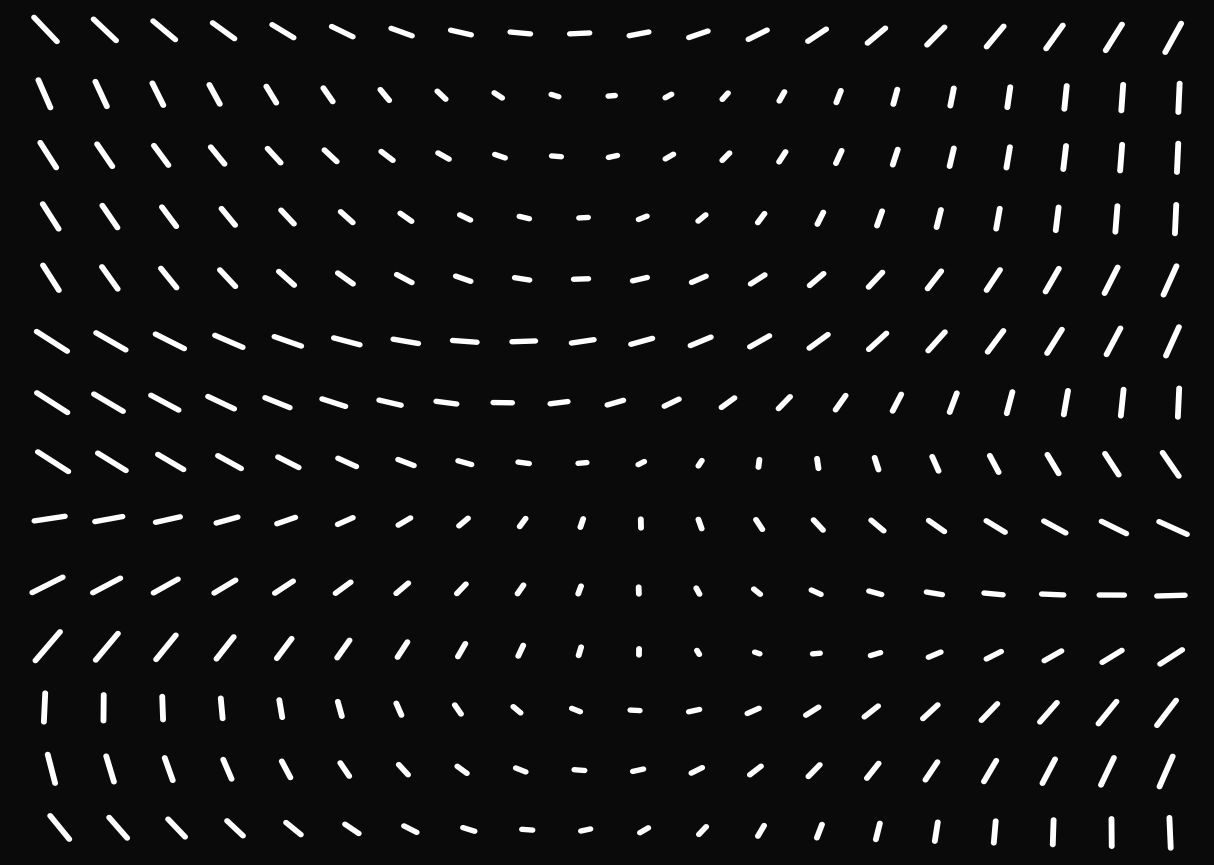
Optimization



The goal in optimization is to find the parameters $\theta \in \Theta$, that minimize a scalar-valued loss function or cost function $\mathcal{L}: \Theta \to \mathbb{R}$

$$\theta^* \in \underset{\theta \in \Theta}{\operatorname{argmin}}\ \mathcal{L}(\theta) \longrightarrow \Theta \in \mathbb{R}^D$$

θ^*: Optimal parameters

If we want to maximize the reward function, then the loss would be $\mathcal{L}(\theta) = -R(\theta)$. Higher the reward, lower the loss and vice-versa.

Algorithm which optimizes the objective function is called as "solver."

8.1.1 Local vs Global Optimization

Any point that acheives θ^* is called global optimum.

Finding global optimum is computationally intractable — why? Non-convex problems have irregular shapes in gradients which indicate the difficult of solution tossed in a random valley.

So we find local optimum instead which is θ^*.

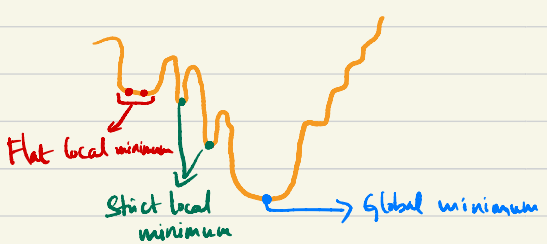
We refer θ^* as local minimum if

$$\exists \delta > 0,\ \forall \theta \in \Theta \text{ s.t. } \|\theta - \theta^*\| < \delta,\ \mathcal{L}(\theta^*) \leq \mathcal{L}(\theta)$$

δ in some optimization implementations reference as tolerance or epsilon.

A local minimum could be surrounded by other local minima. This is called flat local minimum.

Usually we would like to look for strict local minimum where the cost is strictly lower than those of neighboring points



Similar we can define (strict) local maximum for optimizing reward function.

Optimality conditions for local vs global optima

We can characterize the local minima points of any function which is twice differentiable.

Let $g(\theta) = \nabla \mathcal{L}(\theta)$ be the gradient vector

$H(\theta) = \nabla^2 \mathcal{L}(\theta)$ be the hessian matrix

$\theta^* \in R^D$ some stationary point $\sim$ local minima

$g^* = g(\theta)|_{\theta^*}$ be the gradient at that point

$H^* = H(\theta)|_{\theta^*}$ be the corresponding Hessian

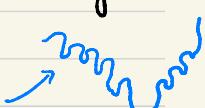
We can now characterize every local minimum θ^* with the following conditions:

Necessary condition: If θ^* is local minimum, then $g^* = 0$ and H^* must be positive semi definite ie., $z^T H^* z \geq 0$

Sufficient condition: If $g^* = 0$ and H^* is positive definite, then θ^* is local minimum.

To illustrate, why first condition is required, take a stationary point θ^*. The gradient for this θ^* would be a non-zero (for starters). So now we would like to move the gradient close to zero. We do this by subtracting the gradient of θ^* i.e., $\theta^* = \theta - \alpha g(\theta)$. Ignore α for now, it is some arbitary constant. By doing this the gradient would be zero.

So, that's it? End of optimization?



- No, not really. In non-smooth functions like this there are several local minimums so θ^* may have reached local minimum but not the global minimum. When the gradient starts climbing the $g^* > 0$, so we cannot stop optimization at an arbitary local minimum.

How do we know we're not local minimum?

- The Hessian of the function will be positive definite whereas in local minimum the eigenvalues of Hessian $H(\theta)$ are both positive and negative indicating some directions are pointing uphill and flat locals. In positive definite Hessian all directions point uphill

8.1.2 Constrained vs Unconstrained optimization

Unconstrained means the choice variable can take any value - there are no restrictions. Constrained means that choice variable can only take certain values within a larger range.

This variable could be a vector or matrix which will have certain equality or inequality constraints or both for certain optimization problem in constrained optimization.

Let's suppose we'd like to optimize our loss ($\mathcal{L}(\theta)$). We could certainly allow the parameters to be optimized without any constraints - i.e., $\theta \in C$ where C is constraints and when there are no constraints $C \subseteq \mathbb{R}^D$ ⇝ which suggest C has no constraints and θ can take any parameter. This is called Unconstrained Optimization

Let's impose certain constraints - i.e.,

Gradient ← $g_j(\theta) \leq 0$ for $j \in \mathcal{I}$ → Inequality constraint

$h_k(\theta) = 0$ for $k \in \mathcal{E}$ → Equality constraint

↓ Hessian

→ for instance, $h(\theta) = 1 - \sum_{i=1}^{D} \theta_i = 0$ (sum-to-one constraint)

Now the Constrained Optimization will be

$$C = \{\theta : g_j(\theta) \leq 0 : j \in \mathcal{I}, h_k(\theta) = 0 : k \in \mathcal{E}\} \subseteq \mathbb{R}^D$$

This subset is called feasible set

Task of finding any point in this is called "feasibility problem"

We can solve any problem as unconstrained optimization while still using constraints. We can achieve this by creating a penalty term for every violated constraint and add the penalties to the objective function. Thereforth, still solve as an unconstrained optimization.

→ It still seems like constrained, though!?

8.1.3 Convex vs nonconvex optimization

In convex optimization problems, every local minimum is also a global minimum.

Convex sets

We say S is a convex set if we randomly pick two points x and x' such that $x, x' \in S$ follows

$$\lambda x + (1-\lambda)x' \in S, \quad \forall \lambda \in [0,1]$$

So any line between x and x' will fall inside the set (S).

Convex sets

Non-convex sets

Convex functions

We call function 'f' as a convex function if it is defined on a convex set. ie, for any $x, y \in S$ and $0 \leq \lambda \leq 1$, we have

$$f(\lambda x + (1-\lambda)y) \leq \lambda f(x) + (1-\lambda) f(y)$$

If the above inequality is strict, then it is called strictly convex

A function is concave, if $-f(x)$ is convex is strictly convex.

Theorem 8.1.1 : A function f is convex iff $\nabla^2 f(x)$ i.e., Hessian is positive semi definite. If $\nabla^2 f(x)$ is positive definite, then the function is strictly convex.

Strongly Convex

A function f is strongly convex if the following holds

$$(\nabla f(x) - \nabla f(y))^T (x-y) \geq m \|x-y\|_2^2 \quad \text{where } m > 0$$

A strongly convex function is also strictly convex but not vice versa

what's the difference? They seem so subtle in pg. 275.

8.1.4 ## Smooth vs non smooth optimization

In smooth optimization, the objective and constraints are continuously differentiable functions. In 1D case,

$$|f(x_1) - f(x_2)| \leq L|x_1 - x_2|$$

$L \geq 0$

L: Lipschitz constant: quantifies the degree of smoothness

In non smooth optimization, there are some points where gradient of the objective function or the constraints is not well-defined.

We can partition such objectives with smooth and non-smooth terms as:

$$\mathcal{L}(\theta) = f_s(\theta) + f_r(\theta)$$

$\mathcal{L}$: a.k.a composite objective; f_s: smooth; f_r: nonsmooth ("rough")

Subgradients

A function f is convex and is differentiable, then its gradient at x is subgradient. However, subgradient can exist even when f is not differentiable.

But how are subgradients computed?

Before we dive into that we need to get a quick primer on what is linear approximation and how it simplifies complex computation?

Linear Approximation:

There is exist a tangent line for any function which is quite close to the function i.e., at the point of tangency. If this tangent can be easily computed, then we can also approximate the function. It is given as:

$$L(y) = f(x) + f'(x) \cdot (y - x)$$

$L(y)$: This is linear approximation of $f(y)$ i.e., $L(y) \simeq f(y)$

Linear Approximation Notes: https://npflueger.people.amherst.edu/math1a/lecture10.pdf

To give you an example, how linear approximation works. Let's compute square root of any number.

Let $f(x) = \sqrt{x}$ and we would like to approximate $\sqrt{26}$.

We know $\sqrt{25} = 5$ a closest value to $\sqrt{26}$

$$f(x) = \sqrt{x} = x^{1/2}$$

$$f'(x) = \frac{1}{2}x^{-1/2} = \frac{1}{2\sqrt{x}}$$

$$f'(25) = \frac{1}{2 \cdot 5} = \frac{1}{10}$$

Therefore linear approximation of $\sqrt{26}$ would be

$$L(x) = f(25) + f'(25)(x-25)$$

$$= 5 + \frac{1}{10}(x-25)$$

Here $x = 26$, so

$$L(26) = 5 + 0.1(26-25)$$

$$= 5.1 \quad \text{which is close to } 5.0990195$$

For a function $f(x)$, if it is double derivative $f''(x)$ is positive then we call the function as concave up and if $f''(x)$ is negative then we call the function as concave down. (Mostly we could see in functions with square & n^{th} roots)

When $f(x)$ is concave up in some interval around $x=c$, then $L(y)$ underestimates in this interval. i.e., the value will be slightly lower than $f(x)$

Similarly, when $f(x)$ is concave down, then $L(y)$ overestimates i.e., values will be slightly higer

With this background, we can write

$$f(y) \geq f(x) + \nabla f(x)^T (y-x) \quad \forall\, x, y$$

i.e., linear approximation always underestimates f since f is concave up.

A **subgradient** of a convex function f at x is any $g \in \mathbb{R}^n$ such that

$$f(y) \geq f(x) + g^T(y-x) \quad \forall\, x.$$

Linear approximation can always be performed. Subgradients are linear approximations.

Subgradients

- always exists.
- If f is differentiable at x, then $g = \nabla f(x)$ uniquely
- Same definitions works for non-convex functions.

Repeating again,

Subgradients can exist when f is not differentiable at a point.

A function f is called subdifferentiable at x if there is atleast one subgradient at x. The set of such subgradients is called subdifferential of f at x i.e., $\partial f(x)$

Example: $f(x) = |x|$

$$\partial f(x) = \begin{cases} \{-1\} & \text{if } x < 0 \\ [-1, 1] & \text{if } x = 0 \\ \{1\} & \text{if } x > 0 \end{cases}$$

$f(z) = f(x) + g^T(z-x)$
$= |x| + g^T(z-x)$
$= 0 + g^T z \Rightarrow gz$

unique subgradient when $x \neq 0$

gradient must be $[-1, 1]$

8.2 First-order methods

These are iterative optimization methods that leverage first order derivatives of the objective function. In these methods, we compute which directions point "downhill".

$$\theta_{t+1} = \theta_t + \rho_t d_t$$

- θ_{t+1}: New update
- θ_t: Current parameter
- ρ_t: learning rate or step size
- d_t: descent direction.

How are d_t and g_t are related?
- You can use $g_t = d_t$ but there are other optimal ways to compute 'd_t'

$g_t = \nabla L(\theta_t)$

Updates are performed until the gradient is zero.

8.2.1 Descent Direction

A direction d is referred as descent direction if for a step size ρ, the objective function is minimized i.e.,

$$L(\theta + \rho d) < L(\theta)$$

The gradient at the current timestep (or iteration) is

$$g_t \triangleq \nabla \mathcal{L}(\theta)|_{\theta_t} = \nabla \mathcal{L}(\theta_t) = g(\theta_t)$$

Here $g(\theta_t)$ points in the maximal increase in f → Meaning optimizing Loss or any objective function

It can be shown that,

Any direction d is also a descent direction if the angle θ between d and $-g_t$ is less than 90° and satisfies

$$d^T g_t = \|d\| \|g_t\| \cos(\theta) < 0$$

If we pick, $d_t = -g_t$, then the direction is called "direction of steepest descent" This can be quite slow. → Why?

8.2.2 Step size (learning rate)

As we have already mentioned ρ is learning rate, whereas step sizes $\{\rho_t\}$ is called learning rate schedule.

Constant step size

In constant step size $\rho_t = \rho$. Picking the ideal ρ is important. If the step size is too large, then we may fail to converge solution and if it is too small, the convergence will be too slow.

In some cases, we can derive heuristics for step size.
For example, for the following quadratic objective

$$\mathcal{L}(\theta) = \frac{1}{2}\theta^T A \theta + b^T \theta + c$$

Global convergence for this objective could be achieved iff the descent satisfies

$$\rho < \frac{2}{\lambda_{max}(A)}$$

where λ_{max} is the maximum eigen value of A

Intuition: the stepsize won't be larger than slope of steepest direction for a rolling ball.

Line search

Optimal step size can be found as

$$\rho_t = \underset{\rho>0}{\operatorname{argmin}}\ \phi_t(\rho) = \underset{\rho>0}{\operatorname{argmin}}\ f(\theta_t + \rho d_t)$$

This is clear. ρ which minimizes the objective function.

This is not clear. What's $\phi_t(\rho)$? ϕ_t are parameters of a function. $\phi_t(\cdot)$ is not a function. Or maybe could be read as "parameters given ρ". But what does argmin on parameters mean?

This optimization is known as Line Search.

It is called line search because we are searching along the line defined by d_t.

Using the optimal step size (ρ^*) is called "exact line search". However it is not necessary to be so precise.

Other Methods such as Armijo Backtracking Method ensure sufficient reduction in the objective function without spending too much time in solving ρ_t

In particular, we can start with current step size and reduce it by a factor $0 < \beta < 1$ at each step while we satisfy the following condition.

→ This is known as Armijo-Goldstein test

$$f(\theta + \rho d_t) \leq f(\theta_t) + c\,\rho\, d_t^T \nabla f(\theta_t)$$

$f(\theta_t)$ → loss

c → constant, usually $\sim 10^{-t}$ $c \in [0,1]$

$d_t^T \nabla f(\theta_t)$ → descent direction → Gradient

8.2.3 Convergence Rates

In certain convex problems, the gradient descent converges at a linear rate denoted as μ.

$$|f(\theta_{t+1}) - f(\theta^*)| \leq \mu\, |f(\theta_t) - f(\theta^*)|$$

This is known as "Rate of convergence"

For example, for a quadratic objective function

$$\mathcal{L}(\theta) = \frac{1}{2}\theta^T A\theta + b^T\theta$$

One can show the convergence rate as

$$\mu = \left(\frac{d_{max} - d_{min}}{d_{max} + d_{min}}\right)^2$$

We know from the topic of conditional numbers that $K = \frac{d_{max}}{d_{min}}$

So μ can be written as

$$\mu = \left(\frac{K-1}{K+1}\right)^2$$

If we recall, K (conditional number) suggest the numerical stability of the matrix. Numerical stability meaning for small fluctuations the output shouldn't vary much. If it varies, then the matrix is numerically unstable. So, a smaller K indicates numerically stable matrix. Similarly, a smaller conditional number (K) indicates quicker convergence to the solution.

For non-quadratic functions, the convergence rate depend on the condition number of Hessian. Steepest descent with line search will exhibit zig zag behavior which is inefficient. This is due to skews in local curvature. We can overcome such behavior by using Hessians.

8.2.4 Momentum methods

During gradient descent, we want the gradient updates to move faster along the descent direction and slow down when the gradient has changed. This is given as:

$$m_t = \beta m_{t-1} + g_{t-1}$$
$$\theta_t = \theta_{t-1} - \rho_t m_t$$

m_t is the momentum
β is constant $0 < \beta < 1$
$\beta = 0 \sim$ gradient descent

Example:

β	m_{t-1}		g_{t-1}	m_t
0.9 *	0.1	+	0.1	0.19
0.9 *	0.19	+	0.1	0.271
0.9 *	0.271	+	0.1	0.3439
0.9 *	0.3439	+	0.1	0.40951
0.9 *	0.40951	−	0.3	0.068559
0.9 *	0.068559	−	0.4	−0.3382969

m_t's are Faster Updates than g_{t-1}

Updates are reflected when the gradient is in the opposite direction "uphill". Damping the oscillations, basically.

Expanding m_t would yield:

$$m_t = \beta m_{t-1} + g_{t-1}$$
$$= \beta[\beta m_{t-2} + g_{t-2}] + g_{t-1}$$
$$= \beta^2 m_{t-2} + \beta g_{t-2} + g_{t-1}$$
$$m_t = \sum_{\tau=0}^{t-1} \beta^\tau g_{t-\tau-1}$$

This expansion is akin to "exponential weighted moving average". Which is used to compute moving average in time series data by discounting farther time points and weighting near points. Often used in stock analysis.

To simplify, if all the gradient updates are constant:

$$m_t = g \sum_{\tau=0}^{t-1} \beta^\tau$$

In geometric series, this is given as

$$1 + \beta + \beta^2 + \dots = \sum_{i=0}^{\infty} \beta^i = \frac{1}{1-\beta}$$

Thus we can scale up the gradient by a factor of 10 i.e., $\frac{1}{1-0.9} = \frac{1}{0.1} = 10$ when the gradient updates are constant. But this doesn't seem like it slow downs at valley, rather it shoots up

Nesterov Momentum

We alleviate the oscillation issue by modifying the momentum implementation to include an extrapolation step

$$m_{t+1} = \beta m_t - \rho_t \nabla \mathcal{L}(\theta_t + \beta m_t)$$
$$\theta_{t+1} = \theta_t + m_{t+1}$$

- βm_t: momentum in the direction (jump)
- $\nabla \mathcal{L}(\theta_t + \beta m_t)$: measuring gradient at new location (correction)
- m_{t+1}: accumulated gradient with momentum

- The momentum vector m_t is already roughly pointing in the right direction
- Measuring the gradient at the new location, $\theta_t + \beta m_t$, rather than the current θ_t can be more accurate.

8.3 Second-order Methods

Computing gradients of a function is cheap. Methods which rely on gradients for optimization are called first-order methods. However, these methods don't model the curvature of the space. Hence, they can be slow to converge.

Methods which incorporate curvature in various ways (eg: Hessian) are called second order methods. These yield faster convergence.

8.3.1 Newton's Method

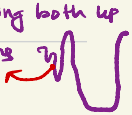
In Newton's method, the hessian-based updates are of the form

$$\theta_{t+1} = \theta_t - \rho_t H_t^{-1} g_t \qquad H_t \triangleq \nabla^2 \mathcal{L}(\theta)|_{\theta_t} = \nabla^2 \mathcal{L}(\theta_t) = H(\theta_t)$$

where H_t is assumed to be positive-definite to ensure the update is well-defined.

Newton method is better than gradient descent because the hessian (H^{-1}) removes any skews in the local curvature. Another assumption is that we're dealing with convex problems only. So the Hessian will be positive definite.

What if H_t is not positive definite?
Then it means some eigenvalues are +ve and some are negative which means pointing both up and down directions



Case for Linear Regression and Ordinary Least Square Estimate

In pure Newton method, we use $\rho_t = 1$.

If we consider, $\rho_t = 1$

$$H = X^T X$$
$$g = X^T X w - X^T y$$

} These are derived from identities

Then, the Newton update for linear regression become

$$\begin{aligned} w_1 &= w_0 - H^{-1} g \\ &= w_0 - (X^T X)^{-1} (X^T X w_0 - X^T y) \\ &= w_0 - w_0 + (X^T X)^{-1} X^T y \end{aligned}$$

This is the OLS estimate and we achieve convergence in single step.

In case of logistic regression, more steps are needed to converge to global optimum.

§3.2 BFGS and other quasi-Newton methods

Quasi meaning seeming, apparently but not really.

Computing Hessian takes $O(D^2)$ space for very large problems. This might be daunting slow for systems with low memory or for even large datasets.

In order to alleviate the issue Broyden, Fletcher, Goldfarb, and Shanno came up with a method called BFGS which iteratively builds up an approximation to the Hessian using information from gradient.

The goal here is to find B_t such that $B_t \approx H_t$.

$$B_{t+1} = B_t + \left(\frac{y_t y_t^T}{y_t^T s_t} - \frac{(B_t s_t)(B_t s_t)^T}{s_t^T B_t s_t} \right)$$

→ Low Rank update

At B_0, $B_0 = I$

$y_t = g_t - g_{t-1}$

$s_t = \theta_t - \theta_{t-1}$

This is considered as rank-two update. What is rank-two update! The highlighted part in (.) is a low-rank update ie rank-2 A low-rank update enforces properties of Hessian while saving computational resources.

If B_0 is positive-definite, then B_{t+1} will remain positive definite, iff satisfies two conditions

1. Step size(ρ) is chosen via line search satisfying Armijo condition
   i.e., $\mathcal{L}(\theta+\rho d_t) \leq \mathcal{L}(\theta_t) + c\rho d_t^T \nabla \mathcal{L}(\theta_t)$
2. Curvature condition
   i.e., $\nabla \mathcal{L}(\theta+\rho d_t) \geq c_2 \rho d_t^T \nabla \mathcal{L}(\theta_t)$

These two conditions are called Wolfe conditions

Instead of approximating H_t via B_t for Newton method updates, we can approximate the H_t^{-1} directly. Let's denote $C_t \approx H_t^{-1}$

$$C_{t+1} = \left(I - \frac{s_t y_t^T}{y_t^T s_t}\right) C_t \left(I - \frac{y_t s_t^T}{y_t^T s_t}\right) + \frac{s_t s_t^T}{y_t^T s_t}$$

In this way, BFGS approximates H^{-1} by only considering most recent (s_t, y_t) pairs while ignoring older information. The new space complexity is therefore $O(MD)$ where M is the approximation dimension typically between 5-20

8.3.3 Trust Region Methods

When the objective function is nonconvex, then the Hessian may not be positive definite. Therefore, descent direction will not be $d_t = -H^{-1} g_t$.

So the Newton method may end up in local maximum instead of local minima because the H^{-1} no longer smooths the curvature appropriately and lead to invalid optimization process.

However, we can iterate over a local region where we can safely approximate the objective by a quadratic.

Let's call this region R_t

and $M_t(\delta)$ be the approximation to objective

$\hookrightarrow \delta = \theta - \theta_t$

We can solve for δ^* as

$$\delta^* = \underset{\delta \in R_t}{\operatorname{argmin}}\, M_t(\delta)$$

} within the region R_t, trying to find the optimal parameters θ or δ where objective function $M(\delta)$ is minimum.

This is called Trust Region Optimization

If we assume $M_t(\delta)$ is a quadratic approximation

$$M_t(\delta) = f(\theta_t) + g_t^T \delta + \frac{1}{2}\delta^T H_t \delta$$

$f(\theta_t)$: We still need Loss $f(\theta_t)$

$\frac{1}{2}\delta^T H_t \delta$: While ensuring the Hessian is always positive definite $x^T A x > 0$

It is common that R_t is a ball of radius r

i.e., $R_t = \{\delta : \|\delta\|_2 \leq r\}$

This indicates the set of points in the circle are distant by less than or equal to the radius of the circle.

And the maximum area of which these points could exist is $\|\delta\|_2^2 \leq \pi r^2$

Bounded within the area of the circle

Hence we can convert constrained optimization i.e., $\delta^* = \underset{\delta \in R_t}{\operatorname{argmin}}\, M_t(\delta)$ ($\delta \in R_t$: constraint)

to unconstrained optimization as follows:

$$\delta^* = \underset{\delta}{\operatorname{argmin}}\, M(\delta) + \lambda \|\delta\|_2^2$$

$$= \underset{\delta}{\operatorname{argmin}}\, g^T\delta + \frac{1}{2}\delta^T(H + \lambda I)\delta$$

($\lambda > 0$)

We solve δ as

$$\delta = -(H + \lambda I)^{-1} g$$

→ This is called Tikhonov Damping or Regularization or L2 regularization

Adding sufficiently large λI to H ensures δ is positive definite
Large λ makes all negative eigenvalues positive and all 0 values becomes λ.

8.4 Stochastic Gradient Descent

Stochastic => Randomly determined.

In stochastic optimization, the objective of a function would be to minimize the average value by

$$\mathcal{L}(\theta) = \mathbb{E}_{q(z)}[\mathcal{L}(\theta, z)]$$

z: Random input or set of training example
$q(z)$: Distribution of z

The gradient descent relies on entire data distribution of data to compute gradient and update parameter. In stochastic gradient descent, the optimization is independent of the data distribution. Thereby, the resulting optimization would be

$$\theta_{t+1} = \theta_t - \rho_t \nabla \mathcal{L}(\theta_t, z_t) = \theta_t - \rho_t g_t$$

$g_t = \nabla_\theta f(\theta_t)$
This means we're computing gradient for every sample or mini batch

This method is known as Stochastic Gradient Descent. The method will eventually converge to a stationary point, providing we decay the step size ρ_t at a certain rate.

8.4.1 Application to finite sum problems

In machine learning, the empirical risk minimization is given as:

$$\mathcal{L}(\theta_t) = \frac{1}{N}\sum_{n=1}^{N} l(y_n, f(x_n; \theta_t)) = \frac{1}{N}\sum_{n=1}^{N} f_n(\theta_t)$$

This is called finite sum problem. The gradient is given as

$$g_t = \frac{1}{N}\sum_{n=1}^{N} \nabla_\theta\, l(y_n, f(x_n, \theta_t))$$

g_t update requires all examples
This is slower

$$g_t = \frac{1}{|B_t|}\sum_{n \in B_t} \nabla_\theta\, l(y_n, f(x_n, \theta_t))$$

g_t update requires a small batch of examples
Plus, this is unbiased approximation.
This is faster

Minibatch

8.4.2 Example: SGD for fitting linear regression

The loss for linear regression is

$$\mathcal{L}(\theta) = \frac{1}{2N}\sum_{n=1}^{N}(x_n^T\theta - y_n)^2$$ → Mean square error loss

Why $1/2$?

Differentiating w.r.t θ for θ^2 would yield 2θ. So $1/2$ and 2 cancel out when gradient is being computed.

$$g_t = \frac{1}{N}\sum_{n=1}^{N}(\theta_t^T x_n - y_n)x_n$$

$$\theta_{t+1} = \theta_t - \rho_t\left[\frac{1}{N}\sum_{n=1}^{N}(\theta_t^T x_n - y_n)x_n\right]$$

With Batch size = 1,

$$\mathcal{L}(\theta) = \frac{1}{2}(x_n^T - \theta)^2$$

$$g_t = (\theta_t^T x_n - y_n)x_n$$

$$\theta_{t+1} = \theta_t - \rho\left[(\theta_t^T x_n - y_n)x_n\right]$$

This is the "least mean square algorithm" aka "delta rule" aka "Widrow-Hoff rule"

8.4.3 Choosing the step size (learning rate)

Choosing learning rate is essential for finding optimal convergence. A model with extremely low learning rate can underfit and model with very high learning rate may overfit.

There are few heuristics for choosing a good learning rate

Start with small: Start with a very low learning rate and gradually increase. Pick the learning rate with the lowest loss. For stability, choose learning rate which is slightly lower than the best loss(ρ).

Learning rate schedule: Adjust the learning rate over time. Common ways are:

Piecewise constant: $\rho_t = \rho_i$ if $t_i \le t \le t_{i+1}$

At each time point, the learning rate is reduced by a factor. For ex: $\rho_0 = \rho_0 \gamma^t$ where γ^t discounts the learning rate(ρ_t) at each threshold. This is also known as **Step decay**.

Exponential decay: $\rho_t = \rho_0 e^{-\lambda t}$. This is typically faster. The ρ decay at every timestep happens by factor $e^{-\lambda t}$.

Polynomial decay: $\rho_t = \rho_0 (\beta t + 1)^{-\alpha}$. Common choices of parameters are $\alpha = 0.5$ and $\beta = 1$. i.e.,

$$\rho_t = \rho_0 (1(t) + 1)^{-0.5} = \rho_t = \frac{\rho_0}{\sqrt{t+1}}$$

} aka Square root schedule

Learning rate Warmup: Here the learning rate is quickly increased and gradually decreased based on the surface. Initially the learning rate will be set very low until flatter surfaces are found. Then the learning rate is increased at flatter surfaces. Eventually the learning rate is decreased to zero to ensure convergence.

Cyclical learning rate: A minimum and maximum learning rate is choosen initially and half cycle is choosen to suggest how many times we would like to restart the learning rate back to lower values. So the learning rate varies between lower and upper bound constantly.

SGD with warm restarts: Store checkpoints after each learning rate cooldown and using all the checkpoint members as ensemble.

Line search for learning rate: Line search requires satisfying the Armijo condition test i.e., for a learning rate

$$f(\theta_t + \rho d_t) \leq f(\theta_t) + c \rho d_t^T \nabla f(\theta_t)$$

This would be tricky because noisy gradients will hamper the condition test. But it can be shown that line search could be helpful and make it to work if the variance of gradient noise goes to zero over time; This only happens if the model perfectly interpolates the training data.

Still unclear about few details here!

8.4.4 Iterate Averaging

The parameter estimates by SGD can be very unstable over time ie., the variance of the estimates tend to be high. To reduce the variance, we can compute average using

$$\bar{\theta}_t = \frac{1}{t}\sum_{i=1}^{t}\theta_i = \frac{1}{t}\theta_t + \frac{t-1}{t}\bar{\theta}_{t-1}$$

This is called iterate averaging or Polyak-Ruppert averaging. This averaging method has better convergence rate among SGD algorithms. In the case of linear regression, this method is equivalent to l_2-regularization.

An alternate extension is Stochastic Weight Averaging, which uses an equal average in conjuction modified learning rate schedule. Further suggests that SWA helps in faster convergence.

8.4.5 Variance Reduction

In previous section, we looked at reducing the variance of the parameters. Here we look at the variance of gradients itself. This can improve convergence from sub linear to linear.

SVRG

Stochastic Variance Reduced Gradient (SVRG) is used to reduce the variance by first estimating the gradient of whole batch and then used to compare the stochastic gradients.

Remember, batch gradient is gradient is computed based on whole batch error. whereas stochastic gradient is computed based on each individual sample error.

Let's denote, the full batch gradient as $\nabla\mathcal{L}(\tilde{\theta})$. (computed once per epoch)

stochastic gradient as $\nabla f_t(\theta_t)$ (computed for every step t)

Snapshot gradient as $\nabla\mathcal{L}_t(\tilde{\theta})$ ⇝ Slightly unclear when this is computed. (Probably every batch)

$$g_t = \nabla f_t(\theta_t) - \nabla f_t(\tilde{\theta}) + \nabla\mathcal{L}(\tilde{\theta})$$

$$E[\nabla f_t(\tilde{\theta})] = \nabla\mathcal{L}(\tilde{\theta})$$

At end of each epoch, we update snapshot parameters, $\tilde{\theta}$ based on the most recent value of θ_t, or a running average of the iterates, and update the expected baseline.

Iterations of SVRG are computationally faster than full batch Gradient Descent. Theoretical convergence of SVRG match with SGD.

SAGA

SAGA stands for "Stochastic Averaged Gradient Descent". In SVRG method we computed full batch gradient after every epoch. Here in this method, the full batch gradient is computed only once. However, we also store the gradients of each batch as well.

By saving these gradients, we replace the old gradients and replace it with new gradient and compute new global gradient average i.e., full batch gradient. This is called aggregated gradient.

Let $g_n^{local} = \nabla f_n(\theta_0)$

$$g^{avg} = \frac{1}{N}\sum_{n=1}^{N} g_n^{local}$$

$$g^t = \nabla f_n(\theta_t) - g_n^{local} + g^{avg}$$

This new value is updated to g_n^{local}. Then g^{avg} is recomputed.

In this method we only perform one large sweep at the start. Although, there is a large memory cost in storing the gradients, we just need to compute the full batch once and iterate.

This method is ideal when the features are sparse. In such cases the memory cost can be reasonable.

Application to deep learning

Variance reduction methods are effective for problems which are convex and can be fit with linear models. Deep learning problems tend to be complex, non-convex and require non-linear methods. Variance reduction methods fails to achieve convergence with its underlying assumptions where the data.

The variance reduction methods fails when loss differ randomly due to change in data or parameters. For example, variance reduction fails in case of batch normalization, data augmentation, and dropout.

In the work of Defazio et al. On the Ineffectiveness of Variance Reduced Optimization for Deep Learning

the authors explain that "the variance of the SVRG estimator is directly dependent on how similar the gradient is between the snapshot point and the current point (or iterate). If the weights of the current iterate lead to moving the current gradient iterate rapidly, then the snapshot point will be out of date. to provide meaningful variance reduction."

8.4.6 ## Preconditioned SGD

In preconditioned SGD,

$$\theta_{t+1} = \theta_t - \rho_t M_t^{-1} g_t$$

M_t: This is a preconditioning matrix which is positive definite.

Here we're basically approximating the Hessian. However, estimating the Hessian would be difficult with the noise in the gradient estimates. Hence most practitioners use a diagonal preconditioner M_t.

Assuming M_t has only diagonal values.

Adagrad

When an input consists of rarely occurring feature elements, for instance rare words in a sentence, the gradient elements corresponding to the sentence vector will be zero.

Infrequent words have information value in some of the use-cases. The informativeness of such rare features has led practitioners in NLP to craft feature weighting such as TF-IDF.

Adaptive Gradient (aka AdaGrad) give frequently occurring features very low learning rates and infrequent features high learning rates, where the intuition is that each time an infrequent feature is seen, the learner should take "notice". The update has the form.

$$\theta_{t+1,d} = \theta_{t,d} - \rho_t \frac{1}{\sqrt{s_{t,d} + \epsilon}} g_{t,d}$$

$\epsilon > 0$

$$s_{t,d} = \sum_{t'=1}^{t} g_{t',d}^2$$

In vector form,

$$\Delta\theta_t = -\rho_t \frac{1}{\sqrt{s_t + \epsilon}} g_t$$

This is equivalent to $M_t = \text{diag}(s_t + \epsilon)^{1/2}$

and hence AdaGrad is viewed as Preconditioned SGD.

Although, the ρ_t is still need to be chosen, the results are less sensitive to it compared to vanilla gradient descent. So the learning rate is fixed to $\rho_t = \rho_0$

RMSProp and ADADELTA

In ADAGRAD, as the denominator becomes larger the effective learning rate drops overtime. While reducing the learning rate helps over time, the denominator tend to become large quickly. This leads to poor convergence.

Exponentially Weighted Moving Average is an alternate way to compute the average gradient statistic. The EWMA weighs most recent values and diminishes the previous averages. EWMA for gradients is given as:

$$s_{t+1,d} = \beta s_{t,d} + (1-\beta) g_{t,d}^2 \qquad \text{where } s_{t,d} = \sum_{t=1}^{t} g_{t,d}^2$$

Generally we use $\beta = 0.9$,

$$\sqrt{s_{t,d}} \approx \text{RMS}(g_{1:t,d}) = \sqrt{\frac{1}{t}\sum_{\tau=1}^{t} g_{\tau,d}^2}$$

} Root Mean Square of gradient

This is known as RMSProp and the update rule is similar to ADAGRAD

$$\Delta\theta_t = -\rho_t \frac{1}{\sqrt{s_t + \epsilon}} g_t$$

ADADELTA

In AdaDelta, along with RMS of gradients, we also consider RMS of parameter updates which would be defined as

$$\Delta\theta_t = -\rho_t \frac{\sqrt{\delta_{t-1} + \epsilon}}{\sqrt{s_t + \epsilon}} g_t \qquad \text{where } \delta_t = \beta \delta_{t-1} + (1-\beta)\Delta\theta_t^2$$

One thing to remember is these adaptive learning rates need not decrease with time. In such case we need a tunable setup for adaptive learning.

ADAM

ADAM stands for "adaptive moment estimation."

RMSPROP + MOMENTUM = ADAM

$$m_t = \beta_1 m_{t-1} + (1-\beta_1) g_t \rightsquigarrow \text{EWMA of momentum}$$
$$s_t = \beta_2 s_{t-1} + (1-\beta_2) g_t^2 \rightsquigarrow \text{EWMA of squared gradient}$$

β_1 is the EWMA coefficients of momentum usually set to $\beta_1 = 0.9$

β_2 is the EWMA coefficients of squared gradients. $\beta_2 = 0.999$

$\epsilon = 10^{-6}$

Final update:

$$\Delta\theta_t = -\rho_t \frac{1}{\sqrt{s_t + \epsilon}} m_t$$

Similar to ADADELTA, convergence is not guaranteed as the adaptive learning rate decreases over time.

If we initialize the $m_0 = s_0 = 0$, the initial estimates will be smaller values. Authors recommend bias correction as follows:

How the values will be smaller?

$$m_t = \beta_1 m_{t-1} + (1-\beta_1) g_1$$
$$m_1 = \beta_1 0 + (1-\beta_1) g_1$$
$$m_1 = (1-\beta_1) g_1$$

m_1 will be smaller than $\hat{m}_1$

$$\hat{m}_1 = \frac{m_1}{1-\beta} = \frac{(1-\beta_1)g_1}{(1-\beta_1)} = g_1$$

$$m_1 < \hat{m}_1 \text{ or } (1-\beta_1) g_1 < g_1$$

Same applies to s_t

$$\hat{m}_t = \frac{m_t}{1-\beta_1^t}$$

$$\hat{s}_t = \frac{s_t}{1-\beta_2^t}$$

Full matrix Adagrad

In the previous methods, we looked at how adaptive learning rate is achieved by $\rho_0 M_t^{-1}$ where M_t is a preconditioned matrix. However, the M_t is ill-conditioned due to the correlation of parameters in these methods. Hence convergence would be slower than vanilla SGD (in some cases).

For faster convergence over SGD, the following preconditioning matrix as

$$M_t = \left[(G_t G_t^T)^{1/2} + \epsilon I_D \right]^{-1}$$

where

$$G_t = [g_t, \cdots g_1]$$

Here $g_i = \nabla_v c(x_i)$ $g_i \in R^D$, $M_t \in R^{D \times D}$

Expensive to compute, store & invert.

Shampoo makes a block diagonal approximation of M and exploits kronecker product structure to effeciently invert it

How kronecker products could be used to invert?

Skipping 8.5, 8.6 temporarily

8.7 Bound Optimization

Here we discuss a class of algorithms for optimization

MM optimization

MM: Majorize – Minorize algorithm for minimization

MM: Minorize – Majorize algorithm for maximization

Expectation Maximization is a special case of MM algorithm

8.7.1 The General Algorithm

Let's assume we want to maximize log likelihood w.r.to its parameters 'θ' i.e., $LL(\theta)$.

Let θ^t represent the parameter θ.

Let Q be a surrogate function such that $Q(\theta, \theta^t)$ indicate the θ^t (approximating parameter of θ) and θ (the parameter of the function i.e., Maximum likelihood function) such that

$$Q(\theta, \theta^t) \leq LL(\theta) \quad \text{and} \quad Q(\theta^t, \theta^t) = LL(\theta^t)$$

If the above conditions are met, then it means Q minorizes LL.

The update step would be:

$$\theta^{t+1} = \underset{\theta}{\operatorname{argmax}}\ Q(\theta, \theta^t)$$

8.7.2 EM Algorithm

Expectation Maximization Algorithm is a bound optimization algorithm. This algorithm is designed to compute MLE or MAP for datasets which have missing data or hidden variables or both.

Two steps in EM algorithm

E-step : Estimate the hidden variables or missing values of the data

M-step : Maximize the MLE by computing the outcome from the fully observed data.

The two steps need to be iterated till convergence, since the E-step depends on M-step and M-step depends on E-step.

Lower Bound

Remember, the goal of the EM algorithm is to maximize the log likelihood (MLE) of the observed data

$$LL(\theta) = \sum_{n=1}^{N} \log p(y_n|\theta) = \sum_{n=1}^{N} \log \left[\sum_{z_n} p(y_n, z_n|\theta) \right]$$

→ hidden variables

Finding z_n is hard. A work around is find distribution which approximates the original distribution.

$$LL(\theta) = \sum_{n=1}^{N} \log \left[\sum_{z_n} q_n(z_n) \frac{p(y_n, z_n|\theta)}{q_n(z_n)} \right]$$

→ Distribution of each feature or latent variable. An efficient approximating function also imputes the missing values in the feature.

Using Jensen inequality, we can push the log inside the expectation to get following lower bound on the log likelihood:

$$LL(\theta) \geq \sum_n \sum_{z_n} q_n(z_n) \log \frac{p(y_n, z_n|\theta)}{q_n(z_n)}$$

$$= \sum_n \mathbb{E}_{q_n}[\log p(y_n, z_n|\theta)] + H(q_n)$$

How this happened?

$\log AB = \log A + \log B$

$\log A = \mathbb{E}_{q_n}[\log p(y_n, z_n|\theta)]$

$\mathbb{E}_{q_n}[\log p(y_n, z_n|\theta)] = \log \frac{p(y_n, z_n|\theta)}{q_n(z_n)}$

$\mathbb{E}_{q_n}$ means Average estimate. So it makes sense

We can define, $\sum_n \mathcal{E}(\theta, q_n|y_n) = \sum_n \mathbb{E}_{q_n}[\log p(y_n, z_n|\theta)] + H(q_n)$

or

$$\sum_n \mathcal{E}(\theta, q_n|y_n) \triangleq \mathcal{E}(\theta, \{q_n\}|D)$$

This is called Evidence Lower Bound since this is the lower bound on the marginal likelihood. i.e, $p(y_{1:N}|\theta)$ ↳ evidence

E step

The evidence lower bound is the sum of N terms (or features) i.e.,

$$\mathcal{E}(\theta, q_n | y_n) = \sum_{z_n} q_n(z_n) \log \frac{p(y_n, z_n | \theta)}{q_n(z_n)}$$

$$= \sum_{z_n} q_n(z_n) \log \frac{p(z_n | y_n, \theta)\, p(y_n | \theta)}{q_n(z_n)}$$

$$= \sum_{z_n} q_n(z_n) \log \frac{p(z_n | y_n, \theta)}{q_n(z_n)} + \sum_{z_n} q_n(z_n) \log p(y_n | \theta)$$

$$= -KL(q_n(z_n) \| p(z_n | y_n, \theta)) + \log p(y_n | \theta)$$

$$KL(q \| p) \triangleq \sum_z q(z) \log \frac{q(z)}{p(z)} \sim \text{Kullback–Leibler divergence}$$

Required: $KL(q\|p) \geq 0$

if $KL(q\|p) = 0$, then $q = p$

In missing value cases, $KL(q\|p)$ may be greater than 0, given $q = p$ for non-missing features.

Rewriting ELBO,

$$\mathcal{E}(\theta, q_n | y_n) = -KL(q_n(z_n) \| p(z_n | y_n, \theta)) + \log p(y_n | \theta)$$

when KL-divergence = 0,

$$\mathcal{E}(\theta, q_n | y_n) = \log p(y_n | \theta)$$

Hence, the tight lower bound for each feature would be:

$$\mathcal{E}(\theta, \{q_n^*\} | D) = \sum_n \log p(y_n | \theta) = LL(\theta | D)$$

In terms of bound optimization,

$$Q(\theta, \theta^t) = \mathcal{E}(\theta, \{p(z_n | y_n; \theta^t)\})$$

Hence we have

$$Q(\theta, \theta^t) \leq LL(\theta) \quad \text{and} \quad Q(\theta^t, \theta^t) = LL(\theta^t)$$

The effect of Jensen's inequality

$$LL(\theta) \geq \sum_n \mathcal{E}(\theta, q_n | y_n)$$

M step

In the E-step, we computed θ and q_n^t (distribution of each feature). Here we try to use θ and q_n^t to maximize the log-likelihood (LL).

$$LL^t(\theta) = \sum_n \mathbb{E}_{q_n^t(z_n)}\left[\log p(y_n, z_n \mid \theta)\right]$$

$H(q_n)$ is dropped as it will be constant.

This is called expected complete data log likelihood.

Maximizing the expected complete data log likelihood

$$\theta^{t+1} = \arg\max_\theta \sum_n \mathbb{E}_{q_n^t}\left[\log p(y_n, z_n \mid \theta)\right]$$

From this expected statistics, we can say that there is in fact no need to return the full set of posterior distributions $\{q(z_n)\}$.

EM & Gaussian Mixture Models use case for clustering could further explain the above statement. We would revisit these again after the Probability & statistics chapters.