



# Advanced Stochastic Gradient Methods

Daniil Merkulov

Optimization methods. MIPT

## Finite-sum problem

## Finite-sum problem

We consider classic finite-sample average minimization:

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

The gradient descent acts like follows:

$$x_{k+1} = x_k - \frac{\alpha_k}{n} \sum_{i=1}^n \nabla f_i(x) \quad (\text{GD})$$

- Iteration cost is linear in  $n$ .

## Finite-sum problem

We consider classic finite-sample average minimization:

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

The gradient descent acts like follows:

$$x_{k+1} = x_k - \frac{\alpha_k}{n} \sum_{i=1}^n \nabla f_i(x) \quad (\text{GD})$$

- Iteration cost is linear in  $n$ .
- Convergence with constant  $\alpha$  or line search.

## Finite-sum problem

We consider classic finite-sample average minimization:

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

The gradient descent acts like follows:

$$x_{k+1} = x_k - \frac{\alpha_k}{n} \sum_{i=1}^n \nabla f_i(x) \quad (\text{GD})$$

- Iteration cost is linear in  $n$ .
- Convergence with constant  $\alpha$  or line search.

## Finite-sum problem

We consider classic finite-sample average minimization:

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

The gradient descent acts like follows:

$$x_{k+1} = x_k - \frac{\alpha_k}{n} \sum_{i=1}^n \nabla f_i(x) \quad (\text{GD})$$

- Iteration cost is linear in  $n$ .
- Convergence with constant  $\alpha$  or line search.

Let's/ switch from the full gradient calculation to its unbiased estimator, when we randomly choose  $i_k$  index of point at each iteration uniformly:

$$x_{k+1} = x_k - \alpha_k \nabla f_{i_k}(x_k) \quad (\text{SGD})$$

With  $p(i_k = i) = \frac{1}{n}$ , the stochastic gradient is an unbiased estimate of the gradient, given by:

$$\mathbb{E}[\nabla f_{i_k}(x)] = \sum_{i=1}^n p(i_k = i) \nabla f_i(x) = \sum_{i=1}^n \frac{1}{n} \nabla f_i(x) = \frac{1}{n} \sum_{i=1}^n \nabla f_i(x) = \nabla f(x)$$

This indicates that the expected value of the stochastic gradient is equal to the actual gradient of  $f(x)$ .

## Results for Gradient Descent

Stochastic iterations are  $n$  times faster, but how many iterations are needed?

If  $\nabla f$  is Lipschitz continuous then we have:

Assumption	Deterministic Gradient Descent	Stochastic Gradient Descent
PL	$O(\log(1/\varepsilon))$	$O(1/\varepsilon)$
Convex	$O(1/\varepsilon)$	$O(1/\varepsilon^2)$
Non-Convex	$O(1/\varepsilon)$	$O(1/\varepsilon^2)$

- Stochastic has low iteration cost but slow convergence rate.

## Results for Gradient Descent

Stochastic iterations are  $n$  times faster, but how many iterations are needed?

If  $\nabla f$  is Lipschitz continuous then we have:

Assumption	Deterministic Gradient Descent	Stochastic Gradient Descent
PL	$O(\log(1/\varepsilon))$	$O(1/\varepsilon)$
Convex	$O(1/\varepsilon)$	$O(1/\varepsilon^2)$
Non-Convex	$O(1/\varepsilon)$	$O(1/\varepsilon^2)$

- Stochastic has low iteration cost but slow convergence rate.
  - Sublinear rate even in strongly-convex case.



## Results for Gradient Descent

Stochastic iterations are  $n$  times faster, but how many iterations are needed?

If  $\nabla f$  is Lipschitz continuous then we have:

Assumption	Deterministic Gradient Descent	Stochastic Gradient Descent
PL	$O(\log(1/\varepsilon))$	$O(1/\varepsilon)$
Convex	$O(1/\varepsilon)$	$O(1/\varepsilon^2)$
Non-Convex	$O(1/\varepsilon)$	$O(1/\varepsilon^2)$

- Stochastic has low iteration cost but slow convergence rate.
  - Sublinear rate even in strongly-convex case.
  - Bounds are unimprovable under standard assumptions.

## Results for Gradient Descent

Stochastic iterations are  $n$  times faster, but how many iterations are needed?

If  $\nabla f$  is Lipschitz continuous then we have:

Assumption	Deterministic Gradient Descent	Stochastic Gradient Descent
PL	$O(\log(1/\varepsilon))$	$O(1/\varepsilon)$
Convex	$O(1/\varepsilon)$	$O(1/\varepsilon^2)$
Non-Convex	$O(1/\varepsilon)$	$O(1/\varepsilon^2)$

- Stochastic has low iteration cost but slow convergence rate.
  - Sublinear rate even in strongly-convex case.
  - Bounds are unimprovable under standard assumptions.
  - Oracle returns an unbiased gradient approximation with bounded variance.

## Results for Gradient Descent

Stochastic iterations are  $n$  times faster, but how many iterations are needed?

If  $\nabla f$  is Lipschitz continuous then we have:

Assumption	Deterministic Gradient Descent	Stochastic Gradient Descent
PL	$O(\log(1/\varepsilon))$	$O(1/\varepsilon)$
Convex	$O(1/\varepsilon)$	$O(1/\varepsilon^2)$
Non-Convex	$O(1/\varepsilon)$	$O(1/\varepsilon^2)$

- Stochastic has low iteration cost but slow convergence rate.
  - Sublinear rate even in strongly-convex case.
  - Bounds are unimprovable under standard assumptions.
  - Oracle returns an unbiased gradient approximation with bounded variance.
- Momentum and Quasi-Newton-like methods do not improve rates in stochastic case. Can only improve constant factors (bottleneck is variance, not condition number).

# SGD with constant stepsize does not converge

Stochastic Gradient Descent. Batch = 2



# Main problem of SGD

$$f(x) = \frac{\mu}{2} \|x\|_2^2 + \frac{1}{m} \sum_{i=1}^m \log(1 + \exp(-y_i \langle a_i, x \rangle)) \rightarrow \min_{x \in \mathbb{R}^n}$$

Strongly convex binary logistic regression.  $m=200$ ,  $n=10$ ,  $\mu=1$ .



## Variance reduction methods

## Key idea of variance reduction

**Principle:** reducing variance of a sample of  $X$  by using a sample from another random variable  $Y$  with known expectation:

$$Z_\alpha = \alpha(X - Y) + \mathbb{E}[Y]$$

- $\mathbb{E}[Z_\alpha] = \alpha\mathbb{E}[X] + (1 - \alpha)\mathbb{E}[Y]$

## Key idea of variance reduction

**Principle:** reducing variance of a sample of  $X$  by using a sample from another random variable  $Y$  with known expectation:

$$Z_\alpha = \alpha(X - Y) + \mathbb{E}[Y]$$

- $\mathbb{E}[Z_\alpha] = \alpha\mathbb{E}[X] + (1 - \alpha)\mathbb{E}[Y]$
- $\text{var}(Z_\alpha) = \alpha^2 (\text{var}(X) + \text{var}(Y) - 2\text{cov}(X, Y))$



## Key idea of variance reduction

**Principle:** reducing variance of a sample of  $X$  by using a sample from another random variable  $Y$  with known expectation:

$$Z_\alpha = \alpha(X - Y) + \mathbb{E}[Y]$$

- $\mathbb{E}[Z_\alpha] = \alpha\mathbb{E}[X] + (1 - \alpha)\mathbb{E}[Y]$
- $\text{var}(Z_\alpha) = \alpha^2(\text{var}(X) + \text{var}(Y) - 2\text{cov}(X, Y))$ 
  - If  $\alpha = 1$ : no bias

## Key idea of variance reduction

**Principle:** reducing variance of a sample of  $X$  by using a sample from another random variable  $Y$  with known expectation:

$$Z_\alpha = \alpha(X - Y) + \mathbb{E}[Y]$$

- $\mathbb{E}[Z_\alpha] = \alpha\mathbb{E}[X] + (1 - \alpha)\mathbb{E}[Y]$
- $\text{var}(Z_\alpha) = \alpha^2 (\text{var}(X) + \text{var}(Y) - 2\text{cov}(X, Y))$ 
  - If  $\alpha = 1$ : no bias
  - If  $\alpha < 1$ : potential bias (but reduced variance).

## Key idea of variance reduction

**Principle:** reducing variance of a sample of  $X$  by using a sample from another random variable  $Y$  with known expectation:

$$Z_\alpha = \alpha(X - Y) + \mathbb{E}[Y]$$

- $\mathbb{E}[Z_\alpha] = \alpha\mathbb{E}[X] + (1 - \alpha)\mathbb{E}[Y]$
- $\text{var}(Z_\alpha) = \alpha^2(\text{var}(X) + \text{var}(Y) - 2\text{cov}(X, Y))$ 
  - If  $\alpha = 1$ : no bias
  - If  $\alpha < 1$ : potential bias (but reduced variance).
- Useful if  $Y$  is positively correlated with  $X$ .

## Key idea of variance reduction

**Principle:** reducing variance of a sample of  $X$  by using a sample from another random variable  $Y$  with known expectation:

$$Z_\alpha = \alpha(X - Y) + \mathbb{E}[Y]$$

- $\mathbb{E}[Z_\alpha] = \alpha\mathbb{E}[X] + (1 - \alpha)\mathbb{E}[Y]$
- $\text{var}(Z_\alpha) = \alpha^2(\text{var}(X) + \text{var}(Y) - 2\text{cov}(X, Y))$ 
  - If  $\alpha = 1$ : no bias
  - If  $\alpha < 1$ : potential bias (but reduced variance).
- Useful if  $Y$  is positively correlated with  $X$ .

## Key idea of variance reduction

**Principle:** reducing variance of a sample of  $X$  by using a sample from another random variable  $Y$  with known expectation:

$$Z_\alpha = \alpha(X - Y) + \mathbb{E}[Y]$$

- $\mathbb{E}[Z_\alpha] = \alpha\mathbb{E}[X] + (1 - \alpha)\mathbb{E}[Y]$
- $\text{var}(Z_\alpha) = \alpha^2(\text{var}(X) + \text{var}(Y) - 2\text{cov}(X, Y))$ 
  - If  $\alpha = 1$ : no bias
  - If  $\alpha < 1$ : potential bias (but reduced variance).
- Useful if  $Y$  is positively correlated with  $X$ .

### Application to gradient estimation ?

- SVRG: Let  $X = \nabla f_{i_k}(x^{(k-1)})$  and  $Y = \nabla f_{i_k}(\tilde{x})$ , with  $\alpha = 1$  and  $\tilde{x}$  stored.

# Key idea of variance reduction

**Principle:** reducing variance of a sample of  $X$  by using a sample from another random variable  $Y$  with known expectation:

$$Z_\alpha = \alpha(X - Y) + \mathbb{E}[Y]$$

- $\mathbb{E}[Z_\alpha] = \alpha\mathbb{E}[X] + (1 - \alpha)\mathbb{E}[Y]$
- $\text{var}(Z_\alpha) = \alpha^2(\text{var}(X) + \text{var}(Y) - 2\text{cov}(X, Y))$ 
  - If  $\alpha = 1$ : no bias
  - If  $\alpha < 1$ : potential bias (but reduced variance).
- Useful if  $Y$  is positively correlated with  $X$ .

## Application to gradient estimation ?

- SVRG: Let  $X = \nabla f_{i_k}(x^{(k-1)})$  and  $Y = \nabla f_{i_k}(\tilde{x})$ , with  $\alpha = 1$  and  $\tilde{x}$  stored.
- $\mathbb{E}[Y] = \frac{1}{n} \sum_{i=1}^n \nabla f_i(\tilde{x})$  full gradient at  $\tilde{x}$ ;

# Key idea of variance reduction

**Principle:** reducing variance of a sample of  $X$  by using a sample from another random variable  $Y$  with known expectation:

$$Z_\alpha = \alpha(X - Y) + \mathbb{E}[Y]$$

- $\mathbb{E}[Z_\alpha] = \alpha\mathbb{E}[X] + (1 - \alpha)\mathbb{E}[Y]$
- $\text{var}(Z_\alpha) = \alpha^2(\text{var}(X) + \text{var}(Y) - 2\text{cov}(X, Y))$ 
  - If  $\alpha = 1$ : no bias
  - If  $\alpha < 1$ : potential bias (but reduced variance).
- Useful if  $Y$  is positively correlated with  $X$ .

## Application to gradient estimation ?

- SVRG: Let  $X = \nabla f_{i_k}(x^{(k-1)})$  and  $Y = \nabla f_{i_k}(\tilde{x})$ , with  $\alpha = 1$  and  $\tilde{x}$  stored.
- $\mathbb{E}[Y] = \frac{1}{n} \sum_{i=1}^n \nabla f_i(\tilde{x})$  full gradient at  $\tilde{x}$ ;
- $X - Y = \nabla f_{i_k}(x^{(k-1)}) - \nabla f_{i_k}(\tilde{x})$

## SAG (Stochastic average gradient, Schmidt, Le Roux, and Bach 2013)

- Maintain table, containing gradient  $g_i$  of  $f_i$ ,  $i = 1, \dots, n$



## SAG (Stochastic average gradient, Schmidt, Le Roux, and Bach 2013)

- Maintain table, containing gradient  $g_i$  of  $f_i$ ,  $i = 1, \dots, n$
- Initialize  $x^{(0)}$ , and  $g_i^{(0)} = \nabla f_i(x^{(0)})$ ,  $i = 1, \dots, n$

## SAG (Stochastic average gradient, Schmidt, Le Roux, and Bach 2013)

- Maintain table, containing gradient  $g_i$  of  $f_i$ ,  $i = 1, \dots, n$
- Initialize  $x^{(0)}$ , and  $g_i^{(0)} = \nabla f_i(x^{(0)})$ ,  $i = 1, \dots, n$
- At steps  $k = 1, 2, 3, \dots$ , pick random  $i_k \in \{1, \dots, n\}$ , then let

$$g_{i_k}^{(k)} = \nabla f_{i_k}(x^{(k-1)}) \quad (\text{most recent gradient of } f_{i_k})$$

Set all other  $g_i^{(k)} = g_i^{(k-1)}$ ,  $i \neq i_k$ , i.e., these stay the same

## SAG (Stochastic average gradient, Schmidt, Le Roux, and Bach 2013)

- Maintain table, containing gradient  $g_i$  of  $f_i$ ,  $i = 1, \dots, n$
- Initialize  $x^{(0)}$ , and  $g_i^{(0)} = \nabla f_i(x^{(0)})$ ,  $i = 1, \dots, n$
- At steps  $k = 1, 2, 3, \dots$ , pick random  $i_k \in \{1, \dots, n\}$ , then let

$$g_{i_k}^{(k)} = \nabla f_{i_k}(x^{(k-1)}) \quad (\text{most recent gradient of } f_{i_k})$$

Set all other  $g_i^{(k)} = g_i^{(k-1)}$ ,  $i \neq i_k$ , i.e., these stay the same

- Update

$$x^{(k)} = x^{(k-1)} - \alpha_k \frac{1}{n} \sum_{i=1}^n g_i^{(k)}$$

## SAG (Stochastic average gradient, Schmidt, Le Roux, and Bach 2013)

- Maintain table, containing gradient  $g_i$  of  $f_i$ ,  $i = 1, \dots, n$
- Initialize  $x^{(0)}$ , and  $g_i^{(0)} = \nabla f_i(x^{(0)})$ ,  $i = 1, \dots, n$
- At steps  $k = 1, 2, 3, \dots$ , pick random  $i_k \in \{1, \dots, n\}$ , then let

$$g_{i_k}^{(k)} = \nabla f_{i_k}(x^{(k-1)}) \quad (\text{most recent gradient of } f_{i_k})$$

Set all other  $g_i^{(k)} = g_i^{(k-1)}$ ,  $i \neq i_k$ , i.e., these stay the same

- Update

$$x^{(k)} = x^{(k-1)} - \alpha_k \frac{1}{n} \sum_{i=1}^n g_i^{(k)}$$

- SAG gradient estimates are no longer unbiased, but they have greatly reduced variance

## SAG (Stochastic average gradient, Schmidt, Le Roux, and Bach 2013)

- Maintain table, containing gradient  $g_i$  of  $f_i$ ,  $i = 1, \dots, n$
- Initialize  $x^{(0)}$ , and  $g_i^{(0)} = \nabla f_i(x^{(0)})$ ,  $i = 1, \dots, n$
- At steps  $k = 1, 2, 3, \dots$ , pick random  $i_k \in \{1, \dots, n\}$ , then let

$$g_{i_k}^{(k)} = \nabla f_{i_k}(x^{(k-1)}) \quad (\text{most recent gradient of } f_{i_k})$$

Set all other  $g_i^{(k)} = g_i^{(k-1)}$ ,  $i \neq i_k$ , i.e., these stay the same

- Update

$$x^{(k)} = x^{(k-1)} - \alpha_k \frac{1}{n} \sum_{i=1}^n g_i^{(k)}$$

- SAG gradient estimates are no longer unbiased, but they have greatly reduced variance
- Isn't it expensive to average all these gradients? Basically just as efficient as SGD, as long we're clever:

$$x^{(k)} = x^{(k-1)} - \alpha_k \underbrace{\left( \frac{1}{n} g_{i_k}^{(k)} - \frac{1}{n} g_{i_k}^{(k-1)} + \underbrace{\frac{1}{n} \sum_{i=1}^n g_i^{(k-1)}}_{\text{old table average}} \right)}_{\text{new table average}}$$

# SAG convergence

Assume that  $f(x) = \frac{1}{n} \sum_{i=1}^n f_i(x)$ , where each  $f_i$  is differentiable, and  $\nabla f_i$  is Lipschitz with constant  $L$ .

Denote  $\bar{x}^{(k)} = \frac{1}{k} \sum_{l=0}^{k-1} x^{(l)}$ , the average iterate after  $k - 1$  steps.

## Theorem

SAG, with a fixed step size  $\alpha = \frac{1}{16L}$ , and the initialization

$$g_i^{(0)} = \nabla f_i(x^{(0)}) - \nabla f(x^{(0)}), \quad i = 1, \dots, n$$

satisfies

$$\mathbb{E}[f(\bar{x}^{(k)})] - f^* \leq \frac{48n}{k} [f(x^{(0)}) - f^*] + \frac{128L}{k} \|x^{(0)} - x^*\|^2$$

where the expectation is taken over random choices of indices.

# SAG convergence

- Result stated in terms of the average iterate  $\bar{x}^{(k)}$ , but also can be shown to hold for the best iterate  $x_{best}^{(k)}$  seen so far.

# SAG convergence

- Result stated in terms of the average iterate  $\bar{x}^{(k)}$ , but also can be shown to hold for the best iterate  $x_{best}^{(k)}$  seen so far.
- This is  $\mathcal{O}\left(\frac{1}{k}\right)$  convergence rate for SAG. Compare to  $\mathcal{O}\left(\frac{1}{k}\right)$  rate for GD, and  $\mathcal{O}\left(\frac{1}{\sqrt{k}}\right)$  rate for SGD.



# SAG convergence

- Result stated in terms of the average iterate  $\bar{x}^{(k)}$ , but also can be shown to hold for the best iterate  $x_{best}^{(k)}$  seen so far.
- This is  $\mathcal{O}\left(\frac{1}{k}\right)$  convergence rate for SAG. Compare to  $\mathcal{O}\left(\frac{1}{k}\right)$  rate for GD, and  $\mathcal{O}\left(\frac{1}{\sqrt{k}}\right)$  rate for SGD.
- But, the constants are different! Bounds after k steps:

# SAG convergence

- Result stated in terms of the average iterate  $\bar{x}^{(k)}$ , but also can be shown to hold for the best iterate  $x_{best}^{(k)}$  seen so far.
- This is  $\mathcal{O}\left(\frac{1}{k}\right)$  convergence rate for SAG. Compare to  $\mathcal{O}\left(\frac{1}{k}\right)$  rate for GD, and  $\mathcal{O}\left(\frac{1}{\sqrt{k}}\right)$  rate for SGD.
- But, the constants are different! Bounds after k steps:
  - GD:  $\frac{L\|x^{(0)}-x^*\|^2}{2k}$

# SAG convergence

- Result stated in terms of the average iterate  $\bar{x}^{(k)}$ , but also can be shown to hold for the best iterate  $x_{best}^{(k)}$  seen so far.
- This is  $\mathcal{O}\left(\frac{1}{k}\right)$  convergence rate for SAG. Compare to  $\mathcal{O}\left(\frac{1}{k}\right)$  rate for GD, and  $\mathcal{O}\left(\frac{1}{\sqrt{k}}\right)$  rate for SGD.
- But, the constants are different! Bounds after  $k$  steps:
  - GD:  $\frac{L\|x^{(0)}-x^*\|^2}{2k}$
  - SAG:  $\frac{48n[f(x^{(0)})-f^*]+128L\|x^{(0)}-x^*\|^2}{k}$

# SAG convergence

- Result stated in terms of the average iterate  $\bar{x}^{(k)}$ , but also can be shown to hold for the best iterate  $x_{best}^{(k)}$  seen so far.
- This is  $\mathcal{O}\left(\frac{1}{k}\right)$  convergence rate for SAG. Compare to  $\mathcal{O}\left(\frac{1}{k}\right)$  rate for GD, and  $\mathcal{O}\left(\frac{1}{\sqrt{k}}\right)$  rate for SGD.
- But, the constants are different! Bounds after  $k$  steps:
  - GD:  $\frac{L\|x^{(0)}-x^*\|^2}{2k}$
  - SAG:  $\frac{48n[f(x^{(0)})-f^*]+128L\|x^{(0)}-x^*\|^2}{k}$
- So the first term in SAG bound suffers from a factor of  $n$ ; authors suggest smarter initialization to make  $f(x^{(0)}) - f^*$  small (e.g., they suggest using the result of  $n$  SGD steps).

# SAG convergence

Assume further that each  $f_i$  is strongly convex with parameter  $\mu$ .

## Theorem

SAG, with a step size  $\alpha = \frac{1}{16L}$  and the same initialization as before, satisfies

$$\mathbb{E}[f(x^{(k)})] - f^* \leq \left(1 - \min\left(\frac{\mu}{16L}, \frac{1}{8n}\right)\right)^k \left(\frac{3}{2} (f(x^{(0)}) - f^*) + \frac{4L}{n} \|x^{(0)} - x^*\|^2\right)$$

## Notes:

- This is linear convergence rate  $\mathcal{O}(\gamma^k)$  for SAG. Compare this to  $\mathcal{O}(\gamma^k)$  for GD, and only  $\mathcal{O}\left(\frac{1}{k}\right)$  for SGD.

# SAG convergence

Assume further that each  $f_i$  is strongly convex with parameter  $\mu$ .

## Theorem

SAG, with a step size  $\alpha = \frac{1}{16L}$  and the same initialization as before, satisfies

$$\mathbb{E}[f(x^{(k)})] - f^* \leq \left(1 - \min\left(\frac{\mu}{16L}, \frac{1}{8n}\right)\right)^k \left(\frac{3}{2} (f(x^{(0)}) - f^*) + \frac{4L}{n} \|x^{(0)} - x^*\|^2\right)$$

## Notes:

- This is linear convergence rate  $\mathcal{O}(\gamma^k)$  for SAG. Compare this to  $\mathcal{O}(\gamma^k)$  for GD, and only  $\mathcal{O}(\frac{1}{k})$  for SGD.
- Like GD, we say SAG is adaptive to strong convexity.

# SAG convergence

Assume further that each  $f_i$  is strongly convex with parameter  $\mu$ .

## i Theorem

SAG, with a step size  $\alpha = \frac{1}{16L}$  and the same initialization as before, satisfies

$$\mathbb{E}[f(x^{(k)})] - f^* \leq \left(1 - \min\left(\frac{\mu}{16L}, \frac{1}{8n}\right)\right)^k \left(\frac{3}{2} (f(x^{(0)}) - f^*) + \frac{4L}{n} \|x^{(0)} - x^*\|^2\right)$$

## Notes:

- This is linear convergence rate  $\mathcal{O}(\gamma^k)$  for SAG. Compare this to  $\mathcal{O}(\gamma^k)$  for GD, and only  $\mathcal{O}(\frac{1}{k})$  for SGD.
- Like GD, we say SAG is adaptive to strong convexity.
- Proofs of these results not easy: 15 pages, computed-aided!

# SAG convergence notes

- Note, that the method in vanilla formulation is not applicable to the large neural networks training, due to the memory requirements.



# SAG convergence notes

- Note, that the method in vanilla formulation is not applicable to the large neural networks training, due to the memory requirements.
- In practice you can use backtracking strategy to estimate Lipschitz constant.

# SAG convergence notes

- Note, that the method in vanilla formulation is not applicable to the large neural networks training, due to the memory requirements.
- In practice you can use backtracking strategy to estimate Lipschitz constant.
  - Choose initial  $L_0$

# SAG convergence notes

- Note, that the method in vanilla formulation is not applicable to the large neural networks training, due to the memory requirements.
- In practice you can use backtracking strategy to estimate Lipschitz constant.
  - Choose initial  $L_0$
  - Increase  $L$ , until the following satisfies

$$f_{i_k}(x^{k+1}) \leq f_{i_k}(x^k) + \nabla f_{i_k}(x^k)(x^{k+1} - x^k) + \frac{L}{2} \|x^{k+1} - x^k\|_2^2$$

# SAG convergence notes

- Note, that the method in vanilla formulation is not applicable to the large neural networks training, due to the memory requirements.
- In practice you can use backtracking strategy to estimate Lipschitz constant.
  - Choose initial  $L_0$
  - Increase  $L$ , until the following satisfies

$$f_{i_k}(x^{k+1}) \leq f_{i_k}(x^k) + \nabla f_{i_k}(x^k)(x^{k+1} - x^k) + \frac{L}{2} \|x^{k+1} - x^k\|_2^2$$

- Decrease  $L$  between iterations

# SAG convergence notes

- Note, that the method in vanilla formulation is not applicable to the large neural networks training, due to the memory requirements.
- In practice you can use backtracking strategy to estimate Lipschitz constant.
  - Choose initial  $L_0$
  - Increase  $L$ , until the following satisfies

$$f_{i_k}(x^{k+1}) \leq f_{i_k}(x^k) + \nabla f_{i_k}(x^k)(x^{k+1} - x^k) + \frac{L}{2} \|x^{k+1} - x^k\|_2^2$$

- Decrease  $L$  between iterations
- Since stochastic gradient  $g(x^k) \rightarrow \nabla f(x^k)$  you can use its norm to track convergence (which is not true for SGD!)

# SAG convergence notes

- Note, that the method in vanilla formulation is not applicable to the large neural networks training, due to the memory requirements.
- In practice you can use backtracking strategy to estimate Lipschitz constant.
  - Choose initial  $L_0$
  - Increase  $L$ , until the following satisfies

$$f_{i_k}(x^{k+1}) \leq f_{i_k}(x^k) + \nabla f_{i_k}(x^k)(x^{k+1} - x^k) + \frac{L}{2} \|x^{k+1} - x^k\|_2^2$$

- Decrease  $L$  between iterations
- Since stochastic gradient  $g(x^k) \rightarrow \nabla f(x^k)$  you can use its norm to track convergence (which is not true for SGD!)
- For the generalized linear models (this includes LogReg, LLS) you need to store much less memory  $\mathcal{O}(n)$  instead of  $\mathcal{O}(pn)$ .

$$f_i(w) = \varphi(w^T x_i) \leftrightarrow \nabla f_i(w) = \varphi'(w^T x_i) x_i$$

## SAG non-uniform sampling

- The step size  $\alpha_k$  and the convergence rate of the method are determined by the constant  $L$  for  $f(x)$ , where  $L = \max_{1 \leq i \leq n} L_i$ ,  $L_i$  is the Lipschitz constant for the function  $f_i$

## SAG non-uniform sampling

- The step size  $\alpha_k$  and the convergence rate of the method are determined by the constant  $L$  for  $f(x)$ , where  $L = \max_{1 \leq i \leq n} L_i$ ,  $L_i$  is the Lipschitz constant for the function  $f_i$
- When selecting components with a probability proportional to  $L_i$ , the constant  $L$  can be reduced from  $\max_i L_i$  to  $\bar{L} = \sum_i L_i / N$ :

$$\begin{aligned} g(x) &= \frac{1}{n} \sum_{i=1}^n f_i(x) \\ &= \frac{1}{n} \sum_{i=1}^n \sum_{j=1}^{L_i} \frac{f_i(x)}{L_i} \\ &= \frac{1}{\sum_k L_k} \sum_{i=1}^n \sum_{j=1}^{L_i} \left( \sum_k \frac{L_k}{n} \frac{f_i(x)}{L_i} \right) \end{aligned}$$

With this approach, the component with a larger value of  $L_i$  is selected more often.



## SAG non-uniform sampling

- The step size  $\alpha_k$  and the convergence rate of the method are determined by the constant  $L$  for  $f(x)$ , where  $L = \max_{1 \leq i \leq n} L_i$ ,  $L_i$  is the Lipschitz constant for the function  $f_i$
- When selecting components with a probability proportional to  $L_i$ , the constant  $L$  can be reduced from  $\max_i L_i$  to  $\bar{L} = \sum_i L_i / N$ :

$$\begin{aligned} g(x) &= \frac{1}{n} \sum_{i=1}^n f_i(x) \\ &= \frac{1}{n} \sum_{i=1}^n \sum_{j=1}^{L_i} \frac{f_i(x)}{L_i} \\ &= \frac{1}{\sum_k L_k} \sum_{i=1}^n \sum_{j=1}^{L_i} \left( \sum_k \frac{L_k}{n} \frac{f_i(x)}{L_i} \right) \end{aligned}$$

With this approach, the component with a larger value of  $L_i$  is selected more often.

- To ensure convergence, component selection should be carried out according to the rule: with probability 0.5, select from a uniform distribution, with probability 0.5, select with probabilities  $L_i / \sum_j L_j$ .

## SAG non-uniform sampling

- The step size  $\alpha_k$  and the convergence rate of the method are determined by the constant  $L$  for  $f(x)$ , where  $L = \max_{1 \leq i \leq n} L_i$ ,  $L_i$  is the Lipschitz constant for the function  $f_i$
- When selecting components with a probability proportional to  $L_i$ , the constant  $L$  can be reduced from  $\max_i L_i$  to  $\bar{L} = \sum_i L_i / N$ :

$$\begin{aligned} g(x) &= \frac{1}{n} \sum_{i=1}^n f_i(x) \\ &= \frac{1}{n} \sum_{i=1}^n \sum_{j=1}^{L_i} \frac{f_i(x)}{L_i} \\ &= \frac{1}{\sum_k L_k} \sum_{i=1}^n \sum_{j=1}^{L_i} \left( \sum_k \frac{L_k}{n} \frac{f_i(x)}{L_i} \right) \end{aligned}$$

With this approach, the component with a larger value of  $L_i$  is selected more often.

- To ensure convergence, component selection should be carried out according to the rule: with probability 0.5, select from a uniform distribution, with probability 0.5, select with probabilities  $L_i / \sum_j L_j$ .
- To generate with probabilities  $L_i / \sum_j L_j$ , there is an algorithm with complexity  $O(\log N)$ .

# Stochastic Variance Reduced Gradient (SVRG)

- **Initialize:**  $\tilde{x} \in \mathbb{R}^d$

# Stochastic Variance Reduced Gradient (SVRG)

- **Initialize:**  $\tilde{x} \in \mathbb{R}^d$
- **For**  $i_{epoch} = 1$  **to** # of epochs

# Stochastic Variance Reduced Gradient (SVRG)

- **Initialize:**  $\tilde{x} \in \mathbb{R}^d$
- **For**  $i_{epoch} = 1$  **to** # of epochs
  - Compute all gradients  $\nabla f_i(\tilde{x})$ ; store  $\nabla f(\tilde{x}) = \frac{1}{n} \sum_{i=1}^n \nabla f_i(\tilde{x})$

# Stochastic Variance Reduced Gradient (SVRG)

- **Initialize:**  $\tilde{x} \in \mathbb{R}^d$
- **For**  $i_{epoch} = 1$  **to** # of epochs
  - Compute all gradients  $\nabla f_i(\tilde{x})$ ; store  $\nabla f(\tilde{x}) = \frac{1}{n} \sum_{i=1}^n \nabla f_i(\tilde{x})$
  - Initialize  $x_0 = \tilde{x}$

# Stochastic Variance Reduced Gradient (SVRG)

- **Initialize:**  $\tilde{x} \in \mathbb{R}^d$
- **For**  $i_{epoch} = 1$  **to** # of epochs
  - Compute all gradients  $\nabla f_i(\tilde{x})$ ; store  $\nabla f(\tilde{x}) = \frac{1}{n} \sum_{i=1}^n \nabla f_i(\tilde{x})$
  - Initialize  $x_0 = \tilde{x}$
  - **For**  $t = 1$  **to** length of epochs (m)

# Stochastic Variance Reduced Gradient (SVRG)

- **Initialize:**  $\tilde{x} \in \mathbb{R}^d$
- **For**  $i_{epoch} = 1$  **to** # of epochs
  - Compute all gradients  $\nabla f_i(\tilde{x})$ ; store  $\nabla f(\tilde{x}) = \frac{1}{n} \sum_{i=1}^n \nabla f_i(\tilde{x})$
  - Initialize  $x_0 = \tilde{x}$
  - **For**  $t = 1$  **to** length of epochs ( $m$ )
    - Pick  $i_t \in \{1, \dots, n\}$  uniformly at random



# Stochastic Variance Reduced Gradient (SVRG)

- **Initialize:**  $\tilde{x} \in \mathbb{R}^d$
- **For**  $i_{epoch} = 1$  **to** # of epochs
  - Compute all gradients  $\nabla f_i(\tilde{x})$ ; store  $\nabla f(\tilde{x}) = \frac{1}{n} \sum_{i=1}^n \nabla f_i(\tilde{x})$
  - Initialize  $x_0 = \tilde{x}$
  - **For**  $t = 1$  **to** length of epochs ( $m$ )
    - Pick  $i_t \in \{1, \dots, n\}$  uniformly at random
    - $x_t = x_{t-1} - \alpha [\nabla f_{i_t}(x_{t-1}) - \nabla f_{i_t}(\tilde{x}) + \nabla f(\tilde{x})]$

# Stochastic Variance Reduced Gradient (SVRG)

- **Initialize:**  $\tilde{x} \in \mathbb{R}^d$
- **For**  $i_{epoch} = 1$  **to** # of epochs
  - Compute all gradients  $\nabla f_i(\tilde{x})$ ; store  $\nabla f(\tilde{x}) = \frac{1}{n} \sum_{i=1}^n \nabla f_i(\tilde{x})$
  - Initialize  $x_0 = \tilde{x}$
  - **For**  $t = 1$  **to** length of epochs ( $m$ )
    - Pick  $i_t \in \{1, \dots, n\}$  uniformly at random
    - $x_t = x_{t-1} - \alpha [\nabla f_{i_t}(x_{t-1}) - \nabla f_{i_t}(\tilde{x}) + \nabla f(\tilde{x})]$
  - Update  $\tilde{x} = x_m$

# Stochastic Variance Reduced Gradient (SVRG)

- **Initialize:**  $\tilde{x} \in \mathbb{R}^d$
- **For**  $i_{epoch} = 1$  **to** # of epochs
  - Compute all gradients  $\nabla f_i(\tilde{x})$ ; store  $\nabla f(\tilde{x}) = \frac{1}{n} \sum_{i=1}^n \nabla f_i(\tilde{x})$
  - Initialize  $x_0 = \tilde{x}$
  - **For**  $t = 1$  **to** length of epochs ( $m$ )
    - Pick  $i_t \in \{1, \dots, n\}$  uniformly at random
    - $x_t = x_{t-1} - \alpha [\nabla f_{i_t}(x_{t-1}) - \nabla f_{i_t}(\tilde{x}) + \nabla f(\tilde{x})]$
  - Update  $\tilde{x} = x_m$

# Stochastic Variance Reduced Gradient (SVRG)

- **Initialize:**  $\tilde{x} \in \mathbb{R}^d$
- **For**  $i_{epoch} = 1$  **to** # of epochs
  - Compute all gradients  $\nabla f_i(\tilde{x})$ ; store  $\nabla f(\tilde{x}) = \frac{1}{n} \sum_{i=1}^n \nabla f_i(\tilde{x})$
  - Initialize  $x_0 = \tilde{x}$
  - **For**  $t = 1$  **to** length of epochs ( $m$ )
    - Pick  $i_t \in \{1, \dots, n\}$  uniformly at random
    - $x_t = x_{t-1} - \alpha [\nabla f_{i_t}(x_{t-1}) - \nabla f_{i_t}(\tilde{x}) + \nabla f(\tilde{x})]$
  - Update  $\tilde{x} = x_m$

## Notes:

- Two gradient evaluations per inner step.

# Stochastic Variance Reduced Gradient (SVRG)

- **Initialize:**  $\tilde{x} \in \mathbb{R}^d$
- **For**  $i_{epoch} = 1$  **to** # of epochs
  - Compute all gradients  $\nabla f_i(\tilde{x})$ ; store  $\nabla f(\tilde{x}) = \frac{1}{n} \sum_{i=1}^n \nabla f_i(\tilde{x})$
  - Initialize  $x_0 = \tilde{x}$
  - **For**  $t = 1$  **to** length of epochs ( $m$ )
    - Pick  $i_t \in \{1, \dots, n\}$  uniformly at random
    - $x_t = x_{t-1} - \alpha [\nabla f_{i_t}(x_{t-1}) - \nabla f_{i_t}(\tilde{x}) + \nabla f(\tilde{x})]$
  - Update  $\tilde{x} = x_m$

## Notes:

- Two gradient evaluations per inner step.
- Two parameters: length of epochs + step-size  $\alpha$ .

# Stochastic Variance Reduced Gradient (SVRG)

- **Initialize:**  $\tilde{x} \in \mathbb{R}^d$
- **For**  $i_{epoch} = 1$  **to** # of epochs
  - Compute all gradients  $\nabla f_i(\tilde{x})$ ; store  $\nabla f(\tilde{x}) = \frac{1}{n} \sum_{i=1}^n \nabla f_i(\tilde{x})$
  - Initialize  $x_0 = \tilde{x}$
  - **For**  $t = 1$  **to** length of epochs ( $m$ )
    - Pick  $i_t \in \{1, \dots, n\}$  uniformly at random
    - $x_t = x_{t-1} - \alpha [\nabla f_{i_t}(x_{t-1}) - \nabla f_{i_t}(\tilde{x}) + \nabla f(\tilde{x})]$
  - Update  $\tilde{x} = x_m$

## Notes:

- Two gradient evaluations per inner step.
- Two parameters: length of epochs + step-size  $\alpha$ .
- Linear convergence rate, simple proof.

## Adaptivity or scaling

## Adagrad (Duchi, Hazan, and Singer 2010)

Very popular adaptive method. Let  $g^{(k)} = \nabla f_{i_k}(x^{(k-1)})$ , and update for  $j = 1, \dots, p$ :

$$v_j^{(k)} = v_j^{(k-1)} + (g_j^{(k)})^2$$
$$x_j^{(k)} = x_j^{(k-1)} - \alpha \frac{g_j^{(k)}}{\sqrt{v_j^{(k)} + \epsilon}}$$

### Notes:

- AdaGrad does not require tuning the learning rate:  $\alpha > 0$  is a fixed constant, and the learning rate decreases naturally over iterations.



## Adagrad (Duchi, Hazan, and Singer 2010)

Very popular adaptive method. Let  $g^{(k)} = \nabla f_{i_k}(x^{(k-1)})$ , and update for  $j = 1, \dots, p$ :

$$v_j^{(k)} = v_j^{(k-1)} + (g_j^{(k)})^2$$
$$x_j^{(k)} = x_j^{(k-1)} - \alpha \frac{g_j^{(k)}}{\sqrt{v_j^{(k)} + \epsilon}}$$

### Notes:

- AdaGrad does not require tuning the learning rate:  $\alpha > 0$  is a fixed constant, and the learning rate decreases naturally over iterations.
- The learning rate of rare informative features diminishes slowly.

## Adagrad (Duchi, Hazan, and Singer 2010)

Very popular adaptive method. Let  $g^{(k)} = \nabla f_{i_k}(x^{(k-1)})$ , and update for  $j = 1, \dots, p$ :

$$v_j^{(k)} = v_j^{(k-1)} + (g_j^{(k)})^2$$
$$x_j^{(k)} = x_j^{(k-1)} - \alpha \frac{g_j^{(k)}}{\sqrt{v_j^{(k)} + \epsilon}}$$

### Notes:

- AdaGrad does not require tuning the learning rate:  $\alpha > 0$  is a fixed constant, and the learning rate decreases naturally over iterations.
- The learning rate of rare informative features diminishes slowly.
- Can drastically improve over SGD in sparse problems.

## Adagrad (Duchi, Hazan, and Singer 2010)

Very popular adaptive method. Let  $g^{(k)} = \nabla f_{i_k}(x^{(k-1)})$ , and update for  $j = 1, \dots, p$ :

$$v_j^{(k)} = v_j^{(k-1)} + (g_j^{(k)})^2$$
$$x_j^{(k)} = x_j^{(k-1)} - \alpha \frac{g_j^{(k)}}{\sqrt{v_j^{(k)} + \epsilon}}$$

### Notes:

- AdaGrad does not require tuning the learning rate:  $\alpha > 0$  is a fixed constant, and the learning rate decreases naturally over iterations.
- The learning rate of rare informative features diminishes slowly.
- Can drastically improve over SGD in sparse problems.
- Main weakness is the monotonic accumulation of gradients in the denominator. AdaDelta, Adam, AMSGrad, etc. improve on this, popular in training deep neural networks.

## Adagrad (Duchi, Hazan, and Singer 2010)

Very popular adaptive method. Let  $g^{(k)} = \nabla f_{i_k}(x^{(k-1)})$ , and update for  $j = 1, \dots, p$ :

$$v_j^{(k)} = v_j^{(k-1)} + (g_j^{(k)})^2$$
$$x_j^{(k)} = x_j^{(k-1)} - \alpha \frac{g_j^{(k)}}{\sqrt{v_j^{(k)} + \epsilon}}$$

### Notes:

- AdaGrad does not require tuning the learning rate:  $\alpha > 0$  is a fixed constant, and the learning rate decreases naturally over iterations.
- The learning rate of rare informative features diminishes slowly.
- Can drastically improve over SGD in sparse problems.
- Main weakness is the monotonic accumulation of gradients in the denominator. AdaDelta, Adam, AMSGrad, etc. improve on this, popular in training deep neural networks.
- The constant  $\epsilon$  is typically set to  $10^{-6}$  to ensure that we do not suffer from division by zero or overly large step sizes.

## RMSProp (Tieleman and Hinton, 2012)

An enhancement of AdaGrad that addresses its aggressive, monotonically decreasing learning rate. Uses a moving average of squared gradients to adjust the learning rate for each weight. Let  $g^{(k)} = \nabla f_{i_k}(x^{(k-1)})$  and update rule for  $j = 1, \dots, p$ :

$$v_j^{(k)} = \gamma v_j^{(k-1)} + (1 - \gamma)(g_j^{(k)})^2$$

$$x_j^{(k)} = x_j^{(k-1)} - \alpha \frac{g_j^{(k)}}{\sqrt{v_j^{(k)} + \epsilon}}$$

### Notes:

- RMSProp divides the learning rate for a weight by a running average of the magnitudes of recent gradients for that weight.

## RMSProp (Tieleman and Hinton, 2012)

An enhancement of AdaGrad that addresses its aggressive, monotonically decreasing learning rate. Uses a moving average of squared gradients to adjust the learning rate for each weight. Let  $g^{(k)} = \nabla f_{i_k}(x^{(k-1)})$  and update rule for  $j = 1, \dots, p$ :

$$v_j^{(k)} = \gamma v_j^{(k-1)} + (1 - \gamma)(g_j^{(k)})^2$$

$$x_j^{(k)} = x_j^{(k-1)} - \alpha \frac{g_j^{(k)}}{\sqrt{v_j^{(k)} + \epsilon}}$$

### Notes:

- RMSProp divides the learning rate for a weight by a running average of the magnitudes of recent gradients for that weight.
- Allows for a more nuanced adjustment of learning rates than AdaGrad, making it suitable for non-stationary problems.

## RMSProp (Tieleman and Hinton, 2012)

An enhancement of AdaGrad that addresses its aggressive, monotonically decreasing learning rate. Uses a moving average of squared gradients to adjust the learning rate for each weight. Let  $g^{(k)} = \nabla f_{i_k}(x^{(k-1)})$  and update rule for  $j = 1, \dots, p$ :

$$v_j^{(k)} = \gamma v_j^{(k-1)} + (1 - \gamma)(g_j^{(k)})^2$$

$$x_j^{(k)} = x_j^{(k-1)} - \alpha \frac{g_j^{(k)}}{\sqrt{v_j^{(k)} + \epsilon}}$$

### Notes:

- RMSProp divides the learning rate for a weight by a running average of the magnitudes of recent gradients for that weight.
- Allows for a more nuanced adjustment of learning rates than AdaGrad, making it suitable for non-stationary problems.
- Commonly used in training neural networks, particularly in recurrent neural networks.

## Adadelta (Zeiler, 2012)

An extension of RMSProp that seeks to reduce its dependence on a manually set global learning rate. Instead of accumulating all past squared gradients, Adadelta limits the window of accumulated past gradients to some fixed size  $w$ . Update mechanism does not require learning rate  $\alpha$ :

$$v_j^{(k)} = \gamma v_j^{(k-1)} + (1 - \gamma)(g_j^{(k)})^2$$

$$\tilde{g}_j^{(k)} = \frac{\sqrt{\Delta x_j^{(k-1)} + \epsilon}}{\sqrt{v_j^{(k)} + \epsilon}} g_j^{(k)}$$

$$x_j^{(k)} = x_j^{(k-1)} - \tilde{g}_j^{(k)}$$

$$\Delta x_j^{(k)} = \rho \Delta x_j^{(k-1)} + (1 - \rho)(\tilde{g}_j^{(k)})^2$$

### Notes:

- Adadelta adapts learning rates based on a moving window of gradient updates, rather than accumulating all past gradients. This way, learning rates adjusted are more robust to changes in model's dynamics.



## Adadelta (Zeiler, 2012)

An extension of RMSProp that seeks to reduce its dependence on a manually set global learning rate. Instead of accumulating all past squared gradients, Adadelta limits the window of accumulated past gradients to some fixed size  $w$ . Update mechanism does not require learning rate  $\alpha$ :

$$v_j^{(k)} = \gamma v_j^{(k-1)} + (1 - \gamma)(g_j^{(k)})^2$$

$$\tilde{g}_j^{(k)} = \frac{\sqrt{\Delta x_j^{(k-1)} + \epsilon}}{\sqrt{v_j^{(k)} + \epsilon}} g_j^{(k)}$$

$$x_j^{(k)} = x_j^{(k-1)} - \tilde{g}_j^{(k)}$$

$$\Delta x_j^{(k)} = \rho \Delta x_j^{(k-1)} + (1 - \rho)(\tilde{g}_j^{(k)})^2$$

### Notes:

- Adadelta adapts learning rates based on a moving window of gradient updates, rather than accumulating all past gradients. This way, learning rates adjusted are more robust to changes in model's dynamics.
- The method does not require an initial learning rate setting, making it easier to configure.

## Adadelta (Zeiler, 2012)

An extension of RMSProp that seeks to reduce its dependence on a manually set global learning rate. Instead of accumulating all past squared gradients, Adadelta limits the window of accumulated past gradients to some fixed size  $w$ . Update mechanism does not require learning rate  $\alpha$ :

$$v_j^{(k)} = \gamma v_j^{(k-1)} + (1 - \gamma)(g_j^{(k)})^2$$

$$\tilde{g}_j^{(k)} = \frac{\sqrt{\Delta x_j^{(k-1)} + \epsilon}}{\sqrt{v_j^{(k)} + \epsilon}} g_j^{(k)}$$

$$x_j^{(k)} = x_j^{(k-1)} - \tilde{g}_j^{(k)}$$

$$\Delta x_j^{(k)} = \rho \Delta x_j^{(k-1)} + (1 - \rho)(\tilde{g}_j^{(k)})^2$$

### Notes:

- Adadelta adapts learning rates based on a moving window of gradient updates, rather than accumulating all past gradients. This way, learning rates adjusted are more robust to changes in model's dynamics.
- The method does not require an initial learning rate setting, making it easier to configure.
- Often used in deep learning where parameter scales differ significantly across layers.

## Adam (Kingma and Ba, 2014) <sup>1</sup> <sup>2</sup>

Combines elements from both AdaGrad and RMSProp. It considers an exponentially decaying average of past gradients and squared gradients.

EMA: 
$$m_j^{(k)} = \beta_1 m_j^{(k-1)} + (1 - \beta_1) g_j^{(k)}$$
$$v_j^{(k)} = \beta_2 v_j^{(k-1)} + (1 - \beta_2) (g_j^{(k)})^2$$

Bias correction: 
$$\hat{m}_j = \frac{m_j^{(k)}}{1 - \beta_1^k}$$
$$\hat{v}_j = \frac{v_j^{(k)}}{1 - \beta_2^k}$$

Update: 
$$x_j^{(k)} = x_j^{(k-1)} - \alpha \frac{\hat{m}_j}{\sqrt{\hat{v}_j} + \epsilon}$$

### Notes:

- It corrects the bias towards zero in the initial moments seen in other methods like RMSProp, making the estimates more accurate.

# Adam (Kingma and Ba, 2014) <sup>1 2</sup>

Combines elements from both AdaGrad and RMSProp. It considers an exponentially decaying average of past gradients and squared gradients.

EMA: 
$$m_j^{(k)} = \beta_1 m_j^{(k-1)} + (1 - \beta_1) g_j^{(k)}$$
$$v_j^{(k)} = \beta_2 v_j^{(k-1)} + (1 - \beta_2) (g_j^{(k)})^2$$

Bias correction: 
$$\hat{m}_j = \frac{m_j^{(k)}}{1 - \beta_1^k}$$
$$\hat{v}_j = \frac{v_j^{(k)}}{1 - \beta_2^k}$$

Update: 
$$x_j^{(k)} = x_j^{(k-1)} - \alpha \frac{\hat{m}_j}{\sqrt{\hat{v}_j} + \epsilon}$$

## Notes:

- It corrects the bias towards zero in the initial moments seen in other methods like RMSProp, making the estimates more accurate.
- Одна из самых цитируемых научных работ в мире

## Adam (Kingma and Ba, 2014) <sup>1 2</sup>

Combines elements from both AdaGrad and RMSProp. It considers an exponentially decaying average of past gradients and squared gradients.

EMA: 
$$m_j^{(k)} = \beta_1 m_j^{(k-1)} + (1 - \beta_1) g_j^{(k)}$$
$$v_j^{(k)} = \beta_2 v_j^{(k-1)} + (1 - \beta_2) (g_j^{(k)})^2$$

Bias correction: 
$$\hat{m}_j = \frac{m_j^{(k)}}{1 - \beta_1^k}$$
$$\hat{v}_j = \frac{v_j^{(k)}}{1 - \beta_2^k}$$

Update: 
$$x_j^{(k)} = x_j^{(k-1)} - \alpha \frac{\hat{m}_j}{\sqrt{\hat{v}_j} + \epsilon}$$

### Notes:

- It corrects the bias towards zero in the initial moments seen in other methods like RMSProp, making the estimates more accurate.
- Одна из самых цитируемых научных работ в мире
- В 2018-2019 годах вышли статьи, указывающие на ошибку в оригинальной статье

# Adam (Kingma and Ba, 2014) <sup>1 2</sup>

Combines elements from both AdaGrad and RMSProp. It considers an exponentially decaying average of past gradients and squared gradients.

EMA: 
$$m_j^{(k)} = \beta_1 m_j^{(k-1)} + (1 - \beta_1) g_j^{(k)}$$
$$v_j^{(k)} = \beta_2 v_j^{(k-1)} + (1 - \beta_2) (g_j^{(k)})^2$$

Bias correction: 
$$\hat{m}_j = \frac{m_j^{(k)}}{1 - \beta_1^k}$$
$$\hat{v}_j = \frac{v_j^{(k)}}{1 - \beta_2^k}$$

Update: 
$$x_j^{(k)} = x_j^{(k-1)} - \alpha \frac{\hat{m}_j}{\sqrt{\hat{v}_j} + \epsilon}$$

## Notes:

- It corrects the bias towards zero in the initial moments seen in other methods like RMSProp, making the estimates more accurate.
- Одна из самых цитируемых научных работ в мире
- В 2018-2019 годах вышли статьи, указывающие на ошибку в оригинальной статье
- Не сходится для некоторых простых задач (даже выпуклых)

# Adam (Kingma and Ba, 2014) <sup>1 2</sup>

Combines elements from both AdaGrad and RMSProp. It considers an exponentially decaying average of past gradients and squared gradients.

EMA: 
$$m_j^{(k)} = \beta_1 m_j^{(k-1)} + (1 - \beta_1) g_j^{(k)}$$
$$v_j^{(k)} = \beta_2 v_j^{(k-1)} + (1 - \beta_2) (g_j^{(k)})^2$$

Bias correction: 
$$\hat{m}_j = \frac{m_j^{(k)}}{1 - \beta_1^k}$$
$$\hat{v}_j = \frac{v_j^{(k)}}{1 - \beta_2^k}$$

Update: 
$$x_j^{(k)} = x_j^{(k-1)} - \alpha \frac{\hat{m}_j}{\sqrt{\hat{v}_j} + \epsilon}$$

## Notes:

- It corrects the bias towards zero in the initial moments seen in other methods like RMSProp, making the estimates more accurate.
- Одна из самых цитируемых научных работ в мире
- В 2018-2019 годах вышли статьи, указывающие на ошибку в оригинальной статье
- Не сходится для некоторых простых задач (даже выпуклых)
- Почему-то очень хорошо работает для некоторых сложных задач

# Adam (Kingma and Ba, 2014)<sup>1 2</sup>

Combines elements from both AdaGrad and RMSProp. It considers an exponentially decaying average of past gradients and squared gradients.

EMA: 
$$m_j^{(k)} = \beta_1 m_j^{(k-1)} + (1 - \beta_1) g_j^{(k)}$$
$$v_j^{(k)} = \beta_2 v_j^{(k-1)} + (1 - \beta_2) (g_j^{(k)})^2$$

Bias correction: 
$$\hat{m}_j = \frac{m_j^{(k)}}{1 - \beta_1^k}$$
$$\hat{v}_j = \frac{v_j^{(k)}}{1 - \beta_2^k}$$

Update: 
$$x_j^{(k)} = x_j^{(k-1)} - \alpha \frac{\hat{m}_j}{\sqrt{\hat{v}_j} + \epsilon}$$

## Notes:

- It corrects the bias towards zero in the initial moments seen in other methods like RMSProp, making the estimates more accurate.
- Одна из самых цитируемых научных работ в мире
- В 2018-2019 годах вышли статьи, указывающие на ошибку в оригинальной статье
- Не сходится для некоторых простых задач (даже выпуклых)
- Почему-то очень хорошо работает для некоторых сложных задач
- Гораздо лучше работает для языковых моделей, чем для задач компьютерного зрения - почему?

---

<sup>1</sup>Adam: A Method for Stochastic Optimization

<sup>2</sup>On the Convergence of Adam and Beyond



## AdamW (Loshchilov & Hutter, 2017)

Addresses a common issue with  $\ell_2$  regularization in adaptive optimizers like Adam. Standard  $\ell_2$  regularization adds  $\lambda\|x\|^2$  to the loss, resulting in a gradient term  $\lambda x$ . In Adam, this term gets scaled by the adaptive learning rate  $(\sqrt{\hat{v}_j} + \epsilon)$ , coupling the weight decay to the gradient magnitudes.

AdamW decouples weight decay from the gradient adaptation step.

Update rule:

$$\begin{aligned}m_j^{(k)} &= \beta_1 m_j^{(k-1)} + (1 - \beta_1) g_j^{(k)} \\v_j^{(k)} &= \beta_2 v_j^{(k-1)} + (1 - \beta_2) (g_j^{(k)})^2 \\ \hat{m}_j &= \frac{m_j^{(k)}}{1 - \beta_1^k}, \quad \hat{v}_j = \frac{v_j^{(k)}}{1 - \beta_2^k} \\ x_j^{(k)} &= x_j^{(k-1)} - \alpha \left( \frac{\hat{m}_j}{\sqrt{\hat{v}_j} + \epsilon} + \lambda x_j^{(k-1)} \right)\end{aligned}$$

### Notes:

- The weight decay term  $\lambda x_j^{(k-1)}$  is added *after* the adaptive gradient step.

## AdamW (Loshchilov & Hutter, 2017)

Addresses a common issue with  $\ell_2$  regularization in adaptive optimizers like Adam. Standard  $\ell_2$  regularization adds  $\lambda\|x\|^2$  to the loss, resulting in a gradient term  $\lambda x$ . In Adam, this term gets scaled by the adaptive learning rate  $(\sqrt{\hat{v}_j} + \epsilon)$ , coupling the weight decay to the gradient magnitudes.

AdamW decouples weight decay from the gradient adaptation step.

Update rule:

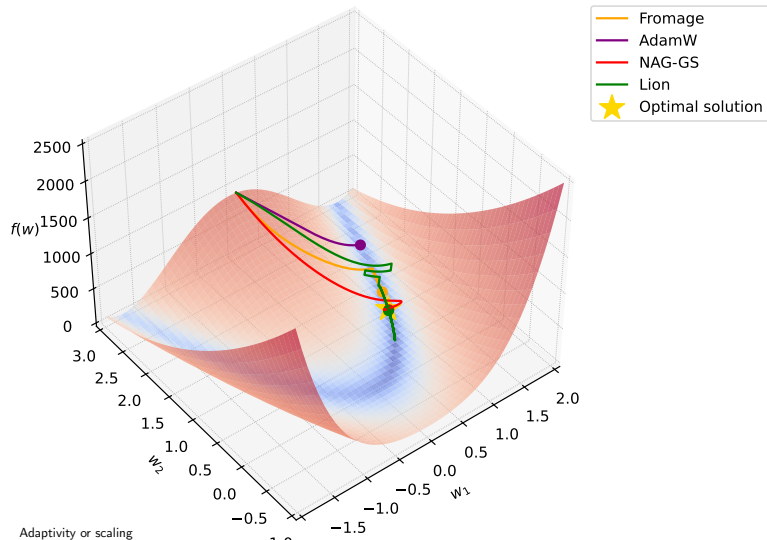
$$\begin{aligned}m_j^{(k)} &= \beta_1 m_j^{(k-1)} + (1 - \beta_1) g_j^{(k)} \\v_j^{(k)} &= \beta_2 v_j^{(k-1)} + (1 - \beta_2) (g_j^{(k)})^2 \\ \hat{m}_j &= \frac{m_j^{(k)}}{1 - \beta_1^k}, \quad \hat{v}_j = \frac{v_j^{(k)}}{1 - \beta_2^k} \\ x_j^{(k)} &= x_j^{(k-1)} - \alpha \left( \frac{\hat{m}_j}{\sqrt{\hat{v}_j} + \epsilon} + \lambda x_j^{(k-1)} \right)\end{aligned}$$

### Notes:

- The weight decay term  $\lambda x_j^{(k-1)}$  is added *after* the adaptive gradient step.
- Widely adopted in training transformers and other large models. Default choice for huggingface trainer.

# A lot of them

Rosenbrock Function.  
Adaptive stochastic gradient algorithms.  
Learning rate 0.003

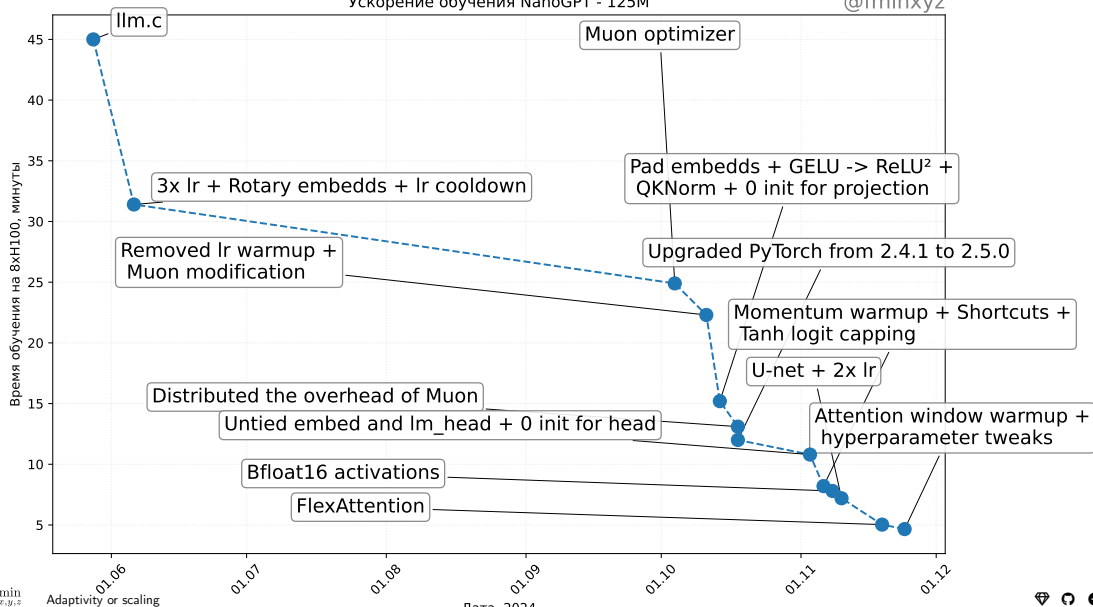


# How to compare them? AlgoPerf benchmark

# NanoGPT speedrun

Ускорение обучения NanoGPT - 125M

@fminxyz



## Shampoo (Gupta, Anil, et al., 2018; Anil et al., 2020)

Stands for **S**tochastic **H**essian-**A**pproximation **M**atrix **P**reconditioning for **O**ptimization **O**f deep networks. It's a method inspired by second-order optimization designed for large-scale deep learning.

**Core Idea:** Approximates the full-matrix AdaGrad pre conditioner using efficient matrix structures, specifically Kronecker products.

For a weight matrix  $W \in \mathbb{R}^{m \times n}$ , the update involves preconditioning using approximations of the statistics matrices  $L \approx \sum_k G_k G_k^T$  and  $R \approx \sum_k G_k^T G_k$ , where  $G_k$  are the gradients.

Simplified concept:

1. Compute gradient  $G_k$ .

**Notes:**

## Shampoo (Gupta, Anil, et al., 2018; Anil et al., 2020)

Stands for **S**tochastic **H**essian-**A**pproximation **M**atrix **P**reconditioning for **O**ptimization **O**f deep networks. It's a method inspired by second-order optimization designed for large-scale deep learning.

**Core Idea:** Approximates the full-matrix AdaGrad pre conditioner using efficient matrix structures, specifically Kronecker products.

For a weight matrix  $W \in \mathbb{R}^{m \times n}$ , the update involves preconditioning using approximations of the statistics matrices  $L \approx \sum_k G_k G_k^T$  and  $R \approx \sum_k G_k^T G_k$ , where  $G_k$  are the gradients.

Simplified concept:

1. Compute gradient  $G_k$ .
2. Update statistics  $L_k = \beta L_{k-1} + (1 - \beta) G_k G_k^T$  and  $R_k = \beta R_{k-1} + (1 - \beta) G_k^T G_k$ .

**Notes:**

## Shampoo (Gupta, Anil, et al., 2018; Anil et al., 2020)

Stands for **S**tochastic **H**essian-**A**pproximation **M**atrix **P**reconditioning for **O**ptimization **O**f deep networks. It's a method inspired by second-order optimization designed for large-scale deep learning.

**Core Idea:** Approximates the full-matrix AdaGrad pre conditioner using efficient matrix structures, specifically Kronecker products.

For a weight matrix  $W \in \mathbb{R}^{m \times n}$ , the update involves preconditioning using approximations of the statistics matrices  $L \approx \sum_k G_k G_k^T$  and  $R \approx \sum_k G_k^T G_k$ , where  $G_k$  are the gradients.

Simplified concept:

1. Compute gradient  $G_k$ .
2. Update statistics  $L_k = \beta L_{k-1} + (1 - \beta) G_k G_k^T$  and  $R_k = \beta R_{k-1} + (1 - \beta) G_k^T G_k$ .
3. Compute preconditioners  $P_L = L_k^{-1/4}$  and  $P_R = R_k^{-1/4}$ . (Inverse matrix root)

**Notes:**



## Shampoo (Gupta, Anil, et al., 2018; Anil et al., 2020)

Stands for **S**tochastic **H**essian-**A**pproximation **M**atrix **P**reconditioning for **O**ptimization **O**f deep networks. It's a method inspired by second-order optimization designed for large-scale deep learning.

**Core Idea:** Approximates the full-matrix AdaGrad pre conditioner using efficient matrix structures, specifically Kronecker products.

For a weight matrix  $W \in \mathbb{R}^{m \times n}$ , the update involves preconditioning using approximations of the statistics matrices  $L \approx \sum_k G_k G_k^T$  and  $R \approx \sum_k G_k^T G_k$ , where  $G_k$  are the gradients.

Simplified concept:

1. Compute gradient  $G_k$ .
2. Update statistics  $L_k = \beta L_{k-1} + (1 - \beta) G_k G_k^T$  and  $R_k = \beta R_{k-1} + (1 - \beta) G_k^T G_k$ .
3. Compute preconditioners  $P_L = L_k^{-1/4}$  and  $P_R = R_k^{-1/4}$ . (Inverse matrix root)
4. Update:  $W_{k+1} = W_k - \alpha P_L G_k P_R$ .

**Notes:**

## Shampoo (Gupta, Anil, et al., 2018; Anil et al., 2020)

Stands for **S**tochastic **H**essian-**A**pproximation **M**atrix **P**reconditioning for **O**ptimization **O**f deep networks. It's a method inspired by second-order optimization designed for large-scale deep learning.

**Core Idea:** Approximates the full-matrix AdaGrad pre conditioner using efficient matrix structures, specifically Kronecker products.

For a weight matrix  $W \in \mathbb{R}^{m \times n}$ , the update involves preconditioning using approximations of the statistics matrices  $L \approx \sum_k G_k G_k^T$  and  $R \approx \sum_k G_k^T G_k$ , where  $G_k$  are the gradients.

Simplified concept:

1. Compute gradient  $G_k$ .
2. Update statistics  $L_k = \beta L_{k-1} + (1 - \beta) G_k G_k^T$  and  $R_k = \beta R_{k-1} + (1 - \beta) G_k^T G_k$ .
3. Compute preconditioners  $P_L = L_k^{-1/4}$  and  $P_R = R_k^{-1/4}$ . (Inverse matrix root)
4. Update:  $W_{k+1} = W_k - \alpha P_L G_k P_R$ .

### Notes:

- Aims to capture curvature information more effectively than first-order methods.

## Shampoo (Gupta, Anil, et al., 2018; Anil et al., 2020)

Stands for **S**tochastic **H**essian-**A**pproximation **M**atrix **P**reconditioning for **O**ptimization **O**f deep networks. It's a method inspired by second-order optimization designed for large-scale deep learning.

**Core Idea:** Approximates the full-matrix AdaGrad pre conditioner using efficient matrix structures, specifically Kronecker products.

For a weight matrix  $W \in \mathbb{R}^{m \times n}$ , the update involves preconditioning using approximations of the statistics matrices  $L \approx \sum_k G_k G_k^T$  and  $R \approx \sum_k G_k^T G_k$ , where  $G_k$  are the gradients.

Simplified concept:

1. Compute gradient  $G_k$ .
2. Update statistics  $L_k = \beta L_{k-1} + (1 - \beta) G_k G_k^T$  and  $R_k = \beta R_{k-1} + (1 - \beta) G_k^T G_k$ .
3. Compute preconditioners  $P_L = L_k^{-1/4}$  and  $P_R = R_k^{-1/4}$ . (Inverse matrix root)
4. Update:  $W_{k+1} = W_k - \alpha P_L G_k P_R$ .

### Notes:

- Aims to capture curvature information more effectively than first-order methods.
- Computationally more expensive than Adam but can converge faster or to better solutions in terms of steps.

## Shampoo (Gupta, Anil, et al., 2018; Anil et al., 2020)

Stands for **S**tochastic **H**essian-**A**pproximation **M**atrix **P**reconditioning for **O**ptimization **O**f deep networks. It's a method inspired by second-order optimization designed for large-scale deep learning.

**Core Idea:** Approximates the full-matrix AdaGrad pre conditioner using efficient matrix structures, specifically Kronecker products.

For a weight matrix  $W \in \mathbb{R}^{m \times n}$ , the update involves preconditioning using approximations of the statistics matrices  $L \approx \sum_k G_k G_k^T$  and  $R \approx \sum_k G_k^T G_k$ , where  $G_k$  are the gradients.

Simplified concept:

1. Compute gradient  $G_k$ .
2. Update statistics  $L_k = \beta L_{k-1} + (1 - \beta) G_k G_k^T$  and  $R_k = \beta R_{k-1} + (1 - \beta) G_k^T G_k$ .
3. Compute preconditioners  $P_L = L_k^{-1/4}$  and  $P_R = R_k^{-1/4}$ . (Inverse matrix root)
4. Update:  $W_{k+1} = W_k - \alpha P_L G_k P_R$ .

### Notes:

- Aims to capture curvature information more effectively than first-order methods.
- Computationally more expensive than Adam but can converge faster or to better solutions in terms of steps.
- Requires careful implementation for efficiency (e.g., efficient computation of inverse matrix roots, handling large matrices).

## Shampoo (Gupta, Anil, et al., 2018; Anil et al., 2020)

Stands for **S**tochastic **H**essian-**A**pproximation **M**atrix **P**reconditioning for **O**ptimization **O**f deep networks. It's a method inspired by second-order optimization designed for large-scale deep learning.

**Core Idea:** Approximates the full-matrix AdaGrad pre conditioner using efficient matrix structures, specifically Kronecker products.

For a weight matrix  $W \in \mathbb{R}^{m \times n}$ , the update involves preconditioning using approximations of the statistics matrices  $L \approx \sum_k G_k G_k^T$  and  $R \approx \sum_k G_k^T G_k$ , where  $G_k$  are the gradients.

Simplified concept:

1. Compute gradient  $G_k$ .
2. Update statistics  $L_k = \beta L_{k-1} + (1 - \beta) G_k G_k^T$  and  $R_k = \beta R_{k-1} + (1 - \beta) G_k^T G_k$ .
3. Compute preconditioners  $P_L = L_k^{-1/4}$  and  $P_R = R_k^{-1/4}$ . (Inverse matrix root)
4. Update:  $W_{k+1} = W_k - \alpha P_L G_k P_R$ .

### Notes:

- Aims to capture curvature information more effectively than first-order methods.
- Computationally more expensive than Adam but can converge faster or to better solutions in terms of steps.
- Requires careful implementation for efficiency (e.g., efficient computation of inverse matrix roots, handling large matrices).
- Variants exist for different tensor shapes (e.g., convolutional layers).

$$\begin{aligned}W_{t+1} &= W_t - \eta(G_t G_t^\top)^{-1/4} G_t (G_t^\top G_t)^{-1/4} \\&= W_t - \eta(U S^2 U^\top)^{-1/4} (U S V^\top) (V S^2 V^\top)^{-1/4} \\&= W_t - \eta(U S^{-1/2} U^\top) (U S V^\top) (V S^{-1/2} V^\top) \\&= W_t - \eta U S^{-1/2} S S^{-1/2} V^\top \\&= W_t - \eta U V^\top\end{aligned}$$