Optimization Techniques for ML (wrap-up)

CS771: Introduction to Machine Learning

Piyush Rai

Today

- Some practical aspects for optimization for ML
- Constrained optimization
- Optimization of non-differentiable functions

Some Practical Aspects: Iterate Averaging for SGD

- SGD iterates $\boldsymbol{w}^{(1)}, \boldsymbol{w}^{(2)}, \boldsymbol{w}^{(3)}, \ldots$ can be noisy (recall SGD computes gradients using randomly picked single training example, or a small minibatch)
- Polyak-Ruppert Averaging: Average SGD iterates and use the average in the end

SGD/mini-batch SGD update at iteration $t+1$

Stochastic gradient on a single/minibatch of examples

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

This way of computing the average is the same as doing $\bar{\boldsymbol{w}}^{(t+1)} = \sum_{i=1}^{t+1} \boldsymbol{w}^{(i)}$ but to avoid storing the previous weights, so we compute a running average

Sometimes, we don't start averaging from iteration 1 but after some warm-up iterations

Running average weight vector at iteration $t+1$

Averaged weight vector at previous iteration t

$$\bar{\boldsymbol{w}}^{(t+1)} = \frac{t}{t+1}\bar{\boldsymbol{w}}^{(t)} + \frac{1}{t+1}\boldsymbol{w}^{(t+1)}$$

- Averaging is quite popular for SGD. Stochastic Weighted Averaging (SWA) is another such recently proposed scheme (similar to Polyak-Ruppert Averaging) used for deep neural networks

Averaging Weights Leads to Wider Optima and Better Generalization (Izmailov et al, UAI 2018)

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

    $$L(\boldsymbol{w}^{(t+1)}) - L\left(\boldsymbol{w}^{(t)}\right) < \epsilon$$ (for some small pre-defined ϵ)

  - The parameter values cease to change much across iterations

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

$$\boldsymbol{w}^{(t+1)} = \boldsymbol{w}^{(t)} - \boldsymbol{e}^{(t)} \odot \boldsymbol{g}^{(t)} \qquad e_d^{(t)} = \frac{1}{\sqrt{\epsilon + \sum_{\tau=1}^{t}\left(g_d^{(t)}\right)^2}}$$

Vector of learning rates along each dimension

Element-wise product of two vectors

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>

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

β usually set as 0.9

The "momentum" term. Set to 0 at initialization

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

Constrained Optimization

Projected Gradient Descent

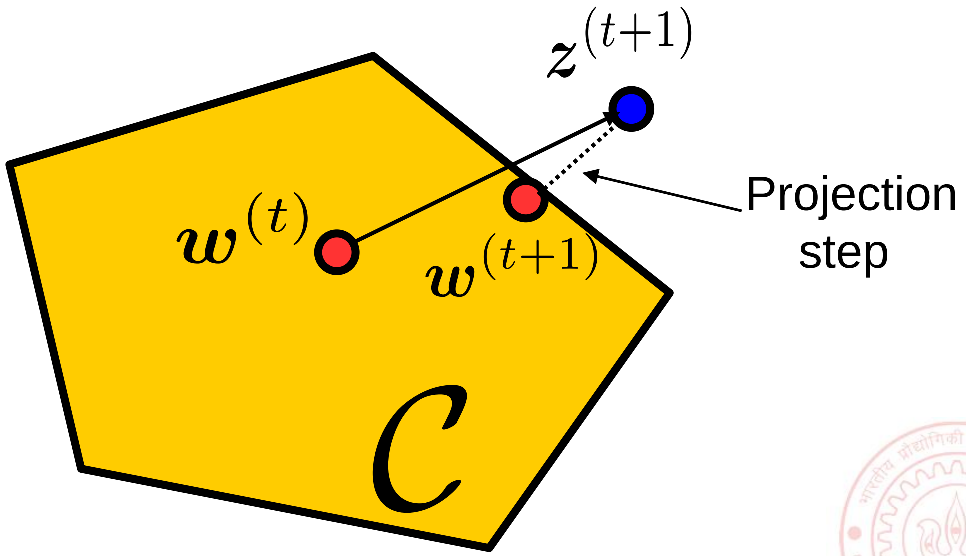
- Consider an optimization problem of the form

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

- 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

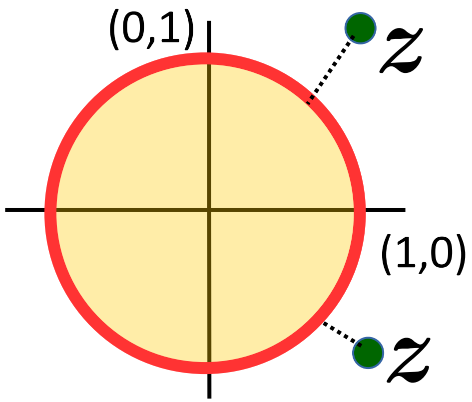
$$\Pi_{\mathcal{C}}[\boldsymbol{z}] = \arg\min_{\boldsymbol{w}\in\mathcal{C}} ||\boldsymbol{z} - \boldsymbol{w}||^2$$

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

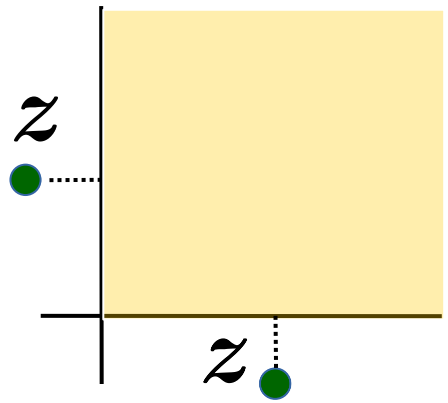
$\mathcal{C}$: Unit radius ℓ_2 ball



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 unconstrained 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

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

Constrained Opt. via Lagrangian

- Therefore, we can write our original problem as

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

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

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

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

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

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

- 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

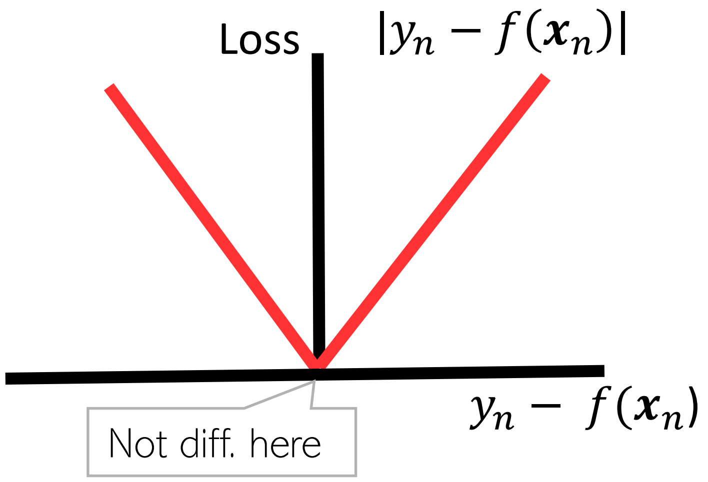
$$\hat{\boldsymbol{w}}_P = \arg\min_{\boldsymbol{w}} \left\{ \max_{\alpha \geq 0, \beta} \left\{ f(\boldsymbol{w}) + \sum\nolimits_{i=1}^{K} \alpha_i g_i(\boldsymbol{w}) + \sum\nolimits_{j=1}^{L} \beta_j h_j(\boldsymbol{w}) \right\} \right\}$$

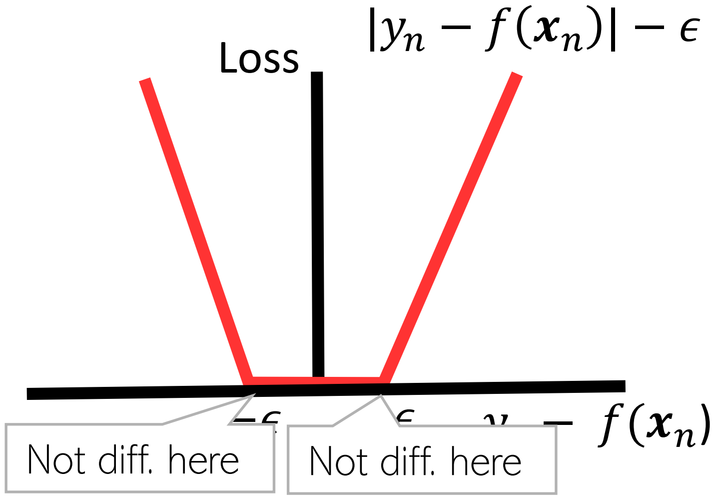
$$\hat{\boldsymbol{w}}_D = \arg\max_{\alpha \geq 0, \beta} \left\{ \min_{\boldsymbol{w}} \left\{ f(\boldsymbol{w}) + \sum\nolimits_{i=1}^{K} \alpha_i g_i(\boldsymbol{w}) + \sum\nolimits_{j=1}^{L} \beta_j h_j(\boldsymbol{w}) \right\} \right\}$$

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 $\boldsymbol{\epsilon}$-insensitive loss; even $\boldsymbol{\ell_1}$ norm regularizer is non-diff

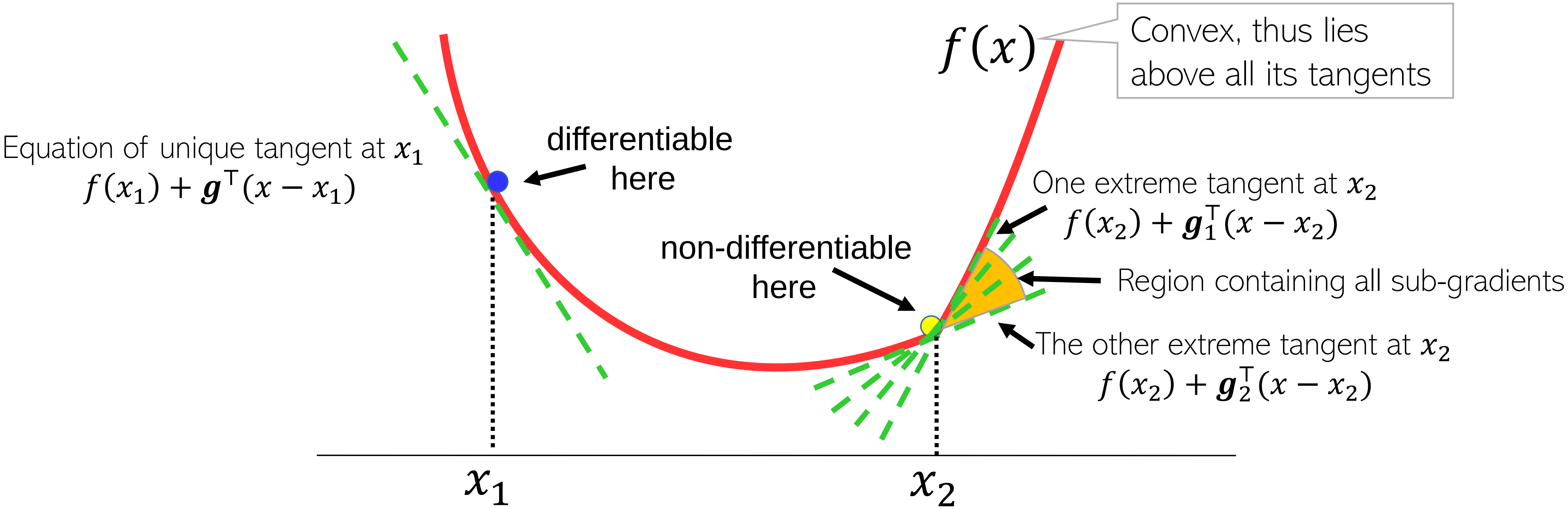




- Basically, any function in which there are points with kink is non-diff
  - At such points, the function is non-differentiable and thus gradients not defined
  - Reason: Can't define a unique tangent at such points

Sub-gradients

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



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

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

Sub-gradients, Sub-differential, and Some Rules

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

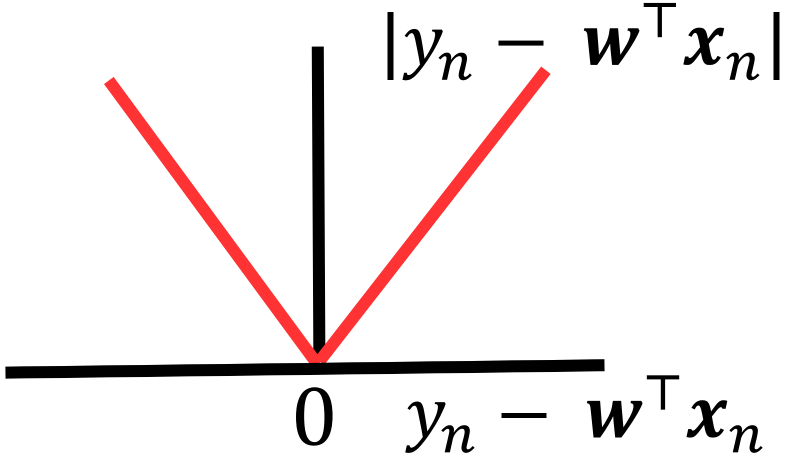
$$\boxed{\partial f(\boldsymbol{x}_*) \triangleq \{\boldsymbol{g} : f(\mathbf{x}) \geq f(\boldsymbol{x}_*) + \boldsymbol{g}^\top(\boldsymbol{x} - \boldsymbol{x}_*) \quad \forall \mathbf{x}\}}$$

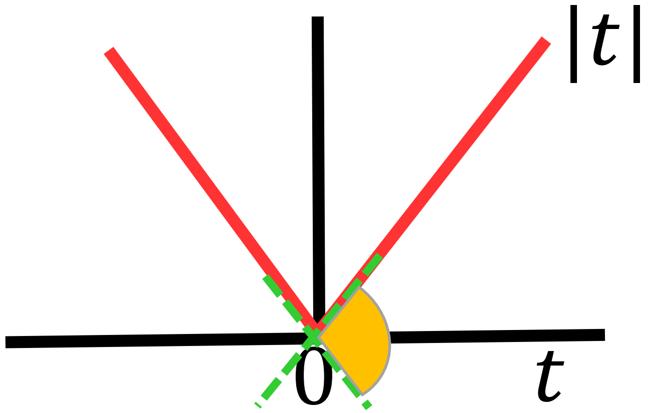
- Some basic rules of sub-diff calculus to keep in mind
  - Scaling rule: $\partial\big(c \cdot f(\mathbf{x})\big) = c \cdot \partial f(\mathbf{x}) = \{c \cdot \mathbf{v} : \mathbf{v} \in \partial f(\mathbf{x})\}$
  - Sum rule: $\partial\big(f(\mathbf{x}) + g(\mathbf{x})\big) = \partial f(\mathbf{x}) + \partial g(\mathbf{x}) = \{\mathbf{u} + \mathbf{v} : \mathbf{u} \in \partial f(\mathbf{x}), \mathbf{v} \in \partial g(\mathbf{x})\}$
  - Affine trans: $\partial f(\mathbf{a}^\top \mathbf{x} + b) = \mathbf{a} \cdot \partial f(t) = \{\mathbf{a} \cdot c : c \in \partial f(t)\}$, 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 $\boldsymbol{x}_*$, i.e., $\mathbf{0} \in \partial f(\boldsymbol{x}_*)$

Sub-Gradient For Absolute Loss Regression





Using max rule of sub-differentials and using $|t| = \max\{t, -t\}$

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

- 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 and max rule of sub-diff calculus
- Assume $t = y_n - \boldsymbol{w}^\top \boldsymbol{x}_n$. Then $\partial L(\boldsymbol{w}) = -\boldsymbol{x}_n \partial|t|$
  - $\partial L(\boldsymbol{w}) = -\boldsymbol{x}_n \times 1 = -\boldsymbol{x}_n$ if $t > 0$
  - $\partial L(\boldsymbol{w}) = -\boldsymbol{x}_n \times -1 = \boldsymbol{x}_n$ if $t < 0$
  - $\partial L(\boldsymbol{w}) = -\boldsymbol{x}_n \times c = -c\boldsymbol{x}_n$ 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 $\boldsymbol{t} = \mathbf{0,1,2,\ldots}$ (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

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

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