

*Annotated  
version*

**Machine Learning Course - CS-433**

# **Optimization**

Sep 25+27, 2018

©Martin Jaggi and Mohammad Emtiyaz Khan 2016

minor changes by Martin Jaggi 2017

minor changes by Martin Jaggi 2018

Last updated on: September 25, 2018



# Learning / Estimation / Fitting

## Loss

Given a cost function  $\mathcal{L}(\mathbf{w})$ , we wish to find  $\mathbf{w}^*$  which minimizes the cost:

$$\min_{\mathbf{w}} \mathcal{L}(\mathbf{w}) \quad \text{subject to } \mathbf{w} \in \mathbb{R}^D$$

$$\mathcal{L}(\mathbf{w}) = \frac{1}{N} \sum_{n=1}^N (y_n - f_{\mathbf{w}}(x_n))^2$$

prediction  
Model

This means the *learning* problem is formulated as an optimization problem.

We will use an optimization algorithm to solve the problem (to find a good  $\mathbf{w}$ ).

## Grid Search

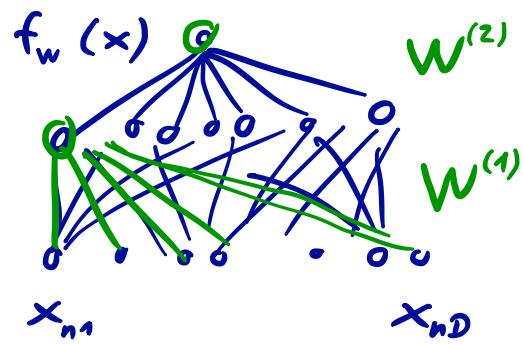
Grid search is one of the simplest optimization algorithms. We compute the cost over all values  $\mathbf{w}$  in a grid, and pick the best among those.

This is brute-force, but extremely simple and works for any kind of cost function when we have very few parameters and the cost is easy to compute.

Linear Model:

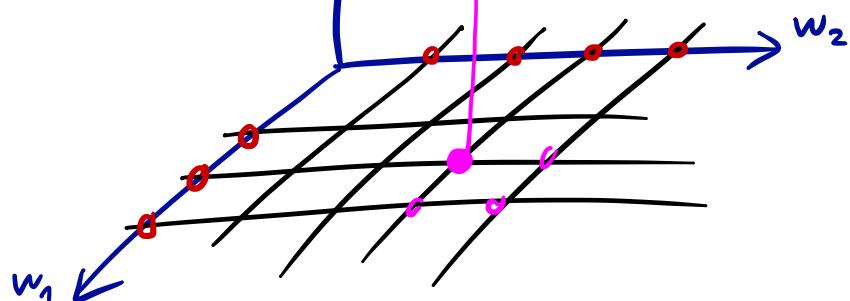
$$f_{\mathbf{w}}(\mathbf{x}) = \mathbf{x}^T \mathbf{w}$$

Neural Network:



$$\mathcal{L}(\mathbf{w})$$

$$\mathcal{L}(\mathbf{w})$$



$w \in \mathbb{R}^D$

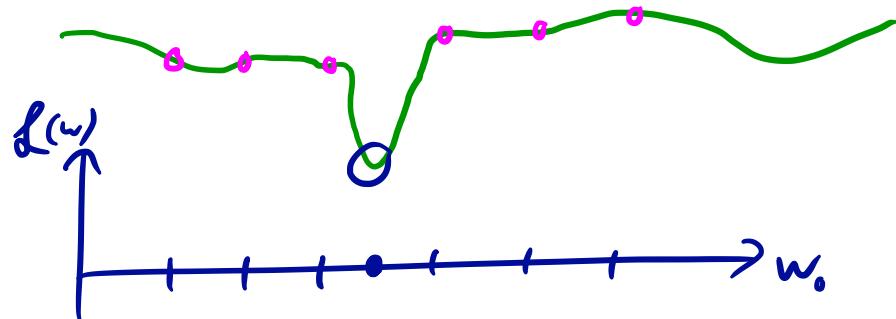
previous Example:  $D=2$

For a large number of parameters  $D$ , however, grid search has too many “for-loops”, resulting in an exponential computational complexity:

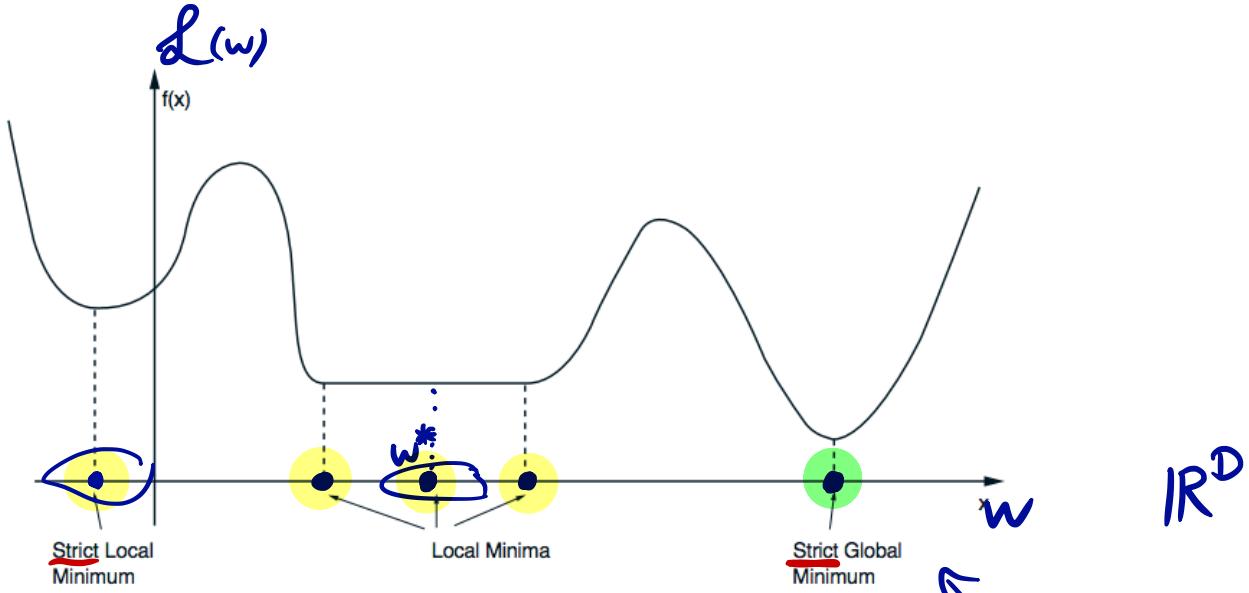
If we decide to use 10 possible values for each dimension of  $\mathbf{w}$ , then we have to check  $10^D$  points. This is clearly impossible for most practical machine learning models, which can often have  $D \approx$  millions of parameters. Choosing a good range of values for each dimension is another problem.

$$\underbrace{10^D}_{D \approx 1 \text{ million}}$$

*Other issues:* No guarantee can be given that we end up close to an optimum.



# Optimization Landscapes



The above figure is taken from Bertsekas, Nonlinear programming.

A vector  $\mathbf{w}^*$  is a local minimum of  $\mathcal{L}$  if it is no worse than its neighbors; i.e. there exists an  $\epsilon > 0$  such that,

$$\underline{\mathcal{L}(\mathbf{w}^*) \leq \mathcal{L}(\mathbf{w})}, \quad \forall \mathbf{w} \text{ with } \|\mathbf{w} - \mathbf{w}^*\| < \epsilon$$

A vector  $\mathbf{w}^*$  is a global minimum of  $\mathcal{L}$  if it is no worse than all others,

$$\mathcal{L}(\mathbf{w}^*) \leq \mathcal{L}(\mathbf{w}), \quad \forall \mathbf{w} \in \mathbb{R}^D$$

A local or global minimum is said to be strict if the corresponding inequality is strict for  $\mathbf{w} \neq \mathbf{w}^*$ .

# Smooth Optimization

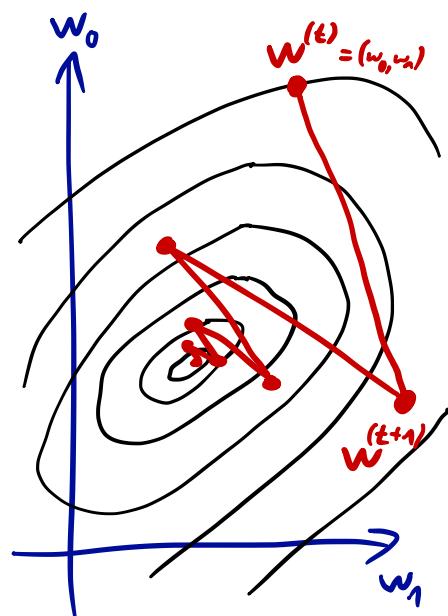
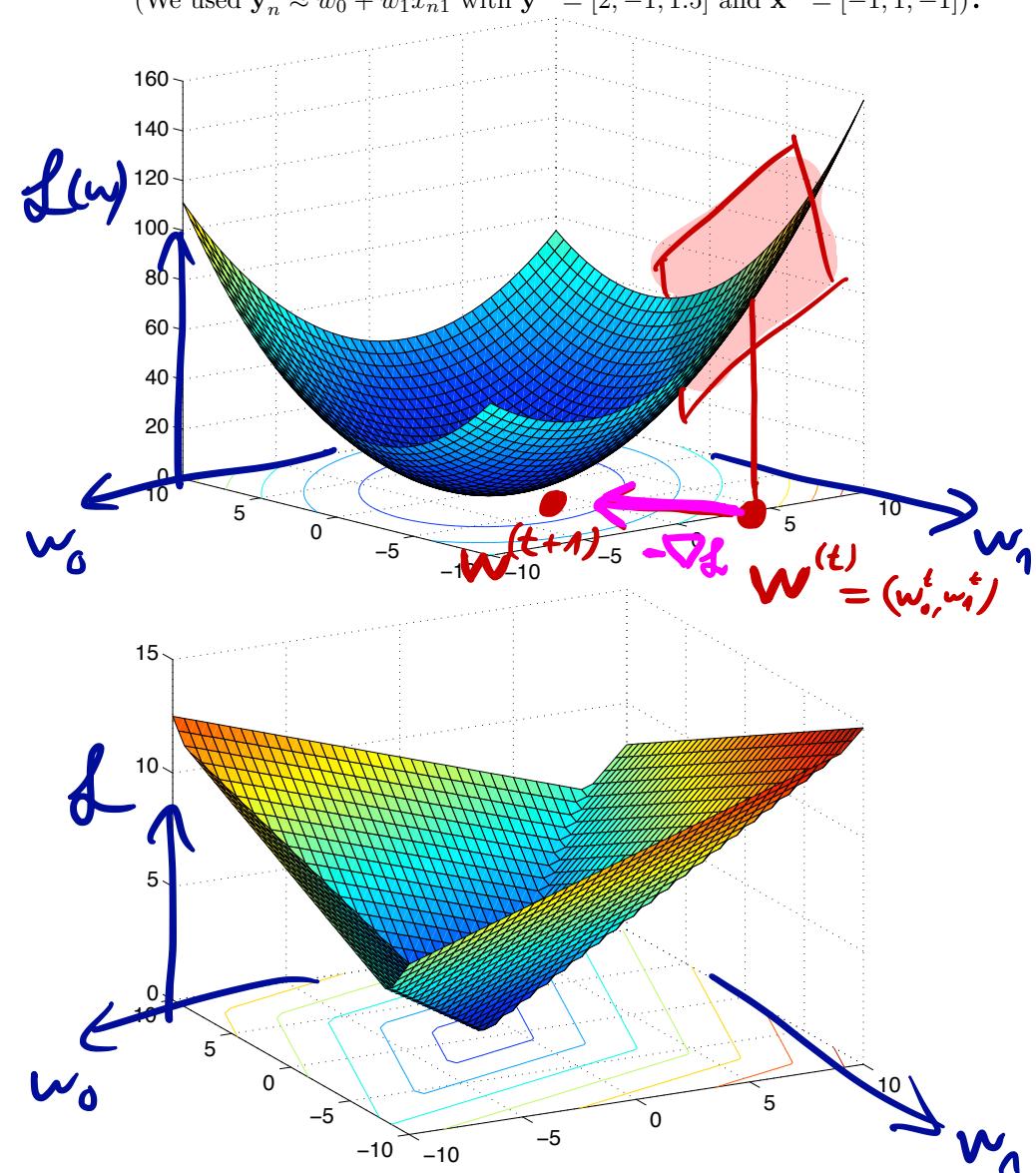
## Follow the Gradient

A gradient (at a point) is the slope of the tangent to the function (at that point). It points to the direction of largest increase of the function.

For a 2-parameter model,  $\text{MSE}(\mathbf{w})$  and  $\text{MAE}(\mathbf{w})$  are shown below.

(We used  $\mathbf{y}_n \approx w_0 + w_1 x_{n1}$  with  $\mathbf{y}^\top = [2, -1, 1.5]$  and  $\mathbf{x}^\top = [-1, 1, -1]$ ).

In parameter space



Definition of the gradient:

How fast does  $\mathcal{L}$  change depending on  $w_1$

$$\nabla \mathcal{L}(\mathbf{w}) := \left[ \frac{\partial \mathcal{L}(\mathbf{w})}{\partial w_1}, \dots, \frac{\partial \mathcal{L}(\mathbf{w})}{\partial w_D} \right]^\top$$

This is a vector,  $\nabla \mathcal{L}(\mathbf{w}) \in \mathbb{R}^D$ .

## Gradient Descent

To minimize the function, we iteratively take a step in the (opposite) direction of the gradient

$$\mathbf{w}^{(t+1)} := \mathbf{w}^{(t)} - \gamma \nabla \mathcal{L}(\mathbf{w}^{(t)})$$

where  $\gamma > 0$  is the step-size (or learning rate). Then repeat with the next  $t$ .

step:

$$t = 0, 1, 2, \dots$$

popular stepsize:

$$\delta = \delta(t) := \frac{1}{t}$$

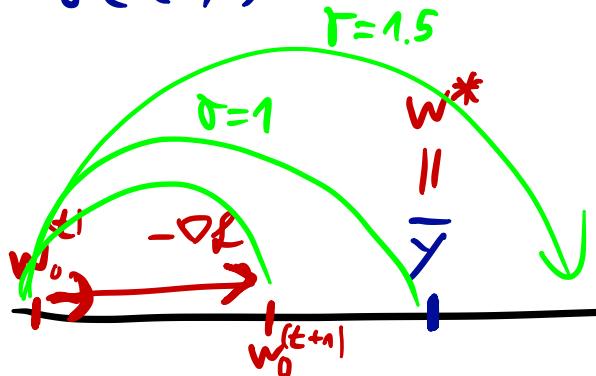
**Example:** Gradient descent for 1-parameter model to minimize MSE:

$$w_0^{(t+1)} := \underline{(1 - \gamma)w_0^{(t)} + \gamma \bar{y}}$$

where  $\bar{y} := \sum_n y_n / N$ . When is this sequence guaranteed to converge?

$\delta \in (0, 1]$  : convergence to optimum

$\delta \in (0, 2)$  : " " "



$\delta > 2$ : divergence!  
 $\delta < 0$ : "

$$\begin{aligned} \mathcal{L}(w) &= \\ &\frac{1}{2N} \sum_{n=1}^N (y_n - w_0)^2 \\ &\parallel f_w(x_n) \\ &\text{Model} \end{aligned}$$

$$\begin{aligned} \nabla \mathcal{L}(w) &= \frac{\partial}{\partial w_0} \mathcal{L} \\ &= \frac{1}{N} \sum_{n=1}^N -(y_n - w_0) \\ &= \underline{w_0 - \bar{y}} \end{aligned}$$

$$w_0^{(t+1)}$$

## Gradient Descent for Linear MSE

$(w_1, \dots, w_D)$

$w \in \mathbb{R}^D$

For linear regression

$$\mathbf{y} = \begin{bmatrix} y_1 \\ y_2 \\ \vdots \\ y_N \end{bmatrix}, \mathbf{X} = \begin{bmatrix} x_{11} & x_{12} & \dots & x_{1D} \\ x_{21} & x_{22} & \dots & x_{2D} \\ \vdots & \vdots & \ddots & \vdots \\ x_{N1} & x_{N2} & \dots & x_{ND} \end{bmatrix} \quad \mathbf{x}_n \in \mathbb{R}^D$$

$$f_w(\mathbf{x}) = \mathbf{x}_n^\top w$$

We define the error vector  $\mathbf{e}$ :

$$\mathbf{e} = \mathbf{y} - \mathbf{X}w \quad \mathbf{e}_n = y_n - \mathbf{x}_n^\top w \in \mathbb{R}^N$$

and MSE as follows:

$$\begin{aligned} \mathcal{L}(w) &:= \frac{1}{2N} \sum_{n=1}^N (y_n - \mathbf{x}_n^\top w)^2 \\ &= \frac{1}{2N} \mathbf{e}^\top \mathbf{e} \end{aligned}$$

*model*  $\downarrow$   $\mathcal{L}_n(w)$

...  $\cdot w_n$

$$\begin{aligned} \frac{\partial \mathcal{L}}{\partial w_1} &= \dots - \frac{1}{N} \underbrace{(y_n - \mathbf{x}_n^\top w)}_{\mathbf{e}_n} \underbrace{\mathbf{x}_{n1}}_{\cancel{x_n}} \dots \\ &= -\frac{1}{N} (\mathbf{x}_{:,1})^\top \mathbf{e} \\ \frac{\partial \mathcal{L}}{\partial w_D} &= -\frac{1}{N} (\mathbf{x}_{:,D})^\top \mathbf{e} \end{aligned}$$

Homework!

**Computational cost.** What is the complexity (# operations) of computing the gradient?

- a) starting from  $\mathbf{w}$  and
- b) given  $\mathbf{e}$  and  $\mathbf{w}$ ?



① compute  $\mathbf{e}$ :

$$\text{cost} \approx 2N \cdot D + N$$

② compute  $\nabla \mathcal{L}$ :

$$\text{cost} \approx 2N \cdot D + D$$

In total:  $\mathcal{O}(N \cdot D)$

**Variant with offset.** Recall: Alternative trick when also incorporating an offset term for the regression:

$$\mathbf{y} = \begin{bmatrix} y_1 \\ y_2 \\ \vdots \\ y_N \end{bmatrix} \quad \tilde{\mathbf{X}} = \begin{bmatrix} 1 & x_{11} & x_{12} & \dots & x_{1D} \\ 1 & x_{21} & x_{22} & \dots & x_{2D} \\ \vdots & \vdots & \vdots & \ddots & \vdots \\ 1 & x_{N1} & x_{N2} & \dots & x_{ND} \end{bmatrix}$$

Exercise:

compute  $\nabla \mathcal{L}$

## Stochastic Gradient Descent

**Sum Objectives.** In machine learning, most cost functions are formulated as a sum over the training examples, that is

$$\mathcal{L}(\mathbf{w}) = \frac{1}{N} \sum_{n=1}^N \mathcal{L}_n(\mathbf{w}) ,$$

where  $\mathcal{L}_n$  is the cost contributed by the  $n$ -th training example.

**Q:** What are the  $\mathcal{L}_n$  for linear MSE?

**The SGD Algorithm.** The stochastic gradient descent (SGD) algorithm is given by the following update rule, at step  $t$ :

① Sample one training point  $n \in \{1, \dots, N\}$  uniformly at random

$$\mathbf{w}^{(t+1)} := \mathbf{w}^{(t)} - \gamma \nabla \mathcal{L}_n(\mathbf{w}^{(t)}) .$$

repeat  
②  
g

## Theoretical Motivation. Idea:

Cheap but unbiased estimate of the gradient!

In expectation over the random choice of  $n$ , we have

$$\mathbb{E} [\underline{\nabla \mathcal{L}_n(\mathbf{w})}] = \nabla \mathcal{L}(\mathbf{w})$$

**g**

which is the true gradient direction.  
(check!)

$$\begin{aligned} \mathbb{E}_n [\nabla \mathcal{L}_n(\mathbf{w})] &= \frac{1}{N} \sum_{n=1}^N \nabla \mathcal{L}_n(\mathbf{w}) = \nabla \mathcal{L}(\mathbf{w}) \\ &= \nabla(\mathbb{E} \dots) \end{aligned}$$

↑

**Mini-batch SGD.** There is an intermediate version, using the update direction being

$$\mathbf{g} := \frac{1}{|B|} \sum_{n \in B} \nabla \mathcal{L}_n(\mathbf{w}^{(t)})$$

**update direction**

again with

$$\mathbf{w}^{(t+1)} := \mathbf{w}^{(t)} - \gamma \mathbf{g}.$$

**mini-batch**

In the above gradient computation, we have randomly chosen a subset  $B \subseteq [N]$  of the training examples. For each of these selected examples  $n$ , we compute the respective gradient  $\nabla \mathcal{L}_n$ , at the same current point  $\mathbf{w}^{(t)}$ .

$B \subseteq \{1, \dots, N\}$

### Example

- $B = \{n\}$  randomly  $\rightarrow$  SGD
- $B = \{1, \dots, N\}$   $\rightarrow$  (full) gradient descent
- $|B| = 5$  mini-batch

The computation of  $\mathbf{g}$  can be parallelized easily. This is how current deep-learning applications utilize GPUs (by running over  $|B|$  threads in parallel).

Note that in the extreme case  $B := [N]$ , we obtain (batch) gradient descent, i.e.  $\mathbf{g} = \nabla \mathcal{L}$ .

Everyone among  $|B|$  workers computes one of the  $\nabla \mathcal{L}_n$  for  $n \in B$

## SGD for Linear MSE

See Exercise Sheet 2.

**Computational cost.** For linear MSE, what is the complexity (# operations) of computing the stochastic gradient?

(using only  $|B| = 1$  data examples)

computational cost

a) starting from  $\mathbf{w}$ ?

$$\text{(GD)} \quad \mathcal{L}(\mathbf{w}) = \frac{1}{N} \sum_{n=1}^N \frac{1}{2} (y_n - \mathbf{x}_n^T \mathbf{w})^2$$

$$\nabla \mathcal{L}(\mathbf{w}) = -\frac{1}{N} X^T (\underbrace{y - Xw}_e) \quad \mathcal{O}(N \cdot D)$$

Mini-Batch SGD

pick  $n$  at random.

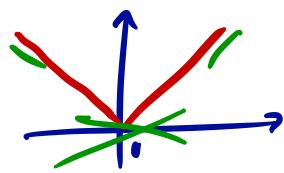
$$\text{(SGD)} \quad \mathcal{L}_n(\mathbf{w}) = \frac{1}{2} (y_n - \mathbf{x}_n^T \mathbf{w})^2$$

$$\nabla \mathcal{L}_n(\mathbf{w}) = \underbrace{(-\mathbf{x}_n)^T (y_n - \mathbf{x}_n^T \mathbf{w})}_{\mathbb{R}^D \quad \mathbb{R}} \quad \mathcal{O}(D)$$

*N times cheaper!*

# // non-differentiable

## Non-Smooth Optimization

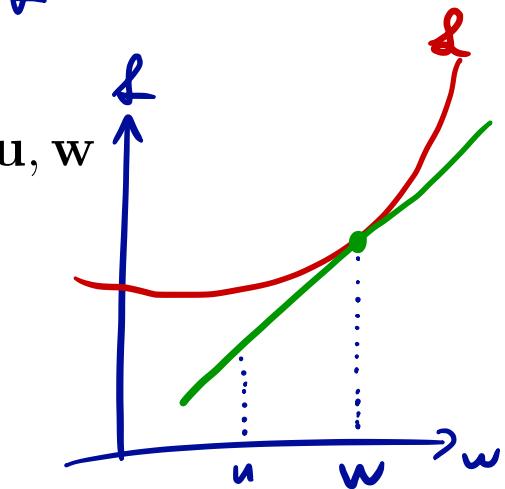


An alternative characterization of convexity, for differentiable functions is given by

$$\mathcal{L}(\mathbf{u}) \geq \mathcal{L}(\mathbf{w}) + \nabla \mathcal{L}(\mathbf{w})^\top (\mathbf{u} - \mathbf{w}) \quad \forall \mathbf{u}, \mathbf{w}$$

meaning that the function must always lie above its linearization.

$\Leftrightarrow$  f convex

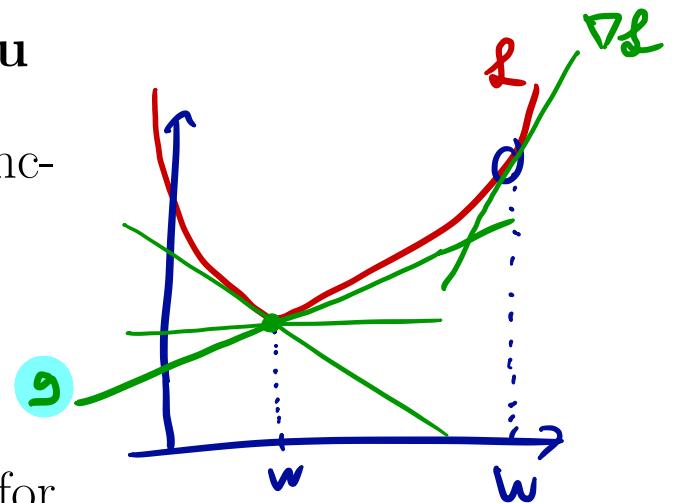


## Subgradients

A vector  $\mathbf{g} \in \mathbb{R}^D$  such that

$$\mathcal{L}(\mathbf{u}) \geq \mathcal{L}(\mathbf{w}) + \mathbf{g}^\top (\mathbf{u} - \mathbf{w}) \quad \forall \mathbf{u}$$

is called a subgradient to the function  $\mathcal{L}$  at  $\mathbf{w}$ .



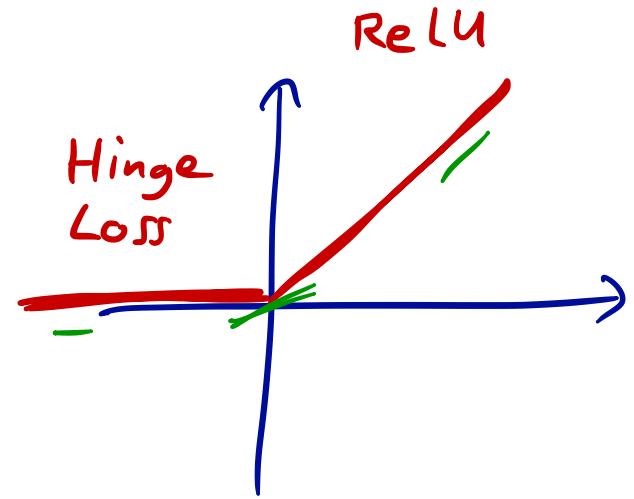
This definition makes sense for objectives  $\mathcal{L}$  which are not necessarily differentiable (and not even necessarily convex).

If  $\mathcal{L}$  is differentiable at  $\mathbf{w}$ , then the only subgradient at  $\mathbf{w}$  is  $\mathbf{g} = \nabla \mathcal{L}(\mathbf{w})$ .

# Subgradient Descent

Identical to the gradient descent algorithm, but using a subgradient instead of gradient. Update rule

$$\mathbf{w}^{(t+1)} := \mathbf{w}^{(t)} - \gamma \mathbf{g}$$



for  $\mathbf{g}$  being a subgradient to  $\mathcal{L}$  at the current iterate  $\mathbf{w}^{(t)}$ .

## Example: Optimizing Linear MAE

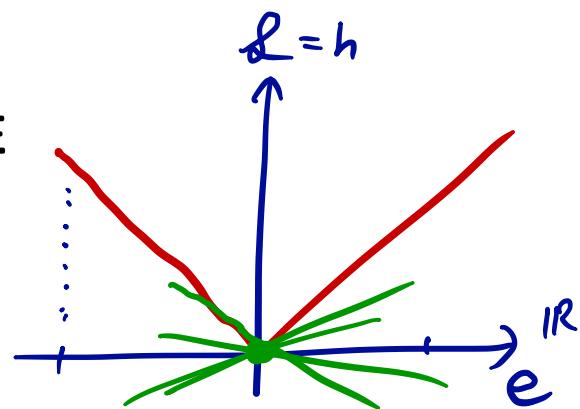
1. Compute a subgradient of the absolute value function

$$h : \mathbb{R} \rightarrow \mathbb{R}, h(e) := |e|.$$

at  $e$ :

2. Recall the definition of the mean absolute error:

$$\mathcal{L}(\mathbf{w}) = \text{MAE}(\mathbf{w}) := \frac{1}{N} \sum_{n=1}^N \|y_n - f(\mathbf{x}_n)\|$$



$$g \in \partial h(e) = \begin{cases} -1 & \text{if } e < 0 \\ [-1, 1] & \text{if } e = 0 \\ 1 & \text{if } e > 0 \end{cases}$$

set of  
all subgradients  
at  $e$

For linear regression, its (sub)gradient is easy to compute using the chain rule.  
Compute it!

$h$  non-diff.  
 $q$  differentiable

$$\mathcal{L}(w) := h(q(w))$$

See Exercise Sheet 2.

At  $w$ :

$$g \in \partial h(q(w)) \cdot \nabla q(w)$$

is a subgradient of  $\mathcal{L}$  at  $w$ .

## Stochastic Subgradient Descent

Stochastic SubGradient Descent  
(still abbreviated SGD commonly).

Same,  $\mathbf{g}$  being a subgradient to the randomly selected  $\mathcal{L}_n$  at the current iterate  $\mathbf{w}^{(t)}$ .

$$\mathcal{L}(\mathbf{w}) = \frac{1}{N} \sum_{n=1}^N \mathcal{L}_n(\mathbf{w})$$

possibly non-differentiable

Linear regression:

complexity per step?

**Exercise:** Compute the SGD update for linear MAE.

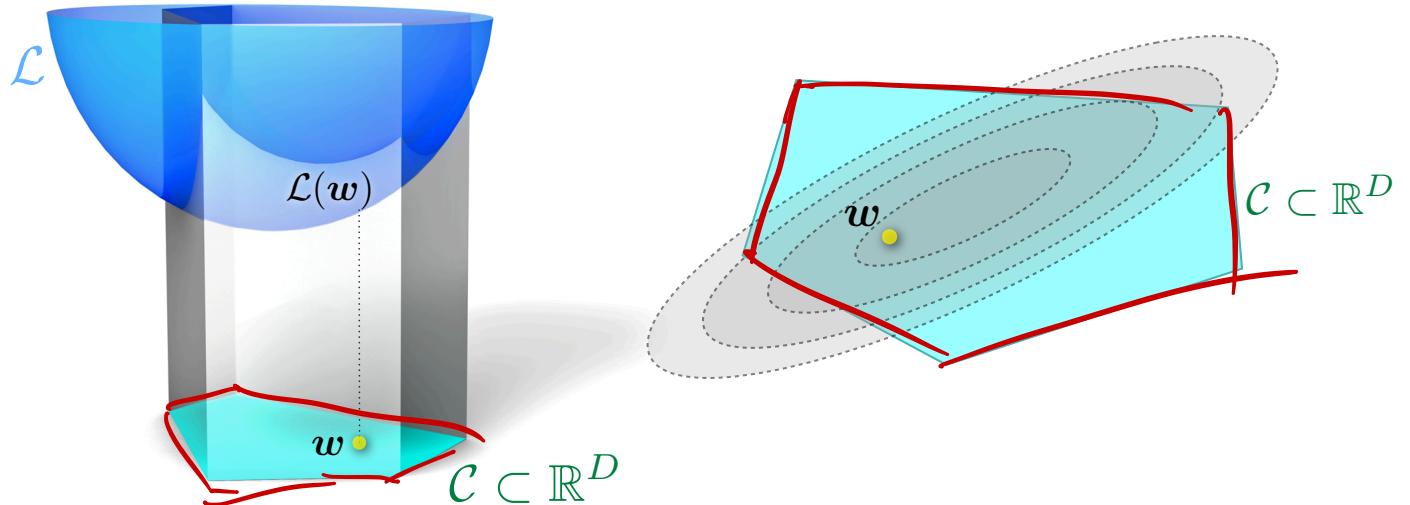
	Smooth <small>MSE</small>	non-smooth <small>MAE</small>
GD <small>(full)</small> $\mathcal{L}$	gradient of $\mathcal{L}$ $\mathcal{O}(N \cdot D)$	subgradient of $\mathcal{L}$ $\mathcal{O}(N \cdot D)$
SGD $\mathcal{L}_n$	gradient of $\mathcal{L}_n$ $\mathcal{O}(D)$	subgradient of $\mathcal{L}_n$ $\mathcal{O}(D)$

# Constrained Optimization

Sometimes, optimization problems come posed with additional constraints:

$$\min_{\mathbf{w}} \mathcal{L}(\mathbf{w}), \quad \text{subject to } \mathbf{w} \in \mathcal{C}.$$

The set  $\mathcal{C} \subset \mathbb{R}^D$  is called the constraint set.



## Solving Constrained Optimization Problems

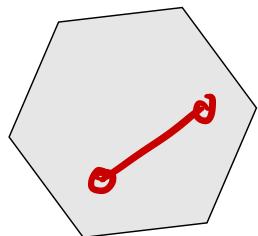
- A) Projected Gradient Descent
- B) Transform it into an *unconstrained* problem

# Convex Sets

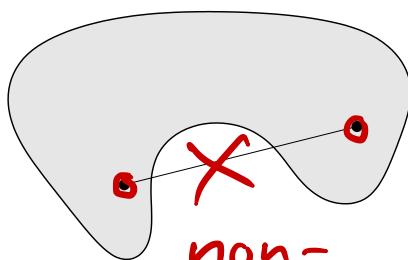
A set  $\mathcal{C}$  is **convex** iff

the line segment between any two points of  $\mathcal{C}$  lies in  $\mathcal{C}$ , i.e., if for any  $\mathbf{u}, \mathbf{v} \in \mathcal{C}$  and any  $\theta$  with  $0 \leq \theta \leq 1$ , we have

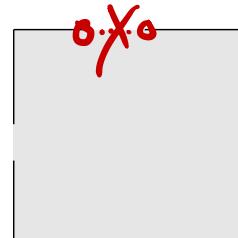
$$\theta\mathbf{u} + (1 - \theta)\mathbf{v} \in \underline{\mathcal{C}}.$$



**convex**



**non-convex**



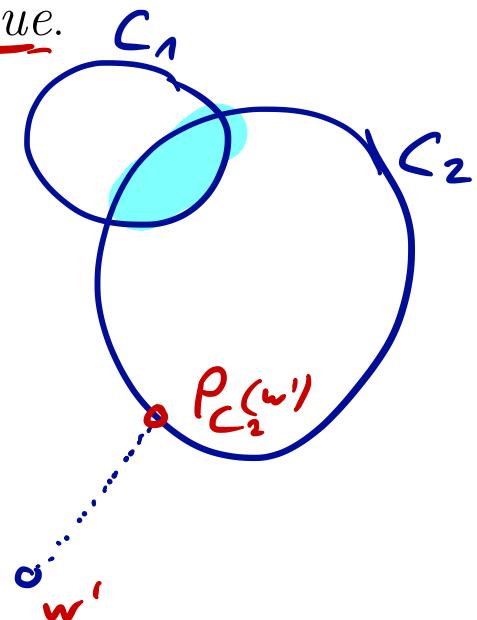
**non-convex**

\*Figure 2.2 from S. Boyd, L. Vandenberghe

## Properties of Convex Sets

- Intersections of convex sets are convex
- Projections onto convex sets are unique.  
(and often efficient to compute)  
Formal definition:  $C$  closed

$$P_{\mathcal{C}}(\mathbf{w}') := \arg \min_{\mathbf{v} \in \mathcal{C}} \|\mathbf{v} - \mathbf{w}'\|.$$



# Projected Gradient Descent

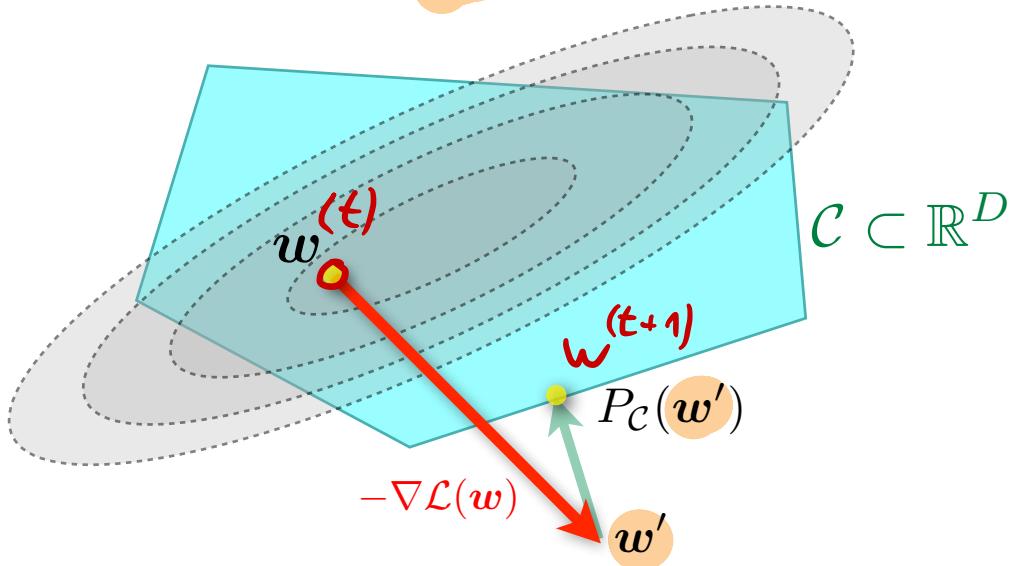
Idea: add a projection onto  $\mathcal{C}$  after every step:

$$P_{\mathcal{C}}(\mathbf{w}') := \arg \min_{\mathbf{v} \in \mathcal{C}} \|\mathbf{v} - \mathbf{w}'\| .$$

Update rule:

$$\mathbf{w}^{(t+1)} := P_{\mathcal{C}} \left[ \mathbf{w}^{(t)} - \gamma \nabla \mathcal{L}(\mathbf{w}^{(t)}) \right] .$$

$\mathbf{w}'$



**Projected SGD.** Same SGD step, followed by the projection step, as above. Same convergence properties.

Computational cost of projection?  
Crucial!

# Turning Constrained into Unconstrained Problems

(Alternatives to projected gradient methods)

Use **penalty functions** instead of directly solving  $\min_{\mathbf{w} \in \mathcal{C}} \mathcal{L}(\mathbf{w})$ .

- “brick wall” (indicator function)

$$I_{\mathcal{C}}(\mathbf{w}) := \begin{cases} 0 & \mathbf{w} \in \mathcal{C} \\ \infty & \mathbf{w} \notin \mathcal{C} \end{cases}$$
$$\Rightarrow \min_{\mathbf{w} \in \mathbb{R}^D} \mathcal{L}(\mathbf{w}) + I_{\mathcal{C}}(\mathbf{w})$$

(disadvantage: non-continuous objective)

- Penalize error. *Example:*

$$\mathcal{C} = \{\mathbf{w} \in \mathbb{R}^D \mid A\mathbf{w} = \mathbf{b}\}$$

$$\Rightarrow \min_{\mathbf{w} \in \mathbb{R}^D} \mathcal{L}(\mathbf{w}) + \lambda \|A\mathbf{w} - \mathbf{b}\|^2$$

- Linearized Penalty Functions  
(see Lagrange Multipliers)

# Implementation Issues

For Gradient Methods:

**Stopping criteria:** When  $\nabla \mathcal{L}(\mathbf{w})$  is (close to) zero, we are (often) close to the optimum.

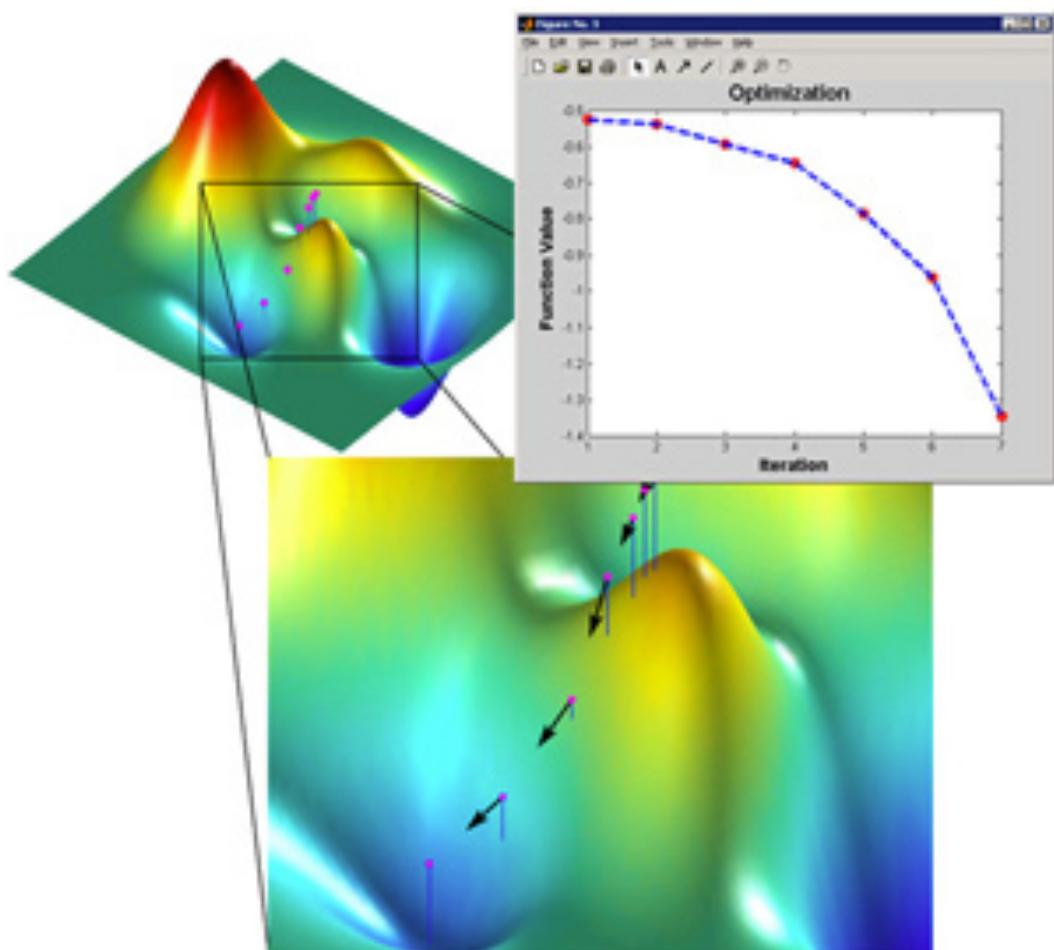
**Optimality:** If the second-order derivative is positive (positive semi-definite to be precise), then it is a (possibly local) minimum. If the function is also convex, then this condition implies that we are at a global optimum. See the supplementary section on [Optimality Conditions](#).

**Step-size selection:** If  $\gamma$  is too big, the method might diverge. If it is too small, convergence is slow. Convergence to a local minimum is guaranteed only when  $\gamma < \gamma_{min}$  where  $\gamma_{min}$  is a fixed constant that depends on the problem.

**Line-search methods:** For some objectives  $\mathcal{L}$ , we can set step-size automatically using a line-search method. More details on “backtracking” methods can be found in Chapter 1 of Bertsekas’ book on “nonlinear programming”.

**Feature normalization and pre-conditioning:** Gradient descent is very sensitive to ill-conditioning. Therefore, it is typically advised to normalize your input features. In other words, we pre-condition the optimization problem. Without this, step-size selection is more difficult since different “directions” might converge at different speed.

# Non-Convex Optimization



\*image from mathworks.com

Real-world problems are **not convex**!

All we have learnt on algorithm design and performance of convex algorithms still helps us in the non-convex world.

# Additional Notes

## Grid Search and Hyper-Parameter Optimization

Read more about grid search and other methods for “hyperparameter” setting:

[en.wikipedia.org/wiki/Hyperparameter\\_optimization#Grid\\_search](https://en.wikipedia.org/wiki/Hyperparameter_optimization#Grid_search).

## Computational Complexity

The computation cost is expressed using the [big-O](#) notation. Here is a definition taken from Wikipedia. Let  $f$  and  $g$  be two functions defined on some subset of the real numbers. We write  $f(x) = \mathcal{O}(g(x))$  as  $x \rightarrow \infty$ , if and only if there exists a positive real number  $c$  and a real number  $x_0$  such that  $|f(x)| \leq c|g(x)|$ ,  $\forall x > x_0$ .

Please read and learn more from this page in Wikipedia:

[en.wikipedia.org/wiki/Computational\\_complexity\\_of\\_mathematical\\_operations#Matrix\\_algebra](https://en.wikipedia.org/wiki/Computational_complexity_of_mathematical_operations#Matrix_algebra) .

- What is the computational complexity of matrix multiplication?
- What is the computational complexity of matrix-vector multiplication?

## Optimality Conditions

For a *convex* optimization problem, the first-order *necessary* condition says that at *an* optimum the gradient is equal to zero.

$$\nabla \mathcal{L}(\mathbf{w}^*) = \mathbf{0} \quad (1)$$

The second-order *sufficient* condition ensures that the optimum is a minimum (not a maximum or saddle-point) using the [Hessian](#) matrix,

which is the matrix of second derivatives:

$$\mathbf{H}(\mathbf{w}^*) := \frac{\partial^2 \mathcal{L}(\mathbf{w}^*)}{\partial \mathbf{w} \partial \mathbf{w}^\top} \quad \text{is positive semi-definite.} \quad (2)$$

The Hessian is also related to the convexity of a function: a twice-differentiable function is convex if and only if the Hessian is positive semi-definite at all points.

## SGD Theory

As we have seen above, when  $N$  is large, choosing a random training example  $(\mathbf{x}_n, y_n)$  and taking an SGD step is advantageous:

$$\mathbf{w}^{(t+1)} := \mathbf{w}^{(t)} + \gamma^{(t)} \nabla \mathcal{L}_n(\mathbf{w}^{(t)})$$

For convergence,  $\gamma^{(t)} \rightarrow 0$  “appropriately”. One such condition called the Robbins-Monroe condition suggests to take  $\gamma^{(t)}$  such that:

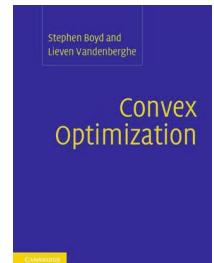
$$\sum_{t=1}^{\infty} \gamma^{(t)} = \infty, \quad \sum_{t=1}^{\infty} (\gamma^{(t)})^2 < \infty \quad (3)$$

One way to obtain such sequences is  $\gamma^{(t)} := 1/(t+1)^r$  where  $r \in (0.5, 1)$ .

# More Optimization Theory

If you want, you can gain a deeper understanding of several optimization methods relevant for machine learning from this survey:

Convex Optimization: Algorithms and Complexity  
- by Sébastien Bubeck



And also from the book of Boyd & Vandenberghe  
(both are free online PDFs)

(> 35 000  
citations)

## Exercises

1. Chain-rule



If it has been a while, familiarize yourself with it again.

2. Revise computational complexity (also see the Wikipedia link in Page 6 of lecture notes).
3. Derive the computational complexity of grid-search, gradient descent and stochastic gradient descent for linear MSE (# steps and cost per step).
4. Derive the gradients for the linear MSE and MAE cost functions.
5. Implement gradient descent and gain experience in setting the step-size.
6. Implement SGD and gain experience in setting the step-size.