Report on Assignment 2

Su Kexin, 24102174G

I. INTRODUCTION

Machine learning enables computer systems to automatically learn patterns and patterns from data without the need for explicit programming. Its core goal is to enable computers to predict, classify, or discover hidden structures in input data through algorithms and statistical models, thereby achieving intelligent decision-making and task execution. Machine learning is mainly divided into supervised learning and unsupervised learning.

Linear regression is a supervised learning algorithm. Assuming there is a set of input features and corresponding output labels, a linear regression model attempts to find a linear function to establish a linear relationship model between variables.

Neural network is a machine learning model that simulates the structure and workings of human brain neurons, capable of learning highly complex nonlinear relationships. A neural network consists of multiple neurons connected in a hierarchical structure, typically including an input layer, a hidden layer, and an output layer. Each neuron receives input signals from the previous layer of neurons, generates output signals by weighted summation and applying activation functions, and then passes the output signals to the next layer of neurons. By adjusting the connection weights and bias terms between neurons, neural networks utilize a large amount of training data for learning to minimize loss functions (such as cross entropy loss, mean square error, etc.), enabling the model to accurately predict or classify input data.

Perceptron is a neural network model primarily used for binary classification problems. The perceptron model is based on the working principle of neurons, which receive multiple input features, each of which has a corresponding weight. The model calculates the weighted sum of input features and weights, and then converts the weighted sum into output categories through an activation function (usually a step function). The perceptron uses training data to minimize classification error rates by continuously adjusting weights and bias terms.

II. METHODOLOGY

A. Task 1

1. Dataset and Analysis

1）Introduction of packages

The data package of ensemble learning (Bagging) based on linear regression model includes：

```
import pandas as pd
from sklearn.preprocessing import LabelEncoder
from sklearn.model_selection import train_test_split
from sklearn.ensemble import BaggingRegressor
from sklearn.linear_model import LinearRegression
from sklearn.metrics import mean_squared_error
```

The data packet for boosting ensemble learning based on linear regression model needs to include the following packets：

```
from sklearn.ensemble import AdaBoostRegressor
```

2）Load data and extract features and labels

Extract the following columns as features: [’symboling’,’fueltype’,’doornumber’,’carbody’, ’carheight’, ’stroke’, ’ c ompressionratio’, ’horsepower’,’peakrpm’, ’Company’] .Extract ‘price’ column as labels.

```
# Load dataset
df = pd.read_csv('CarPrice.csv')

# Extract features and labels
X = df[['symboling', 'fueltype', 'doornumber', 'carbody', 'carheight', 'stroke', 'compressionratio', 'horsepower', 'peakrpm']]
y = df['price']
```

3）Print info, description, and pairwise correlation of columns.

```
print('info, description: ')
df.info()
print('pairwise correlation of columns: ')
print(df.head())
```

4) Transform non-numeric features to numeric features.

Create an empty dictionary *label_encoders* to store the *LabelEncoder* object corresponding to each non numeric column.

Traverse each column of feature *X*, and if the data type of the column is an object (i.e. non numeric type), perform the following operation:

Create a *LabelEncoder* object and store it in the *label_encoders* dictionary, with the key being the column name.

Use the *fit_transform* method to convert non numeric data in the column to numeric data, and assign the converted data back to the original *X* data box using *X.loc [:, column]* to ensure modifications are made to the original data.

```
label_encoders = {}
for column in X.columns:
    if X[column].dtype == 'object':
        label_encoders[column] = LabelEncoder()
        X.loc[:, column] = label_encoders[column].fit_transform(X[column])
```

5) Split the dataset into training and testing sets, adopting an 80/20 ratio.

```
X_train, X_test, y_train, y_test = train_test_split(X, y, test_size=0.2, random_state=42)
```

2. Ensemble Learning with Linear Classifiers

1) Set up BaggingRegressor model and set 10 linear regression models as base models.

BaggingRegressor from sklearn is used to implement Bagging ensemble, specifying linear regression as the base estimator, and training 10 models (n_estimators=10).

In this task, BaggingRegressor defaults to using the **ensemble method of average prediction**, with voting mechanisms typically used for classification problems and average prediction typically used for regression problems.

BaggingRegressor is a black box model that only focuses on the final ensemble prediction results and does not retain the specific information of each base model, so it cannot directly output each base learner.

```
bagging_model = BaggingRegressor(estimator=LinearRegression(), n_estimators=10, random_state=42)
```

2) Training Bagging Model

```
bagging_model.fit(X_train, y_train)
```

3) Evaluate the performance of the model on the test set

Bagging_predictions=bagging_madel. redist (X_test) uses the trained BaggingRegressor model to predict the test set X_test and obtain the prediction results.

Bagging_cse=mean_Squared_error (y_test, bagging_predictions) Use MSE to evaluate the performance of the model on the test set.

```
bagging_predictions = bagging_model.predict(X_test)
```

```
bagging_mse = mean_squared_error(y_test, bagging_predictions)

print(f'Bagging Mean Squared Error: {bagging_mse}')
```

4) Implementing ensemble learning based on boosting algorithm

Using AdaBoostRegressor as the implementation of the boosting algorithm, with a linear regression model as the base model. AdaBoost will continuously adjust the weights of each base model during sequence training, so that subsequent base models can better correct the errors of the previous model.

```
adaboost_model = AdaBoostRegressor(estimator=LinearRegression(), n_estimators=10, random_state=42)

adaboost_model.fit(X_train, y_train)

adaboost_predictions = adaboost_model.predict(X_test)

adaboost_mse = mean_squared_error(y_test, adaboost_predictions)
print(f'AdaBoost Mean Squared Error: {adaboost_mse}')
```

5) the individual linear classifier

```
model = LinearRegression()
model.fit(X_train, y_train)

y_pred = model.predict(X_test)

mse = mean_squared_error(y_test, y_pred)
print(f'the individual linear classifier MSE: {mse}')
```

B. Task 2

1. Dataset and Analysis

1) Import Panda package, output basic information of data and the first few lines of data.

```
import pandas as pd
df_test = pd.read_csv('mnist_test.csv')

print('basic info: ')
df_test.info()
print('first few lines of data: ')
print(df_test.head())
```

2) Extraction of label columns and feature data:

```
# Separate labels and features
y = df_test['label'].values
X = df_test.drop('label', axis=1).values
```

3) convert data into a picture and visualize data:

Define the function show_image, which has a length of 784 (28x28). Reorder this one-dimensional array into a two-dimensional array with 28 rows and 28 columns by calling reshape (28, 28). So that the imshow function in the matplotlib library can be used for display. The imshow function requires the input image data to be a two-dimensional array, where each element represents a pixel value of the image. By adjusting one-dimensional data into a 28x28 two-dimensional array, we can consider it as a 28x28 grayscale image, where each pixel value represents the grayscale intensity of the image.

```
def show_image(image):
    image = image.reshape(28, 28)
    plt.imshow(image, cmap='gray')
    plt.axis('off')
    plt.show()

print('the first image: ')
show_image(X_test[0])
```

4) Normalize features and use one-hot encoding encodes labels:

Normalize the feature data and then perform one hot encoding on the labels to facilitate the preparation of data for input into machine learning models for training and prediction.

```
X = X / 255.0

encoder = OneHotEncoder(sparse_output=False)
y = encoder.fit_transform(y.reshape(-1, 1))
```

2. Neural Network Construction

1) Using Sigmoid function as activation function:

Using Sigmoid function as activation function to introduce nonlinear characteristics into neural networks. Define a function called sigmoid and return the value of the sigmoid function through a formula. Define a function called sigmoid_derivative, calculate and return the derivative of the sigmoid function, which is used in backpropagation algorithms to calculate gradients and update the weights of the neural network.

```
def sigmoid(x):
    return 1 / (1 + np.exp(-x))

def sigmoid_derivative(x):
    return x * (1 - x)
```

2) Initialize neural network:

Define the size of the input layer, two hidden layers, and output layer of a neural network.Initialize weight matrices W1, W2, and W3 with random numbers, and initialize bias vectors b1, b2, and b3 to zero.

Next, define the learning rate, number of training epochs, and batch size. The learning rate controls the step size of weight updates, the number of training epochs determines the number of times the model is trained, and the batch size determines the number of samples used for each training session.

```
# Initialize parameters
input_size = 28 * 28
hidden_size_1 = 128
hidden_size_2 = 64
output_size = 10

# Weights initialization
np.random.seed(42)
W1 = np.random.rand(input_size, hidden_size_1) * 0.01
b1 = np.zeros((1, hidden_size_1))
W2 = np.random.rand(hidden_size_1, hidden_size_2) * 0.01
b2 = np.zeros((1, hidden_size_2))
W3 = np.random.rand(hidden_size_2, output_size) * 0.01
b3 = np.zeros((1, output_size))
class NeuralNetwork:
    def __init__(self, input_size, hidden_size, output_size):
        self.weights1 = np.random.rand(input_size, hidden_size)
        self.bias1 = np.zeros((1, hidden_size))
        self.weights2 = np.random.rand(hidden_size, output_size)
        self.bias2 = np.zeros((1, output_size))
```

```
learning_rate = 0.1
epochs = 100
batch_size = 32
```

3) Implement forward propagation and back propagation algorithms to train the network:

Loop 100 times in the outer loop to control the number of rounds in the entire training process. The inner loop is used to batch process training data.

forward propagation: The forward propagation function is used to calculate the output of a neural network. *Sigmoid (np. dot (X, self. weights 1)+self. bias 1)* activates the hidden layer by using the Sigmoid function to obtain the output of the hidden layer, self. hidden_1ayer. *Sigmoid (np. dot (self. hidden_1ayer, self. weights 2)+self. bias 2)* activates the output layer's pre activation value through the Sigmoid function to obtain the output layer's output self. output_1ayer.

back propagation:The back propagation function is used to calculate gradients and update weights and biases.

```
# Forward propagation
Z1 = np.dot(X_batch, W1) + b1
A1 = sigmoid(Z1)
Z2 = np.dot(A1, W2) + b2
A2 = sigmoid(Z2)
Z3 = np.dot(A2, W3) + b3
A3 = sigmoid(Z3)  # Final output

# Backpropagation
dZ3 = A3 - y_batch
dW3 = np.dot(A2.T, dZ3) / batch_size
db3 = np.sum(dZ3, axis=0, keepdims=True) / batch_size

dZ2 = np.dot(dZ3, W3.T) * sigmoid_derivative(A2)
dW2 = np.dot(A1.T, dZ2) / batch_size
db2 = np.sum(dZ2, axis=0, keepdims=True) / batch_size

dZ1 = np.dot(dZ2, W2.T) * sigmoid_derivative(A1)
dW1 = np.dot(X_batch.T, dZ1) / batch_size
db1 = np.sum(dZ1, axis=0, keepdims=True) / batch_size
```

4) Update the functions for weights and biases:

```
# Update weights and biases
W1 -= learning_rate * dW1
b1 -= learning_rate * db1
W2 -= learning_rate * dW2
b2 -= learning_rate * db2
W3 -= learning_rate * dW3
b3 -= learning_rate * db3
```

5) Evaluate on the validation set:

Perform forward propagation on the validation set and calculate the accuracy of the validation set.For example, Z1nval=np. dot (X_val, W1)+b1 is used to calculate the validation set data X_val, which undergoes a linear transformation of the first layer weight W1 and bias b1 to obtain Z1nval. Then use the Sigmoid activation function to obtain the activation value A1nval for the first layer. And so on, the final output value A3nval is obtained.

```
# Evaluate on validation set
Z1_val = np.dot(X_val, W1) + b1
A1_val = sigmoid(Z1_val)
Z2_val = np.dot(A1_val, W2) + b2
A2_val = sigmoid(Z2_val)
Z3_val = np.dot(A2_val, W3) + b3
A3_val = sigmoid(Z3_val)
A3_val = softmax(A3_val, axis=1)
```

6) Use cross-entropy loss to measure the network’s performance during training:

Add calculation of cross entropy loss and print the accuracy and loss of the validation set at the end of each training round.

```
# Calculate cross - entropy loss
val_loss = log_loss(y_val, A3_val)

print(f'Epoch {epoch+1}/{epochs}, Validation Accuracy: {val_accuracy:.4f}, Validation Loss: {val_loss:.4f}')
```

C. Task 3

This task mainly uses genetic algorithms to find the optimal hyperparameters for artificial neural network tasks, including the hyperparameters designed and obtained in my task2 (learning rate and number of neurons in two hidden layers). Finally, the model accuracy is evaluated on the validation set to measure the superiority or inferiority of the hyperparameters.

Define a genetic algorithm based on Task 2:

1）Set generations=10, populationsize=10, mutation_rate=0.1. Set the creat_chromosome function: This function is used to randomly generate a chromosome (i.e. a set of hyperparameter combinations), including randomly generating learning rates within a specified range, and randomly selecting the number of neurons in two common hidden layers from the given options, representing a potential neural network hyperparameter configuration.

```
generations = 10
population_size = 10
mutation_rate = 0.1

def create_chromosome():
    learning_rate = np.random.uniform(0.001, 0.1)
    hidden_size_1 = np.random.choice([32, 64, 128, 256])
    hidden_size_2 = np.random.choice([32, 64, 128, 256])
    return [learning_rate, hidden_size_1, hidden_size_2]
```

2）Definition of genetic algorithm related functions, including fitness function, select function, crossover function, mutate function：

```
def fitness(chromosome, X, y):
    learning_rate, hidden_size_1, hidden_size_2 = chromosome
    return train_and_evaluate(X, y, learning_rate, hidden_size_1, hidden_size_2)

def select(population, fitness_scores):
    sorted_population = [x for _, x in sorted(zip(fitness_scores, population), reverse=True)]
    return sorted_population[:population_size // 2]

def crossover(parent1, parent2):
    point = np.random.randint(1, len(parent1))
    child1 = parent1[:point] + parent2[point:]
    child2 = parent2[:point] + parent1[point:]
    return child1, child2

def mutate(chromosome, mutation_rate):
    if np.random.rand() < mutation_rate:
        index = np.random.randint(0, len(chromosome))
        if index == 0:
            chromosome[index] = np.random.uniform(0.001, 0.1)
        else:
            chromosome[index] = np.random.choice([32, 64, 128, 256])
    return chromosome

def decode_chromosome(chromosome):
    return chromosome
```

3）In the train_nd_evaluate function, a three-layer neural network was constructed and trained based on given hyperparameters (learning rate, number of neurons in two hidden layers), and its accuracy was evaluated on the validation set. This includes initializing parameters and training loops on the data (forward propagation, backpropagation, updating weights and biases, as shown in the code in task2), and finally evaluating the model to return accuracy.

```
def train_and_evaluate(X, y, learning_rate, hidden_size_1, hidden_size_2):
    # Initialize parameters
    input_size = 28 * 28
    output_size = 10

    # Weights initialization

    epochs = 10
    batch_size = 32

    # Training loop
    for epoch in range(epochs):
        for i in range(0, X_train.shape[0], batch_size):
            X_batch = X_train[i:i + batch_size]
            y_batch = y_train[i:i + batch_size]

            # Forward propagation


            # Backpropagation


            # Update weights and biases


    # Evaluate on validation set
    Z1_val = np.dot(X_val, W1) + b1
    A1_val = sigmoid(Z1_val)
    Z2_val = np.dot(A1_val, W2) + b2
    A2_val = sigmoid(Z2_val)
    Z3_val = np.dot(A2_val, W3) + b3
    A3_val = sigmoid(Z3_val)
    A3_val = softmax(A3_val, axis=1)

    # Calculate accuracy
    val_predictions = np.argmax(A3_val, axis=1)
    val_true = np.argmax(y_val, axis=1)
    val_accuracy = accuracy_score(val_true, val_predictions)

    return val_accuracy
```

4）Execution of Genetic Algorithm

```
# Genetic Algorithm Execution
population = [create_chromosome() for _ in range(population_size)]
for generation in range(generations):
    fitness_scores = [fitness(chromosome, X_train, y_train) for chromosome in population]
    print(f"Generation {generation+1}, Best Fitness: {max(fitness_scores):.4f}")

    selected = select(population, fitness_scores)
    next_generation = []

    while len(next_generation) < population_size:
        parent1, parent2 = random.sample(selected, 2)
        child1, child2 = crossover(parent1, parent2)
        next_generation.append(mutate(child1, mutation_rate))
        next_generation.append(mutate(child2, mutation_rate))

    population = next_generation
```

5）After the iteration of the genetic algorithm is completed, find the hyperparameter combination with the highest fitness (highest accuracy on the validation set) from the final population and print it out.

```
best_chromosome = max(population, key=lambda chromo: fitness(chromo, X_train, y_train))
best_learning_rate, best_hidden_size_1, best_hidden_size_2 = decode_chromosome(best_chromosome)
print(f"Optimal Hyperparameters: Learning Rate = {best_learning_rate}, Hidden Sizes = {best_hidden_size_1}, {best_hidden_size_2}")
```

D. Task 4

The experimental content of Task 4 will be presented in the Experimental results section. The content is generated through OpenAI's GPT (**GPT-4o**), a large language model.

III. Experimental results

A. Task 1

1) Print info, description, and pairwise correlation of columns.

```
info, description:
<class 'pandas.core.frame.DataFrame'>
RangeIndex: 205 entries, 0 to 204
Data columns (total 26 columns):
 #   Column            Non-Null Count  Dtype
---  ------            --------------  -----
 0   car_ID            205 non-null    int64
 1   symboling         205 non-null    int64
 2   CarName           205 non-null    object
 3   fueltype          205 non-null    object
 4   aspiration        205 non-null    object
 5   doornumber        205 non-null    object
 6   carbody           205 non-null    object
 7   drivewheel        205 non-null    object
 8   enginelocation    205 non-null    object
 9   wheelbase         205 non-null    float64
 10  carlength         205 non-null    float64
 11  carwidth          205 non-null    float64
 12  carheight         205 non-null    float64
 13  curbweight        205 non-null    int64
 14  enginetype        205 non-null    object
 15  cylindernumber    205 non-null    object
 16  enginesize        205 non-null    int64
 17  fuelsystem        205 non-null    object
 18  boreratio         205 non-null    float64
 19  stroke            205 non-null    float64
 20  compressionratio  205 non-null    float64
 21  horsepower        205 non-null    int64
 22  peakrpm           205 non-null    int64
 23  citympg           205 non-null    int64
 24  highwaympg        205 non-null    int64
 25  price             205 non-null    float64
dtypes: float64(8), int64(8), object(10)
memory usage: 41.8+ KB
```

```
rrelation of columns:
symboling                  CarName fueltype aspiration doornumber  \
        3        alfa-romero giulia      gas        std        two
        3       alfa-romero stelvio      gas        std        two
        1  alfa-romero Quadrifoglio      gas        std        two
        2               audi 100 ls      gas        std       four
        2                audi 100ls      gas        std       four

ody drivewheel enginelocation  wheelbase  ...  enginesize  \
ble        rwd          front       88.6  ...         130
ble        rwd          front       88.6  ...         130
ack        rwd          front       94.5  ...         152
dan        fwd          front       99.8  ...         109
dan        4wd          front       99.4  ...         136

em  boreratio  stroke compressionratio horsepower  peakrpm citympg  \
fi       3.47    2.68             9.0        111     5000      21
fi       3.47    2.68             9.0        111     5000      21
fi       2.68    3.47             9.0        154     5000      19
fi       3.19    3.40            10.0        102     5500      24
fi       3.19    3.40             8.0        115     5500      18

pg    price
27  13495.0
27  16500.0
26  16500.0
30  13950.0
22  17450.0

6 columns]
```

2) Using bagging algorithm, evaluate the performance of the model on the test set by changing the n_estimators and using mean square error (MSE).

n_estimators	5	50	100	150	300	700
MSE	22231774.8808	21397202.986	21263248.21444	20807093.61413	20775688.6106	21006319.5665

The core idea of Bagging algorithm is to improve the overall predictive performance of the model by integrating multiple weak learners (here multiple linear regression models).

When the number of base models is small (such as 5), the effect of ensemble learning is not very obvious, and the MSE is relatively high.

As the number of base models increases (50, 100,150,300), the advantages of ensemble learning gradually become apparent, and the

MSE value gradually decreases.

When there are too many base models (such as 700), overfitting models have poor generalization ability on new data, resulting in larger deviations in the end.

For individual linear classifier, the following results can be obtained:

```
the individual linear classifier MSE: 21025637.20183116
```

B. Task 2

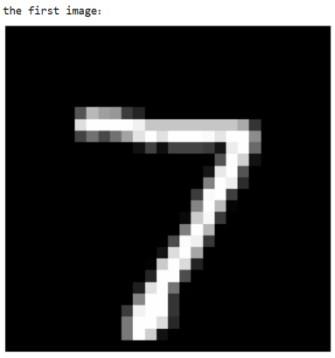
1) Print info, description, and pairwise correlation of columns.

```
basic info:
<class 'pandas.core.frame.DataFrame'>
RangeIndex: 10000 entries, 0 to 9999
Columns: 785 entries, label to 28x28
dtypes: int64(785)
memory usage: 59.9 MB
first few lines of data:
   label  1x1  1x2  1x3  1x4  1x5  1x6  1x7  1x8  1x9  ...  28x19  28x20  \
0      7    0    0    0    0    0    0    0    0    0  ...      0      0
1      2    0    0    0    0    0    0    0    0    0  ...      0      0
2      1    0    0    0    0    0    0    0    0    0  ...      0      0
3      0    0    0    0    0    0    0    0    0    0  ...      0      0
4      4    0    0    0    0    0    0    0    0    0  ...      0      0

   28x21  28x22  28x23  28x24  28x25  28x26  28x27  28x28
0      0      0      0      0      0      0      0      0
1      0      0      0      0      0      0      0      0
2      0      0      0      0      0      0      0      0
3      0      0      0      0      0      0      0      0
4      0      0      0      0      0      0      0      0

[5 rows x 785 columns]
```

2) convert data into a picture and visualize it.



3) Use cross-entropy loss to measure the network's performance during training.

```
Epoch 93/100, Validation Accuracy: 0.9440, Validation Loss: 1.5391
Epoch 94/100, Validation Accuracy: 0.9440, Validation Loss: 1.5385
Epoch 95/100, Validation Accuracy: 0.9435, Validation Loss: 1.5379
Epoch 96/100, Validation Accuracy: 0.9435, Validation Loss: 1.5373
Epoch 97/100, Validation Accuracy: 0.9435, Validation Loss: 1.5367
Epoch 98/100, Validation Accuracy: 0.9435, Validation Loss: 1.5361
Epoch 99/100, Validation Accuracy: 0.9440, Validation Loss: 1.5356
Epoch 100/100, Validation Accuracy: 0.9440, Validation Loss: 1.5351
```

As the number of training epochs (Epoch) increases, the accuracy gradually improves and the loss value gradually decreases, indicating that the model is continuously learning and optimizing, gradually fitting the training data.

C. Task 3

Here, select the genetic algorithm and other parameters as generations=10, Population_Size=10, mutation_rate=0.1, epochs=10, ch_Size=32. This will result in a slightly poorer final fitness, but the computation speed will be much faster. For example:

```
Generation 1, Best Fitness: 0.2660
Generation 2, Best Fitness: 0.2490
Generation 3, Best Fitness: 0.2660
Generation 4, Best Fitness: 0.2660
Generation 5, Best Fitness: 0.2760
Generation 6, Best Fitness: 0.2760
Generation 7, Best Fitness: 0.2760
Generation 8, Best Fitness: 0.2760
Generation 9, Best Fitness: 0.2760
Generation 10, Best Fitness: 0.2760
Optimal Hyperparameters: Learning Rate = 0.08990662719230026, Hidden Sizes = 256, 128
```

If the epochs are increased by 50, the accuracy of the final neural network on the validation set will significantly improve after training on the training set, as shown in the figure:

```
Generation 3, Best Fitness: 0.9050
Generation 4, Best Fitness: 0.9055
Generation 5, Best Fitness: 0.9055
```

We can see from the first chart (epochs=10) that The best fitness score (validation accuracy) improved over the generations, suggesting that the genetic algorithm was effective in optimizing the hyperparameters.

And we can see that by randomly generating learning rates within a specified range and randomly selecting the number of neurons in two common hidden layers from the given options, the final hyperparameters under this parameter can be obtained: Learning Rate: 0.0899, Hidden Layer Sizes: 256 and 128.

D. Task 4

1. Introducing Large Language Model by Large Language Model

1）***Q***：Conduct a survey of existing literature related to large language models, highlighting their applications and advancements.

A：Conducting a survey of existing literature on large language models (LLMs) involves examining a wide range of research papers, articles, and reports that discuss their development, applications, and advancements. Here is an overview of key themes and findings from the literature on LLMs:

Introduction to Large Language Models

Large language models, such as OpenAI's GPT series, Google's BERT, and others, are neural networks trained on vast amounts of text data. They are designed to understand and generate human-like text, making them powerful tools for natural language processing (NLP) tasks.

Key Advancements

Transformer Architecture: The introduction of the transformer architecture by Vaswani et al. in 2017 revolutionized NLP by enabling models to process text in parallel, improving efficiency and performance.

Scaling Laws: Research has shown that increasing the size of models (in terms of parameters and data) generally leads to better performance, as demonstrated by models like GPT-3 and GPT-4.

Pre-training and Fine-tuning: The paradigm of pre-training on large datasets followed by fine-tuning on specific tasks has become standard, allowing models to adapt to a wide range of applications.

Applications

Text Generation: LLMs are used to generate coherent and contextually relevant text, useful in applications like content creation, storytelling, and dialogue systems.

Machine Translation: Models like BERT and GPT have improved the quality of machine translation by understanding context and nuances in language.

Sentiment Analysis: LLMs can analyze text to determine sentiment, aiding in customer feedback analysis and social media monitoring.

Question Answering: Models are employed in systems that answer questions based on a given context, such as virtual assistants and search engines.
Code Generation: LLMs like OpenAI's Codex can generate code snippets, assisting developers in programming tasks.
Healthcare: They are used for tasks like summarizing medical documents, predicting patient outcomes, and assisting in diagnostics.

Challenges and Ethical Considerations

Bias and Fairness: LLMs can perpetuate biases present in training data, leading to unfair or harmful outputs. Ongoing research focuses on mitigating these biases.
Energy Consumption: Training large models requires significant computational resources, raising concerns about environmental impact.
Misuse: The ability of LLMs to generate realistic text can be misused for disinformation or creating harmful content.

Future Directions

Efficiency Improvements: Research is focused on making LLMs more efficient in terms of computation and energy use, such as through model distillation and pruning.
Multimodal Models: Integrating text with other data types (e.g., images, audio) to create more comprehensive AI systems.
Continual Learning: Developing models that can learn continuously from new data without forgetting previous knowledge.
Explainability: Enhancing the interpretability of LLMs to make their decision-making processes more transparent.

Conclusion

The literature on large language models highlights their transformative impact on NLP and various industries. While they offer significant benefits, addressing challenges related to bias, energy consumption, and ethical use remains crucial. Future research will likely focus on making these models more efficient, fair, and capable of handling diverse data types.

2）***Q***：Conduct Provide an overview of the history and evolution of large language models, discussing key milestones and breakthroughs in the field.

A：The history and evolution of large language models (LLMs) is marked by several key milestones and breakthroughs that have significantly advanced the field of natural language processing (NLP). Here is an overview of these developments::

Early Foundations

- 1950s-1980s: Rule-Based Systems and Early Statistical Models
  - Early NLP systems were primarily rule-based, relying on handcrafted rules and linguistic knowledge.
  - The introduction of statistical methods in the 1980s, such as Hidden Markov Models (HMMs), began to shift the focus towards data-driven approaches.

The Rise of Neural Networks

- 1990s-2000s: Neural Networks and Word Embeddings
  - The resurgence of interest in neural networks led to the development of models like recurrent neural networks (RNNs) and long short-term memory networks (LSTMs), which were better suited for sequential data like text.
  - The introduction of word embeddings, such as Word2Vec by Mikolov et al. in 2013, allowed for the representation of words in continuous vector spaces, capturing semantic relationships.

The Transformer Revolution

- 2017: The Transformer Model
  - The publication of "Attention is All You Need" by Vaswani et al. introduced the transformer architecture, which used self-attention mechanisms to process text in parallel, significantly improving efficiency and performance over RNNs and LSTMs.

Emergence of Large Language Models

- 2018: BERT (Bidirectional Encoder Representations from Transformers)
  - Developed by Google, BERT introduced bidirectional training of transformers, allowing models to consider context from both directions in a sentence. This led to state-of-the-art results on various NLP benchmarks.
- 2018: GPT (Generative Pre-trained Transformer)
  - OpenAI released GPT, which demonstrated the effectiveness of unsupervised pre-training followed by supervised fine-tuning. This model laid the groundwork for subsequent iterations.
- 2019: GPT-2
  - GPT-2, with 1.5 billion parameters, showcased the potential of scaling up model size, achieving impressive results in text generation and other tasks. Its release sparked discussions about the ethical implications of powerful LLMs.
- 2020: GPT-3
  - GPT-3, with 175 billion parameters, marked a significant leap in model size and capabilities. It demonstrated few-shot and zero-shot learning, where the model could perform tasks with minimal task-specific training.

Recent Developments

- 2021: Codex and DALL-E

- OpenAI's Codex, based on GPT-3, was designed for code generation, showcasing the versatility of LLMs in programming tasks.
- DALL-E, another OpenAI model, extended the transformer architecture to generate images from textual descriptions, highlighting the potential for multimodal models.

- 2022: ChatGPT
  - Built on the GPT-3 architecture, ChatGPT was fine-tuned for conversational applications, demonstrating the model's ability to engage in coherent and contextually relevant dialogues.
- 2023: GPT-4 and Beyond
  - GPT-4 and other contemporary models continued to push the boundaries of LLMs, focusing on improving efficiency, reducing biases, and enhancing capabilities across diverse applications.

Key Breakthroughs and Trends

- Scaling Laws: Research has consistently shown that increasing model size and training data leads to better performance, driving the trend towards ever-larger models.

- Transfer Learning: The paradigm of pre-training on large datasets followed by fine-tuning on specific tasks has become a standard approach, enabling models to generalize across various applications.

- Ethical and Societal Implications: As LLMs become more powerful, discussions around their ethical use, potential biases, and societal impact have become increasingly important.

Conclusion

The evolution of large language models has been characterized by rapid advancements in model architecture, size, and capabilities. From early rule-based systems to today's sophisticated transformers, LLMs have transformed the field of NLP and continue to drive innovation across numerous domains. Future developments will likely focus on enhancing efficiency, interpretability, and ethical considerations.

2.Model Construction by Large Language Model

Q： When manually building a neural network to solve handwriting recognition problems, you need to complete 1 Design and implement a neural network architecture with multiple layers.
Use an appropriate loss function.
Identify an appropriate loss function that aligns with these requirements with objectives in this task

A： To build a neural network for handwriting recognition, you'll typically want to use a convolutional neural network (CNN) architecture, as these are well-suited for image processing tasks. Here's an outline of the steps and considerations:
Cross-Entropy Loss：

1）Design and Implement a Neural Network Architecture
Layers and Structure

- Input Layer: Accepts the image data (e.g., 28x28 pixels for MNIST).
- Convolutional Layers: Extract features from the images.
  - Use filters (e.g., 3x3, 5x5) and activation functions like ReLU.
- Pooling Layers: Reduce dimensionality while retaining important features.
  - Commonly use max pooling (e.g., 2x2).
- Fully Connected Layers: Connect all neurons in one layer to every neuron in the next layer.
  - These layers help in decision making after feature extraction.
- Output Layer: Use a softmax activation function for classification.
  - Number of neurons = number of classes (e.g., 10 for digits 0-9).

2）Use an Appropriate Loss Function
For handwriting recognition, which is typically a multi-class classification problem, the most appropriate loss function is Categorical Cross-Entropy.
Categorical Cross-Entropy

- Purpose: Measures the difference between the true distribution (actual labels) and the predicted distribution (output of the network).
- Formula: If yyy is the true label and y^\\hat{y}y^ is the predicted probability, the categorical cross-entropy loss is:

$$L(y, \hat{y}) = -\sum_{i=1}^{C} y_i \log(\hat{y}_i)$$

- Usage: Works well with softmax outputs. Use one-hot encoding for the labels.

3）Implementation Tips

- **Data Preprocessing**: Normalize image data to improve convergence.
- **Regularization**: Use dropout or L2 regularization to prevent overfitting.
- **Optimizer**: Use Adam or SGD for efficient training.
- **Batch Normalization**: Helps in faster convergence and stability.

4）Example Code Snippet (Python with TensorFlow/Keras)

```
import tensorflow as tf
from tensorflow.keras.models import Sequential
from tensorflow.keras.layers import Conv2D, MaxPooling2D, Flatten, Dense, Dropout

# Define the model
model = Sequential([
    Conv2D(32, (3, 3), activation='relu', input_shape=(28, 28, 1)),
    MaxPooling2D((2, 2)),
    Conv2D(64, (3, 3), activation='relu'),
    MaxPooling2D((2, 2)),
    Flatten(),
    Dense(128, activation='relu'),
    Dropout(0.5),
    Dense(10, activation='softmax')
])

# Compile the model
model.compile(optimizer='adam',
              loss='categorical_crossentropy',
              metrics=['accuracy'])

# Summary of the model
model.summary()
```

This setup should give you a solid starting point for solving handwriting recognition problems using a neural network.

3）Analyze and discuss the effectiveness of the LLM’s contributions to Task II, reflecting on its performance and any insights gained.

LLM proposed using a framework similar to TensorFlow/Keras, and provided suggestions for regularization and callbacks to facilitate our easy implementation of dropout and batch normalization. They also suggested using Adam or SGD for efficient training of Optimizer. And provided some code examples. Furthermore, it confirms that the most suitable loss function is Categorical Cross Entropy.

IV. Conclusion

A. Task 1

1. Performance comparison between integrated models and single linear classifiers

From the data, when n_estimators=5, although the MSE of the ensemble model is larger than that of a single linear classifier, as n_estimators increase, the MSE of the ensemble model gradually decreases and is generally better than that of a single linear

classifier. This indicates that the overall performance of the ensemble model in predicting car prices is better than that of a single linear classifier. However, when n_estimators=700, the overfitting of the model occurs due to the large size of n_estimators, resulting in a decrease in the performance of the ensemble model on the test set.

2. The benefits of using ensemble learning

1) Improve the accuracy of prediction: integrated learning can reduce the limitations of a single model by integrating the prediction of multiple models, take more comprehensive consideration of factors, and improve the accuracy of price prediction.
2) Enhance stability: reduce the sensitivity of the model to data anomalies and fluctuations, smooth out interference, and make price prediction results more stable and reliable.

3. Factors affecting the performance of integrated models

1) Model types and parameter settings: Different types of models (such as decision trees, neural networks, and linear regression models) have different characteristics and advantages. In addition, parameter settings can affect the fitting ability and complexity of the underlying model.
2) Integration methods and parameter settings: Different integration methods have different principles and operating methods, which have varying impacts on performance. For example, Bagging reduces variance and Boosting reduces bias, which are applicable to different data situations. The parameters in the ensemble method can also affect the complexity and performance of the ensemble model.

B. Task 2

This task mainly designs and implements a multi-layer neural network architecture to visualize the data in the dataset, and then selects the sigmoid function as the activation function to compress the output of the neurons into the [0,1] interval. The forward propagation algorithm is used to calculate the output of the neural network, and the back propagation algorithm is used to update the weights and biases of the network. Finally, the cross entropy loss function is used to measure the performance of the network during the training process.
We can see in the results that as the training epochs increase, the accuracy continues to improve while the loss rate gradually decreases.

C. Task 3

In this task, we learned about using genetic algorithms to participate in neural network hyperparameter optimization. By randomly generating initial populations, selecting, crossing, and mutating, it is possible to automatically search for optimal combinations of hyperparameters, avoiding the tedious and blind process of manual parameter tuning.

In addition, corresponding to the neural network itself, we have learned how to preprocess the MNIST dataset, including data reading, feature extraction, label encoding, and other operations. Familiar with how to introduce nonlinearity using activation functions (such as sigmoid functions) and how to calculate the derivative of activation functions for back propagation algorithms. I have gained a clear understanding of the training process of neural networks, including steps such as forward propagation calculation output, backward propagation calculation gradient, updating weights and biases.

D. Task 4

In this task, we learned about the development history and prospects of LLM, and gained a preliminary understanding of this technology. Its characteristics enable it to continuously learn and update knowledge from new data, improving performance and generalization ability. In addition, LLM's assistance in code and data processing work was also attempted through practical examples.