivative of loss function with respect to $W^{(1)}_{k,1}$

The formula and code for this task was :

$$\frac{\partial \mathcal{L}}{\partial W_{k,l}^{(1)}} = \sum_{i=1}^{N} \frac{\partial \mathcal{L}}{\partial \widehat{y}_i} \times \frac{\partial \widehat{y}_i}{\partial O_i} \times \frac{\partial O_i}{\partial Z_{i,l}} \times \frac{\partial Z_{i,l}}{\partial H_{i,l}} \times \frac{\partial H_{i,l}}{W_{k,l}^{(1)}}, \quad k, l \in \{1, 2, 3\}.$$

```
temp = -(training_y.reshape(200,1) - y_hat) *
                grad_sigmoid(O) * grad_sigmoid(H)

temp = temp @ np.diag(weights2[:-1].reshape(-1))

gradient1 = (np.dot(X.T, temp))/N
```

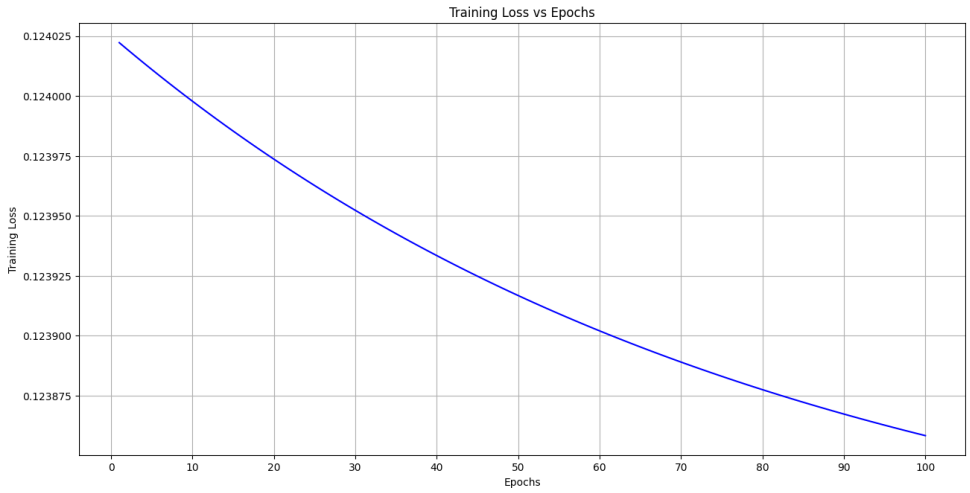
Upadating the weight matrices was done with the following code:

```
weights2 -= learning_rate_gamma * gradient2

weights1 -= learning_rate_gamma * gradient1
```

Training loss was plot on y-axis and epochs on x-axis:

And the plot is :



This competes the end of **Task 1 : Training Artificial Neural Network** ”

By: AI23BECTH11017 Sai Kaushik

Task 2: K- Means Clustering

“

The task in this question is implement a popular clustering technique called K-means clustering to cluster the points in the dataset.The modules used are **numPy** and **matplotlib.** First, I generated the two semicircle points of radius=1 each. One of the semi-circles faces downwards, centered at (0, 0), and the other faces upward, centered at (1, 0). I generated equidistant 500 points in each Individual semi-circle. Using the code snippet:

```
theta_arr = np.linspace(0,np.pi,500)
```

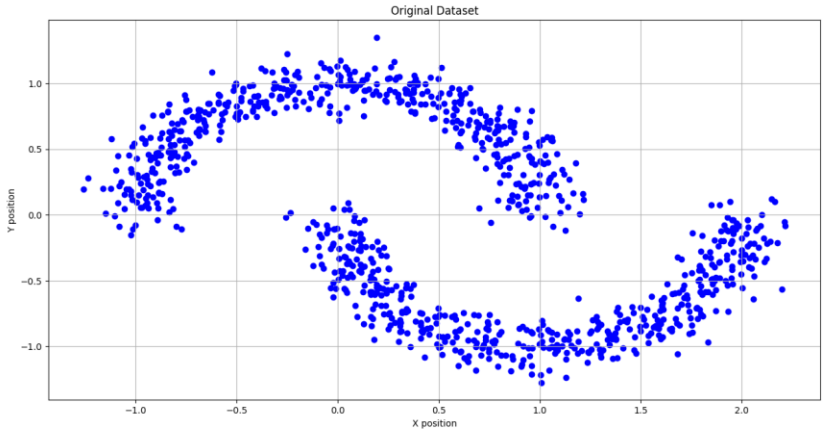
Here, `thera_arr` is the array of angle theta parameter ranging from 0 to π.

We are adding **`np.random.seed(123)`** because we want to produce the same results all the time. And then we are storing the x, y co-ordinates of the respective semi-circles in the array named as `semicircle1_x, semicircle1_y,` etc... with the help of `theta_arr` as parameter. Now, we are creating a noise to the whole co-ordinates with the help of gaussian distribution function of `μ=1` and `σ =0.1`, by adding to itself. To implement it we are using code as

```
np.random.normal(loc=0, scale=0.1, size=500)
```

We are running this code 4 times, so that we add it to all co-ordinates so there will be disturbances in each scatter point.

- We now store all the points in one single array called data_points by code
- **`d1= [(x,y) for x,y in zip(semicircle1_x,semicircle1_y)]`**
- **`d2 = [(x,y) for x,y in zip(semicircle2_x,semicircle2_y)]`**
- **`data_points = d1+ d2`**



And now our main moto is to differentiate the clusters by K- means clustering Algorithm

K-Means Clustering

There are Three steps in it:

• **Initialization:** Since our data_points lies in the region [−2, 3] × [−2, 2] we must pick two points uniformly at random in this region as Centroids for an instant. Let it be called `C_1, C_2`.

```
a1, a2=(-2+5*np.random.random(size=2)).astype(np.float64)

b1, b2=(-2+4*np.random.random(size=2)).astype(np.float64)

C_1= [a1, b1]

C_2= [a2, b2]
```

• **Iteration:**

- ☐ **Defining a dictionary** called mappings to keep note of each note data point and its cluster. We initially define its cluster randomly. Its code:

```
mappings= {}

for data in data_points:

{

    mappings[data]=np.random.choice(range(1,3))

}
```

- ☐ **Initializing** the distinct variable to keep track of data: Its code is:

```
num_points_updated=0

before_number=1

max_iters=10000

current_iters=0
```

- **Algorithm execution:**

  - ☐ **Checking the Euclidean distance** of the `data_points` from each two instant centroids and re-assigning the cluster data in `mappings` corresponding to nearest centroid as given in question.

`np.linalg.norm(data-C_1)`

`linalg` is a function in **`numpy`** which gives our desired Euclidean distance from let say "`C_1`". And here " `data` " means one point in `data _points.`

  - ☐ **Updating the centroids:**

My idea about this I am adding the data of same cluster identity and dividing it with its corresponding length. The corresponding code lies in the function:

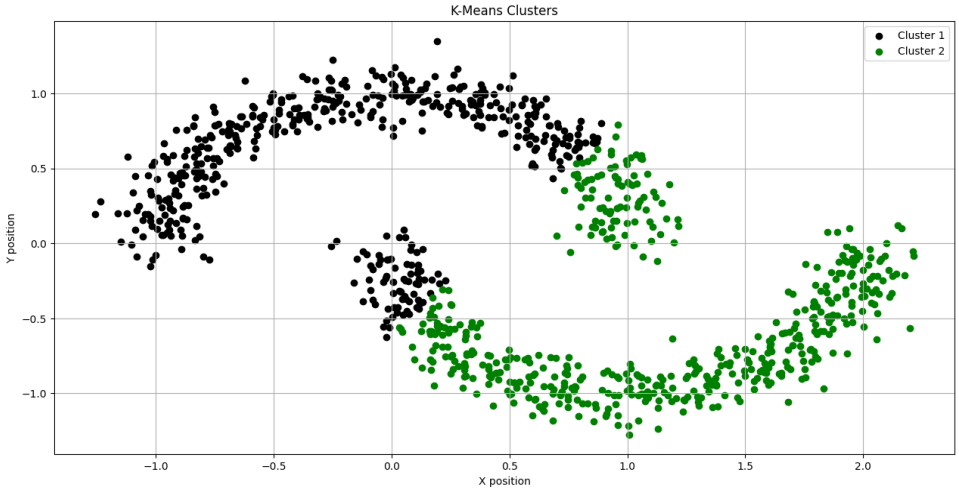
```
def assign_clusters (data_points, C_1, C_2, mappings):
{
''' refer to code[AI1104 Project Q2.py] to see the function '''
}
```

In the general code I wrote a " `while` " loop to always look after a new a Centroids and the condition(based on the given instructions) for loop is:

```
while(max_iters>current_iters and before_number!=num_points_updated)
```

- **Finally plotting the distinguished clusters:**

We are plotting the graphs based on the modified `mappings` dictionary which is modified directly in the defined function `assign_clusters` . We are plotting the graph with the help of `matplotlib`, and the plot is:



This competes the end of **Task 2: K-means Clustering.** ”

By: AI23BTECH11024 Sukesh Kumar

Task 3: Classification using the k-Nearest Neighbors Technique

“ This task in this question is to use the popular k- nearest neighbors' method to learn the classes in a labeled dataset and predict the class of a new data point. In this task, the data was on abalone dataset . Modules required were **`train_test_split`** from **`sklearn.model_selection`**, **`pandas`**,**`random`**, **`numpy`** ,**`statistics`** as **`stats`**, **`pyplot`** from **`matplotlib`**. to load data to program, the following code was used:

```
url = "https://archive.ics.uci.edu/ml/machine-learning-
                          databases/abalone/abalone.data"
abalone = pd.read_csv(url, header=None)
```

The dataframe was given its column names and the column named “ Sex ”was removed from the dataframe since it had no use in calculating the age of **abalone.**
And this was done with the following code:

```
column_names =["Sex","Length","Diameter","Height","Whole
          weight","Shucked weight","Viscera weight",
               "Shell weight","Rings"]
abalone.columns=column_names
abalone=abalone.drop(columns=["Sex"])
```

To train the data, the dataset was split into two parts with testing data of 30% and training data of 70%. A random seed (333) is used to for reproducibility of the same output. The code used for this task is (given in assignment file): used `.values()` method to extract values

```
x=(abalone.drop(columns=['Rings'])).values
y=abalone["Rings"].values
x_train,x_test,y_train,y_test=train_test_split(x,y,test_size=0.3
               ,random_state=333)
```

Using NumPy's `linalg.norm()` method, the Euclidean distance between each row of x_train and that of the new data point (viewed as a row with same column names as x_train, shown below). distances be stored in a variable called distances
For trail, the new data point set was asked to evaluate, with variable values:

Variable	Value
Length	0.569552
Diameter	0.446407
Height	0.154437
Whole weight	1.016849
Shucked weight	0.439051
Viscera weight	0.222526
Shell weight	0.291208

For this task, the code used were:

```
new_data_point=np.array([0.569552,0.446407,0.154437,1.016849,
                         0.439051,0.222526,0.291208])
distances = np.linalg.norm(x_train - new_data_point, axis=1)
```

The values in distances were sorted in increasing order of their values, the indices of the smallest 3 values were collected in `nearest_indices` using the `.argsort()` method. These indices correspond to the indices of the 3 nearest neighbors in the abalone dataset to the new data point. And the code line is:

```
nearest_indices = distances.argsort()[:3]
```

Corresponding to the indices of the 3 nearest neighbors obtained above, extracted the number of rings (ages) from `y_train`. Among the 3 ages thus obtained, the age which appears most frequently (i.e., the "mode") is picked and reported as predicted age. And the following code was used for this task:

```
nearest_ages = y_train[nearest_indices]
age_mode = stats.mode(nearest_ages)
```

Treating each row in `x_test` as a new abalone data point, the above steps were repeated to generate a prediction for the age. The predicted ages were stored in an array called `predicted_ages_test`. The mean squared error is computed and reported with the formula:

$$\text{MSE} = \frac{1}{M}\sum_{j=0}^{M-1}\left(\texttt{predicted\_ages\_test}[j] - \texttt{y\_test}[j]\right)^2$$

where ***M*** is length of `y_test` column. To accomplish this task, the following function was defined and called out whenever needed:

```
def mse_calculator(value, estimator):
     myValue= np.mean((value-estimator) ** 2)
     return myValue
```

Next task: Tuning k to Achieve Optimal Performance
Problem statement: "Vary k from 1 to 50, and compute the MSE value for each k. Store the MSE values in an array named `mse_values`. Generate a plot of k values on the x- axis and `mse_values` on the y- axis. From the plot, report an optimal value of k at which MSE is the smallest. If there are multiple optimal values for k, you may pick any one at random. Also report the MSE corresponding to this optimal value of k "

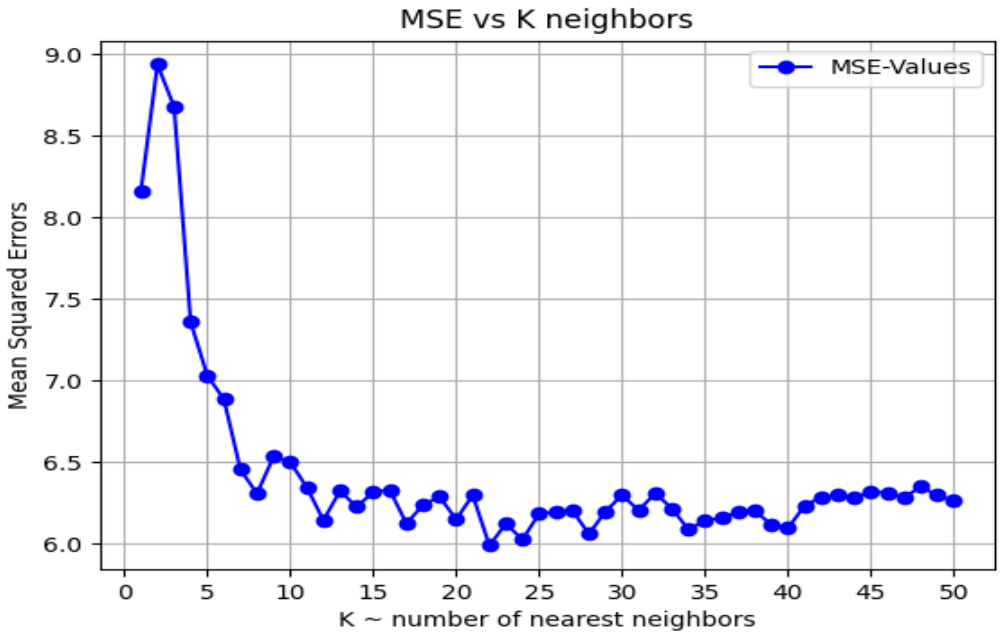
For this task, the loop of finding nearest 3 neighbors is repeated by replacing ' 3 ' with ' k ' and then looping the k values from 1 to 50. After each iteration, mse values and k values were collected in `mse_values` , `k_ values` respectively.
Then to find optimal k, the following code snippet was used:

```
optimal_k = np.argmin(mse_values) + 1
```

`np.argmin(array)` returns the index of minimum value in `array` , hence adding 1 to the value gives the `optimal_k` value.

Then a plot is plotted with `mse_values` on y-axis and `k_values` on x- axis:

A sample graph is as below:



Note: The above graph is a sample , because the same code on multiple runs gives different mode values for predicted ages for Q3.4 part 4, hence the mse values differ. And hence the same in Q3.5 while tuning for `optimal_k` . So the graph , `optimal_k` , mse value @ `optimal_k` differs at each run of the code.

This competes the end of **Task 3 : Classification using the *k*- Nearest Neighbors Technique.** ”

By: AI23BECTH11017 Sai Kaushik