Optimization for Machine Learning

November 7, 2022

Program:

- Large-scale and distributed optimization
  - ↳ Automatic differentiation
  - ↳ Coordinate descent
  - ↳ Consensus optimization

Motivation: $\underset{w \in \mathbb{R}^d}{\text{minimize}} \; f(w) = \frac{1}{n} \sum_{i=1}^{n} f_i(w)$

Approach: Apply a stochastic gradient method because $n \gg 1$ and computing $\nabla f(w)$ is expensive

3 questions:

1) How do we compute $\nabla f(w)$ or $\nabla f_i(w)$ for very complex models (like neural nets)?

2) What kind of methods can we use when $d \gg 1$?

3) How do we deal with data that is distributed / decentralized on several machines?

(1) Automatic differentiation

Complex ML model:

x_i (data point) $\longrightarrow$ [Model (w)] $\longrightarrow f(x_i; w)$

Q) How do we compute $\nabla f(x_i; w)$ with respect to w?

Possibilities

→ Write down a code for $w \mapsto \nabla f(x_i; w)$

⊖ Requires time and effort

⊖ Will change every time the model changes (e.g. add a layer to a NN architecture)

→ Use a symbolic differentiation tool (Wolfram Alpha)

⊖ Not necessarily easy to use

⊖ Need to store the formula, and regenerate it every time the architecture changes

⊖ Provides a formula for the derivative, that still needs to be evaluated

→ Use approximation formulas (finite differences).

$$\nabla f(w) \approx \left[\frac{f(w + t e_i) - f(w)}{t} \right]_{i=1}^{d}$$

$e_i = [0, \dots, 0, \underset{i}{1}, 0, \dots, 0]$, $t > 0$

⊕ Only requires evaluations of f, accuracy of the approximation depends on t

⊖ Requires $d+1$ evaluations of f for one gradient approximation (often too expensive in a deep learning context)

⇒ The most popular approach in Neural Network training is automatic (or algorithmic) differentiation, that uses a graph to represent how $f(x_i; w)$ is computed, in order to evaluate $\nabla f(x_i; w)$ without looking at the data point again.

Toy example

$d=1$

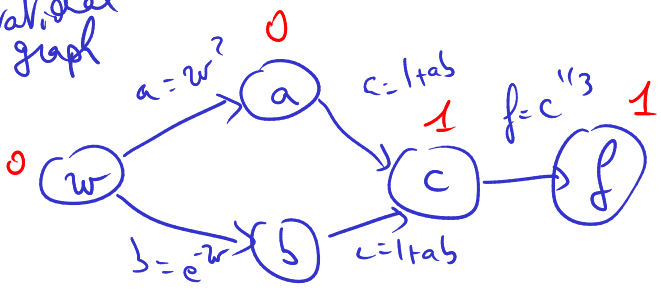
$$f(w) = \left(1 + w^2 e^{-w}\right)^{1/3}$$

Goal: $\nabla f(0) \equiv \frac{\partial f}{\partial w}(0)$

1) Decompose $f(w)$ in "elementary" operations

$$a := w^2$$
$$b := e^{-w}$$
$$c := 1 + ab$$
$$f := c^{1/3}$$

defines a computational graph →



$f(0)$ can be computed by a forward pass on the graph

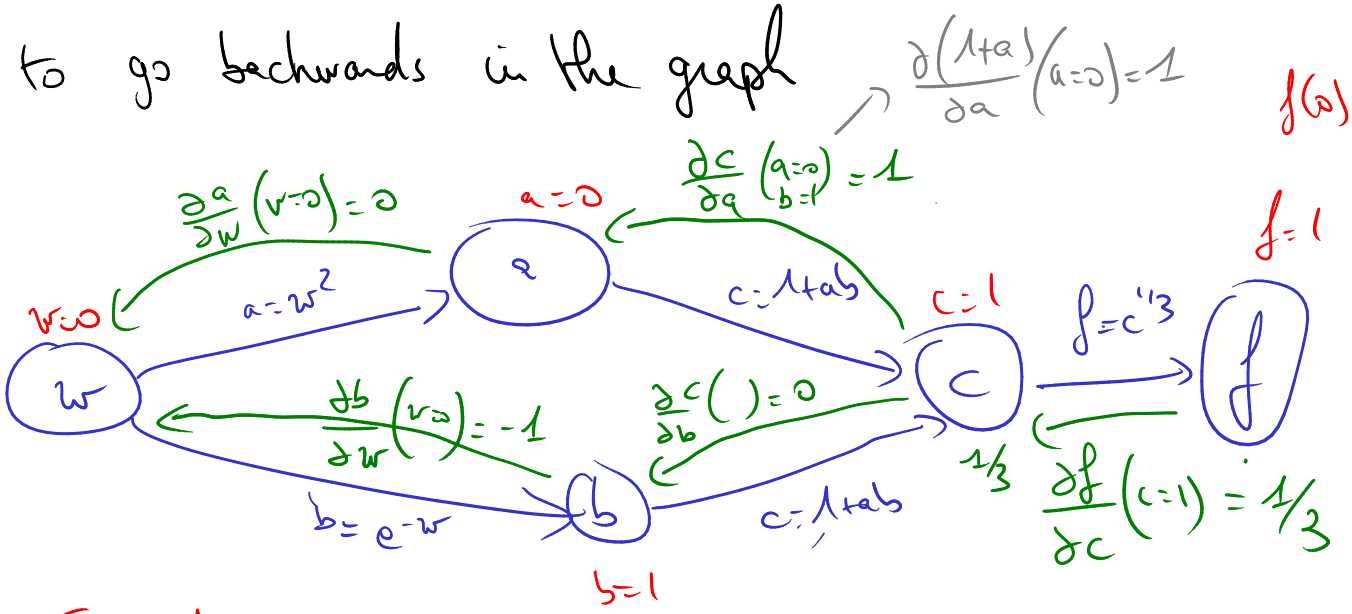
2) Apply the chain rule

$$\frac{\partial f}{\partial w}(0) = \frac{\partial f}{\partial c}\left(c(0)\right)\left(\frac{\partial c}{\partial a}\left(a(0)\right)\frac{\partial a}{\partial w}(0) + \frac{\partial c}{\partial b}\left(b(0)\right)\frac{\partial b}{\partial w}(0)\right)$$

$c(w)$ evaluated at $w=0$

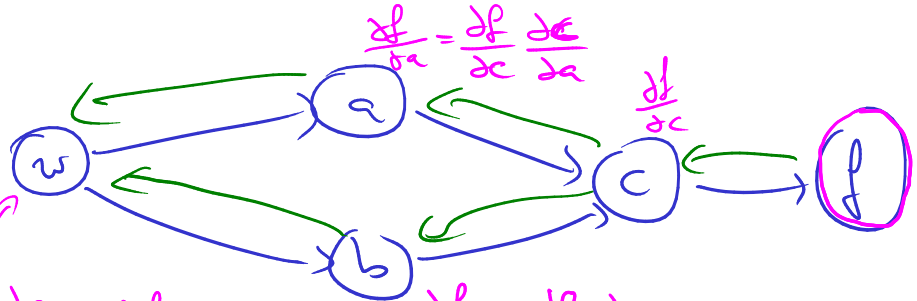
f as a function of c

to go backwards in the graph



— Forward pass

— Backward pass ⇒ For every node "backwards" of f, we store the derivative of f w.r.t. that node



$$\frac{\partial f}{\partial a} = \frac{\partial f}{\partial c}\frac{\partial c}{\partial a} \qquad \frac{\partial f}{\partial c}$$

$$\frac{\partial f}{\partial w} = \frac{\partial f}{\partial a}\frac{\partial a}{\partial w} + \frac{\partial f}{\partial b}\frac{\partial b}{\partial w} \qquad \frac{\partial f}{\partial b} = \frac{\partial f}{\partial c}\frac{\partial c}{\partial b}$$

$$= \frac{\partial f}{\partial c}\left(\frac{\partial c}{\partial a}\frac{\partial a}{\partial w} + \frac{\partial c}{\partial b}\frac{\partial b}{\partial w}\right)$$

Takeaway: In modern ML software (PyTorch, JAX, ...), gradients / derivatives are computed automatically using computational graphs and the chain rule.
⇒ Also true for generalized derivatives

② Large-scale optimization

$$\underset{w \in \mathbb{R}^d}{\text{minimize}} \; f(w) \quad \text{with } d \gg 1 \qquad \underline{f \in C^1}$$

Setup: Performing calculations with d-dimensional vectors is expensive

Approach: Perform optimization steps involving a small number of coordinates per step ⇒ Coordinate descent methods

Basic coordinate descent method ($w_0 \in \mathbb{R}^d$)

Iteration k:
→ Pick a coordinate $j_k \in \{1, \dots, d\}$.
→ Define

$$w_{k+1} = w_k - \alpha_k \underbrace{\nabla_{j_k} f(w_k)}_{j_k^{th} \text{ coordinate of the gradient}} e_{j_k} \quad \text{for } \alpha_k > 0$$

↳ j_k^{th} coordinate

- Changes one coordinate per iteration
  ⇒ Not all coordinates of w_k are necessary to compute w_{k+1} (only $[w_k]_{j_k}$ gets updated)

vector $\begin{bmatrix} 0 \\ \vdots \\ 0 \\ 1 \\ 0 \\ \vdots \\ 0 \end{bmatrix} j_k$

- Interesting when $\nabla_{j_k} f(w_k)$ can be computed without having to access all coordinates of w_k

Ex) $\underset{w \in \mathbb{R}^d}{\text{minimize}} \; \frac{1}{2n} \sum_{i=1}^{n} (x_i^T w - y_i)^2 + \lambda \sum_{j=1}^{d} [w]_j^2$

↓ separable: every term depends on its own coordinate

Linear least squares with ℓ_2 regularization

If the x_is ($\in \mathbb{R}^d$) are sparse, then the gradient of this function with respect to a particular coordinate can be computed without accessing all coordinates ("partial separability")

⇒ The cost of computing $\nabla_j f(w)$ is proportional to the number of x_is that have a non-zero j^{th} coordinate

Variants of coordinate descent depend on how $\{j_k\}$ are chosen:

→ Cyclic: . cycling through $\{1, \dots, d\}$
($j_0 = 1, j_1 = 2, \dots, j_{d-1} = d, j_d = 1, \dots$)

. cycling through a random permutation of $\{1, \dots, d\}$ ⇒ randomized cyclic

← less worst-case guarantees than the previous one

NB. A cycling approach through $(1, \dots, d)$ may not work (counter-example due to Powell)

→ Randomized: draw j_k at random (e.g. uniformly) in $\{1, \dots, d\}$ with replacement

$\Rightarrow$ Nice theoretical guarantees

↳ Can also use blocks of coordinates instead of a single one (block coordinate descent)

Remark (P) minimize $_{w \in \mathbb{R}^d} f(w) = \frac{1}{n} \sum_{i=1}^{n} f_i(w)$

Stochastic gradient: $w_{k+1} = w_k - \alpha_k \nabla f_{i_k}(w_k)$ (i_k: random data point)

Coordinate descent: $w_{k+1} = w_k - \alpha_k \nabla_{j_k} f(w_k)\, e_{j_k}$ (j_k: random coordinate)

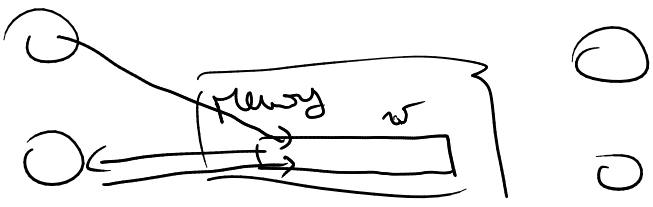
≡ random parameter among the d parameters represented by w

Coordinate descent actually corresponds to stochastic gradient on a problem related to (P), which is a problem in n variables called the dual problem

(3) Distributed and decentralized optimization

Distributed: → Several processors performing calculation

→ Shared memory ⇒ All processors access the same location for the parameter $w \in \mathbb{R}^d$



Coordinate descent methods are designed for distributed setups:

→ At every iteration, every processor performs a coordinate update on its own coordinate

⇒ Efficient, parallel implementation of coordinate descent

→ Can be done in a synchronized way

(w_k → every processor reads and updates its own coordinate → w_{k+1})

or in an asynchronized way

(Every processor runs iterations of coordinate descent and does not wait for updates from the other processors)

⇒ The latter approach is the most efficient and has been used in connection with stochastic gradient (Hogwild!, 2011)

Decentralized:

→ "Federated Learning" paradigm

→ Also called consensus optimization, network optimization

Idea: → No centralized calculation or memory

→ Instead, a set of agents with their own piece of the data are trying to solve an optimization problem that involves the entire dataset

⇒ the agents can communicate using a communication graph and are trying to

reach consensus

Mathematically: → Graph: (V, E)

V set of vertices $i = 1..n$

E set of edges $\{(i,j) \mid i,j \in \{1,..,n\}\}$

V represents the agents with their own portion of data

E represents the communication between two agents

Optimization problem

$$\underset{w \in \mathbb{R}^d}{\text{minimize}} \; f(w) = \frac{1}{n} \sum_{i=1}^{n} f_i(w)$$

, where every f_i is only available to agent i

$$\Rightarrow \quad \underset{w_{[1]} \in \mathbb{R}^d, \ldots, w_{[n]} \in \mathbb{R}^d}{\text{minimize}} \; \frac{1}{n} \sum_{i=1}^{n} f_i(w_{[i]}) \quad \text{s.t.} \quad w_{[i]} = w_{[j]} \;\; \forall (i,j) \in E$$

One vector of parameters per agent

$f_i(w_{[i]})$ known to agent i

Constraints: every agent must agree with its neighbors

Consensus optimization problem

⇒ Specific case of an optimization problem with linear constraints, that can be tackled with dual optimization methods like ADMM

General dual approach for linear constraints

↳ Consider $\underset{w \in \mathbb{R}^d}{\text{minimize}} \; f(w)$ s.t. $Aw = b$

where f is C^1

$A \in \mathbb{R}^{m \times d}$, $b \in \mathbb{R}^m$

↳ this problem can be reformulated as an unconstrained problem:

$$\text{minimize}_{w \in \mathbb{R}^d} \left[\max_{z \in \mathbb{R}^m} \underbrace{L(w,z) := f(w) + z^T(Aw-b)}_{\text{Lagrangian function of the problem}} \right]$$

(z: dual variables)

↳ the dual problem is

$$\text{maximize}_{z \in \mathbb{R}^m} \left[\min_{w \in \mathbb{R}^d} L(w,z) \right]$$

⊕ Always equivalent to a convex minimization problem ($z \mapsto -\left[\min_{w \in \mathbb{R}^d} L(w,z)\right]$ is convex)

⊕ Can rely on subgradients (implicit in the methods below) to define algorithms

Idea: One can solve the dual problem (in z) to obtain a solution to the original problem by mixing unconstrained optimization steps and dual steps.

Algorithm: Dual ascent method ($w_0 \in \mathbb{R}^d$, $z_0 \in \mathbb{R}^m$)

$$w_{k+1} \in \operatorname{argmin}_w L(w, z_k) = f(w) + z_k^T(Aw-b)$$

≈ subgradient step $z_{k+1} = z_k + \alpha_k(Aw_{k+1}-b)$ where $\alpha_k > 0$

⚠ the minimization problem may not have a solution

⇒ One can use augmented Lagrangian (instead of Lagrangian) to guarantee that the subproblem has a (unique) solution.

Augmented Lagrangian $\left(w_0 \in \mathbb{R}^d, z_0 \in \mathbb{R}^m, \lambda_0 > 0 \right)$

- $w_{k+1} \in \underset{w}{\operatorname{argmin}}\ L(w, z_k, \lambda_k) := f(w) + z_k^T (Aw - b) + \frac{\lambda_k}{2} \|Aw - b\|^2$

  "regularization" of the Lagrangian

- $z_{k+1} = z_k + \lambda_k (A w_{k+1} - b)$
- Set $\lambda_{k+1} \geq \lambda_k$] Typical choice, may also work with $\lambda_k \to 0$

⊕ Good convergence guarantees to a solution of the original, constrained optimization problem

Remaining issue: How can we handle decentralized optimization with that method, given that every agent has its own "w"?

⇒ Answer: ADMM (Alternated Direction Method of Multipliers)

ADMM with two "blocks"

Pb: $\underset{\substack{w_{[1]} \in \mathbb{R}^d \\ w_{[2]} \in \mathbb{R}^d}}{\text{minimize}}\ f_1(w_{[1]}) + f_2(w_{[2]})$ s.t. $A_1 w_{[1]} + A_2 w_{[2]} = b$

$A_1 \in \mathbb{R}^{m \times d}$, $A_2 \in \mathbb{R}^{m \times d}$

ADMM is a rewriting of augmented Lagrangian where the updates to $w_{[1]}$ and $w_{[2]}$ are done in a separate fashion

Algo ($w_{0,[1]} \in \mathbb{R}^d$, $w_{0,[2]} \in \mathbb{R}^d$, $z_0 \in \mathbb{R}^m$, $\alpha_0 > 0$)

- $w_{k+1,[1]} \in \underset{w}{\operatorname{argmin}}\ L(w, w_{k+1,[2]}, z_k, \alpha_k)$
- $w_{k+1,[2]} \in \underset{w}{\operatorname{argmin}}\ L(w_{k+1,[1]}, w, z_k, \alpha_k)$
- $z_{k+1} = z_k + \alpha_k (A_1 w_{k+1,[1]} + A_2 w_{k+1,[2]} - b)$
- Choose α_{k+1}

$\Rightarrow$ Can be extended to more than 2 agents !

$$L(w_{[1]}, w_{[2]}, z, \alpha) = f_1(w_{[1]}) + f_2(w_{[2]}) + z^T(A_1 w_{[1]} + A_2 w_{[2]} - b) + \frac{\alpha}{2} \| A_1 w_{[1]} + A_2 w_{[2]} - b \|^2$$