Assignment-7: Implementing Weight learning of an MLP using the Genetic Algorithm

Sajan Kumer Sarker
Id: 2111131642
Email: sajan.sarker@northsouth.edu

Rafsan Jani Chowdhury
Id: 2011424642
Email: rafsan.chowdhury@northsouth.edu

Rosely Mohammad
Id: 2014219642
Email: rosely.mohammad@northsouth.edu

December 7, 2024

1. Introduction

This assignment was focused on implementing weight learning for a Multi-layer Perceptron (MLP) using Genetic Algorithms. The task involves applying this method to a multi-class classification problem using the Car Evaluation [2] dataset. The dataset will be split into training, validation, and testing subsets. The goal is to experiment with the algorithm to achieve the best performance in classifying the data. During the implementation process we've followed some resources [1] [3], which are added in the reference section.

2. Methods

Necessary Imports:

```
import pandas as pd
import numpy as np
import matplotlib.pyplot as plt
%matplotlib inline
import seaborn as sns
import random
import warnings
warnings.filterwarnings('ignore')
from sklearn.model_selection import train_test_split, GridSearchCV
from sklearn.preprocessing import LabelEncoder, StandardScaler
from sklearn.model_selection import train_test_split
from sklearn.metrics import accuracy_score, confusion_matrix, ConfusionMatrixDisplay
```

We have imported all necessary libraries and modules to be used in each notebook.

Dataset Loading:

```
car_evaluation = pd.read_csv('/content/drive/MyDrive/CSE445-Assignment/dataset/car.data')

column_names = ['buying', 'maint', 'doors', 'persons', 'lug_boot', 'safety', 'class']
car_evaluation.columns = column_names

print('Number of missing values: ')
print(car_evaluation.isnull().sum())
```

Import the dataset then assigns column name to the dataset. It then checks for any missing values found or not then print the result.

2.1 Dataset Pre-processing:

Data Encoding:

```
X = car_evaluation.drop(columns='class')
y = car_evaluation['class']

le =LabelEncoder()
scaler = StandardScaler()

X = X.apply(lambda col: le.fit_transform(col))
y = le.fit_transform(y)

X = np.array(X, dtype=float)
y = np.array(y, dtype=int)

X = scaler.fit_transform(X)
```

Here, we separate the features and the target and encode the categorical values into numerical form. It then scales the features to normalize the data for better model performance.

Train, Validation, Test Split:

```
X_train, X_temp, y_train, y_temp = train_test_split(X, y, test_size=0.3, random_state=42)
X_val, X_test, y_val, y_test = train_test_split(X_temp, y_temp, test_size=0.5, random_state=42

# this will used after cross-validation
X_train_val = np.concatenate((X_train, X_val))
y_train_val = np.concatenate((y_train, y_val))

print(f'Train Data Shape (X, y): {X_train.shape, y_train.shape}')
print(f'Validation Data Shape (X, y): {X_val.shape, y_val.shape}')
print(f'Test Data Shape (X, y): {X_test.shape, y_test.shape}')
```

We Splits the dataset into training, validation, and testing sets.

2.2 Multi-Layer Perceptron (MLP):

MLP:

```
class MultiLayerPerceptron:
  def __init__(self, input_layer, hidden_layer, output_layer):
    self.input_layer = input_layer
    self.hidden_layer = hidden_layer
    self.output_layer = output_layer
    self.weights_hidden_input = np.random.randn(input_layer, hidden_layer) * 0.1
    self.weights_hidden_output = np.random.randn(hidden_layer, output_layer) * 0.1
    self.bias_hidden_layer = np.random.randn(hidden_layer) * 0.1
    self.bias_output_layer = np.random.randn(output_layer) * 0.1

  def forward(self, X):
    self.hidden_input = np.dot(X, self.weights_hidden_input) + self.bias_hidden_layer
    self.hidden_output = self.relu(self.hidden_input)
    self.output_input = np.dot(self.hidden_output, self.weights_hidden_output) + self.bias_outp
    self.output_output = self.softmax(self.output_input)
    return self.output_output

  def sigmoid(self, X):
    return 1/(1+np.exp(-X))

  def relu(self, X):
    return np.maximum(0, X)

  def softmax(self, X):
    return np.exp(X - np.max(X, axis=1, keepdims=True)) / np.sum(np.exp(X - np.max(X, axis=1,

  def predict(self, X):
    return np.argmax(self.forward(X), axis=1)

  def set_weights(self, weights):

    self.weights_hidden_input = weights[:self.input_layer * self.hidden_layer].reshape((self.i
    self.weights_hidden_output = weights[self.input_layer * self.hidden_layer:self.input_layer

    self.bias_hidden_layer = weights[self.input_layer * self.hidden_layer + self.hidden_layer
    self.bias_output_layer = weights[self.input_layer * self.hidden_layer + self.hidden_layer

  def get_weights(self):
    return np.concatenate((self.weights_hidden_input.flatten(), self.weights_hidden_output.fla
```

Here we defines a MultiLayerPerceptron (MLP) class with essential functionality for a neural network. Firstly, we setup the network architecture with specified input, hidden, and output layers. Then initializes weights and biases with small random values for the connections between layers.

In the Forward function we perform the forward propagation through the network using the ReLu activation for the hidden layer and the softmax function fo the output layer to handle multi-class classification.

The sigmoid function will maps input to the range (0,1) for probabilities, Then ReLu function will introduces non-linearity by outputting positive values or zero. Then the softmax function will converts input into a probability distribution across multiple classes.

The prediction function will output the predicted class label based on the highest probability from the forward pass. It will take and input, then performs the forward operation through the network using the forward function to obtain the output.

The set weights function will updates the weights and biases of the neural network using a array. It reshapes portions of the array to match the dimension of the input to hidden weights, hidden to output weights, and their corresponding biases.

Lastly, the get weight function will retrieves all the weights and biases of the neural network by flattening them into a single concatenated array. This allows the network's parameters to be easily accessed for optimization purposes.

2.3 Genetic Algorithm Functions:

Fitness Evaluation:

```
def fitness_function(weights, mlp, X, y):
  mlp.set_weights(weights)
  predictions = mlp.predict(X)
  accuracy = np.mean(predictions == y)
  return accuracy
```

This function will assigns the provided weights to the MLP and evaluates its performance by predicting labels for data and comparing them with the true values.

Genetic Crossover:

```
def crossover(parent1, parent2):
  crossover_point = random.randint(0, len(parent1))
  return np.concatenate([parent1[:crossover_point], parent2[crossover_point:]])
```

It will creates a new offspring by combining parts of two parent solution at a randomly chosen crossover points. It will concatenates the first segments of first parents with the second segment of the second parents to form the offspring.

Mutate Function:

```
def mutate(individual, mutation_rate):
  for i in range(len(individual)):
    if random.random() < mutation_rate:
      individual[i] += np.random.normal(scale=0.1)
  return individual
```

This function will alters an individual by randomly modifying its values based on a given mutation rate. If a randomly generated value is less than the mutation rate, a small random change is added to the individual's value.

Genetic Algorithm:

```
def genetic_algorithm(pop_size, mlp, X, y, generations, mutation_rate):
  population = [mlp.get_weights() + np.random.uniform(-1, 1, mlp.get_weights().size) for _ in
  best_fitness_per_generation = []

  for generation in range(generations):
    fitness_scores = [fitness_function(individual, mlp, X, y) for individual in population]

    best_fitness = max(fitness_scores)  # Directly get the best fitness (accuracy)
    best_fitness_per_generation.append(best_fitness)

    mating_pool = [population[i] for i in np.argsort(fitness_scores)[-pop_size//2:]]

    next_generation = []
    for _ in range(pop_size):
      parent1, parent2 = random.choice(mating_pool), random.choice(mating_pool)
      child = crossover(parent1, parent2)
```

```
      child = mutate(child, mutation_rate)
      next_generation.append(child)
    population = next_generation

best_individual = population[np.argmax(fitness_scores)]
return best_individual, best_fitness_per_generation
```

This function will optimized the weights of an MLP using a genetic algorithm. It starts by creating a population of random individuals, each with slightly perturbed weights. In each generation, the fitness of each individual is evaluated, and the top half are selected to form a mating pool. Crossover and mutation are applied to generate a new population, and the algorithm repeats for a set number of generations.

2.4 Running the Genetic Algorithm Functions:

Initialize and Run Genetic Algorithm:

```
m, n = X_train.shape
mlp = MultiLayerPerceptron(n, 15, 4)  # initialize mlp

# run genetic algo
population = 500
generations = 1000
rate = 0.01
w, fitness = genetic_algorithm(population, mlp, X_train, y_train, generations, rate)
```

Here we initialize the MLP and runs the genetic algorithm for 500 population, 1000 generations and 0.01 mutation rate to optimize the MLP's weights.

Fitness vs Generations Graph:

```
plt.plot(range(generations), fitness, label='Best Fitness Score')
plt.title('Best Fitness Score Over Generations')
plt.xlabel('Generation')
plt.ylabel('Fitness Score')
plt.legend()
plt.show()
```

Plots the fitness score from the best fitness from each generations against the number of generations.

Validation Set Evaluation:

```
y_pred_val = mlp.predict(X_val)   # Predict on the val set
print(f"Validation set accuracy: {accuracy_score(y_val, y_pred_val)}")
print()
cm = confusion_matrix(y_val, y_pred_val)
cmv = ConfusionMatrixDisplay(cm)
cmv.plot()
plt.title("Confusion Matrix for Validation  Set")
```

We check the accuracy of the implemented algorithm using the validation dataset and plot the confusion matrix for the validation set.

Test Set Evaluation:

```
y_pred_test = mlp.predict(X_test)   # predicted on the test set
print(f"Test set accuracy: {accuracy_score(y_test, y_pred_test)}")
print()
cm = confusion_matrix(y_test, y_pred_test)
cmv = ConfusionMatrixDisplay(cm)
cmv.plot()
plt.title("Confusion Matrix for Test Set")
```

We again check the accuracy of the implemented algorithm using the test dataset and plot the confusion matrix for the test set.

3. Experiment results

3.1 Comparison With Differents Hidden Layer Size:

Fitness Curve:

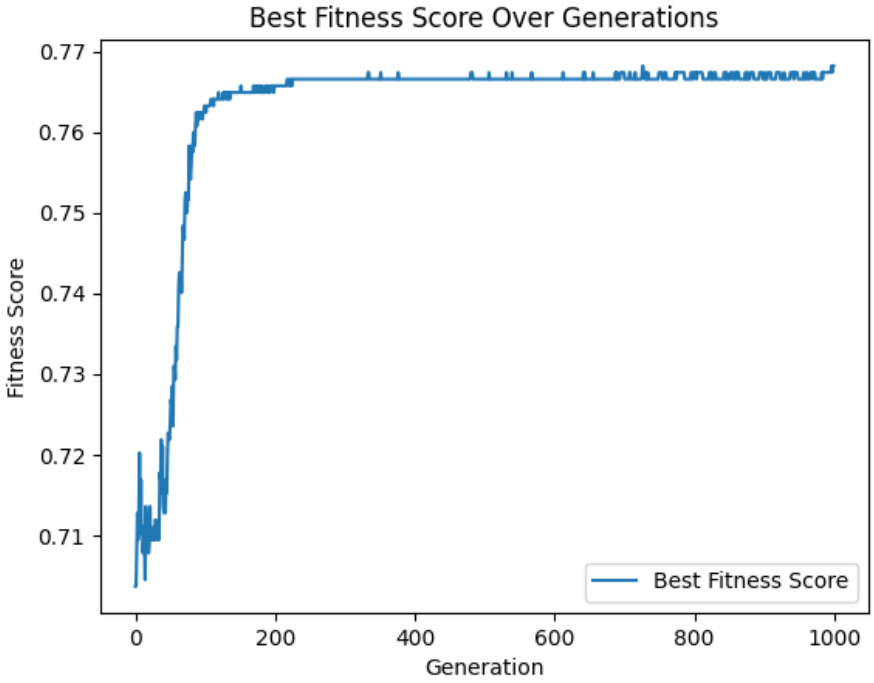

Fig. 1: Fitness Curve for 10 Hidden Layer, 3 Output Layer

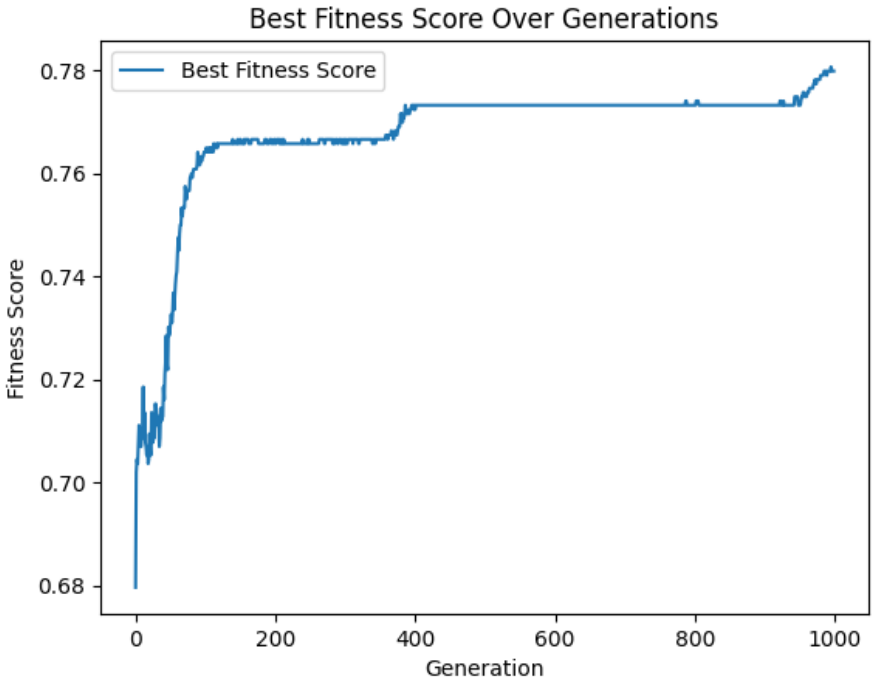

Fig. 2: Fitness Curve for 12 Hidden Layer, 4 Output Layer

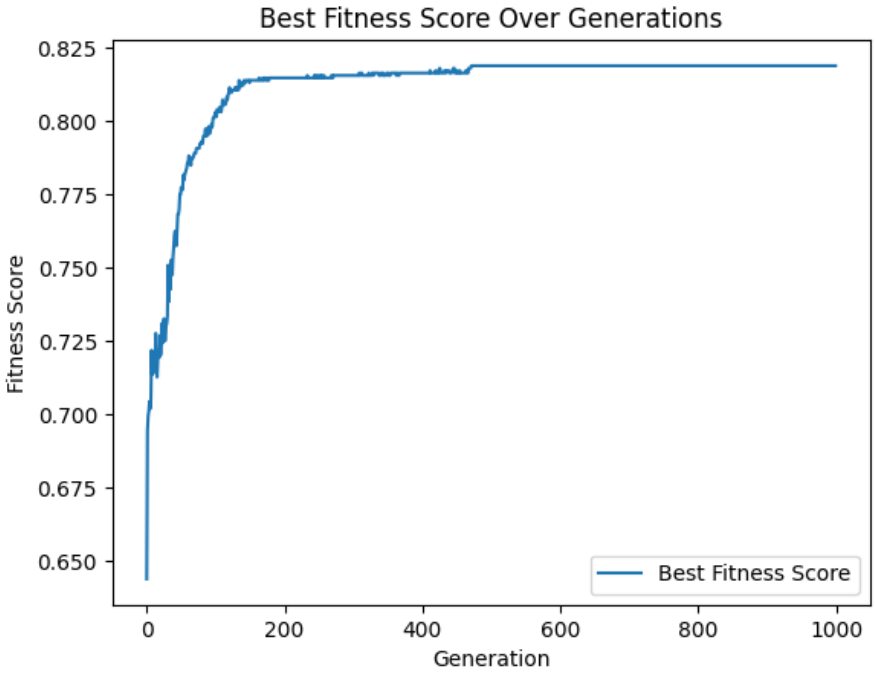

Fig. 3: Fitness Curve for 15 Hidden Layer, 4 Output Layer

We've test our algorithm for 3 time for different combination of layers. Fig. 1 illustrate the best fitness scores on fitness score vs generation for 10 hidden layers and 3 output layers. Here the fitness score increases rapidly, stabilizing around a score of 0.765 after 200 generations. there were no improvement for the later generations. In Fig. 2 where the hidden layer size is 12 and 4 output layer it stabilize the fitness score over 0.77 after 400 generations. Which is a little bit better then the previous one. Then we test it again for 15 hidden layers and 4 output layers. It shows better results then the previous two results. It stabalize the fitness score over 0.820 after generation 500.

3.2 **Validation Set Accuracy VS Test Set Accuracy:**

10 Hidden Layer and 3 Output Layer:

Validation	**Test**
73.74%	73.84%

12 Hidden Layer and 4 Output Layer:

Validation	**Test**
77.60%	75.76%

15 Hidden Layer and 4 Output Layer:

Validation	**Test**
81.85%	81.53%

3.3 **Confusion Matrix of Validation Vs Test Set:**

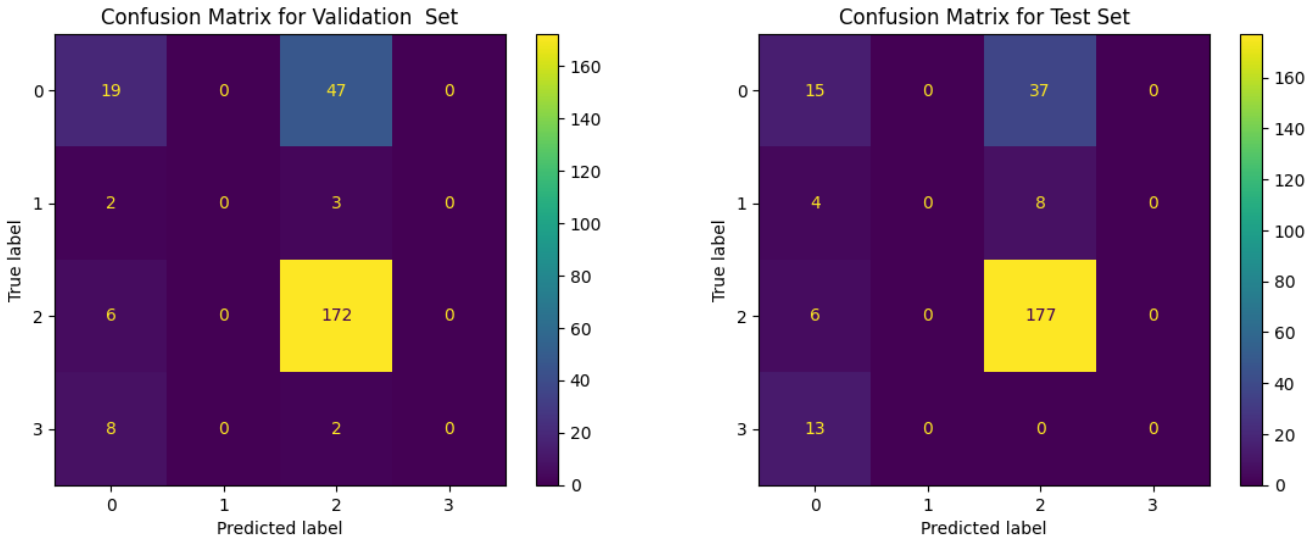

Fig. 4: Confusion Matrix of Hidden Layer 10 and Output Layer 3

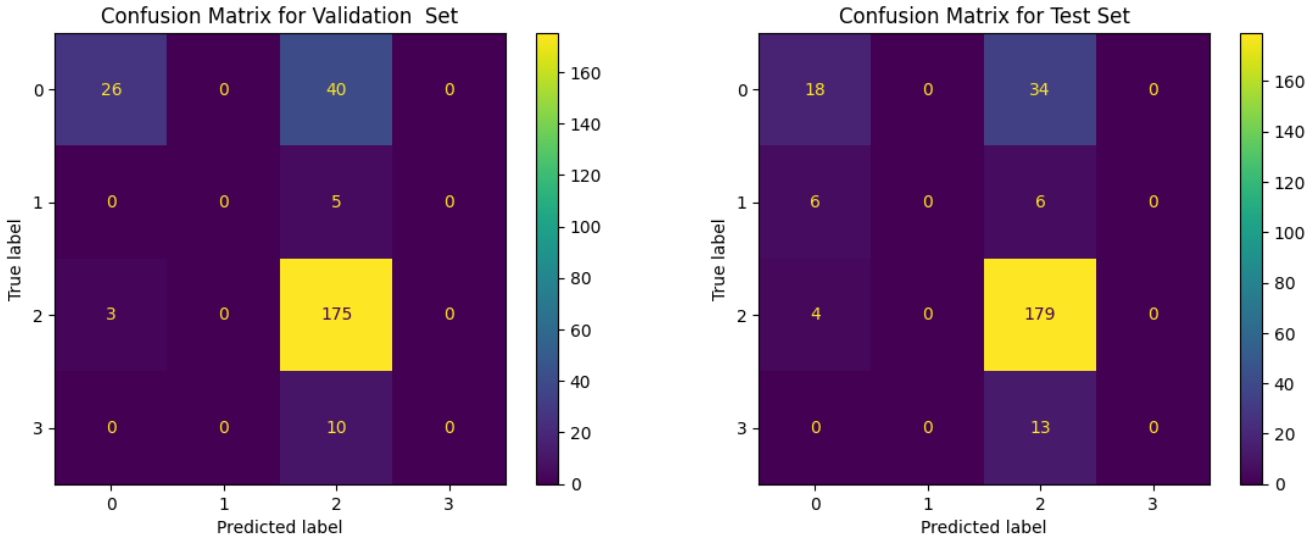

Fig. 5: Confusion Matrix of Hidden Layer 12 and Output Layer 4

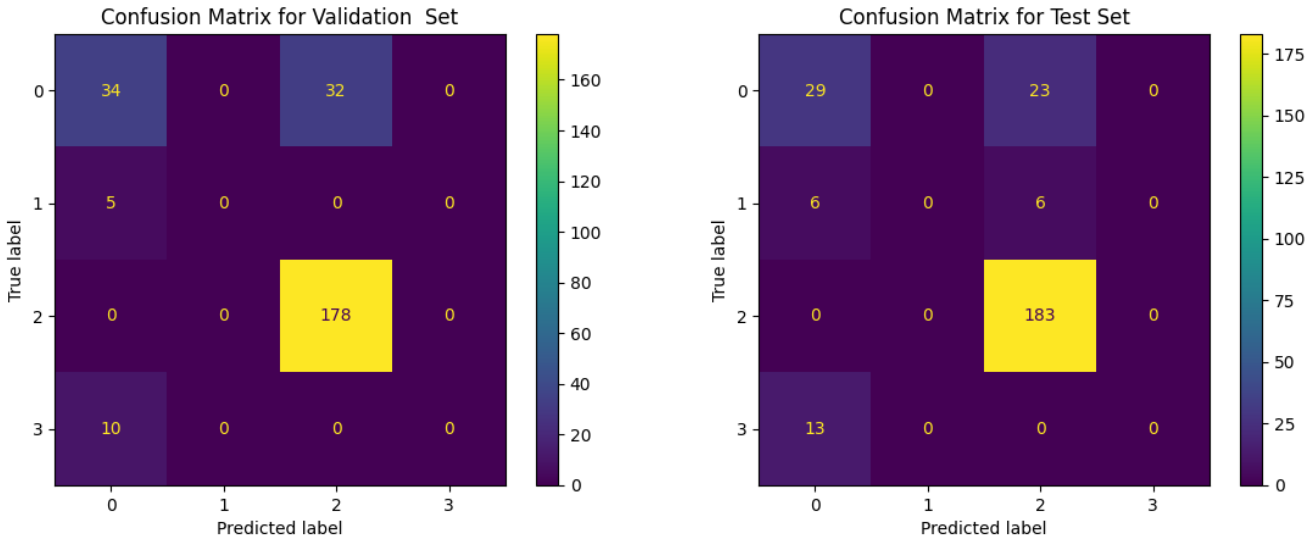

Fig. 6: Confusion Matrix of Hidden Layer 15 and Output Layer 4

4. Discussion

In this assignment we implemented the weight learning for a Multi-layer Perceptron (MLP) using Genetic Algorithms and test it on the Car Evaluation Dataset [2]. Here we test the algorithm for 3 different combinations of hidden layer and output layers. We find the best output while we use 15 hidden layer and 4 output layers. Where the best fitness score was over 0.82 and the accuracy of validation dataset was 81.85% and the accuracy of test dataset was 81.53% which is the best accuracy over 3 combinations results. Here we notice that, while we increase the hidden layer size, the model was perform more better. Which means the model perform more better with more layers. We've follow some online resources during building this algorithm [1] [3].

References

[1] M. Elcaiseri. Building a multi-layer perceptron from scratch with numpy. https://elcaiseri.medium.com/building-a-multi-layer-perceptron-from-scratch-with-numpy-e4cee82ab06d, 2020.
[2] U. M. L. Repository. Car evaluation data set. https://archive.ics.uci.edu/dataset/360/air+quality, 1997.
[3] M. Sena. A step-by-step guide to implementing multi-layer perceptrons in python. https://python.plainenglish.io/building-the-foundations-a-step-by-step-guide-to-implementing-multi-layer-perceptrons-in-python-51ebd9d7ecbe, 2020.