Optimization for Machine Learning
Stochastic Gradient pt 3

Today: Advanced methods (via notebook)

IMPORTANT: COURSE PROJECT

- 5 possible projects online
- Each student should indicate their 3 preferred projects
- Each student will then get 1 project assigned to them according to:
  - → preferences
  - → response time
  - → balance number of students/project

↳ Regardless of the project
- You have to pick a dataset
- Deliverables: Python code/Notebook that can be run by the instructor + Short PDF report
- Projects are individual

↳ Deadline: TBA (last year January 15, likely around that time)

↳ We'll start allocating projects next week (~Oct 30) according to expressed preferences

Back to the exercise from last session

Recall: f $C_L^{1,1}$, μ-strongly convex, $f^* = \min_{x \in \mathbb{R}^d} f(x)$ $= \frac{1}{n} \sum_{i=1}^{n} f_i(x)$

SG: $x_{k+1} = x_k - \alpha_k \nabla f_{i_k}(x_k)$ i_k random index in $\{1, ..., n\}$

batch SG: $x_{k+1} = x_k - \frac{\alpha_k}{|S_k|} \sum_{i \in S_k} \nabla f_i(x_k)$ S_k set of n_b random indices drawn iid as in SG (e.g. uniformly in $\{1, ..., n\}$)

CV rate for SG with $\alpha_k = \frac{1}{L}$

$\forall K \in \mathbb{N}$,

$$\mathbb{E}\left[f(x_K) - f^*\right] \leq \frac{\sigma^2}{2\mu} + \left(1 - \frac{\mu}{L}\right)^K \left(f(x_0) - f^* - \frac{\sigma^2}{2\mu}\right)$$

CV rate for batch SG with $\alpha_k = \frac{1}{L}$

$$\mathbb{E}\left[f(x_K) - f^*\right] \leq \frac{\sigma^2}{2\mu n_b} + \left(1 - \frac{\mu}{L}\right)^K \left(f(x_0) - f^* - \frac{\sigma^2}{2\mu n_b}\right)$$

1) Is the second guarantee better than for SG?

→ Rate is identical $\left(1 - \frac{\mu}{L}\right)^K$

→ Batch SG converges (in function value) in

$$\left[f^*, f^* + \frac{\sigma^2}{2\mu n_b}\right] \subseteq \left[f^*, f^* + \frac{\sigma^2}{2\mu}\right]$$

⇒ Better (smaller) neighborhood

→ Every iteration of batch SG is more expensive when $n_b > 1$

$\frac{m_b}{n}$ epoch VS $\frac{1}{n}$ epoch

For a fixed number of epochs N_E, the rate will be $\left(1-\frac{\mu}{L}\right)^{N_E \times n}$ for SG

$\left(1-\frac{\mu}{L}\right)^{N_E \times \frac{n}{m_b}}$ for batch SG

$\Rightarrow$ Better (faster) for SG!

CONCLUSION : IT DEPENDS!

2) What about running SG with $\alpha_k = \frac{1}{m_b L}$?

Generic formula with constant α

$$\mathbb{E}\left[f(x_k) - f^*\right] \leq \frac{L\sigma^2\alpha}{2\mu} + (1-\alpha\mu)^K\left[f(x_0) - f^* - \frac{L\sigma^2\alpha}{2\mu}\right]$$

$\alpha = \frac{1}{m_b L}$

$$\mathbb{E}\left[f(x_k) - f^*\right] \leq \frac{\sigma^2}{2\mu m_b} + \left(1 - \frac{\mu}{L m_b}\right)^K\left[f(x_0) - f^* - \frac{\sigma^2}{2\mu m_b}\right]$$

$\frac{\sigma^2}{2\mu m_b}$: Same neighborhood

$\left(1-\frac{\mu}{L m_b}\right)^K$: slower CV rate! (Smaller stepsize)

Comparison between SG $\alpha = \frac{1}{L}$ and $\alpha = \frac{1}{L m_b}$: It depends!

ADVANCED STOCHASTIC GRADIENT METHODS

1) Variance reduction methods

Basic SG:

- Randomized algorithm
  $\Rightarrow$ How much do we deviate from the average performance in practice?
- Based on stochastic gradient approximations
  $\Rightarrow$ How bad can those approximations be?

In the analysis, these variance considerations are represented by σ^2

For SG: $$\mathbb{E}_{i_k}\left[\|\nabla f_{i_k}(x_k)\|^2\right] - \underbrace{\left\|\mathbb{E}_{i_k}\left[\nabla f_{i_k}(x_k)\right]\right\|^2}_{=\|\nabla f(x_k)\|^2 \text{ under additional assumption}} \leq \sigma^2$$

Q) What can we change in the algorithm to reduce that variance?

a) Use a batch

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

and a batch method uses n_b indices drawn iid as in SG, then $\mathbb{E}_{S_k}\left[\|\ \underset{\frac{1}{|S_k|}\sum_{i\in S_k}\nabla f_i(x_k)}{\uparrow}\ \|^2\right] - \|\nabla f(x_k)\|^2 \leq \frac{\sigma^2}{n_b}$

$\frac{\sigma^2}{n_b} < \sigma^2$ if $n_b > 1$

b) Iterate averaging

$f(x) = \frac{1}{n}\sum_{i=1}^n f_i(x)$ "average of the f_i functions"

$\Rightarrow$ Under some assumptions, we proved convergence in $\underbrace{\text{expected}}_{\text{on average!}}$ value ($\mathbb{E}[f(x_k) - f^*]$)

Averaging:

- $x_{k+1} = x_k - \alpha_k \nabla f_{i_k}(x_k)$
- $\hat{x}_{k+1} = \frac{1}{k+1}\sum_{\ell=0}^{k} x_\ell$

$= \frac{k}{k+1}\hat{x}_k + \frac{1}{k+1}x_k$

c) Gradient aggregation

Idea: Combine SG steps with a full gradient estimate

First important method: SVRG (2013)

Iteration k of SVRG (x_k)

- Compute $\nabla f(x_k) = \frac{1}{m} \sum_{i=1}^{m} \nabla f_i(x_k)$
- Set $\tilde{x}_0 = x_k$
- Inner loop

For $j = 0, \dots, m-1$ $\quad m \in \mathbb{N},\ m \geq 1$

Draw $i_j \sim \mathcal{U}\{1, \dots, m\}$

Set $\tilde{x}_{j+1} = \tilde{x}_j - \alpha_j \tilde{g}_j$, where $\alpha_j > 0$

and

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

- Draw $j_k \sim \mathcal{U}(\{0, \dots, m-1\})$ and set $x_{k+1} = \tilde{x}_{j_k+1}$.

Motivation: In the inner loop, the stochastic gradient $\nabla f_{i_j}(\tilde{x}_j)$ is corrected using gradient information from the outer loop

1 outer iteration of SVRG = 1 full gradient $\nabla f(x_k)$
+ m stochastic gradients $\nabla f_{i_j}(\tilde{x}_j)$

$\Rightarrow$ A lot more expensive than SG and GD (per iteration)

⟹ But SVRG has much better CV guarantees

Ex) On $C_L^{1,1}$, μ-strongly convex f,

SG with stepsize $\frac{1}{L}$

$$\mathbb{E}\left[f(x_k) - f^*\right] \to \left(f^*, f^* + \frac{\sigma^2}{2\mu}\right)$$

SVRG stepsize $1/L$ $\quad \mathbb{E}\left[f(x_k)\right] \to f^*$

Remark: $\forall k, \forall j,$

$$\mathbb{E}_{i_j}\left[\tilde{g}_j\right] = \mathbb{E}_{i_j}\left[\nabla f_{i_j}(\tilde{x}_j) - \nabla f_{i_j}(x_k) + \nabla f(x_k)\right]$$
$$= \nabla f(\tilde{x}_j)$$

Main drawback of SVRG: per-iteration cost

↳ Multiple attempts to improve SVRG

↳ One important method: SAGA

(Bach, Le Roux, Schmidt ~2015)

SAGA

$x_0 \in \mathbb{R}^d$

1 fill gradient calculation

For $i = 1..n$

Set $\nabla f_{[i]}(x_{[i]}) = \nabla f_i(x_0)$

↳ store n component gradients

Main loop → For $k = 0, 1, \ldots$

Compute new ∇f_i

Draw $i_k \sim \mathcal{U}(\{1, \ldots, n\})$

Compute $\nabla f_{i_k}(x_k)$ → computed at iteration k

Perform step

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

and $x_{k+1} = x_k - \alpha_k g_k$

Update $\nabla f_{[i_k]}(\,)$

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

↓

Every $\nabla f_{[i]}(x_{[i]})$ contains the most recently computed $\nabla f_i(-)$

$\nabla f_{[i_k]}(\;)$ = last gradient $\nabla f_{i_k}(\;)$ that was computed

↳ In terms of accesses to data points, SAGA and SG have the same per-iteration cost (1) except for iteration 0 ($n+1$ for SAGA vs 1 for SG)

↳ In terms of storage, SAGA requires to store n vectors of dimension d $\left(\nabla f_{[1]}(x_{[1]}), \dots, \nabla f_{[n]}(x_{[n]})\right)$

(Notebook: $n = 1000$, $d = 50$
50000)

- Difficult to implement in a large-scale regime
- But efficient with moderate n and d

$\Rightarrow$ Good scikit-learn implementation
+
Ad hoc implementations for certain problems

Ex) Linear regression problem

$$f(x) = \frac{1}{2n}\|Ax-y\|^2 = \frac{1}{n}\sum_{i=1}^{n} \overbrace{\frac{(a_i^T x - y_i)^2}{2}}^{f_i(x)}$$

$$\forall i = 1..n, \quad \nabla f_i(x) = (\underbrace{a_i^T x}_{\in \mathbb{R}} - y_i)\, a_i$$

ith data point

$\Rightarrow$ If we store $a_i^T x$, we can recompute the gradient when needed using an access to (a_i, y_i)

$\Rightarrow$ During a SAGA iteration, you will access

(a_{i_k}, y_{i_k}) and can use this to compute

$\nabla f_{i_k}(x_{i_k})$ and $\nabla f_{[i_k]}(x_{[i_k]})$

$\Rightarrow$ with this approach, can go from nd storage to $n+d$

③ Stochastic gradient methods for deep learning

↳ Used to train deep learning models

↳ Implemented in PyTorch / JAX / Tensorflow

General formula (for SG, can be generalized to batch)

$$\forall k \in \mathbb{N}, \qquad x_{k+1} = x_k - \underset{\alpha_k > 0}{\alpha_k}\, m_k \oslash v_k$$

$$\Leftrightarrow \quad [x_{k+1}]_j = [x_k]_j - \alpha_k \frac{[m_k]_j}{[v_k]_j}$$

for every $j = 1..d$

m_k: direction (computed using a stochastic gradient $\nabla f_{i_k}(x_k)$)

v_k: vector of normalization for the learning rate

Ex: SG fits that framework with
$m_k = \nabla f_{i_k}(x_k)$ and $v_k = 1_{\mathbb{R}^d} = \begin{bmatrix} 1 \\ 1 \\ 1 \end{bmatrix}$

1 SG with momentum / SGD with momentum

$v_k = 1_{\mathbb{R}^d}$

$$m_k = \beta_1 \underbrace{m_{k-1}}_{\text{momentum step}} + (1-\beta_1) \underbrace{\nabla f_{i_k}(x_k)}_{\text{stochastic gradient step}}$$

$\beta_1 = 0 \Rightarrow SG$

$\beta_1 \in [0,1)$

$m_{k-1} = \frac{1}{\alpha_k}(x_k - x_{k-1})$

→ Used in 2012 for the breakthrough paper on NN for ImageNet

⇒ Typically $\beta_1 = 0.9$ and $\alpha_k = 0.01$

2 Adagrad (2011 ~ 2014)

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

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

$$= \sqrt{\sum_{\ell=0}^{k} [\nabla f_{i_\ell}(x_\ell)]_j^2}$$

$\Rightarrow$ For coordinate j, the stepsize is set according to the j^{th} coordinate of all previous stochastic gradients

$\Rightarrow$ Useful for problems with <u>sparse</u> gradients
= lots of zero coordinates

common in training recommender systems

③ RMSProp (2012)

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

$$j=1..d \quad [v_k]_j = \sqrt{\beta_2 [v_{k-1}]_j^2 + (1-\beta_2)[m_k]_j^2}$$

$$\beta_2 \in [0,1) \qquad (\beta_2 = 0 \Rightarrow \text{Adagrad})$$

$\Rightarrow$ Good performance on very deep networks

$\Rightarrow$ Gives better stepsizes than AdaGrad

4 Adam (2015) $\rightarrow$ Kingma & Ba (most cited optimization paper!)

$0 < \beta_1 < 1$

$0 < \beta_2 < 1$

$$m_k = \frac{(1-\beta_1) \overbrace{\sum_{l=0}^{k} \beta_1^{k-l} \nabla f_{i_l}(x_l)}^{\nabla f_{i_k}(x_k) + \beta_1 \nabla f_{i_{k-1}}(x_{k-1}) + \dots + \beta_1^k \nabla f_{i_0}(x_0)}}{1-\beta_1^{k+1}}$$

$$\forall j = 1..d, \quad [v_k]_j = \sqrt{\frac{(1-\beta_2) \sum_{l=0}^{k} \beta_2^{k-l} [\nabla f_{i_l}(x_l)]_j^2}{1-\beta_2^{k+1}}}$$

$\Rightarrow$ Gives more weight to the most recent stochastic gradient through a geometric average

$\Rightarrow$ Empirical estimates of the mean of the stochastic gradients and the variance of all stochastic gradient coordinates

$\hookrightarrow$ <u>THE</u> method of choice today to train large networks, especially those in NLP (LLMs, Transformers, Diffusion models, etc)

- Maybe modern architectures are tuned to give good results when trained with Adam
- Default training algorithm in many platforms (PyTorch, JAX)
- CV proof from 2015 was found to be wrong in 2018