INF367 25H: Selected Topics in Artificial Intelligence

Diamonds and Rust in the AI Treasure Chest

Dario Garigliotti - UiB - Lecture 4

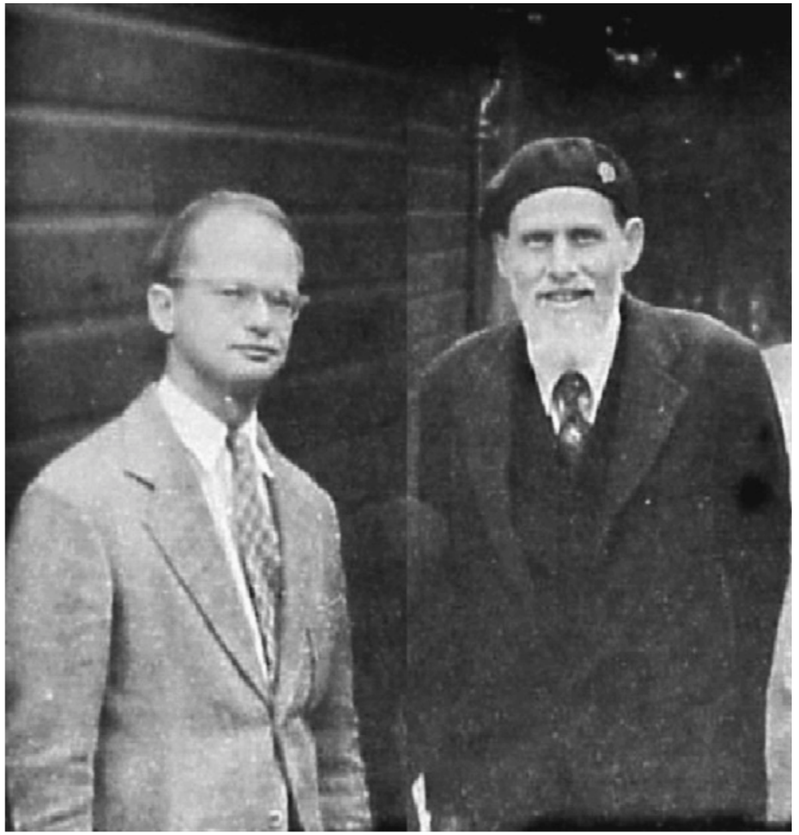
Plan for today

- Admin
- Recap from last lecture

- Primer on optimization
- Activity

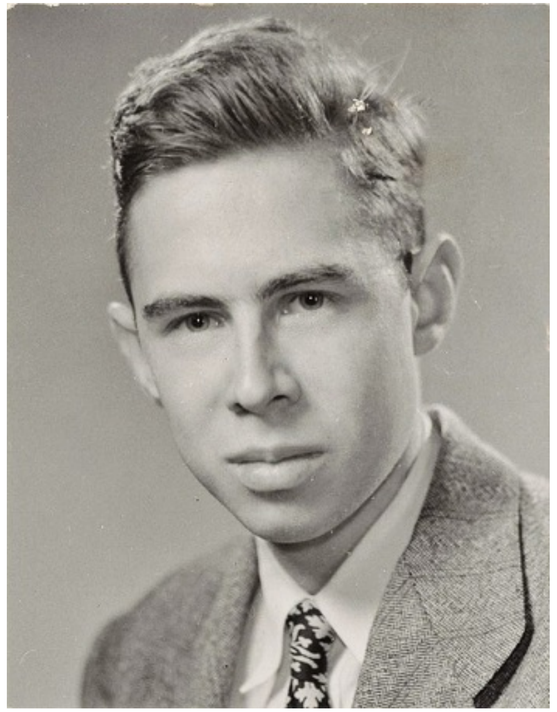
Recap

First artificial neuron



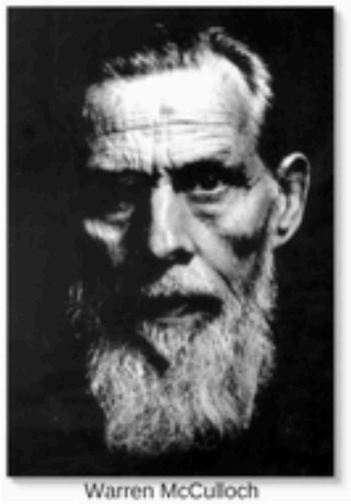
- MCP neuron, introduced by McCulloch and Pitts in 1943
- Context: “all computation reduced to logic”
  - *If brain is a computational device, how does it implement this logic?*
- Perform any computation by mixing basic units
- Unit of computation, without learning

Perceptron



- Introduced by Rosenblatt in 1958
- (Associated hype in press and in academia)
- Why was so groundbreaking?
  - Learns from data
  - Is brain-inspired (combining MCP, Hebbian)
  - Guarantees of existence of solution under conditions
  - Expectations/hype to learn to do every task

Perceptron


Warren McCulloch


Walter Pitts


Donald O. Hebb

- Combines
  - MCP: biologically-inspired unit of computation, extended by networks
  - Hebbian learning: learning due to strengthening neural connections as output of one neuron consistently fires into input of another one
- I.e. basic neuron model with learning algorithm
  - Learned weights “encode”/represent knowledge that is “remembered”

Vectors and vector operations

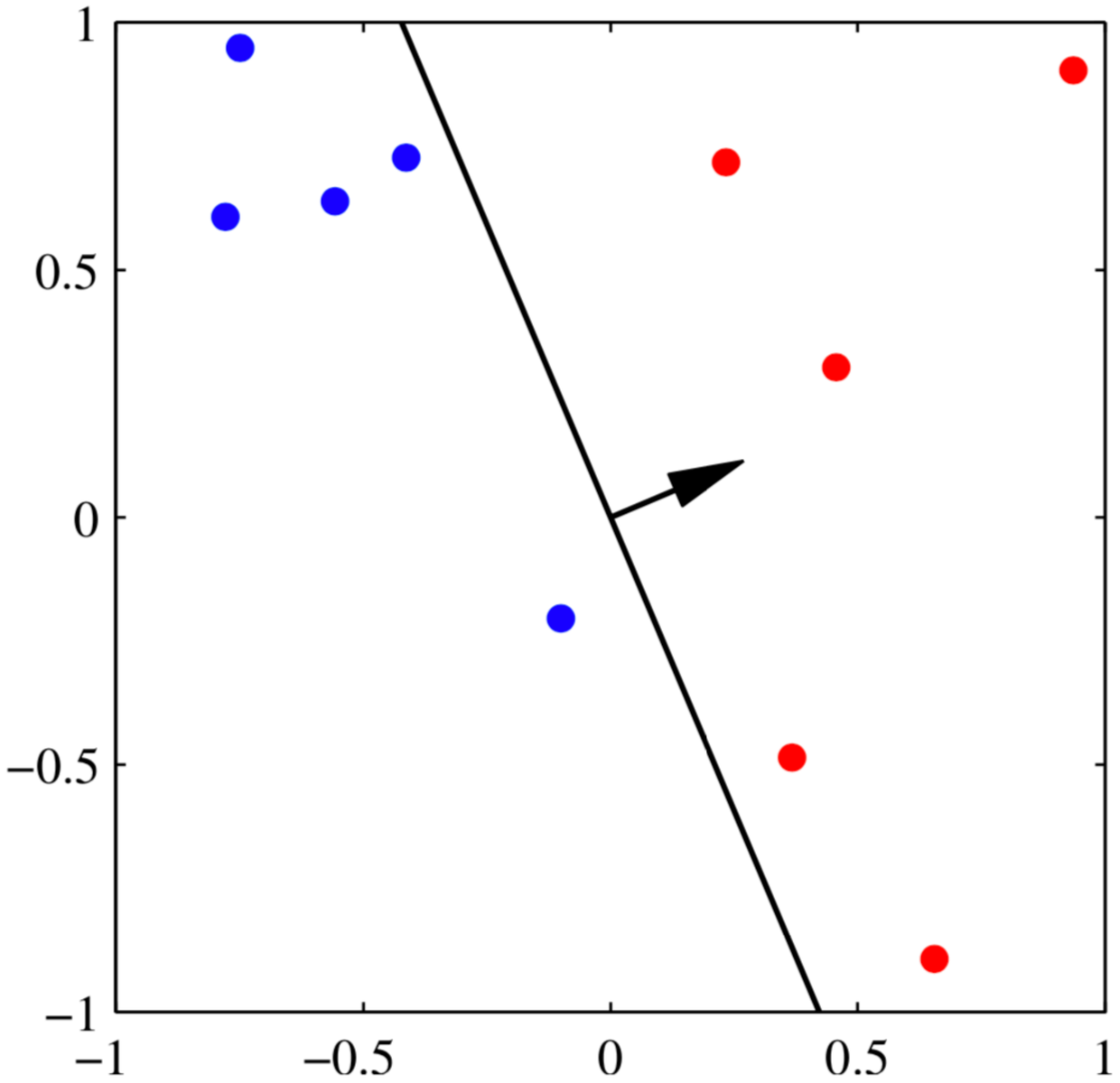
- Vector: magnitude and direction
  - Coordinate notation
  - Magnitude as Euclidean distance
- Operations
  - Product by scalar
  - Sum of vectors
  - Dot product

Perceptron algorithm

- Perceptron finds a hyperplane that separates the linearly separable classes
  - By learning the corresponding weight vector
- Formalization of the perceptron model…
  - using vector notation
  - subsuming the bias as *w0*
- Algorithm for training the model = learning the weights

Perceptron

Example step 4 of 4



Activity: Perceptron and friends

Two possible problems

1. If the two classes are not linearly separable, will the perceptron algorithm converge?
   - After all, it's possible for this Python thing here to terminate with a linear predictor:

```
from sklearn.linear_model import Perceptron; clf = Perceptron(…);
clf.fit(…); clf.predict(…)
```

2. Consider a dataset with 3 possible output classes (e.g. {rainy, cloudy, sunny}).
   - What would it mean for these to be linearly separable?
   - If they are linearly separable, what kind of predictor could you build/learn to separate them?
   - What kind of issues could your efforts present?

Learner vs Predictor/Model

- Some phenomenon *ph* in the real world *RW*
- Some data distribution \cal{Distrib} underlying *ph* according to the true function *f* that “maps input into output” in *RW*
- A dataset *D* ~ \cal{Distrib}, *D* = {(*x_i, y_i*) : *i* = 1..*n*}
- An approximator *h** in *H* = {*h* : *h* approximates *f*} = set of models or predictors
- In relation with *H*, a learner *L* learns *h*’s from *D: L(D)=h*
  - E.g. H1 = {decision trees…}, H2 = {decision forests…}, H3 = {neural nets…}

Infinitesimal Calculus

Calculus and optimization

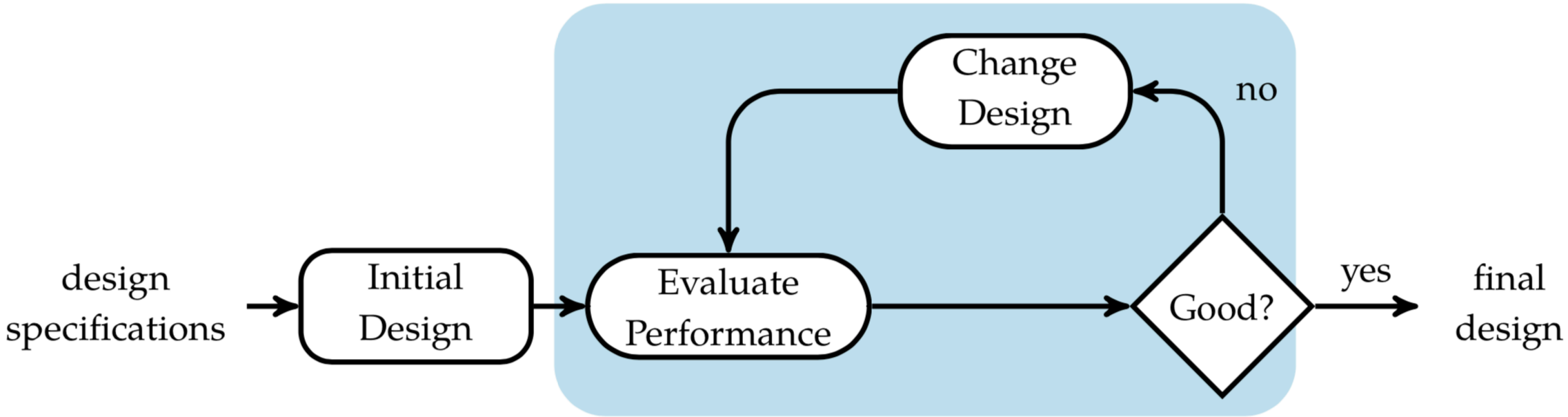
- Derivative
  - Definition
  - Interpretation
- Minimization of functions

- Loss function in supervised machine learning
- All together

Optimization

Optimization

Design Optimization Process



Optimization

- Let's consider a couple of optimization problems
- Problem (1):

$$\begin{aligned} \underset{x_1,x_2}{\text{minimize}} \quad & f(x_1, x_2) \\ \text{subject to} \quad & x_1 \geq 0 \\ & x_2 \geq 0 \\ & x_1 + x_2 \leq 1 \end{aligned}$$

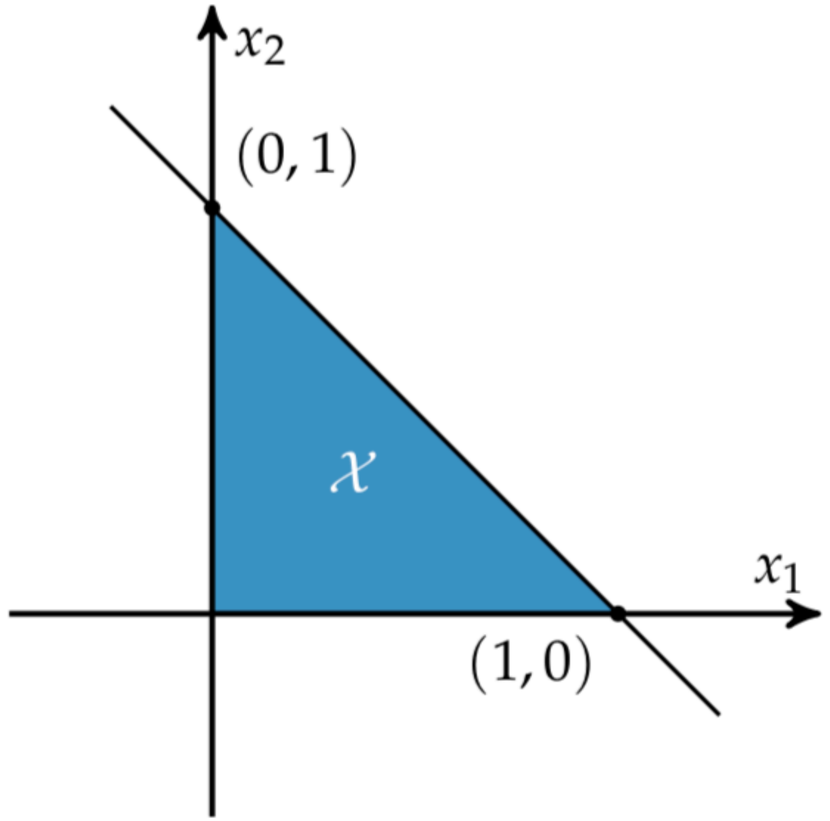


Figure 1.4. The feasible set $\mathcal{X}$ associated with equation (1.5).

Optimization

- Let's consider a couple of optimization problems
- Problem (2):

$$\begin{aligned} \underset{x}{\text{minimize}} \quad & x \\ \text{subject to} \quad & x > 1 \end{aligned}$$

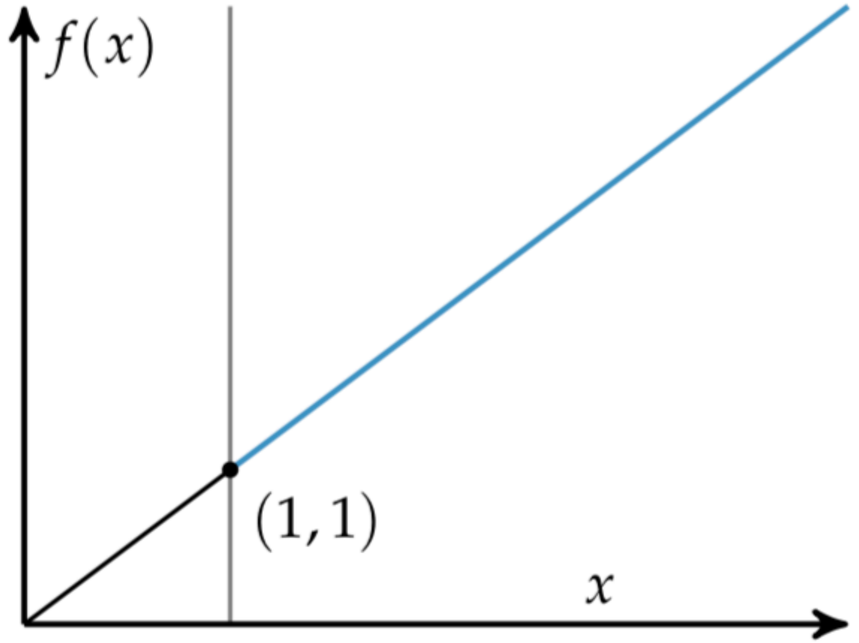


Figure 1.5. The problem in equation (1.6) has no solution because the constraint boundary is not feasible.

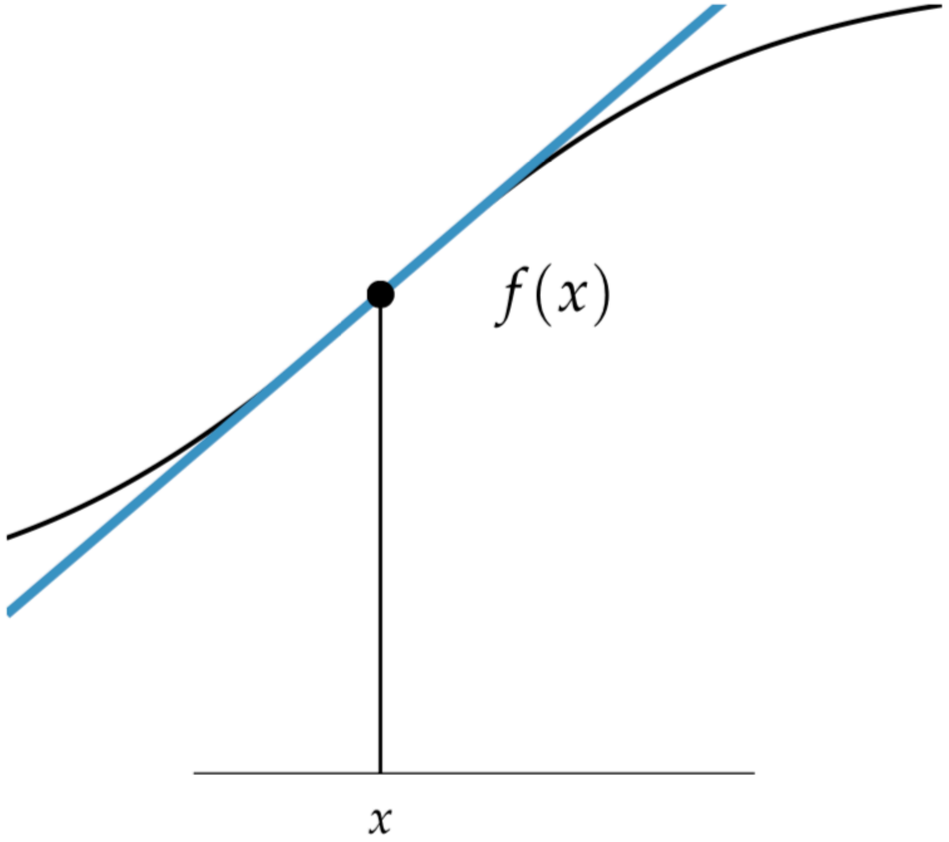
Optimization

Infinitesimal Calculus: Derivative

The *derivative* $f'(x)$ of a function f of a single variable x is the rate at which the value of f changes at x. It is often visualized, as shown in figure 2.1, using the tangent line to the graph of the function at x. The value of the derivative equals the slope of the tangent line.

Optimization

Derivative



Optimization

Derivative

The derivative is the ratio between the change in f and the change in x at the point x:

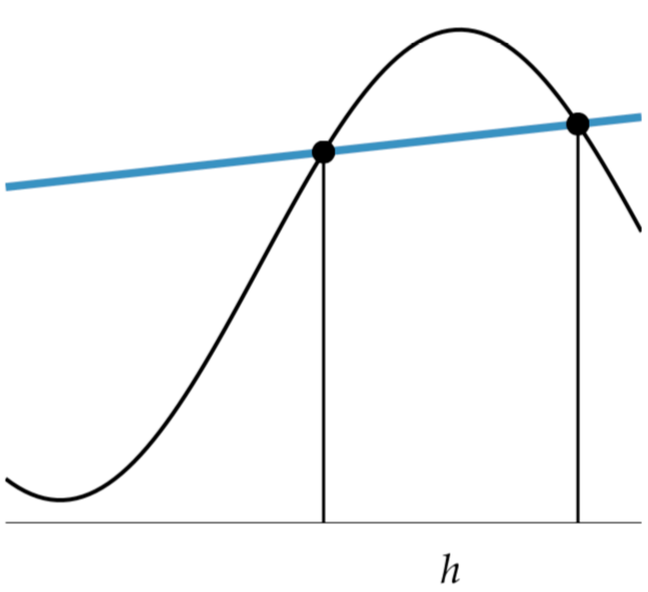
$$f'(x) = \frac{\Delta f(x)}{\Delta x} \tag{2.2}$$

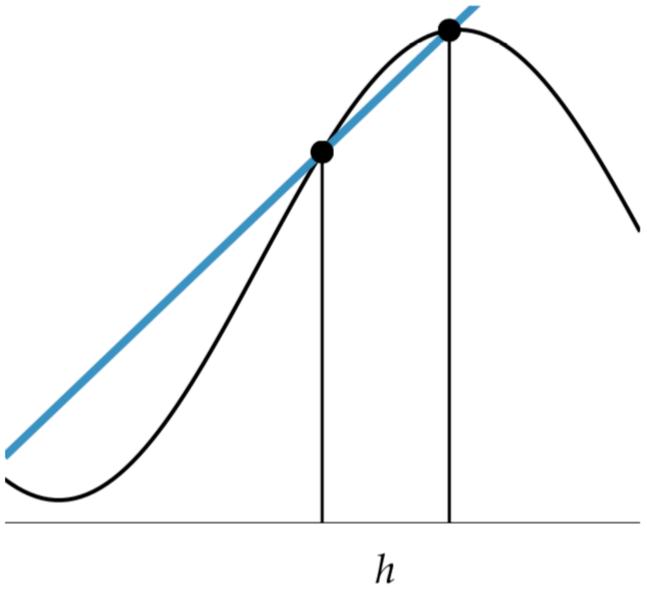
which is the change in $f(x)$ divided by the change in x as the step becomes infinitesimally small as illustrated by figure 2.2.

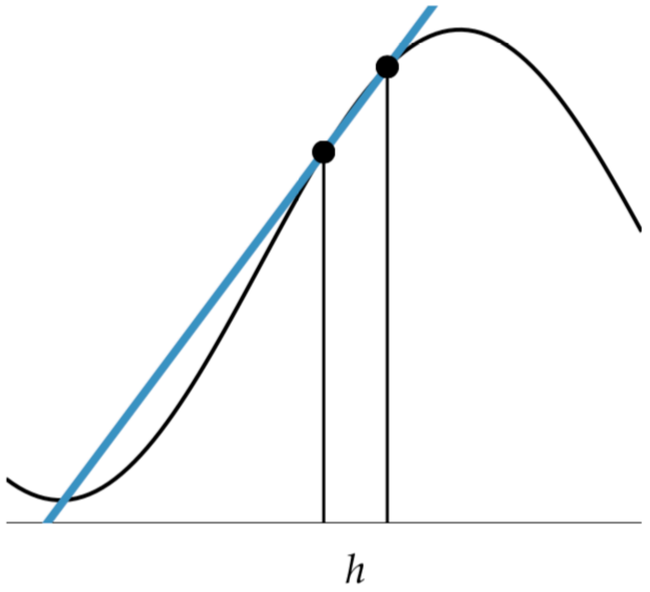
$$f'(x) \equiv \underbrace{\lim_{h \to 0} \frac{f(x+h) - f(x)}{h}}_{\text{forward difference}}$$

Optimization

Derivative as tangent

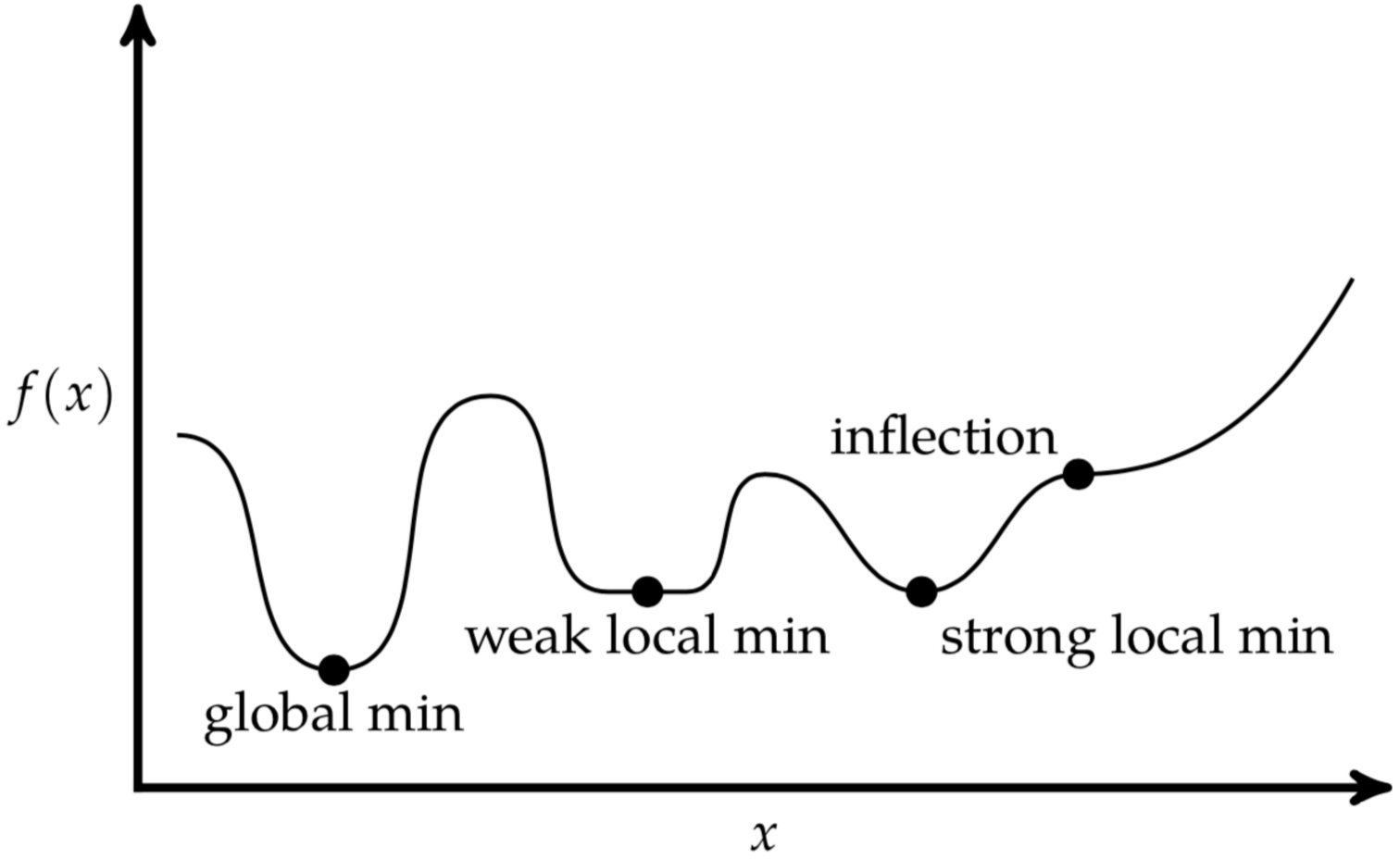






Optimization

Critical points



Optimization

Critical points

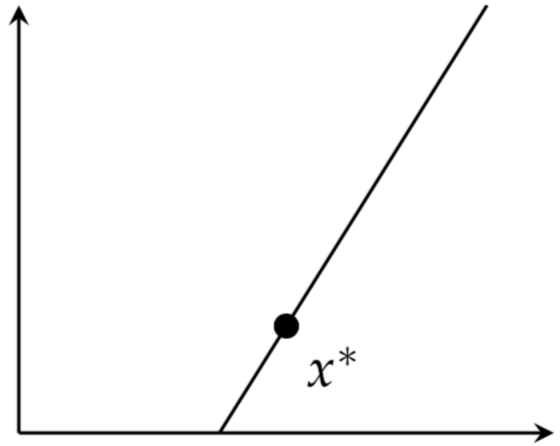
A point can also be at a local minimum if it has a zero derivative and the second derivative is merely nonnegative:

1. $f'(x^*) = 0$, the *first-order necessary condition* (FONC)[19]
2. $f''(x^*) \geq 0$, the *second-order necessary condition* (SONC)

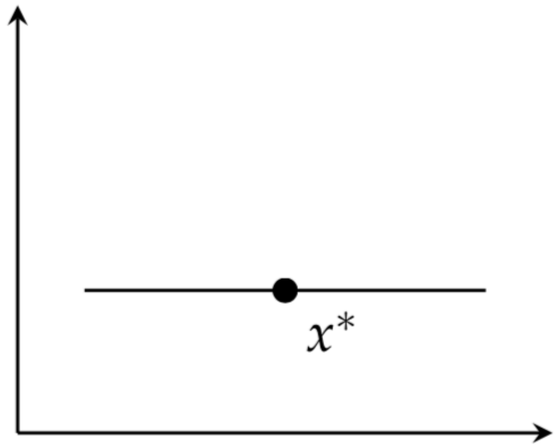
These conditions are referred to as *necessary* because all local minima obey these two rules. Unfortunately, not all points with a zero derivative and a zero second derivative are local minima, as demonstrated in figure 1.7.

Optimization

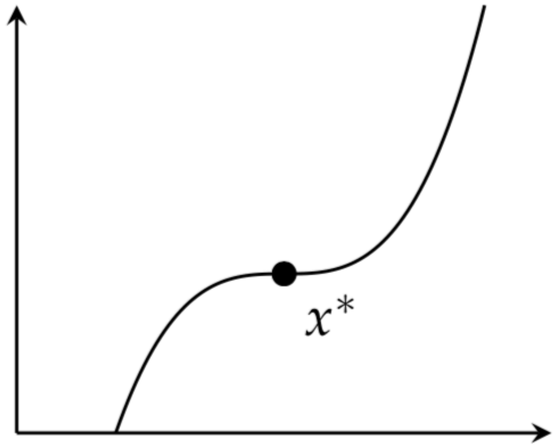
Critical points



SONC but not FONC



FONC and SONC



FONC and SONC

Optimization

Loss

- In supervised machine learning, we will aim to optimize some loss function that compares
  - the predicted outputs *h(x_i)*
  - and desired outputs *y_i*
- for the inputs in a dataset *D* ~ \cal{Distrib}, *D* = {(*x_i, y_i*) : *i* = 1..*n*}
  - E.g. *Loss({(x_i, y_i)})* = average error *h(x_i) - y_i*

Optimization

Gradient

The gradient of f at $\mathbf{x}$ is written $\nabla f(\mathbf{x})$ and is a vector. Each component of that vector is the *partial derivative*[2] of f with respect to that component:

$$\nabla f(\mathbf{x}) = \left[\frac{\partial f(\mathbf{x})}{\partial x_1}, \quad \frac{\partial f(\mathbf{x})}{\partial x_2}, \quad \ldots, \quad \frac{\partial f(\mathbf{x})}{\partial x_n} \right]$$

Optimization

Gradient descent

- Standard approach of finding the roots of the gradient in order to optimize a function *f* is not always (easily) applicable or computable
  - E.g. works by Wiener and Hopf on analog filters
- Gradient descent performs a descent on *f* by the opposite direction of the gradient
  - As *f* is convex and smooth, we go towards the bottom of the multi-dimensional bowl
  - With tiny steps so not to overshoot the minimum

$$\mathbf{w}_{\text{new}} = \mathbf{w}_{\text{old}} + \mu(-\Delta)$$

where:

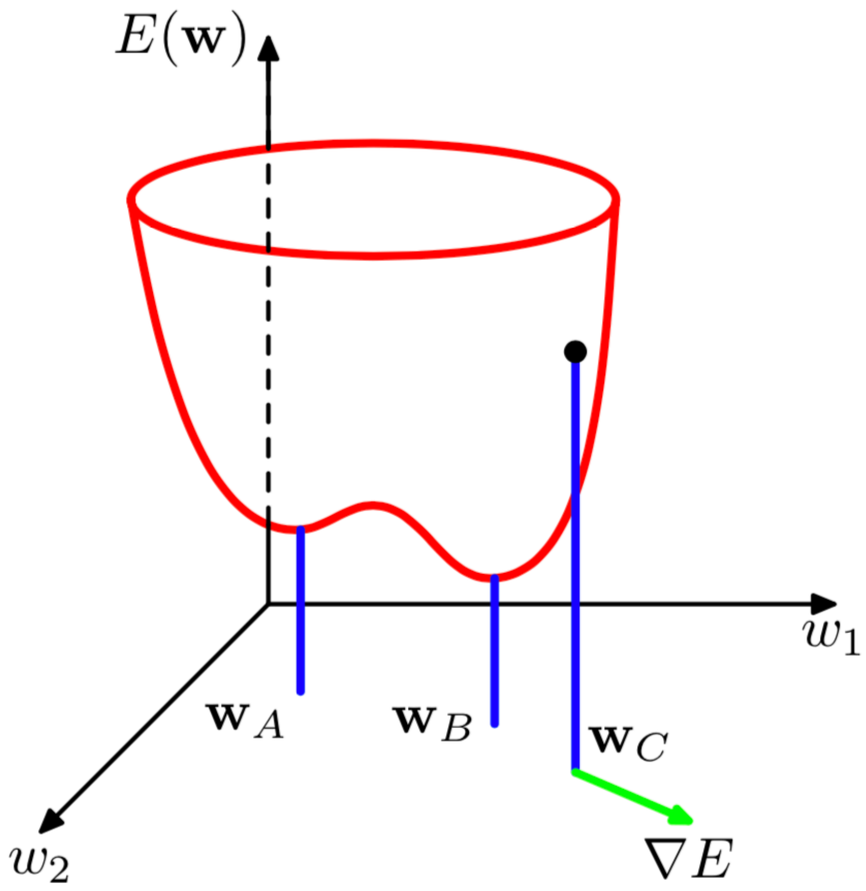
μ = step size

Δ = gradient

Optimization

Gradient descent

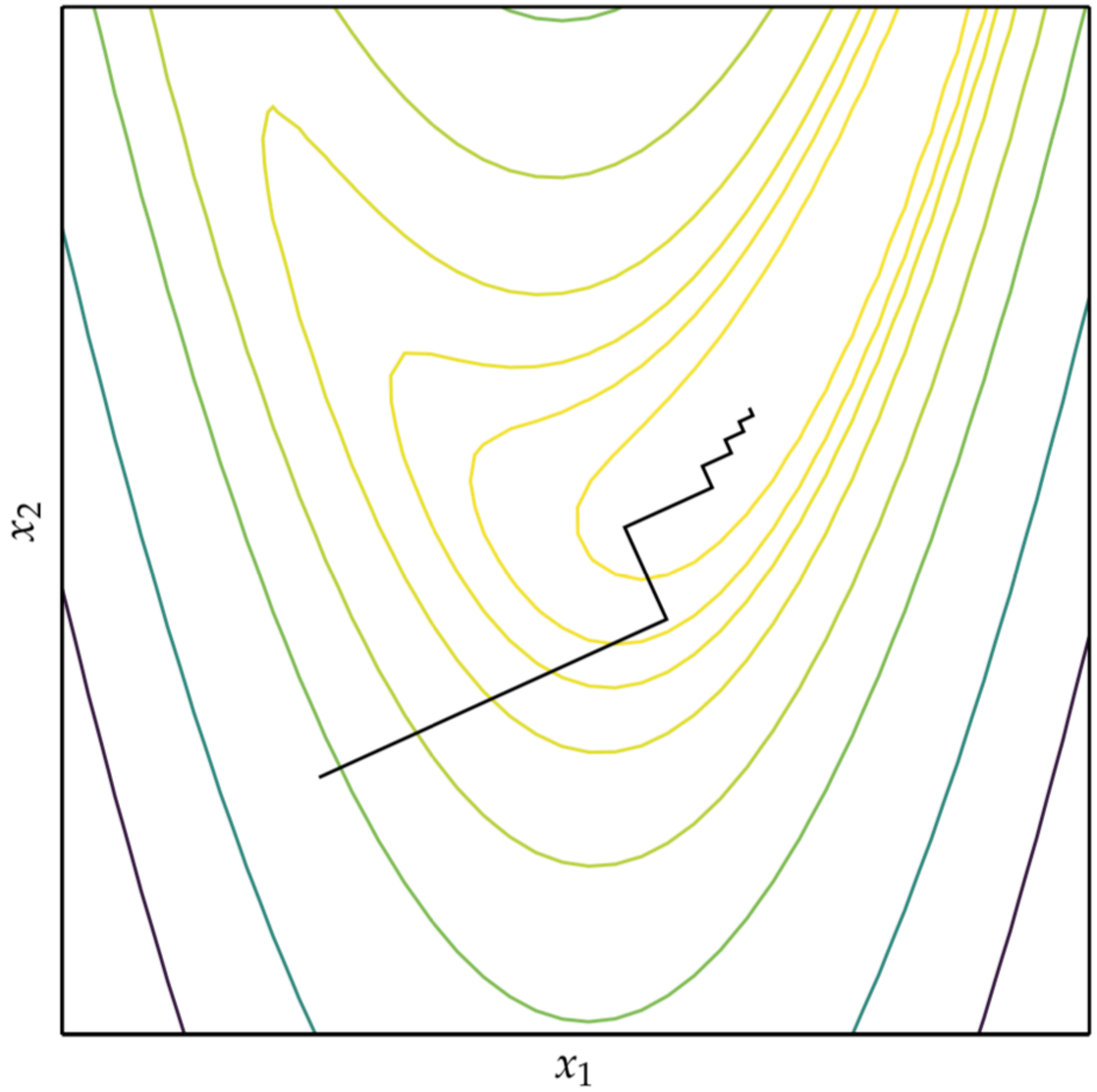
Geometrical view of the error function $E(\mathbf{w})$ as a surface sitting over weight space. Point $\mathbf{w}_A$ is a local minimum and $\mathbf{w}_B$ is the global minimum. At any point $\mathbf{w}_C$, the local gradient of the error surface is given by the vector ∇E.



Optimization

Gradient descent

$$\mathbf{g}^{(k)} = \nabla f(\mathbf{x}^{(k)})$$

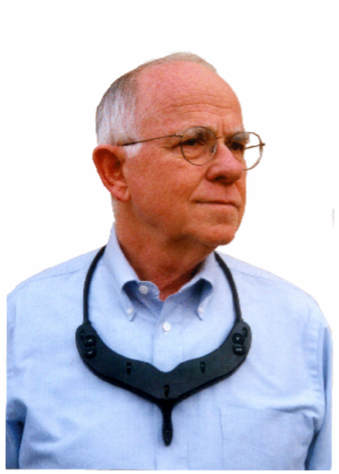


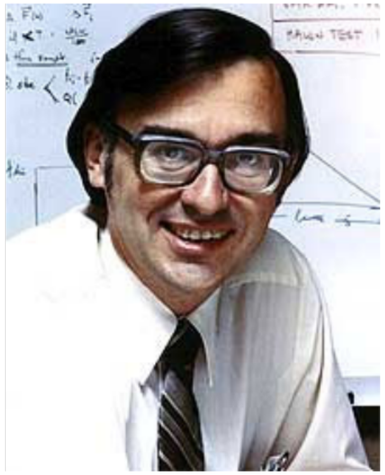
Optimization

Gradient descent

- Stochastic gradient descent (SGD) reduces (drastically) the amount of computations to update the weights while often preserving performance
  - Only a subset of the available training data is used per iteration to update the weights by descent instead of the whole data set
    - So, stochastic: somewhat random (unlike the complete -training- data set which should be good representative of the underlying distribution)
  - Typically: split the training data in ~small disjoint subsets (batches) and use one per iteration
    - Avoiding repeating the batches due to possible overfitting

Optimization
Widrow-Hoff





- Stochastic gradient descent (SGD) takes some inspiration from Widrow and Hoff's Least Mean Squares, ADALINE
  - Developments in the context of (adaptive) filters
  - A proposed approximation to gradient descent: $\mathbf{w}_{\text{new}} = \mathbf{w}_{\text{old}} + 2\mu\varepsilon\mathbf{x}$
    - A bit extremely stochastic as it uses only one data point

Activity

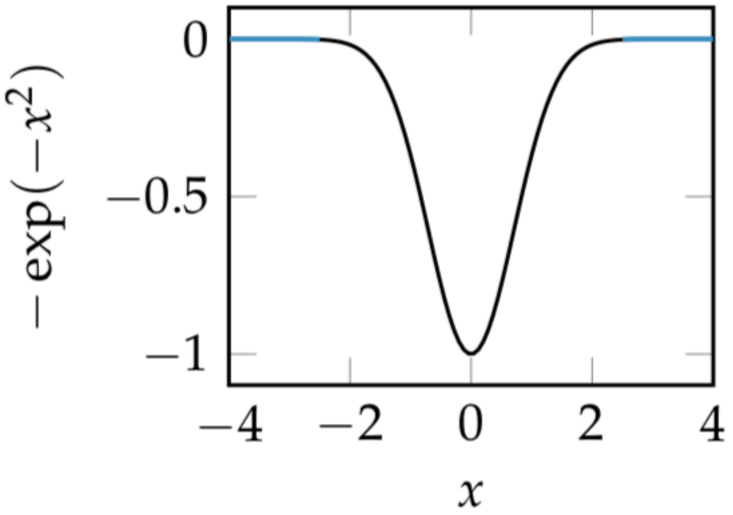
Activity

Gradient descent and friends

- How could you improve gradient descent…
  - … for example, in terms of how fast to reach the bottom?
  - … for example, in terms of how “safe” to reach the bottom, i.e. without going too far?
  - … for example, in terms of how the different multiple dimensions affect how to descend?
- What issues could your efforts arise?

Optimizers
Momentum

- Momentum allows for faster descent, especially in flat regions



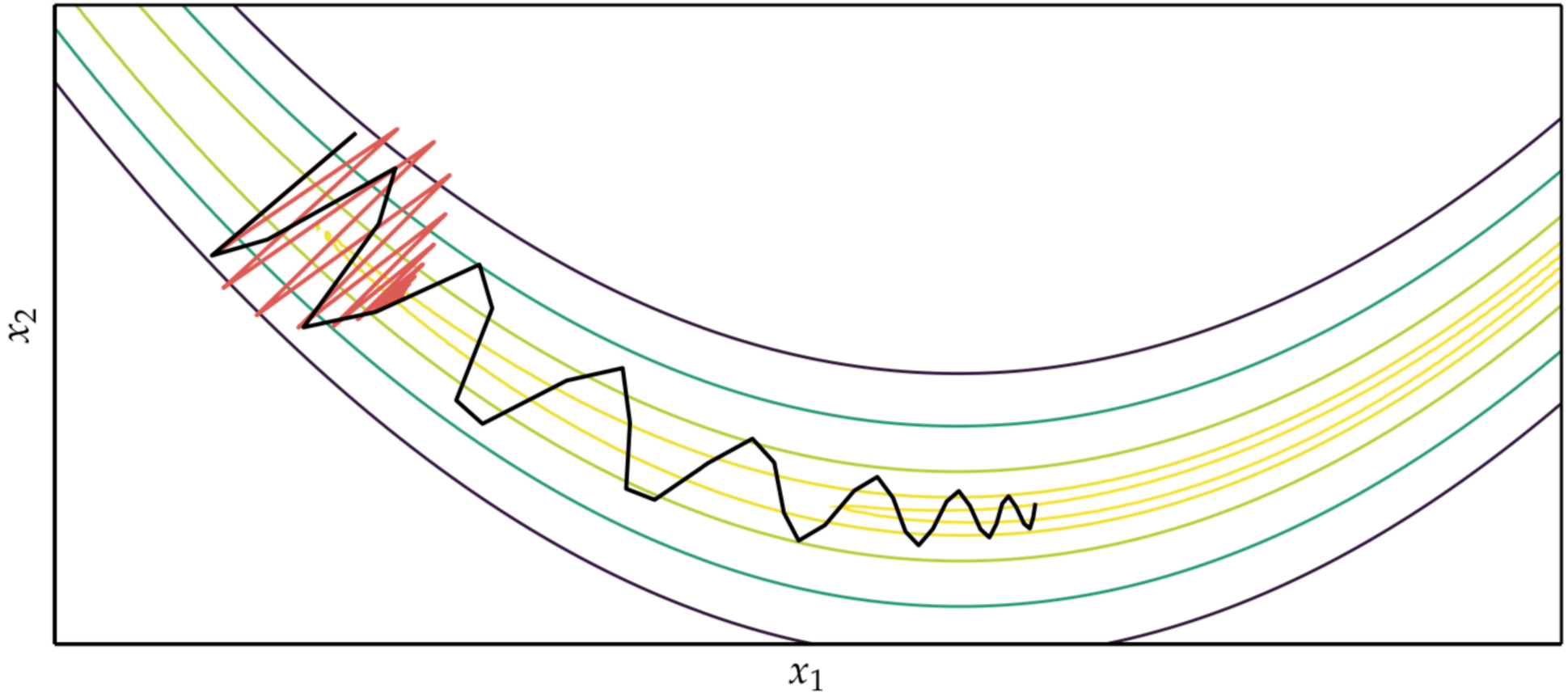
The *momentum* update equations are:

$$\mathbf{v}^{(k+1)} = \beta \mathbf{v}^{(k)} - \alpha \mathbf{g}^{(k)}$$

$$\mathbf{x}^{(k+1)} = \mathbf{x}^{(k)} + \mathbf{v}^{(k+1)}$$

Optimizers

Momentum vs standard gradient descent



Optimizers

Nesterov momentum

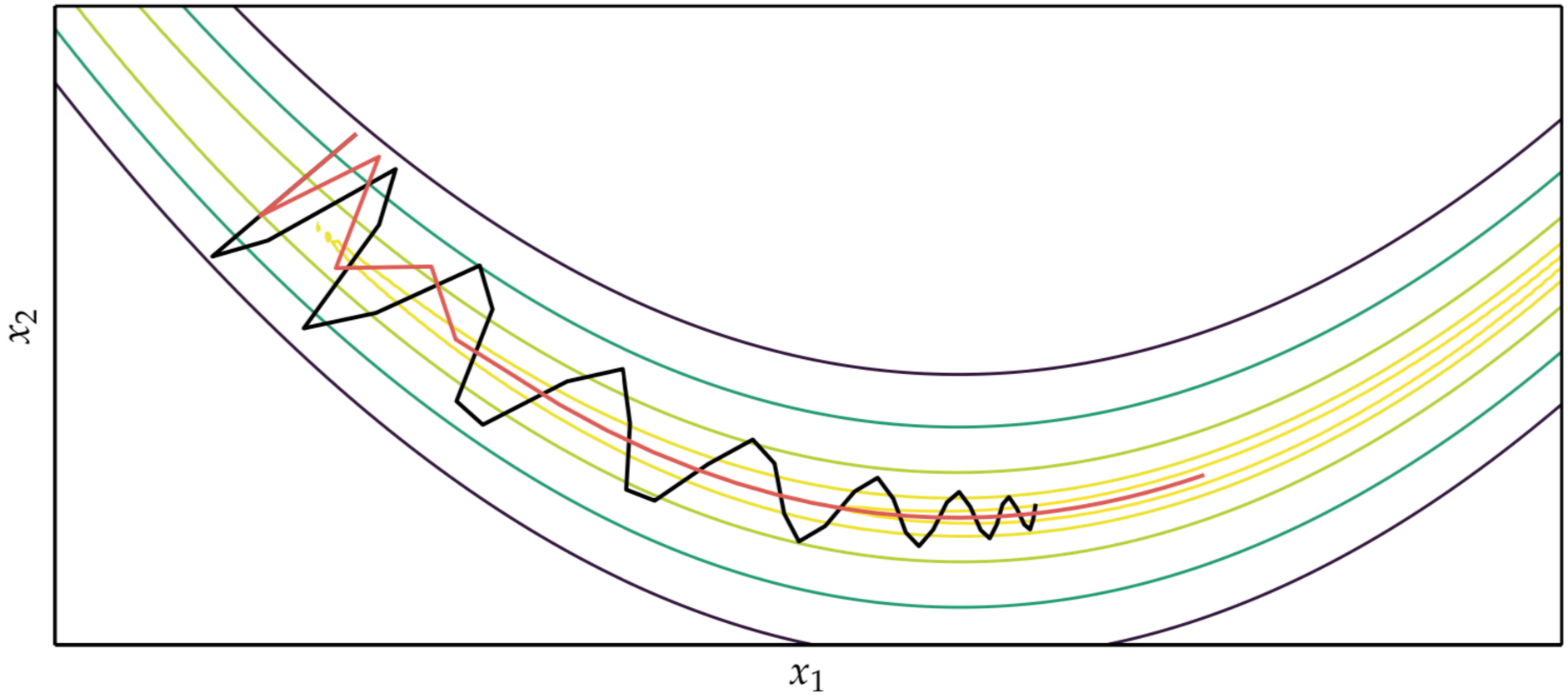
- Momentum does not slow down enough at the bottom of the valley
- Nesterov momentum uses the gradient at the projected future position

$$\mathbf{v}^{(k+1)} = \beta \mathbf{v}^{(k)} - \alpha \nabla f(\mathbf{x}^{(k)} + \beta \mathbf{v}^{(k)})$$

$$\mathbf{x}^{(k+1)} = \mathbf{x}^{(k)} + \mathbf{v}^{(k+1)}$$

Optimizers

Nesterov momentum vs momentum



Optimizers

Nesterov momentum, 1983

Докл. Акад. Наук СССР
Том 269 (1983), № 3

Soviet Math. Dokl.
Vol. 27 (1983), No. 2

A METHOD OF SOLVING A CONVEX PROGRAMMING PROBLEM WITH CONVERGENCE RATE $O(1/k^2)$

UDC 51

YU. E. NESTEROV

Optimizers

Adagrad

- Previous momentum techniques update all components of x with same learning rate
- Adaptive subgradient (Adagrad) adapts a learning rate for each component
  - Dulling components with consistently high gradients
  - And so increasing the influence of infrequent updates

Optimizers

Adagrad

The Adagrad update step is:

$$x_i^{(k+1)} = x_i^{(k)} - \frac{\alpha}{\epsilon + \sqrt{s_i^{(k)}}} g_i^{(k)}$$

where $\mathbf{s}^{(k)}$ is a vector whose ith entry is the sum of the squares with respect to x_i, up to time step k,

$$s_i^{(k)} = \sum_{j=1}^{k} \left(g_i^{(j)} \right)^2$$

Optimizers

Adagrad

- Pros: good for sparse gradients

- Cons: learning rate becomes too small before convergence

Optimizers

Adagrad, 2011

Journal of Machine Learning Research 12 (2011) 2121-2159 Submitted 3/10; Revised 3/11; Published 7/11

Adaptive Subgradient Methods for Online Learning and Stochastic Optimization*

John Duchi JDUCHI@CS.BERKELEY.EDU
Computer Science Division
University of California, Berkeley
Berkeley, CA 94720 USA

Elad Hazan EHAZAN@IE.TECHNION.AC.IL
Technion - Israel Institute of Technology
Technion City
Haifa, 32000, Israel

Yoram Singer SINGER@GOOGLE.COM
Google
1600 Amphitheatre Parkway
Mountain View, CA 94043 USA

Optimizers

AdaDelta, Adam

- Some methods were introduced to avoid the monotonically decreasing learning rate. For example
  - Adadelta
  - Adam
- They make use of techniques like (exponentially) decaying (squared) gradients

Optimizers

AdaDelta, 2012

ADADELTA: AN ADAPTIVE LEARNING RATE METHOD

Matthew D. Zeiler[1,2*]

[1]Google Inc., USA [2]New York University, USA

ABSTRACT

We present a novel per-dimension learning rate method for gradient descent called ADADELTA. The method dynamically adapts over time using only first order information and has minimal computational overhead beyond vanilla stochastic gradient descent. The method requires no manual tuning of a learning rate and appears robust to noisy gradient information, different model architecture choices, various data modal-

Setting the learning rate typically involves a tuning procedure in which the highest possible learning rate is chosen by hand. Choosing higher than this rate can cause the system to diverge in terms of the objective function, and choosing this rate too low results in slow learning. Determining a good learning rate becomes more of an art than science for many problems.

This work attempts to alleviate the task of choosing a learning rate by introducing a new dynamic learning rate that

Optimizers

Adam, 2015

Published as a conference paper at ICLR 2015

ADAM: A METHOD FOR STOCHASTIC OPTIMIZATION

Diederik P. Kingma*
University of Amsterdam, OpenAI
dpkingma@openai.com

Jimmy Lei Ba*
University of Toronto
jimmy@psi.utoronto.ca