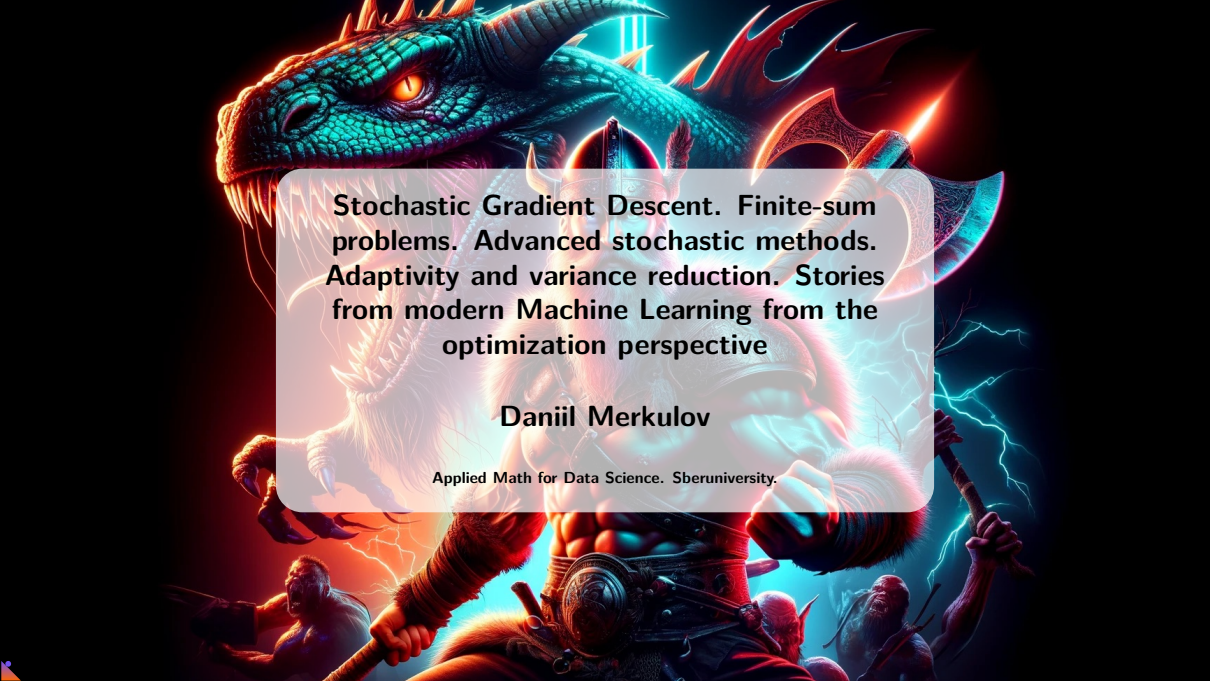


Stochastic Gradient Descent. Finite-sum problems. Advanced stochastic methods. Adaptivity and variance reduction. Stories from modern Machine Learning from the optimization perspective

Daniil Merkulov

Applied Math for Data Science. Sberuniversity.

Convergence rate reminder

Convergence rate

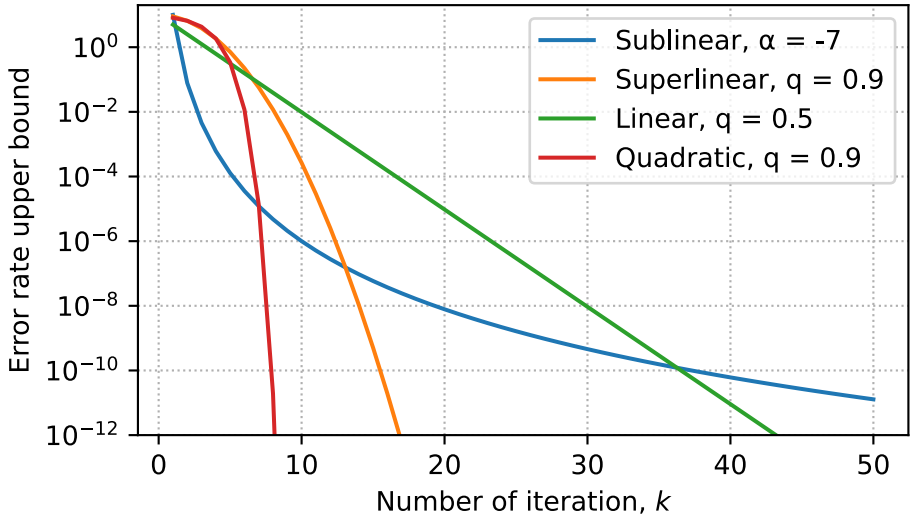


Figure 1: Difference between the convergence speed

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:

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

- Convergence with constant α 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:

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

- Convergence with constant α or line search.
- Iteration cost is linear in n. For ImageNet $n \approx 1.4 \cdot 10^7$, for WikiText $n \approx 10^8$.

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:

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

- Convergence with constant α or line search.
- Iteration cost is linear in n. For ImageNet $n \approx 1.4 \cdot 10^7$, for WikiText $n \approx 10^8$.

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:

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

- Convergence with constant α or line search.
- Iteration cost is linear in n. For ImageNet $n \approx 1.4 \cdot 10^7$, for WikiText $n \approx 10^8$.

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:

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

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 ∇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 ∇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 ∇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 ∇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?

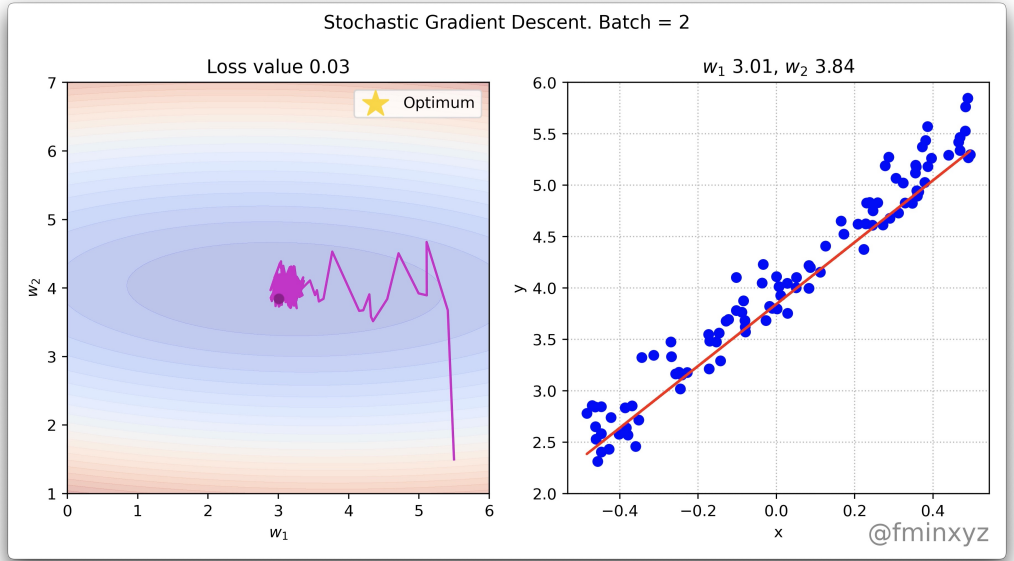
If ∇f is Lipschitz continuous then we have:

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).

Stochastic Gradient Descent (SGD)

Typical behaviour



Convergence

Lipschitz continiity implies:

$$f(x_{k+1}) \leq f(x_k) + \langle \nabla f(x_k), x_{k+1} - x_k \rangle + \frac{L}{2} \|x_{k+1} - x_k\|^2$$

Convergence

Lipschitz continiity implies:

$$f(x_{k+1}) \leq f(x_k) + \langle \nabla f(x_k), x_{k+1} - x_k \rangle + \frac{L}{2} \|x_{k+1} - x_k\|^2$$

using (SGD):

$$f(x_{k+1}) \leq f(x_k) - \alpha_k \langle \nabla f(x_k), \nabla f_{i_k}(x_k) \rangle + \alpha_k^2 \frac{L}{2} \|\nabla f_{i_k}(x_k)\|^2$$

Convergence

Lipschitz continiity implies:

$$f(x_{k+1}) \leq f(x_k) + \langle \nabla f(x_k), x_{k+1} - x_k \rangle + \frac{L}{2} \|x_{k+1} - x_k\|^2$$

using (SGD):

$$f(x_{k+1}) \leq f(x_k) - \alpha_k \langle \nabla f(x_k), \nabla f_{i_k}(x_k) \rangle + \alpha_k^2 \frac{L}{2} \|\nabla f_{i_k}(x_k)\|^2$$

Now let's take expectation with respect to i_k:

$$\mathbb{E}[f(x_{k+1})] \leq \mathbb{E}[f(x_k) - \alpha_k \langle \nabla f(x_k), \nabla f_{i_k}(x_k) \rangle + \alpha_k^2 \frac{L}{2} \|\nabla f_{i_k}(x_k)\|^2]$$

Convergence

Lipschitz continiity implies:

$$f(x_{k+1}) \leq f(x_k) + \langle \nabla f(x_k), x_{k+1} - x_k \rangle + \frac{L}{2}\|x_{k+1} - x_k\|^2$$

using (SGD):

$$f(x_{k+1}) \leq f(x_k) - \alpha_k \langle \nabla f(x_k), \nabla f_{i_k}(x_k) \rangle + \alpha_k^2 \frac{L}{2}\|\nabla f_{i_k}(x_k)\|^2$$

Now let's take expectation with respect to i_k:

$$\mathbb{E}[f(x_{k+1})] \leq \mathbb{E}[f(x_k) - \alpha_k \langle \nabla f(x_k), \nabla f_{i_k}(x_k) \rangle + \alpha_k^2 \frac{L}{2}\|\nabla f_{i_k}(x_k)\|^2]$$

Using linearity of expectation:

$$\mathbb{E}[f(x_{k+1})] \leq f(x_k) - \alpha_k \langle \nabla f(x_k), \mathbb{E}[\nabla f_{i_k}(x_k)] \rangle + \alpha_k^2 \frac{L}{2}\mathbb{E}[\|\nabla f_{i_k}(x_k)\|^2]$$

Convergence

Lipschitz continiity implies:

$$f(x_{k+1}) \leq f(x_k) + \langle \nabla f(x_k), x_{k+1} - x_k \rangle + \frac{L}{2}\|x_{k+1} - x_k\|^2$$

using (SGD):

$$f(x_{k+1}) \leq f(x_k) - \alpha_k \langle \nabla f(x_k), \nabla f_{i_k}(x_k) \rangle + \alpha_k^2 \frac{L}{2}\|\nabla f_{i_k}(x_k)\|^2$$

Now let's take expectation with respect to i_k:

$$\mathbb{E}[f(x_{k+1})] \leq \mathbb{E}[f(x_k) - \alpha_k \langle \nabla f(x_k), \nabla f_{i_k}(x_k) \rangle + \alpha_k^2 \frac{L}{2}\|\nabla f_{i_k}(x_k)\|^2]$$

Using linearity of expectation:

$$\mathbb{E}[f(x_{k+1})] \leq f(x_k) - \alpha_k \langle \nabla f(x_k), \mathbb{E}[\nabla f_{i_k}(x_k)] \rangle + \alpha_k^2 \frac{L}{2}\mathbb{E}[\|\nabla f_{i_k}(x_k)\|^2]$$

Since uniform sampling implies unbiased estimate of gradient: $\mathbb{E}[\nabla f_{i_k}(x_k)] = \nabla f(x_k)$:

$$\mathbb{E}[f(x_{k+1})] \leq f(x_k) - \alpha_k \|\nabla f(x_k)\|^2 + \alpha_k^2 \frac{L}{2}\mathbb{E}[\|\nabla f_{i_k}(x_k)\|^2]$$

Convergence. Smooth PL case.

$$\frac{1}{2}\|\nabla f(x)\|_2^2 \geq \mu(f(x) - f^*), \forall x \in \mathbb{R}^p \tag{PL}$$

Convergence. Smooth PL case.

$$\frac{1}{2}\|\nabla f(x)\|_2^2 \geq \mu(f(x) - f^*), \forall x \in \mathbb{R}^p \tag{PL}$$

This inequality simply requires that the gradient grows faster than a quadratic function as we move away from the optimal function value. Note, that strong convexity implies PL, but not vice versa. Using PL we can write:

$$\mathbb{E}[f(x_{k+1})] - f^* \leq (1 - 2\alpha_k\mu)[f(x_k) - f^*] + \alpha_k^2 \frac{L}{2}\mathbb{E}[\|\nabla f_{i_k}(x_k)\|^2]$$

Convergence. Smooth PL case.

$$\frac{1}{2}\|\nabla f(x)\|_2^2 \geq \mu(f(x) - f^*), \forall x \in \mathbb{R}^p \tag{PL}$$

This inequality simply requires that the gradient grows faster than a quadratic function as we move away from the optimal function value. Note, that strong convexity implies PL, but not vice versa. Using PL we can write:

$$\mathbb{E}[f(x_{k+1})] - f^* \leq (1 - 2\alpha_k\mu)[f(x_k) - f^*] + \alpha_k^2 \frac{L}{2}\mathbb{E}[\|\nabla f_{i_k}(x_k)\|^2]$$

This bound already indicates, that we have something like linear convergence if far from solution and gradients are similar, but no progress if close to solution or have high variance in gradients at the same time.

Convergence. Smooth PL case.

$$\frac{1}{2}\|\nabla f(x)\|_2^2 \geq \mu(f(x) - f^*), \forall x \in \mathbb{R}^p \tag{PL}$$

This inequality simply requires that the gradient grows faster than a quadratic function as we move away from the optimal function value. Note, that strong convexity implies PL, but not vice versa. Using PL we can write:

$$\mathbb{E}[f(x_{k+1})] - f^* \leq (1 - 2\alpha_k\mu)[f(x_k) - f^*] + \alpha_k^2 \frac{L}{2}\mathbb{E}[\|\nabla f_{i_k}(x_k)\|^2]$$

This bound already indicates, that we have something like linear convergence if far from solution and gradients are similar, but no progress if close to solution or have high variance in gradients at the same time.

Now we assume, that the variance of the stochastic gradients is bounded:

$$\mathbb{E}[\|\nabla f_i(x_k)\|^2] \leq \sigma^2$$

Convergence. Smooth PL case.

$$\frac{1}{2}\|\nabla f(x)\|_2^2 \geq \mu(f(x) - f^*), \forall x \in \mathbb{R}^p \tag{PL}$$

This inequality simply requires that the gradient grows faster than a quadratic function as we move away from the optimal function value. Note, that strong convexity implies PL, but not vice versa. Using PL we can write:

$$\mathbb{E}[f(x_{k+1})] - f^* \leq (1 - 2\alpha_k\mu)[f(x_k) - f^*] + \alpha_k^2 \frac{L}{2}\mathbb{E}[\|\nabla f_{i_k}(x_k)\|^2]$$

This bound already indicates, that we have something like linear convergence if far from solution and gradients are similar, but no progress if close to solution or have high variance in gradients at the same time.

Now we assume, that the variance of the stochastic gradients is bounded:

$$\mathbb{E}[\|\nabla f_i(x_k)\|^2] \leq \sigma^2$$

Thus, we have

$$\mathbb{E}[f(x_{k+1}) - f^*] \leq (1 - 2\alpha_k\mu)[f(x_k) - f^*] + \frac{L\sigma^2\alpha_k^2}{2}.$$

Convergence. Smooth PL case.

1. Consider **decreasing stepsize** strategy with $\alpha_k = \frac{2k+1}{2\mu(k+1)^2}$ we obtain

$$\mathbb{E}[f(x_{k+1}) - f^*] \leq \frac{k^2}{(k+1)^2}[f(x_k) - f^*] + \frac{L\sigma^2(2k+1)^2|}{8\mu^2(k+1)^4}$$

Convergence. Smooth PL case.

1. Consider **decreasing stepsize** strategy with $\alpha_k = \frac{2k+1}{2\mu(k+1)^2}$ we obtain

$$\mathbb{E}[f(x_{k+1}) - f^*] \leq \frac{k^2}{(k+1)^2}[f(x_k) - f^*] + \frac{L\sigma^2(2k+1)^2|}{8\mu^2(k+1)^4}$$

2. Multiplying both sides by $(k+1)^2$ and letting $\delta_f(k) \equiv k^2\mathbb{E}[f(x_k) - f^*]$ we get

$$\begin{aligned}\delta_f(k+1) &\leq \delta_f(k) + \frac{L\sigma^2(2k+1)^2}{8\mu^2(k+1)^2} \\ &\leq \delta_f(k) + \frac{L\sigma^2}{2\mu^2},\end{aligned}$$

Convergence. Smooth PL case.

1. Consider **decreasing stepsize** strategy with $\alpha_k = \frac{2k+1}{2\mu(k+1)^2}$ we obtain

$$\mathbb{E}[f(x_{k+1}) - f^*] \leq \frac{k^2}{(k+1)^2}[f(x_k) - f^*] + \frac{L\sigma^2(2k+1)^2|}{8\mu^2(k+1)^4}$$

2. Multiplying both sides by $(k+1)^2$ and letting $\delta_f(k) \equiv k^2\mathbb{E}[f(x_k) - f^*]$ we get

$$\begin{aligned}\delta_f(k+1) &\leq \delta_f(k) + \frac{L\sigma^2(2k+1)^2}{8\mu^2(k+1)^2} \\ &\leq \delta_f(k) + \frac{L\sigma^2}{2\mu^2},\end{aligned}$$

Convergence. Smooth PL case.

1. Consider **decreasing stepsize** strategy with $\alpha_k = \frac{2k+1}{2\mu(k+1)^2}$ we obtain

$$\mathbb{E}[f(x_{k+1}) - f^*] \leq \frac{k^2}{(k+1)^2}[f(x_k) - f^*] + \frac{L\sigma^2(2k+1)^2|}{8\mu^2(k+1)^4}$$

2. Multiplying both sides by $(k+1)^2$ and letting $\delta_f(k) \equiv k^2\mathbb{E}[f(x_k) - f^*]$ we get

$$\begin{aligned}\delta_f(k+1) &\leq \delta_f(k) + \frac{L\sigma^2(2k+1)^2}{8\mu^2(k+1)^2} \\ &\leq \delta_f(k) + \frac{L\sigma^2}{2\mu^2},\end{aligned}$$

where the second line follows from $\frac{2k+1}{k+1} < 2$. Summing up this inequality from $k = 0$ to k and using the fact that $\delta_f(0) = 0$ we get

$$\delta_f(k+1) \leq \delta_f(0) + \frac{L\sigma^2}{2\mu^2}\sum_{i=0}^{k} 1 \leq \frac{L\sigma^2(k+1)}{2\mu^2} \Rightarrow \quad (k+1)^2\mathbb{E}[f(x_{k+1}) - f^*] \leq \frac{L\sigma^2(k+1)}{2\mu^2}$$

which gives the stated rate.

Convergence. Smooth PL case.

3. **Constant step size**: Choosing $\alpha_k = \alpha$ for any $\alpha < 1/2\mu$ yields

$$\begin{aligned}\mathbb{E}[f(x_{k+1}) - f^*] &\leq (1 - 2\alpha\mu)^k [f(x_0) - f^*] + \frac{L\sigma^2\alpha^2}{2} \sum_{i=0}^{k} (1 - 2\alpha\mu)^i \\ &\leq (1 - 2\alpha\mu)^k [f(x_0) - f^*] + \frac{L\sigma^2\alpha^2}{2} \sum_{i=0}^{\infty} (1 - 2\alpha\mu)^i \\ &= (1 - 2\alpha\mu)^k [f(x_0) - f^*] + \frac{L\sigma^2\alpha}{4\mu},\end{aligned}$$

where the last line uses that $\alpha < 1/2\mu$ and the limit of the geometric series.

Mini-batch SGD

Mini-batch SGD

Approach 1: Control the sample size

The deterministic method uses all n gradients:

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

The stochastic method approximates this using just 1 sample:

$$\nabla f_{ik}(x_k) \approx \frac{1}{n} \sum_{i=1}^{n} \nabla f_i(x_k).$$

A common variant is to use a larger sample B_k ("mini-batch"):

$$\frac{1}{|B_k|} \sum_{i \in B_k} \nabla f_i(x_k) \approx \frac{1}{n} \sum_{i=1}^{n} \nabla f_i(x_k),$$

particularly useful for vectorization and parallelization.
For example, with 16 cores set $|B_k| = 16$ and compute 16 gradients at once.

Mini-Batching as Gradient Descent with Error

The SG method with a sample B_k ("mini-batch") uses iterations:

$$x_{k+1} = x_k - \alpha_k \left(\frac{1}{|B_k|} \sum_{i \in B_k} \nabla f_i(x_k) \right).$$

Let's view this as a "gradient method with error":

$$x_{k+1} = x_k - \alpha_k (\nabla f(x_k) + e_k),$$

where e_k is the difference between the approximate and true gradient.

If you use $\alpha_k = \frac{1}{L}$, then using the descent lemma, this algorithm has:

$$f(x_{k+1}) \leq f(x_k) - \frac{1}{2L} \|\nabla f(x_k)\|^2 + \frac{1}{2L} \|e_k\|^2,$$

for any error e_k.

Effect of Error on Convergence Rate

Our progress bound with $\alpha_k = \frac{1}{L}$ and error in the gradient of e_k is:

$$f(x_{k+1}) \leq f(x_k) - \frac{1}{2L}\|\nabla f(x_k)\|^2 + \frac{1}{2L}\|e_k\|^2.$$

Connection between "error-free" rate and "with error" rate:

- If the "error-free" rate is $O(\frac{1}{k})$, you maintain this rate if $\|e_k\|^2 = O(\frac{1}{k})$.

Effect of Error on Convergence Rate

Our progress bound with $\alpha_k = \frac{1}{L}$ and error in the gradient of e_k is:

$$f(x_{k+1}) \leq f(x_k) - \frac{1}{2L}\|\nabla f(x_k)\|^2 + \frac{1}{2L}\|e_k\|^2.$$

Connection between "error-free" rate and "with error" rate:

- If the "error-free" rate is $O(\frac{1}{k})$, you maintain this rate if $\|e_k\|^2 = O(\frac{1}{k})$.
- If the "error-free" rate is $O(\rho^k)$, you maintain this rate if $\|e_k\|^2 = O(\rho^k)$.

Effect of Error on Convergence Rate

Our progress bound with $\alpha_k = \frac{1}{L}$ and error in the gradient of e_k is:

$$f(x_{k+1}) \leq f(x_k) - \frac{1}{2L}\|\nabla f(x_k)\|^2 + \frac{1}{2L}\|e_k\|^2.$$

Connection between "error-free" rate and "with error" rate:

- If the "error-free" rate is $O(\frac{1}{k})$, you maintain this rate if $\|e_k\|^2 = O(\frac{1}{k})$.
- If the "error-free" rate is $O(\rho^k)$, you maintain this rate if $\|e_k\|^2 = O(\rho^k)$.

Effect of Error on Convergence Rate

Our progress bound with $\alpha_k = \frac{1}{L}$ and error in the gradient of e_k is:

$$f(x_{k+1}) \leq f(x_k) - \frac{1}{2L}\|\nabla f(x_k)\|^2 + \frac{1}{2L}\|e_k\|^2.$$

Connection between "error-free" rate and "with error" rate:

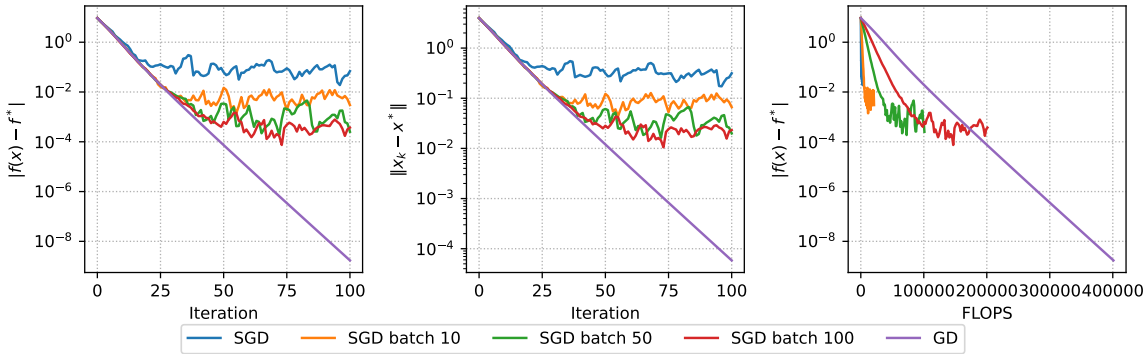
- If the "error-free" rate is $O(\frac{1}{k})$, you maintain this rate if $\|e_k\|^2 = O(\frac{1}{k})$.
- If the "error-free" rate is $O(\rho^k)$, you maintain this rate if $\|e_k\|^2 = O(\rho^k)$.

If the error goes to zero more slowly, then the rate at which it goes to zero becomes the bottleneck.
So, to understand the effect of batch size, we need to know how $|B_k|$ affects $\|e_k\|^2$.

Main problem of SGD

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

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



Conclusions

- SGD with fixed learning rate does not converge even for PL (strongly convex) case

Conclusions

- SGD with fixed learning rate does not converge even for PL (strongly convex) case
- SGD achieves sublinear convergence with rate $\mathcal{O}\left(\frac{1}{k}\right)$ for PL-case.

Conclusions

- SGD with fixed learning rate does not converge even for PL (strongly convex) case
- SGD achieves sublinear convergence with rate $\mathcal{O}\left(\frac{1}{k}\right)$ for PL-case.
- Nesterov/Polyak accelerations do not improve convergence rate

Conclusions

- SGD with fixed learning rate does not converge even for PL (strongly convex) case
- SGD achieves sublinear convergence with rate $\mathcal{O}\left(\frac{1}{k}\right)$ for PL-case.
- Nesterov/Polyak accelerations do not improve convergence rate
- Two-phase Newton-like method achieves $\mathcal{O}\left(\frac{1}{k}\right)$ without strong convexity.

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:

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

- 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:

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

- Iteration cost is linear in n.
- Convergence with constant α 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:

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

- Iteration cost is linear in n.
- Convergence with constant α 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:

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

- Iteration cost is linear in n.
- Convergence with constant α 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:

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

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 ∇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 ∇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 ∇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 ∇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

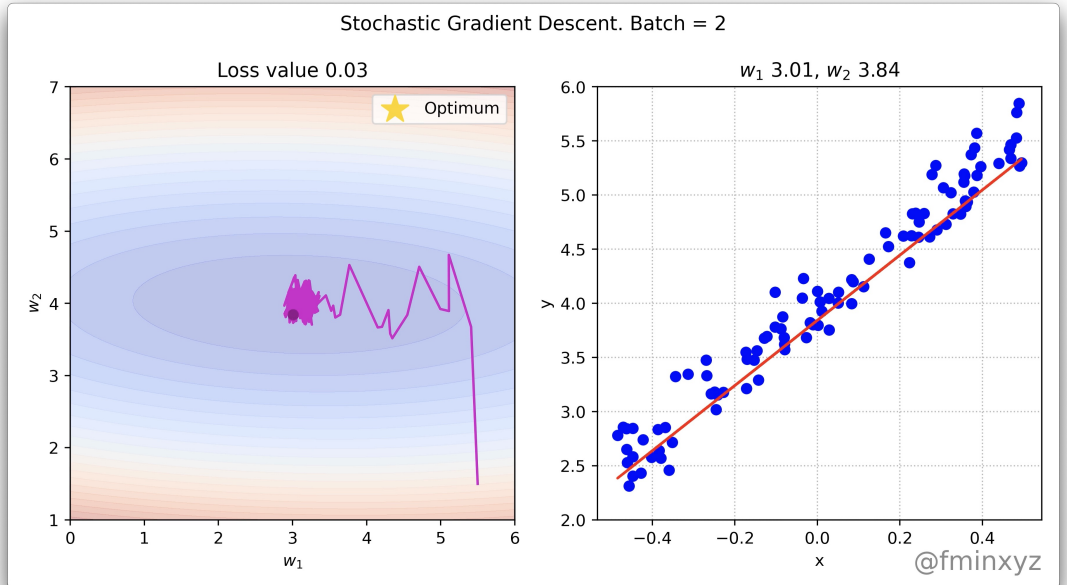
Stochastic iterations are n times faster, but how many iterations are needed?

If ∇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).

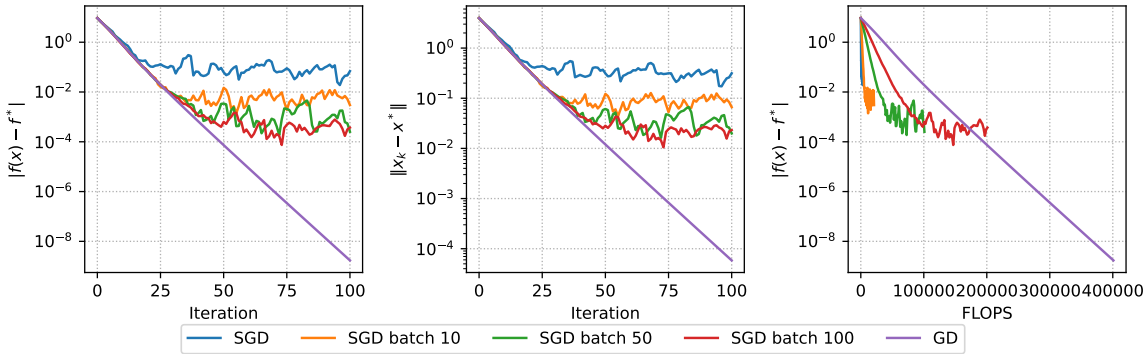
SGD with constant stepsize does not converge



Main problem of SGD

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

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:

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

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

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:

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

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

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

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

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

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

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

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

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

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

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

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

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

- $\mathbb{E}[Z_\alpha] = \alpha\mathbb{E}[X] + (1-\alpha)\mathbb{E}[Y]$
- $\mathrm{var}(Z_\alpha) = \alpha^2\left(\mathrm{var}(X) + \mathrm{var}(Y) - 2\mathrm{cov}(X,Y)\right)$
  - 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, $i = 1, \ldots, n$

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

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

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

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

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

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

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

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

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

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

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

  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:

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

SAG convergence

Assume that $f(x) = \frac{1}{n}\sum_{i=1}^{n} f_i(x)$, where each f_i is differentiable, and ∇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

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

satisfies

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

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: $\frac{L\|x^{(0)}-x^\star\|^2}{2k}$

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: $\frac{L\|x^{(0)}-x^\star\|^2}{2k}$
  - SAG: $\frac{48n[f(x^{(0)})-f^\star]+128L\|x^{(0)}-x^\star\|^2}{k}$

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: $\frac{L\|x^{(0)}-x^\star\|^2}{2k}$
  - SAG: $\frac{48n[f(x^{(0)})-f^\star]+128L\|x^{(0)}-x^\star\|^2}{k}$
- So the first term in SAG bound suffers from a factor of n; authors suggest smarter initialization to make $f(x^{(0)}) - f^\star$ 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 μ.

Theorem

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

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

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 μ.

Theorem

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

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

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.
- Like GD, we say SAG is adaptive to strong convexity.

SAG convergence

Assume further that each f_i is strongly convex with parameter μ.

Theorem

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

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

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.
- 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 $g(x^k) \to \nabla f(x^k)$ 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 $g(x^k) \to \nabla f(x^k)$ 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 α_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 α_k and the convergence rate of the method are determined by the constant L for $f(x)$, where $L = \max_{1 \le i \le n} L_i$, 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 α_k and the convergence rate of the method are determined by the constant L for $f(x)$, where $L = \max_{1 \le i \le n} L_i$, 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 α_k and the convergence rate of the method are determined by the constant L for $f(x)$, where $L = \max_{1 \le i \le n} L_i$, 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)`
    - $x_t = x_{t-1} - \alpha \left[\nabla f(\tilde{x}) + \left(\nabla f_{i_t}(x_{t-1}) - \nabla f_{i_t}(\tilde{x}) \right) \right]$

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)`
    - $x_t = x_{t-1} - \alpha \left[\nabla f(\tilde{x}) + \left(\nabla f_{i_t}(x_{t-1}) - \nabla f_{i_t}(\tilde{x}) \right) \right]$
    - Update $\tilde{x} = x_t$

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)`
    - $x_t = x_{t-1} - \alpha \left[\nabla f(\tilde{x}) + \left(\nabla f_{i_t}(x_{t-1}) - \nabla f_{i_t}(\tilde{x}) \right) \right]$
    - Update $\tilde{x} = x_t$

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)`
    - $x_t = x_{t-1} - \alpha \left[\nabla f(\tilde{x}) + \left(\nabla f_{i_t}(x_{t-1}) - \nabla f_{i_t}(\tilde{x}) \right) \right]$
    - Update $\tilde{x} = x_t$

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)`
    - $x_t = x_{t-1} - \alpha \left[\nabla f(\tilde{x}) + \left(\nabla f_{i_t}(x_{t-1}) - \nabla f_{i_t}(\tilde{x}) \right) \right]$
    - Update $\tilde{x} = x_t$

Notes:

- Two gradient evaluations per inner step.
- Two parameters: length of epochs + step-size γ.

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)`
    - $x_t = x_{t-1} - \alpha \left[\nabla f(\tilde{x}) + \left(\nabla f_{i_t}(x_{t-1}) - \nabla f_{i_t}(\tilde{x}) \right) \right]$
    - Update $\tilde{x} = x_t$

Notes:

- Two gradient evaluations per inner step.
- Two parameters: length of epochs + step-size γ.
- 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 $j = 1, \ldots, p$:

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

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

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 $j = 1, \ldots, p$:

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

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

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 $j = 1, \ldots, p$:

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

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

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 $j = 1, \ldots, p$:

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

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

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 $j = 1, \ldots, p$:

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

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

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 ϵ 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 $j = 1, \ldots, p$:

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

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

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 $j = 1, \ldots, p$:

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

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

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 $j = 1, \ldots, p$:

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

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

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 α:

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

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

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

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

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 α:

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

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

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

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

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 α:

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

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

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

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

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)

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

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

Notes:

- Adam is suitable for large datasets and high-dimensional optimization problems.

Adam (Kingma and Ba, 2014)

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

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

Notes:

- Adam is suitable for large datasets and high-dimensional optimization problems.
- 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)

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

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

Notes:

- Adam is suitable for large datasets and high-dimensional optimization problems.
- It corrects the bias towards zero in the initial moments seen in other methods like RMSProp, making the estimates more accurate.
- Highly popular in training deep learning models, owing to its efficiency and straightforward implementation.

Adam (Kingma and Ba, 2014)

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

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

Notes:

- Adam is suitable for large datasets and high-dimensional optimization problems.
- It corrects the bias towards zero in the initial moments seen in other methods like RMSProp, making the estimates more accurate.
- Highly popular in training deep learning models, owing to its efficiency and straightforward implementation.
- However, the proposed algorithm in initial version does not converge even in convex setting (later fixes appeared)

General introduction

Optimization for Neural Network training

Neural network is a function, that takes an input x and current set of weights (parameters) $\mathbf{w}$ and predicts some vector as an output. Note, that a variety of feed-forward neural networks could be represented as a series of linear transformations, followed by some nonlinear function (say, ReLU (x) or sigmoid):

Optimization for Neural Network training

Neural network is a function, that takes an input x and current set of weights (parameters) $\mathbf{w}$ and predicts some vector as an output. Note, that a variety of feed-forward neural networks could be represented as a series of linear transformations, followed by some nonlinear function (say, ReLU (x) or sigmoid):

$$\mathcal{NN}(\mathbf{w}, x) = \sigma_L \circ w_L \circ \ldots \circ \sigma_1 \circ w_1 \circ x \qquad \mathbf{w} = (W_1, b_1, \ldots W_L, b_L)\,,$$

Optimization for Neural Network training

Neural network is a function, that takes an input x and current set of weights (parameters) $\mathbf{w}$ and predicts some vector as an output. Note, that a variety of feed-forward neural networks could be represented as a series of linear transformations, followed by some nonlinear function (say, ReLU (x) or sigmoid):

$$\mathcal{NN}(\mathbf{w}, x) = \sigma_L \circ w_L \circ \ldots \circ \sigma_1 \circ w_1 \circ x \qquad \mathbf{w} = (W_1, b_1, \ldots W_L, b_L)\,,$$

where L is the number of layers, σ_i - non-linear activation function, $w_i = W_i x + b_i$ - linear layer.

Optimization for Neural Network training

Neural network is a function, that takes an input x and current set of weights (parameters) $\mathbf{w}$ and predicts some vector as an output. Note, that a variety of feed-forward neural networks could be represented as a series of linear transformations, followed by some nonlinear function (say, ReLU (x) or sigmoid):

$$\mathcal{NN}(\mathbf{w}, x) = \sigma_L \circ w_L \circ \ldots \circ \sigma_1 \circ w_1 \circ x \qquad \mathbf{w} = (W_1, b_1, \ldots W_L, b_L)\,,$$

where L is the number of layers, σ_i - non-linear activation function, $w_i = W_i x + b_i$ - linear layer.

Typically, we aim to find $\mathbf{w}$ in order to solve some problem (let say to be $\mathcal{NN}(\mathbf{w}, x_i) \sim y_i$ for some training data x_i, y_i). In order to do it, we solve the optimization problem:

Optimization for Neural Network training

Neural network is a function, that takes an input x and current set of weights (parameters) $\mathbf{w}$ and predicts some vector as an output. Note, that a variety of feed-forward neural networks could be represented as a series of linear transformations, followed by some nonlinear function (say, ReLU (x) or sigmoid):

$$\mathcal{NN}(\mathbf{w}, x) = \sigma_L \circ w_L \circ \ldots \circ \sigma_1 \circ w_1 \circ x \qquad \mathbf{w} = (W_1, b_1, \ldots W_L, b_L)\,,$$

where L is the number of layers, σ_i - non-linear activation function, $w_i = W_i x + b_i$ - linear layer.

Typically, we aim to find $\mathbf{w}$ in order to solve some problem (let say to be $\mathcal{NN}(\mathbf{w}, x_i) \sim y_i$ for some training data x_i, y_i). In order to do it, we solve the optimization problem:

$$L(\mathbf{w}, X, y) \to \min_{\mathbf{w}} \qquad \frac{1}{N}\sum_{i=1}^{N} l(\mathbf{w}, x_i, y_i) \to \min_{\mathbf{w}}$$

Loss functions

In the context of training neural networks, the loss function, denoted by $l(\mathbf{w}, x_i, y_i)$, measures the discrepancy between the predicted output $\mathcal{NN}(\mathbf{w}, x_i)$ and the true output y_i. The choice of the loss function can significantly influence the training process. Common loss functions include:

Mean Squared Error (MSE)

Used primarily for regression tasks. It computes the square of the difference between predicted and true values, averaged over all samples.

$$\text{MSE}(\mathbf{w}, X, y) = \frac{1}{N} \sum_{i=1}^{N} (\mathcal{NN}(\mathbf{w}, x_i) - y_i)^2$$

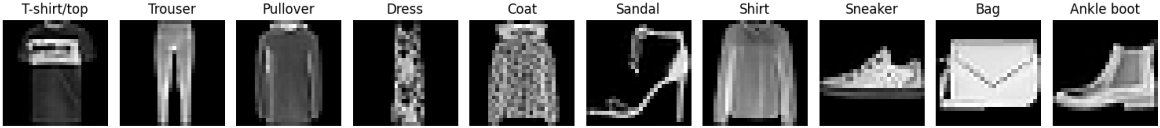
Cross-Entropy Loss

Typically used for classification tasks. It measures the dissimilarity between the true label distribution and the predictions, providing a probabilistic interpretation of classification.

$$\text{Cross-Entropy}(\mathbf{w}, X, y) = -\frac{1}{N} \sum_{i=1}^{N} \sum_{c=1}^{C} y_{i,c} \log(\mathcal{NN}(\mathbf{w}, x_i)_c)$$

where $y_{i,c}$ is a binary indicator (0 or 1) if class label c is the correct classification for observation i, and C is the number of classes.

Simple example: Fashion MNIST classification problem



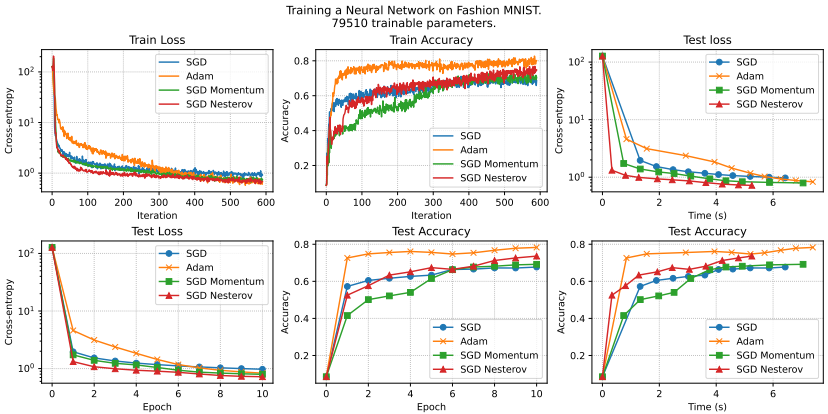


Figure 3: Open in colab

Loss surface of Neural Networks

Visualizing loss surface of neural network via line projection

We denote the initial point as w_0, representing the weights of the neural network at initialization. The weights after training are denoted as $\hat{w}$.

Initially, we generate a random Gaussian direction $w_1 \in \mathbb{R}^p$, which inherits the magnitude of the original neural network weights for each parameter group. Subsequently, we sample the training and testing loss surfaces at points along the direction w_1, situated close to either w_0 or $\hat{w}$.

Mathematically, this involves evaluating:

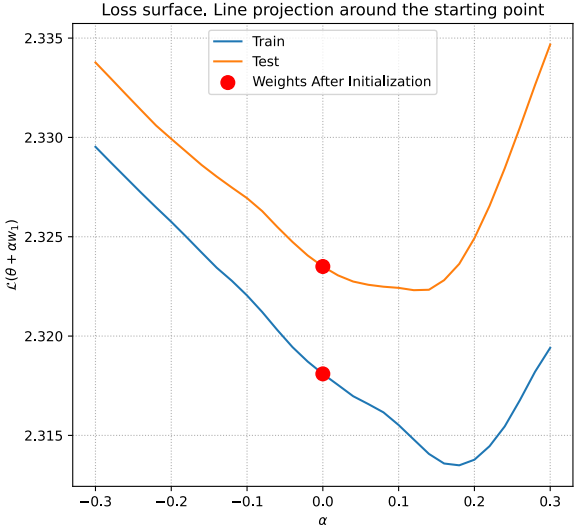
$$L(\alpha) = L(w_0 + \alpha w_1), \text{ where } \alpha \in [-b, b].$$

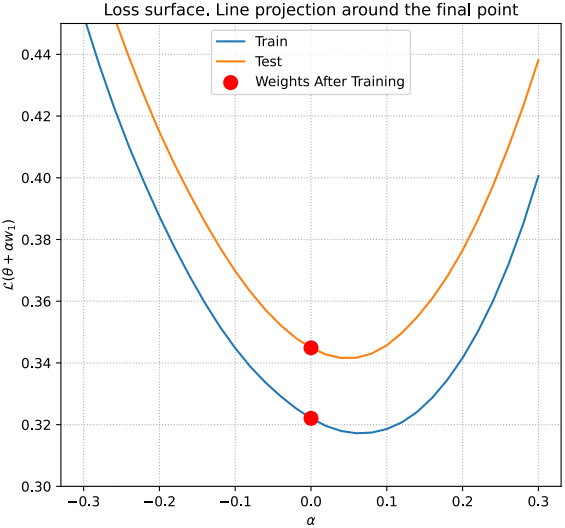
Here, α plays the role of a coordinate along the w_1 direction, and b stands for the bounds of interpolation. Visualizing $L(\alpha)$ enables us to project the p-dimensional surface onto a one-dimensional axis.

It is important to note that the characteristics of the resulting graph heavily rely on the chosen projection direction. It's not feasible to maintain the entirety of the information when transforming a space with 100,000 dimensions into a one-dimensional line through projection. However, certain properties can still be established. For instance, if $L(\alpha)\,|_{\alpha=0}$ is decreasing, this indicates that the point lies on a slope. Additionally, if the projection is non-convex, it implies that the original surface was not convex.

Visualizing loss surface of neural network

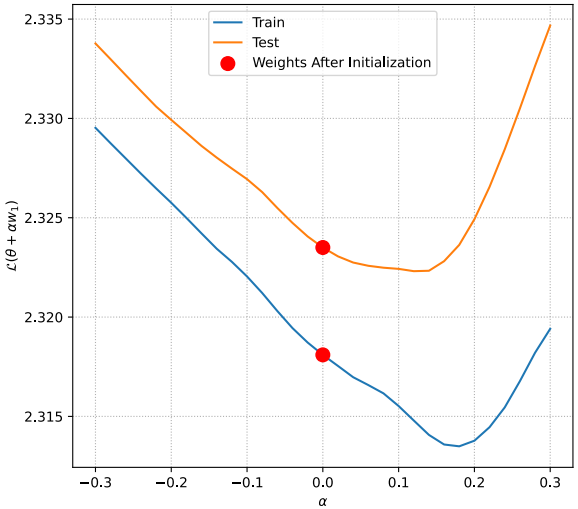
No Dropout

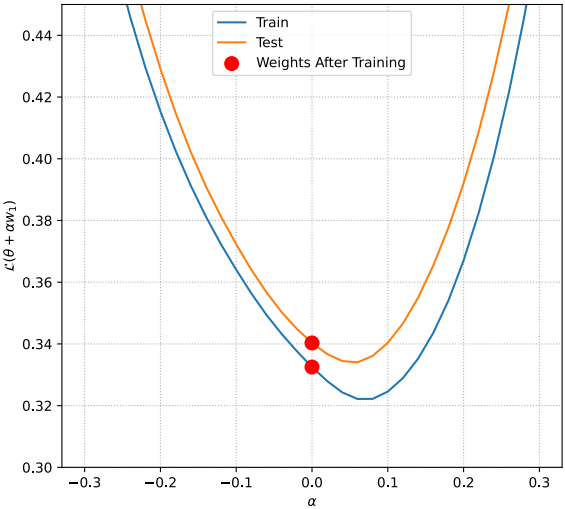




Visualizing loss surface of neural network

Dropout 0.2





Plane projection

We can explore this idea further and draw the projection of the loss surface to the plane, which is defined by 2 random vectors. Note, that with 2 random gaussian vectors in the huge dimensional space are almost certainly orthogonal. So, as previously, we generate random normalized gaussian vectors $w_1, w_2 \in \mathbb{R}^p$ and evaluate the loss function

$$L(\alpha, \beta) = L(w_0 + \alpha w_1 + \beta w_2), \text{ where } \alpha, \beta \in [-b, b]^2.$$

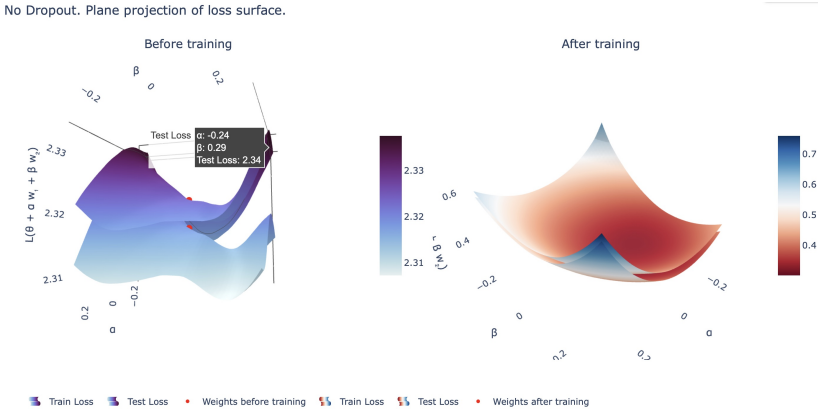


Figure 6: Open in colab

Can plane projections be useful? [1]

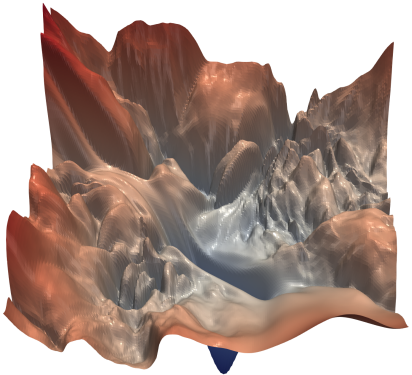


Figure 7: The loss surface of ResNet-56 without skip connections

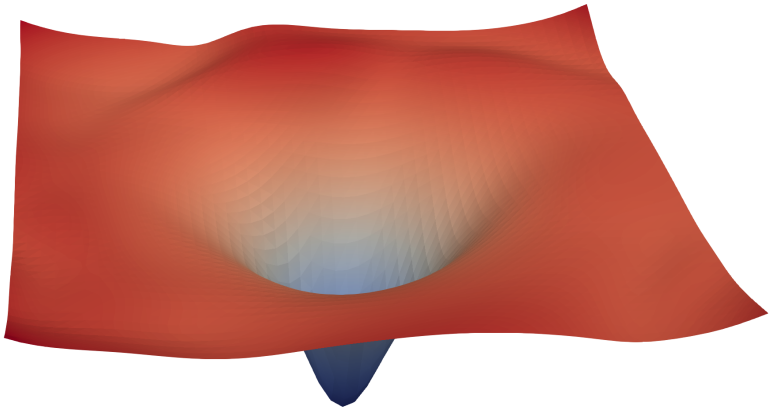


Figure 8: The loss surface of ResNet-56 with skip connections

[1] Visualizing the Loss Landscape of Neural Nets, Hao Li, Zheng Xu, Gavin Taylor, Christoph Studer, Tom Goldstein

Can plane projections be useful, really? [2]

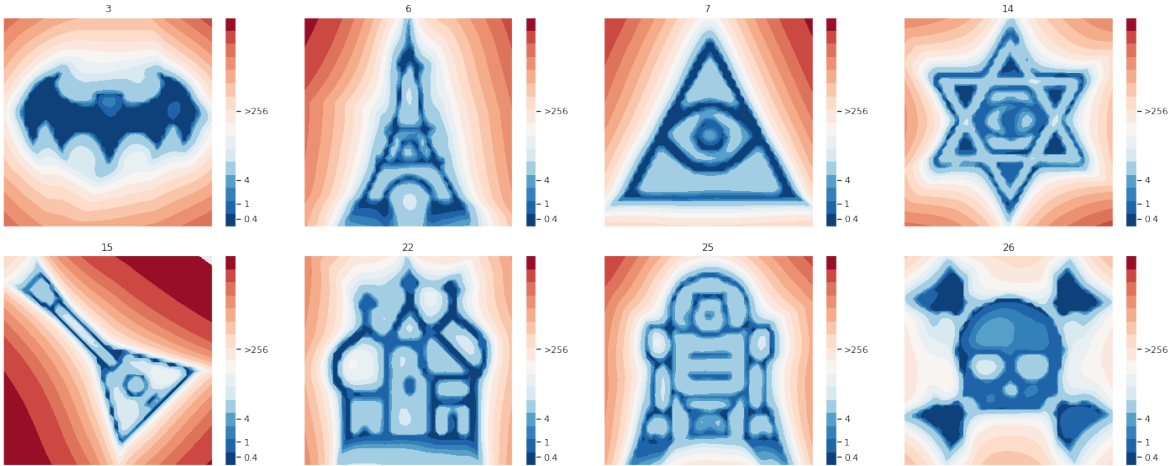


Figure 9: Examples of a loss landscape of a typical CNN model on FashionMNIST and CIFAR10 datasets found with MPO. Loss values are color-coded according to a logarithmic scale

[2] Loss Landscape Sightseeing with Multi-Point Optimization, Ivan Skorokhodov, Mikhail Burtsev

Impact of initialization [3]

Properly initializing a NN important. NN loss is highly nonconvex; optimizing it to attain a "good" solution hard, requires careful tuning.

- Don't initialize all weights to be the same — why?

Impact of initialization [3]

Properly initializing a NN important. NN loss is highly nonconvex; optimizing it to attain a "good" solution hard, requires careful tuning.

- Don't initialize all weights to be the same — why?
- Random: Initialize randomly, e.g., via the Gaussian $N(0, \sigma^2)$, where std σ depends on the number of neurons in a given layer. *Symmetry breaking*.

Impact of initialization [3]

Properly initializing a NN important. NN loss is highly nonconvex; optimizing it to attain a "good" solution hard, requires careful tuning.

- Don't initialize all weights to be the same — why?
- Random: Initialize randomly, e.g., via the Gaussian $N(0, \sigma^2)$, where std σ depends on the number of neurons in a given layer. *Symmetry breaking*.
- One can find more useful advices here

[3] On the importance of initialization and momentum in deep learning Ilya Sutskever, James Martens, George Dahl, Geoffrey Hinton

Impact of initialization [4]

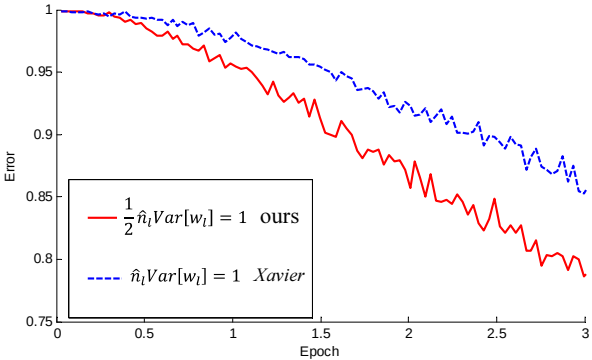


Figure 10: 22-layer ReLU net: good init converges faster

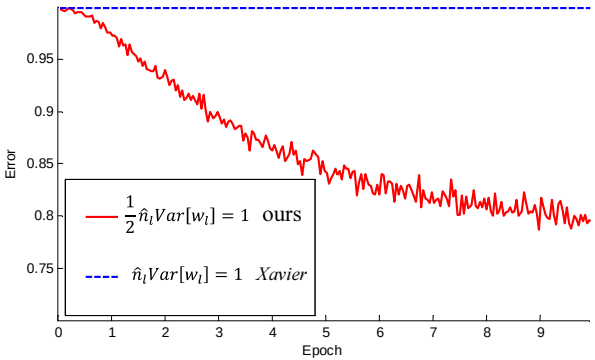


Figure 11: 30-layer ReLU net: good init is able to converge

[4] Delving Deep into Rectifiers: Surpassing Human-Level Performance on ImageNet Classification, Kaiming He, Xiangyu Zhang, Shaoqing Ren, Jian Sun

Grokking [5]

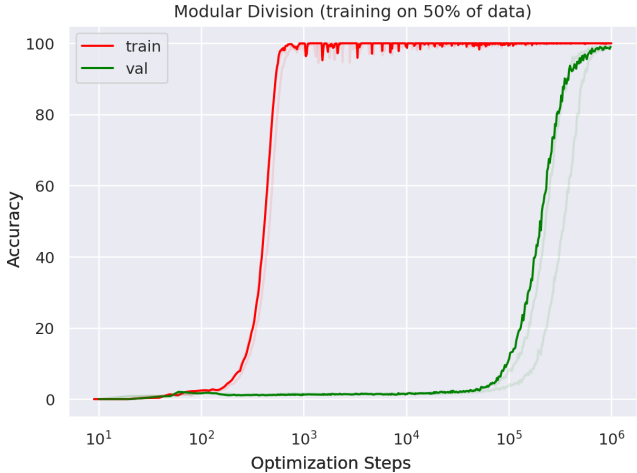
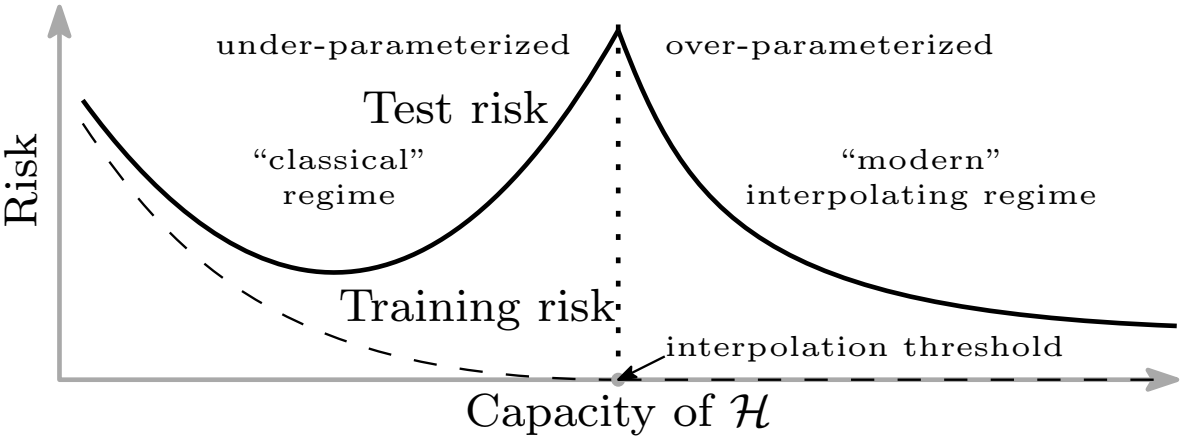


Figure 12: Training transformer with 2 layers, width 128, and 4 attention heads, with a total of about $4 \cdot 10^5$ non-embedding parameters. Reproduction of experiments ($\sim$ half an hour) is available here

[5] Grokking: Generalization Beyond Overfitting on Small Algorithmic Datasets, Alethea Power, Yuri Burda, Harri Edwards, Igor Babuschkin, Vedant Misra

Double Descent [6]



[6] Reconciling modern machine learning practice and the bias-variance trade-off, Mikhail Belkin, Daniel Hsu, Siyuan Ma, Soumik Mandal

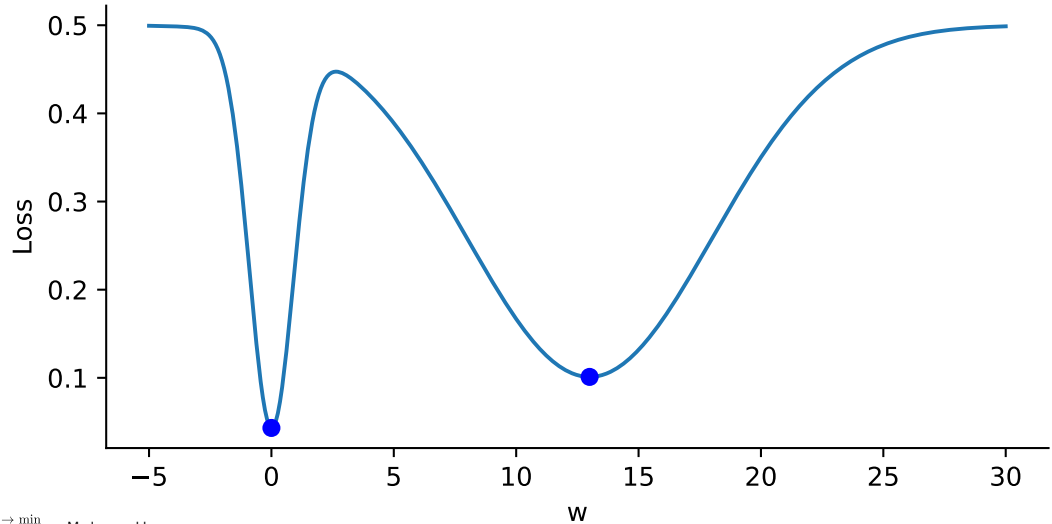
Exponential learning rate

- Exponential Learning Rate Schedules for Deep Learning

Modern problems

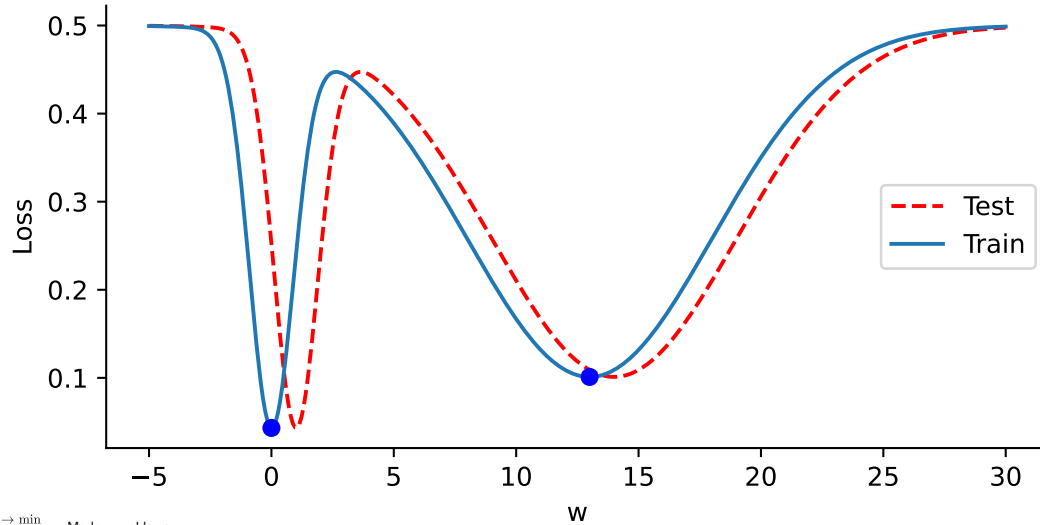
Wide vs narrow local minima

Узкие и широкие локальные минимумы



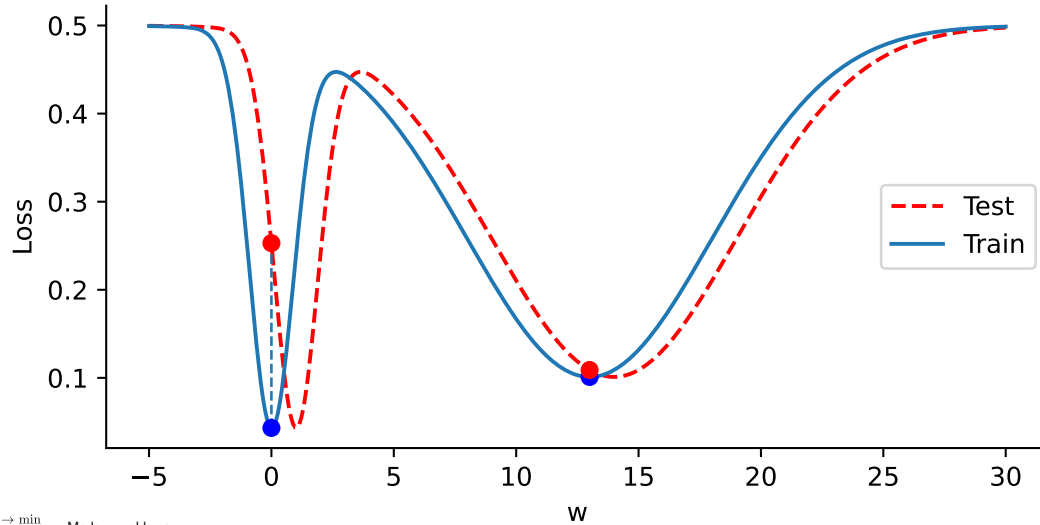
Wide vs narrow local minima

Узкие и широкие локальные минимумы



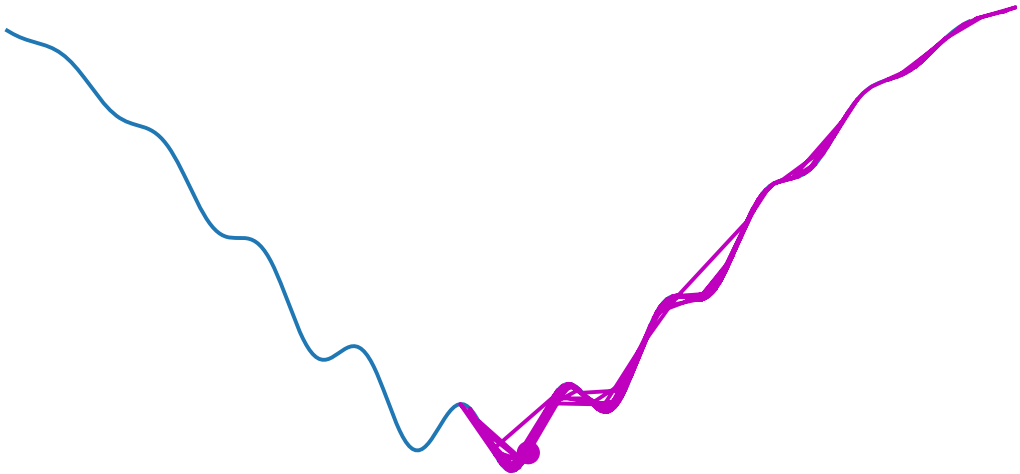
Wide vs narrow local minima

Узкие и широкие локальные минимумы



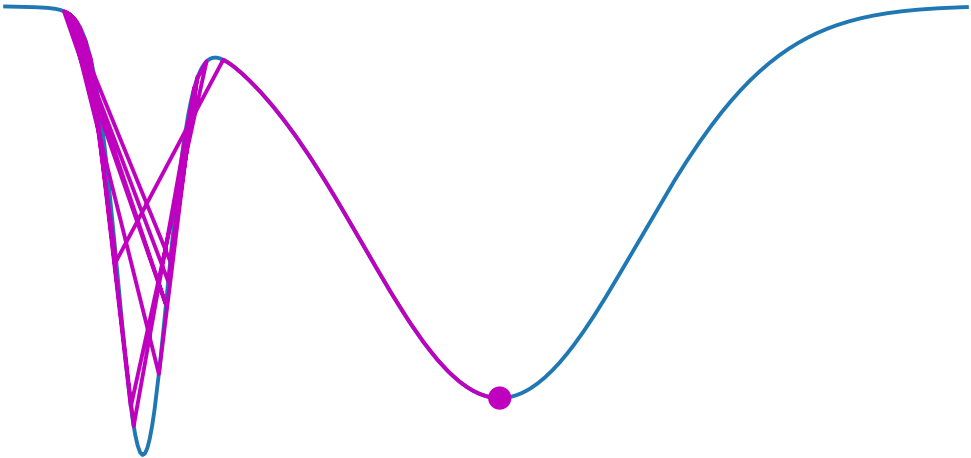
Stochasticity allows to escape local minima

Стохастический градиентный спуск
выпрыгивает из локальных минимумов



Local divergence can also be benefitial

Градиентный спуск с большим шагом
избегает узкого локального минимума



Automatic Differentiation stories

Gradient Vanishing/Exploding

- Multiplication of a chain of matrices in backprop

Gradient Vanishing/Exploding

- Multiplication of a chain of matrices in backprop
- If several of these matrices are "small" (i.e., norms < 1), when we multiply them, the gradient will decrease exponentially fast and tend to vanish (hurting learning in lower layers much more)

Gradient Vanishing/Exploding

- Multiplication of a chain of matrices in backprop
- If several of these matrices are "small" (i.e., norms < 1), when we multiply them, the gradient will decrease exponentially fast and tend to vanish (hurting learning in lower layers much more)
- Conversely, if several matrices have large norm, the gradient will tend to explode. In both cases, the gradients are unstable.

Gradient Vanishing/Exploding

- Multiplication of a chain of matrices in backprop
- If several of these matrices are "small" (i.e., norms < 1), when we multiply them, the gradient will decrease exponentially fast and tend to vanish (hurting learning in lower layers much more)
- Conversely, if several matrices have large norm, the gradient will tend to explode. In both cases, the gradients are unstable.
- Coping with unstable gradients poses several challenges, and must be dealt with to achieve good results.

Feedforward Architecture

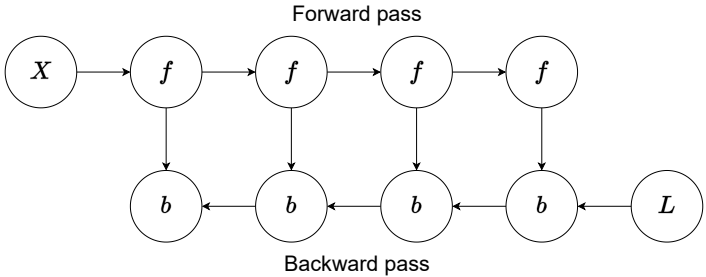


Figure 13: Computation graph for obtaining gradients for a simple feed-forward neural network with n layers. The activations marked with an f. The gradient of the loss with respect to the activations and parameters marked with b.

Feedforward Architecture

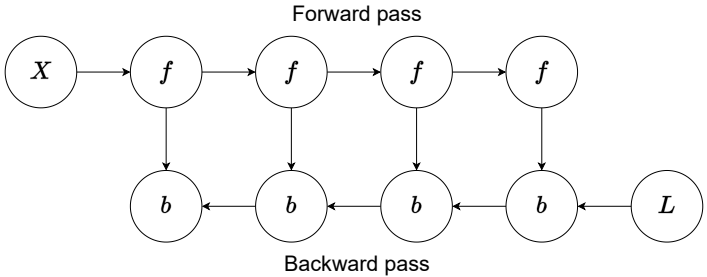


Figure 13: Computation graph for obtaining gradients for a simple feed-forward neural network with n layers. The activations marked with an f. The gradient of the loss with respect to the activations and parameters marked with b.

! Important

The results obtained for the f nodes are needed to compute the b nodes.

Vanilla backpropagation

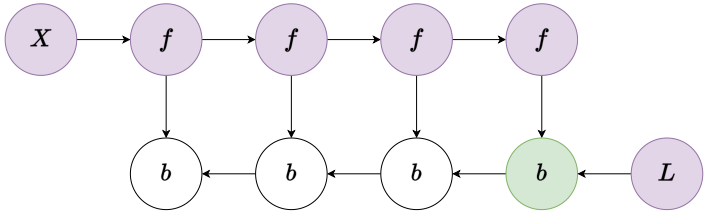


Figure 14: Computation graph for obtaining gradients for a simple feed-forward neural network with n layers. The purple color indicates nodes that are stored in memory.

Vanilla backpropagation

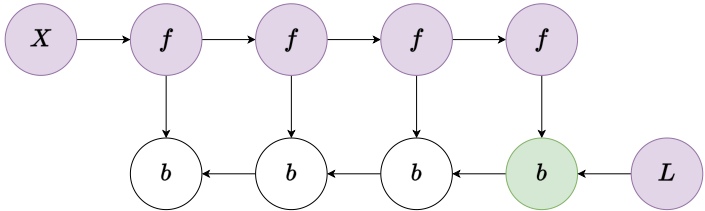


Figure 14: Computation graph for obtaining gradients for a simple feed-forward neural network with n layers. The purple color indicates nodes that are stored in memory.

- All activations f are kept in memory after the forward pass.

Vanilla backpropagation

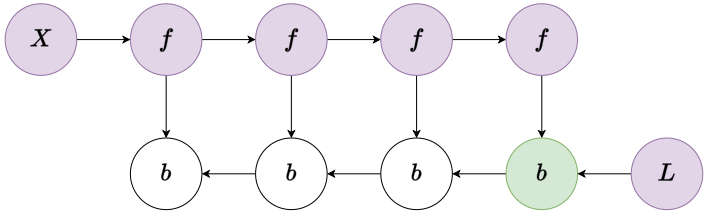


Figure 14: Computation graph for obtaining gradients for a simple feed-forward neural network with n layers. The purple color indicates nodes that are stored in memory.

- All activations f are kept in memory after the forward pass.

Vanilla backpropagation

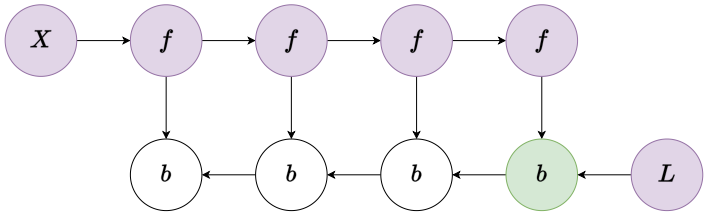


Figure 14: Computation graph for obtaining gradients for a simple feed-forward neural network with n layers. The purple color indicates nodes that are stored in memory.

- All activations f are kept in memory after the forward pass.

- Optimal in terms of computation: it only computes each node once.

Vanilla backpropagation

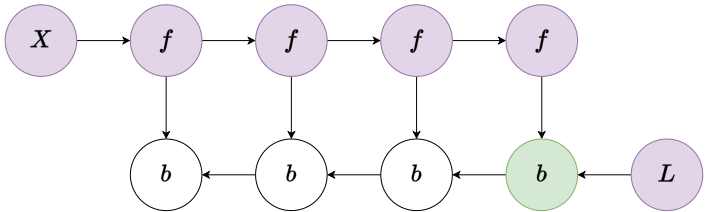


Figure 14: Computation graph for obtaining gradients for a simple feed-forward neural network with n layers. The purple color indicates nodes that are stored in memory.

- All activations f are kept in memory after the forward pass.

- Optimal in terms of computation: it only computes each node once.

Vanilla backpropagation

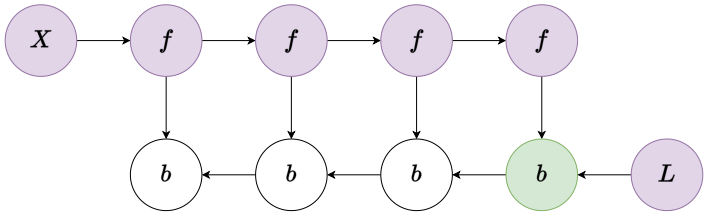


Figure 14: Computation graph for obtaining gradients for a simple feed-forward neural network with n layers. The purple color indicates nodes that are stored in memory.

- All activations f are kept in memory after the forward pass.

- Optimal in terms of computation: it only computes each node once.

- High memory usage. The memory usage grows linearly with the number of layers in the neural network.

Memory poor backpropagation

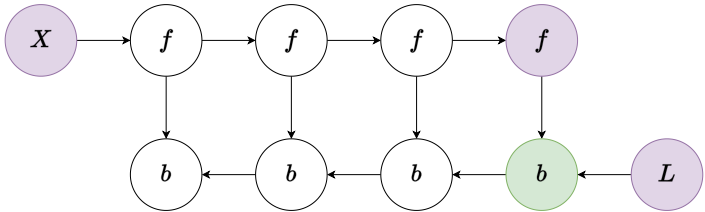


Figure 15: Computation graph for obtaining gradients for a simple feed-forward neural network with n layers. The purple color indicates nodes that are stored in memory.

Memory poor backpropagation

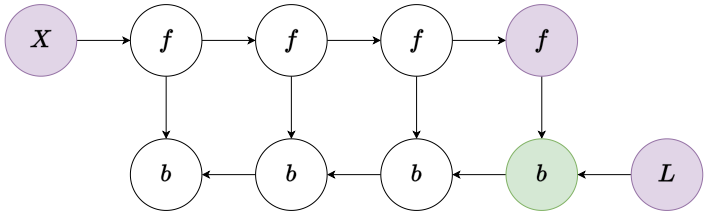


Figure 15: Computation graph for obtaining gradients for a simple feed-forward neural network with n layers. The purple color indicates nodes that are stored in memory.

- Each activation f is recalculated as needed.

Memory poor backpropagation

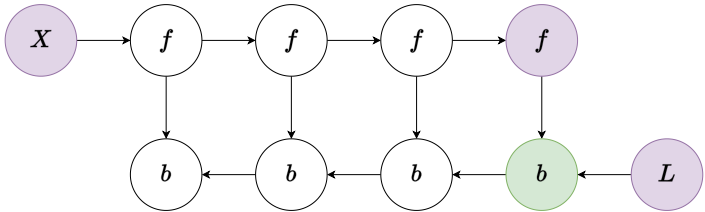


Figure 15: Computation graph for obtaining gradients for a simple feed-forward neural network with n layers. The purple color indicates nodes that are stored in memory.

- Each activation f is recalculated as needed.

Memory poor backpropagation

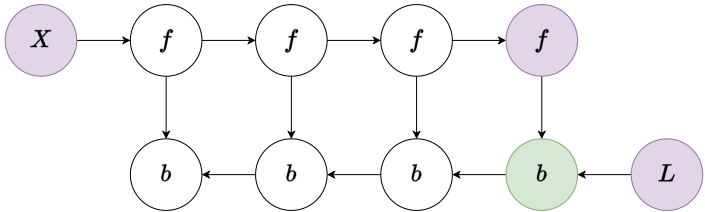


Figure 15: Computation graph for obtaining gradients for a simple feed-forward neural network with n layers. The purple color indicates nodes that are stored in memory.

- Each activation f is recalculated as needed.

- Optimal in terms of memory: there is no need to store all activations in memory.

Memory poor backpropagation

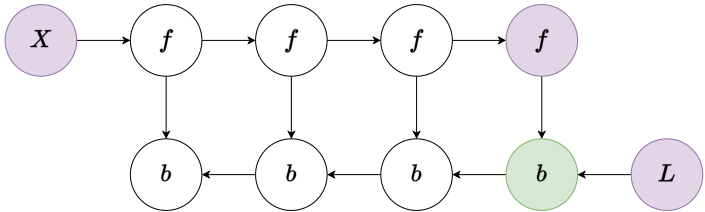


Figure 15: Computation graph for obtaining gradients for a simple feed-forward neural network with n layers. The purple color indicates nodes that are stored in memory.

- Each activation f is recalculated as needed.

- Optimal in terms of memory: there is no need to store all activations in memory.

Memory poor backpropagation

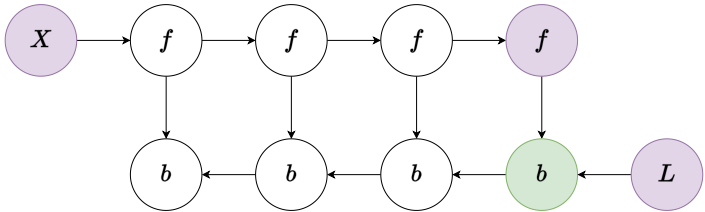


Figure 15: Computation graph for obtaining gradients for a simple feed-forward neural network with n layers. The purple color indicates nodes that are stored in memory.

- Each activation f is recalculated as needed.

- Optimal in terms of memory: there is no need to store all activations in memory.

- Computationally inefficient. The number of node evaluations scales with n^2, whereas it vanilla backprop scaled as n: each of the n nodes is recomputed on the order of n times.

Checkpointed backpropagation

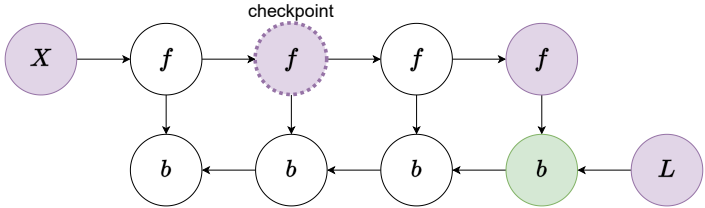


Figure 16: Computation graph for obtaining gradients for a simple feed-forward neural network with n layers. The purple color indicates nodes that are stored in memory.

Checkpointed backpropagation

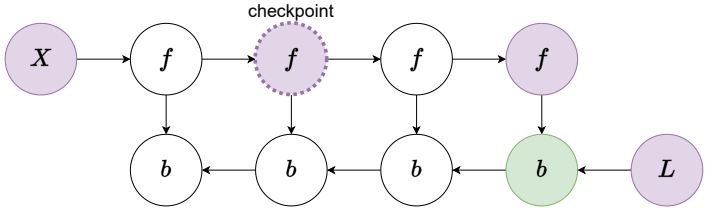


Figure 16: Computation graph for obtaining gradients for a simple feed-forward neural network with n layers. The purple color indicates nodes that are stored in memory.

- Trade-off between the **vanilla** and **memory poor** approaches. The strategy is to mark a subset of the neural net activations as checkpoint nodes, that will be stored in memory.

Checkpointed backpropagation

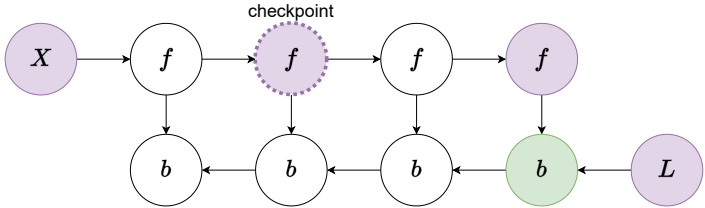


Figure 16: Computation graph for obtaining gradients for a simple feed-forward neural network with n layers. The purple color indicates nodes that are stored in memory.

- Trade-off between the **vanilla** and **memory poor** approaches. The strategy is to mark a subset of the neural net activations as checkpoint nodes, that will be stored in memory.

Checkpointed backpropagation

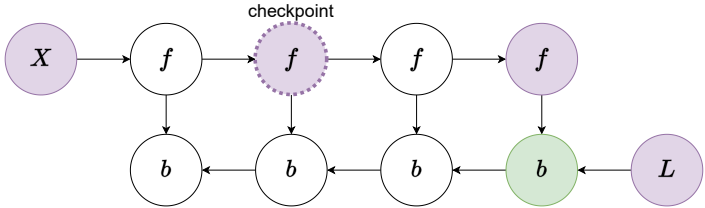


Figure 16: Computation graph for obtaining gradients for a simple feed-forward neural network with n layers. The purple color indicates nodes that are stored in memory.

- Trade-off between the **vanilla** and **memory poor** approaches. The strategy is to mark a subset of the neural net activations as checkpoint nodes, that will be stored in memory.

- Faster recalculation of activations f. We only need to recompute the nodes between a b node and the last checkpoint preceding it when computing that b node during backprop.

Checkpointed backpropagation

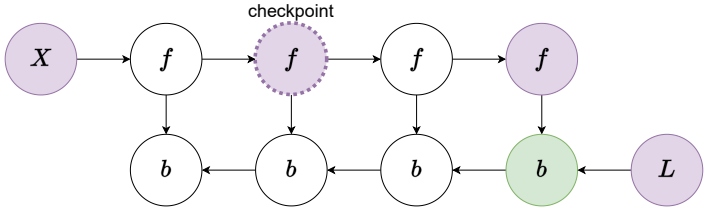


Figure 16: Computation graph for obtaining gradients for a simple feed-forward neural network with n layers. The purple color indicates nodes that are stored in memory.

- Trade-off between the **vanilla** and **memory poor** approaches. The strategy is to mark a subset of the neural net activations as checkpoint nodes, that will be stored in memory.

- Faster recalculation of activations f. We only need to recompute the nodes between a b node and the last checkpoint preceding it when computing that b node during backprop.

Checkpointed backpropagation

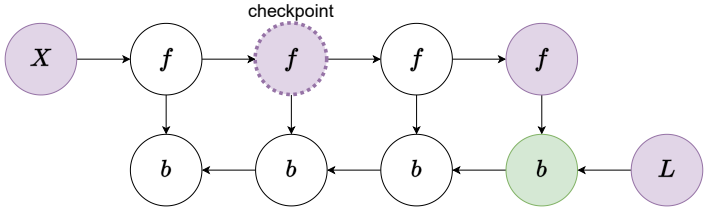


Figure 16: Computation graph for obtaining gradients for a simple feed-forward neural network with n layers. The purple color indicates nodes that are stored in memory.

- Trade-off between the **vanilla** and **memory poor** approaches. The strategy is to mark a subset of the neural net activations as checkpoint nodes, that will be stored in memory.

- Faster recalculation of activations f. We only need to recompute the nodes between a b node and the last checkpoint preceding it when computing that b node during backprop.

- Memory consumption depends on the number of checkpoints. More effective then **vanilla** approach.

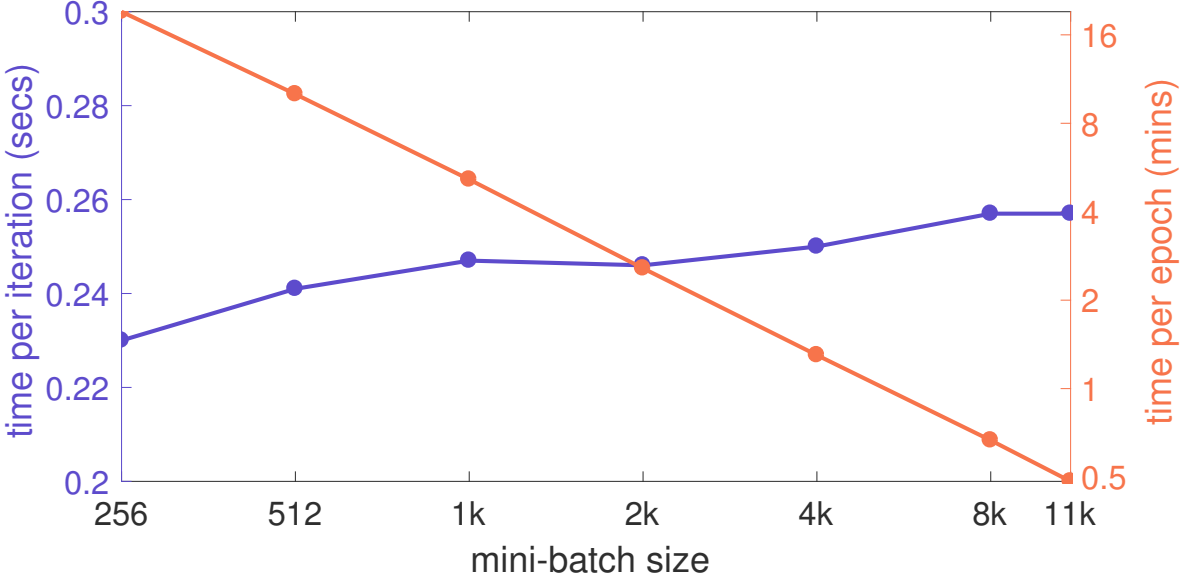
Gradient checkpointing visualization

The animated visualization of the above approaches

An example of using a gradient checkpointing

Large batch training

Large batch training



Large batch training