Gradient Descent Algorithms

Mingjun Zhong
Department of Computing Science
University of Aberdeen

Topics

- The main Gradient Descent algorithm
- The Stochastic Gradient Descent algorithm

Gradient Descent

- The main algorithm used in Machine Learning
- GD is used in all Deep Learning algorithms
- Machine learning frameworks can ***automatically implement GD algorithm***
- They use ***Automatic Differentiation*** (AD) to compute gradients
- <u>Only need to know the concepts of Gradient</u>
- <u>No need to know how to calculate the gradients</u>

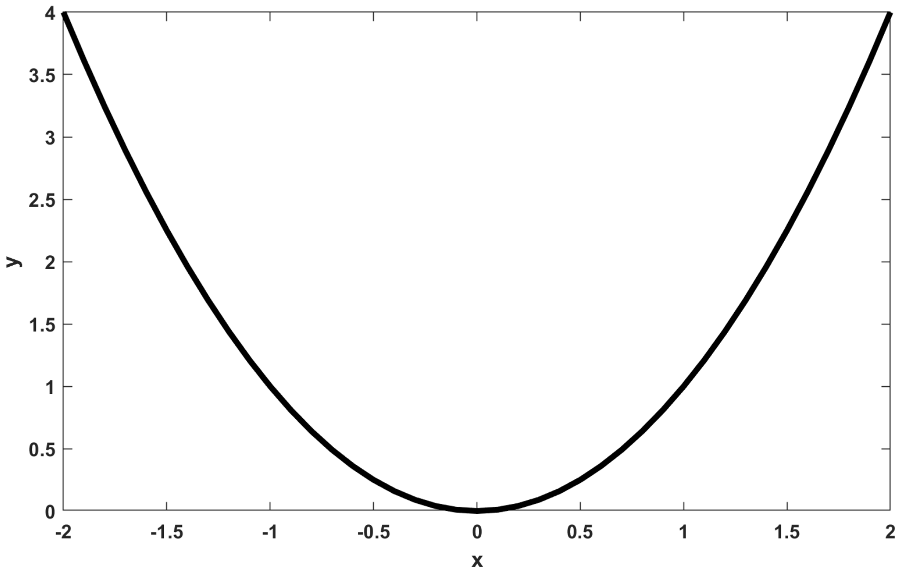
Motivation

- Gradient descent is used to find the local or global optima (maxima/minima) of a function:

$$y = f(x)$$

- Models are represented as functions in Machine Learning.
- Gradient descent used for optimizing models in ML:
  - Linear regression
  - Logistic regression
  - (Deep) neural networks
  - Back-propagation algorithms

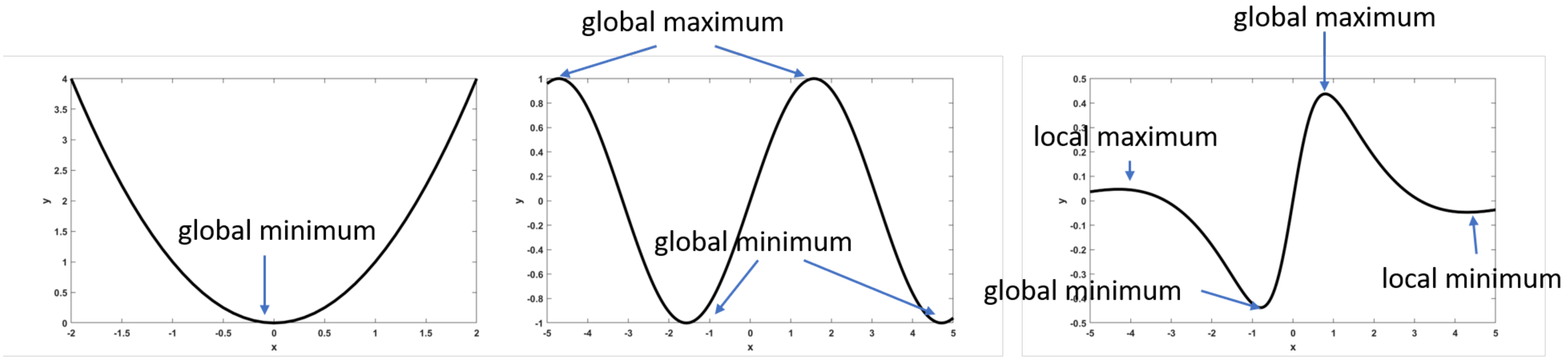
$y = f(x)$



Examples of Functions

Function	Input	Output
$y = x^2$	$x \in (-\infty, +\infty)$	$y \in [0, \infty)$
$y = \sin(x)$	$x \in (-\infty, \infty)$	$\mathrm{y} \in [-1,1]$

Graphs of functions



$y = x^2$

(a)

$y = \sin(x)$

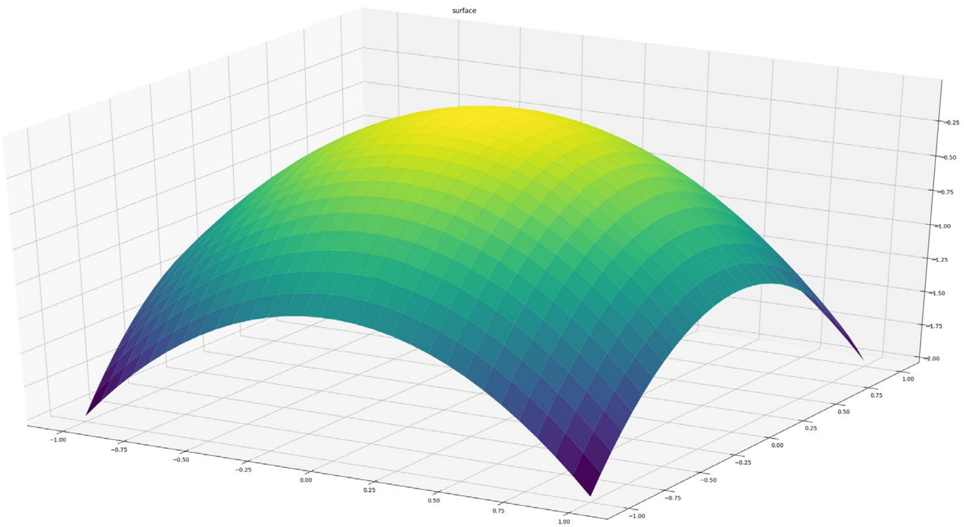
(b)

$$y = \frac{\sin(x)}{1 + x^2}$$

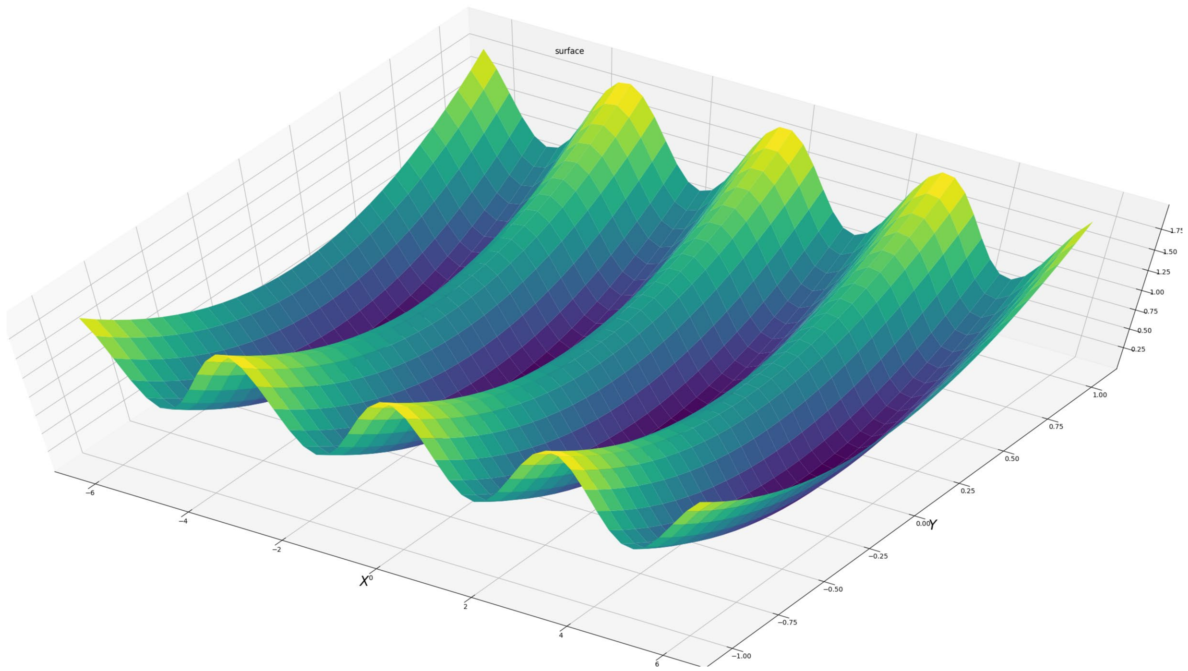
(c)

Graphs of functions

- Examples of multivariate functions:



$$f(x, y) = -x^2 - y^2$$



$$f(x, y) = \cos^2(x) + y^2$$

How to compute the gradient of a function?

- For univariate function $f(x)$, the **gradient** is the **derivative** of $f(x)$:

$$\frac{df(x)}{dx}$$

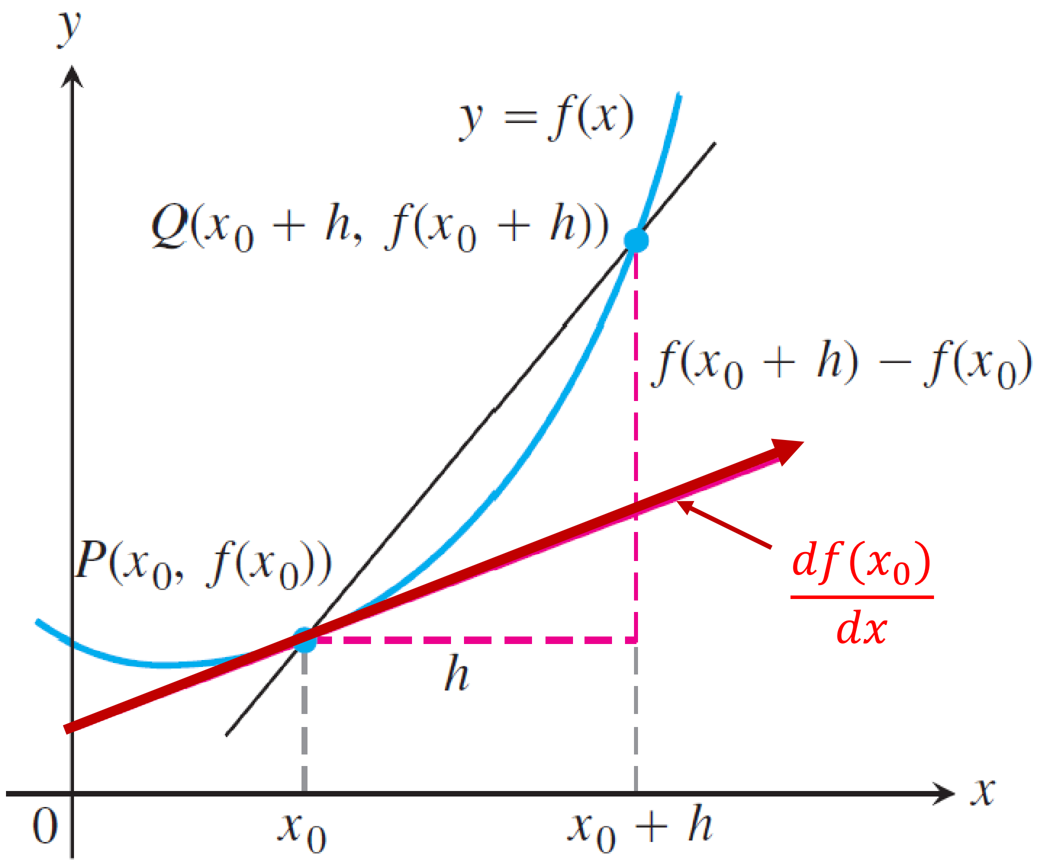
- For multivariate function $f(x_1, x_2)$, the **gradient** is **the vector of the derivatives** along each variable:

$$\left[\frac{df(x_1, x_2)}{dx_1}, \frac{df(x_1, x_2)}{dx_2}\right]$$

The **derivative** of a function is the **slope of the curve** $y = f(x)$ at the point $P_0 = (x_0, f(x_0))$ is the number:

$$m = \lim_{h \to 0} \frac{f(x_0 + h) - f(x_0)}{h}$$

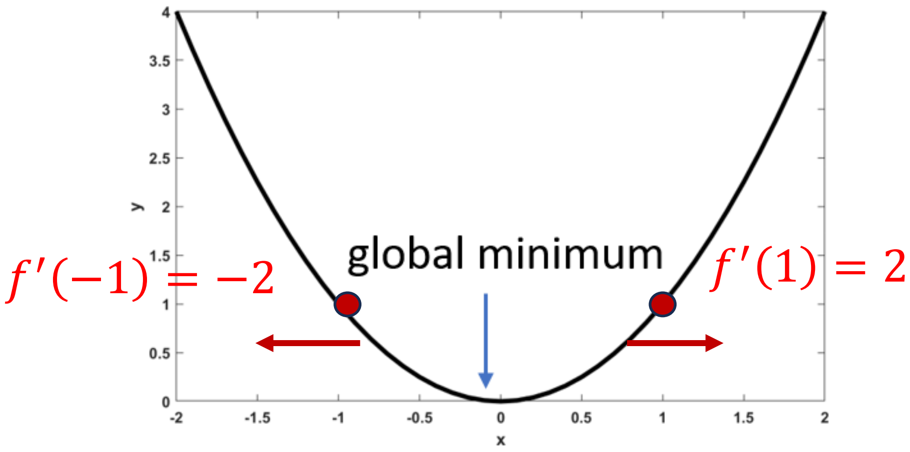
The **tangent line** to the curve at P_0 is the line through P_0 with this slope.
We call this limit, when it existed, the derivative of f at x_0.



Example

- The function $f(x) = x^2$
- $f'(x) = \frac{\mathrm{df(x)}}{\mathrm{d}x} = 2x$
- At $x = -1$, $f'(-1) = -2$, which is a **negative** value.
- At $x = 1$, $f'(1) = 2$, which is a **positive** value.

- Gradient points to the direction of increasing the value of $f(x)$

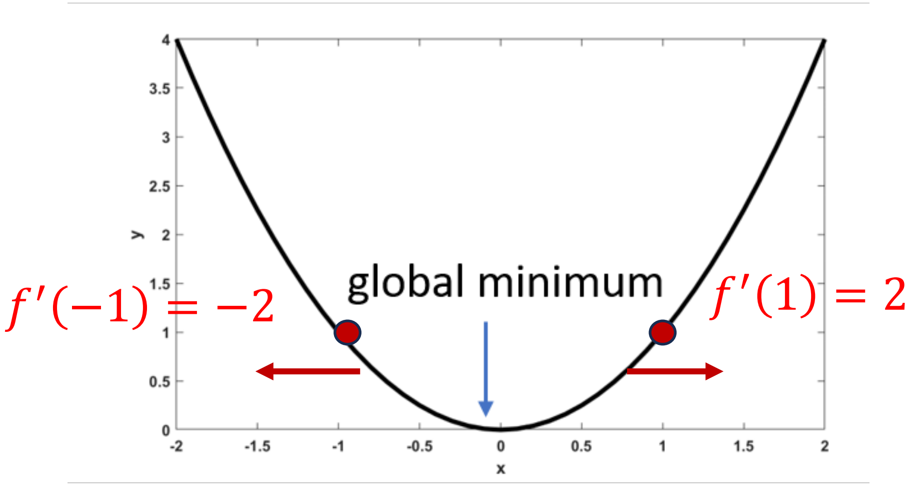


$y = x^2$

Example

- The function $f(x) = x^2$
- $f'(x) = \frac{\mathrm{df(x)}}{\mathrm{d}x} = 2x$
- At $x = -1$, $f'(-1) = -2$, which is a **negative** value.
- At $x = 1$, $f'(1) = 2$, which is a **positive** value.

- Gradient points to the direction of increasing the value of $f(x)$



$y = x^2$

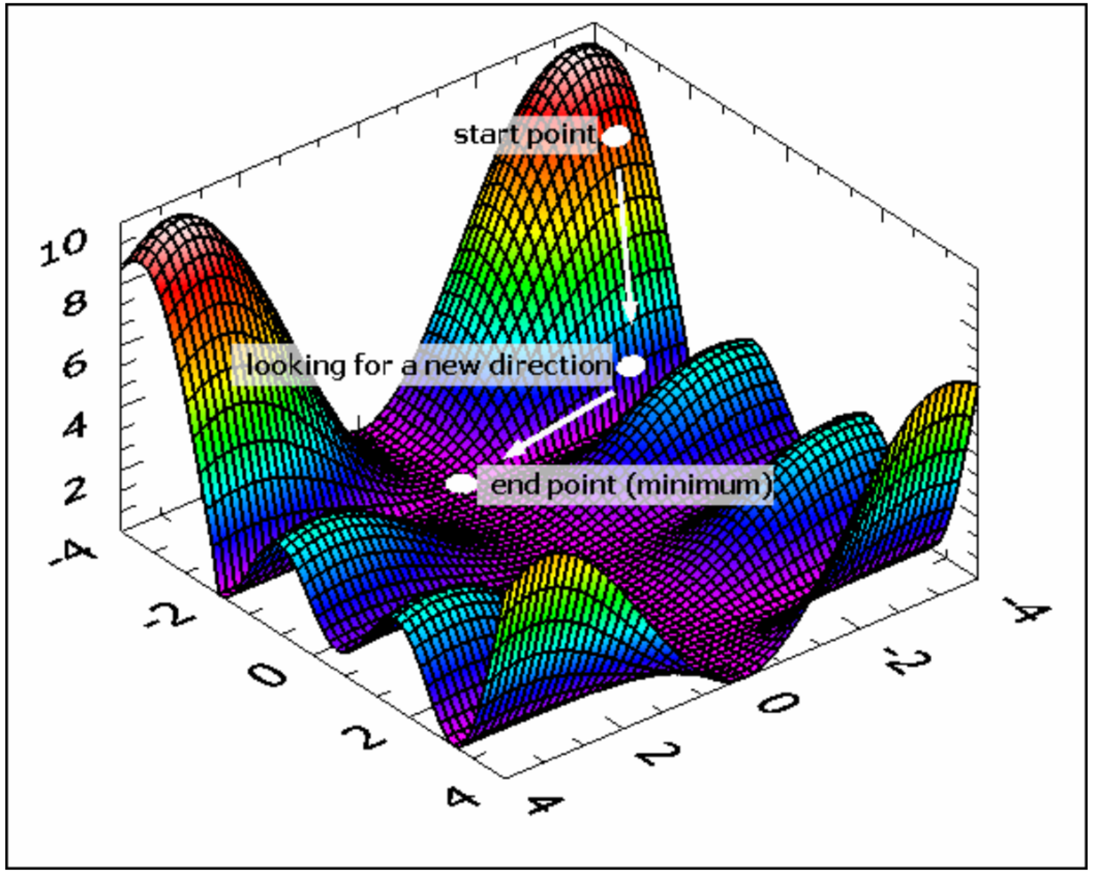
- As gradient point to increasing the value of $f(x)$, we should change x to the negative gradient direction
- For example, if we are at $x_0 = -1$, we can move a little along the negative gradient:

$$x_1 = x_0 - 0.01 * f'(-1) = -1 - 0.01 * (-2) = -0.98$$

- So $f(x_0) > f(x_1)$
- This is the idea of Gradient Descent algorithm

Gradient Descent Algorithm

- To find a minimum of a function $f(x)$
- Idea:
  1. Start with some ``initial guess'' for x;
  2. Repeatedly change x to make $f(x)$ smaller;
  3. Stop when a stopping condition is satisfied.



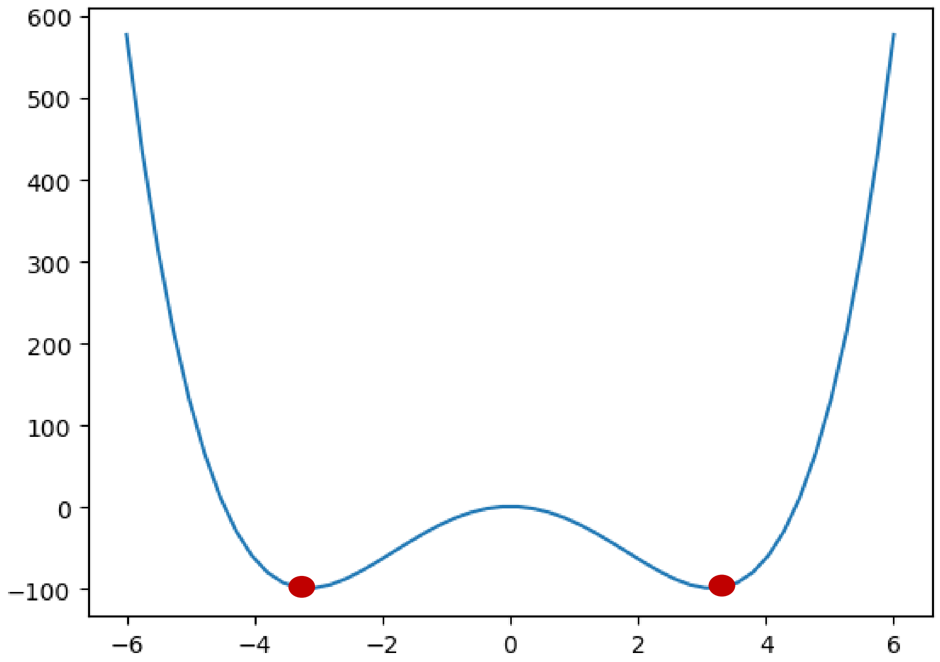
Gradient Descent Algorithm

- To find a minimum of a function $f(x)$
- Idea:
  1. Start with some ``initial guess'' for x, i.e., $x^{(0)}$;
  2. Repeatedly change x to make $f(x)$ smaller;
  3. Stop when a stopping condition is satisfied.
- Notations:
  - $\epsilon > 0$ (e.g., 0.0001): the criterion to stop updating
  - $\alpha > 0$ (e.g., 0.01): the step size for updating – how fast we want to update x.

- <u>Gradient descent algorithm</u>:
  1. Given $x^{(0)}$, ϵ, α and gradient $\frac{df(x)}{dx}$
  2. Repeat the following step, until $|x^{(k+1)} - x^{(k)}| \leq \epsilon$:

$$x^{(k+1)} = x^{(k)} - \alpha \frac{df(x^{(k)})}{dx}$$

Gradient descent algorithm (Python example) - 0



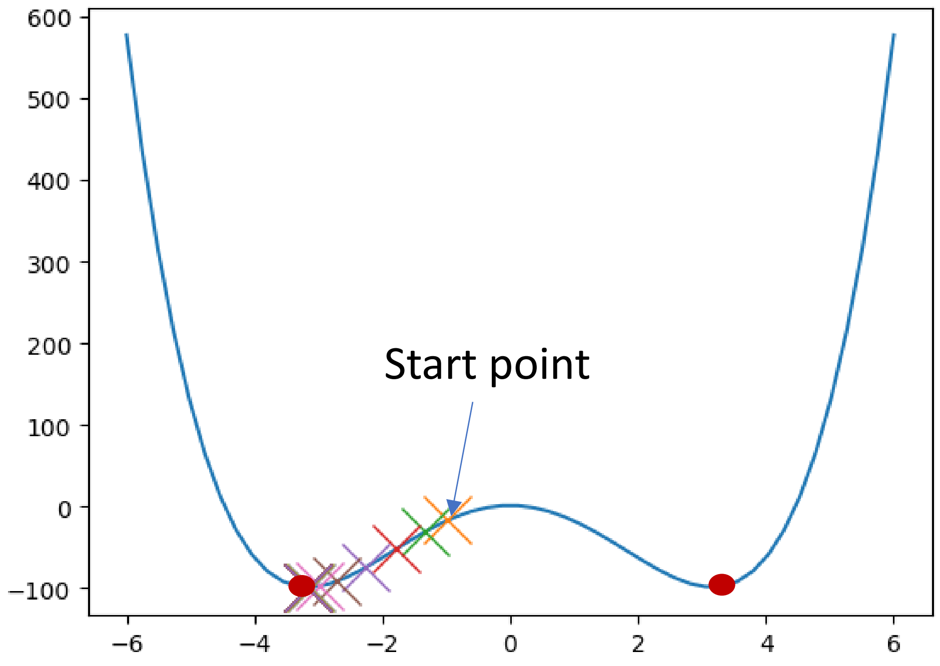
$$f(x) = x^4 - 20x^2 + 1$$

Gradient descent algorithm (Python example) - 1

- import numpy as np
- # find minimum of function f(x)=x^4-20x^2+1
- x_old = np.random.randn()
- x_new = np.random.randn()
- epsilon = 0.00001
- alpha = 0.001
- print(x_new)

- # the gradient function
- def df(x):
-     return 4*x**3 - 40*x

- while abs(x_new-x_old)>epsilon:
-     x_old = x_new
-     x_new = x_old - alpha*df(x_old)
-     print('x_new={}'.format(x_new))
-
- print("Local minimum occurs at {}".format(x_new))



$$f(x) = x^4 - 20x^2 + 1$$

$$f'(x) = 4x^3 - 40x$$

Machine Learning algorithms

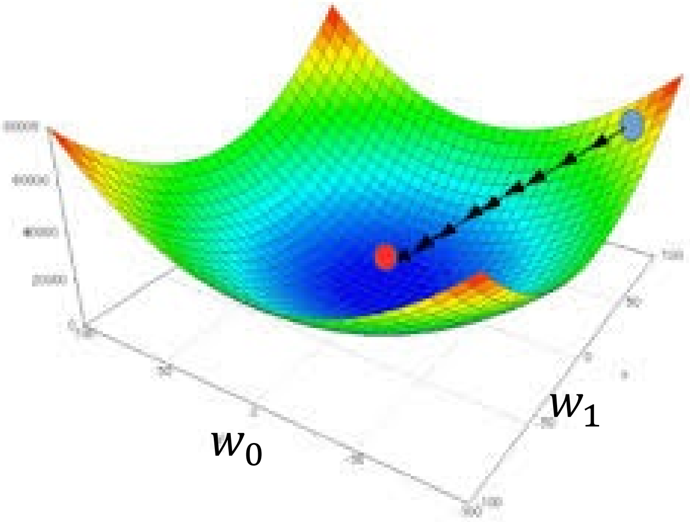
- Define a ***loss function*** $f(\theta)$ in ML (e.g., the SSE in Linear Regression)
- Require evaluation of derivatives (gradient) of the loss function
- Then minimize the loss function with respect to θ:
$$\min_{\theta} f(\theta)$$
- The notation means finding the minimum value of $f(\theta)$ with respect to θ.

Loss function: Linear regression model

- Loss function:

$$f(w_0, w_1) = \sum_{n=1}^{N} (y_n - w_0 - w_1 x_n)^2$$

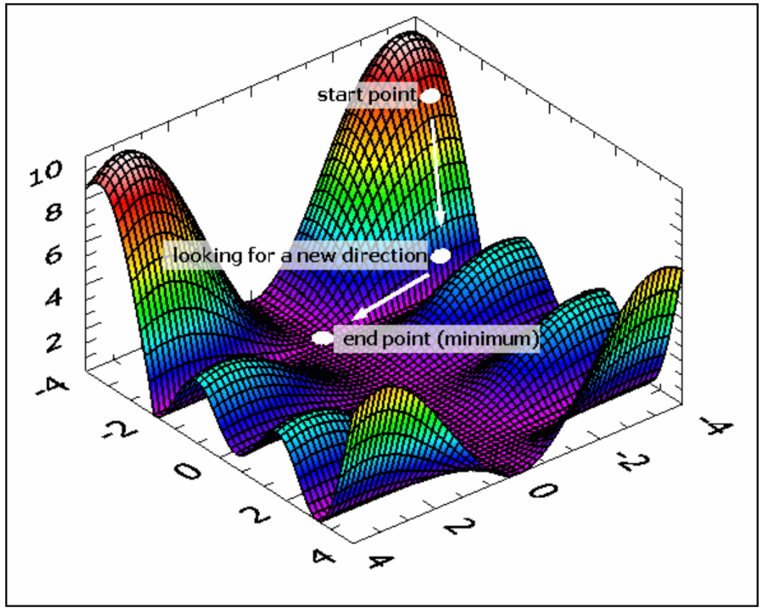
- w_0 and w_1 are the weights to be optimized
- $f(w_0, w_1)$ is a ***convex*** function, which has a unique minimum function value
- GD is guaranteed to find the minimum value



The graph of a *convex* function

Loss function: Deep learning model

- Loss function:
$$f(w) = complex\ \ form$$
- w are the weights to be optimized
- $f(w)$ is a ***non-convex*** function, which has many local minimum function values
- GD could only find one local minimum value



Methods for computing derivatives

1. Computing derivatives is the key for ML modelling
2. How to compute derivatives?
3. Manually working out derivatives and programming them
4. Numerical differentiation using finite difference approximations
5. Symbolic differentiation using expression manipulation in computer algebra: Mathematica, Maxima, & Maple
6. ***Automatic differentiation*** – algorithmic differentiation

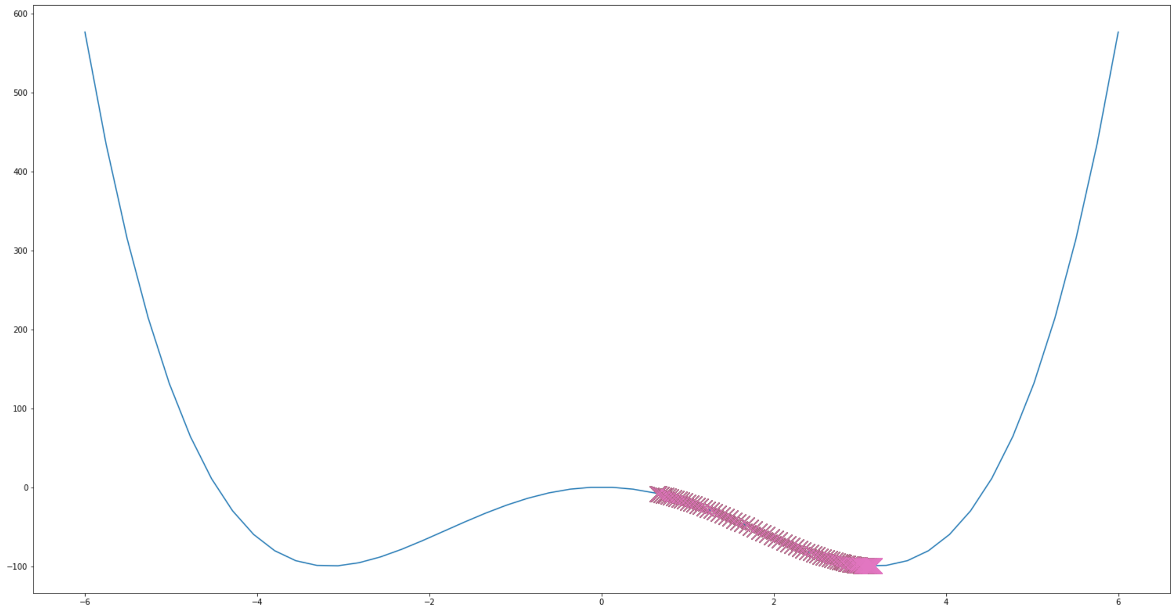
Automatic Differentiation

- The key is to compute derivatives for using GD
- Automatic Differentiation (AD) can automatically compute derivatives (Baydin, et al., 2018).
- Deep learning tools has AD: ***TensorFlow, Pytorch,*** etc.
- You don’t worry about computing gradients

Baydin, et al., 2018. Automatic differentiation in machine learning: a survey. JMLR 18: 1-43

Automatic differentiation in TensorFlow

Please see tutorials
https://www.tensorflow.org/guide/autodiff



$$f(x) = x^4 - 20x^2 + 1$$

Stochastic Gradient Descent

Motivation of Stochastic Gradient Descent (SGD)

- The workhorse of Machine Learning
- ML applications analyse big data
  - - huge number of data samples
  - - need ML able to *efficiently* deal with big data

Example: $f(\theta, x_i) = y_i - \theta^T x_i$

$y_1 = £495; x_1 = (1, 3060, 3)^T$

$$\sum_{n=1}^{2} f(\theta, x_i) = (y_1 - \theta^T x_1) + (y_2 - \theta^T x_2)$$

Gradient Descent

- Much of ML algorithms to optimize the problem

$$min_\theta \frac{1}{N} \sum_{n=1}^{N} f(\theta, x_i)$$

  - where x_i is the i^{th} data point; θ is the parameters to optimize
  - the loss is the average over the loss function of all data samples

- Example: linear regression, logistic regression, neural networks.
- We can use Gradient Descent: compute the gradient, then update.

$$\theta^{t+1} = \theta^t - \alpha_t \frac{1}{N} \sum_{n=1}^{N} \nabla_\theta f(\theta, x_i)$$

- The challenge: N is huge – too expensive to evaluate full gradient.

Stochastic gradient descent

- The ideas to mitigate big data:
  - Standard SGD: use one sample to compute gradient
  - Mini-batch: use a subset of data sample to compute gradient
- SGD:

$$\theta^{t+1} = \theta^{t} - \alpha_t \nabla_\theta f(\theta, x_{i_t})$$

  - Where $i_t \in \{1, \cdots, N\}$ is a randomly chosen index at iteration t.
- SGD is unbiased – supported by the theorem, this is good way to find the best θ.
- The index i_t chosen *without replacement* to complete full cycle over whole data set

Mini-batch SGD

- Instead of using one sample, choose a random subset for gradient
- I_t is a subset of $\{1,2,\cdots,N\}$ with size d:

$$\theta^{t+1} = \theta^{t} - \alpha_t \frac{1}{d} \sum_{i_t \in I_t} \nabla_\theta f(\theta, x_{i_t})$$

- Mini-batch SGD is also unbiased – again, supported by the theorem, this is good way to find the best θ.

SGD vs Mini-batch SGD

- Mini-batch has more cost at each step comparing to SGD
- Smaller batch sizes converge more quickly to a ***less optimal value***
- In ML, we likely do not need global optimal value
- A reasonable optimal value most of time is sufficient for the problem

SGD for linear regression

- Simple linear regression with samples $(x_1, y_1), (x_2, y_2), \cdots, (x_N, y_N)$:

$$f(w_0, w_1) = \frac{1}{N}\sum_{n=1}^{N}(y_n - w_0 - w_1 x_n)^2 = \frac{1}{N}\sum_{n=1}^{N} f_n(w_0, w_1)$$

- Gradient descent ($t = 0, 1, 2, \cdots$) to update w_1:

$$w_1^{t+1} = w_1^t - \alpha_t \frac{1}{N}\sum_{n=1}^{N}\frac{df_n(w_0,w_1)}{dw_1}$$

- Stochastic gradient descent:

$$w_1^{t+1} = w_1^t - \alpha_t \frac{df_n(w_0,w_1)}{dw_1}$$

- Note: $\frac{df_n(w_0,w_1)}{dw_1} = -2x_n(y_n - w_0 - w_1 x_n)$

Computational cost: linear regression

- A *linear regression* problem with n data points, mini-batch size d, and feature dimension p:

$$f(w) = \frac{1}{N}\sum_{i=1}^{N}(y_i - w^T x_i)^2$$

1) $w = \left(w_1, w_2, \cdots, w_p\right)^T; x_i = (x_{i1}, x_{i2}, \cdots, x_{ip})$

- Gradient: $\frac{df(w)}{dw} = -\frac{2}{N}\sum_{i=1}^{N}(y_i - w^T x_i)x_i$ (**the main cost**)
- **flop** is an acronym for floating point operation. Counting the number of flops roughly computes the relative speed of an algorithm.
- The computational cost:

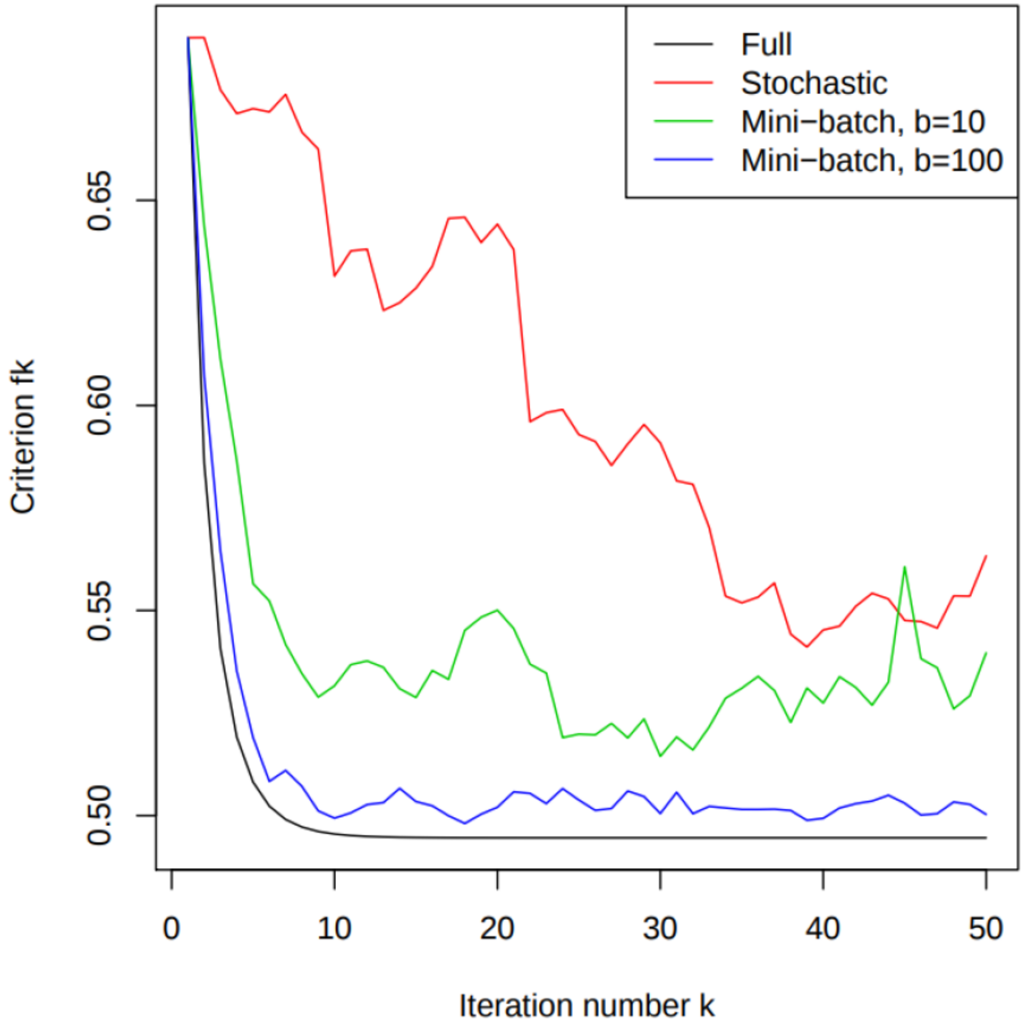
1. Full gradient: $O(Np)$
2. Mini-batch: $O(dp)$
3. Standard SGD: $O(p)$

The flops of full gradient descent:

- To compute $w^T x_i = w_1 x_{i1} + w_2 x_{i2} + \cdots + w_p x_{ip}$, it needs p multiplies and $p-1$ adds which gives $2p-1$ flops.
- $y_i - w^T x_i$ requires another 1 flop.
- $(y_i - w^T x_i)x_i$ requires another p flops, since x_i is a vector with p elements.
- So far, it requires $2p - 1 + 1 + p = 3p$ flops in total for one data sample.
- $\sum_{i=1}^{N}(y_i - w^T x_i)x_i$ is to sum up all data samples, and so it requires $3Np$ flops
- Thus, the total flops is proportional to Np, denoted by $O(Np)$.

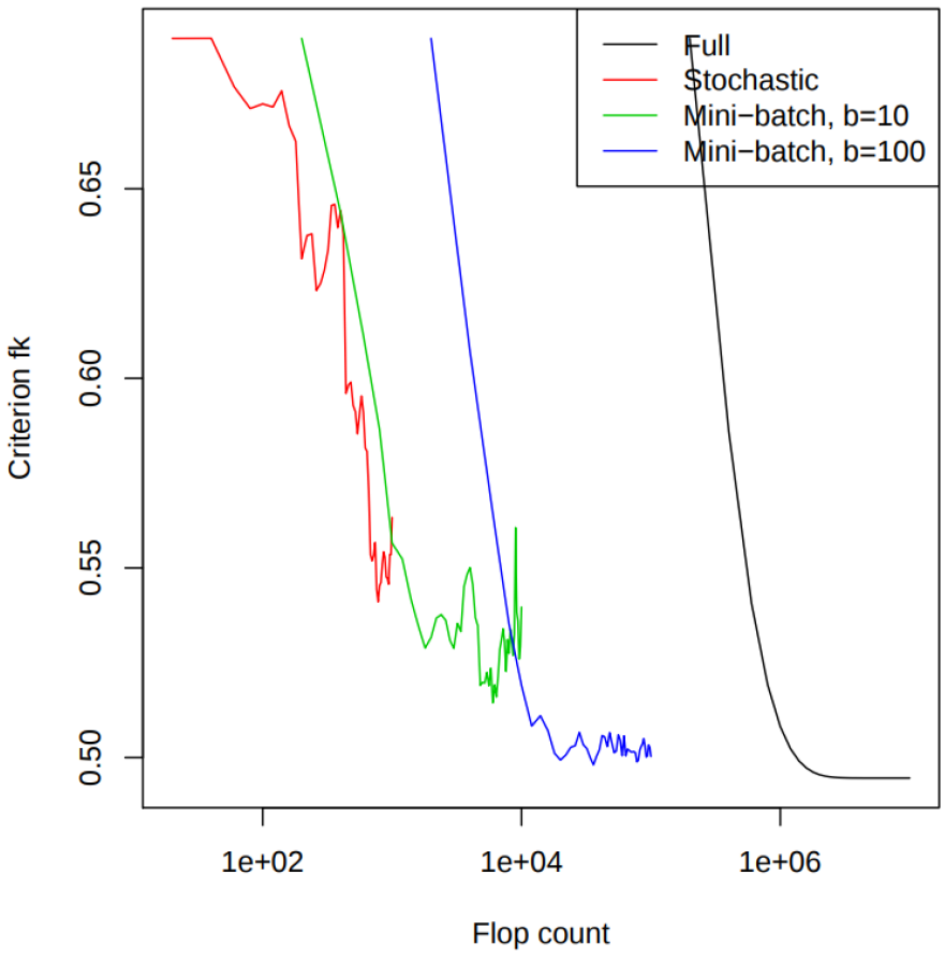
Computational cost: linear regression

- $N = 10000, p = 20$
- Regardless of mini-batch size, the criterion (objective) bounce around the optimal value for a while
- Reflects the variance in the gradient updates



Computational cost: linear regression

- $N = 10000, p = 20.$
- Faster running time for smaller batch size.
- Smaller batch sizes do not converge to optimum.



Learning rate

- SGD:

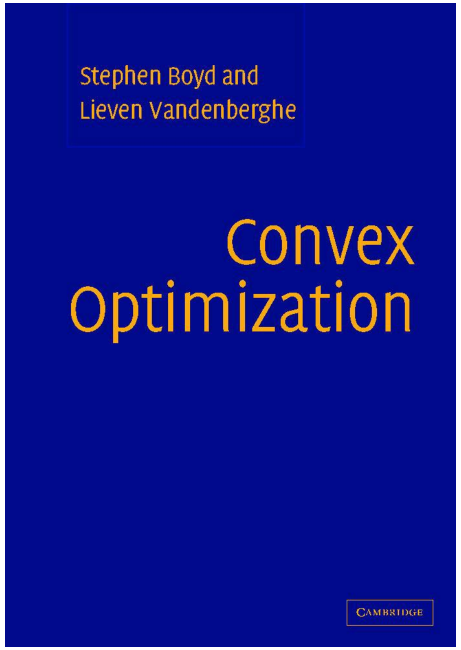
$$\theta^{t+1} = \theta^t - \alpha_t \nabla_\theta f(\theta, x_{i_t})$$

- Important to choose the learning rate α_t
- Trial and error, monitoring the value of loss function against iteration
- Example:

$$\alpha_t = \frac{1}{t}$$

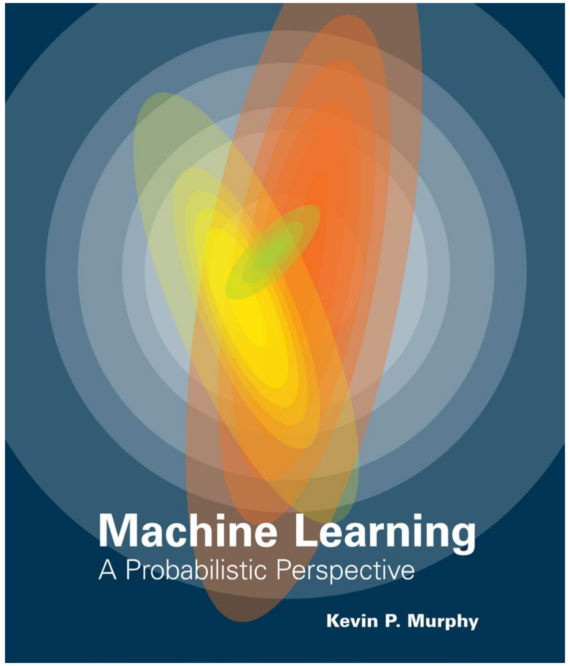
$$\alpha_t = \frac{\alpha_0}{1+\beta(t-1)}, (\alpha_0 \ and \ \beta \ are \ constant, t = 1,2, \dots)$$

Textbooks

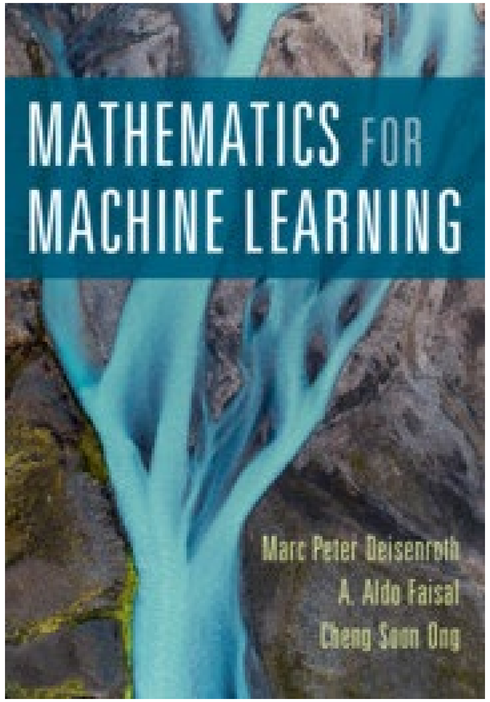


Chapter 9

https://web.stanford.edu/~boyd/cvxbook/



https://www.cs.ubc.ca/~murphyk/MLbook/



https://mml-book.github.io/book/mml-book.pdf