EURECOM
Sophia Antipolis

MALCOM

Machine Learning for Communication Systems

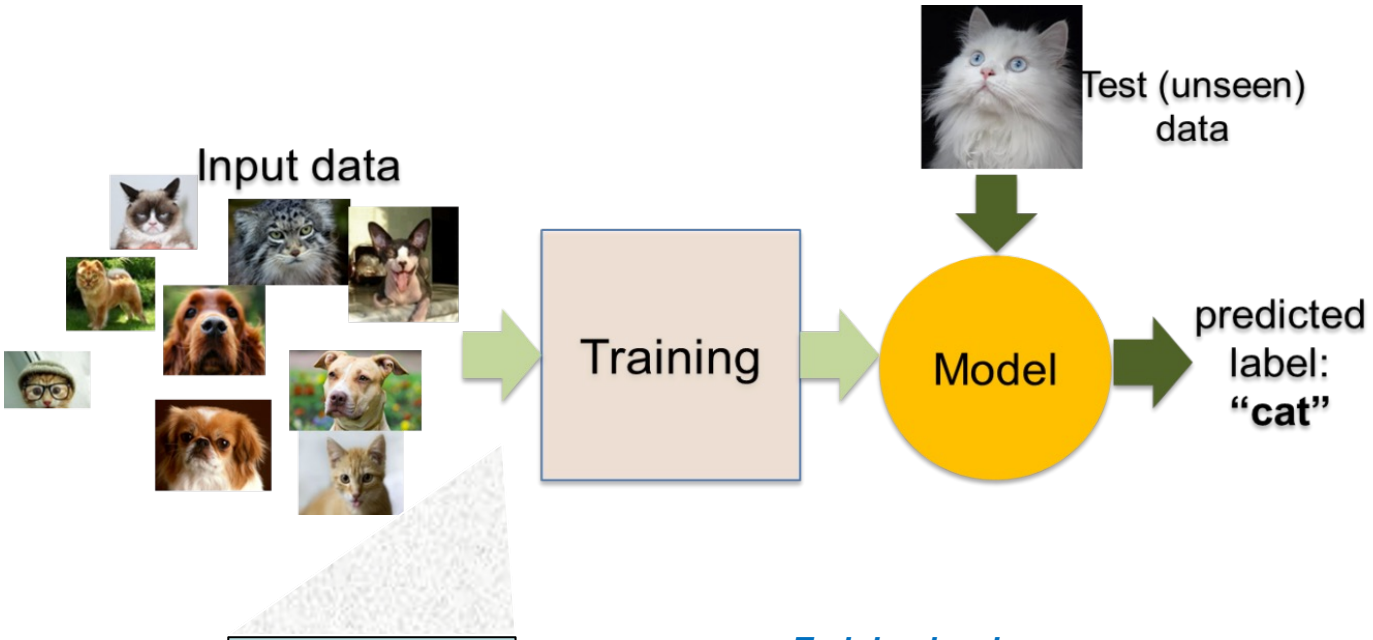
Lecture 7
Large-Scale Machine Learning & Optimization

Photios Stavrou

(Slides are courtesy of Marios Kountouris)

Spring 2024

“Standard” ML Pipeline



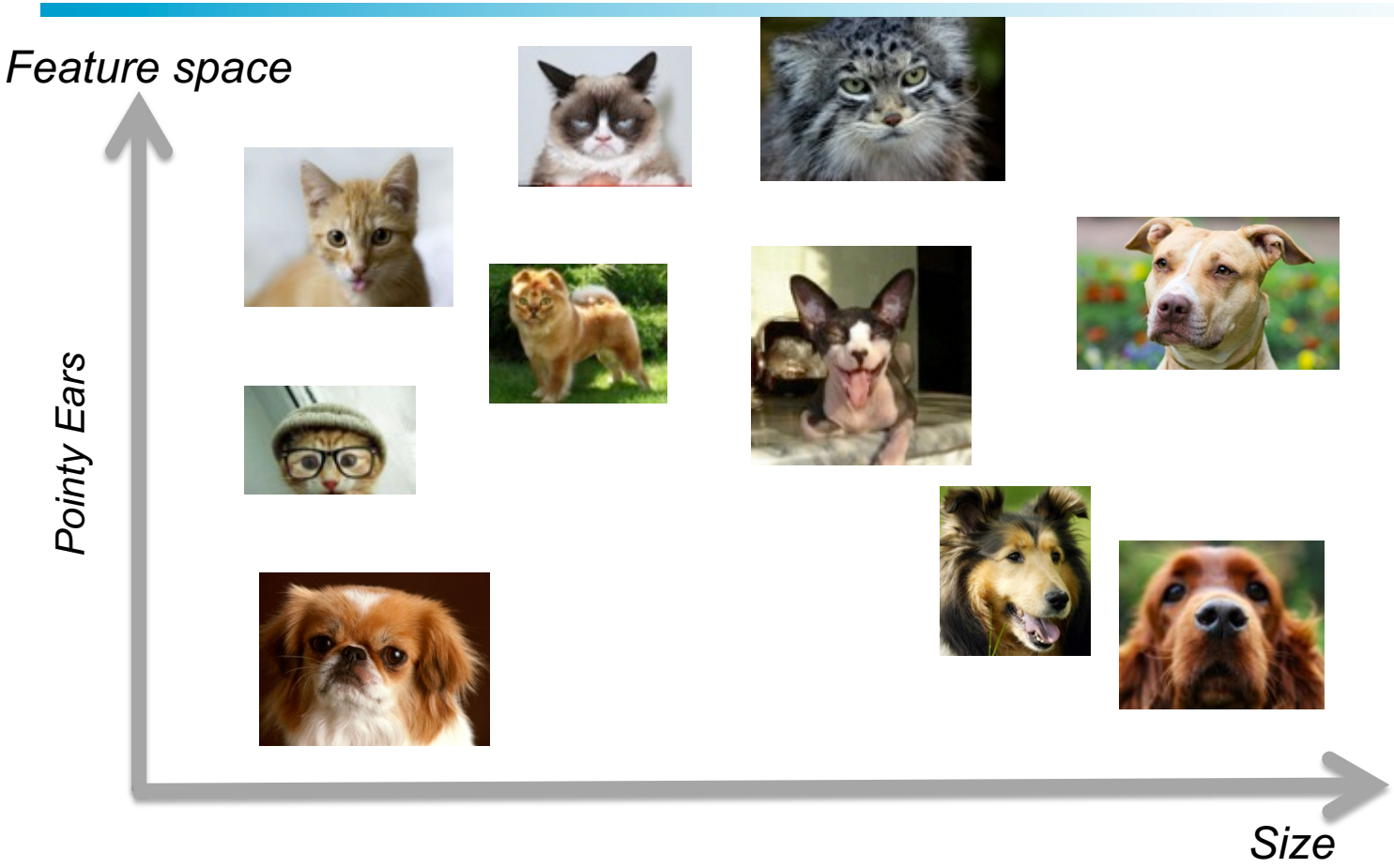
Data Scalability

- *Datasets: TBs or PBs of raw data*
- *Worldwide LHC computing grid can generate 15 PB of data/month*
- *Size of sequencing data generated on human genomes may reach 250 PBs in a few years*
- **Distributed computing to the rescue**

Training burden

- *Training on ImageNet takes*
  - *10+ days (Oct. 2017)*
  - *0.5 hour (TPU Pods) – 2018*
  - *nowdays 2.5 mins on Alibaba Cloud*
- *Very large neural nets can take many hours or weeks to train (2-3 years ago)*
- *Check: DAWNBench* https://dawn.cs.stanford.edu/benchmark/

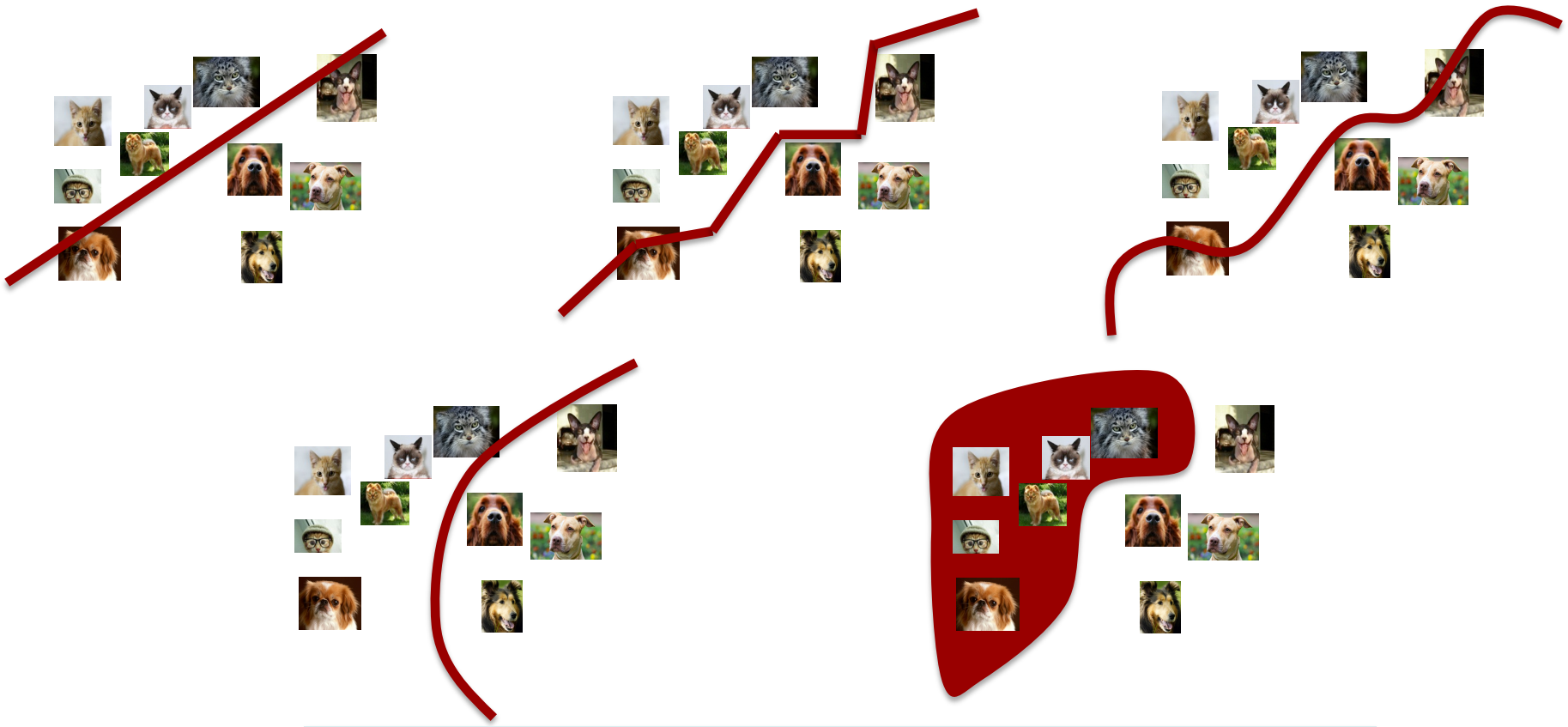
Feature Selection



- Simplification of models
- Improves Accuracy
- Reduces Training Time

Goal: use “informative” features

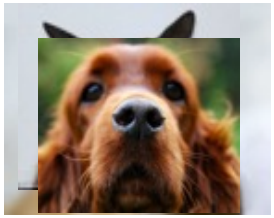
Predictor Selection (hypothesis class)

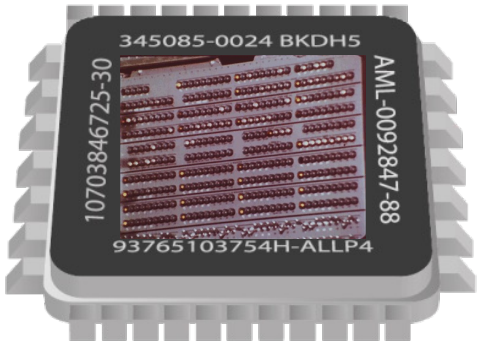


Goal: pick a predictor that

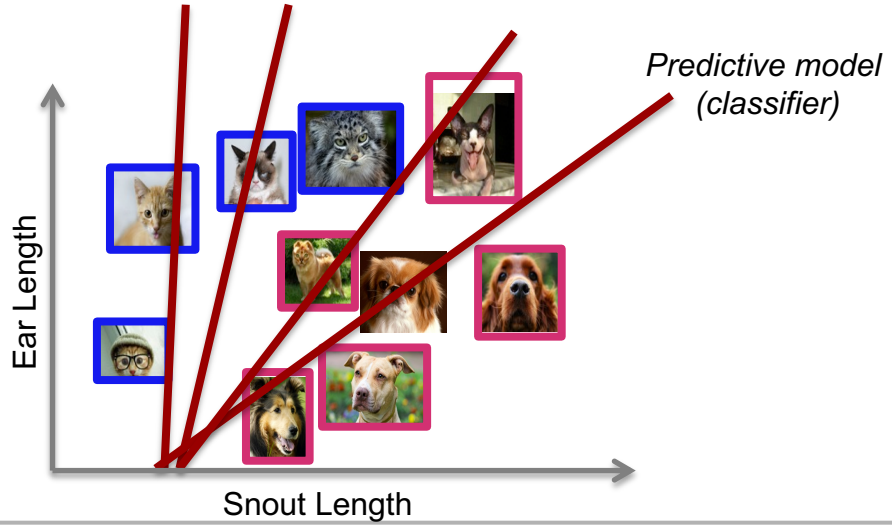
(i) is expressive (ii) is easy to train (iii) does not overfit

Statistical Learning





Goal: Train a model to minimize **training** (and testing) **error**



EURECOM
Sophia Antipolis

Statistical Learning Setting

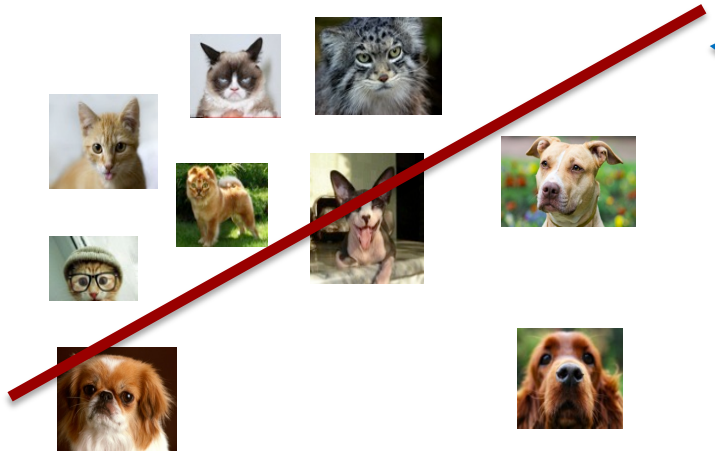
- $\mathcal{X}$ – domain set (input space)
- $\mathcal{Y}$ – label set (output space)
- n – number of training examples
- $\mathcal{D}$ – distribution over the data, i.e., $(x_i, y_i) \sim \mathcal{D}$

$\mathcal{H}$: vector space of prediction functions h can be:
- separating hyperplanes
- NNs of depth-D (depth of network)
- …

- Data: $S = \{(x_1, y_1), (x_2, y_2), \ldots, (x_n, y_n)\}$ – training set where $(x_i, y_i) \in \mathcal{X} \times \mathcal{Y}$ (i.i.d. from distribution $\mathcal{D}$)
- Loss function $\ell(y, h(x))$
- $\mathcal{H}$ - Hypothesis class (class of possible models we can learn)
- $\mathcal{H} \subset \{h: \mathcal{X} \to \mathcal{Y}\}$: $\mathcal{H}$ is a subset of all possible functions that map from input space to output space. Choosing this subset (hypothesis class), $\mathcal{H}$, introduce inductive bias
- *Example:* binary classification on a d dimensional real-valued dataset, we have $h(x_i) = \hat{y}_i$, where $h \in \mathcal{H}, x_i \in \mathbb{R}^d \equiv \mathcal{X}, \hat{y}_i \in \{0,1\} \equiv \mathcal{Y}$
- **Goal:** $\min_{h \in \mathcal{H}} \mathbb{E}_{(x,y)\sim\mathcal{D}}[\ell(y; h(x))]$

Goal of Training

- Train a **model**: prediction function h that performs well on unseen data.



Our trained predictor

- How to measure performance? Loss function: $\ell(y; h(x))$

Measures **disagreement** between predicted and true label

Examples of loss functions:
$\mathbf{1}_{h(x,w)\neq y}$
$(h(x) - y)^2$
$|h(x) - y|$
Cross entropy
Logistic loss

Goal: small loss on unseen data (e.g., on the test set)

$$\sum_{(x,y)} \ell(y; h(x))$$

unseen example

But, we haven't seen… unseen examples

Goal of Training

- Identify the hypothesis $h \in \mathcal{H}$ that gives the best performance on $\mathcal{D}$.
- The loss on the "unseen" examples converges to the **expected loss**

$$\sum_{(x,y)\sim\mathcal{D}} \ell(y; h(x)) \longrightarrow R(h) \equiv \mathbb{E}_{(x,y)\sim\mathcal{D}}[\ell(y; h(x))]$$

Expected (True) Risk

Testing cost

- What else converges to the **expected loss**? (why?)

$$\hat{R}_S(h) \equiv \frac{1}{n}\sum_{i=1}^{n} \ell(y_i; h(x_i))$$

law of large numbers

Empirical Risk

Training cost

Generalization Gap

- Given an sample set $S = (x_i, y_i), i \in \{1, \dots, n\}$, drawn i.i.d. from $\mathcal{D}$, a hypothesis h_S learnt on S, and a specific definition of loss ℓ, the generalization gap is defined as

$$\epsilon_{gen}(h_S) = \left| R(h_S) - \hat{R}_S(h_S) \right|$$

- We want to upper bound the generalization gap, i.e., $R(h_S) \leq \hat{R}_S(h_S) + \epsilon$

Key Result

- For all $h \in \mathcal{H}$, if $n = \Omega\left(\frac{\log\left(\frac{|\mathcal{H}|}{\delta}\right)}{\epsilon^2}\right)$, then with probability at least $1 - \delta$ we have

$$\left| R(h) - \hat{R}_S(h) \right| \leq \epsilon$$

Generalization Gap (cont'd)

Occam's (Razor) Bound

- Put a distribution over the countably infinite hypothesis class $\mathcal{H}$ that is independent of dataset S we will receive.
- Assumption: $\mathcal{H}$ is discrete (countable) and bounded loss function $\ell(\cdot)$
- Given a prior distribution P on $\mathcal{H}$, $\sum_{\forall h} P(h) = 1$, with probability at least $\geq 1-\delta$ over $S \sim \mathcal{D}^n$ we have that the following holds true for all $h \in \mathcal{H}$

$$R(h) \leq \hat{R}_S(h) + \sqrt{\frac{\log \frac{1}{P(h)} + \log \frac{2}{\delta}}{2n}}$$

Note: *a set $\mathcal{A}$ is countably infinite if $\mathbb{N} \approx \mathcal{A}$, i.e., $\mathcal{A}$ has the same cardinality as the natural numbers. In other words, a set for which there exists some bijective map $f: \mathbb{N} \to \mathcal{A}$ or the other way around.*

PAC Bayes Bound

- Given a prior probability distribution P over a hypothesis class $\mathcal{H}$ and a posterior probability distribution Q over $\mathcal{H}$. Then

$$\mathbb{E}_{h\sim Q}[R(h)] \leq \mathbb{E}_{h\sim Q}[\hat{R}_S(h)] + \sqrt{\frac{D_{KL}(Q||P) + \log \frac{n}{\delta}}{2(n-1)}}$$

with probability $\geq 1-\delta$.

Large-Scale Learning

Two main approaches

- Minimize Expected Risk with SGD

$$\min_{h \in \mathbb{R}^d} \mathbb{E}_{(x,y)\sim\mathcal{D}}[\ell(y; h(x))]$$

- Empirical Risk Minimization (ERM) with batch/incremental methods:

$$\min_{h \in \mathbb{R}^d} \frac{1}{n} \sum_{i=1}^{n} \ell(y_i; h(x_i))$$

- Often, we minimize the (regularized) empirical risk:

$$\min_{h \in \mathbb{R}^d} \frac{1}{n} \sum_{i=1}^{n} \ell(y_i; h(x_i)) \quad + \quad \Omega(h)$$

Measures model fit for data point i
(avoids under-fitting)

Measures model "complexity"
(avoids over-fitting)

- Convex loss ℓ, convex regularizer Ω

Large-Scale Learning

Two main approaches

- Minimize Expected Risk with SGD

$$\min_{h \in \mathbb{R}^d} \left\{ R(h) \coloneqq \mathbb{E}_{(x,y)\sim\mathcal{D}}[\ell(y; h(x))] \right\}$$

- Empirical Risk Minimization (ERM) with batch/incremental methods:

$$\min_{h \in \mathbb{R}^d} \left\{ \hat{R}(h) \coloneqq \frac{1}{n} \sum_{i=1}^{n} \ell(y_i; h(x_i)) \right\} \quad \text{s.t. } \Omega(h) \leq D$$

Lagrange duality

- Which one is better?

Large-Scale Learning Tradeoffs

- We seek the prediction function from a parametrized family of functions $\mathcal{H}$
- We optimize the empirical risk instead of the expected risk (the empirical optimum is denoted $h_S = \arg\min_{h \in \mathcal{H}} \hat{R}(h)$)
- Since this optimization can be costly, we stop the algorithm when it reaches a solution $\tilde{h}_S$ that minimizes the objective function with a predefined accuracy ρ

The excess error can be decomposed as $\mathcal{E} = \mathbb{E}\left[R(\tilde{h}_S) - R(h^*)\right] = \mathcal{E}_{\text{app}} + \mathcal{E}_{\text{est}} + \mathcal{E}_{\text{opt}}$

where $h^* = \arg\min_{h \in \mathbb{R}^d} R(h)$

Approximation error $\mathcal{E}_{\text{app}}$

- Measures how closely functions in $\mathcal{H}$ can approximate the optimal solution h^*.
- Can be reduced by choosing a larger family of functions

Estimation error $\mathcal{E}_{\text{est}}$

- Measures the effect of minimizing the empirical risk $\hat{R}(h)$ instead of the expected risk $R(h)$
- Can be reduced by choosing a smaller family of functions or by increasing the size of the training set.

Optimization error $\mathcal{E}_{\text{opt}}$

- Measures the impact of the approximate optimization on the expected risk.
- Can be reduced by running the optimizer longer. The additional computing time depends of course on the family of function and on the size of the training set.

Empirical Risk Minimization

Given constraints on

- the maximal computation time $T_{\max}$
- the maximal training set size $n_{\max}$,

Tradeoff involving the size of the family of functions $\mathcal{H}$, the optimization accuracy ρ and the number of examples n effectively processed by the optimization algorithm.

$$\min_{\mathcal{H},\rho,n} \mathcal{E} = \mathcal{E}_{\text{app}} + \mathcal{E}_{\text{est}} + \mathcal{E}_{\text{opt}} \quad \text{subject to } n \leq n_{\max} \text{ and } T(\mathcal{H},\rho,n) \leq T_{\max}$$

- Small-scale learning problems: constrained by the maximal number of examples $n_{\max}$. Since computing time is not an issue, we can reduce $\mathcal{E}_{\text{opt}}$ to insignificant levels by choosing ρ arbitrarily small, and we can minimize the estimation error $\mathcal{E}_{\text{est}}$ by choosing $n = n_{\max}$ (approximation-estimation tradeoff)

- Large-scale learning problems: constrained by $T_{\max}$, usually because the supply of training examples is very large. Approximate optimization can achieve better expected risk because more training examples can be processed during the allowed time. The specifics depend on the computational properties of the chosen optimization algorithm.

(Unconstrained) Optimization

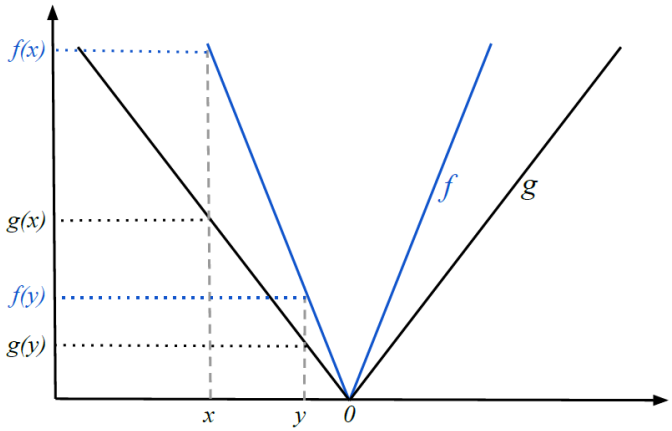
Gradient Descent Methods

Lipschitz continuity

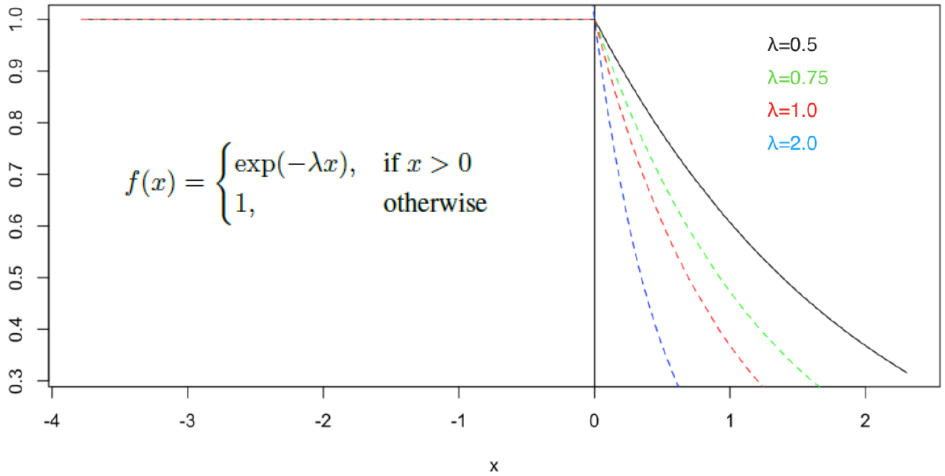
Let $L \geq 0$. A real-valued function f is L-Lipschitz continuous iff $\forall x, y \in \mathrm{dom}(f)$

$$|f(x) - f(y)| \leq L\|x - y\|$$

- Intuitively, a Lipschitz continuous function is bounded in how fast it can change.



If f and g (two Lipschitz continuous functions) change as fast as they can, then $L_f > L_g$



The value of *L* increases with λ. As λ increases, function gets closer to discontinuity (λ→ ∞, step function – not Lipschitz continuous for any *L*)

- Does not need to be differentiable, but a differentiable function f is *L*-Lipschitz continuous *iff*

$$\forall x \in \mathrm{dom}(f), \|\nabla f(x)\| \leq L$$ (**first order condition** for Lipschitz continuity)

- Special case of continuity: all Lipschitz continuous functions are continuous, but not all continuous functions are Lipschitz continuous

Convex Sets

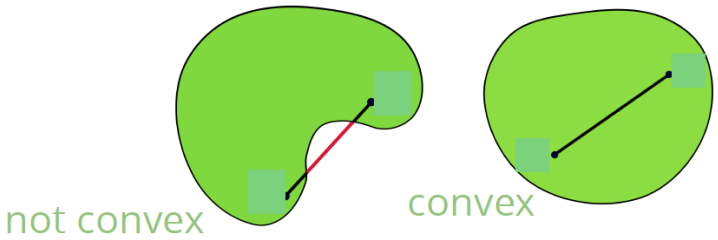
Convex combination: If $z \in \mathbb{R}^d$ is a linear combination of $x_1, x_2, \ldots, x_n \in \mathbb{R}^d$ and the coefficients are non-negative and sum to 1, then z is a convex combination of $x_1, x_2, \ldots, x_n$:

$$z = \sum_{i=1}^{n} \theta_i x_i, \quad \text{where } \forall i \in \{1, \ldots, n\}, \theta_i \geq 0 \quad \text{and} \quad \sum_{i=1}^{n} \theta_i = 1$$

All convex combinations $z = \theta x + (1-\theta)y$ of two points x and y lie on the line segment from x to y ($\theta = 1$ we get x, $\theta = 0$ we get y)

Convex Set: $\mathcal{X}$ is a convex set if the convex combination of any two points in $\mathcal{X}$ is also in $\mathcal{X}$

$$\forall x, y \in \mathcal{X};\ \forall \theta \in [0,1], \qquad z = \theta x + (1-\theta)y \in \mathcal{X}$$

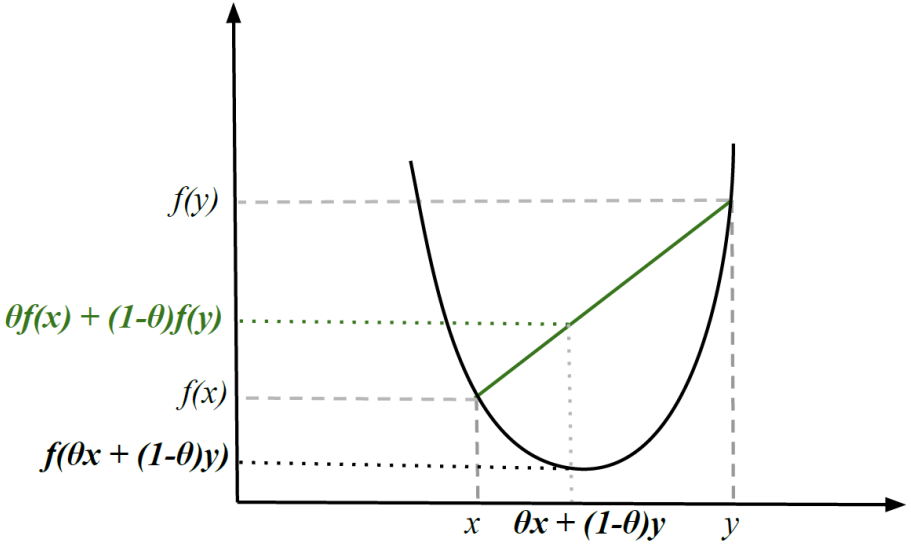


- A convex subset of $\mathbb{R}^d$ intersects any line in at most one line segment
- Convex hull $\operatorname{co} X := \left\{ \bar{x} \,\middle|\, \bar{x} = \sum_{i=1}^{n} \alpha_i x_i \ \textit{where}\ n \in \mathbb{N}, \alpha_i \geq 0 \ \textit{and} \ \sum_{i=1}^{n} \alpha_i \leq 1 \right\}$
- Convex hull of a set is the smallest convex set that contains it
- Supremum on convex hull $\sup_{x \in X} f(x) = \sup_{x \in \operatorname{co} X} f(x)$

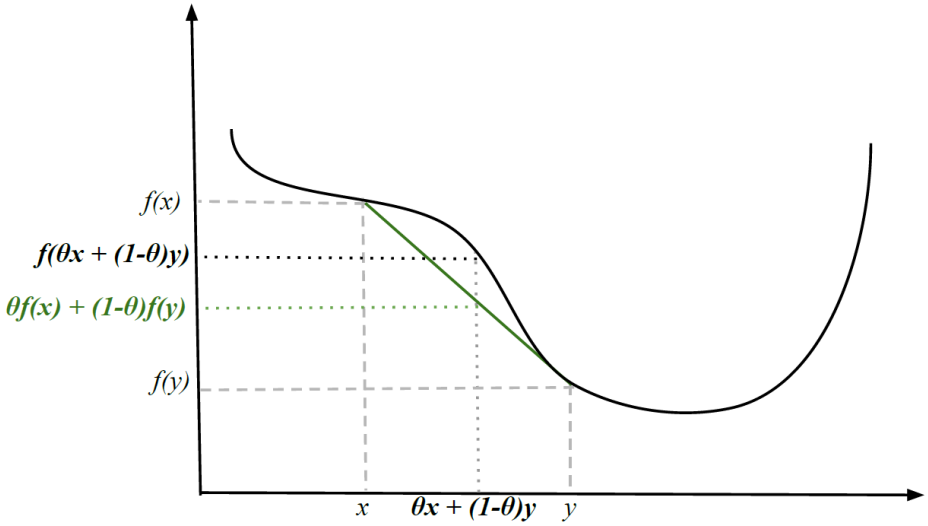
Convex Functions

***Convex function**: A function $f(x)$ (e.g., $f: \mathbb{R}^d \rightarrow \mathbb{R}$) is convex *iff* the domain $\mathrm{dom}(f)$ is a convex set and $\forall x, y \in \mathrm{dom}(f)$, $\forall \theta \in [0,1]$*

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

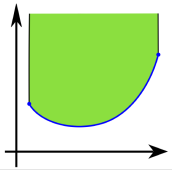


Convex function
the green chord lies above the function value



Non-convex function
there exists points x and y for which the chord lies below the curve between $f(x)$ and $f(y)$

- Convex functions are those whose *epigraph* is a convex set
  epigraph: set of all points above the graph



Convex Functions - Properties

- Every segment between two elements of $\mathcal{X}$ is also contained in $\mathcal{X}$
- *Corollary:* every local minimum is a global minimum
- Non-negative combinations of convex functions are convex (e.g., $\|WX - Y\|_2^2 + \lambda\|w\|_2^2$)

$$h(x) = af(x) + bg(x)$$

- Affine scaling of convex functions is convex

$$h(x) = f(Ax + b)$$

- Compositions of convex functions are NOT generally convex (e.g., Neural Nets)

$$h(x) = f(g(x))$$

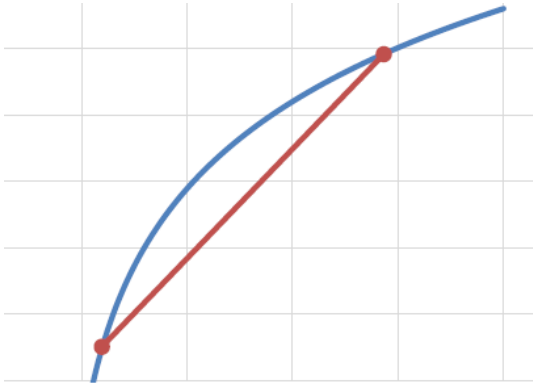
However, if f, g are convex and f non-decreasing, $h(x)$ is convex (e.g. $e^{g(x)}$)

Concave functions

$$f \text{ is convex } \Leftrightarrow h(x) = -f(x) \text{ is concave}$$

Example: $f(x) = \log x$

$$f''(x) = -\frac{1}{x^2} \leq 0$$



Convex Functions – Alternative Definitions

First-order condition: A differentiable function f is convex *iff* $\mathrm{dom}(f)$ is convex and $\forall x, y \in \mathrm{dom}(f)$

$$f(y) \geq f(x) + \nabla f(x)^{\mathrm{T}}(y - x)$$

Second-order condition: A twice differentiable function f is convex *iff* $\mathrm{dom}(f)$ is a convex set and $\forall x \in \mathrm{dom}(f)$

$$\nabla^2 f(x) \succcurlyeq 0$$

$$\nabla^2 f(x) = \begin{bmatrix} \frac{\partial^2 f}{\partial x_1^2} & \frac{\partial^2 f}{\partial x_1 \partial x_2} & \cdots & \frac{\partial^2 f}{\partial x_1 \partial x_d} \\ \vdots & \vdots & \ddots & \vdots \\ \frac{\partial^2 f}{\partial x_d \partial x_1} & \frac{\partial^2 f}{\partial x_d \partial x_2} & \cdots & \frac{\partial^2 f}{\partial x_d^2} \end{bmatrix}$$

- For a twice differentiable function to be convex, its Hessian must be positive semi-definite
- In general, for any non-negative eigenvalue of the Hessian, the curvature of the function is non-negative along the corresponding eigenvector, thus the function is convex in that direction.
- On the other hand, a non-positive eigenvalue represents non-positive curvature along the eigenvector, and the function is concave in that direction. So, for a function to be convex, it has to have non-negative curvature, thus non-negative eigenvalues, in all directions

A symmetric matrix $M \in \mathbb{R}^{d \times d}$ is positive semi-definite ($M \succcurlyeq 0$) if $\forall x \in \mathbb{R}^d$, $x^{\mathrm{T}} M x \geq 0$

A symmetric matrix $M \in \mathbb{R}^{d \times d}$ is positive semi-definite iff all its eigenvalues λ_i are non-negative

$\forall i \in \{1, \ldots, d\}, \lambda_i \geq 0$

Convex Functions - Examples

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

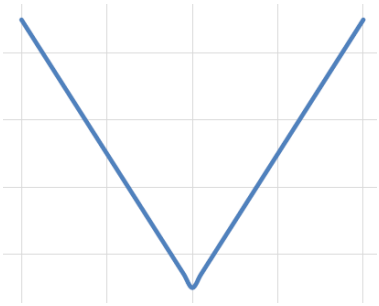
Quadratic $f(x) = x^2$

$$
\begin{aligned}
&(\alpha x + (1-\alpha)y)^2 \\
&= \alpha^2 x^2 + 2\alpha(1-\alpha)xy + (1-\alpha)^2 y^2 \\
&= \alpha x^2 + (1-\alpha)y^2 - \alpha(1-\alpha)(x^2 + 2xy + y^2) \\
&\leq \alpha x^2 + (1-\alpha)y^2
\end{aligned}
$$

$$f''(x) = 2 \geq 0$$

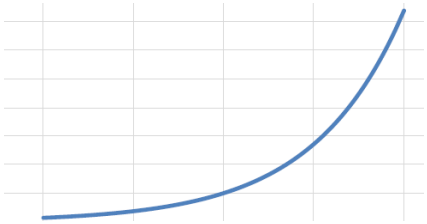
Abs $f(x) = |x|$

$$
\begin{aligned}
|\alpha x + (1-\alpha)y| &\leq |\alpha x| + |(1-\alpha)y| \\
&= \alpha|x| + (1-\alpha)|y|
\end{aligned}
$$



Exponential $f(x) = e^x$

$$
\begin{aligned}
e^{\alpha x + (1-\alpha)y} &= e^y e^{\alpha(x-y)} = e^y \sum_{n=0}^{\infty} \frac{1}{n!} \alpha^n (x-y)^n \\
&\leq e^y \left(1 + \alpha \sum_{n=1}^{\infty} \frac{1}{n!}(x-y)^n\right) \qquad (\text{if } x > y) \\
&= e^y \left((1-\alpha) + \alpha e^{x-y}\right) \\
&= (1-\alpha)e^y + \alpha e^x
\end{aligned}
$$



$$f''(x) = e^x \geq 0$$

Smoothness

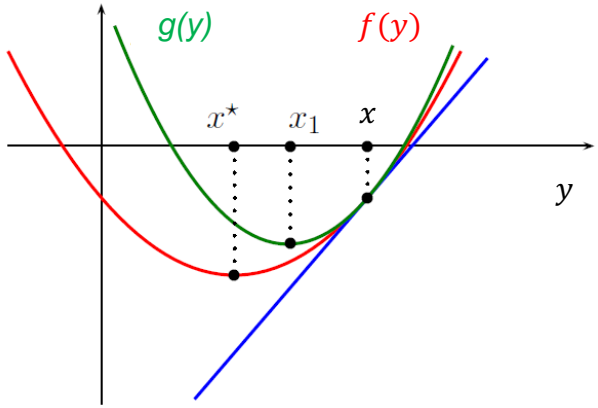
A (differentiable) function f is said to be L-smooth if its gradients are Lipschitz continuous

$$\|\nabla f(x) - \nabla f(y)\| \leq L\|x - y\|.$$

Lemma *Let $f : \mathbb{R}^d \mapsto \mathbb{R}$ be a twice differentiable function. If f is L–smooth then the following holds*

$$\langle \nabla^2 f(x)v, v\rangle \leq L\|v\|_2^2, \quad \forall x, v \in \mathbb{R}^d,$$

$$f(y) \leq \underbrace{f(x) + \langle \nabla f(x), y - x\rangle}_{\text{first order Taylor expansion } g(y)} + \frac{L}{2}\|y - x\|_2^2.$$



Lemma *If f is L–smooth then*

$$f(x - \tfrac{1}{L}\nabla f(x)) - f(x) \leq -\frac{1}{2L}\|\nabla f(x)\|_2^2,$$

and

$$f(x^*) - f(x) \leq -\frac{1}{2L}\|\nabla f(x)\|_2^2,$$

hold for all $x \in \mathbb{R}^d$.

Lemma *If $f(x)$ is convex and L–smooth then*

$$f(y) - f(x) \leq \langle \nabla f(y), y - x\rangle - \frac{1}{2L}\|\nabla f(y) - \nabla f(x)\|_2^2.$$

$$\langle \nabla f(y) - \nabla f(x), y - x\rangle \geq \frac{1}{L}\|\nabla f(x) - \nabla f(y)\| \quad (\textit{Co-coercivity}).$$

Strongly Convex Functions

We can "strengthen" the notion of convexity by defining μ-strong convexity, that is

$$f(y) \geq \underbrace{f(x) + \langle \nabla f(x), y - x \rangle}_{\text{first order Taylor expansion}} + \frac{\mu}{2}\|y - x\|_2^2, \quad \forall x, y \in \mathbb{R}^d$$

Lemma *Let f be twice continuously differentiable. The following is equivalent to f being μ–strongly convex*

$$\langle \nabla^2 f(x) v, v \rangle \geq \mu \|v\|_2^2.$$

Equivalent second-order characterization (for twice differentiable functions)

$$\nabla^2 f(\boldsymbol{x}) \succeq \mu \boldsymbol{I}, \quad \forall \boldsymbol{x}$$

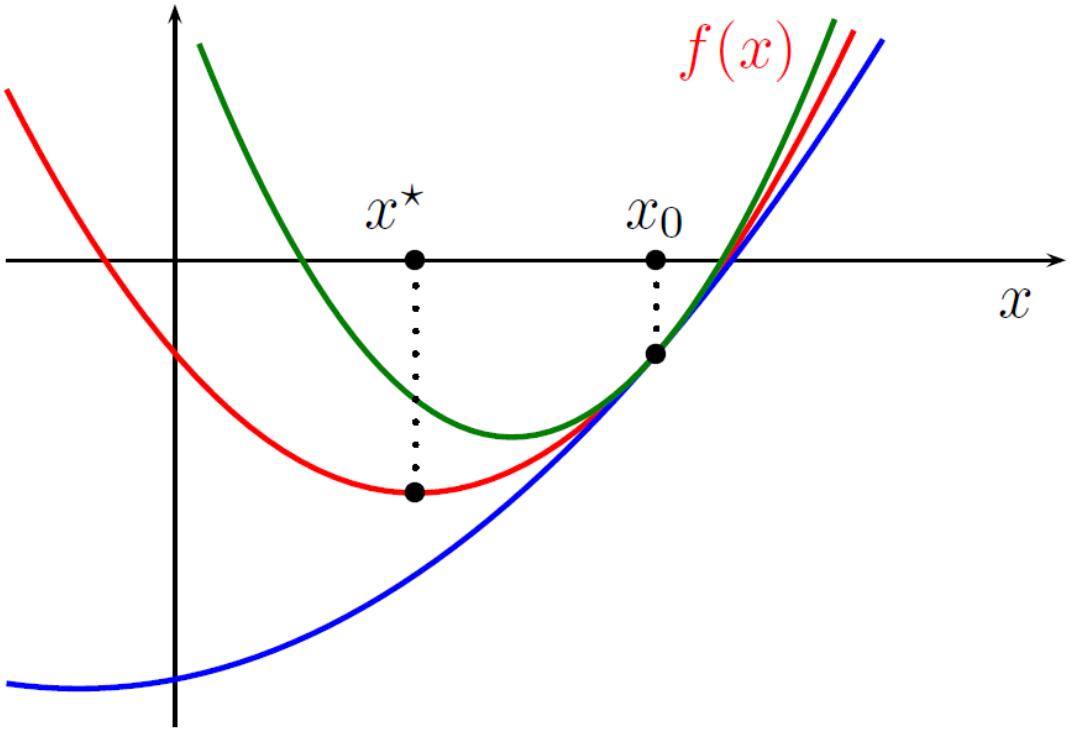
A twice differentiable function f is said to be μ-strongly convex and L-smooth if

$$\boldsymbol{0} \preceq \mu \boldsymbol{I} \preceq \nabla^2 f(\boldsymbol{x}) \preceq L \boldsymbol{I}, \qquad \forall \boldsymbol{x}$$

Strongly Convex and Smooth Functions

Main Inequality

If ∇f is L-Lipschitz continuous and f is μ-strongly convex



$$f(x) \leq f(x_0) + \nabla f(x_0)^\top (x - x_0) + \frac{L}{2}\|x - x_0\|_2^2$$

$$f(x) \geq f(x_0) + \nabla f(x_0)^\top (x - x_0) + \frac{\mu}{2}\|x - x_0\|_2^2$$

Gradient Methods

- We have the following unconstrained optimization problem

$$\min_{\boldsymbol{x}} f(\boldsymbol{x})$$
$$\text{subject to } \boldsymbol{x} \in \mathbb{R}^n$$

where the objective or cost function f is differentiable

Iterative descent algorithms

Start with a point $\boldsymbol{x}^0$, and construct a sequence $\{\boldsymbol{x}^t\}$ s.t.

$$f(\boldsymbol{x}^{t+1}) < f(\boldsymbol{x}^t), \qquad t = 0, 1, \cdots$$

- $\boldsymbol{d}$ is said to be a descent direction at $\boldsymbol{x}$ if

$$f'(\boldsymbol{x}; \boldsymbol{d}) := \underbrace{\lim_{\tau \downarrow 0} \frac{f(\boldsymbol{x} + \tau \boldsymbol{d}) - f(\boldsymbol{x})}{\tau}}_{\text{directional derivative}} = \nabla f(\boldsymbol{x})^\top \boldsymbol{d} < 0$$

Gradient Methods

- We have the following unconstrained optimization problem

$$\min_{\boldsymbol{x}} f(\boldsymbol{x})$$
$$\text{subject to } \boldsymbol{x} \in \mathbb{R}^n$$

where the objective or cost function f is differentiable

Iterative descent algorithms

Start with a point $\boldsymbol{x}^0$, and construct a sequence $\{\boldsymbol{x}^t\}$ s.t.

$$f(\boldsymbol{x}^{t+1}) < f(\boldsymbol{x}^t), \qquad t = 0, 1, \cdots$$

- In each iteration, search in descent direction

$$\boldsymbol{x}^{t+1} = \boldsymbol{x}^t + \eta_t \boldsymbol{d}^t$$

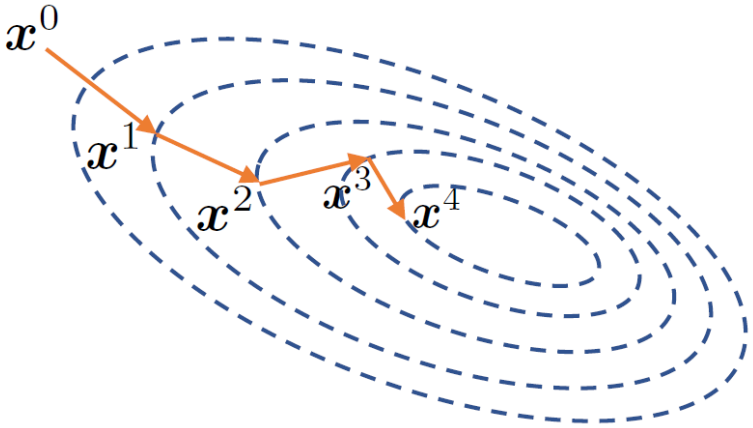
where $\boldsymbol{d}^t$: descent direction at $\boldsymbol{x}^t$; $\eta_t > 0$: stepsize

Gradient Descent

- One of the most important examples: $\boldsymbol{x}^{t+1} = \boldsymbol{x}^t - \eta_t \nabla f(\boldsymbol{x}^t)$

$$\text{descent direction: } \boldsymbol{d}^t = -\nabla f(\boldsymbol{x}^t)$$

- Traced to Augustin Louis Cauchy (1857)



ANALYSE MATHÉMATIQUE. — *Méthode générale pour la résolution des systèmes d'équations simultanées; par* M. AUGUSTIN CAUCHY.

« Étant donné un système d'équations simultanées qu'il s'agit de résoudre, on commence ordinairement par les réduire à une seule, à l'aide d'éliminations successives, sauf à résoudre définitivement, s'il se peut, l'équation résultante. Mais il importe d'observer, 1° que, dans un grand nombre de cas, l'élimination ne peut s'effectuer en aucune manière; 2° que l'équation résultante est généralement très-compliquée, lors même que les équations données sont assez simples. Pour ces deux motifs, on conçoit qu'il serait très-utile de connaître une méthode générale qui pût servir à résoudre directement un système d'équations simultanées. Telle est celle que j'ai obtenue, et dont je vais dire ici quelques mots. Je me bornerai pour l'instant à indiquer les principes sur lesquels elle se fonde, me proposant de revenir avec plus de détails sur le même sujet, dans un prochain Mémoire.

» Soit d'abord

$$u = f(x, y, z)$$

une fonction de plusieurs variables $x, y, z, \ldots$, qui ne devienne jamais négative et qui reste continue, du moins entre certaines limites. Pour trouver les valeurs de $x, y, z, \ldots$, qui vérifieront l'équation

$$(1) \qquad u = 0,$$

il suffira de faire décroître indéfiniment la fonction u, jusqu'à ce qu'elle

- Also known as steepest descent

$$\underbrace{\arg \min_{\boldsymbol{d}:\|\boldsymbol{d}\|_2 \leq 1} f'(\boldsymbol{x}; \boldsymbol{d}) = \arg \min_{\boldsymbol{d}:\|\boldsymbol{d}\|_2 \leq 1} \nabla f(\boldsymbol{x})^\top \boldsymbol{d} = -\|\nabla f(\boldsymbol{x})\|_2}_{\text{direction with the greatest rate of objective value improvement}}$$

Convex and Smooth Functions

Theorem: Let f be convex and L-smooth (∇f is L-Lipschitz).

Consider the problem $x^* = \arg\min_{x \in \mathbb{R}^d} f(x)$, and the following GD method with $\alpha = \frac{1}{L}$.

$$x^{t+1} = x^t - \alpha \nabla f(x^t)$$

Let x^t for $t = 1, \dots, n$ be the sequence of iterates generated by the above GD algorithm. Then,

$$f(x^n) - f(x^*) \leq \frac{2L}{n-1} \|x^1 - x^*\|_2^2$$

Proof

Let f be convex and L-smooth. It follows that

$$\begin{aligned}
\|x^{t+1} - x^*\|_2^2 &= \|x^t - x^* - \tfrac{1}{L}\nabla f(x^t)\|_2^2 \\
&= \|x^t - x^*\|_2^2 - 2\tfrac{1}{L}\left\langle x^t - x^*, \nabla f(x^t)\right\rangle + \tfrac{1}{L^2}\|\nabla f(x^t)\|_2^2 \\
&\overset{\text{Co-coercivity}}{\leq} \|x^t - x^*\|_2^2 - \tfrac{1}{L^2}\|\nabla f(x^t)\|_2^2.
\end{aligned}$$

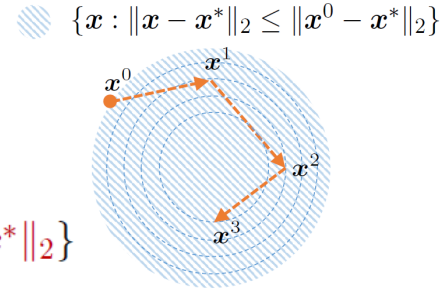
GD Convergence Rates

- For $\alpha = 1/L \Rightarrow$ linear convergence (note: $(1-\mu/L)^t \leq \exp(-t/\kappa)$)

- If f is **strongly convex**, GD converges with rate $O(c^k)$ for $0 < c < 1$, where k is the number of iterations. This means that a bound of $f\left(x^{(k)}\right) - f(x^*) \leq \epsilon$ can be achieved using only $O(\log(1/\epsilon))$ iterations [linear convergence]
- Convergence rate of GD with **convex** f is $O(1/k)$. This implies that in order to achieve a bound of $f\left(x^{(k)}\right) - f(x^*) \leq \epsilon$, we must run $O(1/\epsilon)$ iterations of GD. [sub-linear convergence]

Q: Is strong convexity necessary for linear convergence?

A: Strong convexity requirement can often be relaxed:



- Local strong convexity $\mu \boldsymbol{I} \preceq \nabla^2 f(\boldsymbol{x}) \preceq L\boldsymbol{I}, \quad \forall \boldsymbol{x} \in \mathcal{B}_0$

$$\mathcal{B}_0 := \{\boldsymbol{x} : \|\boldsymbol{x} - \boldsymbol{x}^*\|_2 \leq \|\boldsymbol{x}^0 - \boldsymbol{x}^*\|_2\}$$

- Regularity condition $\langle \nabla f(\boldsymbol{x}), \boldsymbol{x} - \boldsymbol{x}^* \rangle \geq \frac{\mu}{2}\|\boldsymbol{x} - \boldsymbol{x}^*\|_2^2 + \frac{1}{2L}\|\nabla f(\boldsymbol{x})\|_2^2, \quad \forall \boldsymbol{x}$

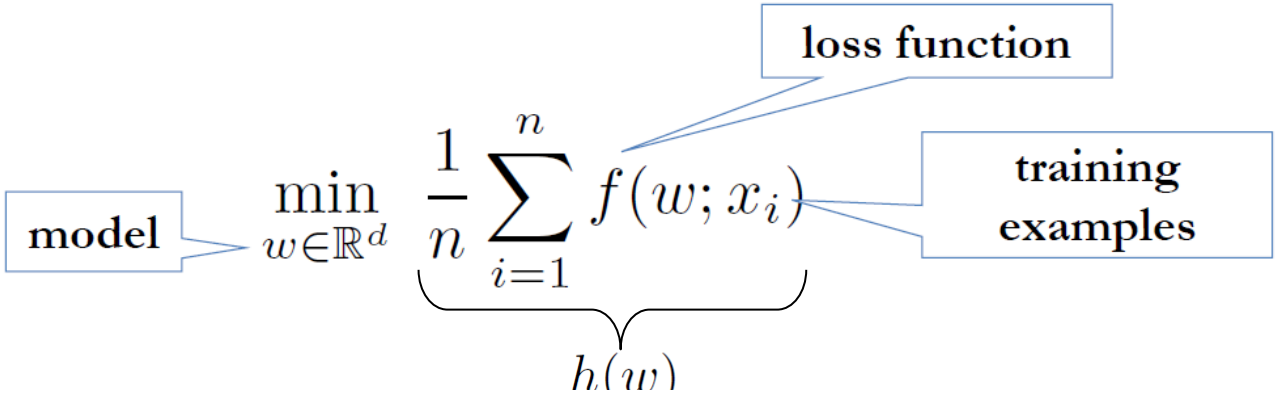
- Polyak-Lojasiewicz condition

$$\|\nabla f(\boldsymbol{x})\|_2^2 \geq 2\mu(f(\boldsymbol{x}) - f(\underbrace{\boldsymbol{x}^*}_{\text{minimizer}})), \quad \forall \boldsymbol{x}$$

  - Guarantees that gradient grows fast as we move away from the optimal objective value
  - Guarantees that every stationary point is a global minimum (does not imply uniqueness!)

Gradient Descent & Large Scale Optimization

- Most ML problems can be written as an optimization problem



- Computing the gradient takes $O(n)$ time $\nabla h(w) = \frac{1}{n} \sum_{i=1}^{n} \nabla f(w; x_i)$
- GD with More Data (e.g. add an extra copy of every examples in the training set)

$$\nabla h(w) = \frac{1}{2n} \sum_{i=1}^{n} \nabla f(w; x_i) + \frac{1}{2n} \sum_{i=1}^{n} \nabla f(w; x_i)$$

- Same objective function
- But gradients take 2x the time to compute
- How to scale up to large datasets???

Gradient Descent in a Nutshell

Pros

- Simple algorithm - easy to implement
  - each iteration is computationally cheap; just need to compute a gradient
- Can be very fast for smooth objective functions, i.e. well-conditioned and strongly convex

Cons

- Often slow - many relevant optimization problems are not *strongly convex*
- Cannot handle non-differentiable functions (need for sub-gradients methods)

- GD is not subject to variance
  - at every step we compute the average gradient using the whole dataset.

- However, every step can be very computationally expensive
  - $O(nd)$ per iteration, where n is # samples in dataset and d is dimensions of x.
  - Since we need $O(d)$ iterations to converge, the total problem cost is about $O(nd^2)$

Stochastic Gradient Descent

- **Key idea:** Do not process the entire training set to compute the gradient at every step.
  Just use one training example [*sample a data point + locally optimize*]

- **Intuition:** in expectation, SGD equals the full gradient, i.e., it is an unbiased estimate of the full gradient

- Desirable practical behavior
  - Applicable (at least) to least-squares and logistic regression
  - Robustness to (potentially unknown) constants (*L*, *B*, *μ*)
  - Adaptivity to difficulty of the problem (e.g., strong convexity)

f *B*-Lipschitz continuous
f *μ*-strongly convex
f' *L*-Lipschitz continuous

- SGD has been around for a while, for good reasons
  - Robust to noise
  - Simple to implement
  - Near-optimal learning performance *
  - Small computational footprint (much faster per iteration than GD: $\mathrm{O}(d)$ instead of $\mathrm{O}(nd)$.

- Not a descent method: unlike GD, SGD does not necessarily decrease the value of the loss at each iteration/step (progress “on average”)

Stochastic Programming

$$\text{minimize}_{\boldsymbol{x}} \quad \underbrace{F(\boldsymbol{x}) = \mathbb{E}\left[f(\boldsymbol{x};\boldsymbol{\xi})\right]}_{\text{expected risk, population risk, ...}}$$

- $\boldsymbol{\xi}$: randomness in the problem
- *Assumption*: $f(\cdot, \boldsymbol{\xi})$ is convex for every $\boldsymbol{\xi}$ (and hence $F(\cdot)$ is convex)

- Under "mild" technical conditions

$$\begin{aligned}\boldsymbol{x}^{t+1} &= \boldsymbol{x}^t - \eta_t \nabla F(\boldsymbol{x}^t) \\ &= \boldsymbol{x}^t - \eta_t \nabla \mathbb{E}\left[f(\boldsymbol{x}^t;\boldsymbol{\xi})\right] \\ &= \boldsymbol{x}^t - \eta_t \mathbb{E}\left[\nabla_{\boldsymbol{x}} f(\boldsymbol{x}^t;\boldsymbol{\xi})\right]\end{aligned}$$

- **Issues:**
  - Distribution of $\boldsymbol{\xi}$ may be unknown
  - Even if it is known, evaluating high dimensional expectation is often very computationally expensive

Stochastic Gradient Descent

Stochastic Approximation (SGD) [Robbins, Monro'51]

$$\boldsymbol{x}^{t+1} = \boldsymbol{x}^{t} - \eta_t \boldsymbol{g}(\boldsymbol{x}^{t}; \boldsymbol{\xi}^{t})$$

where $\boldsymbol{g}(\boldsymbol{x}^{t}; \boldsymbol{\xi}^{t})$ is unbiased estimate of $\nabla F(\boldsymbol{x}^{t})$, i.e., $\mathbb{E}[\boldsymbol{g}(\boldsymbol{x}^{t}; \boldsymbol{\xi}^{t})] = \nabla F(\boldsymbol{x}^{t})$

- Typically, $\boldsymbol{\xi}^{t}$ is simply i_t, the random index drawn from $\{1, \dots, M\}$, and the stochastic vector $\boldsymbol{g}(\boldsymbol{x}^{t}; \boldsymbol{\xi}^{t})$ is simply $\nabla f_{i_t}(\boldsymbol{x}^{t})$
- Bias: $\text{bias}\big(\boldsymbol{g}(\boldsymbol{x}^{t}; \boldsymbol{\xi}^{t})\big) \equiv \mathbb{E}_{\boldsymbol{\xi}^{t}}[\boldsymbol{g}(\boldsymbol{x}^{t}; \boldsymbol{\xi}^{t})] - \nabla f(\boldsymbol{x}^{t})$
- Variance: $\text{Var}\big(\boldsymbol{g}(\boldsymbol{x}^{t}; \boldsymbol{\xi}^{t})\big) \equiv \mathbb{E}_{\boldsymbol{\xi}^{t}}\big[\|\boldsymbol{g}(\boldsymbol{x}^{t}; \boldsymbol{\xi}^{t})\|_2^2\big] - \big\|\mathbb{E}_{\boldsymbol{\xi}^{t}}[\boldsymbol{g}(\boldsymbol{x}^{t}; \boldsymbol{\xi}^{t})]\big\|_2^2$

SGD for Empirical Risk Minimization

$$\text{minimize}_{\boldsymbol{x}} \quad \underbrace{F(\boldsymbol{x}) := \frac{1}{n}\sum_{i=1}^{n} f(\boldsymbol{x}; \{\boldsymbol{a}_i, y_i\})}_{\text{empirical risk}}$$

$\{\boldsymbol{a}_i, y_i\}_{i=1}^n$

n random samples

e.g., quadratic loss

$$f(\boldsymbol{x}; \{\boldsymbol{a}_i, y_i\}) = (\boldsymbol{a}_i^\top \boldsymbol{x} - y_i)^2$$

for $t = 0, 1, \ldots$

choose i_t uniformly at random, and run

$$\boldsymbol{x}^{t+1} = \boldsymbol{x}^t - \eta_t \nabla_{\boldsymbol{x}} f_{i_t}(\boldsymbol{x}^t; \{\boldsymbol{a}_i, y_i\})$$

- **Benefits:** SGD exploits information more efficiently than batch methods
  - Real-world data usually involves lots of redundancy; using all data simultaneously in each iteration might be inefficient
  - SGD is particularly efficient at the very beginning, as it achieves fast initial improvement with very low per-iteration cost

Convergence – Fixed stepsizes

$$\text{minimize}_{\boldsymbol{x}} \quad F(\boldsymbol{x}) := \mathbb{E}\big[f(\boldsymbol{x};\boldsymbol{\xi})\big]$$

- F: μ-strongly convex, L-smooth
- $g(\boldsymbol{x}^t;\boldsymbol{\xi}^t)$: an **unbiased** estimate of $\nabla F(\boldsymbol{x}^t)$ given $\{\boldsymbol{\xi}^0,\cdots,\boldsymbol{\xi}^{t-1}\}$
- for all $\boldsymbol{x}$,

$$\mathbb{E}\big[\|g(\boldsymbol{x};\boldsymbol{\xi})\|_2^2\big] \le \sigma_g^2 + c_g\|\nabla F(\boldsymbol{x})\|_2^2$$

Gradient "Variance"
(bound on the 2nd moment)

$$c_g \ge \mu^2 > 0$$

$$\nabla F(\boldsymbol{x})^T \mathbb{E}_{\boldsymbol{\xi}}[\boldsymbol{g}(\boldsymbol{x};\boldsymbol{\xi})] \ge \mu\|\nabla F(\boldsymbol{x})\|_2^2$$

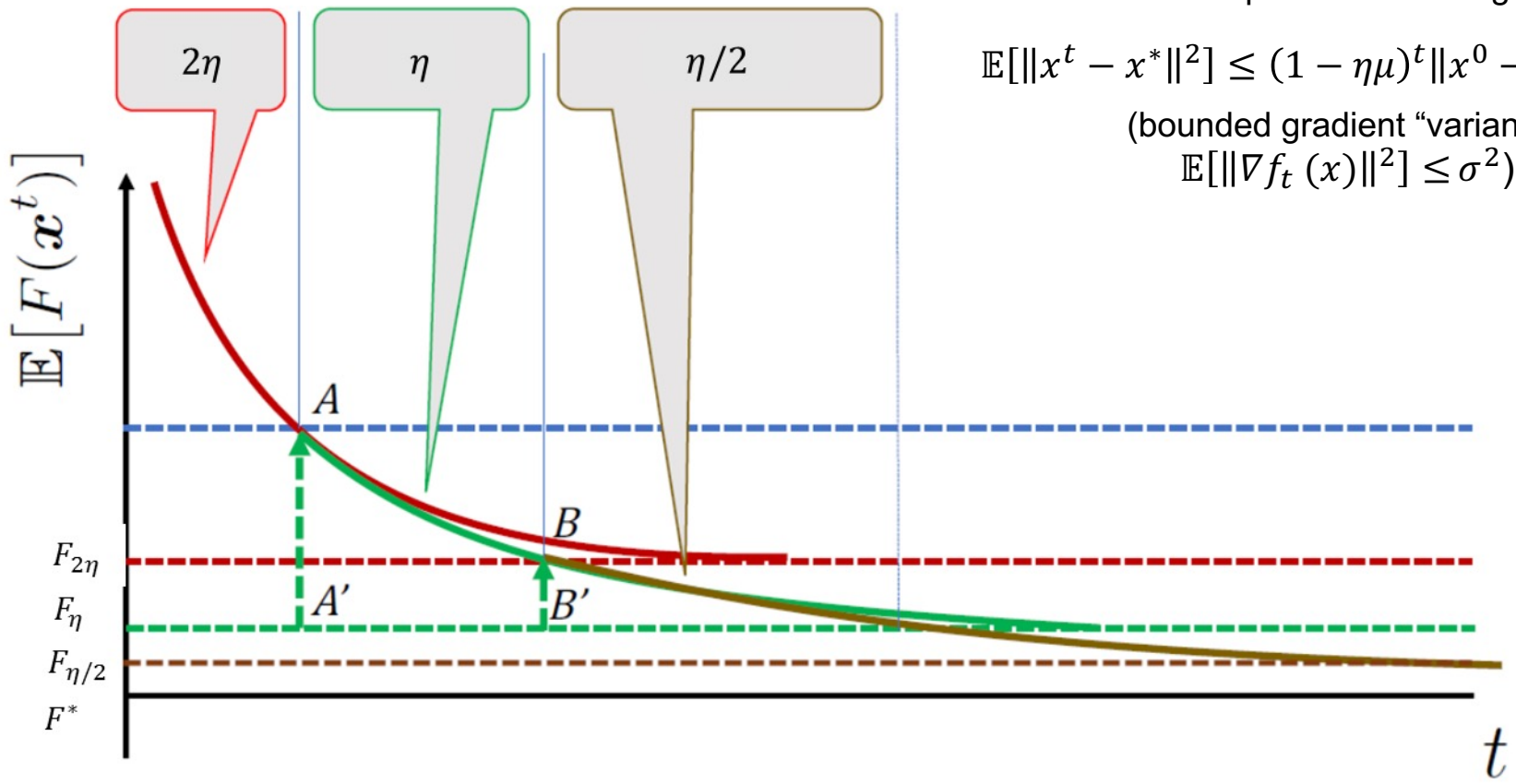
- **Fixed stepsizes**: If $\eta_t \equiv \eta \le \frac{1}{Lc_g}$, then SGD achieves

$$\mathbb{E}[F(\boldsymbol{x}^t) - F(\boldsymbol{x}^*)] \le \frac{\eta L\sigma_g^2}{2\mu} + (1-\eta\mu)^t\big(F(\boldsymbol{x}^0) - F(\boldsymbol{x}^*)\big) \xrightarrow{t\to\infty} \frac{\eta L\sigma_g^2}{2\mu}$$

- Fast (linear) convergence at the very beginning
- Converges to some neighborhood of $\boldsymbol{x}^*$ – variation in gradient computation prevents further progress
- For noiseless gradient computation (i.e., $\sigma_g^2 = 0$), it converges linearly to optimal points
- Smaller stepsizes η yield better converging points

One practical strategy

- Run SGD with fixed stepsize



- Intuition: SGD (with fixed stepsize) converges to a ***noise ball*** (iterates are randomly jumping around some space surrounding the optimum).

$$\mathbb{E}[\|x^t - x^*\|^2] \leq (1-\eta\mu)^t \|x^0 - x^*\|^2 + \frac{\eta\sigma^2}{\mu}$$

(bounded gradient "variance" $\mathbb{E}[\|\nabla f_t(x)\|^2] \leq \sigma^2$)

- Whenever progress stalls, reduce the stepsizes (e.g., half) and continue

Bottou, Curtis, Nocedal, Optimization methods for large-scale machine learning, SIAM Review, 2018

Convergence – Diminishing stepsizes

- **Diminishing stepsizes:** If $\eta_t = \frac{\beta}{\gamma+t}$ for some $\beta > \frac{1}{c\mu}$, and $\gamma > 0$, such that $\eta_1 \leq \frac{\mu}{Lc_g}$ then

$$\mathbb{E}[F(\boldsymbol{x}^t) - F(\boldsymbol{x}^*)] \leq \frac{\nu}{\gamma + t}$$

where $\nu = \max\left\{\frac{\beta^2 L\sigma_g^2}{2(\beta c\mu - 1)}, (\gamma+1)(F(\boldsymbol{x}^1) - F(\boldsymbol{x}^*))\right\}$

- Convergence rate $O(1/t)$ with decreasing stepsize $\eta_t \asymp \frac{1}{t}$
- SGD with stepsizes $\eta_t \asymp \frac{1}{t}$ is optimal!
- Sufficient conditions for convergence:

$$\sum_{t=1}^{\infty} {\eta_t}^2 < \infty \qquad \sum_{t=1}^{\infty} \eta_t = \infty$$

- Role of the Initial Point:
  - Fixed stepsize: the initial gap $\|F(x^0) - F(x^*)\|$ appears with an exponentially decreasing factor
  - Diminishing stepsizes: gap appears prominently in the second term defining ν
  - In practice, the initial stepsize should be chosen as large as allowed, i.e., $\eta_t = \mu/(Lc_g)$.

GD vs. SGD in a Nutshell

Convergence rates

- Full (batch) GD: For convex f, with L-Lipschitz gradient, under suitable stepsizes:

$$f(\boldsymbol{x}^t) - f(\boldsymbol{x}^*) = O(1/t)$$

- SGD: For convex f, under diminishing stepsizes (along with other conditions):

$$\mathbb{E}[f(\boldsymbol{x}^t)] - f(\boldsymbol{x}^*) = O(1/\sqrt{t})$$

(Here t can be a full iteration or a batch)

Convergence rates (strong convexity)

Under *strong convexity* assumptions on f (with parameter μ)

- Full GD: For strongly convex f, with L-Lipschitz gradient, for suitable stepsizes GD has a linear rate:

$$f(\boldsymbol{x}^t) - f(\boldsymbol{x}^*) = O(\rho^t) \quad \text{where } \rho < 1$$

- SGD: under strong convexity (plus other assumptions as before), SG sequence has sublinear rate:

$$\mathbb{E}[f(\boldsymbol{x}^t)] - f(\boldsymbol{x}^*) = O(1/t)$$

- Can we do better than sublinear convergence for the SG method?

SGD - Remarks

- Convergence rates are not great, but the complexity per iteration is small (1 gradient evaluation for minimizing an empirical risk vs. n for the full batch algorithm)
- When the amount of data is infinite, the method minimizes the expected risk (which is what we want)
- Choosing a good learning rate automatically is an open problem

The Hidden Cost of SGD

- By switching to SGD, we eliminated the costly sum over the dataset

$$\boldsymbol{x}^{t+1} = \boldsymbol{x}^t - \eta_t \frac{1}{n} \sum_{i=1}^{n} \nabla f(\boldsymbol{x}^t, \boldsymbol{\xi}_i) \quad \Longrightarrow \quad \boldsymbol{x}^{t+1} = \boldsymbol{x}^t - \eta_t \nabla f(\boldsymbol{x}^t, \boldsymbol{\xi}_{i_t})$$

- But the cost of computing the individual gradients remains

- The capabilities of ML methods are limited by the computing time rather than the sample size
- This is where SGD performs very well
- So, use SGD when training time is the bottleneck.

Minibatch Gradient Descent

- GD: all examples at once $\boldsymbol{x}^{t+1} = \boldsymbol{x}^t - \eta_t \frac{1}{n} \sum_{i=1}^{n} \nabla f(\boldsymbol{x}^t; \boldsymbol{\xi}_i)$
- SGD: one example at a time $\boldsymbol{x}^{t+1} = \boldsymbol{x}^t - \eta_t \nabla f(\boldsymbol{x}^t; \boldsymbol{\xi}_{i_t})$

- Is it *really all or nothing*? What about an intermediate approach?

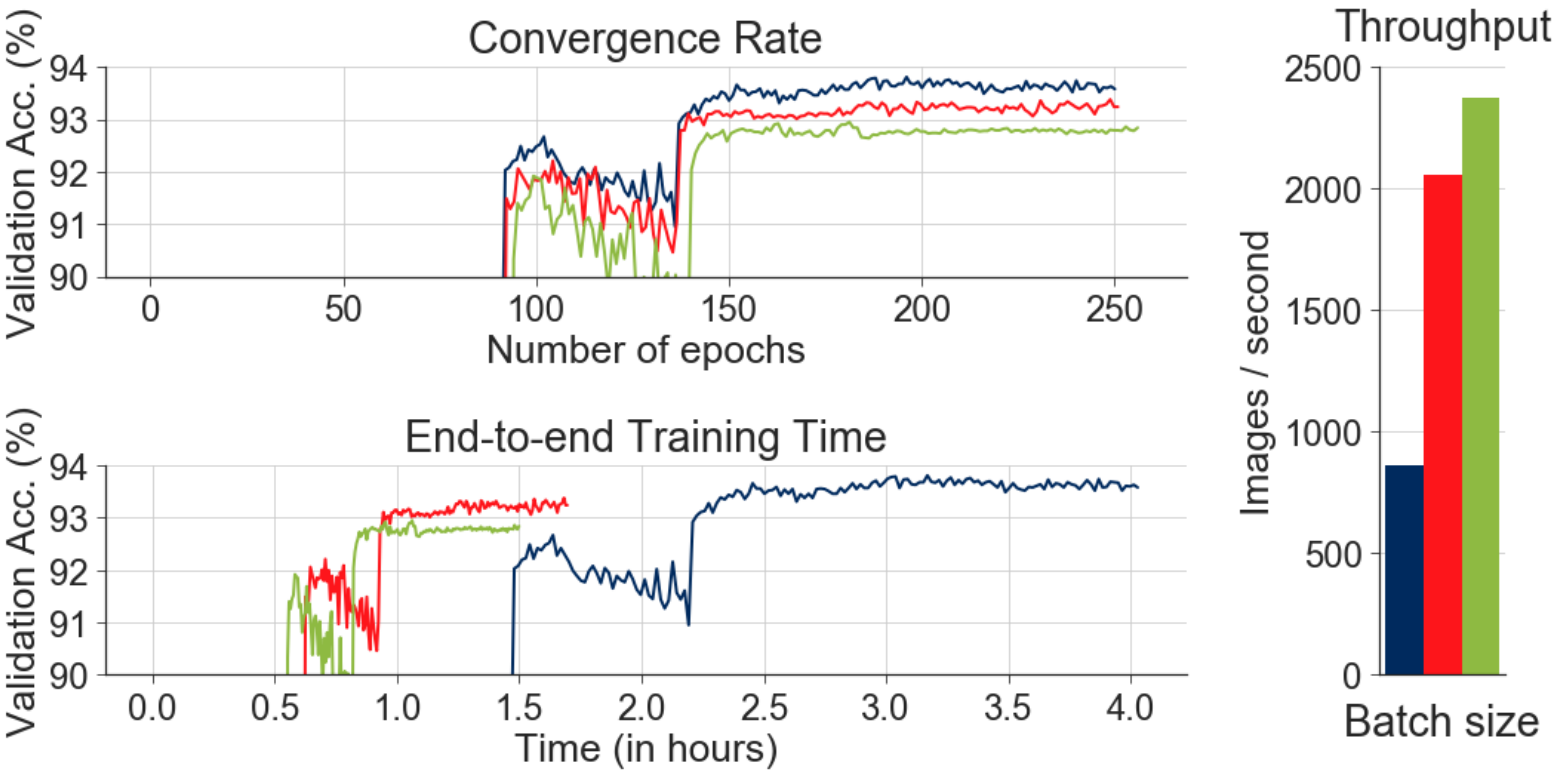
- **Minibatch GD**: calculate gradients over a subset of training points instead of just one

$$\boldsymbol{x}^{t+1} = \boldsymbol{x}^t - \eta_t \frac{1}{|B_t|} \sum_{i \in B_t}^{n} \nabla f(\boldsymbol{x}^t; \boldsymbol{\xi}_i)$$

where B_t is sampled uniformly from the set of all subsets of $\{1, \dots, n\}$ of size b.

- The parameter b is the **batch size** (typically $b \ll n$)
- Not a descent method, but "closer" to one
- Less time to compute each update than GD (only needs to sum up b gradients, rather than n)
- But iterations more expensive than SGD
- *What's the benefit? Maybe it converges faster than SGD?* (converges to a smaller noise ball)

Minibatch Gradient Descent

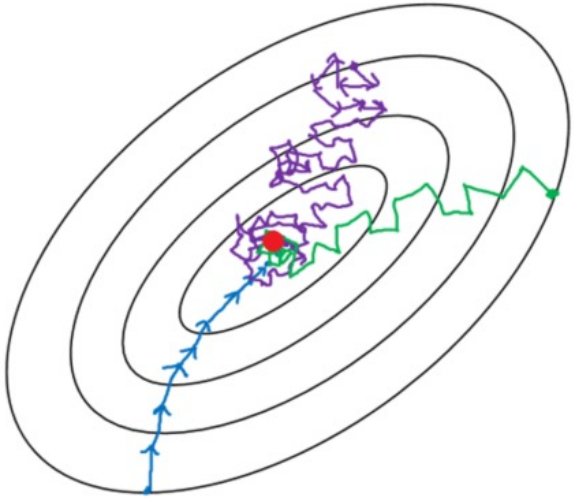


Effect of minibatch size on convergence rate, throughput, and end-to-end training time of a ResNet56 CIFAR10 model.

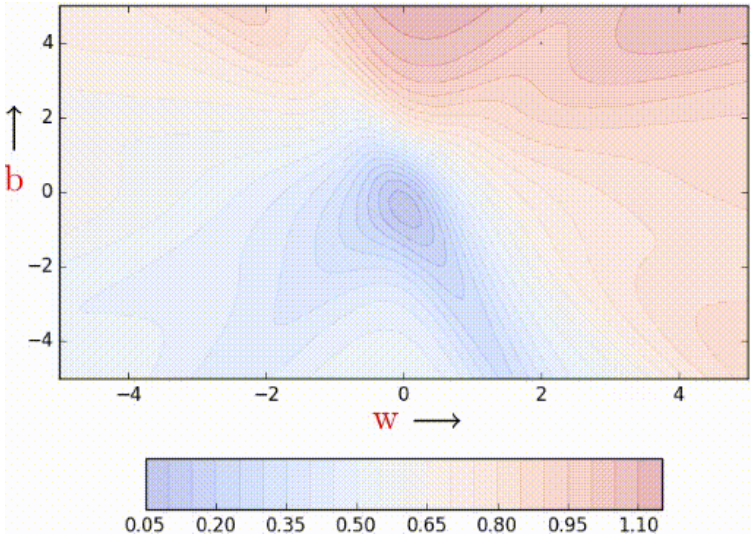
- A larger minibatch size gives better throughput (images processing per unit time), but lower final validation accuracy!

https://dawn.cs.stanford.edu/benchmark/index.html

GD vs. SGD vs. Minibatch GD



Batch gradient descent
Mini-batch gradient Descent
Stochastic gradient descent



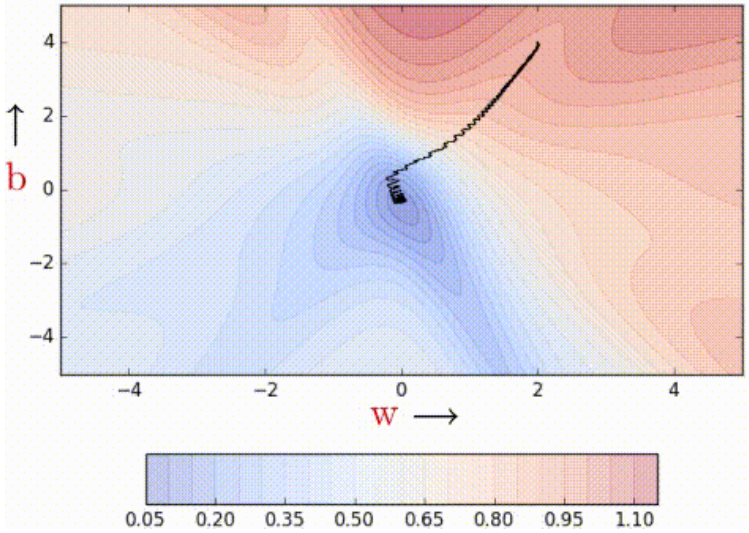
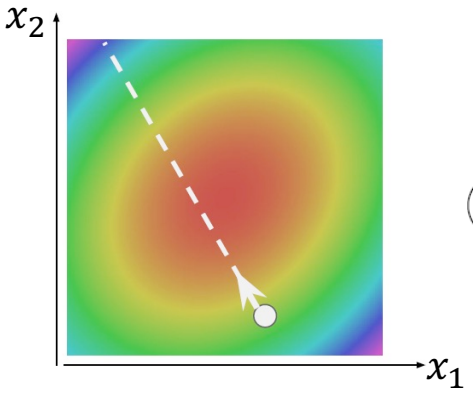
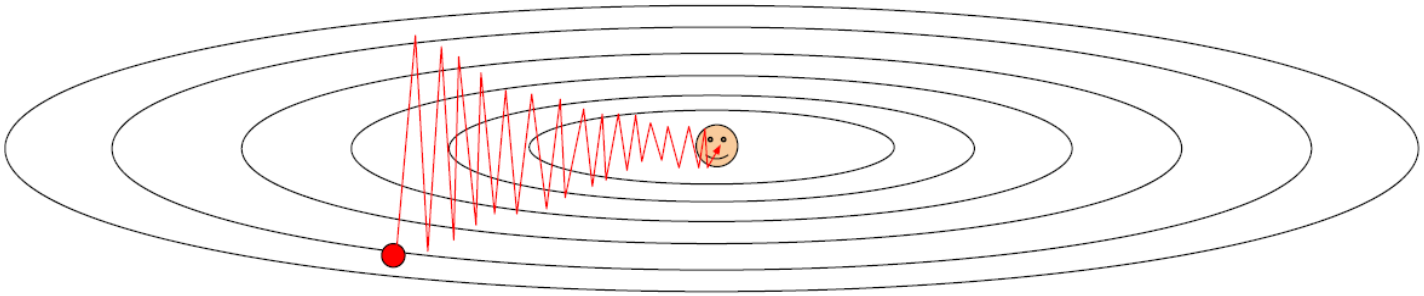


Image credit: A.L. Chandra / Ethan Irby

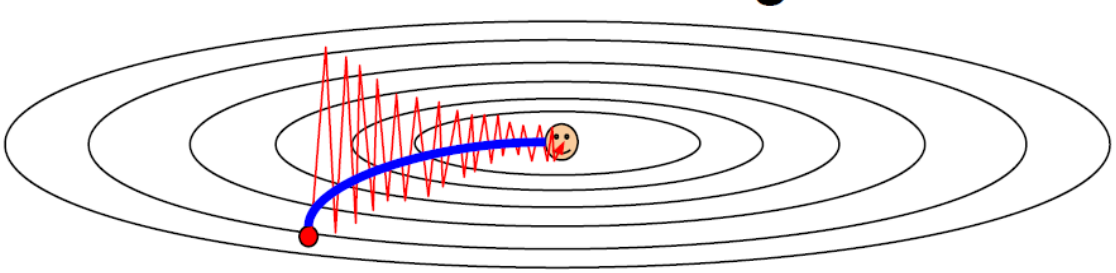
Problems with SGD

- What if loss changes quickly in one direction and slowly in another?
- What does gradient descent do?
- Very slow progress along shallow dimension, jitter along steep direction



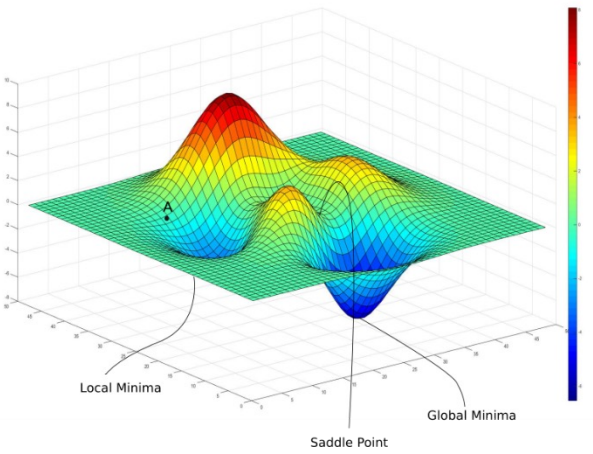


- **Poor conditioning:** loss function has high **condition number,** i.e., ratio of largest to smallest singular value of the Hessian matrix is large

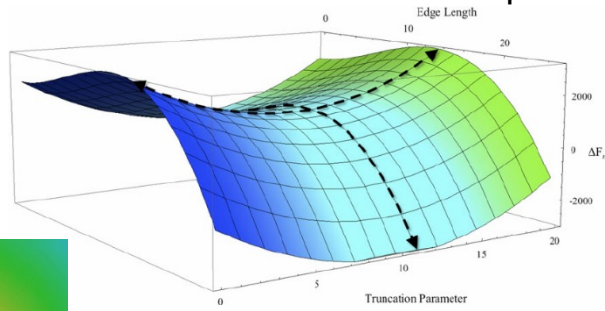


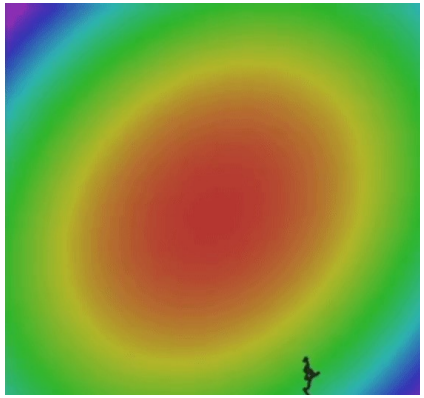
Problems with SGD (cont'd)

- What if the loss function has a local minima or saddle point?
- Zero gradient, gradient descent gets stuck
- **Saddle point**: minima in one direction (x) and local maxima in another direction (y). If the contour is flatter towards the x direction, (S)GD would keep oscillating in the y direction (illusion we have converged to a minima).
- Saddle points much more common in high dimensions
- **Noisy gradients**: common in practice (mini-batches used)



Saddle point





EURECOM
Sophia Antipolis

Improving SGD

- Diminishing stepsizes (learning rates) are needed due to the variability of stochastic gradients.
- Can we reduce the variability of those stochastic gradients and use a constant learning rate?
- Can we use better learning rate schedules to accelerate SG method?

Noise reduction Methods

Reduce the errors in the gradient estimates and/or iterate sequence

- Dynamic sampling: gradually increase the minibatch size used in the gradient computation, thus employing increasingly more accurate gradient estimates as the optimization process proceeds.
- Gradient aggregation: improve the quality of the search directions by storing gradient estimates corresponding to samples employed in previous iterations, updating one (or some) of these estimates in each iteration, and defining the search direction as a weighted average of these estimates.
- Iterative averaging: instead of averaging gradient estimates, maintain an average of iterates computed during the optimization process.

Gradient Methods with Momentum

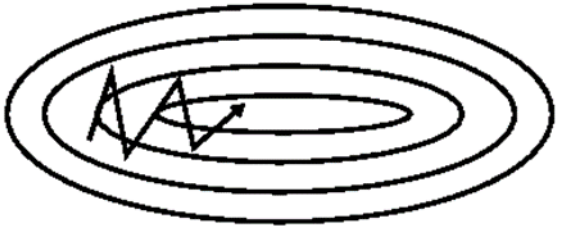
Each step is chosen as a combination of the steepest descent direction and the most recent iterate displacement.

$$x_{t+1} = x_t - \eta_t \nabla F(x_t) + \beta_t (x_t - x_{t-1})$$

scalar sequences $\{\eta_t\}$ and $\{\beta_t\}$ are either predetermined or set dynamically

SGD

$$x_t = x_{t-1} - \eta \nabla f(x_{t-1})$$



SGD + Momentum

$$v_t = \rho v_{t-1} - \eta \nabla f(x_{t-1}) \longleftarrow$$

Effect of gradient is to increment the previous velocity. The velocity also decays by ρ

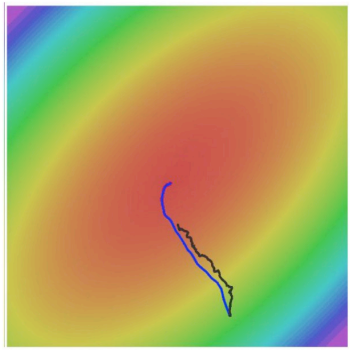
$$x_t = x_{t-1} + v_t \longleftarrow$$

Weight change is equal to the current velocity.

$$\Delta x_t = \rho v_{t-1} - \eta \nabla f(x_{t-1}) = \rho \Delta x_{t-1} - \eta \nabla f(x_{t-1}) \longleftarrow$$

Weight change can be expressed in terms of the previous weight change and the current gradient.

- ρ gives "friction"; typically ρ = 0.9 or 0.99



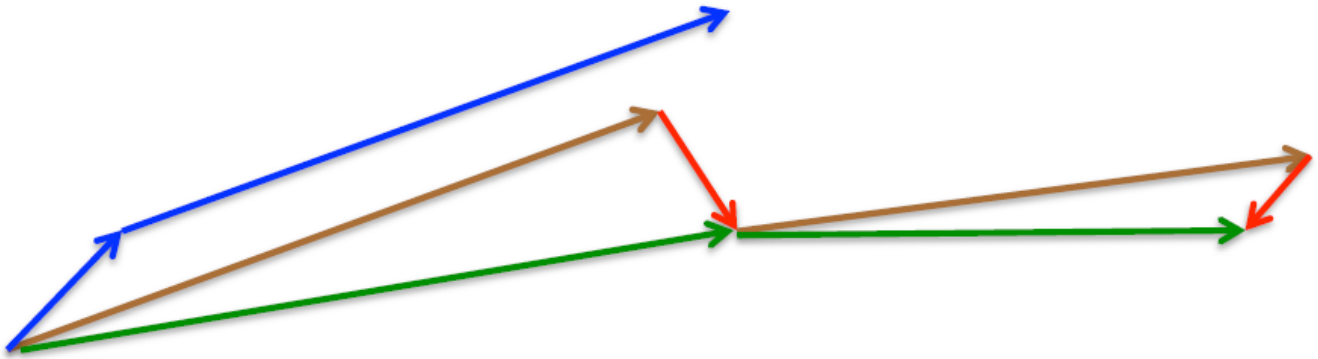
www.willamette.edu/~gorr/classes/cs449/momrate.html

A better type of Momentum (Nesterov)

- The standard momentum method first computes the gradient at the current location and then takes a big jump in the direction of the updated accumulated gradient.

Nesterov momentum

- First make a big jump in the direction of the previous accumulated gradient.
- Then measure the gradient where you end up and make a correction.
  - It's better to correct a mistake after you have made it!

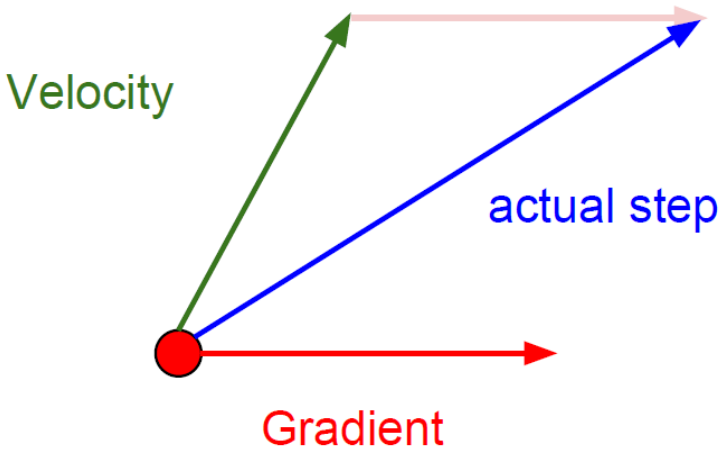


brown vector = jump, red vector = correction,
green vector = accumulated gradient
blue vectors = standard momentum

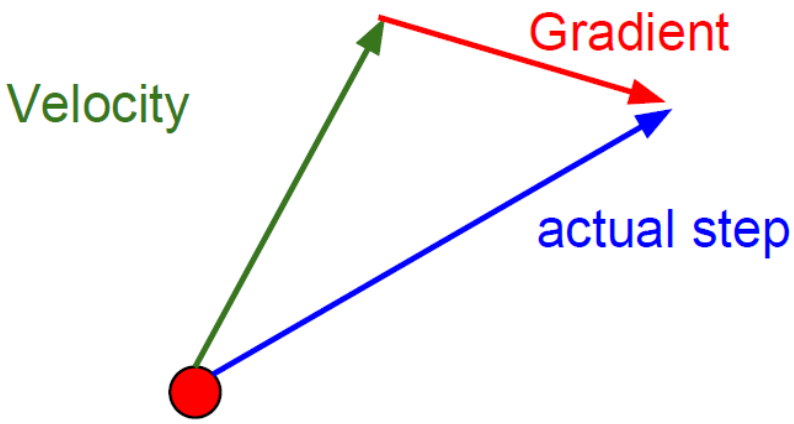
- Nesterov, "A method of solving a convex programming problem with convergence rate $O(1/k^2)$", 1983
- I. Sutskever et al, "On the importance of initialization and momentum in deep learning", ICML 2013

A better type of Momentum (Nesterov)(cont'd)

SGD + Momentum



Nesterov Momentum



Nesterov Momentum

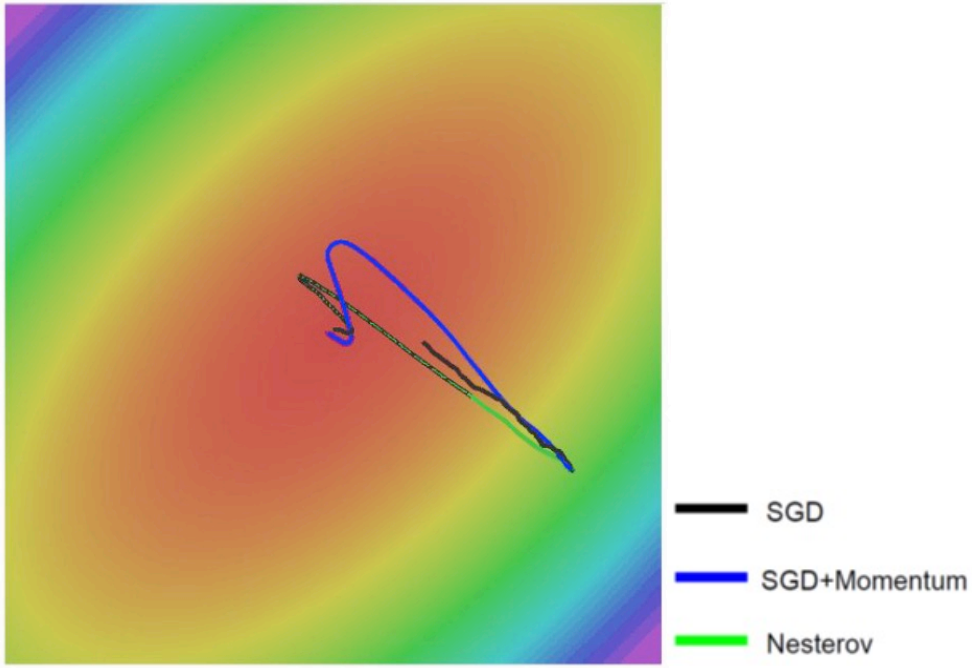
$$v_t = \rho v_{t-1} - \eta \nabla f(x_{t-1} + \rho v_{t-1})$$
$$x_t = x_{t-1} + v_t$$

usually, we want update in terms of $x_{t-1}, \nabla f(x_{t-1})$

Change of variables $\tilde{x}_{t-1} = x_{t-1} + \rho v_{t-1}$ and rearrange:

$$v_t = \rho v_{t-1} - \eta \nabla f(\tilde{x}_{t-1})$$

$$\tilde{x}_t = \tilde{x}_{t-1} - \rho v_{t-1} + (1+\rho) v_t$$
$$= \tilde{x}_{t-1} + v_t + \rho(v_t - v_{t-1})$$



AdaGrad

- **In a Nutshell:** use history of sampled gradients to adapt the step size for the next SGD step to be inversely proportional to the usual magnitude of gradient steps in that direction.
- **Key idea:** "learn slowly" from frequent features but "pay attention" to rare but informative feature

- Define *per-feature learning rate* for feature *j* as

$$\eta_{t,j} = \frac{\eta}{\sqrt{G_{t,j}}} \qquad G_{t,j} = \sum_{i=1}^{t} g_{k,j}^2$$

$G_t \in \mathbb{R}^{d \times d}$ diagonal matrix where each diagonal element (j,j) is the sum of squares of gradients of feature j up to time step t

$$\boldsymbol{x}^{t+1} = \boldsymbol{x}^t - \frac{\eta}{\zeta + \sqrt{G_t}} \odot \boldsymbol{g}^t$$

$\odot$ matrix-vector product

$\zeta > 0$: smoothing constant to prevent division by 0 (order of 10^{-8})

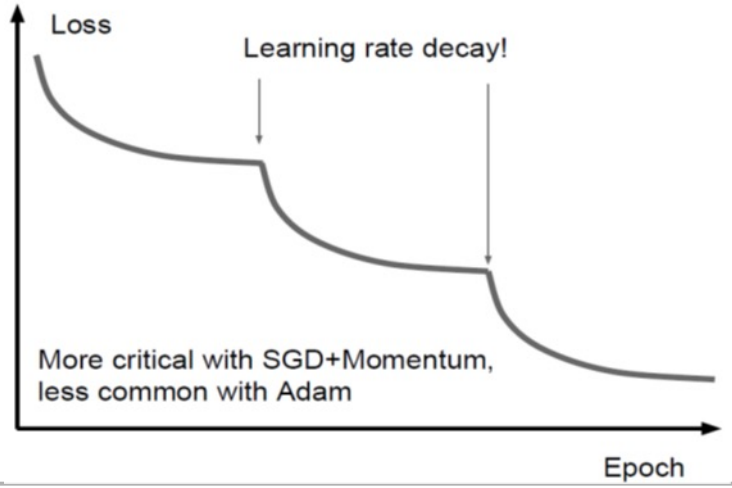
- Frequently occurring features in the gradients get smaller updates (low learning rates); rare/infrequent features get higher learning rates
- Well suited for dealing with sparse data
- No need to manually tune the learning rate (default value 0.01).
- $G_{t,j}$ keeps growing during training $\Rightarrow$ learning rate shrinks and may become infinitesimally small, at which point the algorithm is no longer able to acquire additional knowledge.

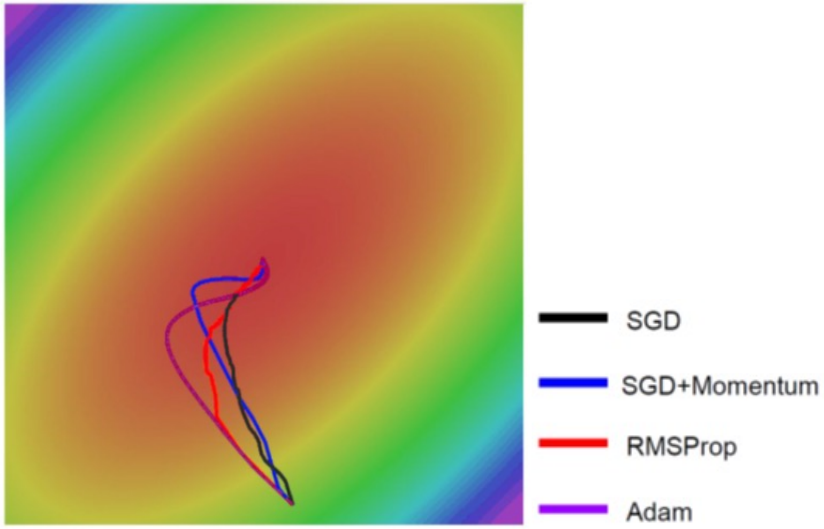
Gradient-based Optimization Methods

- **RMS Prop:** use a moving average of squared gradients to normalize the gradient.
  - This normalization balances the step size (momentum), decreasing the step for large gradients to avoid exploding, and increasing the step for small gradients to avoid vanishing.
  - RMSprop uses an adaptive learning rate instead of treating the learning rate as a hyperparameter. This means that the learning rate changes over time.

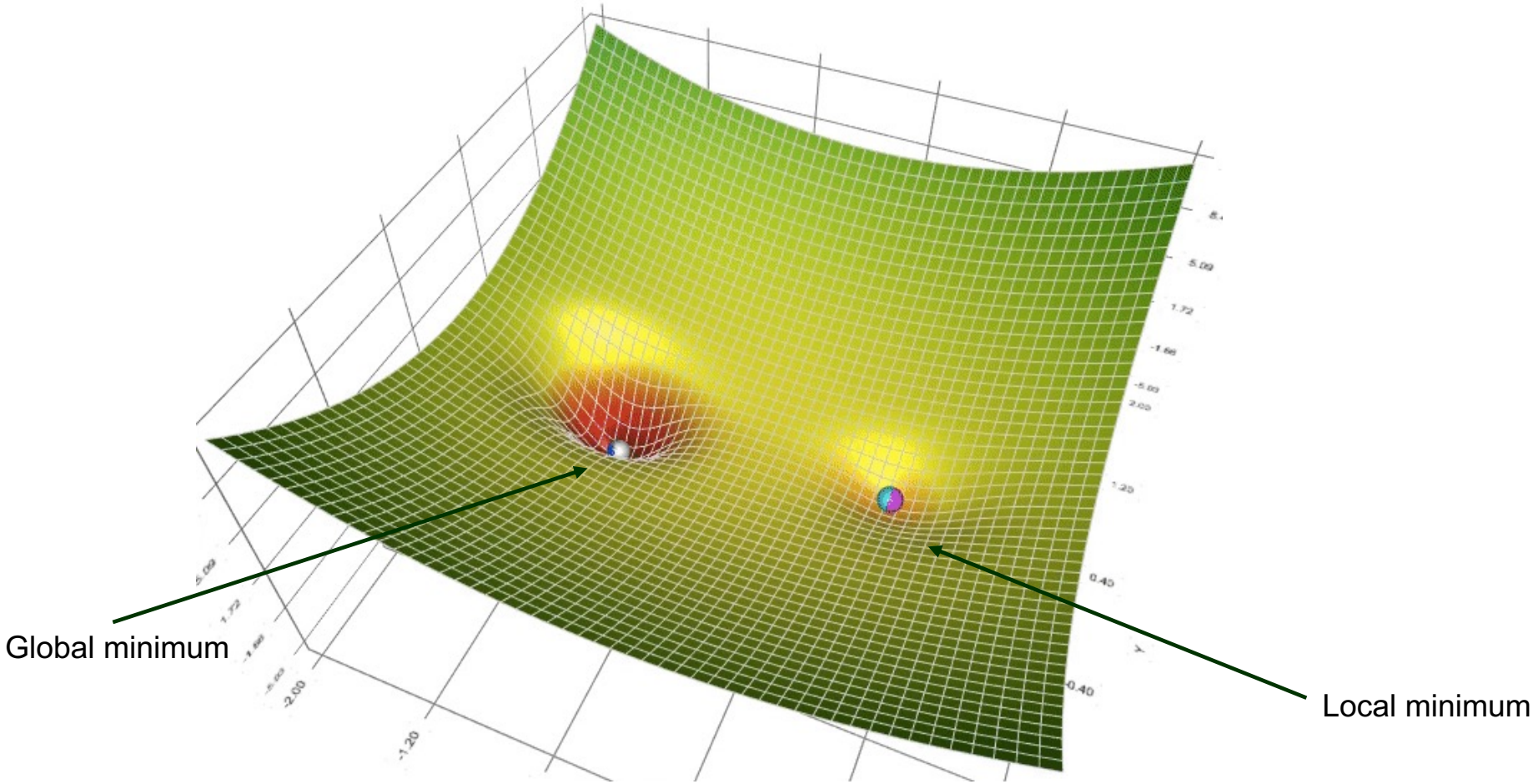
- **Adam:** it is basically RMS Prop + Momentum - default choice in practice!
- Many other methods: AdaDelta, AdaMax, Nadam, AMSGrad, AdamW, QHAdam, AggMo,…

Learning rate decay over time





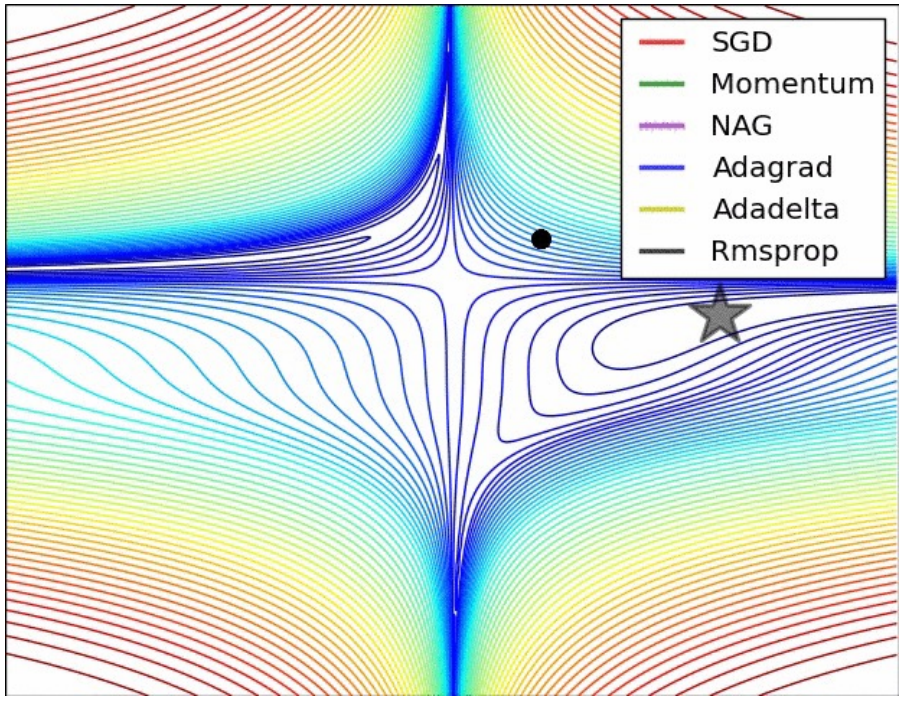
Comparing Gradient-based Optimization Methods



Animation of 5 gradient descent methods on a surface:
GD (cyan), **Momentum** (magenta), **AdaGrad** (white), **RMSProp** (green), **Adam** (blue)

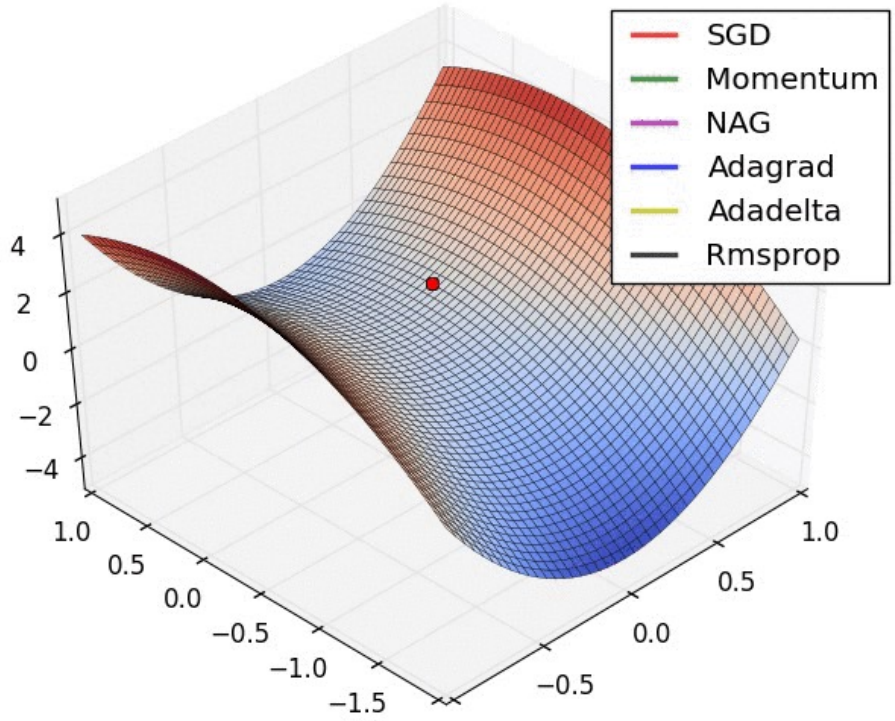
Credit: Lili Jiang

Learning process dynamics



- Contours of a loss surface and time evolution of different optimization algorithms.
- Adagrad, Adadelta, and RMSprop almost immediately head off in the right direction and converge similarly fast,
- Momentum and NAG are led off-track and make optimization look like a ball rolling down the hill (overshooting).

Credit: Alec Radford.& cs231n



- Visualization of a saddle point, where the curvature along different dimensions has different signs.
- SGD, Momentum, and NAG have a very hard time breaking symmetry (SGD gets stuck on the top).
- Adagrad, RMSprop, and Adadelta quickly head down the negative slope.