M2
IASD
2025-2026

OPTIMIZATION FOR MACHINE LEARNING

November 21, 2025

Today: Last lecture (Second-order methods ?)

Exam: December 11 2.15 - 4.15 pm

Allowed: 1 A4 sheet of notes (two-sided, typed or handwritten)

SECOND-ORDER METHODS

(1) Classical second-order methods for smooth optimization

$$\min_{w \in \mathbb{R}^d} f(w) \qquad f : \mathbb{R}^d \to \mathbb{R} \qquad C^2 \text{ (twice continuously differentiable)}$$

$f \in C^2$ (for us) means that at every point w, there

exist a gradient $\nabla f(w) \in \mathbb{R}^d$

and a Hessian matrix $\nabla^2 f(w) \in \mathbb{R}^{d \times d}$ symmetric $\left[\nabla^2 f(w)\right]_{ij} = \left[\nabla^2 f(w)\right]_{ji}$ $\forall\, 1 \le i \le j \le d$

such that

$$\forall v \in \mathbb{R}^d, \quad f(v) \approx f(w) + \nabla f(w)^T (v - w) + \frac{1}{2} (v-w)^T \nabla^2 f(w) (v - w)$$

when v is close to w

We can apply GD to minimize f:

$$w_{k+1} = w_k - \alpha_k \nabla f(w_k), \quad \alpha_k > 0$$

We can also use Adagrad-like methods

$$w_{k+1} = w_k - \alpha_k H_k \nabla f(w_k)$$

H_k diagonal matrix

$\alpha_k > 0$

$$\left[H_k\right]_{ii} = \left[\sum_{l=0}^{k} \left[\nabla f(w_l)\right]_i^2\right]^{-1/2}$$

Goal of diagonal scaling/Adagrad: Capture how the gradient coordinates change $\Rightarrow$ We can do

that using $\nabla^2 f$!

Newton's algorithm for minimize $f(w)$ $w \in \mathbb{R}^d$

$$w_{k+1} = w_k - \alpha_k \underbrace{\left[\nabla^2 f(w_k)\right]^{-1} \nabla f(w_k)}_{\text{Newton direction}}$$

α_k: stepsize, $\alpha_k > 0$

⚠ The inverse of the Hessian is not always well-defined

Ex) $\nabla^2 f(w_k) = \begin{bmatrix} 0 & 0 \\ 0 & 0 \end{bmatrix}$? $\nabla^2 f(w_k) = \begin{bmatrix} 1 & 0 \\ 0 & 10^{-13} \end{bmatrix}$

Motivation: Newton's method for nonlinear equations

Nonlinear equations: $F(w) = 0_{\mathbb{R}^d}$ $\quad F: \mathbb{R}^d \to \mathbb{R}^d$ $\quad C^1$

Newton's method: $w_{k+1} = w_k - J_F(w_k)^{-1} F(w_k)$

where $J_F(w_k) \in \mathbb{R}^{d \times d}$ Jacobian / Derivative of F at w_k

⚠ Might not be invertible

Newton's algorithm for optimization is Newton's method for $\nabla f(w) = 0_{\mathbb{R}^d}$

How is Newton's method used in practice?

- If f is strongly convex, then $\nabla^2 f(w) \succ 0 \ \forall w$ and thus it is invertible $\Rightarrow$ Newton's method is well-defined

• If f is convex, then $\nabla^2 f(w) \succeq 0$ and a regularized version of Newton's method is well defined:

$$w_{k+1} = w_k - \alpha_k \left[\nabla^2 f(w_k) + \lambda_k I\right]^{-1} \nabla f(w_k)$$
$$\lambda_k > 0$$

• When f is not convex, can still use regularized Newton provided λ_k is chosen so that $\nabla^2 f(w_k) + \lambda_k I \succeq 0$ (explicit regularization) or can use an implicit regularization through a trust-region

Idea: • when $\nabla^2 f(w_k) \succeq 0$, Newton iteration can be rewritten as a minimization problem:

$$w_{k+1} \in \operatorname{argmin}_w \nabla f(w_k)^T (w - w_k) + \frac{1}{2\alpha_k} (w - w_k)^T \nabla^2 f(w_k) (w - w_k)$$

• Trust-region idea: At every iteration, compute w_{k+1} by solving

Always has a solution!

$$\left[\begin{array}{l} \text{minimize}_w \ \nabla f(w_k)^T (w - w_k) + \frac{1}{2} (w - w_k)^T \nabla^2 f(w_k) (w - w_k) \\ \text{subject to} \ \|w - w_k\|_2 \le \delta_k \end{array}\right.$$

$\|w - w_k\|_2 \le \delta_k$ → Trust-region constraint

$\delta_k > 0$ → Trust-region radius (plays the role of a stepsize, chosen adaptively)

If $\nabla^2 f(w_k) = I$,

$$\operatorname{argmin}_w \nabla f(w_k)^T (w - w_k) + \frac{1}{2\alpha_k} (w - w_k)^T \nabla^2 f(w_k) (w - w_k) = w_k - \alpha_k \nabla f(w_k)$$

$$\operatorname{argmin} \nabla f(w_k)^T (w - w_k) + \frac{1}{2} (w - w_k)^T \nabla^2 f(w_k) (w - w_k) = w_k - \alpha_k \nabla f(w_k)$$
$$\text{s.t.} \ \|w - w_k\| \le \delta_k \qquad \alpha_k = \min(1, \delta_k)$$

Newton + Trust Region: Compute the direction and the stepsize simultaneously

→ Newton-type methods are not so popular in large-scale ML because the Hessian is too expensive to compute

$w \in \mathbb{R}^d \qquad \nabla f(w) \in \mathbb{R}^d \qquad \nabla^2 f(w) \in \mathbb{R}^{d \times d}$

(2) Practical Newton variants (some for ML)

- Hessian-free method

To compute a Newton direction $[\nabla^2 f(w_k)]^{-1} \nabla f(w_k)$, you need to solve $\nabla^2 f(w_k)\, d = \nabla f(w_k)$

=> You can do that using only Hessian-vector products

$\underbrace{\nabla^2 f(w_k)}_{d \times d} \underbrace{v}_{d \times 1}$ where $v \in \mathbb{R}^d$ $\quad d \times 1$

Martens 2010 for deep learning

=> Hessian-vector products can be computed at cost that is ≤ 6 times that of a gradient. (using automatic differentiation)

↳ Series of works around subsampling Newton methods for machine learning

$$\text{minimize}_{w \in \mathbb{R}^d} \; f(w) = \frac{1}{n} \sum_{i=1}^{n} f_i(w) \qquad f_i : \mathbb{R}^d \to \mathbb{R} \; C^2$$

$$w_{k+1} = w_k - \alpha_k H_k^{-1} g_k$$

compute this using only products $H_k v$

$$g_k = \frac{1}{|S_k^g|} \sum_{i \in S_k^g} \nabla f_i(w_k)$$

$$H_k = \frac{1}{|S_k^H|} \sum_{i \in S_k^H} \nabla^2 f_i(w_k)$$

⚠ In theory, $|S_k^H|$ must be chosen quite large compared to $|S_k^g|$ to guarantee convergence

- Quasi-Newton methods

Generic: $w_{k+1} = w_k - \alpha_k H_k \nabla f(w_k)$ $\quad H_k \succ 0$ such that $H_k \simeq \nabla^2 f(w_k)^{-1}$

Key idea: Build H_k using only the gradients seen so far

Most famous variant: BFGS (Broyden, Fletcher, Goldfarb, Shanno, 1970s)

$H_0 = I$ (1st step: GD step)

$\forall k \geq 0$, $s_k = \underbrace{w_{k+1} - w_k}_{\text{step at iteration } k} = -\alpha_k H_k g_k$, $y_k = \nabla f(w_{k+1}) - \nabla f(w_k)$

$$H_{k+1} = \begin{cases} \left(I - \frac{y_k s_k^T}{y_k^T s_k}\right)^T H_k \left(I - \frac{y_k s_k^T}{y_k^T s_k}\right) + \underbrace{\frac{s_k s_k^T}{s_k^T y_k}}_{\text{rank-1 update to } H_k} & \text{if } s_k^T y_k > 0 \\ H_k & \text{otherwise} \end{cases}$$

⊕ No need to invert any matrix

⊕ Can avoid storing H_k by saving $\{(s_k, y_k)\}_{k=0..k}$ (first and last found)

→ If BFGS is run for a large number of iterations, the cost of storing $\{(s_k, y_k)\}$ becomes prohibitive

⇒ Limited-memory BFGS (L-BFGS, Liu and Nocedal 1989)

Instead of keeping all pairs, just keep the m most recent ones

(s_k, y_k), (s_{k-1}, y_{k-1}), ..., (s_{k-m}, y_{k-m})

⇒ Works extremely well with $m \in \{5, 10\}$

⇒ Works (without comprehensive theory) when the function is non-convex and when the function is not C^2

⇒ works with stochastic gradients, typically requires smaller batch sizes than subsampling Newton methods

2017-2020: stochastic L-BFGS implementation in PyTorch

NB: Stochastic L-BFGS iteration

$$w_{k+1} = w_k - \alpha_k H_k \nabla f_{S_k}(w_k)$$

where $\nabla f_{S_k}(w_k) = \frac{1}{|S_k|} \sum_{i \in S_k} \nabla f_i(w_k)$

and H_k is built from $\{\nabla f_{S_k}(w_k), \dots, \nabla f_{S_{k-m}}(w_{k-m})\}$, $\{w_k, \dots, w_{k-m}\}$

α_k computed through a line search so as to satisfy

$$f_{S_k}(w_{k+1}) \leq f_{S_k}(w_k) + c_1 \nabla f_{S_k}(w_k)^T (w_{k+1} - w_k)$$

$$|\nabla f_{S_k}(w_{k+1})^T (w_{k+1} - w_k)| \geq c_2 |\nabla f_{S_k}(w_k)^T (w_{k+1} - w_k)|$$

$\frac{1}{|S_k|} \sum_{i \in S_k} f_i(w_{k+1})$

Backward pass needed

⇒ Fundamentally different implementation than others in PyTorch

---

→ Lots of effort in implementing stochastic QN for general NL, but the structure of the problems was largely ignored

NB: Ref for (L-)BFGS: Trust Region

J. Nocedal & S. J. Wright

Numerical Optimization

(2006)

(3) AdamW (published 2019)

W = Weight decay

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

Vanilla SG

$$w_{k+1} = w_k - \alpha \nabla f_{i_k}(w_k)$$

i_k drawn at random

$\alpha > 0$ fixed stepsize/learning rate

↳ In practice, SG has a "weight decay" parameter $\lambda \in [0,1)$

SG + weight decay: $w_{k+1} = \underbrace{(1-\lambda)}_{\in (0,1]} w_k - \alpha \nabla f_{i_k}(w_k)$

A couple of courses ago, I said: "weight decay $\Leftrightarrow$ ℓ_2 regularization"

Suppose that we run SG with learning rate α on

$$\text{minimize}_{w \in \mathbb{R}^d} \; f(w) + \frac{\lambda}{2\alpha}\|w\|_2^2 = \frac{1}{n}\sum_{i=1}^{n}\left(f_i(w) + \frac{\lambda}{2\alpha}\|w\|_2^2\right)$$

SG iteration: $w_{k+1} = w_k - \alpha \nabla\left(f_{i_k} + \frac{\lambda}{2\alpha}\|\cdot\|_2^2\right)(w_k)$

$$= w_k - \alpha \nabla f_{i_k}(w_k) - \alpha \times \frac{\lambda}{\alpha} w_k$$

$$= (1-\lambda) w_k - \alpha \nabla f_{i_k}(w_k)$$

This equivalence between weight decay and ℓ_2 regularization no longer holds when you add momentum (in particular, does not hold for Adam)

$\Rightarrow$ AdamW: Decouple weight decay from the regularization

Adam: $w_{k+1} = w_k - \alpha\, m_k \oslash v_k$ (componentwise division)

m_k: combination of m_{k-1} and $\nabla f_{i_k}(w_k)$ (geometric averages)

$[v_k]_i$: combination of $[v_{k-1}]_i$ and $[\nabla f_{i_k}(w_k)]_i$

AdamW: Weight decay (λ) and l_2 regularization (μ)

$$w_{k+1} = w_k - \alpha\, m_k \oslash v_k - u_k$$

m_k: depends on m_{k-1} and $\nabla f_{i_k}(w_k) + \mu w_k$ ($\nabla\left(\frac{\mu}{2}\|\cdot\|_2^2\right)(w_k)$)

$u_k = \lambda w_k$

↳ AdamW was observed to outperform Adam on image recognition datasets (in particular) and has been consistently chosen for training large networks for natural language processing

(4) Back to second-order methods: Shampoo and Noah

↳ Quasi-Newton methods only use gradients but try to approximate Hessian / second-order information

↳ In order for this to be successful in deep learning, you need to take the network structure into account

⇒ Idea behind K-FAC and Shampoo

Shampoo (Gupta, Koren, Singer 2018) → minimize $_{W \in \mathbb{R}^{d\times p}} f(W)$

↳ For matrix variables, with the idea that these matrices are proper to certain layers and certain calculations

on different layers can be done in parallel

Shampoo initialization: $L_{-1} = I_d$, $R_{-1} = I_p$, $W_0 = 0_{\mathbb{R}^{d \times p}}$

Iteration k

Compute $G_k = \nabla f(W_k)$ (or a stochastic estimate of this gradient)

Left preconditioner: $L_k = L_{k-1} + G_k G_k^T$ → $G_k = \begin{bmatrix} g_1^T \\ \vdots \\ g_d^T \end{bmatrix}$ $G_k G_k^T = \left[g_i^T g_j \right]_{ij}$

generalization of $g_k g_k^T$ for 1 vector

Right preconditioner: $R_k = R_{k-1} + G_k^T G_k$

Preconditioned gradient descent (generalization of $H_k G_k$):

$$W_{k+1} = W_k - \alpha L_k^{-1/4} G_k R_k^{-1/4}$$

$\alpha > 0$

Generalization of Full Matrix Adagrad

Adagrad: Scale g_k by $\operatorname{diag}\left[\frac{1}{\sqrt{\sum [g_k]_i^2}} \right]$

$\operatorname{diag}\left(G_k^T G_k \right)^{-1/2}$

$G_k = [g_1 \cdots g_k]$

→ Good performance (when combined with weight decay)

→ Some theory thanks to the $-1/4$ (Regret bounds)

→ MLS Commons: Yearly competition with ML benchmarks

⇓ Winner in 2024: A version of Shampoo (for the edition before June 2024)

Since then, a method called Muon attracted a lot of attention

Ref: Blog post from December 2024 revised in 2025 (Keller Jordan et al)

↳ For matrix variables

Iteration k: $G_k = \nabla f(W_k)$ (or a stochastic gradient)

$W \in \mathbb{R}^{d \times p}$

$M_k = \beta M_{k-1} + (1-\beta) G_t$ (≈ momentum update in Adam)

$Q_k = \text{NewtonSchulz}(M_k)$

$W_{t+1} = W_t - \alpha Q_k$

Goal of Newton-Schulz: Solve

Find the nearest (to M_k) orthogonal matrix → minimize $\|Q - M_k\|_F^2$ (= $\sum [Q - G_k]_{ij}^2$) subject to $Q \in \mathbb{R}^{d \times p}$, Q orthogonal ($Q^T Q = I_p$ if $p \le d$ or $Q Q^T = I_d$ if $p > d$)

→ Newton-Schulz is an iterative method to solve this problem (based on Newton's method for nonlinear equations)

→ Only requires matrix-matrix products

→ Does not require to do SVD (singular value decomposition) which is expensive but would give a solution

→ Just do 5 iterations

Numerically

Truly leverages the network structure

- Train a NN by applying ≠ algorithms to ≠ parameters
- For vectors, scalars, first and last layer parameters, apply AdamW (with finely tuned hyperparameters $a, \lambda, \dots$)
- For other (matrix) parameters like attention matrices (K, Q, V), apply the Muon update

Theoretical insights: Some with momentum turned off ($\beta = 0$)

(Bernstein and Newhouse. 2024)

"old optimizer, new norm"

Recommendation (from Hardt & Recht 2022
"Patterns, predictions and actions")

Recipe for choosing your training algorithm. (works for most cases)

- Choose SG with momentum = 0.9 or 0.
- Choose the batch size according to how many cores you have (32, 64, ...)
- Choose $\alpha > 0$ as the largest possible value such that the method does not diverge
- Run the method for a few epochs until you start seeing no progress (in the loss).
- Decrease α ($\alpha/10$) and repeat