Optimization ⇒

① In Direct numerical computation, we have

$$\frac{\partial F(x)}{\partial x_i} = \lim_{h \to 0} \frac{f(x_1 \cdots x_i + h, \cdots, x_n) - f(x_1, \cdots x_n)}{h}$$

Where h = 'a hyperparameter'

To compute the gradient for a given 'xi', the value 'h' is added into in function 'F' and then $\left(\frac{\partial F}{\partial x_i}\right)$ is calculated while this method is very straight forward, it is ~~Cntea~~ enterely slow. Say we have very lenge number of cofficients (in millions), computing loss for each coefficent will take a long time. Since we have to push the training data with the change in the parameter, for every coefficient.

Therefore, using lack propagation is preferred because it is comparitively faster than Numerical Computation.

A possible ~~us~~ use are of direct Numerical Computation is to cross-chuck the value of gradients that we have obtained using any other method and hence verify our result.

② Gradient descent is

$$\Theta^{i+1} \leftarrow \Theta^i + \eta \sum_{i=1}^{k} \nabla L(\Theta^i)$$

This gives us the ~~Gumi~~ gradient for the loss while stochostic gradient descent is:-

$$\Theta^{i+1} \leftarrow \Theta^i - \eta \nabla L(\Theta^i)$$

This provides the weight update at each node Output. ~~It is done~~ In gradient descent, we have to run through all the examples is your

training set to do a single update for a parameter in a particular iteration. Using gradient descent may take too ~~log~~ long because in every iteration where you update the values of parameters you were running through the complete training set.

Stochastic gradient descent (SGD) is an approximation of the gradient descent algorithm by computing the gradient over mini-batches instead of individual datapoints.

SGD is faster because we only use training sample as a time & it starts improving itself right away from the first sample.

$$\theta^{i+1} \leftarrow \theta^i - \eta (x^{(i)} \theta^i - y^{(i)}) x^{(i)}$$

→ SGD after converges must faster compared to ~~GD~~ GD but the error function is not as well minimized as in the case of ~~GD~~, GD.

(3) The trade off in selecting the batch size for SGD is while selecting a large batch size is comparitively faste, it has been observed that the generalization of the model decreases drastically. Where as selecting a smaller batch size for SGD is slower, the model actually trains better and the generalization increases.

four problems with SGD are :→

① Selecting the learning rate is difficult, since it is a hyper parameter and can affect the loss computed by the model.

② the loss can be sensitive towards different hyper parameters.

1.

⑧ Avoiding getting stuck out local min or saddle points can be a challenge.

⑨ The mini batch gradient estimates are noisy.

Q ④ For NOISY gradients :→

Poor Conditioning → It is the condition when the gradient is more tensitive to any one parameter. It is fixed by momentum by averaging with previous gradients

SGD with momentum :-

$$v^{i+1} \leftarrow \rho v^i + \nabla L(\theta^{(i)})$$

↓ ↳ old direction ↑ Smooth out Changes

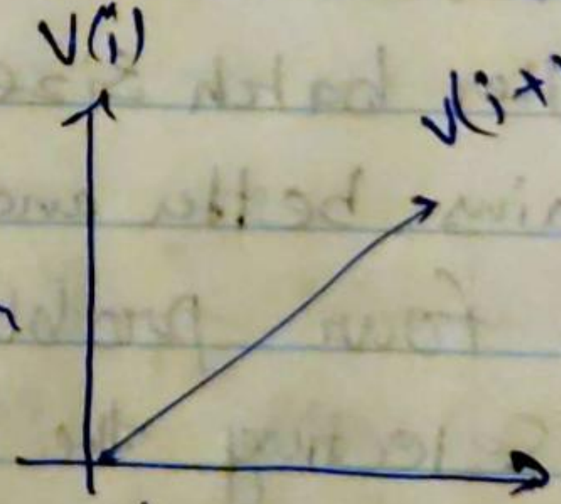
direction used is the direction of

in update gradient.

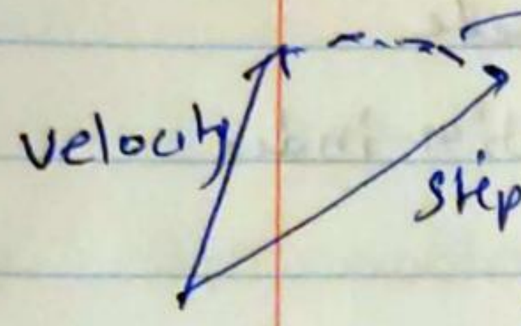
$$\theta^{t+1} \leftarrow \theta^i - \eta v^{i+1}$$

Saddle points :→ SGD with momentum helps the algorithm to not get stack at the local minim. & keep taking steps with the magnitude of the velocity instead of the gradient descent value.

Noise :→ Using subset of example to compute gradients while producing faster gradients, It also produces noisier gradients SGD with momentum smoothes out changes to the gradient using momentum.



Q (5) Nesterov-Accelerated Gradient (NAG): takes the first the momentum steps befor taking the gradient step. This makes is reach the changes in gradient direction leading to the term accelerated is the NAG.



corrected gradient descent

$$v^{i+1} \leftarrow \rho v^i - \eta \nabla L(\theta^{(i)} + \rho v^{(i)})$$

↳ old direction

Simple momentum ⇒

$$\tilde{\theta}^{(i)} \equiv \theta^{(i)} + \rho v^{(i)}$$

$$\theta^{(i)} = \theta^i - \rho v^{(i)}$$

$$\theta^{i+1} = \tilde{\theta}^{i+1} - \rho v^{(i+1)}$$

$$\tilde{\theta}^{i+1} - \rho v^{(i+1)} \leftarrow \tilde{\theta}^i - \rho v^{(i)} + v^{i+1}$$

$$\tilde{\theta}^{i+1} \leftarrow \tilde{\theta}^i + v^{i+1} + \rho(v^{i+1} - v^{(i)})$$

↓ acceleration term

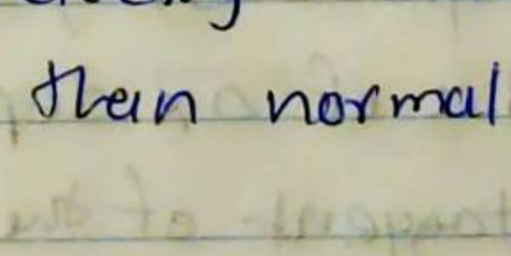
Difference b/w new velocity and old velocity. This converges highly better than normal momentum.

Q6 Learning Rate need not to be fixed. Start with some high value and then slowly converge to the optimal minimal point. make learning smaller as the iterations progeress.

Strategies ⇒

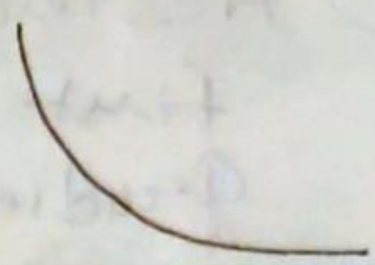
① Step Decay: In every K iteration, half the learning rate.

$$\eta \leftarrow \eta/2$$



② Exponential decay ← decay rate

$$\eta = \eta_0 \cdot e^{-k/t} \leftarrow \text{iteration index}$$



$k \rightarrow$ decay rate

$t \rightarrow$ iteration index.

③ fractional decay ⇒ LR is scaled down by a factor of $\frac{1}{(1+kt)}$

$k \leftarrow$ decay rate

$t \leftarrow$ iteration index

$$\eta = \frac{\eta_0}{(1+kt)}$$

mosly the step decay function is used to reduce the learning rate.

Q7 Newton's Method :→

Goal: Compute learing rate 'η' instead of specifying it The learning rate from diffrent features may vary (loss may be more sensitive to some feature in this case LR should be low for this paremeter.

Newton's Algorithm (1D) ⇒

1. find x such that $f(x)=0$
2. Start with guess x_0
3. find update Δx, such that $f(x_0+\Delta x)=0$

the value of Δx depends on $f(x)$. Δx is the slope or the tangent of the fuction.

L.R. in Newton's method using a taylor's series expansion ⇒

$$f(x_0+\Delta x) = f(x_0) + \Delta x f'(x_0) + \cdots = 0$$

$$= f(x_0) + \Delta x f'(x_0) = 0$$

$$\Delta x = \frac{-f(x_0)}{\Delta f'(x_0)}$$

Continue while $f(x_0+\Delta x) \neq 0$

Small eg ⇒ $f(x) = x^2 + 3x + 2$ $\quad x_0 = 1$

$f'(x) = 2x + 3$ $\quad f(x_0) = f(1) = 6$

$$\Delta(x) = \frac{-f(x_0)}{\Delta f(x_0)} = -\frac{6}{5}$$

$$f(x_0 + \Delta x) = f\left(1 - \frac{6}{5}\right) = f\left(-\frac{1}{5}\right) = \frac{1}{25} - \frac{3}{5} + 2 \simeq 2.05$$

$$\boxed{\Delta x = -2.5/8}$$

Our problem : $\underbrace{\nabla J(\theta)}_{\exists f(x)} = 0$ ← loss

$f(x_0 + \Delta x) = f(x_0) + \Delta x^T \nabla f(x_0) + \cdots = 0$

$\Rightarrow f(x_0) + \Delta x^T \cdot \nabla f(x_0) = 0$

→ Replacing $f(x_0)$ with $\nabla J(\theta_0)$ $\quad \theta_0 \leftarrow x_0$

$\nabla J(\theta_0) + \Delta x^T \underbrace{\nabla(\nabla J\theta_0)}_{\text{↳ Hessian matrix. 'H'}} \cdot \Delta\theta = 0$

$$H = \begin{bmatrix} \frac{\partial J^2}{\partial\theta_1 \partial\theta_1} & \cdots & \frac{\partial J^2}{\partial\theta_1 \partial\theta_n} \\ \vdots & & \\ \frac{\partial J^2}{\partial\theta_n \partial\theta_1} & \cdots & \frac{\partial J^2}{\partial\theta_n \partial\theta_n} \end{bmatrix} \qquad \theta = \begin{bmatrix} \theta_1 \\ \vdots \\ \theta_n \end{bmatrix}$$

Rewrite the equation in H form.

loss derivation → $\nabla J(\theta_0) + H \cdot \Delta\theta = 0$ ⟶ Hessian parameter update of loss.

$\{\Delta\theta = -H^{-1} \nabla J(\theta_0)\}$ //

* Newton's method ⇒

$\theta^{(i+1)} \leftarrow \theta^{(i)} - H^{-1} \nabla J(\theta^{(i)})$

Gradient descent ⇒

$\theta^{i+1} \leftarrow \theta^i - \eta \nabla \pi(\theta^i)$

* Interpretation of H ⇒ (Assume that H is diagonal)

$$H = \begin{bmatrix} h_1 & & \\ & \ddots & \\ & & h_n \end{bmatrix} \Rightarrow H^{-1} = \begin{bmatrix} 1/h_1 & & \\ & \ddots & \\ & & 1/h_n \end{bmatrix}$$

element of Secound derivatives of loss curvature.

Slop → first derivative

high Curvature (h_i) → second derivative

* If function is changing fast w.r.t θ_i then 'h_i' is be large and '$1/h_i$' inverse is small.

⇒ If curvature is high smaller step will taken and vice -versa.

* Learning Rate for i^{th} feature : $1/h_i$

* high Curvature (h_i) → loco learing rate.

* Too large to store and invers $O(N^3)$ Complexety.

* Problems: ① Hession matrix derivative are sensitive to noise ② matrix can ~~there~~turn too lenige and problematic to store.

Q8: Condition number ⇒ are generally used to determine the sensitivity of the parameters using the singular values of the Hession matrix. Condition number is the ratio of largest singular values is the hession matrix to the ~~Sa~~ smallest singular values

$$\text{Condition Number} = \frac{Sv \text{ largest}}{Sv \text{ Smallest.}}$$

It tells how difficult or complex the problem is In case of poor conditioning the condition number ↑. ~~be~~ increase.

(9) AdaGrad ⇒ It addresses the problem of Hessian matrix being too big, too expensive to invert and noisy.

-Replace H with different pre conditions.

Pre conditions :- Adjusts the learning rate depending upon the curvature. that will not require us to calculate 2nd order derivatives and find the inverse.

$$B^i = \text{diag}\left(\sum_{j=1}^{i} \nabla J(\theta^{(i)})\, \nabla J(\theta^{(i)})^T\right)^{1/2}$$

$$B^{(i)} = \begin{bmatrix} \sqrt{\sum \left(\frac{\partial J}{\partial \theta_1}\right)^2} & & \\ & \ddots & \\ & & \sqrt{\sum \left(\frac{\partial J}{\partial \theta_n}\right)^2} \end{bmatrix} \Rightarrow B^{(i)-1} = \begin{bmatrix} \frac{1}{\sqrt{\sum \left(\frac{\partial J}{\partial \theta_1}\right)^2}} & & \\ & \ddots & \\ & & \frac{1}{\sqrt{\sum \left(\frac{\partial J}{\partial \theta_n}\right)^2}} \end{bmatrix}$$

$$\theta^{i+1} \leftarrow \theta^i - \eta\, B^{(i)-1} \nabla J(\theta^{(i)})$$

In case of Hessian matrix :

$$\frac{\partial^2 J}{\partial \theta^2} = \sum \left(\frac{\partial J}{\partial \theta_1}\right)^2 \rightarrow$$ the need to calculate second order derivative is removed.

So, Adagrad uses the squared sum of past derivatives to approximate the inverse of Hessian.
(limitate)

(10) RMSProp ⇒ is improvement of AdaGrad. In AdaGrad because we normalize by element wise sum of square gradients, the step size will become smaller as iterations progress. To control this we use in RMSProp as a decay factor (e.g 0.9) when adding new gradients to the gradients sum.

Scale factor for j^{th} component $\leftarrow s_j^{(i+1)} = \gamma s_j^{(i)} + (1-\gamma) \|\nabla_j L(\theta^{(i)})\|^2$

$$\theta_j^{(i+1)} \leftarrow \theta_j^{(i)} - \eta \nabla_j L(\theta^{(i)}) \circ \frac{1}{s_j^{(i+1)} + \epsilon}$$

eg $\gamma = 0.9$

Q11 Adam algorithm :→ Combine RMSprop (scale by sum of gradient elements) with momentum.

$$m_1^{(i+1)} = \beta_1 \cdot m_1^{(i)} + (1-\beta_1) \nabla L(\theta^{(i)})$$ first moment: velocity with momentum.

$$m_2^{(i+1)} = \beta_2 \cdot m_2^{(i)} + (1-\beta_2)\left(\nabla L(\theta^{(i)}) \odot \nabla L(\theta^{(i)})\right)$$ Second moment: elementwise Step size Scale

$$\theta^{(i+1)} = \theta^{(i)} - \eta\, m_1^{(i+1)} \odot \frac{1}{\sqrt{m_2^{(i+1)} + \epsilon}}$$

$m_1^{(0)} = m_2^{(0)} = 0$

$$\theta^{i+1} \leftarrow \theta^{(i)} - \eta\, m_1^{(i+1)} \cdot \frac{1}{\sqrt{m_1^{(i+1)} + \epsilon}}$$

↓ velocity gradient ↓ Scaling factor.

It might fail, as we initialize $m_1^{(0)} = m_2^{(0)} = 0$ the algorithm might explode due to high initial/big step factor.

* Need for Bias corrected item :-

Because the moments are initialized to 0, when dividing by

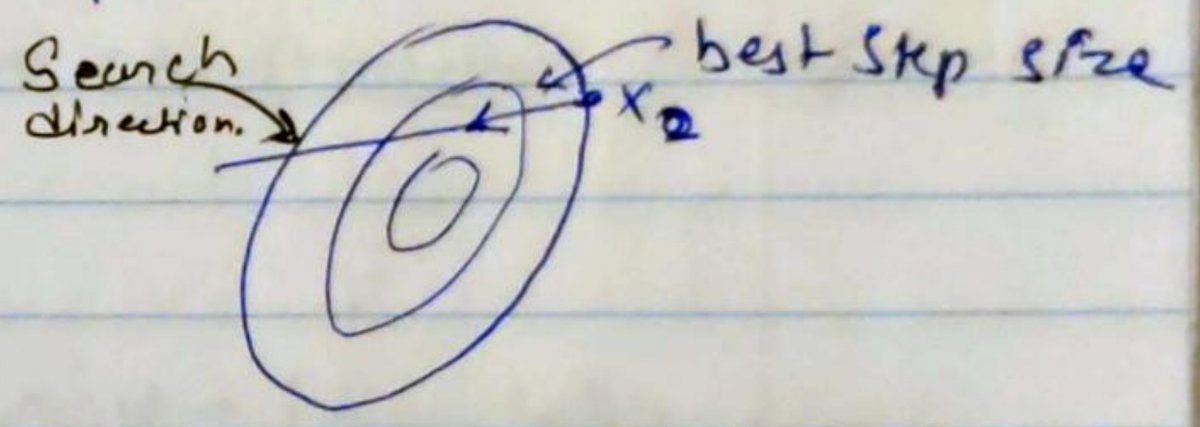
the second moment we will get a large step
→ use a bias correction term → dividing the momentem.
by a number depending on ~~Condition~~ Number iteration
(so that initial moments are large)

$$\hat{m}_1^{(i+1)} = \frac{m_1^{(i)}}{1-\beta_1^{(i)}}$$

$$\hat{m}_2^{(i+1)} = \frac{m_2^{(i)}}{1-\beta_2}$$

↘ unbiased estimates of m_1 and m_2.

Q12 ⇒ Gradient descent with line search :→
Instead of a fixed step size, find the "best" step size.



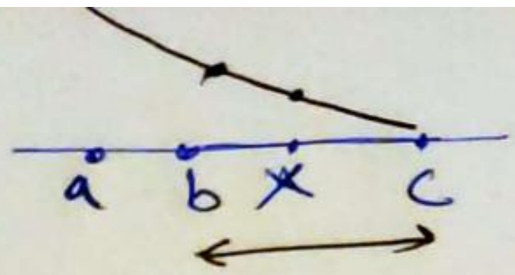
- Given direction:
$u = \nabla f(x)$
- Best step size :
$\eta^* = \underset{\eta}{\operatorname{argmin}} f(x+\eta u)$ ← Solve another optimization problem in each step.
↳ parameter that we are trying to optimize keep doing G.D. to find optimal step.
- Gradient descent:

$$\theta^{(i+1)} \leftarrow \theta^{(i)} - \eta^{(i)} \nabla f(\theta^{(i)}) \quad (\eta^* \leftarrow \eta^{(i)})$$

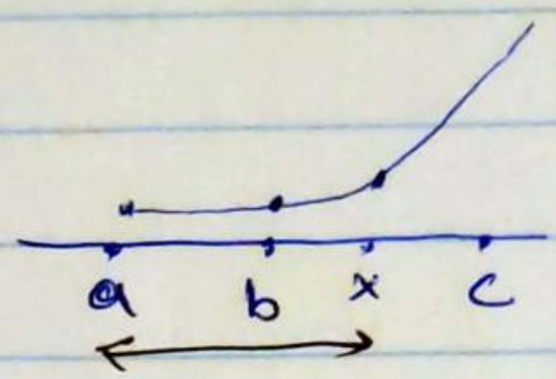
* To solve for η^*: find explicit computation or use gradient descent (expensive) on perform simple line ~~seach~~ search.
* Simple procedure for line ~~seac~~ search : Bracketing.
Given Bracket : $[a, b, c]$

$$x = \frac{b+c}{2}$$

; if $f(x) \leq f(b) \Rightarrow [b, x, c]$



if $f(x) > f(b) \Rightarrow [a, b, x]$.



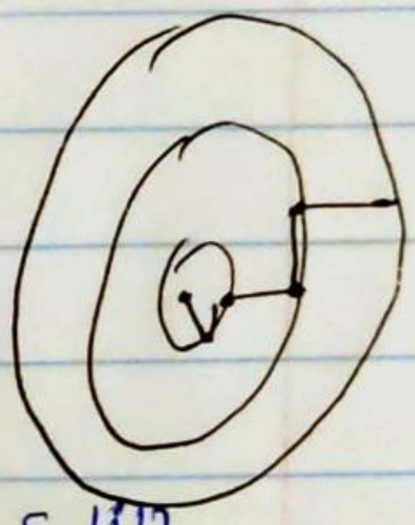
Continue with smaller and smaller brackets until the bracket is small enough.

* Succesive line Seareh.

→ Stert with θ_0 and direction set $\{u^{(i)}\}$

$$\eta^{(i)} = \underset{\eta}{\operatorname{argmin}} f(\theta^{(i)} + \eta u^{(i)})$$

$$\theta^{(i+1)} \leftarrow \theta^{(i)} + \eta^{(i)} u^{(i)}$$



eg coordinate descent $\{u^{(i)}\} = \{e^{(i)}\}$

→ This is not as efficient as gradient descent because Step calculation is not great though iterations are faster here, computational cost is low.

Q13

Quasi network methods. :-

Inverting the Hessian is expensive with complexity of $O(n^3)$ to invert n^2 elements.

- These methods approximate the Hessian inverse using gradient evaluations. → scaling will be more accurace & will require less steps.

→ these methode normally require large no. of sample to work.

Algorithm :→

1) Compute Quasi-Newton direction.

$$\Delta\theta = -(H^{(i-1)})^{-1} \nabla J(\theta^{(i-1)})$$

2) Determine step size η^* (e.g. bracketing line search).

3) Compute parameter update.

$$\theta^{(i)} = \theta^{(i-1)} + \eta^* \Delta\theta$$

4) Compute updated Hessian approximate $H^{(*)}$

- Instead of computing updated Hessian approximate it is possible to compute update $(H^{(*)})^{-1}$

* BFGS update: has the advantage of being faster than the newton Algorithm.

While the BFGS Algorithm is more accurate than Adam, Adam is a faster ~~alg~~ algorithm.

$$\Delta\theta = \theta^{(i)} - \theta^{(i-1)}$$

$$\Delta J = \nabla J(\theta^{(i)}) - \nabla J(\theta^{(i-1)})$$

$$H^{(i)} = H^{(i-1)} + \frac{\Delta J \Delta J^T}{\Delta J^T \Delta\theta} - \frac{H^{(i-1)} \Delta\theta \Delta\theta^T H^{(i-1)}}{\Delta\theta^T H^{(i-1)} \Delta\theta}$$

Inverse update.

$$(H^{(i)})^{-1} = \left(I - \frac{\Delta\theta \Delta J^T}{\Delta J^T \Delta\theta}\right)(H^{(i-1)})^{-1}\left(I - \frac{\Delta J \Delta\theta^T}{\Delta J^T \Delta\theta}\right) + \frac{\Delta\theta \Delta\theta^T}{\Delta J^T \Delta\theta}$$

inverse update cost $O(n^2)$