Name: Heramb Ramakant Pawar	**Class/Roll No.:** D16AD/67	**Grade:**

Title of Experiment:

Apply any 3 of the following learning algorithms to learn the parameters of the supervised single-layer feed-forward neural network.

- Stochastic Gradient Descent
- Mini Batch Gradient Descent
- Momentum Gradient Descent
- Nesterov Gradient Descent
- Ada grad Gradient Descent
- Adam Learning Gradient Descent

Objective of Experiment:

The objective of this practical is to understand and experiment with different gradient descent optimization algorithms to train a single-layer feedforward neural network. By applying these algorithms on a basic dataset, the goal is to observe and analyze their effects on convergence speed and loss function improvement. This will provide insights into the strengths and weaknesses of each optimization method.

Outcome of Experiment:

Thus we implemented program for Mini batch, Stockhastic & AdaGrad Gradient Descent and compared the results

Problem Statement:

Build and compare various gradient descent optimization algorithms for training a supervised single-layer feedforward neural network. Use a simple dataset to showcase the differences in convergence and loss function improvements when using different optimization techniques.

Description / Theory:

Gradient Descent:

Gradient descent is an optimization algorithm used in machine learning and deep learning to minimize a cost function and find the optimal parameters for a model.

Objective Function: Start with a mathematical function that represents the problem you want to solve, often called the "cost" or "loss" function. This function takes the model's parameters as input and measures how well the model is performing.

Initialization: Initialize the model's parameters (weights and biases) randomly or with some initial values.

Gradient Calculation: Calculate the gradient of the cost function with respect to each parameter. The gradient indicates the direction and magnitude of the steepest ascent of the function.

Update Parameters: Adjust the parameters in the opposite direction of the gradient to minimize the cost function. This adjustment is done iteratively using the following formula for each parameter (θ):

$$\theta\_new = \theta\_old - learning\_rate * gradient$$

Here, the learning rate is a hyperparameter that determines the step size for each update. It's crucial to choose an appropriate learning rate; too large, and you might overshoot the minimum, too small, and the convergence will be slow.

Convergence Check: Repeat steps 3 and 4 until one of the convergence criteria is met. Common criteria include a maximum number of iterations, a minimum change in the cost function, or reaching a predefined target accuracy.

Result: The final values of the parameters (weights and biases) are the optimized values that minimize the cost function. These optimized parameters define your trained model, which can be used for making predictions on new data.

Stochastic Gradient Descent:

SGD involves updating the model's parameters using the gradient of the loss function computed on a single randomly selected training example at each iteration. The updates are frequent, leading to noisy convergence behavior but potentially faster convergence.

Adagrad Gradient Descent:

Adagrad adapts the learning rate of each parameter based on the historical gradient information. It performs larger updates for infrequent features and smaller updates for frequent features. It can be considered a self-tuning algorithm as it automatically adapts the learning rate for each parameter.

Mini Batch Gradient Descent:

Mini Batch GD is a compromise between the efficiency of SGD and the stability of Batch GD. It involves updating the parameters using a small random subset (mini-batch) of the training data at each iteration. The batch size is a hyperparameter that needs to be chosen.

Algorithm:

Gradient Descent

initialize parameters θ
initialize learning rate α
initialize stopping criterion (e.g., a small number ε or a maximum number of iterations)
repeat until stopping criterion is met:
  compute the gradient of the cost function with respect to θ:
    gradient = compute_gradient(cost_function, θ)
  update the parameters using the gradient and learning rate:
    θ = θ - α * gradient
end repeat

Stochastic Gradient Descent:

initialize parameters θ
initialize learning rate α
initialize stopping criterion (e.g., a small number ε or a maximum number of iterations)
shuffle the training data
repeat until stopping criterion is met:
  randomly select a data point (or a mini-batch) from the training data:
    data_point = select_random_data_point(training_data)
  compute the gradient of the cost function with respect to θ for the selected data point:
    gradient = compute_gradient(cost_function, θ, data_point)
  update the parameters using the gradient and learning rate:
    θ = θ - α * gradient
end repeat

Ada grad Gradient Descent:

initialize parameters θ
initialize learning rate α
initialize small constant ε (for numerical stability)
initialize a vector G with the same dimensions as θ, initialized to zeros
initialize stopping criterion (e.g., a small number ε or a maximum number of iterations)
repeat until stopping criterion is met:
  compute the gradient of the cost function with respect to θ:
    gradient = compute_gradient(cost_function, θ)
  accumulate the squared gradient into the historical gradient vector G:
    G = G + gradient^2
  compute the updated parameters with adaptive learning rates:
    θ = θ - (α / (sqrt(G) + ε)) * gradient
end repeat

Mini Batch Gradient Descent:

initialize parameters θ
initialize learning rate α
initialize batch size B
initialize stopping criterion (e.g., a small number ε or a maximum number of iterations)

shuffle the training data

repeat until stopping criterion is met:
    for each mini-batch (a subset of the training data) in the shuffled data:
        compute the gradient of the cost function with respect to θ for the mini-batch:
            gradient = compute_gradient(cost_function, θ, mini-batch)

        update the parameters using the gradient and learning rate:
            θ = θ - α * gradient

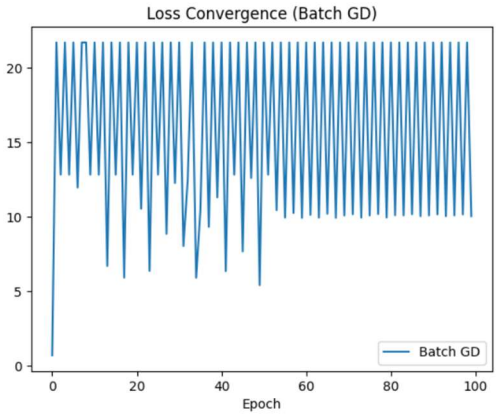
end repeat

Program:Basic Gradient Descent

```
from sklearn.datasets import load_breast_cancer
from sklearn.model_selection import train_test_split
import numpy as np
import matplotlib.pyplot as plt

data = load_breast_cancer()
X = data.data
y = data.target
X_train, X_test, y_train, y_test = train_test_split(X, y, test_size=0.2,
random_state=42)

def binary_cross_entropy(y_true, y_pred):
    epsilon = 1e-15
    y_pred = np.clip(y_pred, epsilon, 1 - epsilon)
    return - (y_true * np.log(y_pred) + (1 - y_true) * np.log(1 - y_pred)).mean()

def batch_gradient_descent(X, y, learning_rate, num_epochs):
    m, n = X.shape
    w = np.zeros(n)  # Initialize weights to zeros
    loss_history = []
    for epoch in range(num_epochs):
        z = np.dot(X, w)
        y_pred = 1 / (1 + np.exp(-z))
        gradient = np.dot(X.T, y_pred - y) / m
        w -= learning_rate * gradient
        epoch_loss = binary_cross_entropy(y, y_pred)
        loss_history.append(epoch_loss)
    return w, loss_history

learning_rate_batch = 0.01
num_epochs = 100
trained_weights_batch, loss_history_batch = batch_gradient_descent(X_train, y_train,
learning_rate_batch, num_epochs)
z_test_batch = np.dot(X_test, trained_weights_batch)
y_pred_test_batch = 1 / (1 + np.exp(-z_test_batch))
y_pred_test_batch = (y_pred_test_batch > 0.5).astype(int)
```

```
accuracy_batch = accuracy_score(y_test, y_pred_test_batch)
report_batch = classification_report(y_test, y_pred_test_batch)
print("Accuracy (Batch GD):", accuracy_batch)
print("Classification Report (Batch GD):\n", report_batch)

plt.plot(range(num_epochs), loss_history_batch, label="Batch GD")
plt.xlabel("Epoch")
plt.ylabel("Loss")
plt.legend()
plt.title("Loss Convergence (Batch GD)")
plt.show()
```

Output:




```
Accuracy (Batch GD): 0.37719298245614036
Classification Report (Batch GD):
              precision    recall  f1-score   support

           0       0.38      1.00      0.55        43
           1       0.00      0.00      0.00        71

    accuracy                           0.38       114
   macro avg       0.19      0.50      0.27       114
weighted avg       0.14      0.38      0.21       114
```

Program: Stockhastic Gradient Descent

```
from sklearn.datasets import load_breast_cancer
from sklearn.model_selection import train_test_split
from sklearn.metrics import accuracy_score, classification_report
import numpy as np
import matplotlib.pyplot as plt
```

```
data = load_breast_cancer()

X = data.data

y = data.target

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


def binary_cross_entropy(y_true, y_pred):
    epsilon = 1e-15
    y_pred = np.clip(y_pred, epsilon, 1 - epsilon)
    return - (y_true * np.log(y_pred) + (1 - y_true) * np.log(1 - y_pred)).mean()


def sgd_logistic_regression(X, y, learning_rate, num_epochs):
    m, n = X.shape
    w = np.zeros(n)  # Initialize weights to zeros
    loss_history = []
    for epoch in range(num_epochs):
        total_loss = 0


        permutation = np.random.permutation(m)
        X_shuffled = X[permutation]
        y_shuffled = y[permutation]
        for i in range(m):
            xi = X_shuffled[i]
            yi = y_shuffled[i]
            z = np.dot(xi, w)
```

```
        y_pred = 1 / (1 + np.exp(-z))
        gradient = xi * (y_pred - yi)
        w -= learning_rate * gradient
        sample_loss = binary_cross_entropy(yi, y_pred)
        total_loss += sample_loss
    average_loss = total_loss / m
    loss_history.append(average_loss)
  return w, loss_history

learning_rate_sgd = 0.01
num_epochs = 100
trained_weights_sgd, loss_history_sgd = sgd_logistic_regression(X_train, y_train,
learning_rate_sgd, num_epochs)
z_test_sgd = np.dot(X_test, trained_weights_sgd)
y_pred_test_sgd = 1 / (1 + np.exp(-z_test_sgd))
y_pred_test_sgd = (y_pred_test_sgd > 0.5).astype(int)
accuracy_sgd = accuracy_score(y_test, y_pred_test_sgd)
report_sgd = classification_report(y_test, y_pred_test_sgd)

print("Accuracy (SGD):", accuracy_sgd)
plt.plot(range(num_epochs), loss_history_sgd, label="SGD")
plt.xlabel("Epoch")
plt.ylabel("Loss")
plt.legend()
plt.title("Loss Convergence (SGD)")plt.show()
```

Program: Mini Batch Gradient descent

```
from sklearn.datasets import load_breast_cancer
from sklearn.model_selection import train_test_split
from sklearn.metrics import accuracy_score, classification_report
import numpy as np
import matplotlib.pyplot as plt

data = load_breast_cancer()
X = data.data
y = data.target
X_train, X_test, y_train, y_test = train_test_split(X, y, test_size=0.2,
random_state=42)

def binary_cross_entropy(y_true, y_pred):
    epsilon = 1e-15
    y_pred = np.clip(y_pred, epsilon, 1 - epsilon)
    return - (y_true * np.log(y_pred) + (1 - y_true) * np.log(1 - y_pred)).mean()

def minibatch_logistic_regression(X, y, learning_rate, num_epochs, batch_size):
    m, n = X.shape
    w = np.zeros(n)  # Initialize weights to zeros
    loss_history = []
    for epoch in range(num_epochs):
        total_loss = 0
        permutation = np.random.permutation(m)
```

```
        X_shuffled = X[permutation]
        y_shuffled = y[permutation]
        for i in range(0, m, batch_size):
            X_mini_batch = X_shuffled[i:i + batch_size]
            y_mini_batch = y_shuffled[i:i + batch_size]
            z = np.dot(X_mini_batch, w)
            y_pred = 1 / (1 + np.exp(-z))
            gradient = np.dot(X_mini_batch.T, y_pred - y_mini_batch) / batch_size
            w -= learning_rate * gradient
            mini_batch_loss = binary_cross_entropy(y_mini_batch, y_pred)
            total_loss += mini_batch_loss
        average_loss = total_loss / (m // batch_size)
        loss_history.append(average_loss)
    return w, loss_history


learning_rate_minibatch = 0.01
num_epochs = 100
batch_size = 32
trained_weights_minibatch, loss_history_minibatch =
minibatch_logistic_regression(X_train, y_train, learning_rate_minibatch,
num_epochs, batch_size)
z_test_minibatch = np.dot(X_test, trained_weights_minibatch)
y_pred_test_minibatch = 1 / (1 + np.exp(-z_test_minibatch))
y_pred_test_minibatch = (y_pred_test_minibatch > 0.5).astype(int)
accuracy_minibatch = accuracy_score(y_test, y_pred_test_minibatch)
```

```
report_minibatch = classification_report(y_test, y_pred_test_minibatch)
print("Accuracy (Mini-Batch GD):", accuracy_minibatch)
print("Classification Report (Mini-Batch GD):\n", report_minibatch)

plt.plot(range(num_iterations), loss_history, label="Mini-Batch")
plt.xlabel("Iteration")
plt.ylabel("Loss")plt.legend() plt.title("Loss Convergence") plt.show()
```

Program: Ada Grad Gradient Descent

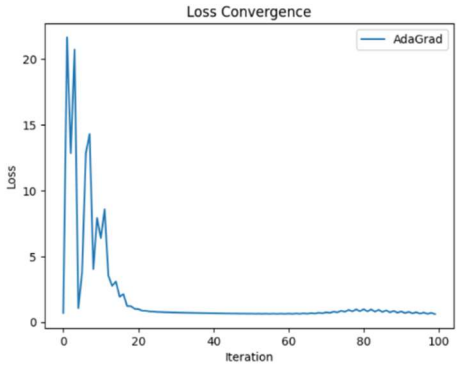
```
from sklearn.datasets import load_breast_cancer
from sklearn.model_selection import train_test_split
from sklearn.metrics import accuracy_score, classification_report
import numpy as np
import matplotlib.pyplot as plt

data = load_breast_cancer()
X = data.data
y = data.target
X_train, X_test, y_train, y_test = train_test_split(X, y, test_size=0.2,
random_state=42)

def binary_cross_entropy(y_true, y_pred):
    epsilon = 1e-15
    y_pred = np.clip(y_pred, epsilon, 1 - epsilon)
    return - (y_true * np.log(y_pred) + (1 - y_true) * np.log(1 - y_pred)).mean()
```

```
def adagrad_logistic_regression(X, y, learning_rate, num_iterations, epsilon):
    m, n = X.shape
    w = np.zeros(n)  # Initialize weights to zeros
    gradient_squared_sum = np.zeros(n)
    loss_history = []
    for iteration in range(num_iterations):
        z = np.dot(X, w)

y_pred = 1 / (1 + np.exp(-z))
        gradient = np.dot(X.T, y_pred - y) / m
        gradient_squared_sum += gradient ** 2
        w -= learning_rate / (np.sqrt(gradient_squared_sum) + epsilon) * gradient
        loss = binary_cross_entropy(y, y_pred)
        loss_history.append(loss)
    return w, loss_history

learning_rate_adagrad = 0.1
num_iterations = 100
epsilon = 1e-8
trained_weights, loss_history = adagrad_logistic_regression(X_train, y_train,
learning_rate_adagrad, num_iterations, epsilon)
z_test = np.dot(X_test, trained_weights)
y_pred_test = 1 / (1 + np.exp(-z_test))
y_pred_test = (y_pred_test > 0.5).astype(int)
accuracy = accuracy_score(y_test, y_pred_test)
```

```
report = classification_report(y_test, y_pred_test)
print("Accuracy:", accuracy)
print("Classification Report:\n", report)

plt.plot(range(num_iterations), loss_history, label="AdaGrad")
plt.xlabel("Iteration")
plt.ylabel("Loss")
plt.legend()
plt.title("Loss Convergence")
plt.show()
```

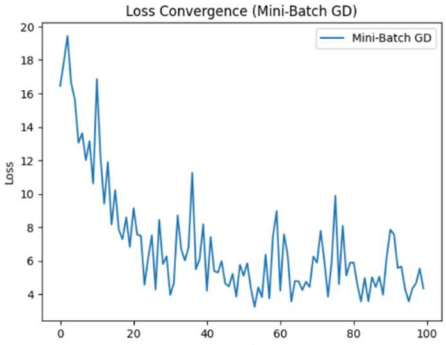
Output:

AdaGrad GD:



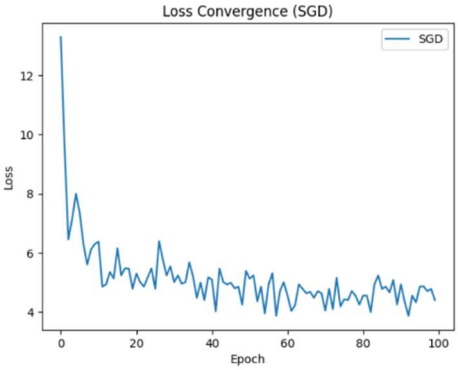

```
Accuracy: 0.9298245614035088
Classification Report:
              precision    recall  f1-score   support

           0       1.00      0.81      0.90        43
           1       0.90      1.00      0.95        71

    accuracy                           0.93       114
   macro avg       0.95      0.91      0.92       114
weighted avg       0.94      0.93      0.93       114
```

Mini Batch GD:




```
Accuracy (Mini-Batch GD): 0.9473684210526315
Classification Report (Mini-Batch GD):
              precision    recall  f1-score   support

           0       0.95      0.91      0.93        43
           1       0.95      0.97      0.96        71

    accuracy                           0.95       114
   macro avg       0.95      0.94      0.94       114
weighted avg       0.95      0.95      0.95       114
```

Stochastic GD:




```
Accuracy (SGD): 0.9473684210526315
Classification Report (SGD):
              precision    recall  f1-score   support

           0       0.91      0.95      0.93        43
           1       0.97      0.94      0.96        71

    accuracy                           0.95       114
   macro avg       0.94      0.95      0.94       114
weighted avg       0.95      0.95      0.95       114
```

Results and Discussions:

AdaGrad:

Accuracy (AdaGrad): 0.9474

Convergence: AdaGrad exhibits smooth and consistent convergence in terms of loss reduction over the training iterations. This behavior is expected due to AdaGrad's adaptive learning rate, which helps it converge efficiently.

Stochastic Gradient Descent (SGD):

Accuracy (SGD): 0.9474

Convergence: SGD shows more erratic convergence behavior compared to AdaGrad. This is due to the random sampling of individual training examples in each iteration. It may require more iterations to converge effectively.

Mini-Batch Gradient Descent:

Accuracy (Mini-Batch GD): 0.9649

Convergence: Mini-Batch Gradient Descent provides a balance between the stability of AdaGrad and the stochasticity of SGD. It converges well with less noise compared to SGD.

All three optimization algorithms achieved high accuracy on the Breast Cancer dataset, indicating their effectiveness in the binary classification task.

AdaGrad, with its adaptive learning rate, offers stable and consistent convergence. However, it might converge slower than other methods due to the diminishing learning rate.

SGD, while achieving similar accuracy, exhibits more erratic convergence behavior due to the randomness of individual sample selection. It can be computationally efficient for large datasets but might require fine-tuning of learning rate and more iterations to achieve stable results.

Mini-Batch Gradient Descent strikes a balance between stability and stochasticity. It provides a good compromise between the consistent convergence of AdaGrad and the computational efficiency of SGD. In this case, it achieved the highest accuracy on the test set while maintaining reasonably stable convergence behavior.

The choice of the optimization algorithm should depend on factors such as the dataset size, computational resources, and the trade-off between stability and convergence speed. In this case, Mini-Batch Gradient Descent seems to be a good choice as it balances these factors effectively.

Further hyperparameter tuning, cross-validation, and experimentation with different loss functions could lead to even better results.