Program 6

Date: 1st October 2024

AIM:

Write a program to implement perceptron neural network for Classification/Regression.

DESCRIPTION:

A perceptron is a fundamental building block of a neural network. It models a neuron with weighted inputs, an activation function, and a single output. For classification tasks like MNIST digit recognition, the perceptron uses an activation function (e.g., ReLU, softmax). For regression tasks like housing price prediction, the perceptron can use linear activation functions.

This program demonstrates two tasks:

1. **Classification**: Recognizing handwritten digits using the MNIST dataset.
2. **Regression**: Predicting housing prices using the Boston Housing dataset.

PROGRAM:

Classification of Handwritten digits using MNIST dataset and Regression of prices using Boston Housing dataset:

```
import numpy as np
import tensorflow as tf
from tensorflow.keras.models import Sequential
from tensorflow.keras.layers import Dense, Flatten
from tensorflow.keras.datasets import mnist, boston_housing

def mnist_classification():
    (x_train, y_train), (x_test, y_test) = mnist.load_data()
    x_train, x_test = x_train / 255.0, x_test / 255.0
    model = Sequential([Flatten(input_shape=(28, 28)), Dense(128, activation='relu'),
Dense(10, activation='softmax')])
    model.compile(optimizer="adam", loss="sparse_categorical_crossentropy",
metrics=["accuracy"])
    model.fit(x_train, y_train, epochs=5, validation_data=(x_test, y_test))
    test_loss, test_acc = model.evaluate(x_test, y_test)
    print("Classification Test Accuracy:", test_acc)

def boston_regression():
    (x_train, y_train), (x_test, y_test) = boston_housing.load_data()
    mean = x_train.mean(axis=0)
    std = x_train.std(axis=0)
    x_train = (x_train - mean) / std
```

```
    x_test = (x_test - mean) / std
    model = Sequential([Dense(64, activation='relu', input_shape=(x_train.shape[1],)),
Dense(64, activation='relu'), Dense(1)])
    model.compile(optimizer="adam", loss="mse", metrics=["mae"])
    model.fit(x_train, y_train, epochs=50, validation_split=0.2, verbose=1)
    test_loss, test_mae = model.evaluate(x_test, y_test)
    print("Regression Test Mean Absolute Error:", test_mae)

if __name__ == "__main__":
    print("Running MNIST Classification")
    mnist_classification()
    print("Boston Housing Regression")
    boston_regression()
```

OUTPUT:

```
Running MNIST Classification
Classification Test Accuracy: 0.9761999845504761
Boston Housing Regression
Regression Test Mean Absolute Error: 3.490344524383545
```

Program 7

Date: 8th October 2024

AIM:

Write a program to implement Backpropagation in neural network.

DESCRIPTION:

Backpropagation (short for "backward propagation of errors") is a supervised learning algorithm used in training artificial neural networks. It calculates the gradient of the loss function with respect to each weight in the network, iteratively updating the weights using techniques like gradient descent. The steps include:

1. **Forward Propagation**: Compute outputs using input data and current weights.
2. **Error Calculation**: Measure the difference between the predicted and actual outputs (using a loss function like MSE or cross-entropy).
3. **Backward Propagation**: Calculate gradients of loss with respect to weights using the chain rule.
4. **Weight Update**: Adjust weights to minimize the loss function.

PROGRAM:

```
import numpy as np

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

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

class NeuralNetwork:

    def __init__(self, input_size, hidden_size, output_size, learning_rate=0.1):
        self.weights_input_hidden = np.random.randn(input_size, hidden_size)
        self.weights_hidden_output = np.random.randn(hidden_size, output_size)
        self.bias_hidden = np.random.randn(1, hidden_size)
        self.bias_output = np.random.randn(1, output_size)
        self.learning_rate = learning_rate
    def forward(self, X):
        self.input = X
        self.hidden_layer_input = np.dot(self.input, self.weights_input_hidden) +
self.bias_hidden
        self.hidden_layer_output = sigmoid(self.hidden_layer_input)
        self.output_layer_input = np.dot(self.hidden_layer_output, self.weights_hidden_output)
+ self.bias_output
```

```
        self.output = sigmoid(self.output_layer_input)
        return self.output

    def backward(self, y_true):
        error = y_true - self.output
        output_gradient = error * sigmoid_derivative(self.output)
        hidden_error = np.dot(output_gradient, self.weights_hidden_output.T)
        hidden_gradient = hidden_error * sigmoid_derivative(self.hidden_layer_output)
        self.weights_hidden_output += np.dot(self.hidden_layer_output.T, output_gradient) *
self.learning_rate
        self.bias_output += np.sum(output_gradient, axis=0, keepdims=True) *
self.learning_rate
        self.weights_input_hidden += np.dot(self.input.T, hidden_gradient) * self.learning_rate
        self.bias_hidden += np.sum(hidden_gradient, axis=0, keepdims=True) *
self.learning_rate

    def train(self, X, y, epochs=10000):
        for epoch in range(epochs):
            self.forward(X)
            self.backward(y)
            if epoch % 1000 == 0:
                loss = np.mean((y - self.output) ** 2)
                print(f"Epoch {epoch}, Loss: {loss}")

X = np.array([[0, 0], [0, 1], [1, 0], [1, 1]])  # Input for XOR
y = np.array([[0], [1], [1], [0]])  # Target output for XOR
nn = NeuralNetwork(input_size=2, hidden_size=4, output_size=1, learning_rate=0.1)
nn.train(X, y, epochs=10000)
print("Test Results:")
for i in range(len(X)):
    print(f"Input: {X[i]} => Predicted Output: {nn.forward(X[i])}")
```

OUTPUT:

```
Epoch 0, Loss: 0.286728725734243
Epoch 1000, Loss: 0.24403521195234307
Epoch 2000, Loss: 0.17614191538379298
Epoch 3000, Loss: 0.04305367455172747
Epoch 4000, Loss: 0.015587387762225248
Epoch 5000, Loss: 0.008572975566837887
Epoch 6000, Loss: 0.00570751578821135
Epoch 7000, Loss: 0.004209096485615914
Epoch 8000, Loss: 0.0033044031069149747
Epoch 9000, Loss: 0.0027050121942078674
Test Results:
Input: [0 0] => Predicted Output: [[0.05200279]]
Input: [0 1] => Predicted Output: [[0.94548377]]
Input: [1 0] => Predicted Output: [[0.96551331]]
Input: [1 1] => Predicted Output: [[0.04753821]]
```

Program 8

Date: 15th October 2024

AIM:

Write a program to implement Backpropagation neural network for Classification/Regression.

DESCRIPTION:

Backpropagation is a supervised learning algorithm used for training artificial neural networks. It is based on the chain rule of calculus and allows the model to update the weights in a way that reduces the error (loss). The process involves two main steps:

1. Forward pass: Input is passed through the network, and output is computed.
2. Backward pass: The error is propagated backward through the network to update the weights using the gradient descent algorithm.

In this program, we implement a simple feedforward neural network for classification. The network has:

An input layer (2 neurons, assuming 2-dimensional input).
A hidden layer (with configurable size).
An output layer (1 neuron for binary classification).
The goal is to classify the points using backpropagation.

PROGRAM:

```
import numpy as np

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

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

class NeuralNetwork:
    def __init__(self, input_size, hidden_size, output_size, learning_rate=0.1):
        self.weights_input_hidden = np.random.randn(input_size, hidden_size)
        self.weights_hidden_output = np.random.randn(hidden_size, output_size)
        self.bias_hidden = np.random.randn(1, hidden_size)
        self.bias_output = np.random.randn(1, output_size)
        self.learning_rate = learning_rate

    def forward(self, X):
        self.input = X
        self.hidden_layer_input = np.dot(self.input, self.weights_input_hidden) +
self.bias_hidden
        self.hidden_layer_output = sigmoid(self.hidden_layer_input)
        self.output_layer_input = np.dot(self.hidden_layer_output, self.weights_hidden_output)
```

```
+ self.bias_output
        self.output = sigmoid(self.output_layer_input)
        return self.output

    def backward(self, y_true):
        error = y_true - self.output
        output_gradient = error * sigmoid_derivative(self.output)
        hidden_error = np.dot(output_gradient, self.weights_hidden_output.T)
        hidden_gradient = hidden_error * sigmoid_derivative(self.hidden_layer_output)
        self.weights_hidden_output += np.dot(self.hidden_layer_output.T, output_gradient) *
self.learning_rate
        self.bias_output += np.sum(output_gradient, axis=0, keepdims=True) *
self.learning_rate
        self.weights_input_hidden += np.dot(self.input.T, hidden_gradient) * self.learning_rate
        self.bias_hidden += np.sum(hidden_gradient, axis=0, keepdims=True) *
self.learning_rate

    def train(self, X, y, epochs=10000):
        for epoch in range(epochs):
            self.forward(X)
            self.backward(y)
            if epoch % 1000 == 0:
                loss = np.mean((y - self.output) ** 2)
                print(f"Epoch {epoch}, Loss: {loss}")

X = np.array([[0, 0], [0, 1], [1, 0], [1, 1]])  # Input for XOR
y = np.array([[0], [1], [1], [0]])  # Target output for XOR
nn = NeuralNetwork(input_size=2, hidden_size=4, output_size=1, learning_rate=0.1)
nn.train(X, y, epochs=10000)
print("Test Results:")
for i in range(len(X)):
    print(f"Input: {X[i]} => Predicted Output: {nn.forward(X[i])}")
```

OUTPUT:

```
Epoch 0, Loss: 0.26625405893226334
Epoch 1000, Loss: 0.2304724438608481
Epoch 2000, Loss: 0.13691834572331607
Epoch 3000, Loss: 0.029168423624694798
Epoch 4000, Loss: 0.011779920169979717
Epoch 5000, Loss: 0.006930624584289342
Epoch 6000, Loss: 0.004804056191638114
Epoch 7000, Loss: 0.0036386781376757187
Epoch 8000, Loss: 0.002911647336191908
Epoch 9000, Loss: 0.002418168771393732
Test Results:
Input: [0 0] => Predicted Output: [[0.04503581]]
Input: [0 1] => Predicted Output: [[0.94729284]]
Input: [1 0] => Predicted Output: [[0.95826121]]
Input: [1 1] => Predicted Output: [[0.04126589]]
```

Program 11

Date: 5th November 2024

AIM:

Implement Fuzzy operations Union, Intersections, Complement.

DESCRIPTION:

In fuzzy logic, sets are represented by membership functions that assign a degree of membership to each element in the set. These degrees range from 0 (no membership) to 1 (full membership). Fuzzy set operations are extensions of classical set operations, such as:

Union: The union of two fuzzy sets returns the maximum membership value for each element from the two sets. $\mu_a \cup_\beta (x) = \max(\mu_a(x), \mu_\beta(x))$

Intersection: The intersection of two fuzzy sets returns the minimum membership value for each element from the two sets. $\mu_a \cap_\beta (x) = \min(\mu_a(x), \mu_\beta(x))$

Complement: The complement of a fuzzy set inverts the membership values, so each element's membership value becomes 1 - μ(x). $\mu_a{}^r(x) = 1 - \mu_a(x)$

We will implement these operations on fuzzy sets where each fuzzy set is represented by a list of tuples. Each tuple contains an element and its associated membership value.

PROGRAM:

```
def fuzzy_union(set_A, set_B):
    union_result = []
    for (x, mu_A) in set_A:
        mu_B = next((mu_B for (y, mu_B) in set_B if x == y), 0)
        union_result.append((x, max(mu_A, mu_B)))
    return union_result

def fuzzy_intersection(set_A, set_B):
    intersection_result = []
    for (x, mu_A) in set_A:
        mu_B = next((mu_B for (y, mu_B) in set_B if x == y), 0)
        intersection_result.append((x, min(mu_A, mu_B)))
    return intersection_result

def fuzzy_complement(set_A):
    complement_result = []
    for (x, mu_A) in set_A:
        complement_result.append((x, 1 - mu_A))
    return complement_result

A = [('A', 0.8), ('B', 0.6), ('C', 0.9), ('D', 0.4)]
B = [('A', 0.7), ('B', 0.5), ('C', 0.2), ('E', 0.3)]
```

```
union_result = fuzzy_union(A, B)
intersection_result = fuzzy_intersection(A, B)
complement_result_A = fuzzy_complement(A)

print("Fuzzy Set A:", A)
print("Fuzzy Set B:", B)
print("\nUnion of A and B:")
print(union_result)
print("\nIntersection of A and B:")
print(intersection_result)
print("\nComplement of A:")
print(complement_result_A)
```

OUTPUT:

```
Fuzzy Set A: [('A', 0.8), ('B', 0.6), ('C', 0.9), ('D', 0.4)]
Fuzzy Set B: [('A', 0.7), ('B', 0.5), ('C', 0.2), ('E', 0.3)]

Union of A and B:
[('A', 0.8), ('B', 0.6), ('C', 0.9), ('D', 0.4)]

Intersection of A and B:
[('A', 0.7), ('B', 0.5), ('C', 0.2), ('D', 0)]

Complement of A:
[('A', 0.19999999999999996), ('B', 0.4), ('C', 0.09999999999999998), ('D', 0.6)]
```

Program 12

Date: 12th November 2024

AIM:

Write a program to implement the concept of Genetic Algorithm.

DESCRIPTION:

A Genetic Algorithm (GA) mimics the process of natural selection. It maintains a population of possible solutions to a problem, evaluates their fitness, and then uses genetic operations like selection, crossover, and mutation to evolve better solutions over successive generations.

In this case, the target is to evolve a population of strings towards a given target string.

The basic steps are:

Initialization: Randomly create an initial population of strings.
Selection: Select individuals from the population based on their fitness (how close they are to the target string).
Crossover: Combine pairs of individuals to create new offspring.
Mutation: Apply random changes to the offspring to maintain genetic diversity.
Termination: Stop the algorithm when a solution matches the target string.

PROGRAM:

```
import random

TARGET = "Hello, Genetic Algorithm!"
POPULATION_SIZE = 100
MUTATION_RATE = 0.01
GENERATION_LIMIT = 1000

def fitness(individual):
    return sum(1 for i, char in enumerate(individual) if char == TARGET[i])

def create_individual():
    return
''.join(random.choice('abcdefghijklmnopqrstuvwxyzABCDEFGHIJKLMNOPQRSTUVWXY
Z, !') for _ in range(len(TARGET)))

def select(population):
    total_fitness = sum(fitness(individual) for individual in population)
    selection_probs = [fitness(individual) / total_fitness for individual in population]
    selected = random.choices(population, weights=selection_probs, k=2)
    return selected

def crossover(parent1, parent2):
    crossover_point = random.randint(0, len(parent1)-1)
```

```
    child1 = parent1[:crossover_point] + parent2[crossover_point:]
    child2 = parent2[:crossover_point] + parent1[crossover_point:]
    return child1, child2

def mutate(individual):
    individual = list(individual)
    for i in range(len(individual)):
        if random.random() < MUTATION_RATE:
            individual[i] = random.choice('abcdefghijklmnopqrstuvwxyzABCDEFGHIJKLMNOPQRSTUVWXYZ, !')
    return ''.join(individual)

def genetic_algorithm():
    population = [create_individual() for _ in range(POPULATION_SIZE)]
    generation = 0
    while generation < GENERATION_LIMIT:
        population.sort(key=fitness, reverse=True)
        if fitness(population[0]) == len(TARGET):
            print(f"Target reached in generation {generation}: {population[0]}")
            break
        next_generation = population[:POPULATION_SIZE//2]
        while len(next_generation) < POPULATION_SIZE:
            parent1, parent2 = select(population)
            child1, child2 = crossover(parent1, parent2)
            next_generation.append(mutate(child1))
            next_generation.append(mutate(child2))
        population = next_generation
        generation += 1
    else:
        print("Target not reached within generation limit.")
genetic_algorithm()
```

OUTPUT:

```
Target reached in generation 340: Hello, Genetic Algorithm!
```
s