OPTIMIZATION FOR MACHINE LEARNING

November 7, 2024

Today: Advanced stochastic gradient (with numerical illustration)

Next week (Nov. 14): Lab session on Stochastic gradient

Back to the exercise

$$\underset{x \in \mathbb{R}^d}{\text{minimize}} \quad f(x) = \frac{1}{n} \sum_{i=1}^{n} f_i(x) \qquad f_i \in C^{1,1}_{L_i}$$

Variant of SG : $$x_{k+1} = x_k - \frac{\alpha_k}{c_{i_k}} \nabla f_{i_k}(x_k)$$

where i_k is drawn randomly in $\{1, \dots, n\}$

$$\forall i \in \{1, \dots, n\}, \quad \mathbb{P}(i_k = i) = \frac{c_i}{\sum_{j=1}^{n} c_j}$$

with $$c_i = \frac{n L_i}{\sum_{j=1}^{n} L_j}$$

(1) Show that $$\mathbb{E}_{i_k}\left[\frac{1}{c_{i_k}} \nabla f_{i_k}(x_k) \right] = \nabla f(x_k)$$

$$\mathbb{E}_{i_k}\left[\frac{1}{c_{i_k}} \nabla f_{i_k}(x_k) \right] = \sum_{i=1}^{n} \mathbb{P}(i_k = i) \times \frac{1}{c_i} \nabla f_i(x_k)$$

$$= \sum_{i=1}^{n} \frac{c_i}{\sum_{j=1}^{n} c_j} \times \frac{1}{c_i} \nabla f_i(x_k)$$

does not depend on i

$$= \frac{1}{\sum_{j=1}^{n} c_j} \sum_{i=1}^{n} \nabla f_i(x_k)$$

$$\sum_{j=1}^{n} c_j = \sum_{j=1}^{n} \frac{n L_j}{\sum_{\ell=1}^{n} L_\ell} = \frac{1}{\sum_{\ell=1}^{n} L_\ell} \sum_{j=1}^{n} n L_j = n \frac{\sum_{j=1}^{n} L_j}{\sum_{\ell=1}^{n} L_\ell} = n$$

Hence $$\mathbb{E}_{i_k}\left[\frac{1}{c_{i_k}} \nabla f_{i_k}(x_k) \right] = \frac{1}{n} \sum_{i=1}^{n} \nabla f_i(x_k) = \nabla f(x_k)$$

(2) $f \in C_L^{1,1}$ $\quad L = \frac{\sum_{i=1}^n L_i}{n}$

a) If $\alpha_k = \frac{1}{L}$, what is the value of $\frac{\alpha_k}{c_{ik}}$?

$$\frac{\alpha_k}{c_{ik}} = \frac{1}{L} \times \frac{\sum_{j=1}^n L_j}{n L_{ik}} = \frac{n}{\sum_{j=1}^n L_j} \times \frac{\sum_{j=1}^n L_j}{n} \times \frac{1}{L_{ik}} = \frac{1}{L_{ik}}$$

b) Interest of importance sampling ?

Stepsize is tuned to the stochastic gradient / data point that is used at every iteration

$$L_{ik} \gg \frac{1}{n}\sum_{j=1}^n L_j = L \quad \Rightarrow \quad \frac{\alpha_k}{c_{ik}} << \alpha_k$$

ADVANCED STOCHASTIC GRADIENT METHODS

(1) Variance reduction methods

↳ Theory (and practical relevance) of SG relies on the stochastic gradients not varying too much from the true gradient

⇒ In the analysis, represented by the condition

$$\mathbb{E}_{i_k}\left[\|\nabla f_{i_k}(x_k)\|^2\right] - \|\nabla f(x_k)\|^2 \leq \sigma^2$$

↑ variance parameter

Q] what can we change in the SG method to reduce the variance of the stochastic gradient estimates ?

(without reverting to GD !)

a) Use a batch

$$\nabla f_{i_k}(x_k) \longrightarrow \frac{1}{m_b} \sum_{i \in S_k} \nabla f_i(x_k)$$

where S_k is a set of m_b indices drawn with or without replacement in $\{1, \dots, m\}$

If the indices in S_k are drawn iid from a distribution that satisfies $\mathbb{E}_{i_k}\left[\|\nabla f_{i_k}(x_k)\|^2\right] - \|\nabla f(x_k)\|^2 \leq \sigma^2$

when used in SG, then

$$\mathbb{E}_{S_k}\left[\left\| \frac{1}{m_b} \sum_{i \in S_k} \nabla f_i(x_k) \right\|^2\right] - \|\nabla f(x_k)\|^2 \leq \frac{\sigma^2}{m_b}$$

Theoretical consequences

- With 1 sample at every iteration and stepsize $\alpha > 0$, we showed that SG converges in function value (and in expectation) to a neighborhood of the optimum

$$\mathbb{E}\left[f(x_k) - f^*\right] \longrightarrow \left[0, \frac{L \alpha \sigma^2}{2\mu}\right]$$

$f \in C^{1,1}_L$ μ-strongly convex

at a rate $(1-\alpha\mu)^K$

$\frac{\sigma^2}{m_b} \leq \sigma^2$

- With a batch size m_b, can show

$$\mathbb{E}[f(x_K) - f^*] \longrightarrow \left[0, \frac{L\alpha}{2\mu}\frac{\sigma^2}{m_b}\right]$$

at a slower rate than SG $\left(1-\frac{\alpha}{m_b}\mu\right)^K$

Practical consequences: Runs of batch SG tend to vary less than runs of vanilla SG

b) Iterate averaging

↳ SG: $x_{k+1} = x_k - \alpha_k \nabla f_{i_k}(x_k)$

$$\mathbb{E}[f(x_K) - f^*] \longrightarrow \text{neighborhood of } 0$$

If we analyze the behavior of the average iterate, can show

$$\mathbb{E}\left[f\left(\frac{1}{K+1}\sum_{k=0}^{K} x_k\right) - f^*\right] \longrightarrow 0$$

⊕ Smoother behavior (like for subgradient methods)

⊖ Memory cost

Naive implementation: store $K+1$ iterates

Better implementation: store 2 vectors (may still be too much for large models)

$$\left.\begin{aligned} &x_k, x_{k+1} \\ &\hat{x}_k = \frac{1}{k+1}\sum_{l=0}^{k} x_l \end{aligned}\right) \Rightarrow \hat{x}_{k+1} = \frac{k+1}{k+2}\hat{x}_k + \frac{1}{k+2}x_{k+1}$$

(Running average of the iterates)

c) Gradient aggregation methods

↳ Idea: Combine SG steps with full gradient calculations

- First major algorithm of that form: SVRG (2013)
  Stochastic Variance-Reduced Gradient

Outer loop
Iteration k (x_k)

- Compute $\nabla f(x_k) = \frac{1}{n} \sum_{i=1}^{n} \nabla f_i(x_k)$

  Full gradient calculation (all $\nabla f_i(x_k)$'s have been computed)

- Set $\tilde{x}_0 = x_k$

- Inner loop

  $(m \in \mathbb{N}, m \geq 1)$

  For $j = 0, \dots, m-1$

  - Draw i_j uniformly in $\{1, \dots, n\}$
  - Set $\tilde{x}_{j+1} = \tilde{x}_j - \alpha_j \tilde{g}_j$ with $\alpha_j > 0$

    and

    $\tilde{g}_j = \nabla f_{i_j}(\tilde{x}_j) - \nabla f_{i_j}(x_k) + \nabla f(x_k)$

    correction

1 iteration of the inner loop = 1 stochastic gradient calculation

- Draw j_k uniformly in $\{0, \dots, m-1\}$ and set $x_{k+1} = \tilde{x}_{j_k+1}$.

Picking an iterate at random → Better behavior than the last iterate

$$\mathbb{E}_{j_k}[x_{k+1}] = \frac{1}{m} \sum_{j=0}^{m-1} \tilde{x}_{j+1}$$

Key property: $\mathbb{E}_{ij}[\tilde{g}_j] = \nabla f(\tilde{x}_j)$

→ Can also show that this method has better convergence properties than SG $\quad \mathbb{E}[f(x_k) - f^*] \to 0$ vs $\to (0, \cdot\cdot)$ for SG

Downside of SVRG: → Need to be able to compute $\nabla f(x_k)$!

→ 1 outer iteration costs $n + m$ sample gradients ⇒ Higher than 1 iteration of GD!

An important alternative to SVRG: SAGA (2015) Bach, Le Roux, Schmidt

↳ Do (1) full gradient calculation at iteration 0

$x_0 \to \{\nabla f_1(x_0), \cdots, \nabla f_n(x_0)\}$

At iteration k, you know x_k as well as an "old" gradient for every component $\nabla f_{[1]}(x_{[1]}), \cdots, \nabla f_{[n]}(x_{[n]})$

(new) stochastic gradient calculation

- Sample i_k uniformly in $\{1, \cdots, n\}$
- Compute $\nabla f_{i_k}(x_k)$
- Set $x_{k+1} = x_k - \alpha_k g_k$

correction using "old" stochastic gradients

where $g_k = \nabla f_{i_k}(x_k) - \nabla f_{[i_k]}(x_{[i_k]}) + \frac{1}{n}\sum_{i=1}^{n} \nabla f_{[i]}(x_{[i]})$

- Set $\nabla f_{[i_k]}(x_{[i_k]}) = \nabla f_{i_k}(x_k)$

↓ the last gradient of the form $\nabla f_{i_k}(\cdot)$ that was computed

Properties:

- Except at iteration 0, the cost of 1 iteration is the same than an iteration of SG
- $\mathbb{E}_{i_k}[g_k] = \nabla f(x_k)$
- Similar convergence guarantees than SVRG (better than SG)

Caveat

- Memory cost (store n gradients)
- Can be done efficiently for certain problems

Ex) Linear regression $f_i(x) = \frac{1}{2}(a_i^T x - y_i)^2$

$$\nabla f_i(x) = \underbrace{(a_i^T x - y_i)}_{\text{can store } a_i^T x \in \mathbb{R} \text{ and recompute } \nabla f_i(x) \text{ when needed}} a_i$$

$\Rightarrow$ Implementation of SAGA in scikit-learn

Refs: R.M. Gower, M. Schmidt, F. Bach and P. Richtárik
Variance-reduced methods for machine learning
(2020)

(2) Stochastic gradient methods for deep learning

→ Most popular methods used to train deep learning models that you can find in PyTorch, JAX, ...

Generic iteration

$$x_{k+1} = x_k - \alpha_k \frac{m_k}{v_k}$$

direction of displacement; coordinate-wise division; v_k: scaling vector

$\alpha_k > 0$, $m_k \in \mathbb{R}^d$, $v_k \in \mathbb{R}^d$

$$\Longleftrightarrow \quad [x_{k+1}]_j = [x_k]_j - \alpha_k \frac{[m_k]_j}{[v_k]_j} \qquad \forall j = 1..d$$

SG special case: $v_k = \begin{bmatrix} 1 \\ \vdots \\ 1 \end{bmatrix}$, $m_k = \nabla f_{i_k}(x_k)$ i_k random

Batch SG — $v_k = \begin{bmatrix} 1 \\ \vdots \\ 1 \end{bmatrix}$, $m_k = \frac{1}{|S_k|} \sum_{i \in S_k} \nabla f_i(x_k)$ S_k random

⇒ For every method below (presented using SG), there is a batch counterpart

(1) SG with momentum

$$v_k = \begin{bmatrix} 1 \\ \vdots \\ 1 \end{bmatrix}$$

$$m_k = \beta_1 m_{k-1} + (1-\beta_1) \nabla f_{i_k}(x_k)$$

previous direction; new stochastic gradient

$\beta_1 \in [0, 1)$

$\beta_1 = 0 \Rightarrow$ SG

→ Inspired by heavy ball

→ Default settings: PyTorch: $\beta_1 = 0$ JAX: $\beta_1 = 0.9$

→ Was the method of choice for training neural networks in 2012

(2) Adagrad (2011 ~ 2014)

$m_k = \nabla f_{i_k}(x_k)$ (like SG) + diagonal scaling

$$\forall j = 1..d, \quad [v_k]_j = \sqrt{\sum_{\ell=0}^{k} [m_\ell]_j^2} = \sqrt{\sum_{\ell=0}^{k} [\nabla f_{i_\ell}(x_\ell)]_j^2}$$

⇒ At every iteration, the stepsize along the j^{th} coordinate is normalized according to all past values of stochastic gradients

⇒ Equivalent to $x_{k+1} = x_k - \alpha_k H_k^{-1/2} \nabla f_{i_k}(x_k)$

$$H_k = \text{diag}\left(\sum_{\ell=0}^{k} m_\ell m_\ell^T\right)$$

$$\text{diag}(A) = \begin{bmatrix} A_{11} & & 0 \\ & \ddots & \\ 0 & & A_{dd} \end{bmatrix}$$

→ Efficient on problems in which the (stochastic) gradients have coordinates that differ in magnitude or when the gradients are sparse (lots of zero coefficients)

→ Has been applied successfully to problems from recommender systems

⚠ Tends to produce exceedingly small stepsizes very quickly

→ theoretical guarantees!

(3) RMSProp

(Root Mean Square Propagation, contemporary to Adagrad)

$$m_k = \nabla f_{i_k}(x_k)$$

$$\forall j = 1..d, \quad [v_k]_j = \sqrt{\beta_2 [v_{k-1}]_j^2 + (1-\beta_2)\left[\nabla f_{i_k}(x_k)\right]_j^2}$$

for some $\beta_2 \in [0,1)$

Adagrad $\simeq \beta_2 = \frac{1}{2}$

→ PyTorch: $\beta_2 = 0.95$, JAX $\beta_2 = 0.9$

→ RMSProp was found effective for training very deep neural networks (early 2010s)

→ Some theory, including recent results (2022-2024)

(4) Adam (2015) → Most cited optimization paper (Kingma & Ba)

$0 < \beta_1 < 1 \qquad 0 < \beta_2 < 1$

$$m_k = \frac{(1-\beta_1)}{1-\beta_1^{k+1}} \sum_{l=0}^{k} \beta_1^{k-l} \nabla f_{i_l}(x_l)$$

$\nabla f_{i_l}(x_l)$: stochastic gradient at iteration l

$\sum_{l=0}^{k} \beta_1^{k-l}$: Geometric weighted average: More weight given to the most recent estimates

$\frac{(1-\beta_1)}{1-\beta_1^{k+1}}$: Adds up an estimate of the average stochastic gradient

$\forall j = 1..d$,

$$[v_k]_j = \sqrt{\frac{1-\beta_2}{1-\beta_2^{k+1}} \sum_{l=0}^{k} \beta_2^{k-l} \left[\nabla f_{i_l}(x_l)\right]_j^2}$$

$[v_k]_j$: estimate of the variance of stochastic gradients in the j^{th} coordinate

Geometric average

→ THE method used today (Transformers, LLMs, etc)
+ Batch, stepsize tuning

→ PyTorch / JAX / Original paper: $\beta_1 = 0.9$, $\beta_2 = 0.999$

→ * Original paper had a proof of convergence (2015)
* Flaw was found in 2018 ⇒ People kept using it!

M. Hardt and B. Recht (2022)
Patterns, predictions and actions

Recommendations for SG. (+ momentum)

(1) Pick as large a batch size as possible given your computer's RAM ⇒ Memory / parallelism concerns

(2) Set $\beta_1 = 0$ or $\beta_1 = 0.9$ ⇒ Do not tune that too much (same holds for β_2)

(3) Find the largest constant stepsize such that SG doesn't diverge

(4) Run SG with that stepsize until the loss plateaus

(5) Reduce the stepsize by a constant factor (e.g 10)

$0.1 \rightarrow 0.01 \rightarrow 0.001$

(6) Repeat (4) and (5) until convergence/budget exhausted.

Exercise

$$\min_{x \in \mathbb{R}^d} f(x) = \frac{1}{n} \sum_{i=1}^{n} f_i(x)$$

Batch SG: $x_{k+1} = x_k - \frac{\alpha_k}{|S_k|} \sum_{i \in S_k} \nabla f_i(x_k)$

where $|S_k| = m_b \ \forall k \quad 1 \leq m_b \leq n$

$$\forall S \subseteq \{1, \dots, n\}, \ \mathbb{P}(S_k = S) = \begin{cases} 0 & \text{if } |S| \neq m_b \\ \frac{1}{\binom{n}{m_b}} = \frac{m_b! (n-m_b)!}{n!} & \text{if } |S| = m_b \end{cases}$$

a) Show that $\mathbb{E}_{S_k}\left[\frac{1}{|S_k|} \sum_{i \in S_k} \nabla f_i(x_k)\right] = \nabla f(x_k)$

b) Describe two modifications to the algorithm that guarantee $\mathbb{E}[f(x_k) - f^*] \xrightarrow[k\to\infty]{} 0$ for $f \in C^{1,1}_{L,\mu}$-strongly convex

---

Very final note: SG = Stochastic Gradient

SGD = Stochastic Gradient Descent

SG is not a descent method like GD
$\Rightarrow$ only guarantee descent on average