39_Sanskruti Nijai

EXPERIMENT NO 03

Apply any of the following learning algorithms to learn the parameters of the supervised single layer feed forward neural network. a. Stochastic Gradient Descent b. Mini Batch Gradient Descent c. Momentum GD d. Nestorev GD e. Adagrad GD f. Adam Learning GD

1. Batch Gradient Descent:

```
import numpy as np
def batch_gradient_descent(gradient_func, initial_position, learning_rate=0.01, num_iterations=100, batch_size=10):
    position = initial_position
    for _ in range(num_iterations):
        # Generate a batch of random data points
        batch_data = np.random.uniform(-10, 10, size=batch_size)

        # Calculate the gradient using the batch data
        gradient = np.mean([gradient_func(data_point) for data_point in batch_data])

        # Update the position using the gradient and learning rate
        position -= learning_rate * gradient
    return position
# Example usage:
def quadratic_function(x):
    return 2 * x - 4 # Gradient of the function 2x^2 - 4x
initial_position = 5.0 # Initial position of the optimization process
final_position_bgd = batch_gradient_descent(quadratic_function, initial_position)
print("Optimal solution using Batch Gradient Descent:", final_position_bgd)
```

```
Optimal solution using Batch Gradient Descent: 9.294998124226744
```

2. Stochastic Gradient Descent:

```
import random

def stochastic_gradient_descent(gradient_func, initial_position, learning_rate=0.01, num_iterations=100):
    position = initial_position

    for _ in range(num_iterations):
        # Randomly select a data point (in this case, only one data point)
        random_data_point = random.uniform(-10, 10)
        gradient = gradient_func(random_data_point)
        position -= learning_rate * gradient


    return position


# Example usage:
def quadratic_function(x):
    return 2 * x - 4  # Gradient of the function 2x^2 - 4x


initial_position = 0  # Initial position of the optimization process
final_position_sgd = stochastic_gradient_descent(quadratic_function, initial_position)
print("Optimal solution using Stochastic Gradient Descent:", final_position_sgd)
```

```
Optimal solution using Stochastic Gradient Descent: 4.575760522917483
```

3. Mini-batch Gradient Descent:

```python
import numpy as np

def mini_batch_gradient_descent(gradient_func, initial_position, learning_rate=0.01, num_iterations=100, batch_size=10):
    position = initial_position
    for _ in range(num_iterations):
        # Generate a mini-batch of random data points
        mini_batch_data = np.random.uniform(-10, 10, size=batch_size)

        # Calculate the gradient using the mini-batch data
        gradient = np.mean([gradient_func(data_point) for data_point in mini_batch_data])

        # Update the position using the gradient and learning rate
        position -= learning_rate * gradient
    return position

# Example usage:
def quadratic_function(x):
    return 2 * x - 4 # Gradient of the function 2x^2 - 4x
initial_position = 5.0 # Initial position of the optimization process
final_position_mbgd = mini_batch_gradient_descent(quadratic_function, initial_position)
print("Optimal solution using Mini-Batch Gradient Descent:", final_position_mbgd)
```

```
Optimal solution using Mini-Batch Gradient Descent: 8.679812124302996
```