Optimization Techniques for ML (contd)

CS771: Introduction to Machine Learning

Piyush Rai

Optimization Problems in ML

- The general form of an optimization problem in ML will usually be

$$w_{opt} = \arg\min_{\boldsymbol{w}} L(\boldsymbol{w})$$

$L(\boldsymbol{w})$ may denote the training loss, or training loss + regularizer term

or

$$w_{opt} = \arg\min_{w \in \boldsymbol{C}} L(\boldsymbol{w})$$

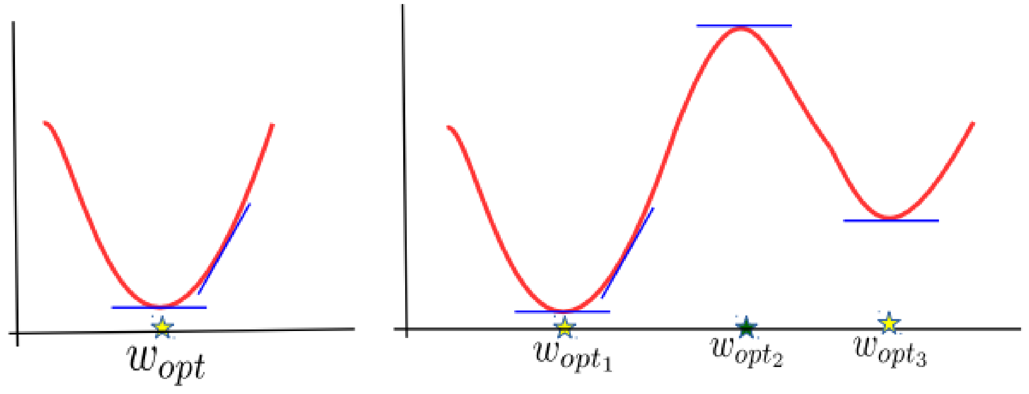
Training loss with a constraint on $\boldsymbol{w}$

- $\boldsymbol{C}$ is the constraint set that the solution must belong to, e.g.,
  - Non-negativity constraint: All entries in $\boldsymbol{w_{opt}}$ must be non-negative
  - Sparsity constraint: $\boldsymbol{w_{opt}}$ is a sparse vector with at most $\boldsymbol{K}$ non-zeros
- Constrained opt. probs can be converted into unconstrained opt. (will see later)
- For now, assume we have an unconstrained optimization problem

Methods for Solving Optimization Problems

Method 1: Using First-Order Optimality

- Very simple. Already used this approach for linear and ridge regression



Called "first order" since only gradient is used and gradient provides the first order info about the function being optimized

The approach works only for very simple problems where the objective is convex and there are no constraints on the values $\boldsymbol{w}$ can take

- First order optimality: The gradient $\boldsymbol{g}$ must be equal to zero at the optima

$$\boldsymbol{g} = \nabla_{\boldsymbol{w}}[L(\boldsymbol{w})] = \mathbf{0}$$

E.g., linear/ridge regression, but not for logistic/softmax regression

- Sometimes, setting $\boldsymbol{g} = \mathbf{0}$ and solving for $\boldsymbol{w}$ gives a closed form solution
- If closed form solution is not available, the gradient vector $\boldsymbol{g}$ can still be used in iterative optimization algos, like gradient descent

Method 2: Iterative Optimiz. via Gradient Descent

Can I used this approach to solve maximization problems?

For max. problems we can use gradient ascent $\boldsymbol{w}^{(t+1)} = \boldsymbol{w}^{(t)} + \eta_t \boldsymbol{g}^{(t)}$

Iterative since it requires several steps/iterations to find the optimal solution

Fact: Gradient gives the direction of steepest change in function's value

Will move in the direction of the gradient

For convex functions, GD will converge to the global minima

Good initialization needed for non-convex functions

Gradient Descent

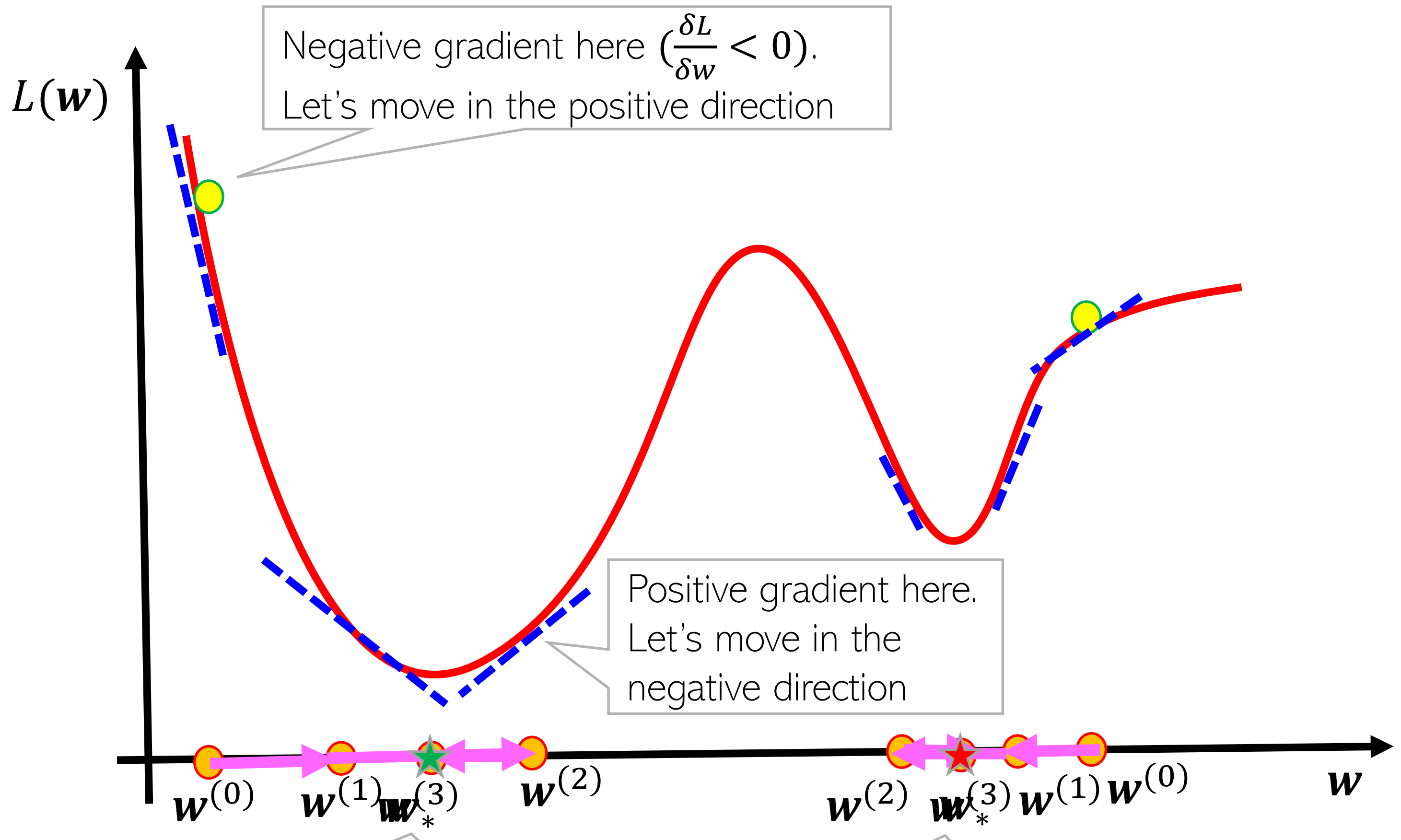
The learning rate very imp. Should be set carefully (fixed or chosen adaptively). Will discuss some strategies later

- Initialize $\boldsymbol{w}$ as $\boldsymbol{w}^{(0)}$
- For iteration $t = 0,1,2,\ldots$ (or until convergence)
  - Calculate the gradient $\boldsymbol{g}^{(t)}$ using the current iterates $\boldsymbol{w}^{(t)}$
  - Set the learning rate η_t
  - Move in the opposite direction of gradient

Will see the justification shortly

$$\boldsymbol{w}^{(t+1)} = \boldsymbol{w}^{(t)} - \eta_t \boldsymbol{g}^{(t)}$$

Gradient Descent: An Illustration



Negative gradient here ($\frac{\delta L}{\delta w} < 0$). Let's move in the positive direction

Positive gradient here. Let's move in the negative direction

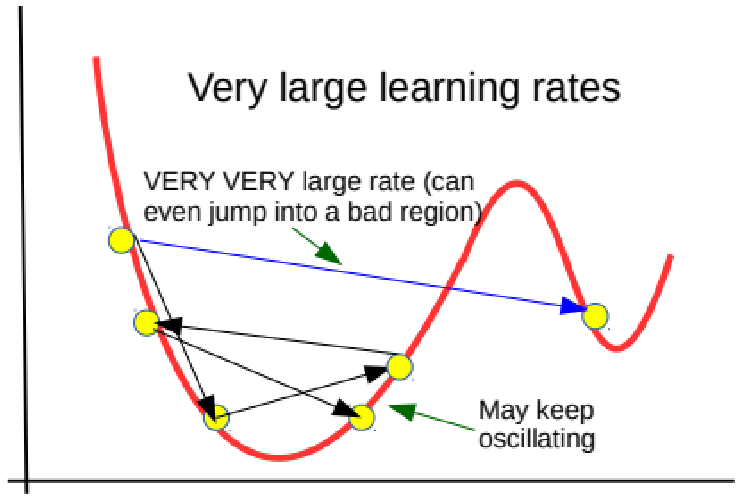
Woohoo! ☺ Global minima found!!!

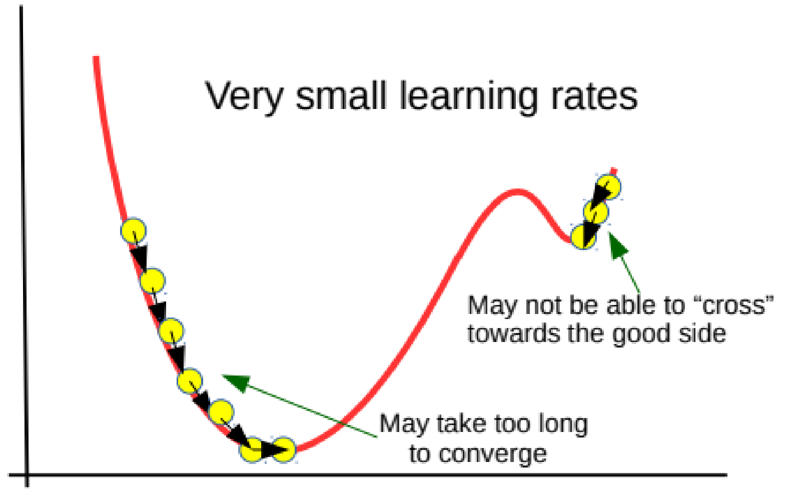
GD thanks you for the good initialization ☺

Stuck at a local minima ☹

Good initialization is very important

Learning rate is very important


Very large learning rates
VERY VERY large rate (can even jump into a bad region)
May keep oscillating


Very small learning rates
May not be able to "cross" towards the good side
May take too long to converge

GD: An Example

- Let's apply GD for least squares linear regression

$$\boldsymbol{w}_{ridge} = \text{arg min}_{\boldsymbol{w}}\, L_{reg}(\boldsymbol{w}) = \text{arg min}_{\boldsymbol{w}}\, \frac{1}{N}\sum_{n=1}^{N}(y_n - \boldsymbol{w}^\top \boldsymbol{x}_n)^2$$

- The gradient: $\boldsymbol{g} = -\frac{2}{N}\sum_{n=1}^{N}(y_n - \boldsymbol{w}^\top \boldsymbol{x}_n)\boldsymbol{x}_n$
- Each GD update will be of the form

$$\boldsymbol{w}^{(t+1)} = \boldsymbol{w}^{(t)} + \eta_t \frac{2}{N}\sum_{n=1}^{N}\left(y_n - \boldsymbol{w}^{(t)^\top}\boldsymbol{x}_n\right)\boldsymbol{x}_n$$

Prediction error of current model $\boldsymbol{w}^{(t)}$ on the n^{th} training example

Training examples on which the current model's error is large contribute more to the update

- Exercise: Assume $N = 1$, and show that GD update improves prediction on the training input $(\boldsymbol{x}_n, y_n)$, i.e, y_n is closer to $\boldsymbol{w}^{(t+1)^\top}\boldsymbol{x}_n$ than to $\boldsymbol{w}^{(t)^\top}\boldsymbol{x}_n$
  - This is sort of a proof that GD updates are "corrective" in nature (and it actually is true not just for linear regression but can also be shown for various other ML models)

Faster GD: Stochastic Gradient Descent (SGD)

- Consider a loss function of the form $\boldsymbol{L}(\boldsymbol{w}) = \frac{1}{N}\sum_{n=1}^{N} \ell_n(\boldsymbol{w})$

Writing as an average instead of sum. Won't affect minimization of $L(\boldsymbol{w})$

- The gradient in this case can be written as

Expensive to compute – requires doing it for all the training examples in each iteration ☹

$$\boldsymbol{g} = \nabla_{\boldsymbol{w}} L(w) = \nabla_{\boldsymbol{w}}\left[\frac{1}{N}\sum_{n=1}^{N} \ell_n(\boldsymbol{w})\right] = \frac{1}{N}\sum_{n=1}^{N} \boldsymbol{g}_n$$

Gradient of the loss on n^{th} training example

- Stochastic Gradient Descent (SGD) approximates $\boldsymbol{g}$ using a single training example

- At iter. $\boldsymbol{t}$, pick an index $\boldsymbol{i} \in \{1,2,\ldots,N\}$ uniformly randomly and approximate $\boldsymbol{g}$ as

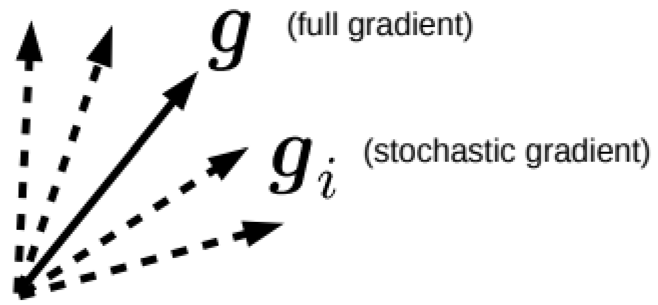
$$\boldsymbol{g} \approx \boldsymbol{g}_i = \nabla_{\boldsymbol{w}} \ell_i(\boldsymbol{w})$$

Can show that $\boldsymbol{g}_i$ is an unbiased estimate of $\boldsymbol{g}$, i.e.,$\mathbb{E}[\boldsymbol{g}_i] = \boldsymbol{g}$

- May take more iterations than GD to converge but each iteration is much faster ☺
  - SGD per iter cost is $O(D)$ whereas GD per iter cost is $O(ND)$

Minibatch SGD

- Gradient approximation using a single training example may be noisy



The approximation may have a high variance – may slow down convergence, updates may be unstable, and may even give sub-optimal solutions (e.g., local minima where GD might have given global minima)

- We can use $B > 1$ unif. rand. chosen train. ex. with indices $\{i_1, i_2, \ldots, i_B\} \in \{1,2,\ldots,N\}$
- Using this "minibatch" of examples, we can compute a minibatch gradient

$$\boldsymbol{g} \approx \frac{1}{B}\sum_{b=1}^{B} \boldsymbol{g}_{i_b}$$

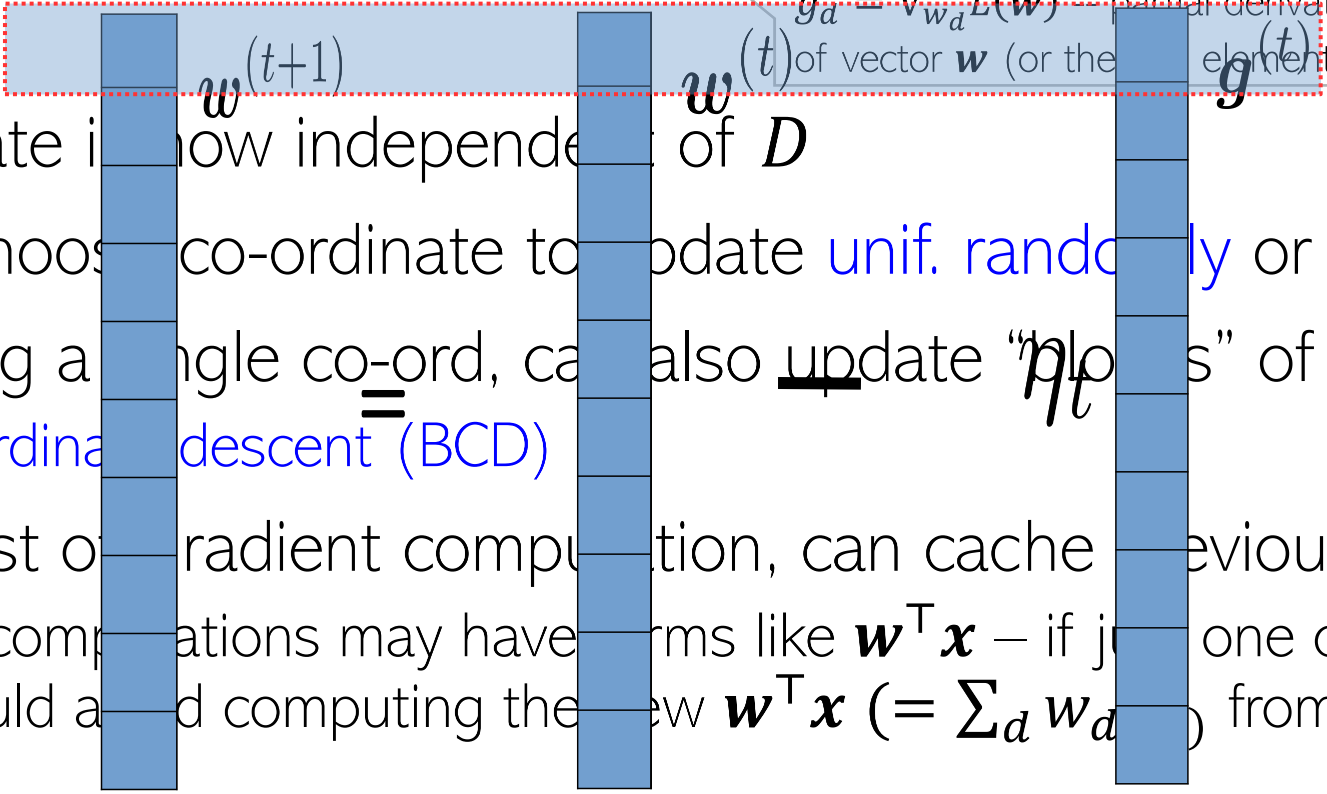
- Averaging helps in reducing the variance in the stochastic gradient
- Time complexity is $O(BD)$ per iteration in this case

Co-ordinate Descent (CD)

- Standard gradient descent update for : $\boldsymbol{w}^{(t+1)} = \boldsymbol{w}^{(t)} - \eta_t \boldsymbol{g}^{(t)}$
- CD: In each iter, update only one entry (co-ordinate) of $\boldsymbol{w}$. Keep all others fixed

$$w_d^{(t+1)} = w_d^{(t)} - \eta_t g_d^{(t)} \qquad d \in \{1,2,\ldots,D\}$$

$g_d = \nabla_{w_d} L(\boldsymbol{w})$ -- partial derivative w.r.t. the d^{th} element of vector $\boldsymbol{w}$ (or the element of the gradient vector g)



- Cost of each update i now independe of D
- In each iter, can choos co-ordinate to pdate unif. rando ly or in cyclic order
- Instead of updating a ngle co-ord, ca also update "blo s" of co-ordinates
  - Called block co-ordina descent (BCD)
- To avoid $O(D)$ cost o radient compu tion, can cache evious computations
  - Recall that grad. comp ations may have rms like $\boldsymbol{w}^\top \boldsymbol{x}$ – if j one co-ordinate of $\boldsymbol{w}$ changes, we should a d computing the ew $\boldsymbol{w}^\top \boldsymbol{x}$ $(= \sum_d w_d$ $)$ from scratch

Alternating Optimization (ALT-OPT)

- Consider opt. problems with several variables, say two variables $\boldsymbol{w}_1$ and $\boldsymbol{w}_2$

$$\{\hat{\boldsymbol{w}}_1, \hat{\boldsymbol{w}}_2\} = \arg\min_{\boldsymbol{w}_1, \boldsymbol{w}_2} \mathcal{L}(\boldsymbol{w}_1, \boldsymbol{w}_2)$$

- Often, this "joint" optimization is hard/impossible to solve
- We can take an alternating optimization approach to solve such problems

ALT-OPT

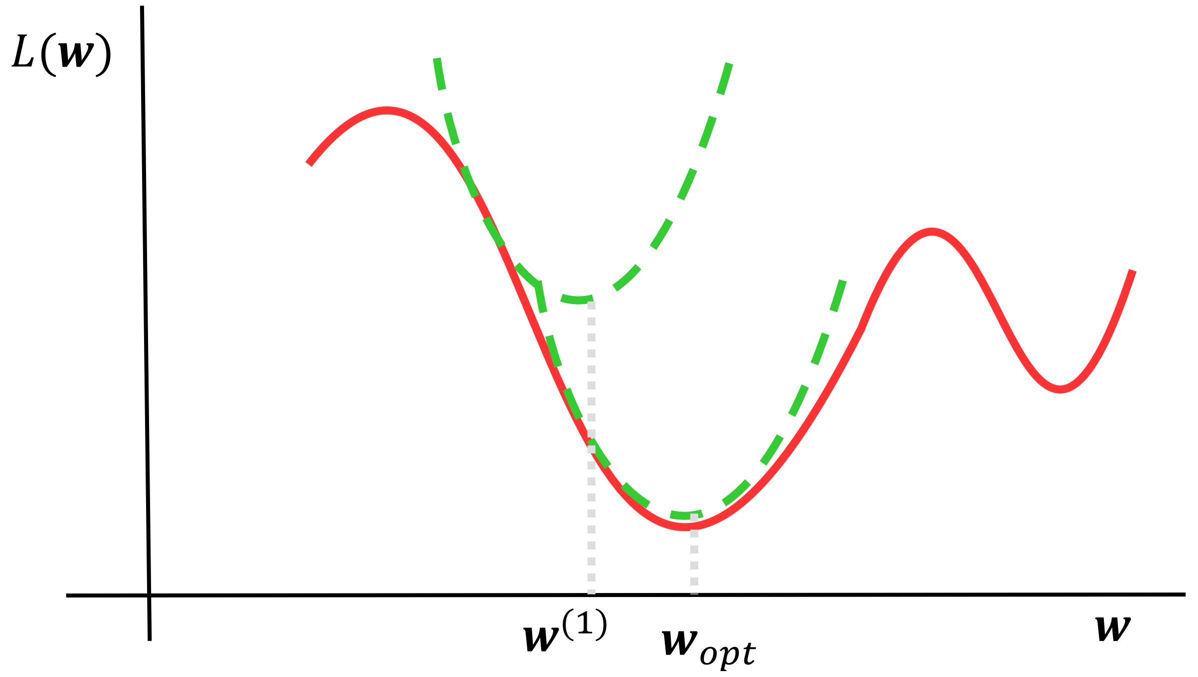
1. Initialize one of the variables, e.g., $\boldsymbol{w}_2 = \boldsymbol{w}_2^{(0)}, t = 0$
2. Solve $\boldsymbol{w}_1^{(t+1)} = \arg\min_{\boldsymbol{w}_1} \mathcal{L}(\boldsymbol{w}_1, \boldsymbol{w}_2^{(t)})$ //$\boldsymbol{w}_2$ "fixed" at its most recent value $\boldsymbol{w}_2^{(t)}$
3. Solve $\boldsymbol{w}_2^{(t+1)} = \arg\min_{\boldsymbol{w}_2} \mathcal{L}(\boldsymbol{w}_1^{(t+1)}, \boldsymbol{w}_2)$ //$\boldsymbol{w}_1$ "fixed" at its most recent value $\boldsymbol{w}_1^{(t+1)}$
4. $t = t + 1$. Go to step 2 if not converged yet.

- Usually converges to a local optima. But very very useful. Will see examples later
  - Also related to the Expectation-Maximization (EM) algorithm which we will see later

Second Order Methods: Newton’s Method

- Unlike GD and its variants, Newton’s method uses second-order information (second derivative, a.k.a. the Hessian). Iterative method, just like GD
- Given current $\boldsymbol{w}^{(t)}$, minimize the quadratic (second-order) approx. of $L(\boldsymbol{w})$

$$\boldsymbol{w}^{(t+1)} = \arg\min_{\boldsymbol{w}} \left[L(\boldsymbol{w}^{(t)}) + \nabla L(\boldsymbol{w}^{(t)})^\top (\boldsymbol{w} - \boldsymbol{w}^{(t)}) + \frac{1}{2}(\boldsymbol{w} - \boldsymbol{w}^{(t)})^\top \nabla^2 L(\boldsymbol{w}^{(t)})\,(\boldsymbol{w} - \boldsymbol{w}^{(t)})\right]$$



Show that $\boldsymbol{w}^{(t+1)} = \boldsymbol{w}^{(t)} - \left(\nabla^2 L(\boldsymbol{w}^{(t)})\right)^{-1} \nabla L(\boldsymbol{w}^{(t)})$
$= \boldsymbol{w}^{(t)} - (\boldsymbol{H}^{(t)})^{-1} \boldsymbol{g}^{(t)}$

Converges much faster than GD (very fast for convex functions). Also no “learning rate”. But per iteration cost is slower due to Hessian computation and inversion

Faster versions of Newton’s method also exist, e.g., those based on approximating Hessian using previous gradients (see L-BFGS which is a popular method)

Coming up next

- Constrained optimization
- Optimizing non-differentiable functions
- Some practical issue in optimization for ML

Constrained Optimization

Projected Gradient Descent

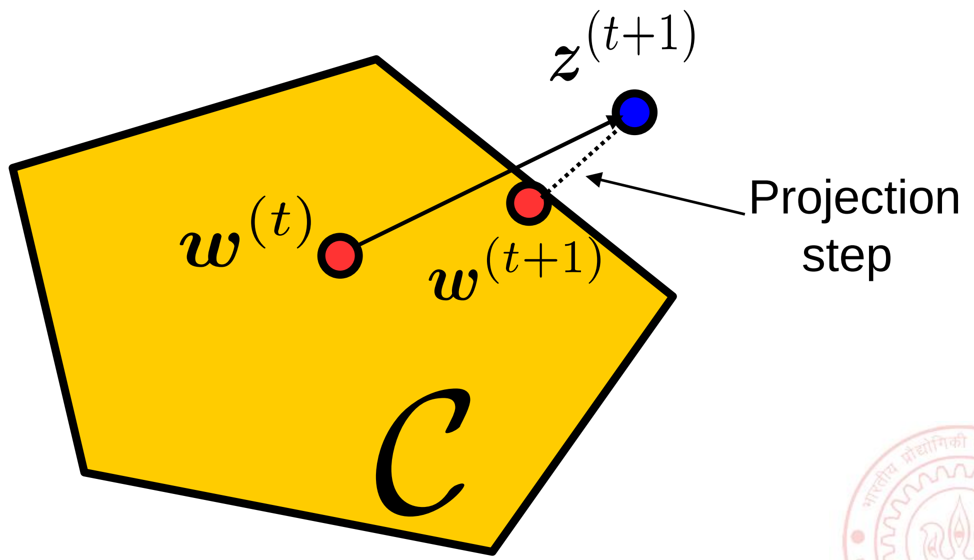
- Consider an optimization problem of the form

$$w_{opt} = \arg\min_{w \in \mathcal{C}} L(\boldsymbol{w})$$

- Projected GD is very similar to GD with an extra projection step

- Each iteration $\boldsymbol{t}$ will be of the form
  - Perform update: $\boldsymbol{z}^{(t+1)} = \boldsymbol{w}^{(t)} - \eta_t \boldsymbol{g}^{(t)}$
  - Check if $\boldsymbol{z}^{(t+1)}$ satisfies constraints
    - If $\boldsymbol{z}^{(t+1)} \in \mathcal{C}$, set $\boldsymbol{w}^{(t+1)} = \boldsymbol{z}^{(t+1)}$
    - If $\boldsymbol{z}^{(t+1)} \notin \mathcal{C}$, project as $\boldsymbol{w}^{(t+1)} = \Pi_{\mathcal{C}}[\boldsymbol{z}^{(t+1)}]$

Projection operator



Projected GD: How to Project?

- Here projecting a point means finding the “closest” point from the constraint set

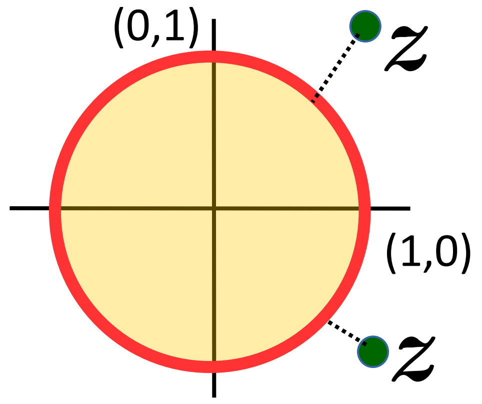
$$\Pi_{\mathcal{C}}[\boldsymbol{z}] = \arg\min_{\boldsymbol{w}\in\mathcal{C}} ||\boldsymbol{z} - \boldsymbol{w}||^2$$

Another constrained optimization problem! But simpler to solve! ☺

- For some sets $\mathcal{C}$, the projection step is easy

Projected GD commonly used only when the projection step is simple and efficient to compute

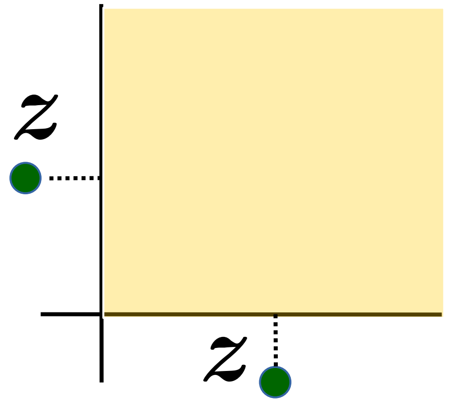
$\mathcal{C}$: Unit radius ℓ_2 ball



Projection = Normalize to unit Euclidean length vector

$$\hat{\mathbf{x}} = \begin{cases} \mathbf{x} & \text{if } \|\mathbf{x}\|_2 \leq 1 \\ \frac{\mathbf{x}}{\|\mathbf{x}\|_2} & \text{if } \|\mathbf{x}\|_2 > 1 \end{cases}$$

$\mathcal{C}$: Set of non-negative reals



Projection = Set each negative entry in $\boldsymbol{z}$ to be zero

$$\hat{\mathbf{x}}_i = \begin{cases} \mathbf{x}_i & \text{if } \mathbf{x}_i \geq 0 \\ 0 & \text{if } \mathbf{x}_i < 0 \end{cases}$$

Constrained Opt. via Lagrangian

- Consider the following constrained minimization problem (using f instead of L)

$$\hat{\boldsymbol{w}} = \arg\min_{\boldsymbol{w}} f(\boldsymbol{w}), \quad \text{s.t.} \quad g(\boldsymbol{w}) \leq 0$$

- Note: If constraints of the form $g(\boldsymbol{w}) \geq 0$, use $-g(\boldsymbol{w}) \leq 0$
- Can handle multiple inequality and equality constraints too (will see later)
- Can transform the above into the following equivalent <u>unconstrained</u> problem

$$\hat{\boldsymbol{w}} = \arg\min_{\boldsymbol{w}} f(\boldsymbol{w}) + c(\boldsymbol{w})$$

$$c(\boldsymbol{w}) = \max_{\alpha \geq 0} \alpha g(\boldsymbol{w}) = \begin{cases} \infty, & \text{if } g(\boldsymbol{w}) > 0 \quad \text{(constraint violated)} \\ 0 & \text{if } g(\boldsymbol{w}) \leq 0 \quad \text{(constraint satisfied)} \end{cases}$$

- Our problem can now be written as

$$\boxed{\hat{\boldsymbol{w}} = \arg\min_{\boldsymbol{w}} \left\{ f(\boldsymbol{w}) + \arg\max_{\alpha \geq 0} \alpha g(\boldsymbol{w}) \right\}}$$

Constrained Opt. via Lagrangian

The Lagrangian: $\mathcal{L}(\boldsymbol{w}, \boldsymbol{\alpha})$

- Therefore, we can write our original problem as

$$\hat{\boldsymbol{w}} = \arg\min_{\boldsymbol{w}} \left\{ f(\boldsymbol{w}) + \arg\max_{\alpha \geq 0} \; \alpha g(\boldsymbol{w}) \right\} = \arg\min_{\boldsymbol{w}} \left\{ \arg\max_{\alpha \geq 0} \{ f(\boldsymbol{w}) + \alpha g(\boldsymbol{w}) \} \right\}$$

- The Lagrangian is now optimized w.r.t. $\boldsymbol{w}$ and $\boldsymbol{\alpha}$ (Lagrange multiplier)
- We can define Primal and Dual problem as

$$\hat{\boldsymbol{w}}_P = \arg\min_{\boldsymbol{w}} \left\{ \arg\max_{\alpha \geq 0} \{ f(\boldsymbol{w}) + \alpha g(\boldsymbol{w}) \} \right\} \qquad \text{(primal problem)}$$

$$\hat{\boldsymbol{w}}_D = \arg\max_{\alpha \geq 0} \left\{ \arg\min_{\boldsymbol{w}} \{ f(\boldsymbol{w}) + \alpha g(\boldsymbol{w}) \} \right\} \qquad \text{(dual problem)}$$

Both equal if $f(\boldsymbol{w})$ and the set $g(\boldsymbol{w}) \leq 0$ are convex

$$\alpha_D g(\hat{\boldsymbol{w}}_D) = 0$$

complimentary slackness/Karush-Kuhn-Tucker (KKT) condition

Constrained Opt. with Multiple Constraints

- We can also have multiple inequality and equality constraints

$$\begin{aligned} \hat{\boldsymbol{w}} &= \arg\min_{\boldsymbol{w}} f(\boldsymbol{w}) \\ \text{s.t.} \quad & g_i(\boldsymbol{w}) \leq 0, \quad i = 1, \ldots, K \\ & h_j(\boldsymbol{w}) = 0, \quad j =, 1, \ldots, L \end{aligned}$$

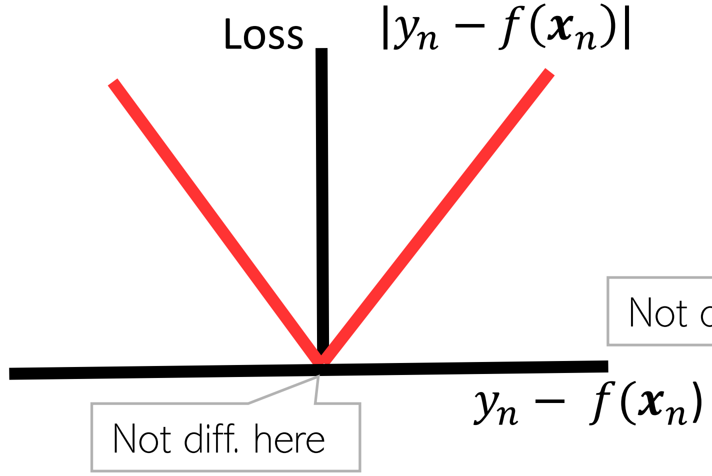
- Introduce Lagrange multipliers $\boldsymbol{\alpha} = [\alpha_1, \alpha_2, \ldots, \alpha_K]$ and $\boldsymbol{\beta} = [\beta_1, \beta_2, \ldots, \beta_L]$
- The Lagrangian based primal and dual problems will be

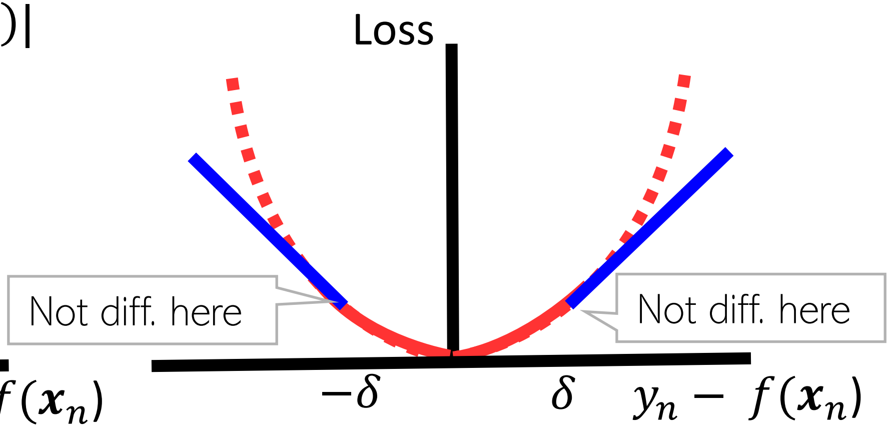
$$\begin{aligned} \hat{\boldsymbol{w}}_P &= \arg\min_{\boldsymbol{w}}\{\arg\max_{\boldsymbol{\alpha}\geq 0, \boldsymbol{\beta}}\{f(\boldsymbol{w}) + \sum_{i=1}^{K}\alpha_i g_i(\boldsymbol{w}) + \sum_{j=1}^{L}\beta_j h_j(\boldsymbol{w})\}\} \\ \hat{\boldsymbol{w}}_D &= \arg\max_{\boldsymbol{\alpha}\geq 0, \boldsymbol{\beta}}\{\arg\min_{\boldsymbol{w}}\{f(\boldsymbol{w}) + \sum_{i=1}^{K}\alpha_i g_i(\boldsymbol{w}) + \sum_{j=1}^{L}\beta_j h_j(\boldsymbol{w})\}\} \end{aligned}$$

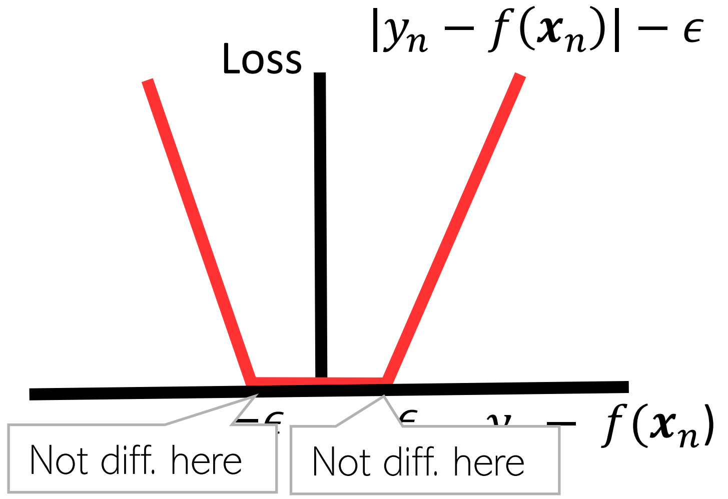
Optimization of Non-differentiable Functions

Dealing with Non-differentiable Functions

- In many ML problems, the objective function will be non-differentiable
- Some examples that we have already seen: Linear regression with absolute loss, or Huber loss, or $\boldsymbol{\epsilon}$-insensitive loss; even $\boldsymbol{\ell_1}$ norm regularizer is non-diff



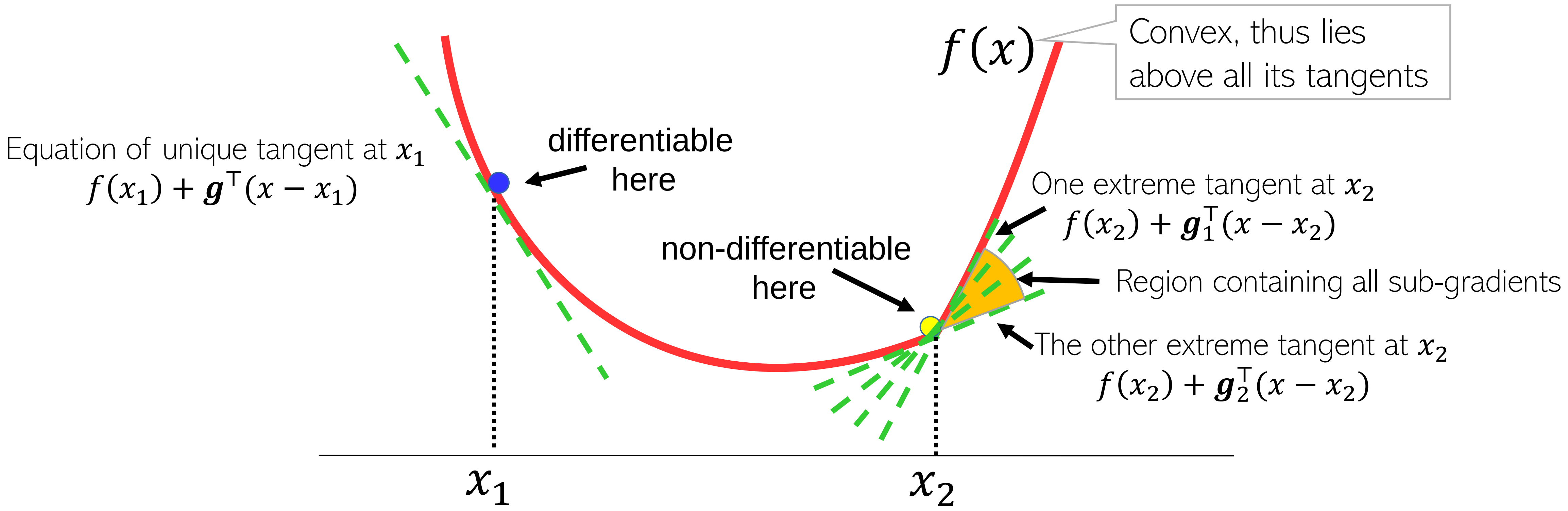




- Basically, any function in which there are points with kink is non-diff
  - At such points, the function is non-differentiable and thus gradients not defined
  - Reason: Can't define a unique tangent at such points

Sub-gradients

- For convex non-diff fn, can define sub-gradients at point(s) of non-differentiabilty



Convex, thus lies above all its tangents
$f(x)$
Equation of unique tangent at x_1
$f(x_1) + \boldsymbol{g}^\top(x - x_1)$
differentiable here
non-differentiable here
One extreme tangent at x_2
$f(x_2) + \boldsymbol{g}_1^\top(x - x_2)$
Region containing all sub-gradients
The other extreme tangent at x_2
$f(x_2) + \boldsymbol{g}_2^\top(x - x_2)$
x_1
x_2

- For a convex, non-diff function $f(\boldsymbol{x})$, sub-gradient at $\boldsymbol{x}_*$ is <u>any</u> vector $\boldsymbol{g}$ s.t. $\forall \boldsymbol{x}$

$$f(x) \geq f(\boldsymbol{x}_*) + \boldsymbol{g}^\top(\boldsymbol{x} - \boldsymbol{x}_*)$$

Sub-gradients, Sub-differential, and Some Rules

- Set of all sub-gradient at a non-diff point $\boldsymbol{x}_*$ is called the sub-differential

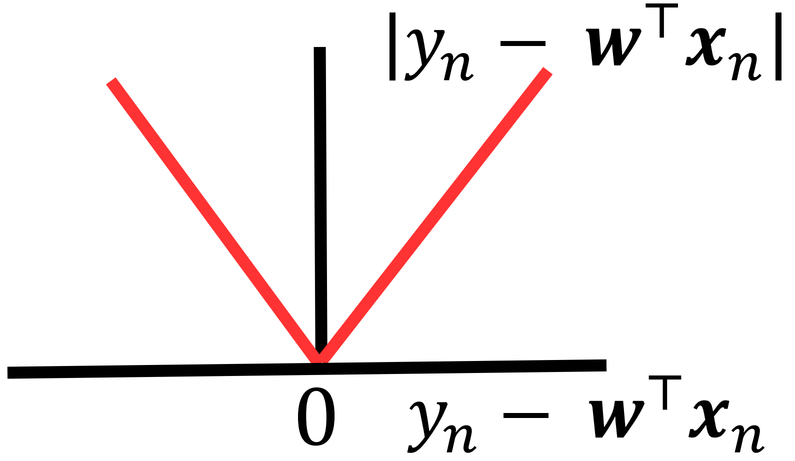
$$\boxed{\partial f(\boldsymbol{x}_*) \triangleq \{\boldsymbol{g} : f(\mathbf{x}) \geq f(\boldsymbol{x}_*) + \boldsymbol{g}^\top(\boldsymbol{x} - \boldsymbol{x}_*) \quad \forall \mathbf{x}\}}$$

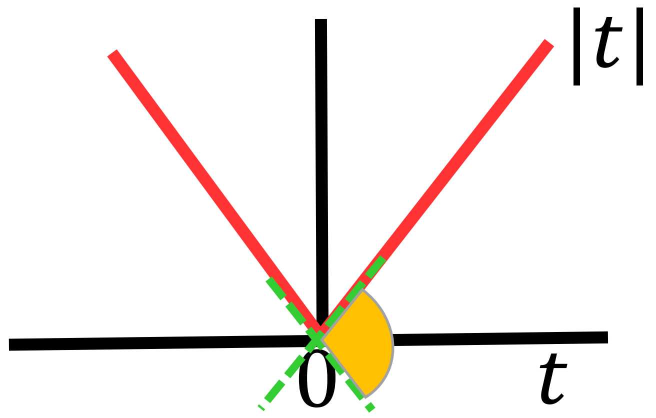
- Some basic rules of sub-diff calculus to keep in mind
  - Scaling rule: $\partial\big(c \cdot f(\mathbf{x})\big) = c \cdot \partial f(\mathbf{x}) = \{c \cdot \mathbf{v} : \mathbf{v} \in \partial f(\mathbf{x})\}$
  - Sum rule: $\partial\big(f(\mathbf{x}) + g(\mathbf{x})\big) = \partial f(\mathbf{x}) + \partial g(\mathbf{x}) = \{\mathbf{u} + \mathbf{v} : \mathbf{u} \in \partial f(\mathbf{x}), \mathbf{v} \in \partial g(\mathbf{x})\}$
  - Affine trans: $\partial f(\mathbf{a}^\top \mathbf{x} + b) = \mathbf{a}^\top \partial f(t) = \{\mathbf{a}^\top \boldsymbol{c} : \boldsymbol{c} \in \partial f(t)\}$, where $t = \mathbf{a}^\top \mathbf{x} + b$
  - Max rule: If $h(\boldsymbol{x}) = \max\{f(\boldsymbol{x}), g(\boldsymbol{x})\}$ then we calculate $\partial h(\boldsymbol{x})$ at $\boldsymbol{x}_*$ as
    - If $f(\boldsymbol{x}_*) > g(\boldsymbol{x}_*), \partial h(\boldsymbol{x}_*) = \partial f(\boldsymbol{x}_*)$, If $g(\boldsymbol{x}_*) > f(\boldsymbol{x}_*), \partial h(\boldsymbol{x}_*) = \partial g(\boldsymbol{x}_*)$
    - If $f(\boldsymbol{x}_*) = g(\boldsymbol{x}_*), \partial h(\boldsymbol{x}_*) = \{\alpha\mathbf{a} + (1-\alpha)\mathbf{b} : \mathbf{a} \in \partial f(\boldsymbol{x}_*), \mathbf{b} \in \partial g(\boldsymbol{x}_*), \alpha \in [0,1]\}$

The affine transform rule is a special case of the more general chain rule

- $\boldsymbol{x}_*$ is a stationary point for a non-diff function $f(\boldsymbol{x})$ if the zero vector belongs to the sub-differential at x_*, i.e., $\mathbf{0} \in \partial f(\boldsymbol{x}_*)$

Sub-Gradient For Absolute Loss Regression





$$\partial|t| = \begin{cases} 1 & \text{if } t > 0 \\ -1 & \text{if } t < 0 \\ [-1,+1] & \text{if } t = 0 \end{cases}$$

- The loss function for linear reg. with absolute loss: $L(\boldsymbol{w}) = |y_n - \boldsymbol{w}^\top \boldsymbol{x}_n|$
- Non-differentiable at $y_n - \boldsymbol{w}^\top \boldsymbol{x}_n = 0$
- Can use the affine transform rule of sub-diff calculus
- Assume $t = y_n - \boldsymbol{w}^\top \boldsymbol{x}_n$. Then $\partial L(\boldsymbol{w}) = -\boldsymbol{x}_n \partial|t|$
  - $\partial L(\boldsymbol{w}) = -\boldsymbol{x}_\mathrm{n} \times 1 = -x_n$ if $t > 0$
  - $\partial L(\boldsymbol{w}) = -\boldsymbol{x}_\mathrm{n} \times -1 = x_n$ if $t < 0$
  - $\partial L(\boldsymbol{w}) = -\boldsymbol{x}_\mathrm{n} \times c = -cx_n$ where $c \in [-1,+1]$ if $t = 0$

Sub-Gradient Descent

- Suppose we have a non-differentiable function $\boldsymbol{L(w)}$
- Sub-gradient descent is almost identical to GD except we use subgradients

Sub-Gradient Descent

- Initialize $\boldsymbol{w}$ as $\boldsymbol{w}^{(0)}$
- For iteration $\boldsymbol{t} = \boldsymbol{0,1,2,\dots}$ (or until convergence)
  - Calculate the sub-gradient $\boldsymbol{g}^{(t)} \in \partial L(\boldsymbol{w}^{(t)})$
  - Set the learning rate η_t
  - Move in the opposite direction of subgradient

$$\boldsymbol{w}^{(t+1)} = \boldsymbol{w}^{(t)} - \eta_t \boldsymbol{g}^{(t)}$$

Some Practical Aspects: Initialization

- Iterative opt. algos like GD, SGD, etc need to be initialized to "good" values
  - Bad initialization can result on bad local optima

But still be careful with learning rate

- Mainly a concern for non-convex loss functions, not so much for convex loss functions

If the goal is to learn the same model but for a different training set

- Transfer Learning: Initialize using params of a model trained on a related dataset
- Initialize using solution of a simpler but related model
  - E.g., for multitask regression (say $\boldsymbol{T}$ coupled regression problems), initialize using the solutions of the $\boldsymbol{T}$ independently trained regression problems
- For deep learning models, initialization is very important
  - Transfer learning approach is often used (initialize using "pre-trained" model from another dataset)
  - Bad initialization can make the model be stuck at saddle points. Need more care.
  - Random restarts: Running with several random initializations can often help

Some Practical Aspects: Assessing Convergence

- Various ways to assess convergence, e.g. consider converged if
  - The objective's value (on train set) ceases to change much across iterations

$$L(\boldsymbol{w}^{(t+1)}) - L(\boldsymbol{w}^{(t)}) < \epsilon$$

(for some small pre-defined ϵ)

  - The parameter values cease to change much across iterations

$$\| \boldsymbol{w}^{(t+1)} - \boldsymbol{w}^{(t)} \| < \tau$$

(for some small pre-defined τ)

  - Above condition is also equivalent to saying that the gradients are close to zero

$$\| \boldsymbol{g}^{(t)} \| \rightarrow 0$$

Caution: May not yet be at the optima. Use at your own risk!

  - The objective's value has become small enough that we are happy with ☺
  - Use a validation set to assess if the model's performance is acceptable (early stopping)

Some Practical Aspects: Learning Rate (Step Size)

- Some guidelines to select good learning rate (a.k.a. step size) η_t

C is a hyperparameter

- For convex functions, setting η_t something like C/t or $C/\sqrt{t}$ often works well
  - These step-sizes are actually theoretically optimal in some settings
  - In general, we want the learning rates to satisfy the following conditions
    - $\eta_t \to 0$ as t becomes very very large
    - $\sum \eta_t = \infty$ (needed to ensure that we can potentially reach anywhere in the parameter space)
  - Sometimes carefully chosen constant learning rates (usually small, or initially large and later small) also work well in practice
- Can also search for the "best" step-size by solving an opt. problem in each step

Also called "line search"

$$\eta_t = \arg\min_{\eta \geq 0} f\left(\mathbf{w}^{(t)} - \eta \cdot \mathbf{g}^{(t)}\right)$$

A one-dim optimization problem (note that $\mathbf{w}^{(t)}$ and $\mathbf{g}^{(t)}$ are fixed)

- A faster alternative to line search is the Armijo-Goldstein rule
  - Starting with current (or some large) learning rate (from prev. iter), and try a few values in decreasing order until the objective's value has a sufficient reduction

Some Practical Aspects: Adaptive Gradient Methods

- Can also use different learning rate in different dimensions

$$\boldsymbol{w}^{(t+1)} = \boldsymbol{w}^{(t)} - \boldsymbol{e}^{(t)} \odot \boldsymbol{g}^{(t)}$$

Vector of learning rates along each dimension

Element-wise product of two vectors

$$e_d^{(t)} = \frac{1}{\sqrt{\epsilon + \sum_{\tau=1}^{t}\left(g_d^{(t)}\right)^2}}$$

If some dimension had big updates recently (marked by large gradient values), slow down along those directions by using smaller learning rates - AdaGrad (Duchi et al, 2011)

- Can use a momentum term to stabilize gradients by reusing info from past grads
  - Move faster along directions that were <u>previously</u> good
  - Slow down along directions where gradient has <u>changed abruptly</u>

β usually set as 0.9

The "momentum" term. Set to 0 at initialization

$$\boldsymbol{m}^{(t)} = \beta \boldsymbol{m}^{(t-1)} + \eta_t \boldsymbol{g}^{(t)}$$
$$\boldsymbol{w}^{(t+1)} \leftarrow \boldsymbol{w}^{(t)} - \boldsymbol{m}^{(t)}$$

In an even faster version of this, $\boldsymbol{g}^{(t)}$ is replaced by the gradient computed at the next step if previous direction were used, i.e., $\nabla L(\boldsymbol{w}^{(t)} - \beta \boldsymbol{m}^{(t-1)})$. Called Nesterov's Accelerated Gradient (NAG) method

- Also exist several more advanced methods that combine the above methods
  - RMS-Prop: AdaGrad + Momentum, Adam: NAG + RMS-Prop
  - These methods are part of packages such as PyTorch, Tensorflow, etc

Optimization for ML: Some Final Comments

- Gradient methods are simple to understand and implement
- More sophisticated optimization methods also often use gradient methods
- Backpropagation algo used in deep neural nets is GD + chain rule of differentiation
- Use subgradient methods if function not differentiable
- Constrained optimization can use Lagrangian or projected GD
- Second order methods such as Newton's method faster but computationally expensive
- But computing all this gradient related stuff by hand looks scary to me. Any help?
  - Don't worry. **Automatic Differentiation (AD)** methods available now (will see them later)
  - AD only requires specifying the loss function (especially useful for deep neural nets)
  - Many packages such as Tensorflow, PyTorch, etc. provide AD support
  - But having a good understanding of optimization is still helpful