Assignment-7: Genetic Algorithm

Mahbub Ahmed Turza
ID: 2211063042
North South University
mahbub.turza@northsouth.edu

December 7, 2024

Abstract

The implementation of a Multi-Layer Perceptron (MLP) for weight training in the context of multi-class classification using a Genetic Algorithm (GA) is explained in this research. The **Car Evaluation** dataset from the UCI Machine Learning Repository was used for this study. The goal is to forecast the evaluation of a car's acceptance based on a number of input features, such safety, number of doors, maintenance costs, and purchase price. 70% of the dataset is applied for training, 15% is used for validation, and 15% is used for testing. By replicating the stages of crossover, mutation, and selection that take place in a natural evolutionary process, the Genetic Algorithm is applied to optimize the weights of the MLP. The training stage is used to evaluate the algorithm's performance, and hyperparameter tuning is done to figure out the best genetic search configuration. Model performance is evaluated utilizing key measures like accuracy on testing, validation, and training sets. The GA-based weight learning procedure is fully clarified in the report, along with the Python code implementation utilizing NumPy, Pandas, and Scikit-learn. It additionally provides the outcomes in terms of accuracy and model behavior and talks about the experimental tactics, including methods of trial and error for adjusting hyperparameters. The paper additionally highlights the challenges faced during setting up and strategies used to improve the model's performance.

I. Introduction

The application of evolutionary algorithm—in particular, Genetic Algorithms, or GAs—for machine learning model optimization has attracted a lot of attention recently. A collection of optimization techniques known as genetic algorithms are inspired by the concepts of genetics and natural selection. These algorithms evolve solutions to challenging issues across many generations using processes such as crossover, mutation, and selection. Training machine learning models—more particularly optimizing neural network weights—is one of the interesting uses of GAs.

The implementation of a Multi-Layer Perceptron (MLP) with weight learning based on Genetic Algorithms to deal with a multi-class classification problem is the primary focus of this study. The Car Evaluation dataset from the UCI Machine Learning Repository was utilized in this experiment. Predicting an automobile's acceptability based on a number of categorical factors, including purchase price, maintenance costs, safety, and more, was the problem at issue. The goal of this project is to look into the possibilities for the Genetic Algorithm for optimizing the MLP's weights and evaluate its effectiveness when compared to more traditional methods of optimization such as backpropagation.

Three sections of the Car Evaluation dataset are utilized: 15% for testing, 15% for validation, and 70% for training. With the goal to achieve a low error rate on the validation set, a GA is used to evolve the weights of the MLP over a number of generations. A unique perspective on neural network training is offered by the program of a GA, which offers an alternative to gradient-based techniques, particularly for complex or non-differentiable objective functions.

The GA-based weight optimization method, data prior to treatment procedures, and code implementation for MLP training are all discussed in detail in this study. The challenges experienced during the experimental phase and the strategies used to achieve the best outcomes are also covered. With this work, we hope to add to the increasing corpus of research on evolutionary machine learning addresses and show how effectively Genetic Algorithms enhance MLPs.

II. Methodology

This project implements weight learning of a Multilayer Perceptron (MLP) for multiclass classification on the Car Evaluation dataset using a Genetic Algorithm (GA). The following steps describe the methodology:

A. Dataset Preprocessing

The Car Evaluation dataset was preprocessed to make it suitable for training.

- **Categorical Encoding:** Categories in features such as `buying`, `maint`, and `doors` were converted to numeric values using ordinal encoding. This step ensures numerical representation for categorical features:

```
df['doors'] = df['doors'].replace(['5more'], '5')
df['persons'] = df['persons'].replace(['more'], '4')
for col in ['buying', 'maint', 'doors', 'persons', 'lug_boot', 'safety']:
    df[col] = df[col].astype('category').cat.codes

```

Listing 1. Coding categorical features

- **Feature Selection:** To avoid redundancy, one feature from each one-hot encoded group was dropped:

```
df = df.drop(['buying_3', 'maint_3', 'doors_3',
              'persons_1', 'lug_boot_2', 'safety_2'], axis=1)

```

Listing 2. Removing redundant features

- **Data Splitting:** The dataset was divided into training (70%), validation (15%), and testing (15%) subsets to evaluate model performance during and after training:

```
train_size = int(0.7 * len(x))
val_size = int(0.15 * len(x))
test_size = len(x) - train_size - val_size

x_train, y_train = x[:train_size], y[:train_size]
x_val, y_val = x[train_size:train_size + val_size],
               y[train_size:train_size + val_size]
x_test, y_test = x[train_size + val_size:], y[train_size + val_size:]

```

Listing 3. Splitting the dataset

B. Multilayer Perceptron (MLP) Architecture

An MLP was chosen for its capability to model complex patterns in data. The architecture was designed with:

- **Hidden Layer:** Applied a `tanh` activation function to introduce non-linearity and handle complex decision boundaries:

```
hidden_output = np.tanh(np.dot(X_train, individual['weights_hidden']) +
                        individual['bias_hidden'])

```

Listing 4. Hidden layer forward pass

- **Output Layer:** Used softmax activation to ensure the output represents class probabilities:

```
output = np.dot(hidden_output, individual['weights_output']) +
         individual['bias_output']
softmax_output = np.exp(output) / np.sum(np.exp(output), axis=1, keepdims=True)

```

Listing 5. Softmax computation

C. Genetic Algorithm for Weight Optimization

The Genetic Algorithm was employed to optimize weights and biases efficiently without relying on gradient information. The steps include:

1) Population Initialization: Each individual represents an MLP with random weights and biases:

```
def initialize_population(self, input_size, hidden_size, output_size):
    return [
        {
            "weights_hidden": np.random.randn(input_size, hidden_size),
            "bias_hidden": np.random.randn(hidden_size),
            "weights_output": np.random.randn(hidden_size, output_size),
            "bias_output": np.random.randn(output_size)
        }
        for _ in range(self.population_size)
    ]
```

Listing 6. Initializing population

2) Fitness Function: Validation accuracy, adjusted for overfitting, was used to evaluate individuals:

```
def fitness_function(ind, X_train, y_train, X_val, y_val):
    # Forward pass
    hidden_output_train = np.tanh(np.dot(X_train, ind['weights_hidden']) +
                                  ind['bias_hidden'])
    output_train = np.dot(hidden_output_train, ind['weights_output']) +
                   ind['bias_output']
    train_acc = accuracy_score(y_train, np.argmax(output_train, axis=1))

    # Forward pass validation data
    hidden_output_val = np.tanh(np.dot(X_val, ind['weights_hidden']) +
                                ind['bias_hidden'])
    output_val = np.dot(hidden_output_val, ind['weights_output']) +
                 ind['bias_output']
    val_acc = accuracy_score(y_val, np.argmax(output_val, axis=1))

    # Penalize overfitting
    return val_acc - 0.5 * abs(train_acc - val_acc)
```

Listing 7. Fitness function definition

3) Genetic Operations:

- **Crossover:** Uniform crossover combined two parents' weights and biases:

```
def crossover(self, parent1, parent2):
    child = {}
    for key in parent1.keys():
        mask = np.random.rand(*parent1[key].shape) > 0.5
        child[key] = np.where(mask, parent1[key], parent2[key])
    return child

```

Listing 8. Crossover implementation

- **Mutation:** Introduced random noise to diversify the population:

```
def mutate(self, individual):
    if random.random() < self.mutation_rate:
        idx = random.choice(list(individual.keys()))
        individual[idx] += np.random.normal(0, 0.1, individual[idx].shape)

```

Listing 9. Mutation step

4) Evolution Process: The population evolved over generations by applying fitness evaluation, selection, crossover, and mutation:

```
for gen in range(self.num_generations):
    fitness_scores = [self.fitness_function(ind, X_train, y_train,
                                            X_val, y_val) for ind in population]
```

Listing 10. Evolution loop

D. Hyperparameter Optimization

A grid search over parameters such as population size, mutation rate, and hidden layer size was conducted to identify the optimal configuration. Train, validation, and test accuracies were recorded for each combination.

E. Results and Visualization

Accuracy trends across hyperparameter combinations and cost evolution over generations were visualized:

```
plt.plot(best_cost_per_gen, label="Best Cost", color="blue")
plt.plot(avg_cost_per_gen, label="Average Cost", color="green")
plt.legend()
plt.show()
```

Listing 11. Plotting results

The best model was selected based on test accuracy and documented for further analysis.

III. Experiment Results

In this experiment, we improved the weights of a Multi-Layer Perceptron (MLP) model for a classification problem using a Genetic Algorithm (GA). Following is a list of all the various hyperparameter combinations that were evaluated throughout the optimization process:

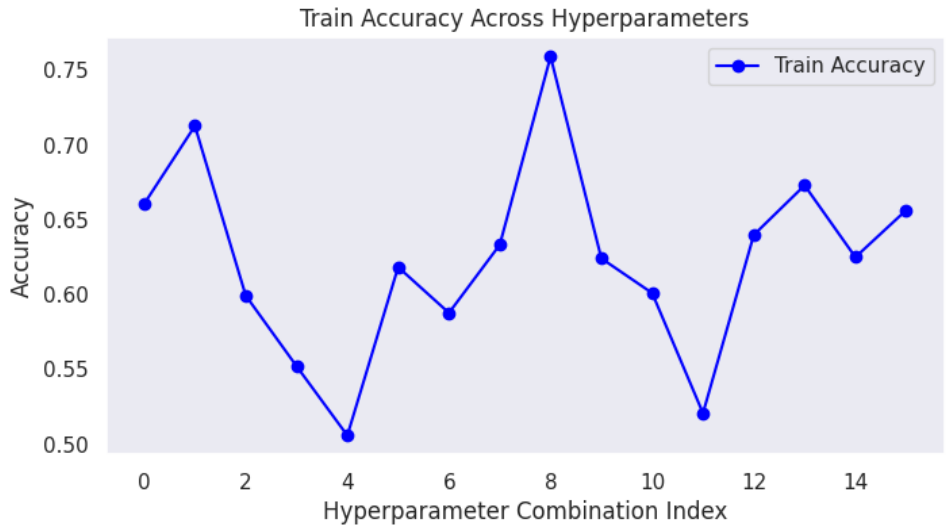

Fig. 1. Training

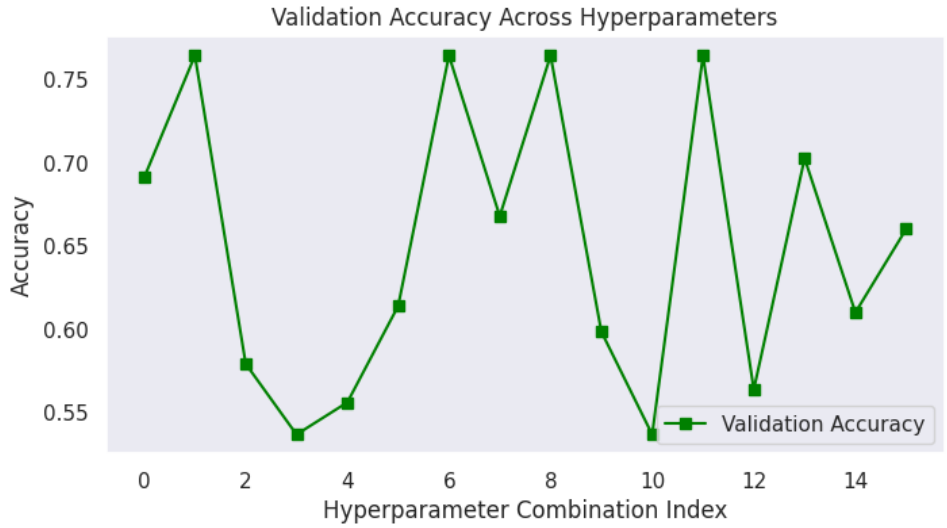

Fig. 2. Validation

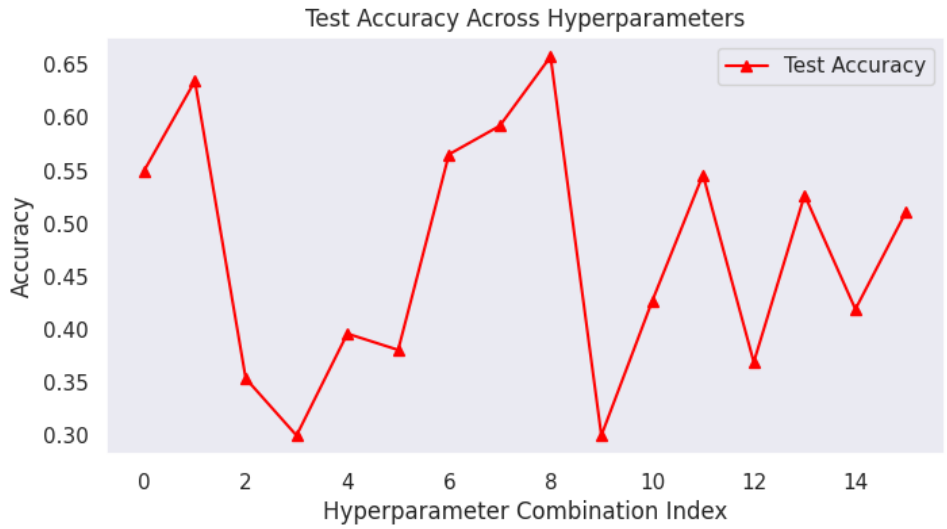

Fig. 3. Testing

- **Population Size:** 50
- **Mutation Rate:** 0.5
- **Crossover Rate:** 0.7
- **Hidden Layer Size:** 64
- **Number of Generations:** 150

For the purpose of to optimize the model's classification accuracy, the network weights have been refined over multiple generations utilizing the GA. The review of the training, validation, and test sets provided these final results for the best model:

- **Best Training Accuracy:** 75.91%
- **Best Validation Accuracy:** 76.45%
- **Best Test Accuracy:** 65.77%

With the highest validation accuracy slightly surpassing the training accuracy, these results demonstrate that the model performed rather well in both the training and validation sets. On the contrary hand, the test set performance was more severe, which may indicate that the training data was overfit or that more hyperparameter tuning needs to be done.

IV. Discussion

The results of the experiment indicate that the Multi-Layer Perceptron (MLP) model's weights were effectively improved for classification utilizing the Genetic Algorithm (GA). The test accuracy (65.77%) indicates that there may be a difference between the model's performance on the training/validation data and its generalization to unseen data, even though the best training and validation accuracies achieved were relatively good (75.91% and 76.45%, respectively).

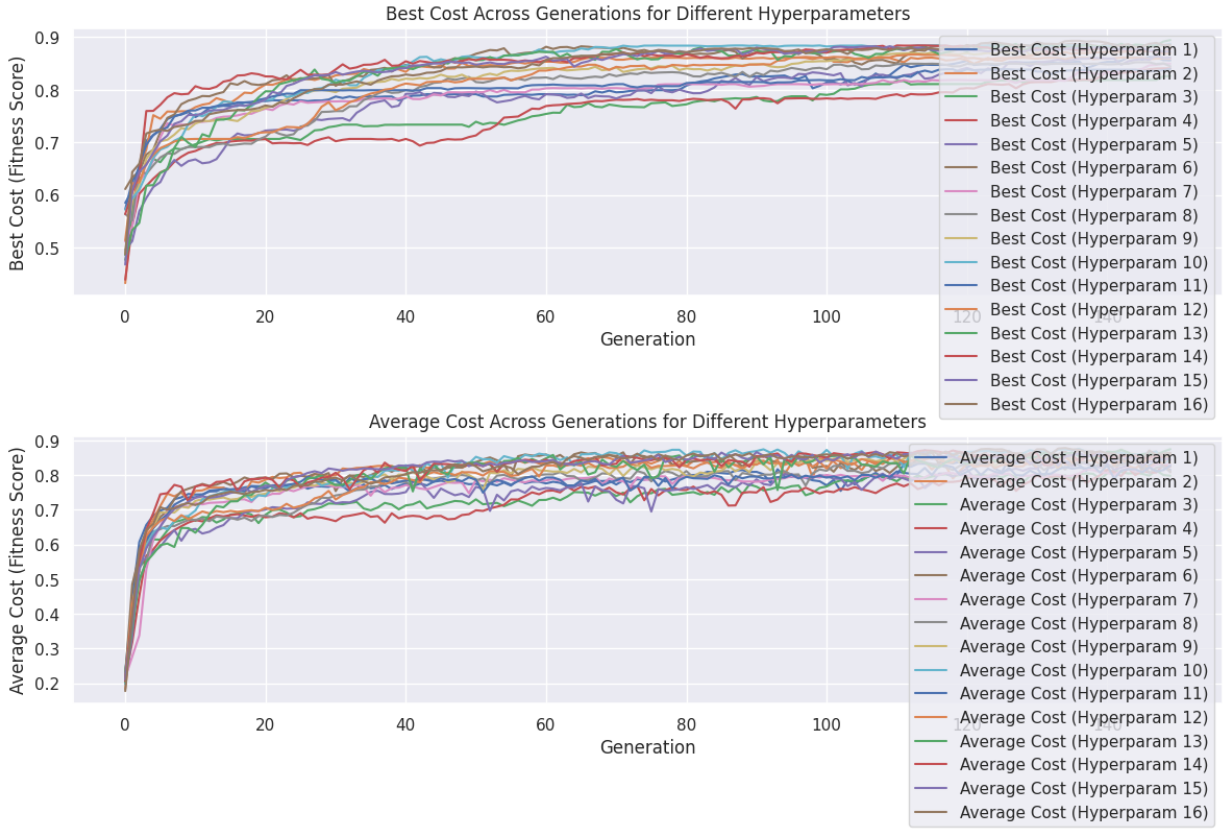

Fig. 4. Gradient

Several factors could explain this performance gap:

- **Overfitting:** On either the training and validation sets, the model's high accuracy can frequently be an indication of overfitting. The model might have fit the training data too closely as a result of the optimization method, which could have rendered it less applicable to the test data. Applying regularization techniques, modifying the model architecture, or attempting out additional instructional methods may assist with this.
- **Hyperparameter Selection:** While a grid search was applied to tune the hyperparameters used in this experiment (population size, mutation rate, crossover rate, hidden layer size, and number of generations), further tuning could enhance performance. Deep learning model performance still heavily depends on hyperparameter modification; advanced techniques like Bayesian optimization or an expanded search space can yield better results.
- **Model Complexity:** Having a hidden layer size of 64, the MLP architecture utilized in this experiment might be both too basic or overly complex for the job at hand. The ability of the model to identify appropriate patterns in the data may be enhanced by further experimentation with different patterns and layer sizes.
- **Training Data:** Another important factors involve the training dataset's size and quality. It's probable that the experiment's dataset was too small and unorganized to enable the model to generalize effectively. The issue

could potentially be addressed by the use of a more representative dataset, pre-processing, or additional data augmentation.

- **Algorithm Limitations:** The GA could not have been the most efficient method of optimization for training the MLP, even though it was able to determine appropriate weight configurations. Because they take a more direct approach to minimizing the loss function, additional optimization methods like gradient descent or Adam might work well for training deep learning models.

In order to sum up, the experiment successfully demonstrated how to maximize an MLP model utilizing a genetic algorithm; however, more improvements are required to improve the model's ability for generalization. For the purpose of to validate the findings, future research will look at other optimization strategies, test the model on different data sets, and fine-tune hyperparameters

V. Full Code With Comment

```
from google.colab import drive
drive.mount('/content/drive')

import numpy as np
import pandas as pd
import seaborn as sns
import matplotlib.pyplot as plt
from sklearn.model_selection import train_test_split
from sklearn.metrics import precision_recall_curve, average_precision_score
from sklearn.preprocessing import label_binarize
import random
from sklearn.metrics import accuracy_score
import sklearn
import warnings
%matplotlib inline
sns.set()
warnings.filterwarnings('ignore')

df = pd.read_csv("/content/drive/MyDrive/Colab Notebooks/car.data")

df.head()

df.columns = ["buying", "maint", "doors", "persons", "lug_boot", "safety", "class values"]

df['doors'] = df['doors'].replace(['5more'], '5')
df['persons'] = df['persons'].replace(['more'], '4')

for col in ['buying', 'maint', 'doors', 'persons', 'lug_boot', 'safety']:
    df[col] = df[col].astype('category')

for col in ['buying', 'maint', 'doors', 'persons', 'lug_boot', 'safety']:
    df[col] = df[col].cat.codes

df = pd.get_dummies(df, columns=['buying', 'maint', 'doors', 'persons', 'lug_boot', 'safety'])
df.head()

df['class values'].unique()

df = df.drop(['buying_3', 'maint_3', 'doors_3', 'persons_1', 'lug_boot_2', 'safety_2'], axis=1)

from sklearn.preprocessing import LabelEncoder
labelencoder = LabelEncoder()
df['class values'] = labelencoder.fit_transform(df['class values'])

x = df.drop(["class values"], axis=1)
y = df["class values"]

x.head()

y.head()

y.unique()

train_size = int(0.7 * len(x))
val_size = int(0.15 * len(x))
test_size = len(x) - train_size - val_size

x_train, y_train = x[:train_size], y[:train_size]
x_val, y_val = x[train_size:train_size + val_size], y[train_size:train_size + val_size]
x_test, y_test = x[train_size + val_size:], y[train_size + val_size:]

```

```python
def backward_pass(individual, X_train, y_train, learning_rate=0.01):

    hidden_output = np.tanh(np.dot(X_train, individual['weights_hidden']) + individual['
    bias_hidden'])
    output = np.dot(hidden_output, individual['weights_output']) + individual['bias_output']

    softmax_output = np.exp(output) / np.sum(np.exp(output), axis=1, keepdims=True)
    y_one_hot = np.eye(softmax_output.shape[1])[y_train]
    loss = -np.sum(y_one_hot * np.log(softmax_output)) / X_train.shape[0]

    output_error = (softmax_output - y_one_hot) / X_train.shape[0]
    grad_weights_output = np.dot(hidden_output.T, output_error)
    grad_bias_output = np.sum(output_error, axis=0)

    hidden_error = np.dot(output_error, individual['weights_output'].T) * (1 - hidden_output
    **2)
    grad_weights_hidden = np.dot(X_train.T, hidden_error)
    grad_bias_hidden = np.sum(hidden_error, axis=0)

    individual['weights_output'] -= learning_rate * grad_weights_output
    individual['bias_output'] -= learning_rate * grad_bias_output
    individual['weights_hidden'] -= learning_rate * grad_weights_hidden
    individual['bias_hidden'] -= learning_rate * grad_bias_hidden

    return loss


class GeneticAlgorithm:
    def __init__(self, population_size, mutation_rate, crossover_rate, num_generations,
    fitness_function):
        self.population_size = population_size
        self.mutation_rate = mutation_rate
        self.crossover_rate = crossover_rate
        self.num_generations = num_generations
        self.fitness_function = fitness_function
        self.best_cost_per_generation = []
        self.average_cost_per_generation = []

    def initialize_population(self, input_size, hidden_size, output_size):
        return [
            {
                "weights_hidden": np.random.randn(input_size, hidden_size),
                "bias_hidden": np.random.randn(hidden_size),
                "weights_output": np.random.randn(hidden_size, output_size),
                "bias_output": np.random.randn(output_size)
            }
            for _ in range(self.population_size)
        ]

    def mutate(self, individual):
        if random.random() < self.mutation_rate:
            idx = random.choice(list(individual.keys()))
            individual[idx] += np.random.normal(0, 0.1, individual[idx].shape)

    def crossover(self, parent1, parent2):
        if random.random() < self.crossover_rate:
            child = {}
            for key in parent1.keys():
                mask = np.random.rand(*parent1[key].shape) > 0.5
                child[key] = np.where(mask, parent1[key], parent2[key])
            return child
        return parent1 if random.random() > 0.5 else parent2

```

```
    def select_parents(self, population, fitness_scores):
        indices = np.argsort(fitness_scores)[-2:]
        return population[indices[0]], population[indices[1]]

    def optimize(self, X_train, y_train, X_val, y_val, input_size, hidden_size, output_size):
        population = self.initialize_population(input_size, hidden_size, output_size)
        best_individual = None
        best_accuracy = 0

        for gen in range(self.num_generations):
            fitness_scores = [
                self.fitness_function(ind, X_train, y_train, X_val, y_val)
                for ind in population
            ]
            best_idx = np.argmax(fitness_scores)
            if fitness_scores[best_idx] > best_accuracy:
                best_accuracy = fitness_scores[best_idx]
                best_individual = population[best_idx]

            self.best_cost_per_generation.append(fitness_scores[best_idx])
            self.average_cost_per_generation.append(np.mean(fitness_scores))

            new_population = []
            for _ in range(self.population_size):
                parent1, parent2 = self.select_parents(population, fitness_scores)
                child = self.crossover(parent1, parent2)
                self.mutate(child)
                # Add backward pass to refine individuals
                backward_pass(child, X_train, y_train, learning_rate=0.01)
                new_population.append(child)

            population = new_population
            print(f"Generation {gen+1}, Best Validation Accuracy: {best_accuracy:.4f}")

        return best_individual


    def plot_cost(self):
        plt.figure(figsize=(10, 6))
        plt.plot(self.best_cost_per_generation, label="Best Cost", color="blue", marker="o")
        plt.plot(self.average_cost_per_generation, label="Average Cost", color="green", marker=
    "s")
        plt.title("Cost Across Generations")
        plt.xlabel("Generation")
        plt.ylabel("Cost (Fitness Score)")
        plt.legend()
        plt.grid()
        plt.show()


def fitness_function(ind, X_train, y_train, X_val, y_val):

    hidden_output_train = np.tanh(np.dot(X_train, ind['weights_hidden']) + ind['bias_hidden'])
    output_train = np.dot(hidden_output_train, ind['weights_output']) + ind['bias_output']
    train_acc = accuracy_score(y_train, np.argmax(output_train, axis=1))

    hidden_output_val = np.tanh(np.dot(X_val, ind['weights_hidden']) + ind['bias_hidden'])
    output_val = np.dot(hidden_output_val, ind['weights_output']) + ind['bias_output']
    val_acc = accuracy_score(y_val, np.argmax(output_val, axis=1))

    return val_acc - 0.5 * abs(train_acc - val_acc)

from itertools import product
```

```

hyperparameter_grid = {
    "population_size": [40, 50],
    "mutation_rate": [0.5, 0.9],
    "crossover_rate": [0.7,0.9],
    "hidden_size": [64, 128],
    "num_generations": [200]
}

def evaluate_hyperparameters(params, X_train, y_train, X_val, y_val, input_size, output_size):
    ga = GeneticAlgorithm(
        population_size=params["population_size"],
        mutation_rate=params["mutation_rate"],
        crossover_rate=params["crossover_rate"],
        num_generations=params["num_generations"],
        fitness_function=fitness_function
    )
    best_model = ga.optimize(
        X_train, y_train,
        X_val, y_val,
        input_size=input_size,
        hidden_size=params["hidden_size"],
        output_size=output_size
    )
    return best_model, ga.best_cost_per_generation, ga.average_cost_per_generation

train_accuracies = []
val_accuracies = []
test_accuracies = []
hyperparams_list = []
best_model_overall = None
best_hyperparams = None
best_test_accuracy = 0
best_train_accuracy = 0
best_val_accuracy = 0

all_best_costs = []
all_average_costs = []

for param_combination in product(*hyperparameter_grid.values()):
    params = dict(zip(hyperparameter_grid.keys(), param_combination))
    print(f"Testing Hyperparameters: {params}")

    current_best_model, best_cost_per_gen, avg_cost_per_gen = evaluate_hyperparameters(
        params, x_train, y_train, x_val, y_val,
        input_size=x_train.shape[1],
        output_size=len(np.unique(y_train))
    )

    hidden_output_train = np.tanh(np.dot(x_train, current_best_model['weights_hidden']))
    output_train = np.dot(hidden_output_train, current_best_model['weights_output'])
    train_predictions = np.argmax(output_train, axis=1)
    train_accuracy = accuracy_score(y_train, train_predictions)

    hidden_output_val = np.tanh(np.dot(x_val, current_best_model['weights_hidden']))
    output_val = np.dot(hidden_output_val, current_best_model['weights_output'])
    val_predictions = np.argmax(output_val, axis=1)
    val_accuracy = accuracy_score(y_val, val_predictions)

    hidden_output_test = np.tanh(np.dot(x_test, current_best_model['weights_hidden']))
    output_test = np.dot(hidden_output_test, current_best_model['weights_output'])
    test_predictions = np.argmax(output_test, axis=1)
    test_accuracy = accuracy_score(y_test, test_predictions)
```

```
247
248     print(f"Train Accuracy: {train_accuracy:.4f}, Validation Accuracy: {val_accuracy:.4f}, Test
        Accuracy: {test_accuracy:.4f}")
249
250     train_accuracies.append(train_accuracy)
251     val_accuracies.append(val_accuracy)
252     test_accuracies.append(test_accuracy)
253     hyperparams_list.append(params)
254
255     if test_accuracy > best_test_accuracy:
256         best_test_accuracy = test_accuracy
257         best_train_accuracy = train_accuracy
258         best_val_accuracy = val_accuracy
259         best_model_overall = current_best_model
260         best_hyperparams = params
261
262     all_best_costs.append(best_cost_per_gen)
263     all_average_costs.append(avg_cost_per_gen)
264
265
266 plt.figure(figsize=(8, 4))
267 plt.plot(train_accuracies, label='Train Accuracy', color='blue', marker='o')
268 plt.title("Train Accuracy Across Hyperparameters")
269 plt.xlabel("Hyperparameter Combination Index")
270 plt.ylabel("Accuracy")
271 plt.legend()
272 plt.grid()
273 plt.show()
274
275 plt.figure(figsize=(8, 4))
276 plt.plot(val_accuracies, label='Validation Accuracy', color='green', marker='s')
277 plt.title("Validation Accuracy Across Hyperparameters")
278 plt.xlabel("Hyperparameter Combination Index")
279 plt.ylabel("Accuracy")
280 plt.legend()
281 plt.grid()
282 plt.show()
283
284 plt.figure(figsize=(8, 4))
285 plt.plot(test_accuracies, label='Test Accuracy', color='red', marker='^')
286 plt.title("Test Accuracy Across Hyperparameters")
287 plt.xlabel("Hyperparameter Combination Index")
288 plt.ylabel("Accuracy")
289 plt.legend()
290 plt.grid()
291 plt.show()
292
293
294 print(f"Best Hyperparameters: {best_hyperparams}")
295 print(f"Best Train Accuracy: {best_train_accuracy:.4f}")
296 print(f"Best Validation Accuracy: {best_val_accuracy:.4f}")
297 print(f"Best Test Accuracy: {best_test_accuracy:.4f}")
298
299
300 plt.figure(figsize=(12, 8))
301
302 for i, (best_cost, avg_cost) in enumerate(zip(all_best_costs, all_average_costs)):
303     plt.subplot(2, 1, 1)
304     plt.plot(best_cost, label=f"Best Cost (Hyperparam {i+1})")
305     plt.title("Best Cost Across Generations for Different Hyperparameters")
306     plt.xlabel("Generation")
307     plt.ylabel("Best Cost (Fitness Score)")
308     plt.legend()
```

```
309
310     plt.subplot(2, 1, 2)
311     plt.plot(avg_cost, label=f"Average Cost (Hyperparam {i+1})")
312     plt.title("Average Cost Across Generations for Different Hyperparameters")
313     plt.xlabel("Generation")
314     plt.ylabel("Average Cost (Fitness Score)")
315     plt.legend()
316
317 plt.tight_layout()
318 plt.show()
```