Optimization

Optimization means finding the best possible solution from a set of feasible solutions. In machine learning, best usually means minimum error/loss.

$\min_{\theta} J(\theta)$; where $\theta \rightarrow$ model parameters

$J(\theta) \rightarrow$ objective function/ cost function

Gradient Descent Optimization

For simple problems, we could

$$\frac{dJ(\theta)}{d\theta} = 0,$$ and solve analytically

But in case of machine learning,

$J(\theta)$ is high dimentional and often non-linear.

So, iterative numerical optimization is used.

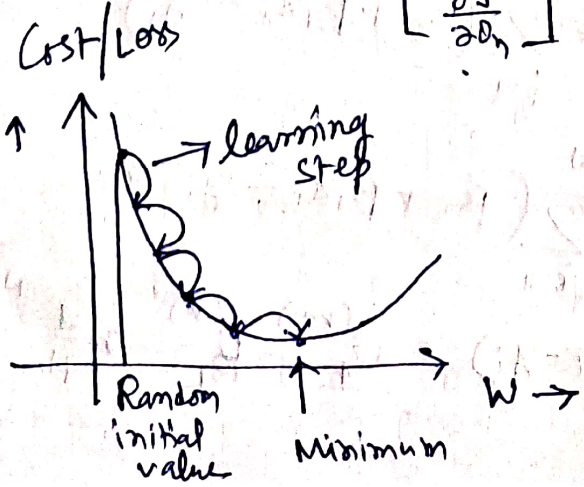
→ the gradient of a function:

$\nabla J(\theta)$ is a vector of partial derivatives

$$\nabla J(\theta) = \begin{bmatrix} \frac{\partial J}{\partial \theta} \\ \frac{\partial J}{\partial \theta_1} \\ \vdots \\ \frac{\partial J}{\partial \theta_n} \end{bmatrix}$$

Gives the direction of the steepest increase of the function.

∴ Opposite direction → steepest decrease



Formally writing the definition of Gradient Descent Algorithm,

Gradient Descent is one of the most fundamental optimization algorithms in machine learning. It is used to minimize a loss function by iteratively updating model parameters in the direction of steepest descent.

Mathematical idea,

$$\theta := \theta - \eta \nabla J(\theta)$$

η: learning rate. $\nabla J(\theta)$: gradient / direction of max^{m} increase

Getting back to Linear Regression Model,

$$\hat{y}_i = \beta_0 + \beta_1 x_i$$

$$J(\beta_0, \beta_1) = \frac{1}{2N} \sum_{i=1}^{N} (\hat{y}_i - y_i)^2$$

↑ for simplicity.

$$J(\beta_0, \beta_1) = \frac{1}{2N} \sum_{i=1}^{N} (\beta_0 + \beta_1 x_i - y_i)^2$$

Optimization goal, $\min_{\beta_0, \beta_1} J(\beta_0, \beta_1)$

$$\frac{\partial J}{\partial \beta_0} = \frac{1}{N} \sum_{i=1}^{N} (\beta_0 + \beta_1 x_i - y_i)$$

$$\frac{\partial J}{\partial \beta_1} = \frac{1}{N} \sum_{i=1}^{N} (\beta_0 + \beta_1 x_i - y_i) x_i$$

$$\beta_0^{(k+1)} = \beta_0^{(k)} - \eta \frac{1}{N} \sum_{i=1}^{N} (\beta_0 + \beta_1 x_i - y_i) \quad \beta_1^{(k+1)} = \beta_1^{(k)} - \eta \frac{1}{N} \sum_{i=1}^{N} (\beta_0 + \beta_1 x_i - y_i) x_i$$

↳ Stochastic gradient descent :- Computes the gradient of the cost function using only a single training example. in each iteration. faster convergence as the gradient is updated after each individual data point.

↳ Batch Gradient descent :- Computes the gradient of the cost function using the entire training dataset. for each iteration. This ensures that the computed gradient is precise, but it can be computationally expensive.

↳ Mini-batch Gradient descent :- Compromise between the above. Uses small batches of samples.

Loss function (Batch of size b)

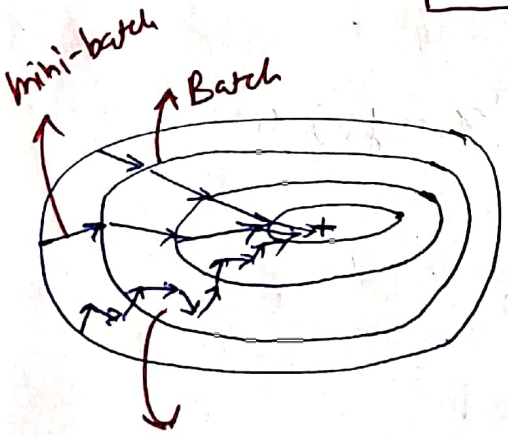
$$J_B(\theta) = \frac{1}{2b} \sum_{i \in B} (\theta^T x_i - y_i)^2$$

Gradient

$$\Delta J_B(\theta) = \frac{1}{b} \sum (\theta^T x_i - y_i) x_i$$

Gradient update rule :

$$\theta^{(k+1)} = \theta^{(k)} - \eta \frac{1}{b} \sum_{i \in B} (\theta^T x_i - y_i) x_i$$



#Momentum

Momentum is where an objects motion depends not only on the current force but also on its previous velocity. In the context of gradient optimization it refers to a method that smoothens the optimization trajectory by adding a term that helps the optimizer remember the past gradient.

∴ Current update = previous velocity + current gradient

$$v^{(k)} = \beta v^{(k-1)} + \nabla J(\theta^{(k)})$$ → Defining velocity.

$v \rightarrow$ velocity vector ; $\beta \in [0,1]$ = momentum coeff (usually 0.9)

parameter update

$$\theta^{(k+1)} = \theta^{(k)} - \eta v^{(k)}$$

$$\theta^{(k+1)} = \theta^{(k)} - \eta \left(\beta v^{(k-1)} + \nabla J(\theta^{(k)}) \right)$$

Momentum for Linear Regression (mini batch):

$$\nabla J_B(\theta) = \frac{1}{b} \left(\theta^T x_i - y_i \right) x_i$$

$$v^{(k)} = \beta v^{(k-1)} + \frac{1}{b} \sum_{i \in B} \left(\theta^T x_i - y_i \right) x_i$$

$$\theta^{(k+1)} = \theta^{(k)} - \eta v^{(k)}$$

Adaptive Gradient Algorithm

Features may have different scales, thus same learning rate for all features doesnot makes much sense. Let each feature has its own learning rate.

Adagrad scales the learning rate inversely with the square root of the sum of past squared gradients.

Let $g^{(k)} = \nabla J(\theta^{(k)})$ → Current gradient.

$r^{(k)}$ = accumulated squared gradients.

$$r^{(k)} = r^{(k-1)} + g^{(k)} \odot g^{(k)}$$

$\odot$ → element wise square.

$$\boxed{\theta^{(k+1)} = \theta^{(k)} - \frac{\eta}{\sqrt{r^{(k)}} + \epsilon} \odot g^{(k)}}$$

Parameters with larger accumulated gradients are effectively penalized in subsequent updates, while those with smaller accumulated gradients retain higher learning rates.

Adagrad for Linear Regression,

$$g^{(k)} = \frac{1}{b} \sum_{i \in B} (\theta^T x_i - y_i) x_i$$

Updates $r^{(k)} = r^{(k-1)} + g^{(k)} \odot g^{(k)}$

$$\theta^{(k+1)} = \theta^{(k)} - \frac{\eta}{\sqrt{r^{(k)}} + \epsilon} \odot g^{(k)}$$

$r^{(k)}$

#Root-mean square Propagation (RMS prop)

In Adagrad optimization algorithm, it accumulates the past squared gradients. RMSProp fixes this by using a moving average instead of a cumulative sum.

$$g_t = \nabla J(\theta_t)$$

s_t = exponentially weighted average of squared gradients.

$\rho \in (0,1)$ decay rate (usually 0.9)

$$s_t = \rho s_{t-1} + (1-\rho) g_t^2$$

$$\boxed{\theta_{t+1} = \theta_t - \frac{\eta}{\sqrt{s_t + \epsilon}} \odot g_t} \rightarrow \text{Update equation}$$

RMS prop for linear regression

$$g_t = \frac{1}{b} \sum_{i \in B} (\theta^T x_i - y_i)\, x_i$$

updates,

$$\boxed{s_t = \rho s_{t-1} + (1-\rho) g_t^2}$$

$$\boxed{\theta_{t+1} = \theta_t - \frac{\eta}{\sqrt{s_t + \epsilon}} \odot g_t}$$

RMS prop is an adaptive optimization algorithm that adjusts learning rate for each parameter by maintaining an exponentially decaying average of squared gradients, thereby preventing the aggressive learning rate decay observed in AdaGrad.

Suppose we update parameters as

$$\theta \leftarrow \theta - \eta g_t$$

if gradients are large, updates explodes whereas if gradients are small, learning is slow. Another thing is the learning rate is same in all directions.

Gradient tells us direction where to move and the magnitude tells how steep the surface is.

AdaGrad uses,

$$r_t = \sum_{k=1}^{t} g_k^2$$

Instead of remembering everything, remember only recent behaviour

$$s_t = \rho s_{t-1} + (1-\rho) g_t^2$$

Recent gradients → high weight

Old gradients → exponentially forgotten

It tracks local curvature

$$\boxed{s_t \approx E[g_t^2]}$$

Adam

Adam, Adaptive moment Estimation is an optimization algorithm that computes adaptive learning rates for each parameter by maintaining exponentially decaying averages for both first order (mean) and second order (variance) gradients, along with bias correction to ensure stable and fast convergence.

In SGD we have noisy and slow optimization; in case of momentum the optimization is fast but fixed L.R. AdaGrad is adaptive LR but it decays too fast.

Adam → Momentum + Adaptive Learning Rates.

Adam maintains two running averages,

first moment → mean of gradients (momentum)

second moment → mean of squared gradients (AdaGrad)

Hence Adaptive moment Estimation.

At iteration t, the gradient is $\boxed{g_t = \nabla J(\theta_t)}$

Learning rate (η); Exponential decay rates β_1, β_2

$$\boxed{m_t = \beta_1 m_{t-1} + (1-\beta_1) g_t}$$ Exponentially weighted moving avg. of gradients

Second moment estimation, (Adaptive Scaling)

$$\boxed{v_t = \beta_2 v_{t-1} + (1-\beta_2) g_t^2}$$ Exponentially — sq. gradients

$$\hat{m}_t = \frac{m_t}{1-\beta_1^t} \; ; \; \hat{v}_t = \frac{v_t}{1-\beta_2^t}$$

$$\boxed{\theta_{t+1} = \theta_t - \eta \frac{\hat{m}_t}{\sqrt{\hat{v}_t} + \epsilon}}$$ → Adam update equation

#Adadelta.

In Adagrad optimization learning rate decays to zero for large gradients. RMS prop tries to fix it by using moving average. But it still needs a manually chosen learning rate η.

Mathematical formulation,

$$g_t = \nabla J(\theta_t)$$

$E[g^2]_t$ = moving average of squared gradients.

$E[\Delta\theta]_t$ = moving averge of squared updates.

ρ = decay rate ($\simeq 0.9$)

Accumulating squared gradients like RMSProp

$$E[g^2]_t = \rho E[g^2]_{t-1} + (1-\rho) g_t^2$$

Compute parameter update,

$$\boxed{\Delta\theta_t = -\frac{\sqrt{E[\Delta\theta^2]_{t-1} + \epsilon}}{\sqrt{E[g^2]_t + \epsilon}} g_t}$$

Accumulate squared updates,

$$E[\Delta\theta^2]_t = \rho E[\Delta\theta^2]_{t-1} + (1-\rho)(\Delta\theta_t)^2$$

$$\boxed{\theta_{t+1} = \theta_t + \Delta\theta_t}$$

Adadelta adapts its learning rate by comparing how large updates should be based on past sucessful steps rather than relying on a manually chosen learning rate.

Q1 Optimize a simple function $f(x) = x^2$ using Batch Gradient Descent Algorithm. Consider $x_0 = 10$ and $\eta = 10$.

Soln

$$f(x) = x^2$$

Optimization problem $\min_x f(x) = x^2$.

$$\frac{df}{dx} = 2x$$

Batch Gradient Descent update rule,

$$x_{k+1} = x_k - \eta \frac{df}{dx}$$

$$x_{k+1} = x_k - 2\eta x_k$$

Initial values of x_0 is 10 and $\eta = 0.1$.

$$x_1 = x_0 - \eta \frac{df}{dx}$$

$$= 10 - (0.1)(2)(10) = 8.$$

$$x_2 = 8 - 2(0.1)(8) = 6.4$$

$$x_3 = 6.4 - 2(0.1)(6.4) = 6.4 - 1.28 = 5.12$$

$$x_4 = 5.12 - 2(0.1)(5.12) = 4.096.$$

Q2. Optimize $f(x,y) = x^2 + 10y^2$ using momentum. Consider $(x_0, y_0) = (5, 5)$; Initial velocity $(v_0) = (0, 0)$; learning rate = 0.05 and momentum $(\beta) = 0.9$

Soln

$f(x,y) = x^2 + 10y^2 \rightarrow$ function to be optimised

$$\nabla f(x,y) = \begin{bmatrix} 2x \\ 20y \end{bmatrix}$$

velocity equation in momentum,

$$v_{k+1} = \beta v_k + \nabla f(x_k, y_k)$$

$$(x_{k+1}, y_{k+1}) = (x_k, y_k) - \eta v_{k+1}$$

$$\therefore \quad v_1 = \beta v_0 + \nabla f(x_0, y_0)$$

$$= 0.9 \begin{bmatrix} 0 \\ 0 \end{bmatrix} + \begin{bmatrix} 2*5 \\ 20 \times 5 \end{bmatrix} = \begin{bmatrix} 10 \\ 100 \end{bmatrix}$$

$\Rightarrow x_1 = x_0 - \eta v_1 \Rightarrow x_1 = 5 - 0.05(10) = 4.5 \quad [\because \eta = 0.05]$

$\Rightarrow y_1 = y_0 - \eta v_1 \Rightarrow y_1 = 5 - 0.05(100) = 0$

Second iteration,

$$\nabla f(x_1, y_1) = \nabla f(4.5, 0) = \begin{bmatrix} 9 \\ 0 \end{bmatrix}$$

$$v_2 = \beta v_1 + \nabla f(x_1, y_1)$$

$$= 0.9 \begin{bmatrix} 10 \\ 100 \end{bmatrix} + \begin{bmatrix} 9 \\ 0 \end{bmatrix} = \begin{bmatrix} 18 \\ 90 \end{bmatrix}$$

$x_2 = 4.5 - 0.05(18) = 3.6$

$y_2 = 0 - 0.05(90) = -4.5$

Third iteration,

$$\nabla f(3.6, -4.5) = (7.2, -90)$$

$$v_3 = 0.9\begin{bmatrix}18\\90\end{bmatrix} + \begin{bmatrix}7.2\\90\end{bmatrix} = \begin{bmatrix}23.4\\-9\end{bmatrix}$$

$$x_3 = 3.6 - 0.05(23.4) = 2.43$$

$$y_3 = -4.5 - 0.05(-9) = -4.05$$

fourth gradient,

$$\nabla f(2.43, -4.05) = (4.86, -81)$$

$$v_4 = 0.9\begin{bmatrix}23.4\\-9\end{bmatrix} + \begin{bmatrix}4.86\\-81\end{bmatrix} = \begin{bmatrix}25.92\\-89.1\end{bmatrix}$$

$$x_4 = 2.43 - 0.05(25.92) \approx 1.13$$

$$y_4 = -4.05 - 0.05(-89.1) \approx 0.405$$

Q3. Consider $f(x,y) = x^2 + 10y^2$ and compute its minima using Adam optimizer (upto three iterations). Consider the initial points, $x_0 = 5$ and $y_0 = 5$; $m_0 = \begin{bmatrix} 0 \\ 0 \end{bmatrix}$ and $v_0 = \begin{bmatrix} 0 \\ 0 \end{bmatrix}$, learning rate of 0.01 and $\beta_1 = 0.9$, $\beta_2 = 0.999$. Consider $\epsilon \approx 10^{-8}$.

Soln The objective function to be minimized

$$f(x,y) = x^2 + 10y^2.$$

$$g_t = \nabla J(\theta_t) = \nabla f(x,y) \Rightarrow \begin{bmatrix} 10 \\ 100 \end{bmatrix} = g_1$$

$$m_t = \beta_1 m_{t-1} + (1-\beta_1) g_t$$

$$m_1 = \beta_1 m_0 + (1-\beta_1) g_1$$

$$= 0.9 \begin{bmatrix} 0 \\ 0 \end{bmatrix} + (1-0.9) \begin{bmatrix} 10 \\ 100 \end{bmatrix} = \begin{bmatrix} 1 \\ 10 \end{bmatrix} \rightarrow \text{first moment}$$

Similarly, $v_t = \beta_2 v_{t-1} + (1-\beta_2) g_t^2$

$$v_1 = 0.999\, v_0 + (1-0.999) g_1^2$$

$$= 0.999 \begin{bmatrix} 0 \\ 0 \end{bmatrix} + 0.001 \begin{bmatrix} 100 \\ 10000 \end{bmatrix} = \begin{bmatrix} 0.1 \\ 10 \end{bmatrix}$$

$\rightarrow$ second moment

Bias corrections, $\hat{m}_1 = \dfrac{m_t}{1-\beta_1^t} = \begin{bmatrix} 10 \\ 100 \end{bmatrix}$; $\hat{v}_1 = \dfrac{v_t}{1-\beta_2^t} = \begin{bmatrix} 100 \\ 10000 \end{bmatrix}$

$$x_1 = x_0 - \eta \frac{\hat{m}_1}{\sqrt{\hat{v}_1 + \epsilon}} = 5 - 0.1 \frac{10}{\sqrt{100}} = 4.9$$

$$y_1 = y_0 - \eta \frac{\hat{m}_1}{\sqrt{\hat{v}_1 + \epsilon}} = 5 - 0.1 \frac{100}{\sqrt{10000}} = 4.9$$

Continue.....