Gradient Update Methods:

1. Batch Gradient Descent (Vanilla Gradient Descent Method)

Solution:

Vanilla gradient method computes the gradient of the cost function with respect to the parameter θ for the entire training dataset. Basic step of gradient descent is as follow [1]:

$$\theta_{new} = \theta_{old} - \alpha * \frac{d(Loss_{function})}{d\theta}$$

Let us say the we have our objective function which is F(x) and set us say we have your Regression gradient as θ then the gradient update would take place after the entire training dataset is iterated at least once.

Below image gives the Pseudo code of the gradient descent algorithm:

Pseudocode:

while loss_function_old – loss_function_new <= precision:
    for j in range(len(X_train)):
        *hypothesis = X_train[j]*theta + bias*
        activation = sigmoid(hypothesis)
    loss_function = log loss(activation) + regularization factor
    #Calculating gradient:
    $\frac{d(loss_{function})}{d(theta)} = d(log_{loss}(activation))/d(theta)$
    $theta_{new} = theta_{old} - \alpha * \frac{d(loss_{function})}{dtheta}$

Legends:

- X_train: - training data
- Theta = regression parameter
- alpha = learning rate
- precision = any difference in loss_function applicable to functionality

2. Stochastic Gradient Descent

Solution:

Stochastic gradient descent works in contrast performs a parameter update for each training example $x^{(i)}\ and\ y^{(i)}$ labels [1]:

The basic step of Stochastic gradient descent is as follow as:

$$\theta = \theta - \alpha * \frac{d(Loss_{function})}{d\theta}$$

Stochastic gradient descent performs redundant computations for large datasets, as it re computes gradients for similar examples before each parameter update. SGD does away with this redundancy by performing one update at a time. It is therefore usually much faster and can also be used to learn online

Pseudocode:

while loss_function_old – loss_function_new <= precision:
    for j in range(len(X_train)):
        *hypothesis = X_train[j]*theta + bias*
        activation = sigmoid(hypothesis)
        loss_function = log loss(activation) + regularization factor
        #Calculating gradient of smooth convex function:
        $\frac{d(loss_{function})}{d(theta)} = d(log_{loss}(activation))/d(theta)$
        $theta_{new} = theta_{old} - \alpha * \frac{d(loss_{function})}{dtheta}$

Legends:

- X_train: - training data
- Theta = regression parameter
- alpha = learning rate
- precision = any difference in loss_function applicable to functionality

3. Mini-batch gradient descent:

Mini-batch gradient is a combination of both vanilla gradient descent and stochastic gradient, it updates the gradient after every batch of the training data, the batches are usually of size 50 to 256, thus it studies the data over a period of batch training and updates the gradient after that [1].

The mini batch gradient has following advantages

- Reduces the variation in the parameters updates, which leads to better convergences
- Very efficient for highly optimized matrix optimizations and used in deep learning libraries

Pseudocode:

while loss_function_old – loss_function_new <= precision:
S = 0
    for i in range (0, len(X_train), batch: [50 to 256])):
        for j in range (S, S+batch)
        *hypothesis = X_train[j]*theta + bias*
        activation = sigmoid(hypothesis)
    loss_function = log loss(activation) + regularization factor
    #Calculating gradient of smooth convex function:
    $\frac{d(loss_{function})}{d(theta)} = d(log_{loss}(activation))/d(theta)$
    $theta_{new} = theta_{old} - \alpha * \frac{d(loss_{function})}{dtheta}$

Legends:

- X_train = training data
- Theta = regression parameter
- alpha = learning rate
- precision = any difference in loss_function applicable to functionality
- batch = training data batch

4. Stochastic gradient descent with Momentum

The issue with the stochastic gradient descent is when the surface curves are more steeply in one dimension than in another, this condition occurs around the local optima. In these scenarios, SGD oscillates across the slopes of the ravine which furthers slows down the progress and takes time in reaching the convergence point[1].

Momentum helps in smoothing the convergence process in the stochastic gradient descent by adding a fraction γ of the update vector of the past time step to the current update vector [1].

Basic step of the momentum is as follow as:

$$\theta_t = \gamma\theta_{t-1} + \alpha \frac{d\left(Loss_{function}\right)}{d\theta}$$

$$\theta_{current} = \theta_{current} - \theta_t$$

Where γ is the momentum factor or term which is usually equal to 0.9

The basic pseudo code is same as the stochastic gradient descent however the gradient update would change

while loss_function_old – loss_function_new <= precision:
    for j in range(len(X_train)):
        *hypothesis = X_train[j]*theta + bias*
        activation = sigmoid(hypothesis)
        loss_function = log loss(activation) + regularization factor
        #Calculating gradient of smooth convex function:

$$\frac{d(loss_{function})}{d(theta)} = d(log_{loss}(activation))/d(theta)$$

$$theta_t = \gamma * theta_{t-1} + alpha * \frac{d\left(Loss_{function}\right)}{d(theta)}$$

$$theta_{current} = theta_{current} - theta_t$$

Legends:

- X_train = training data
- Theta = regression parameter
- alpha = learning rate
- precision = any difference in loss_function applicable to functionality
- γ = momentum term

5. Nesterov accelerated gradient method

Nesterov accelerated gradient method helps in identifying the direction of convergence and gives the momentum term notion of the direction in which it converges. The Nesterov accelerated is a combination of gradient descent algorithm and momentum hyperparameter which makes this a two-step update algorithm. Thus, in this update process the Nesterov gradient method, we calculate the gradient using normal gradient descent update step in 1st phase of the gradient method. While in the 2nd phase we multiply the momentum to the gradient found in step and add the old gradient calculated in previous step to give us a new gradient [1] &

The convergence rate of this algorithm for t steps is $1/t^2$.

Basic Update step of the Nesterov accelerated gradient method as follow as:

$$\theta_t = \gamma\theta_{t-1} + \alpha * \frac{d(Loss_{function})}{d(\theta - \gamma * \theta_{t-1})}$$

$$\theta_{current} = \theta_{current} - \theta_t$$

$$\alpha_t = \left(1 + \frac{\sqrt{1 + 4\alpha_{t-1}^2}}{2}\right)$$

The Pseudocode of the Nesterov' s gradient descent algorithm is as follow as:

while loss_function_old – loss_function_new <= precision:
    for j in range(len(X_train)):
        *hypothesis = X_train[j]*theta + bias*
        activation = sigmoid(hypothesis)
        loss_function = log loss(activation) + regularization factor
#Calculating gradient for the smooth convex function:

$$\frac{d(loss_{function})}{d(theta)} = d(log_{loss}(activation))/d(theta)$$

$$theta_t = \gamma * theta_{t-1} + alpha * \frac{d(Loss_{function})}{d(theta - theta_{t-1})}$$

$$theta_{current} = theta_{current} - theta_t$$

Legends:

- X_train = training data
- Theta = regression parameter
- alpha = learning rate
- precision = any difference in loss_function applicable to functionality
- γ = momentum term

6. Proximal Gradient Descent Algorithms

Proximal gradient descent helps in solving the convex optimization problem involving a smooth convex function and a non-smooth convex function. The proximal gradient descent is also called as composite gradient descent algorithm or a generalized gradient descent algorithm.

The basic form of loss function on which we apply proximal gradient descent is as follow as:

$$f(x) = g(x) + h(x)$$

Where $g(x)$ is a convex differentiable function and$h(x)$ is convex and not necessarily differentiable

Thus, according to the idea behind the quadratic approximation of function f around x, replacing the hessian of the matrix $\nabla^2 f(x)$ by $\frac{1}{t}(I)$

Basic minimization criterion would be

$$x^+ = argmin\, f(x) + \nabla f(x)^T(z - x) + \frac{1}{2t}\,||z - x||$$

Thus, in our case we have $g(x)$ as differentiable and $h(x)$ with is non- differentiable thus, we will take the quadratic approximation to g and leave h $alone$

Update

$$x^+ = argmin\, g_t(z) + h(z)$$

$$= \; argmin\, g(x) + \nabla g(x)^T(z - x) + \frac{1}{2t}\,\left||z - x|\right|_2^2 + h(z)$$

$$= \; argmin\, \frac{1}{2t}\,\left||z - (x - t\nabla \mathrm{g(x)}|\right|_2^2 + h(z)$$

The basic idea of proximal gradient descent:

$$prox_t(x) = argmin \frac{1}{2t}\,\left||x - z|\right|_2^2 + h(z)$$

Thus, we initialize x_0 and the repeat:

$$x^k = prox_{t_k}(x^{(k-1)} - t_k \nabla g(x^{k-1}))$$

We can make the above step look similar to gradient descent step as

$$x^k = x^{k-1} - t_k G_{t_k}\left(x^{k-1}\right)$$

Where G_{t_k} is the generalized gradient of f

$$G_t(x) = \frac{x - prox_t\left(x - t\nabla g(x)\right)}{t}$$

For example, on the proximal gradient descent algorithm and its thresholding algorithm kindly refer to slide at the below link

http://www.stat.cmu.edu/~ryantibs/convexopt/lectures/prox-grad.pdf

Advantage of $prox_t(x)$ is it is not dependent on $g(x)$ it only dependent on $h(x)$.

Smooth part of g can be complicated however our only concern there is that we should be able to calculate the gradient of that term.

The Pseudocode of the above gradient descent algorithm is as follow as:

```
while loss_function_old – loss_function_new <= precision:
        for j in range(len(X_train)):
                hypothesis = X_train[j]*theta + bias
                activation = sigmoid(hypothesis)
loss_function = log loss(activation) + regularization factor
```

#Calculating the gradient term of smooth part and using proximal mapping for the non-smooth part in order to update the overall gradient.

Let, say our loss function looks like something as follow as

$$f(\theta) = \frac{1}{2}\left||y - X\theta|\right|_2^2 + \lambda\left||\theta|\right|_1$$

Thus, we have $g(\theta) = \frac{1}{2}\left||y - X\theta|\right|_2^2$ *and* $h(\theta) = \lambda\left||\theta|\right|_1$

Thus, proximal gradient update step is

$$prox_t(\theta) = argmin_\theta(f(\theta))$$
$$= S_{\lambda_t}(\theta)$$

Thus, the thresholding operator here is given as

$[S_\lambda(\theta)] = \theta_i - \lambda \quad if\ \theta_i > \lambda$

$= \ 0 \quad if - \lambda \leq \ \theta_i \leq \ \lambda, for\ i = 1 \dots n$

$= \theta_i + \lambda \quad if\ \theta_i \leq -\lambda$

Thus, the above type of gradient update or soft thresholding operation is also called as Iterative soft-thresholding algorithm. We also have a fast-iterative soft thresholding algorithm which we would see later.

Reference:

1. https://arxiv.org/abs/1609.04747