Optimization in Training Neural Networks

Lecture “Introduction to Control and Machine Learning”

Yongcun Song, Ziqi Wang, and Enrique Zuazua

05.03.2026

Friedrich-Alexander-Universität Erlangen-Nürnberg

Preliminaries

Preliminaries of Optimization

Consider the problem

$$\min_{x \in \mathbb{R}^n} f(x), \tag{UP}$$

where $x \in \mathbb{R}^n$ is a real vector with $n \geq 1$ components and $f : \mathbb{R}^n \to \mathbb{R}^1$ is a smooth function.

- A point x^* is a global minimizer if $f(x^*) \leq f(x)$ for all x;
- A point x^* is a local minimizer if there is a neighborhood $\mathcal{N}$ of x^* such that $f(x^*) \leq f(x)$ for all $x \in \mathcal{N}$;
- A point x^* is a strict local minimizer (also called a strong local minimizer) if there is a neighborhood $\mathcal{N}$ of x^* such that $f(x^*) < f(x)$ for all $x \in \mathcal{N}$ with $x \neq x^*$;
- A point x^* is an isolated local minimizer if there is a neighborhood $\mathcal{N}$ of x^* such that x^* is the only local minimizer in $\mathcal{N}$.

Convexity

Convex function

Let $f : \mathbb{R}^n \to \mathbb{R}$ be a twice continuously differentiable function. We say that f is convex if

$$f(tx + (1-t)y) \leq tf(x) + (1-t)f(y), \forall x, y \in \mathbb{R}^n, t \in [0,1]. \quad (1)$$

Lemma 1.1

Let $f : \mathbb{R}^n \to \mathbb{R}$ be twice continuously differentiable. Then f is convex if and only if any of the following equivalent conditions hold:

$$f(y) \geq f(x) + \nabla f(x)^T (y - x), \forall x, y \in \mathbb{R}^n \quad (2)$$

or

$$v^T \nabla^2 f(x) v \geq 0, \forall x, v \in \mathbb{R}^n \quad (3)$$

Convexity

Proof: We will prove the result by showing that (1)⇔(2)⇔(3).

(1)⇒ (2): We can deduce (2) from (1) by dividing t and rearranging

$$\frac{f(y + t(x - y)) - f(y)}{t} \leq f(x) - f(y)$$

Taking the limit $t \to 0$ gives

$$\nabla f(y)^T (x - y) \leq f(x) - f(y).$$

Convexity

(2)$\Rightarrow$ (1): Let $x_t = tx + (1-t)y$. It follows from (2) that

$$f(x) \geq f(x_t) + \nabla f(x_t)^T (x - x_t) = f(x_t) - (1-t)\nabla f(x_t)^T (y - x)$$
$$f(y) \geq f(x_t) + \nabla f(x_t)^T (y - x_t) = f(x_t) + t\nabla f(x_t)^T (y - x)$$

Multiplying the first inequality by t and the second inequality by $(1-t)$ and adding the results together gives

$$tf(x) + (1-t)f(y) \geq f(x_t)$$

which is equivalent to (1).

Convexity

(2)⇒(3) First, for any $t \in \mathbb{R}$ and $v \in \mathbb{R}^n$, using Taylor expansion we have

$$f(x+tv) = f(x) + t\nabla f(x)^T v + \frac{1}{2}t^2 v^T \nabla^2 f(x) v + o(t^2)o(v^2).$$

Moreover, it follows from (2), which implies
$f(x+tv) \geq f(x) + t\nabla f(x)^T v$, that

$$\frac{1}{2}t^2 v^T \nabla^2 f(x) v + o(t^2)o(v^2) \geq 0.$$

Cancelling the t^2 term and then taking the limit as $t \to 0$ shows

$$v^T \nabla^2 f(x) v \geq 0$$

Convexity

(3)⇒ (2): Using Taylor expansion we have

$$f(x) = f(y) + \nabla f(y)^T (x - y) + \frac{1}{2}(x - y)^T \nabla^2 f(x + \tau(x - y))(x - y),$$

for some $\tau \in [0, 1]$.

Then the desired result follows from (3) directly.

Strongly convex function

We say that f is μ-strongly convex if

$$f(y) \geq f(x) + \nabla f(x)^T (y - x) + \frac{\mu}{2}\|x - y\|^2, \forall\, x, y \in \mathbb{R}^n, \mu > 0. \quad (4)$$

Lemma 1.2

Let $f : \mathbb{R}^n \to \mathbb{R}$ be twice continuously differentiable. Then f is μ-strongly convex if and only if

$$v^T \nabla^2 f(x) v \geq \mu \|v\|^2, \forall\, x, v \in \mathbb{R}^n \tag{5}$$

Proof: The proof is left for exercise.

Smoothness

L-smooth function

A differential function f is said to be L-smooth if its gradients are Lipschitz continuous, that is

$$\|\nabla f(x) - \nabla f(y)\| \leq L\|x - y\|. \tag{6}$$

Necessary and sufficient conditions

Theorem 1.1 (First-order optimality conditions)

If x^* is a local minimizer and f is continuously differentiable in an open neighborhood of x^*, then $\nabla f(x^*) = 0$.

Necessary and sufficient conditions

- We call x^* a stationary point if $\nabla f(x^*) = 0$. According to the above theorem, any local minimizer must be a stationary point. Note that a stationary point may represent different situations.
- Consider

$$f(x) = \sin x, x = k\pi,$$

$$f(x) = x^3, x = 0.$$

Necessary and sufficient conditions

Theorem 1.2 (Second-order optimality conditions)

If x^* is a local minimizer and $\nabla^2 f$ exists and is continuous in an open neighborhood of x^*, then $\nabla f(x^*) = 0$ and $\nabla^2 f(x^*)$ is positive semidefinite.

Necessary and sufficient conditions

Theorem 1.3 (Second-order sufficient conditions)

Suppose that $\nabla^2 f$ is continuous in an open neighborhood of x^* and that $\nabla f(x^*) = 0$ and $\nabla^2 f(x^*)$ is positive definite. Then x^* is a strict local minimizer of f.

Necessary and sufficient conditions

- The second-order sufficient conditions are not necessary: A point x^* may be a strict local minimizer, and yet may fail to satisfy the sufficient conditions.
- A simple example is given by the function $f(x) = x^4$, for which the point $x^* = 0$ is a strict local minimizer at which the Hessian matrix vanishes.

Necessary and sufficient conditions

Theorem 1.4

When f is convex, any local minimizer x^* is a global minimizer of f. If in addition f is differentiable, then any stationary point x^* is a global minimizer of f.

An overview of optimization algorithms

Motivations: generate a sequence of points $\{x_k\}$ (x_0 given) such that

- $f(x_k) > f(x_{k+1})$;
- $\{x_k\}$ is convergent
- the limit point is a stationary point.

Steps: **(a)** find a descent direction, i.e. find a $d_k \in \mathbb{R}^n$ such that $d_k^T \nabla f(x_k) < 0$

(b) find a proper step along the direction d_k

(c) stop the process when x_k is close enough to the local minimizer.

Classification The choice of various descent directions determines various methods. **(b)** and **(c)** above are always required in all methods.

Note: There are some methods which do not require $f(x_k) > f(x_{k+1})$.

Desecent direction

A descent direction d for f at $x \iff f(x+\rho d) < f(x), \rho > 0$ (small enough)
f is differentiable:

$$\nabla f(x)^T d < 0 \Rightarrow d \text{ is a descent direction for } f \text{ at } x$$

f is convex and differentiable:

$$\nabla f(x)^T d < 0 \iff d \text{ is a descent direction for } f \text{ at } x$$

Descent direction

Using a first-order expansion,

$$f(x+\rho d) = f(x) + \rho \nabla f(x)^T d + \rho \|d\| \varepsilon(\rho d) \tag{7}$$

$$\text{where} \quad \varepsilon(\rho d) \to 0 \text{ as } \rho \to 0 \tag{8}$$

Then (7) can be rewritten as

$$f(x) - f(x+\rho d) = \rho \left(-\nabla f(x)^T d - \|d\| \varepsilon(\rho d) \right).$$

Consider that $\nabla f(x)^T d < 0$. By (8), there is $\bar{\rho} > 0$ such that $-\nabla f(x)^T d > \|d\| \varepsilon(\rho d), 0 < \rho < \bar{\rho}$. Then d is a descent direction since $f(x) - f(x + \rho \mathrm{d}) > 0, 0 < \rho < \bar{\rho}$.

Gradient Descent Method

Gradient descent method

- Choose initial point $x_0 \in \mathbb{R}^n$
- Repeat
- Choose descent direction $-\nabla f(x_k)$ and step size $t > 0$
- Update

$$x_{k+1} = x_k + t\nabla f(x_k) \tag{9}$$

- Until stopping criterion is satisfied

Convergence analysis

Lemma 1.5

For L-smooth f, the sequence$\{x_k\}$ produced by gradient descent satisfies

$$f(x_{k+1}) \leq f(x_k) - t\left(1 - \frac{Lt}{2}\right)\|\nabla f(x_k)\|^2$$

Note:

- if $\nabla f(x_k) \neq 0$ and $0 < t < \frac{2}{L}$, then $f(x_{k+1}) < f(x_k)$, so gradient descent with step size $t \in (0, 2/L)$ is indeed a descent method.
- We can lower bound the decrease in function value in each step. In particular, for $0 < t < \frac{1}{L}$,

$$f(x_k) - f(x_{k+1}) \geq \frac{t}{2}\|\nabla f(x_k)\|^2$$

Convergence analysis

Theorem 1.5

If f is convex and L-smooth, and x^* is a minimum of f, then for step size $t \in (0, \frac{1}{L}]$, the sequence$\{x_k\}$ produced by the gradient descent algorithm satisfies

$$f(x_k) - f(x^*) \leq \frac{\|x_0 - x^*\|^2}{2tk}$$

Convergence analysis

Note:

- $f(x_k) \downarrow f^*$ as $k \to \infty$.
- Any limiting point of x_k is an optimal solution.
- The rate of convergence is $O(1/k)$, i.e. the number of iterations to guarantee $f(x_k) - f(x^*) \leq \epsilon$ is $O(1/\epsilon)$. For $\epsilon = 10^{-p}, k = O(10^p)$, exponential in the number of significant digits!
- Faster convergence with larger t; best $t = \frac{1}{L}$, but L is unknown.
- Good initial guess helps.

Convergence analysis

Theorem 1.6

Let f be L-smooth and μ-strongly convex. From a given $x_0 \in \mathbb{R}^d$ and $\frac{1}{L} \geq t > 0$, the iterates (9) converge according to

$$\|x_{k+1} - x^*\|_2^2 \leq (1 - t\mu)^{k+1} \|x_0 - x^*\|_2^2. \tag{10}$$

In particular, or $t = \frac{1}{L}$ the gradient descent iterates enjoy a linear convergence with a rate of μ/L.

Line search

- Recall that the update rule of Gradient Descent requires a step size t_k controlling the amount of gradient updated to the current point at each iteration.
- A naive strategy is to set a constant $t_k = t$ for all iterations. (see (9)).
- This strategy poses two problems.
  - A too large step size t can lead to divergence, meaning the learning function oscillates away from the optimal point.
  - A too small step size t takes longer time for the function to converge.
- Hence, a good selection of step size t is necessary to make the algorithm converge faster.
- Two examples of such approaches are exact line search and backtracking line search.

Line search

Exact line search:

For any descent direction p_k at x_k, compute the step size t_k by solving

$$t_k = \arg\min_{s>0} f(x_k + sp_k)$$

- In general, the above one-dimensional optimization problem for computing t_k cannot be solved exactly, and only an approximate solution can be pursued.
- On the other hand, one my consider starting with the full step, i.e. $t_k = 1$, if this fails to satisfy the criterion in use, to backtrack in a systematic way along the same direction. It seems fine as long as to accept the t_k,

$$f(x_{k+1}) = f(x_k + t_k p_k) < f(x_k).$$

- Unfortunately this simple condition does not guarantee that $\{x_k\}$ will converge to a minimizer of f.

Line search

Now we look at two examples:

Example 1.1

$f(x) = x^2$, $x_0 = 2$. If $\{p_k\} = \{(-1)^{k+1}\}$, $\{t_k\} = \{2 + 3 \cdot 2^{-(k+1)}\}$, then $\{x_k\} = \{(-1)^k(1 + 2^{-k})\}$. Each p_k is a descent direction from x_k, and $f(x_k)$ is monotonically decreasing with $f(x_k) \to 1$ as $k \to \infty$. $\{x_k\}$ has limit points ± 1, so it does not converge.

Example 1.2

$f(x) = x^2$, $x_0 = 2$. If $\{p_k\} = \{-1\}$, $\{t_k\} = \{2^{-k-1}\}$, then $\{x_k\} = \{1 + 2^{-k}\}$. Each p_k is a descent direction from x_k, and $f(x_k)$ decreases monotonically, and $x \to 1$ as $k \to \infty$. But 1 is not a minimizer of f.

Line search

What went wrong in both examples? In Example 1.1, we achieved very small decreases in f values relative to the lengths of the steps. This can be fixed as follows: pick an $\alpha \in (0, 1)$ and choose t_k from among those $t > 0$ that satisfy

$$f(x_k + t_k p_k) \leq f(x_k) + \alpha t_k \nabla f(x_k)^T p_k. \tag{11}$$

Equivalently, t_k must be chosen so that

$$f(x_{k+1}) \leq f(x_k) + \alpha \nabla f(x_k)^T (x_{k+1} - x_k). \tag{12}$$

Line search

It can be verified that this precludes the $\{x_k\}$ in Example 1.1 but not the $\{x_k\}$ in Example 1.2. In Example 1.2, the situation is the opposite, the steps are too small relative to the rate of decrease of f. This can be fixed by requiring

$$\nabla f(x_{k+1})^T p_k := \nabla f(x_k + t_k p_k)^T p_k \geq \beta_k \nabla f(x_k)^T p_k \tag{13}$$

or equivalently

$$\nabla f(x_{k+1})^T (x_{k+1} - x_k) \geq \beta_k \nabla f(x_k)^T (x_{k+1} - x_k) \tag{14}$$

where $\beta \in (\alpha, 1)$. The condition $\beta > \alpha$ guarantees that (11) and (13) can be satisfied simultaneously.

Line search

Conditions (11) and (13) are based on work of Armijo (1966) and Goldstein (1967). (11) is sometimes called Armijo's rule, and (13) is sometimes called Goldstein's rule.

Line search

Backtracking: The modern strategy is to start with $t_k = 1$, and then, if $x_k + p_k$ is not acceptable, "backtrack" (reduce t_k) until an acceptable $x_k + t_k p_k$ is found.

Backtracking line search framework:

Given $\alpha \in (0, \frac{1}{2}), \quad 0 < l < u < 1$

$t_k = 1$;

while $f(x_k + t_k p_k) > f(x_k) + \alpha t_k \nabla f(x_k)^T p_k$, do

    $t_k := \rho t_k$ for some $\rho \in [l, u]$;

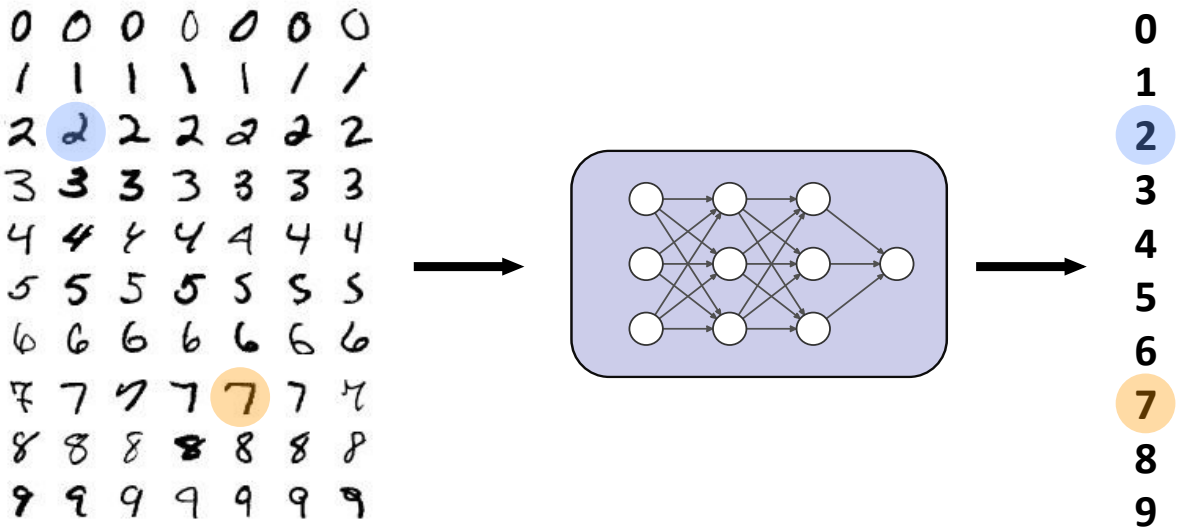
    ($^*\rho$ is chosen anew each time by the line search*)

$x_{k+1} := x_k + t_k p_k$;

Training of machine learning models

Background: optimization in machine learning

- Machine learning aims to model observed data to make predictions on unobserved data by:
  (1) Parametrizing a set of model parameters $w \in \mathbb{R}^d$;
  (2) Defining a loss function $F(w)$ to measure the total discrepancy between the model prediction and the real data;
  (3) Finding the optimal parameters w^* which minimize the loss $F(w)$.



Loss function

- Consider minimizing the sum of functions:

$$\min_{w \in \mathbb{R}^d} F(w) := \frac{1}{n} \sum_{i=1}^{n} f_i(w).$$

- In the context of supervised learning:
  - $f_i : \mathbb{R}^d \to \mathbb{R}$ is the loss of the i-th training data (x_i, y_i):

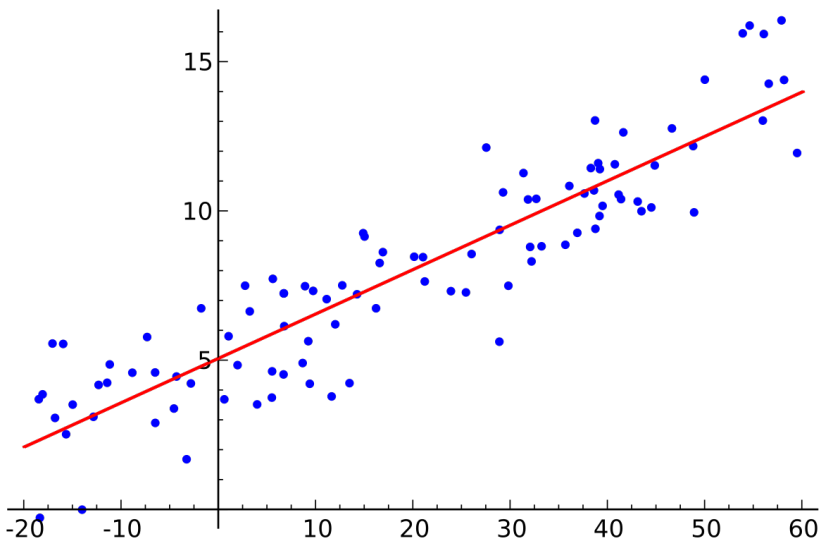
$$f_i(w) := \text{dist}\left(\text{NN}(x_i), y_i\right);$$

  - $F(w)$ is the empirical loss of the whole training dataset.

Example: linear regression

Given input variables $x_i \in \mathbb{R}^d$ and the corresponding labels y_i, $i = 1, \cdots, n$, a linear regression model reads as

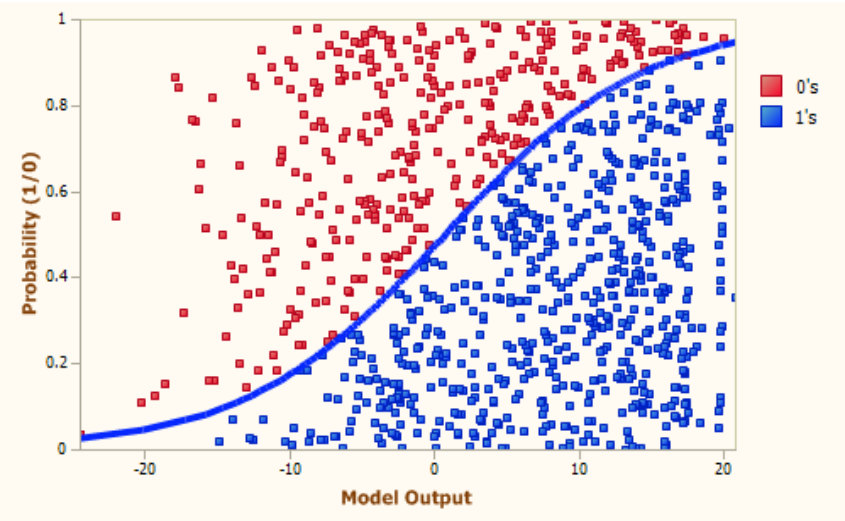
$$\min_w \frac{1}{n} \sum_{i=1}^{n} f_i(w), \quad \text{with} \quad f_i(w) = (w^T(x_i; 1) - y_i)^2.$$



Example: logistic regression

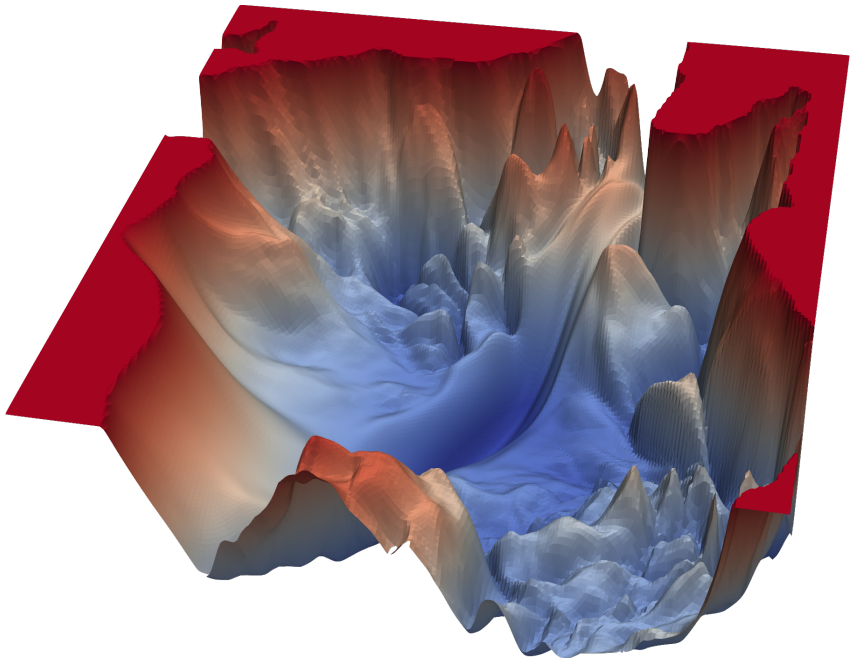
- Given features $x_i \in \mathbb{R}^d$ and the corresponding labels $y_i = \{0, 1\}$, $i = 1, \cdots, n$, a logistic regression model reads as

$$\min_{w \in \mathbb{R}^d} \frac{1}{n} \sum_{i=1}^{n} f_i(w), \quad \text{with} \quad f_i(w) = -y_i x_i^T w + \log(1 + e^{x_i^T w}).$$



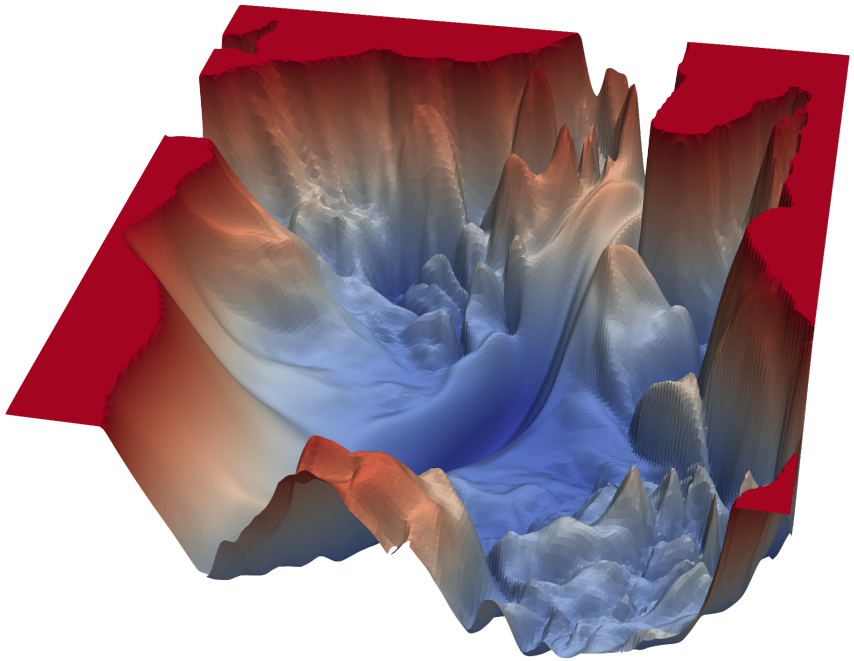
Loss function for neural networks

- Loss function for neural networks is in general high-dimensional and non-convex with many local minima.



Loss function for neural networks

- Loss function for neural networks is in general high-dimensional and non-convex with many local minima.

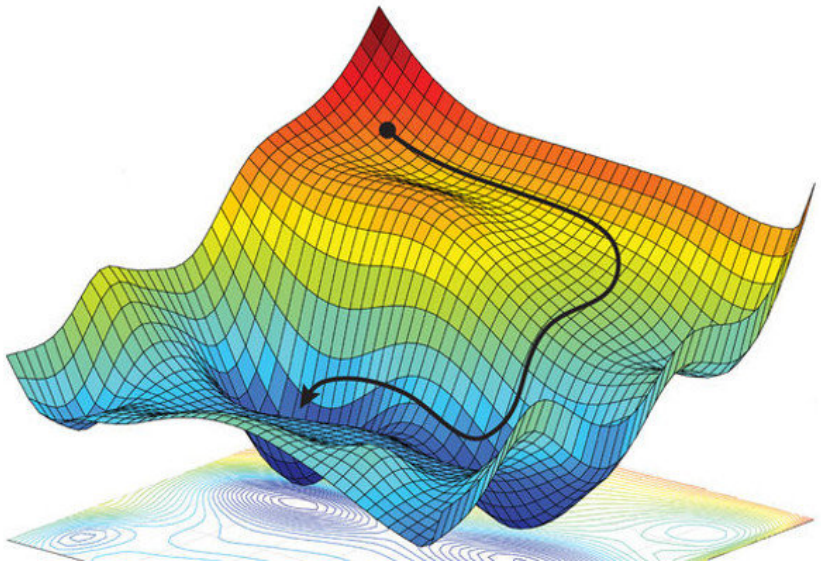


- **Reason for non-convexity:**
  If w^* is a global minimizer of the loss function $F(w)$, then any permutation of neurons and its connected parameters in the hidden layer leads to the same global loss and thus must also be a global minimizer.

Minimization of loss function

(Ambitiuous) Goal:

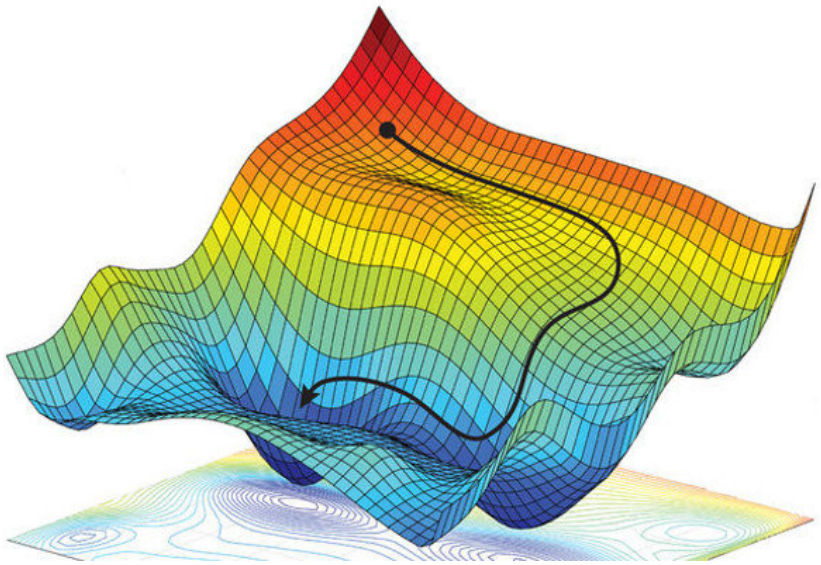
Compute the global minimum of the loss function.



Minimization of loss function

(Ambitiuous) Goal:

Compute the global minimum of the loss function.

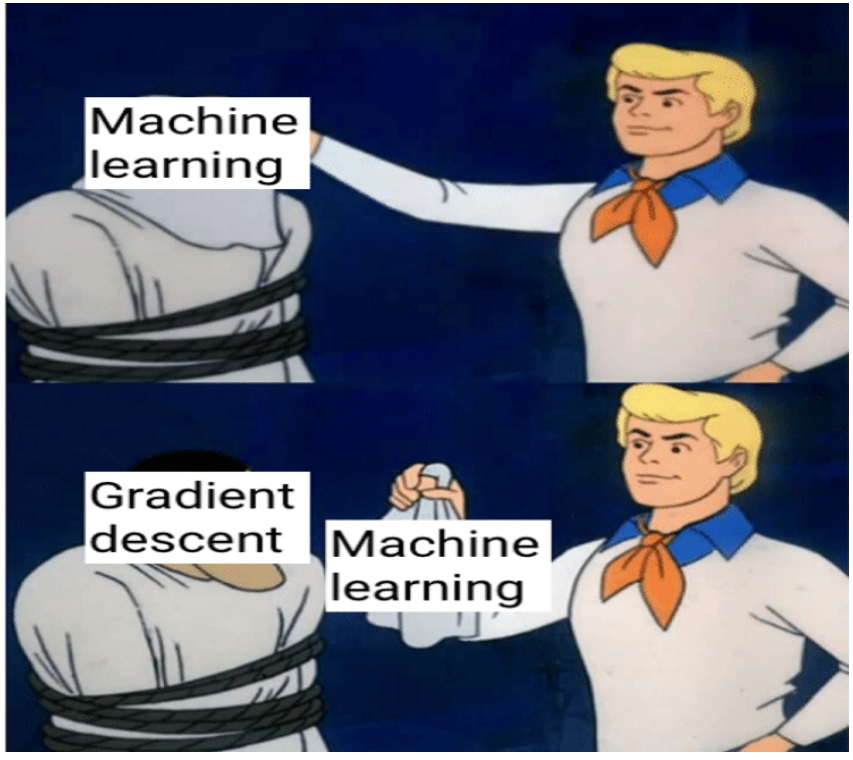


(More realistic) Goal:

Decrease the loss function value iteratively.

Minimization of loss function

- How to iteratively minimize the high-dimensional loss function $F(w)$?
- Essentially all ML models are trained using gradient-type methods.



Gradient descent

- **Gradient descent (GD)** (a.k.a batch gradient descent or full batch gradient descent):

$$w^{k+1} = w^k - \alpha_k(\frac{1}{n}\sum_{i=1}^{n}\nabla f_i(w^k)),$$

where α_k is a positive learning rate.

Challenges with GD

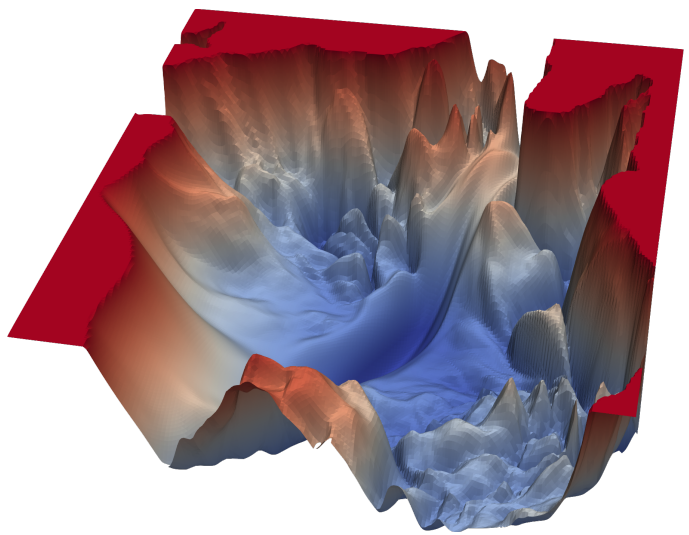
- **Computational cost**

  When n is large and no simple formulas exist, evaluating the sums of gradients becomes prohibitively expensive.

- **Redundant information**

  Each iteration goes over all samples where some samples may be redundant and do not contribute much to the update.

- **Bad local minima**



Stochastic gradient descent

- **Stochastic gradient descent (SGD):**
  - Choose an integer i uniformly at random from $\{1, 2, \cdots, n\}$.
  - Update
$$w^{k+1} = w^k - \alpha_k \nabla f_i(w^k),$$
where α_k is a positive learning rate.

- **Notes on SGD:**
  - Above, the index i is chosen by sampling with replacement.
  - An alternative is to sample without replacement. Performing n steps in this manner is referred to as completing an epoch.
  - The full gradient is approximated by an unbiased estimate, i.e.,
$$\mathbb{E}[\nabla f_i(w)] = \nabla F(w).$$

Convergence of SGD

Assumptions

1: The loss is bounded from below, i.e., $F(w) \geq F^*$ for all $w \in \mathbb{R}^d$.

2: The loss is L-smooth, i.e., $\|\nabla F(w) - \nabla F(\tilde{w})\| \leq L\|w - \tilde{w}\|$ for all $w, \tilde{w} \in \mathbb{R}^d$.

3: $\mathbb{E}[\|\nabla f_i(w^k) - \nabla F(w^k)\|^2] \leq \sigma^2$.

4: The learning rate satisfies $\alpha_k < \frac{2}{L}$.

Convergence of SGD

Suppose that Assumptions 1-4 hold, we have:

- The expected loss decays up to the noise level:

$$F(w^{k+1}) \leq F(w^k) + \alpha_k \sigma^2.$$

- If $\sum_k \alpha_k = \infty$ and $\sum_k \alpha_k^2 < \infty$, we have that

$$\min_{k=1,\ldots,K} \|\nabla F(w^k)\|^2 \to 0 \text{ as } K \to \infty.$$

Proof of convergence

For the first part, using that F is L-smooth

$$F(w^{k+1}) \leq F(w^k) + \langle \nabla F(w^k), w^{k+1} - w^k \rangle + \frac{L}{2}\|w^{k+1} - w^k\|^2$$
$$= F(w^k) - \alpha_k \langle \nabla F(w^k), g_k \rangle + \alpha_k^2 \frac{L}{2}\|g_k\|^2, \text{ with } g_k = \nabla f_i(w^k).$$

Moreover, $\mathbb{E}[\|g_k - \nabla F(w^k)\|^2] \leq \sigma^2$ implies that $\mathbb{E}[\|g_k\|^2] \leq \sigma^2 + \|\nabla F(w^k)\|^2$. Then taking expectations yields

$$F(w^{k+1}) \leq F(w^k) - \alpha_k \|\nabla F(w^k)\|^2 + \alpha_k^2 \frac{L}{2}\left(\sigma^2 + \|\nabla F(w^k)\|^2\right)$$

If $\alpha_k < \frac{2}{L}$, we have that

$$F(w^{k+1}) \leq F(w^k) + \alpha_k \sigma^2$$

Proof of convergence-Cont'd

Notice that we have

$$\alpha_k \left(1 - \alpha_k \tfrac{L}{2}\right) \|\nabla F(w^k)\|^2 \leq F(w^k) - F(w^{k+1}) + \alpha_k^2 \frac{L}{2}\sigma^2$$

Summing up, using $F \geq F^*$, one can show that

$$\sum_{k=1}^{K} \alpha_k \left(1 - \alpha_k \tfrac{L}{2}\right) \|\nabla F(w^k)\|^2 \leq F(w^1) - F^* + \frac{L}{2}\sigma^2 \sum_{k=1}^{K} \alpha_k^2.$$

Hence, if $\alpha_k < \frac{2}{L}$, $\alpha_k \left(1 - \alpha_k \frac{L}{2}\right) \geq c\alpha_k$ for some $c > 0$, we then have

$$\min_{k=1,\ldots,K} \|\nabla F(w^k)\|^2 c \sum_{k=1}^{K} \alpha_k \leq F(w^1) - F^* + \frac{L}{2}\sigma^2 \sum_{k=1}^{K} \alpha_k^2$$

Proof of convergence-Cont'd

The above estimate implies that

$$\min_{k=1,\ldots,K} \|\nabla F(w^k)\|^2 \leq \frac{F(\theta_1) - F^*}{c\sum_{k=1}^{K} \alpha_k} + \frac{L}{2}\sigma^2 \frac{\sum_{k=1}^{K} \alpha_k^2}{c\sum_{k=1}^{K} \alpha_k}$$

For convergence as $K \to \infty$, it is required that

$$\sum_{k=1}^{\infty} \alpha_k = \infty \qquad \text{and} \qquad \sum_{k=1}^{\infty} \alpha_k^2 < \infty.$$

SGD vs. GD

- **Intuitively:** SGD employs information more efficiently than GD;

SGD vs. GD

- **Intuitively:** SGD employs information more efficiently than GD;

- **Theoretically:**
  - **GD:** $F(w^k) - F^* \leq O(\rho^k)$ with $\rho \in (0, 1)$
    The total work required to obtain an ε-optimal solution by GD is proportional to $n \log(1/\varepsilon)$;
  - **SGD:** $\mathbb{E}[F(w^k) - F^*] = O(1/k)$,
    The total work required to obtain an ε-optimal solution by SGD is proportional to $1/\varepsilon$;

SGD vs. GD

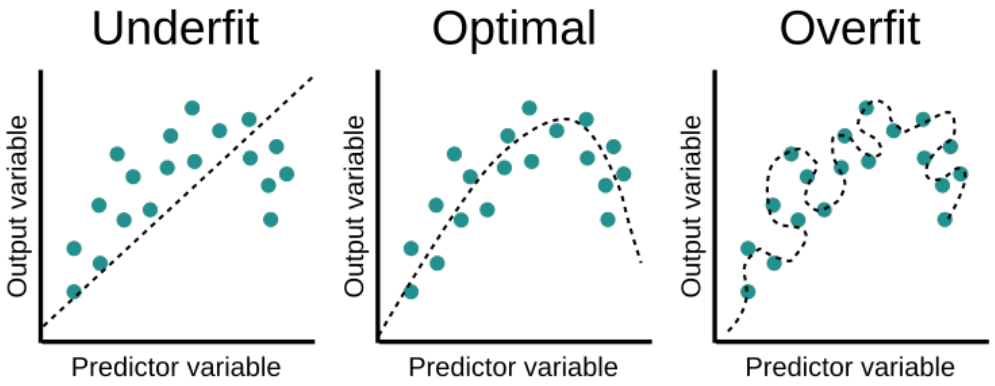
- **Intuitively:** SGD employs information more efficiently than GD;

- **Theoretically:**
  - **GD:** $F(w^k) - F^* \leq O(\rho^k)$ with $\rho \in (0, 1)$
    The total work required to obtain an ε-optimal solution by GD is proportional to $n \log(1/\varepsilon)$;
  - **SGD:** $\mathbb{E}[F(w^k) - F^*] = O(1/k)$,
    The total work required to obtain an ε-optimal solution by SGD is proportional to $1/\varepsilon$;

- Key: For SGD, neither the per-iteration cost nor the convergence rate depends on the dataset size n. Therefore, SGD is preferred when one moves to the big data regime where n is large.

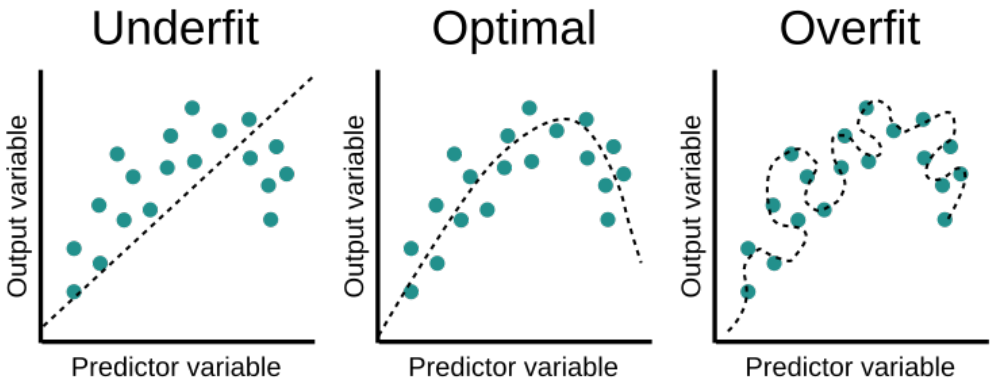
SGD vs. GD

- SGD is less prone to overfitting because the data are chosen randomly.

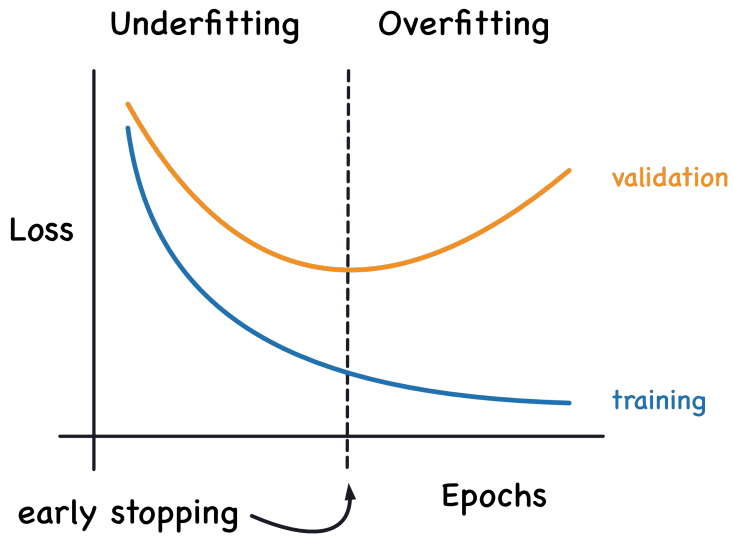


SGD vs. GD

- SGD is less prone to overfitting because the data are chosen randomly.

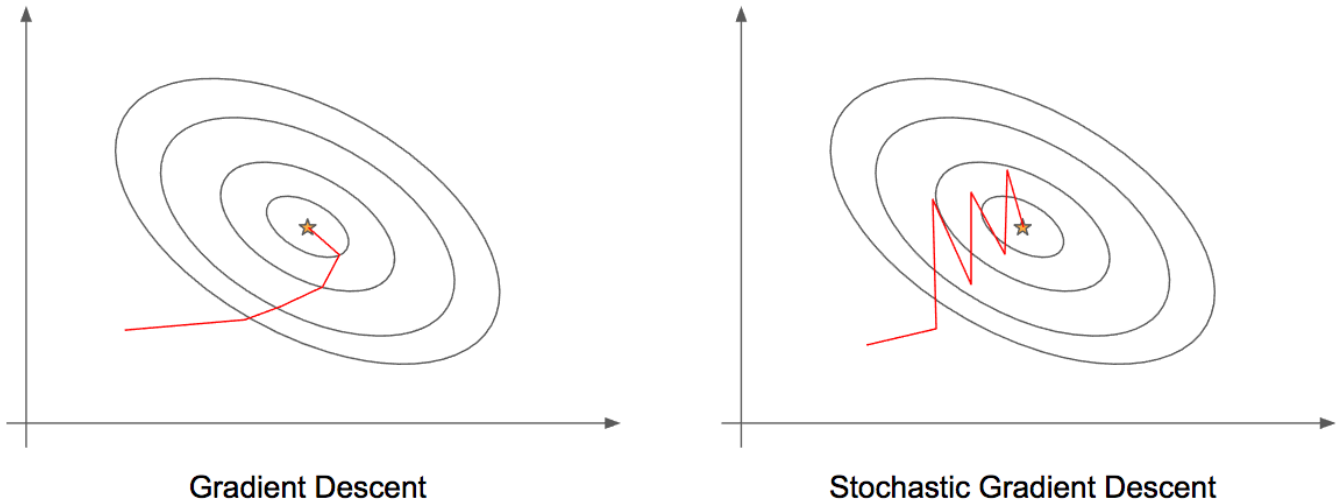


- Use a validation set to indicate early stopping. "Early stopping is a beautiful free lunch." – Geoffrey Hinton at NIPS 2015.



SGD vs. GD

- The computation of SGD is faster as we take one data point at a time, but SGD takes many more iterations to converge.



- SGD is more sensitive to outliers.
  That means, while randomly choosing data points, if we encounter an outlier, then it will take some more extra updates to get back on track of convergence.

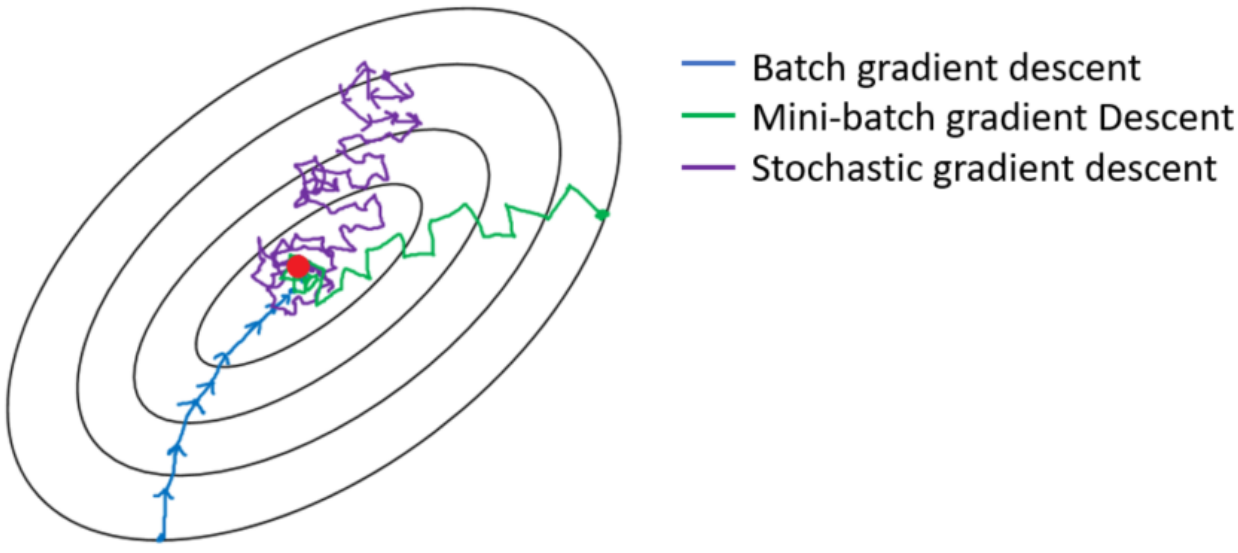
Mini-batch gradient descent

- **Mini-batch gradient descent (mini-batch GD)**
  - Choose a random subset $I_k \subset \{1, 2, \cdots, n\}$ of size $b \ll n$;
  - Update

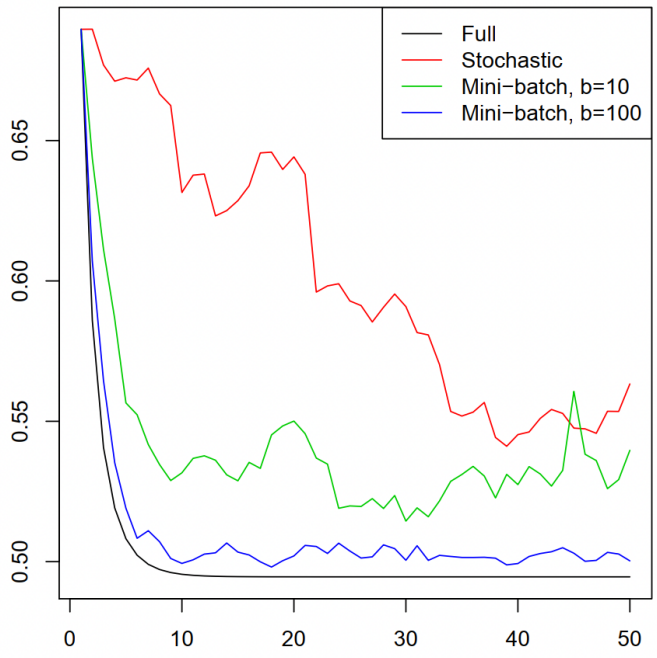
$$w^{k+1} = w^k - \alpha_k \left(\frac{1}{b} \sum_{i \in I_k} \nabla f_i(w^k)\right).$$

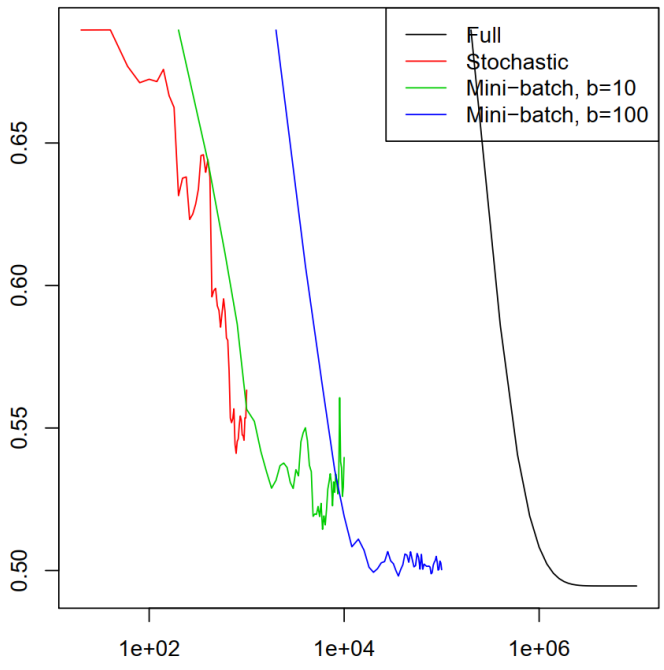
- **Notes on mini-batch GD:**
  - Reduce the variance of the updates, more stable convergence;
  - Make use of highly optimized matrix optimizations inside deep learning libraries that make computing very efficient.
  - Common mini-batch sizes range between 50 and 256, but can vary for different applications.

Comparison: GD, SGD, and mini-batch GD



Comparison: GD, SGD, and minibatch SGD - Cont'd





Numerical

(a) Loss value vs. Iterations (b) Loss value vs. Flops

comparison for solving a regularized logistic regression

Improved stochastic gradient methods

- Variance reduction of the gradient
  - SVRG (Stochastic variance reduced gradient)
  - SAG (Stochastic average gradient)
  - SAGA
- Adaptive learning rate
  - AdaGrad (Aadptive Gradient algorithm)
  - RMSprop
  - Adam
  - Adadelta
  - AdaMax
- Acceleration
  - Momentum
  - Nesterov accelerated gradient (NAG)

Visualization

https://imgur.com/a/Hqolp

- The animation shows the behavior of algorithms at a saddle point;
- SGD, Momentum, and NAG find it difficult to break symmetry;
- Adagrad, RMSprop, and Adadelta quickly find the negative slope.

Recap: optimization in machine learning

- Machine learning aims to model observed data to make predictions on unobserved data by:
  1. Parametrizing model parameters $w \in \mathbb{R}^d$;
  2. Defining a loss function $F(w)$ to measure the total discrepancy;
  3. Finding the optimal parameters w^* which minimize the loss $F(w)$.
- What we have learned about neural networks:
  - Loss function is in general high-dimensional and non-convex;
  - The parameters are optimized using the SGD;
  - Does SGD converge?
  - Can SGD be improved?

SGD with momentum

- **Observations about SGD:**
  - SGD takes a lot of time to navigate regions having a gentle slope;
  - This is because the gradient in these regions is very small.

SGD with momentum

- **Observations about SGD:**
  - SGD takes a lot of time to navigate regions having a gentle slope;
  - This is because the gradient in these regions is very small.
- **SGD with momentum:**

$$\begin{cases} v_{k+1} = \alpha_k \nabla f_i(w_k) + \beta v_k, \\ w_{k+1} = w_k - v_{k+1}; \end{cases}$$

SGD with momentum

- **Observations about SGD:**
  - SGD takes a lot of time to navigate regions having a gentle slope;
  - This is because the gradient in these regions is very small.
- **SGD with momentum:**

$$\begin{cases} v_{k+1} = \alpha_k \nabla f_i(w_k) + \beta v_k, \\ w_{k+1} = w_k - v_{k+1}; \end{cases}$$

- Or equivalently:

$$w_{k+1} = w_k - \alpha_k \nabla f_i(w_k) + \beta(w_k - w_{k-1});$$

SGD with momentum

- **Observations about SGD:**
  - SGD takes a lot of time to navigate regions having a gentle slope;
  - This is because the gradient in these regions is very small.
- **SGD with momentum:**

$$\begin{cases} v_{k+1} = \alpha_k \nabla f_i(w_k) + \beta v_k, \\ w_{k+1} = w_k - v_{k+1}; \end{cases}$$

- Or equivalently:

$$w_{k+1} = w_k - \alpha_k \nabla f_i(w_k) + \beta(w_k - w_{k-1});$$

- The term $\beta(w_k - w_{k-1})$ is referred to as the momentum term.
- When $\beta = 0$, it reduces to SGD;
- Here, β is a damping parameter slightly less than one, such as 0.9.

Why momentum really works

(a) SGD without momentum

(b) SGD with momentum

- Gradient descent is a man walking down a hill;
- Momentum is a heavy ball rolling down the same hill;
- Visualization.

Nesterov accelerated SGD

- **Nesterov accelerated SGD:**

$$\begin{cases} v_{k+1} = \alpha_k \nabla f_i(w_k - \beta v_k) + \beta v_k, \\ w_{k+1} = w_k - v_{k+1}; \end{cases}$$

Nesterov accelerated SGD

- **Nesterov accelerated SGD:**

$$\begin{cases} v_{k+1} = \alpha_k \nabla f_i(w_k - \beta v_k) + \beta v_k, \\ w_{k+1} = w_k - v_{k+1}; \end{cases}$$

- Or equivalently:

$$w_{k+1} = w_k - \alpha_k \nabla f_i(w_k + \beta(w_k - w_{k-1})) + \beta(w_k - w_{k-1});$$

Nesterov accelerated SGD

- **Nesterov accelerated SGD:**

$$\begin{cases} v_{k+1} = \alpha_k \nabla f_i(w_k - \beta v_k) + \beta v_k, \\ w_{k+1} = w_k - v_{k+1}; \end{cases}$$

- Or equivalently:

$$w_{k+1} = w_k - \alpha_k \nabla f_i(w_k + \beta(w_k - w_{k-1})) + \beta(w_k - w_{k-1});$$

- Or equivalently:

$$\begin{cases} \tilde{w}_k = w_k + \beta(w_k - w_{k-1}), \\ w_{k+1} = \tilde{w}_k - \alpha_k \nabla f_i(\tilde{w}_k); \end{cases}$$

Nesterov accelerated SGD

- **Nesterov accelerated SGD:**

$$\begin{cases} v_{k+1} = \alpha_k \nabla f_i(w_k - \beta v_k) + \beta v_k, \\ w_{k+1} = w_k - v_{k+1}; \end{cases}$$

- Or equivalently:

$$w_{k+1} = w_k - \alpha_k \nabla f_i(w_k + \beta(w_k - w_{k-1})) + \beta(w_k - w_{k-1});$$

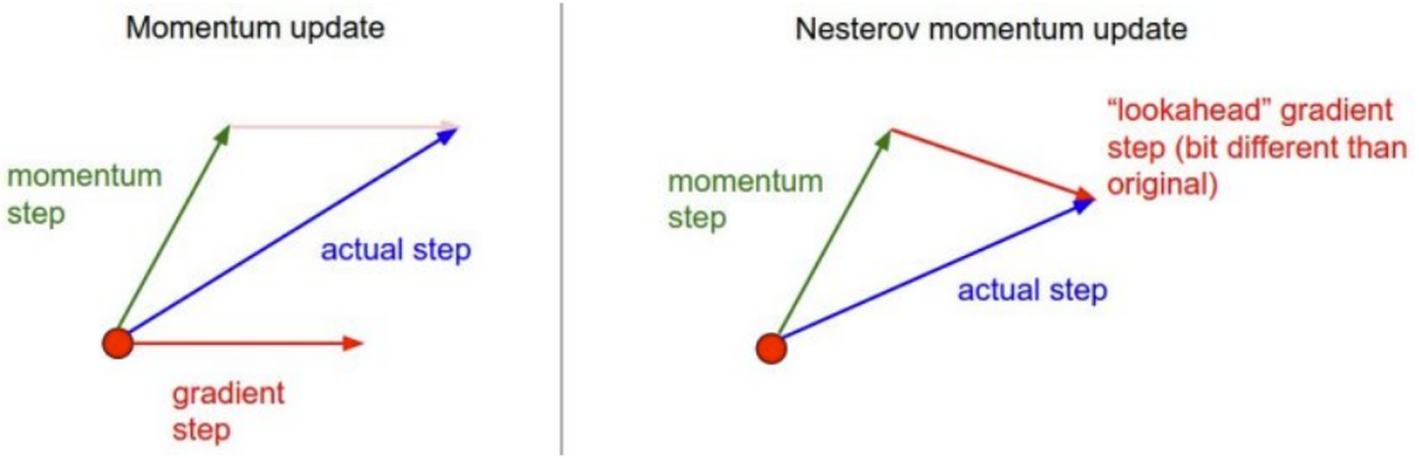
- Or equivalently:

$$\begin{cases} \tilde{w}_k = w_k + \beta(w_k - w_{k-1}), \\ w_{k+1} = \tilde{w}_k - \alpha_k \nabla f_i(\tilde{w}_k); \end{cases}$$

- The scheme was invented by Nesterov in 1983;
- Sutskever et al. (2013) popularized it in machine learning.

Nesterov accelerated SGD

- Different from the SGD with momentum.



Nesterov momentum. Instead of evaluating gradient at the current position (red circle), we know that our momentum is about to carry us to the tip of the green arrow. With Nesterov momentum we therefore instead evaluate the gradient at this "looked-ahead" position.

Momentum v.s. Nesterov

Different implementation in PyTorch

- SGD with Momentum/Nesterov in PyTorch.

---

input : γ (lr), θ_0 (params), $f(\theta)$ (objective), λ (weight decay),
μ (momentum), τ (dampening), *nesterov, maximize*

---

for $t = 1$ **to** $\ldots$ **do**
  $g_t \leftarrow \nabla_\theta f_t(\theta_{t-1})$
  if $\lambda \neq 0$
    $g_t \leftarrow g_t + \lambda\theta_{t-1}$
  if $\mu \neq 0$
    if $t > 1$
      $\mathbf{b}_t \leftarrow \mu\mathbf{b}_{t-1} + (1-\tau)g_t$
    else
      $\mathbf{b}_t \leftarrow g_t$
    if *nesterov*
      $g_t \leftarrow g_t + \mu\mathbf{b}_t$
    else
      $g_t \leftarrow \mathbf{b}_t$
  if *maximize*
    $\theta_t \leftarrow \theta_{t-1} + \gamma g_t$
  else
    $\theta_t \leftarrow \theta_{t-1} - \gamma g_t$

---

return θ_t

Coordinate adaptive learning rate

- **Limitation:** Use the same learning rate for all gradient coordinates;
- **Q:** Could we use different learning rates for different coordinates?

Coordinate adaptive learning rate

- **Limitation:** Use the same learning rate for all gradient coordinates;
- **Q:** Could we use different learning rates for different coordinates?
- In other words, for $1 \leq j \leq d$:

$$(w_{k+1})_j = (w_k)_j - \alpha_{k,j}(\nabla f_i(w_k))_j.$$

Or equivalently:

$$w_{k+1} = w_k - \begin{pmatrix} \alpha_{k,1} \\ \alpha_{k,2} \\ \cdots \\ \alpha_{k,d} \end{pmatrix} \odot \begin{pmatrix} (\nabla f_i(w_k))_1 \\ (\nabla f_i(w_k))_2 \\ \cdots \\ (\nabla f_i(w_k))_d \end{pmatrix}.$$

Improving stochastic gradient descent

Problem:

$$\min F(w) = \frac{1}{n} \sum_{i=0}^{n} f_i(w)$$

Issues:

- Let $X = \nabla f_I(w)$with I uniformly chosen at random in $\{1, \ldots, n\}$
- In SGD we use $X = \nabla f_I(w)$ as an approximation of $\mathbb{E}X = \nabla F(w)$
- How to reduce $\mathbf{V}X$?

Improving stochastic gradient descent

An idea

- Reduce it by finding C s.t. $\mathbb{E}C$ is "easy" to compute and such that C is highly correlated with X
- Let $Z_\alpha = \alpha(X - C) + \mathbb{E}C$ for $\alpha \in [0, 1]$. We have

$$\mathbb{E}Z_\alpha = \alpha\mathbb{E}X + (1 - \alpha)\mathbb{E}C$$

and

$$\mathbf{V}Z_\alpha = \alpha^2(\mathbf{V}X + \mathbf{V}C - 2\mathbb{C}(X, C))$$

- Standard variance reduction: $\alpha = 1$, so that $\mathbb{E}Z_\alpha = \mathbb{E}X$ (unbiased)

Improving stochastic gradient descent

Variance reduction of the gradient

In the iterations of SGD, replace $\nabla f_{i_k}(w^{(k-1)})$ by

$$\alpha(\nabla f_{i_k}(w^{(k-1)}) - \nabla f_{i_k}(\tilde{w})) + \nabla f(\tilde{w})$$

where $\tilde{w}$ is an "old" value of the iterate.

Important remark

- In these algorithms, the step-size η is kept **constant**
- Leads to **linearly convergent algorithms**, with a numerical complexity comparable to SGD!

Methods for finite sum minimization

- **GD**: at step k, use $\frac{1}{n}\sum_{i=0}^{n}\nabla f_i(w_k)$
- **SGD**: at step k, sample $i_k \sim \mathcal{U}[1;n]$, use $\nabla f_{i_k}(w_k)$
- **SAG**: at step k,
  - keep a "full gradient" $\frac{1}{n}\sum_{i=0}^{n}\nabla f_i(w_{k_i})$, with $w_{k_i} \in \{w_1, \dots, w_k\}$
  - sample $i_k \sim \mathcal{U}[1;n]$, use

$$\frac{1}{n}\left(\sum_{i=0}^{n}\nabla f_i(w_{k_i}) - \nabla f_{i_k}(w_{k_{i_k}}) + \nabla f_{i_k}(w_k)\right),$$

In other words:

- Keep in memory past gradients of all functionsf_i, $i = 1, \dots, n$
- Random selection $i_k \in \{1, \dots, n\}$ with replacement
- Iteration: $w_k = w_{k-1} - \frac{\eta}{n}\sum_{i=1}^{n} g_k(i)$ with $g_k(i) = \begin{cases} \nabla f_i(w_{k-1}) & \text{if } i = i_k \\ g_{k-1}(i) & \text{otherwise} \end{cases}$

SAG

- Keep in memory past gradients of all functions $f_i, i = 1, \ldots, n$
- Random selection $i_k \in \{1, \ldots, n\}$ with replacement
- Iteration: $w_k = w_{k-1} - \frac{\eta}{n} \sum_{i=1}^{n} g_k(i)$ with $g_k(i) = \begin{cases} \nabla f_i(w_{k-1}) & \text{if } i = i_k \\ g_{k-1}(i) & \text{otherwise} \end{cases}$

$\oplus$ update costs the same as SGD

$\ominus$ needs to store all gradients $\nabla f_i(w_{k_i})$ at "points in the past"

SAG

Stochastic Average Gradient

Initialization: initial weight vector $w^{(0)}$

Parameter: learning rate $\eta > 0$

For $k = 1, 2, \ldots$ until convergence do

- Pick uniformly at random i_k in $\{1, \ldots, n\}$
- $$g_k(i) = \begin{cases} \nabla f_i(w^{(k-1)}) & \text{if } i = i_k \\ g_{k-1}(i) & \text{otherwise} \end{cases}$$
- Compute $$w^{(k)} = w^{(k-1)} - \eta \left(\frac{1}{n} \sum_{i=1}^{n} g_k(i) \right)$$

Output: Return last $w^{(k)}$

SVRG

Stochastic Variance Reduced Gradient

Initialization: initial weight vector $\tilde{w}$

Parameters: learning rate $\eta > 0$, phase size (typically $m = n$ or $m = 2n$).

For $k = 1, 2, \ldots$ until convergence do

- Compute $\nabla f(\tilde{w})$
- Put $w^{(0)} \leftarrow \tilde{w}$
- For $t = 1, \ldots, m$

  Pick uniformly at random i_t in $\{1, \ldots, n\}$

  Apply the step

$$w^{(t+1)} \leftarrow w^{(t)} - \eta(\nabla f_{i_t}(w^{(t)}) - \nabla f_{i_t}(\tilde{w}) + \nabla f(\tilde{w}))$$

- Set $\tilde{w} \leftarrow \frac{1}{m} \sum_{t=1}^{m} w^{(t+1)}$

Output: Return $\tilde{w}$.

SAGA

Initialization: initial weight vector $w^{(0)}$

Parameters: learning rate $\eta > 0$

For all $i = 1, \ldots, n$, compute $g_0(i) \leftarrow \nabla f_i(w^{(0)})$

For $k = 1, 2, \ldots$ until convergence do

- Pick uniformly at random i_k in $\{1, \ldots, n\}$
- Compute $\nabla f_{i_k}(w^{(k-1)})$
- Apply

$$w^{(k)} \leftarrow w^{(k-1)} - \eta \left(\nabla f_{i_k}(w^{(k-1)}) - g_{k-1}(i_k) + \frac{1}{n} \sum_{i=1}^{n} g_{k-1}(i) \right)$$

- Store $g_k(i_k) \leftarrow \nabla f_{i_k}(w^{(k-1)})$

Output: Return last $w^{(k)}$

Variance reduced methods

Convergence rate for $f(\tilde{w}_k) - f(\theta_*)$, smooth objective f .

	SGD	GD	SAG
Convex	$O\left(\frac{1}{\sqrt{k}}\right)$	$O\left(\frac{1}{k}\right)$	
Stgly-Cvx	$O\left(\frac{1}{\mu k}\right)$	$O(e^{-\mu k})$	$O\left(1 - (\mu \wedge \frac{1}{n})\right)^k$

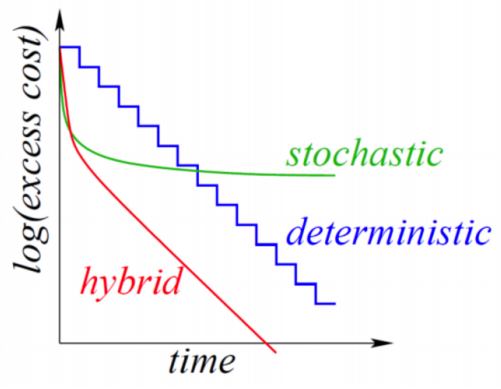


Figure 1: Comparison between GD, SGD, and SAG (Fig. from Sch-LeR-Bac-2013)

Summary

- Variance reduced algorithms can have both:
  - low iteration cost
  - fast asymptotic convergence

However:

- High precision is not always useful
- Typically not used in deep learning:
  - Memory constraints for SAG
  - Convergence to "bad" (?) minima $\Rightarrow$ bad generalization...

Bad generalization in Deep Learning

Reasoning:

- There are 2 types of local minima: flat and sharp.
- Algorithm that converge to "high precision" may converge to sharper minima.
- Sharp minima have poorer generalization performance.

Challenges in Deep Learning

Challenges

- Non convex $\Rightarrow$ Local minima
- Extremely large dimension
- Extremely large number of parameters (+ different scales)
- Bad conditioning + flat areas + saddle points

Ingredients of popular algorithms:

- First order
- Stochastic
- Momentum
- Different steps per coordinates : adaptive methods

Generalization and overfitting problems are poorly understood but:

- Noise helps
- "Too precise" methods (e.g. variance reduction, second order) are not used. e.g.: SVRG is great for convex, but not even implemented in Keras.

Adaptation: notations

- Same learning rate for all coordinates. Could we use a different learning rate for all coordinates ?
  i.e., for $1 \leq j \leq d$:

$$(w^k)_j = (w^{k-1})_j - \eta_{k,j}(\nabla f_{l_k}(w^{k-1}))_j$$

  Equivalently:

$$w^k = w^{k-1} - \begin{pmatrix} \eta_{k,1} \\ \eta_{k,2} \\ \dots \\ \eta_{k,d} \end{pmatrix} \odot \begin{pmatrix} \nabla f_{l_k}(w^{k-1})_1 \\ \nabla f_{l_k}(w^{k-1})_2 \\ \dots \\ \nabla f_{l_k}(w^{k-1})_d \end{pmatrix}$$

  - $g^k = \nabla f_{l_k}(w^{k-1})$ stochastic gradient at k-th iteration: $(w^k)_j = (w^{k-1})_j - \eta_{k,j}(g^k)_j$

ADAGRAD

Most following algorithms are in the following framework:

$$w_j^k = w_j^{k-1} - \eta_j^k g_j^k$$

Special choice for step-sizes:

$$w_j^k = w_j^{k-1} - \frac{\eta}{\sqrt{\sum_{\tau=1}^{k} (g_j^\tau)^2 + \epsilon}} g_j^k$$

ADAGRAD

ADAptive GRADient algorithm

Initialization: initial weight vector w^0

Parameter: learning rate $\eta > 0$

For $k = 1, 2, \ldots$ until *convergence* do, component-wise.

- For all $j = 1, \ldots, d$,

$$w_j^k \leftarrow w_j^{k-1} - \frac{\eta}{\sqrt{\sum_{\tau=1}^{k} (g_j^\tau)^2 + \epsilon}} g_j^k$$

Output: Return last $w^{(k)}$

ADAGRAD

Update equation for ADAGRAD

$$w^k \leftarrow w^{k-1} - \frac{\eta}{\sqrt{\sum_{\tau=1}^{k}(g^\tau)^2 + \epsilon}} \odot g^k$$

Pros:

- Different dynamic rates on each coordinate
- Dynamic rates grow as the inverse of the gradient magnitude:
  - Large/small gradients have small/large learning rates
  - The dynamic over each dimension tends to be of the same order
  - Interesting for neural networks in which gradient at different layers can be of different order of magnitude.
- Accumulation of gradients in the denominator act as a decreasing learning rate.

Cons:

- Susceptible to initial condition.
- Can be fought by increasing the learning rate thus making the algorithm sensitive to the choice of the learning rate.

ADAGRAD - Summary of parameters

ADAGRAD:

$$w_j^k = w_j^{k-1} - \eta_j^k g_j^k$$

Special choice for step-sizes:

$$w_j^k = w_j^{k-1} - \frac{\eta}{\sqrt{\sum_{\tau=1}^{k}(g_j^\tau)^2 + \epsilon}} g_j^k$$

ADAptive GRADient algorithm

Initialization: initial weight vector w^0

Parameter: learning rate $\eta > 0$

For $k = 1, 2, \ldots$ until *convergence* do, component-wise.

- For all $j = 1, \ldots, d$,

$$w_j^k \leftarrow w_j^{k-1} - \frac{\eta}{\sqrt{\sum_{\tau=1}^{k}(g_j^\tau)^2 + \epsilon}} g_j^k$$

Output: Return last w^k

Improving upon AdaGrad: RMS-prop

Idea : restricts the window of accumulated past gradients to some limited size through moving average.

- starting point w^0, constant ε,
- new params : decay rate $\rho > 0$ Update:

$$w_j^{k+1} = w_j^k - \frac{\eta_j^k}{\sqrt{C_{j,k} + \varepsilon}} g_j^k$$

Adagrad:

- $C_{j,k} = \sum_{\tau=1}^{k} (g_j^\tau)^2$
- $\eta_j^k = \eta$

RMSProp ((for Root Mean Square Propagation)):

- $C_{j,k} = \rho C_j^{k-1} + (1-\rho)(g_j^k)^2$
- $\eta_j^k = \eta$ constant.

RMSProp

Unpublished method, from the course of Geoff Hinton
http://www.cs.toronto.edu/~tijmen/csc321/slides/lecture_slides_lec6.pdf

RMSProp algorithm

Initialization: initial weight vector w^0

Parameters: learning rate $\eta > 0$ (default $\eta = 0.001$), decay rate ρ (default $\rho = 0.9$)

For $k = 1, 2, \ldots$ until convergence do

- First, compute the accumulated gradient

$$\overline{(\nabla f)^2}^{(k)} = \rho \overline{(\nabla f)^2}^{(k-1)} + (1 - \rho)(g^k)^2$$

- Compute

$$w^{(k+1)} \leftarrow w^{(k)} - \frac{\eta}{\sqrt{\overline{(\nabla f)^2}^{(k)} + \varepsilon}} \odot g^k$$

Output: Return last $w^{(k)}$

Improving upon AdaGrad & RMSProp: AdaDelta

Idea :RMSProp + Second order style approach.

Less sensitivity to initial parameters.

Update:

$$w_j^{k+1} = w_j^k - \frac{\eta_j^k}{\sqrt{C_{j,k} + \varepsilon}} g_j^k$$

Adagrad:

- $C_{j,k} = \sum_{\tau=1}^{k} (g_j^\tau)^2$
- $\eta_j^k = \eta$

RMS prop:

- $C_{j,k} = \rho C_{j,k-1} + (1-\rho)(g_j^k)^2$
- $\eta_j^k = \eta$ constant.

Adadelta:

- $C_{j,k} = \rho C_{j,k-1} + (1-\rho)(g_j^k)^2$
- η_j^k variable.

ADADELTA

Update equation for adadelta

$$w^{(k+1)} = w^{(k)} - \frac{\sqrt{\overline{(\Delta w)^2}^{(k-1)} + \varepsilon}}{\sqrt{\overline{(\nabla f)^2}^{(k)} + \varepsilon}} \odot g^k$$

Interpretation:

- The numerator keeps the size of the previous step in memory and enforce larger steps along directions in which large steps were made.
- The denominator keeps the size of the previous gradients in memory and acts as a decreasing learning rate. Weights are lower than in Adagrad due to the decay rate ρ.
  Inspired by second order methods (unit invariance + Hessian approximation)

$$\Delta w \simeq (\nabla^2 f)^{-1} \nabla f.$$

Roughly,

$$\Delta w = \frac{\frac{\partial f}{\partial w}}{\frac{\partial^2 f}{\partial w^2}} \Leftrightarrow \frac{1}{\frac{\partial^2 f}{\partial w^2}} = \frac{\Delta w}{\frac{\partial f}{\partial w}}.$$

ADADELTA

AdaDelta algorithm

Initialization: initial weight vector $w^{(0)}$, $\overline{(\nabla f)^2}^0 = 0$, $\overline{(\Delta x)^2}^0 = 0$

Parameters: decay rate $\rho > 0$, constant ε,

For $k = 1, 2, \ldots$ until *convergence* do

- For all $j = 1, \ldots, d$,
  - Compute the accumulated gradient

$$\overline{(\nabla f)^2}^{(k)} = \rho \overline{(\nabla f)^2}^{(k-1)} + (1-\rho)(g^k)^2$$

  - Compute the update

$$w^{(k)} = w^{(k-1)} - \frac{\sqrt{\overline{(\Delta w)^2}^{(k-1)} + \varepsilon}}{\sqrt{\overline{(\nabla f)^2}^{(k)} + \varepsilon}} \odot g^k$$

  - Compute the aggregated update

$$\overline{(\Delta w)^2}^{(k)} = \rho \overline{(\Delta w)^2}^{(k-1)} + (1-\rho)(w^{(k+1)} - w^{(k)})^2$$

Output: Return last $w^{(k)}$

ADAM: ADAptive Moment estimation

General idea: store the estimated first and second moment of the gradient and use them to update the parameters.

Equations - first and second moment

Let m_k be an exponentially decaying average over the past gradients

$$m_k = \beta_1 m_{k-1} + (1 - \beta_1) g^k$$

Similarly, let v_t be an exponentially decaying average over the past square gradients

$$v_k = \beta_2 v_{k-1} + (1 - \beta_2)(g^k)^2.$$

Initialization: $m_0 = v_0 = 0$.

ADAM: ADAptive Moment estimation

With this initialization, estimates m_t and v_t are biased towards zero in the early steps of the gradient descent.

Final equations

$$\tilde{m}_k = \frac{m_k}{1-\beta_1^k} \quad \tilde{v}_k = \frac{v_k}{1-\beta_2^k}.$$

$$w^{(k)} = w^{(k-1)} - \frac{\eta}{\sqrt{\tilde{v}_k} + \varepsilon}\tilde{m}_k.$$

ADAM algorithm

Initialization: $m_0 = 0$ (Initialization of the first moment vector), $v_0 = 0$ (Initialization of the second moment vector), w_0 (initial vector of parameters).

Parameters: stepsize η (default $\eta = 0.001$), exponential decay rates for the moment estimates $\beta_1, \beta_2 \in [0, 1)$ (default: $\beta_1 = 0.9, \beta_2 = 0.999$), numeric constant ε (default $\varepsilon = 10^{-8}$).

For $k = 1, 2, \ldots$ until *convergence* do

- Compute first and second moment estimate

$$m^{(k)} = \beta_1 m^{(k-1)} + (1 - \beta_1) g^k \quad v^{(k)} = \beta_2 v^{(k-1)} + (1 - \beta_2)(g^k)^2.$$

- Compute their respective correction

$$\tilde{m}^{(k)} = \frac{m^{(k)}}{1 - \beta_1^k} \quad \tilde{v}^{(k)} = \frac{v^{(k)}}{1 - \beta_2^k}.$$

- Update the parameters accordingly

Adamax algorithm

Initialization: $m_0 = 0$ (Initialization of the first moment vector), $u_0 = 0$ (Initialization of the exponentially weighted infinity norm), w_0 (initial vector of parameters).

Parameters: stepsize η (default $\eta = 0.001$), exponential decay rates for the moment estimates $\beta_1, \beta_2 \in [0, 1)$ (default: $\beta_1 = 0.9, \beta_2 = 0.999$)

For $k = 1, 2, \ldots$ until *convergence* do

- Compute first moment estimate and its correction

$$m^{(k)} = \beta_1 m^{(k-1)} + (1 - \beta_1) g^k, \quad \tilde{m}^{(k)} = \frac{m^{(k)}}{1 - \beta_1^k}$$

- Compute the quantity

$$u^{(k)} = max(\beta_2 u^{(k-1)}, |g^k|).$$

- Update the parameters accordingly

$$w^{(k+1)} = w^{(k)} - \frac{\eta}{u^{(k)}} \odot \tilde{m}^{(k)}$$

Animation of Stochastic Gradient algorithms

https://imgur.com/a/Hqolp

Credits to Alec Radford for the animations.

Summary

What we have seen so far

- Why optimization is important, what makes it difficult
- Simple first order methods, from GD to SGD
- Acceleration techniques (momentum, Nesterov)
- Advanced first order methods, variance reduction and coordinate adaptive step sizes

Newton Methods

Newton's method

- At point x_k, if $\nabla^2 f(x_k)$ is p.d., the function $f(x)$ can be approximated by a quadratic function based on the Taylor expansion:

$$f(x) \cong f(x_k) + \nabla f(x_k)^T (x - x_k) + \frac{1}{2}(x - x_k)^T \nabla^2 f(x_k)(x - x_k) \quad (15)$$

- The minimizer of (15) is given by

$$\begin{aligned} \nabla f(x) = 0 \quad &\Rightarrow \quad \nabla f(x_k) + \nabla^2 f(x_k)(x - x_k) = 0 \\ &\Rightarrow \quad x = x_k - \left[\nabla^2 f(x_k)\right]^{-1} \nabla f(x_k), \\ &\Rightarrow \quad x = x_k - \left[\nabla^2 f(x_k)\right]^{-1} \nabla f(x_k) \end{aligned}$$

  where $-\left[\nabla^2 f(x_k)\right]^{-1} \nabla f(x_k)$ is called Newton's direction.

- Define

$$x_{k+1} = x_k - \left[\nabla^2 f(x_k)\right]^{-1} \nabla f(x_k),$$

  and the resulting method of computing $\{x_k\}$ is called the Newton's method.

Newton's method

Newton's method

Given x_0, compute $x_{k+1} = x_k - \left[\nabla^2 f(x_k)\right]^{-1} \nabla f(x_k)$, $k \leftarrow k+1$.

A key requirement for Newton's method is the p.d. of $\nabla^2 f(x_k)$.

Theorem 1.8

Assume that $f \in C^2$ and that $\nabla^2 f(x)$ is Lipschitz continuous in $B(x^*, \delta)$ with constant γ, where x^* is a local minimizer. If $\nabla^2 f(x^*)$ is p.d. and $x^{(0)}$ is close enough to x^*. Then Newton's method is well defined and $\{x^{(k)}\}$, generated by Newton's method, converges to x^* q-quadratically.

Newton's method

Proof:

- Since $f \in C^2$ and $\nabla^2 f(x^*)$ is p.d., there exists a $\delta_1 > 0$ ($\delta_1 \leq \delta$) s.t.
  (a) $\nabla^2 f(x)$ is p.d. $\forall\, x \in B(x^*, \delta_1)$
  (b) $\| \left[\nabla^2 f(x)\right]^{-1} \| \leq 2M$, $\forall\, x \in B(x^*, \delta_1)$, and $M = \|[\nabla^2 f(x^*)]^{-1}\|$.
- Let $\delta_2 = \min\{\delta_1, \frac{1}{2\gamma M}\}$.
- Prove that if $x^{(0)} \in B(x^*, \delta_2)$, $x^{(k)} \in B(x^*, \delta_2)$ $\forall\, k \geq 1$.
- Since $\delta_2 \leq \delta_1 \leq \delta$, $\nabla^2 f(x)$ is Lipschitz continuous in $B(x^*, \delta_2)$. Obviously, we have

$$\|\nabla f(x+p) - \nabla f(x) - \nabla^2 f(x)p\| \leq \frac{\gamma}{2}\|p\|^2 \quad \forall\, x+p \in B(x^*, \delta_2). \tag{16}$$

  Note that $x^{(0)}$ is assumed to be close enough to x^*. We can assume that $x^{(0)} \in B(x^*, \delta_2)$.

Newton's method

- Let $x = x^{(k)}$ and $p = x^* - x^{(k)}$ in (16), we have

$$\left\|\nabla f(x^*) - \nabla f(x^{(k)}) + \nabla^2 f(x^{(k)})(x^{(k)} - x^*)\right\| \leq \frac{\gamma}{2}\left\|x^{(k)} - x^*\right\|^2.$$

$$\Rightarrow \left\|-\left[\nabla^2 f(x^{(k)})\right]^{-1}\nabla f(x^{(k)}) + x^{(k)} - x^*\right\| \leq \frac{\gamma}{2}\left\|x^{(k)} - x^*\right\|^2 \cdot \left\|\left[\nabla^2 f(x^{(k)}\right]^{-1}\right\|$$

$$\leq \gamma M \|x^{(k)} - x^*\|^2$$

$$\Rightarrow \left\|x^{(k+1)} - x^*\right\| \leq \gamma M \left\|x^{(k)} - x^*\right\|^2$$

$$\Rightarrow \left\|x^{(k+1)} - x^*\right\| \leq \frac{1}{2}\left\|x^{(k)} - x^*\right\|$$

- By using induction on the above procedure, it is easy to see $x^{(k)} \in B(x^*, \delta_2)$. Therefore, Newton's method is well defined while the last inequality implies the q-quadratic convergence. □

Newton's method

Attractiveness:

1. Simple
2. Quadratic convergence (Very fast)

Weakness:

1. Requiring the p.d. of all $\nabla^2 f(x)$ ($O(n^3)$ flops)
2. Matrix inversion (or solving a system of linear equations) ($O(n^3)$ flops)
3. Second directives required
4. Locally convergent

Quasi-Newton methods

Background

Consider the following methods:

Newton's method
- $+$: very fast convergence if applicable
- $-$: expansive, second-order information matrix inversion

Steepest descent method
- $+$: simple and inexpansive, guarantee descent
- $-$: slow convergence

Quasi-Newton methods

Idea for new method

(a) no second-order information, i.e. no Hessian;

(b) fast convergence.

Remember

(i) first-order information normally gives slow (linear) convergence;

(ii) second-order information normally gives fast (quadratic) convergence.

What if we do not use the Hessian matrix but approximate it?
—This is the essence of quasi-Newton methods.

Quasi-Newton methods

- In quasi-Newton methods, the inverse of the Hessian matrix is approximated in each iteration by a p.d. matrix, say H_k, where k is the iteration index.
- The kth iteration has the following basic structure:
  (a) set $p_k = -H_k g_k$, $(g_k = \nabla f(x_k))$
  (b) line search along p_k giving $x_{k+1} = x_k + \lambda_k p_k$,
  (c) update H_k giving H_{k+1}
- The initial matrix H_0 can be any positive definite symmetric matrix, the choice $H_0 = I$ often made.
- Potential advantages of the method are:
  (i) only first-order information required;
  (ii) H_k p.d. implies the descent property; and
  (iii) $O(n^2)$ multiplications per iteration.

Quasi-Newton update formula

Much of interest lies in the updating formula which enables the symmetric matrix H_{k+1} to be calculated from H_k. One way of doing this is the following. If the difference

$$s_k = x_{k+1} - x_k \tag{17}$$

$$y_k = \nabla f(x_{k+1}) - \nabla f(x_k) = g_{k+1} - g_k \tag{18}$$

are defined, then the Taylor series gives

$$y_k = H_k^{-1} s_k + o(\|s_k\|). \tag{19}$$

Since s_k and y_k can only be calculated after the line search is completed, the matrix H_k does not usually relate them correctly. Thus H_{k+1} is chosen so that it does correctly relate these difference, that is so that

$$H_{k+1} y_k = s_k. \tag{20}$$

This is sometimes called the quasi-Newton condition.

Quasi-Newton update formula

Note: H_k is an approximation for the inverse of Hessian. Sometimes a direct approximation for the Hessian matrix is preferred, in this case (20) becomes

$$G_{k+1} s_k = y_k. \tag{21}$$

This is normally called the secant equation.

Quasi-Newton update formula

Perhaps the simplest possibility is to have the following symmetric rank one update

$$H_{k+1} = H_k + auu^T. \tag{22}$$

Since (20) must be met, we have

$$(H_k + auu^T)y_k = s_k \rightarrow auu^T y_k = s_k - H_k y_k.$$

Since a can be scaled up or down, we have

$$u = s_k - H_k y_k.$$

In the case $au^T y_k = 1 \Rightarrow a = \frac{1}{u^T y_k}$. This gives the rank one update formula

$$H_{k+1} = H_k + \frac{(s_k - H_k y_k)(s_k - H_k y_k)^T}{(s_k - H_k y_k)^T y_k}. \tag{23}$$

Comments:

(1) If H_k is p.d., H_{k+1} may not be p.d.

(2) The denominator in (23) may be zero so H_{k+1} may not be defined.

Quasi-Newton update formula

A more flexible formula is obtained by allowing the correction to be of rank two, and such a formula can always be written

$$H_{k+1} = H_k + auu^T + bvv^T.$$

Again, we must have

$$H_{k+1}y_k = s_k.$$

This gives

$$s_k = H_k y_k + auu^T y_k + bvv^T y_k. \tag{24}$$

Now u and v are no longer determined uniquely. However, an obvious choice in (24) is to try $u = s_k$ and $v = H_k y_k$. Then $au^T y_k = 1$ and $bv^T y_k = -1$ determine a and b. Thus

$$H_{k+1} = H_k + \frac{s_k s_k^T}{s_k^T y_k} - \frac{H_k y_k y_k^T H_k}{y_k^T H_k y_k}. \tag{25}$$

This formula was known as DFP (Davidon, Fletcher, Powell) formula.

Quasi-Newton update formula

Theorem 1.9

If H_k is p.d. and $s_k^T y_k > 0$, H_{k+1} in (25) is also p.d.

Proof: For any $z \neq 0$, it is sufficient to prove

$$z^T \left(H_k + \frac{s_k s_k^T}{s_k^T y_k} - \frac{H_k y_k y_k^T H_k}{y_k^T H_k y_k} \right) z > 0.$$

In the rest of the proof, the subscript k will be omitted. Since H is p.d., we can write $H = LL^T$, and let $\boldsymbol{a} = L^T z$ and $\boldsymbol{b} = L^T y$, then

$$z^T \left(H - \frac{H y y^T H}{y^T H y} \right) z = \mathbf{a}^T \boldsymbol{a} - \frac{(\boldsymbol{a}^T \boldsymbol{b})^2}{\boldsymbol{b}^T \boldsymbol{b}} \geq 0$$

by Cauchy's inequality. Since $z \neq 0$, equality holds only if $\boldsymbol{a} \propto \boldsymbol{b}$, that is if $z \propto y$. But since $s^T y > 0$,

$$z^T \left(\frac{s s^T}{s^T y} \right) z \geq 0$$

which becomes a strict inequality if $z \propto y$. Then the result is proved. □

How to guarantee $s_k^T y_k > 0$?

Notice that $s_k^T y_k = (x_{k+1} - x_k)^T g_{k+1} - (x_{k+1} - x_k)^T g_k$.

- Since $(x_{k+1} - x_k)^T g_k = \lambda_k p_k^T g_k$ and p_k is a descent direction,

$$(x_{k+1} - x_k)^T g_k < 0, \quad \text{then} \quad -(x_{k+1} - x_k)^T g_k > 0.$$

- Now look at $(x_{k+1} - x_k)^T g_{k+1} = \lambda_k p_k^T g_{k+1}$.
  - If an exact line search is used, i.e.

$$\lambda_k = \arg\min_{\lambda} f(x_k + \lambda p_k) \quad \Rightarrow \quad p_k^T g_{k+1} = 0.$$

    This together with $-(x_{k+1} - x_k)^T g_k > 0$ imply $s_k^T y_k > 0$.
  - If an inexact line search is used, say backtracking, from the Goldstein rule, we have

$$g_{k+1}^T s_k \geq \beta g_k^T s_k > g_k^T s_k \quad \Rightarrow \quad s_k^T y_k > 0.$$

- So $s_k^T y_k > 0$ can always be achieved.

BFGS

- The most important formula was suggested by Broyden, Fletcher, Goldfarb, and Shanno independently in 1970, and is known as BFGS formula

$$H_{k+1}^{BFGS} = H_k + \left(1 + \frac{y_k^T H_k y_k}{s_k^T y_k}\right) \frac{s_k s_k^T}{s_k^T y_k} - \left(\frac{s_k y_k^T H_k + H_k y_k s_k^T}{s_k^T y_k}\right). \quad (26)$$

- If H^{-1} is denoted by B which approximates the Hessian, then it can be verified (with $BH = I$) that

$$B_{k+1}^{BFGS} = B_k + \frac{y_k y_k^T}{y_k^T s_k} - \frac{B_k s_k s_k^T B_k}{s_k^T B_k s_k}. \quad (27)$$

- This resembles the DFP formula (25) but with the interchanges $B \longleftrightarrow H$ and $y \longleftrightarrow s$ having been made. Formulae related in this way are said to be dual or complementary. Similarly, it follows from (26) that

$$B_{k+1}^{DFP} = B_k + \left(1 + \frac{s_k^T B_k s_k}{y_k^T s_k}\right) \frac{y_k y_k^T}{y_k^T s_k} - \left(\frac{y_k s_k^T B_k + B_k s_k y_k^T}{y_k^T s_k}\right). \quad (28)$$

BFGS

Comments:

(1) Taking the dual operation twice restores the original formula.

(2) The quasi-Newton condition (20) is preserved by the duality operation.

(3) It can be established that the rank one formula is self-dual.

(4) By far, the BFGS method is best known quasi-Newton method.

For BFGS, we can prove

1. If B_k^{BFGS} is p.d. and $s_k^T y_k > 0$, B_{k+1}^{BFGS} is also p.d.
2. $\{H_k^{BFGS}\}$ is bounded, therefore the BFGS method is well-defined.
3. If x^* is a locally minimizer and $x_k \to x^*$, then x_k converges x^* q-superlinearly.

Limited-memory BFGS method

- The inverse Hessian approximations generated by the BFGS formula (26) are usually dense, even when the true Hessian is sparse.
- The cost of storing and working with these approximations is prohibitive when n is large.
- To address this issue, a limited-memory variant of the BFGS method, which is called L-BFGS method.
- The L-BFGS method only requires storing a sequence of vectors g_k and s_k from the most recent iterations to compute $H_k \nabla f(x_k)$ without construting H_k explicitly.
- It turns out that the L-BFGS method is fairly robust, inexpensive, and easy to implement.

Limited-memory BFGS method

- Note that the BFGS formula (26) can be rewritten as

$$H_{k+1} = V_k^T H_k V_k + \rho_k s_k s_k^T \tag{29}$$

  with $s_k = x_{k+1} - x_k, y_k = \nabla f(x_{k+1}) - \nabla f(x_k), \rho_k = \frac{1}{y_k^T s_k}$, and $V_k = I - \rho_k y_k s_k^T$.
- At iteration k, we denote by x_k the current iterate and by $\{s_i, y_i\}$ $i = k - m, \cdots, k - 1$ the set of vector pairs including the curvature information from the m most recent iterations.
- We choose some initial inverse Hessian approximation H_k^0 (e.g., $H_k^0 = \gamma_k I, \gamma_k > 0$), and then apply the BFGS formula (29) repeatedly.

Limited-memory BFGS method

One can easily show that the approximation marix H_k is given by the following formula:

$$\begin{aligned} H_k =& (V_{k-1}^T, \cdots, V_{k-m}^T) H_k^0 (V_{k-m}, \cdots, V_{k-1}) \\ &+ \rho_{k-m} (V_{k-1}^T, \cdots, V_{k-m+1}^T) s_{k-m} s_{k-m}^T (V_{k-m+1}, \cdots, V_{k-1}) \\ &+ \rho_{k-m+1} (V_{k-1}^T, \cdots, V_{k-m+2}^T) s_{k-m+1} s_{k-m+1}^T (V_{k-m+2}, \cdots, V_{k-1}) \\ &+ \cdots \\ &+ \rho_{k-2} V_{k-1}^T s_{k-2} s_{k-2}^T V_k - 1 \\ &+ \rho_{k-1} s_{k-1} s_{k-1}^T. \end{aligned}$$

Note that, in contrast to the standard BFGS iteration, the initial approximation H_k^0 is allowed to vary from iteration to iteration.

Limited-memory BFGS method

With the above expression we can derive a two-loop recursive procedure to compute the product $H_k \nabla f(x_k)$ efficiently.

L-BFGS two-loop recursion

- $q = \nabla f_k$;
- for $i = k-1, k-2, \ldots, k-m$

  Compute $\alpha_i = \rho_i s_i^T q$ and update $q = q - \alpha_i y_i$;
- end for
- $r = H_k^0 q$;
- for $i = k-m, k-m+1, \ldots, k-1$

  Compute $\beta = \rho_i y_i^T r$ and update $r = r + s_i(\alpha_i - \beta)$;
- end for
- Output: $r = H_k \nabla f_k$

Hands-on practice session using Python

- The code tutorial is available at:
  link to code
- Please first install the Google Colab:
  link to Colab
- After that, it is possible to open the tutorial with the Colab.