Name: Subrato Tapaswi	Class/Roll No.: D16AD/60	Grade:

Title of Experiment: Learn the parameters of the supervised single layer feed forward neural network.

Objective of Experiment:
Understanding different gradient descent algorithms is crucial for efficient optimization in machine learning. Variants like batch, stochastic, and mini-batch gradient descent offer trade-offs in convergence speed, stability, memory efficiency, and adaptation. This knowledge aids in selecting suitable optimization strategies, enhancing model training, and managing hyperparameters for improved performance.

Outcome of Experiment: Comprehending diverse gradient descent techniques is vital for efficient machine learning optimization. Batch, stochastic, and mini-batch variants balance convergence speed, stability, and memory efficiency, enhancing model training and hyperparameter management.

Problem Statement: To apply the following learning algorithms to learn the parameters of the supervised single layer feed forward neural network (Stochastic Gradient Descent, Mini Batch Gradient Descent and Adam Learning GD).

Description / Theory:
Gradient:
Gradient in plain terms means slope or slant of a surface. So gradient descent literally means descending a slope to reach the lowest point on that surface.
Gradient descent is an iterative algorithm that starts from a random point on a function and travels down its slope in steps until it reaches the lowest point of that function.

This algorithm is useful in cases where the optimal points cannot be found by equating the slope of the function to 0. In the case of linear regression, the sum of squared residuals as the function “y” and the weight vector as “x” in theparabola above.

Stochastic Gradient Descent:

Gradient descent has some problems, especially when it comes to how much math it does in each step.

Stochastic Gradient Descent, or SGD for short, is a tweaked version of gradient descent used in machine learning. It helps solve the math problem when you have lots of data.

In SGD, instead of using all your data for every step, you just pick one random piece (or a small group) to do the math on. This randomness is why it's called "stochastic". It makes things faster and works well for big datasets in machine learning.

Mini Batch Gradient Descent:

Mini-batch gradient descent is a clever twist on the gradient descent method. Instead of using the entire training dataset at once, it breaks it into smaller groups, called mini-batches. These mini-batches are used to figure out how wrong the model is and to make it better.

Sometimes, instead of just using one mini-batch, you can add up the results from several mini-batches to make things even more stable.

Mini-batch gradient descent tries to find a middle ground between two other methods: one that's strong but slow and another that's fast but not so reliable. It's the most commonly used method in deep learning because it strikes a good balance between getting good results and being efficient.

Adam Learning Gradient Descent:

Gradient descent is a method to find the lowest point of a function. It does this by going in the opposite direction of the slope. However, it has a drawback: it uses the same step size (learning rate) for all parts of the problem, which can be a problem when different parts need different step sizes.

To address this, there are improved versions like AdaGrad and RMSProp that adjust the step size for each part, but they can sometimes make the step size too small too quickly.

Adam is a newer and smarter version of gradient descent. It not only adjusts the step size for each part but also uses a special method to make the search for the lowest point smoother. It's like having a GPS that not only adjusts the speed of your car for each road but also anticipates the turns ahead to give you a smoother and faster journey.

Algorithm:

Stochastic Gradient Descent:

Initialize:
  Choose initial values for the model parameters (weights) θ
  Set learning rate α
  Set number of epochs or iterations

For epoch = 1 to number of epochs:
  Randomly shuffle the training dataset

  For each training example (x, y) in the shuffled dataset:
    Compute the gradient of the loss function with respect to the parameters: $\nabla\theta$ Loss(x, y, θ)
    Update the parameters using the gradient and learning rate:
      $\theta = \theta - \alpha * \nabla\theta$ Loss(x, y, θ)

  Calculate the average loss over the entire dataset for this epoch:
    epoch_loss = TotalLoss / Number of training examples

  Print or record the epoch number and the corresponding epoch_loss

Mini Batch Gradient Descent:

Initialize:
  Choose initial values for the model parameters (weights) θ
  Set learning rate α
  Set batch size (mini-batch size)
  Set number of epochs or iterations
For epoch = 1 to number of epochs:
  Randomly shuffle the training dataset
  Divide the training dataset into mini-batches of size batch_size

For each mini-batch (mini_x, mini_y) in the divided dataset:
Compute the gradient of the loss function with respect to the parameters: $\nabla\theta$ Loss(mini_x, mini_y, θ)
Update the parameters using the gradient and learning rate:
$\theta = \theta - \alpha * \nabla\theta$ Loss(mini_x, mini_y, θ)

Calculate the average loss over the entire dataset for this epoch:
epoch_loss = TotalLoss / Number of training examples

Print or record the epoch number and the corresponding epoch_loss

Adam Learning Gradient Descent:

Initialize:
Choose initial values for the model parameters (weights) θ
Initialize first moment estimate m = 0
Initialize second moment estimate v = 0
Set learning rate α
Set $\beta1$ (exponential decay rate for the first moment estimate)
Set $\beta2$ (exponential decay rate for the second moment estimate)
Set ε (small constant to prevent division by zero)
Set t = 0 (time step)

For epoch = 1 to number of epochs:
Randomly shuffle the training dataset
For each training example (x, y) in the shuffled dataset:
Increment time step t: t = t + 1

Compute the gradient of the loss function with respect to the parameters: $\nabla\theta$ Loss(x, y, θ)
Update the first moment estimate:
$m = \beta1 * m + (1 - \beta1) * \nabla\theta$ Loss(x, y, θ)
Update the second moment estimate:
$v = \beta2 * v + (1 - \beta2) * (\nabla\theta \text{ Loss}(x, y, \theta))^2$
Correct for bias in the first and second moment estimates:
$m\_hat = m / (1 - \beta1^t)$

$v\_hat = v / (1 - \beta2^t)$
Update the parameters using the corrected moments, learning rate, and epsilon:
$\theta = \theta - \alpha * m\_hat / (\text{sqrt}(v\_hat) + \varepsilon)$
Calculate the average loss over the entire dataset for this epoch:
epoch_loss = TotalLoss / Number of training examples
Print or record the epoch number and the corresponding epoch_loss

Program:

1. **Programs on Basic programming constructs like branching and looping.**

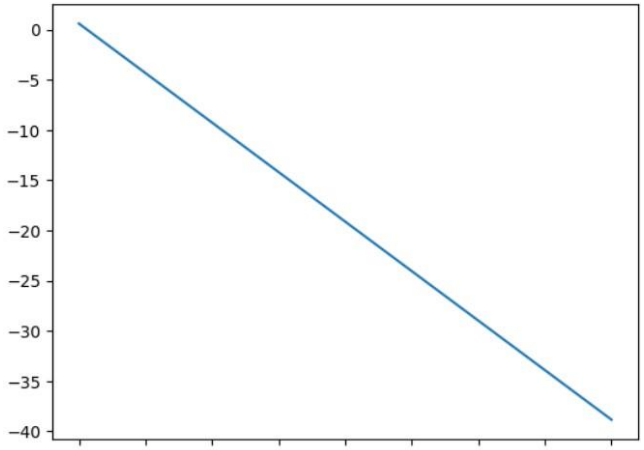
```
cost_history=[]
theta0_history=[]
theta1_history=[]
for e in range(epochs):
    for j in range(0,m,batches):
        x_batch=x[j:j+batches]
        y_batch=y[j:j+batches]
        h = theta0 + theta1*x_batch
        cost = sum([error**2 for error in (h-y_batch)]) / 2*m
        cost_history.append(cost)

        theta0 = theta0 - (learning_rate * np.sum(h-y_batch) / m)
        theta1 = theta1 - (learning_rate * (np.sum((h-y_batch)*x_batch) / m))

        theta0_history.append(theta0)
        theta1_history.append(theta1)
return theta0_history,theta1_history,cost_history


theta0_history,theta1_history,cost_history=GD_single_Mini_Patches_LR(x,y,1000,0.004,10)
h= theta0_history[-1] + theta1_history[-1]*x
```

Output Screenshots :




```
r2_score(y, h)

    0.9996931809799962
```

2. **Programs on Basic programming constructs like branching and looping.**

```
def Stochastic_GD(x,y,learning_rate,epochs):
    theta0=0
    theta1=0
    m=float(len(y))
    cost_history=[]
    hypothesis=[]
    theta_0_history=[]
    theta_1_history=[]
    for i in range(epochs):
        h = theta0 + theta1*x
        cost = sum([data**2 for data in (h-y)]) / 2*m
        cost_history.append(cost)
        hypothesis.append(h)
        theta_0_history.append(theta0)
        theta_1_history.append(theta1)

        theta0 = theta0 - (learning_rate * (h-y) / m)
        theta1 = theta1 - (learning_rate * (((h-y)*x) / m))
    return theta0,theta1,cost_history,hypothesis,theta_0_history,theta_1_history
```

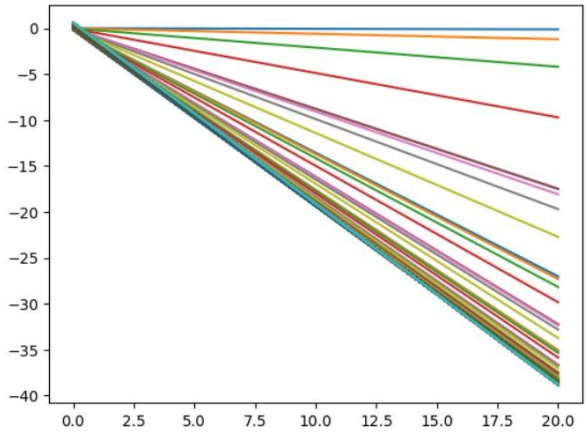
Output Screenshots:

```
r2_score(y, hypothesis[-1])
```

```
0.9964938986390629
```

```
fig, ax = plt.subplots()
for t0,t1 in zip(theta0_history,theta1_history):
    plt.plot(x,x*t1+t0)

plt.show()
```



3. Programs on Basic programming constructs like branching and looping.

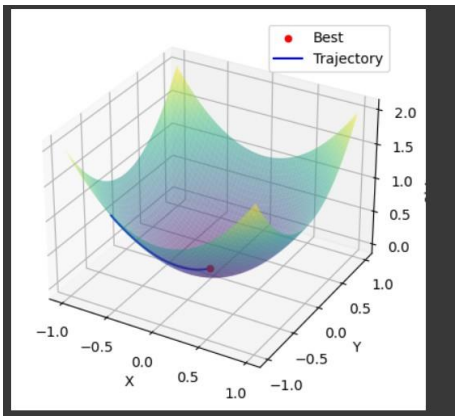
```
def adam(objective, derivative, bounds, n_iter,
    alpha, beta1, beta2, eps=1e-8):
  # Generate an initial point
  x = bounds[:, 0] + np.random.rand(len(bounds))\
  * (bounds[:, 1] - bounds[:, 0])
  scores = []
  trajectory = []

  # Initialize first and second moments
  m = np.zeros(bounds.shape[0])
  v = np.zeros(bounds.shape[0])

  # Run the gradient descent updates
  for t in range(n_iter):
    # Calculate gradient g(t)
    g = derivative(x[0], x[1])

    # Build a solution one variable at a time
    for i in range(x.shape[0]):
      # m(t) = beta1 * m(t-1) + (1 - beta1) * g(t)
      m[i] = beta1 * m[i] + (1.0 - beta1) * g[i]
      # v(t) = beta2 * v(t-1) + (1 - beta2) * g(t)^2
      v[i] = beta2 * v[i] + (1.0 - beta2) * g[i] ** 2
      # mhat(t) = m(t) / (1 - beta1(t))
      mhat = m[i] / (1.0 - beta1 ** (t + 1))
      # vhat(t) = v(t) / (1 - beta2(t))
      vhat = v[i] / (1.0 - beta2 ** (t + 1))
      # x(t) = x(t-1) - alpha * mhat(t) / (sqrt(vhat(t)) + eps)
      x[i] = x[i] - alpha * mhat / (np.sqrt(vhat) + eps)
```

Output Screenshots:



Results and Discussions:

We have understood the working of different types of gradient descent algorithms. Learnt the parameters of the supervised single layer feed forward neural network using different learning algorithms and successfully implemented them.