First-Order Optimization

Optimization Techniques (ENGG*6140)

School of Engineering,
University of Guelph, ON, Canada

Course Instructor: Benyamin Ghojogh
Winter 2023

Fundamental Theorem of Calculus

Fundamental theorem of calculus

Lemma (Fundamental theorem of calculus for multivariate functions)

Consider a differentiable function $f(.)$ with domain $\mathcal{D}$. For any $\boldsymbol{x}, \boldsymbol{y} \in \mathcal{D}$, we have:

$$\begin{aligned} f(\boldsymbol{y}) &= f(\boldsymbol{x}) + \nabla f(\boldsymbol{x})^\top (\boldsymbol{y} - \boldsymbol{x}) + \int_0^1 \Big(\nabla f\big(\boldsymbol{x} + t(\boldsymbol{y} - \boldsymbol{x})\big) - \nabla f(\boldsymbol{x})\Big)^\top (\boldsymbol{y} - \boldsymbol{x}) dt \\ &= f(\boldsymbol{x}) + \nabla f(\boldsymbol{x})^\top (\boldsymbol{y} - \boldsymbol{x}) + o(\boldsymbol{y} - \boldsymbol{x}), \end{aligned} \tag{1}$$

where $o(.)$ is the small-o complexity.

Lemma (Corollary of the fundamental theorem of calculus)

Consider a differentiable function $f(.)$, with domain $\mathcal{D}$, whose gradient is L-smooth:

$$|\nabla f(\boldsymbol{x}) - \nabla f(\boldsymbol{y})| \leq L \|\boldsymbol{x} - \boldsymbol{y}\|_2, \quad \forall \boldsymbol{x}, \boldsymbol{y} \in \mathcal{D}. \tag{2}$$

For any $\boldsymbol{x}, \boldsymbol{y} \in \mathcal{D}$, we have:

$$f(\boldsymbol{y}) \leq f(\boldsymbol{x}) + \nabla f(\boldsymbol{x})^\top (\boldsymbol{y} - \boldsymbol{x}) + \frac{L}{2} \|\boldsymbol{y} - \boldsymbol{x}\|_2^2. \tag{3}$$

Fundamental theorem of calculus

Proof for the corollary of the fundamental theorem of calculus:

$$
\begin{aligned}
f(\boldsymbol{y}) &\overset{(1)}{=} f(\boldsymbol{x}) + \nabla f(\boldsymbol{x})^\top (\boldsymbol{y}-\boldsymbol{x}) + \int_0^1 \Big(\nabla f\big(\boldsymbol{x}+t(\boldsymbol{y}-\boldsymbol{x})\big) - \nabla f(\boldsymbol{x})\Big)^\top (\boldsymbol{y}-\boldsymbol{x}) dt \\
&\overset{(a)}{\leq} f(\boldsymbol{x}) + \nabla f(\boldsymbol{x})^\top (\boldsymbol{y}-\boldsymbol{x}) + \int_0^1 \|\nabla f\big(\boldsymbol{x}+t(\boldsymbol{y}-\boldsymbol{x})\big) - \nabla f(\boldsymbol{x})\|_2 \|\boldsymbol{y}-\boldsymbol{x}\|_2 dt \\
&\overset{(b)}{\leq} f(\boldsymbol{x}) + \nabla f(\boldsymbol{x})^\top (\boldsymbol{y}-\boldsymbol{x}) + \int_0^1 Lt\|\boldsymbol{y}-\boldsymbol{x}\|_2^2 dt \\
&= f(\boldsymbol{x}) + \nabla f(\boldsymbol{x})^\top (\boldsymbol{y}-\boldsymbol{x}) + L\|\boldsymbol{y}-\boldsymbol{x}\|_2^2 \int_0^1 t dt \\
&= f(\boldsymbol{x}) + \nabla f(\boldsymbol{x})^\top (\boldsymbol{y}-\boldsymbol{x}) + \frac{L}{2}\|\boldsymbol{y}-\boldsymbol{x}\|_2^2,
\end{aligned}
$$

where (a) is because of the Cauchy-Schwarz inequality and (b) is because, according to Eq. (2), $|\nabla f(\boldsymbol{x}) - \nabla f(\boldsymbol{y})| \leq L\,\|\boldsymbol{x}-\boldsymbol{y}\|_2, \forall \boldsymbol{x}, \boldsymbol{y} \in \mathcal{D}$, we have:
$\|\nabla f(\boldsymbol{x}+t(\boldsymbol{y}-\boldsymbol{x})) - \nabla f(\boldsymbol{x})\|_2 \leq L\|\boldsymbol{x}+t(\boldsymbol{y}-\boldsymbol{x})-\boldsymbol{x}\|_2 = Lt\|\boldsymbol{y}-\boldsymbol{x}\|_2$. Q.E.D.

Gradient Descent

Gradient descent: introduction

- **Gradient descent** is one of the fundamental first-order methods.
- It was first suggested by Cauchy in 1874 [1] and Hadamard in 1908 [2] and its convergence was later analyzed in [3].
- Unconstrained optimization:

$$\underset{\boldsymbol{x}}{\text{minimize}} \quad f(\boldsymbol{x}). \tag{4}$$

- In numerical optimization for unconstrained optimization, we start with a random feasible initial point and iteratively update it by step $\Delta \boldsymbol{x}$:

$$\boldsymbol{x}^{(k+1)} := \boldsymbol{x}^{(k)} + \Delta \boldsymbol{x}. \tag{5}$$

- Continue until we converge to (or get sufficiently close to) the desired optimal point $\boldsymbol{x}^*$.
- The step $\Delta \boldsymbol{x}$ is also denoted by $\boldsymbol{p}$ in the literature, i.e., $\boldsymbol{p} := \Delta \boldsymbol{x}$.

Gradient descent: step size

- Let the function $f(.)$ be differentiable and its gradient be L-smooth.
- Recall Eq. (3) from the corollary of the fundamental theorem of calculus:

$$f(\boldsymbol{y}) \leq f(\boldsymbol{x}) + \nabla f(\boldsymbol{x})^\top (\boldsymbol{y} - \boldsymbol{x}) + \frac{L}{2}\|\boldsymbol{y} - \boldsymbol{x}\|_2^2.$$

If we set $\boldsymbol{x} = \boldsymbol{x}^{(k)}$ and $\boldsymbol{y} = \boldsymbol{x}^{(k+1)} = \boldsymbol{x}^{(k)} + \Delta\boldsymbol{x}$ in this equation, we have:

$$f(\boldsymbol{x}^{(k)} + \Delta\boldsymbol{x}) \leq f(\boldsymbol{x}^{(k)}) + \nabla f(\boldsymbol{x}^{(k)})^\top \Delta\boldsymbol{x} + \frac{L}{2}\|\Delta\boldsymbol{x}\|_2^2$$

$$\implies f(\boldsymbol{x}^{(k)} + \Delta\boldsymbol{x}) - f(\boldsymbol{x}^{(k)}) \leq \nabla f(\boldsymbol{x}^{(k)})^\top \Delta\boldsymbol{x} + \frac{L}{2}\|\Delta\boldsymbol{x}\|_2^2. \tag{6}$$

- Until reaching the minimum, we want to decrease the cost function $f(.)$ in every iteration; hence, we desire:

$$f(\boldsymbol{x}^{(k)} + \Delta\boldsymbol{x}) - f(\boldsymbol{x}^{(k)}) < 0. \tag{7}$$

- According to Eq. (6), one way to achieve Eq. (7) is:

$$\nabla f(\boldsymbol{x}^{(k)})^\top \Delta\boldsymbol{x} + \frac{L}{2}\|\Delta\boldsymbol{x}\|_2^2 < 0.$$

Gradient descent: step size

- We have:

$$\nabla f(\boldsymbol{x}^{(k)})^\top \Delta \boldsymbol{x} + \frac{L}{2}\|\Delta \boldsymbol{x}\|_2^2 < 0.$$

- Hence, we should minimize $\nabla f(\boldsymbol{x}^{(k)})^\top \Delta \boldsymbol{x} + \frac{L}{2}\|\Delta \boldsymbol{x}\|_2^2$ w.r.t. $\Delta \boldsymbol{x}$:

$$\underset{\Delta \boldsymbol{x}}{\text{minimize}}\ \nabla f(\boldsymbol{x}^{(k)})^\top \Delta \boldsymbol{x} + \frac{L}{2}\|\Delta \boldsymbol{x}\|_2^2. \tag{8}$$

- This function is convex w.r.t. $\Delta \boldsymbol{x}$ and we can optimize it by setting its derivative to zero:

$$\begin{aligned} &\frac{\partial}{\partial \Delta \boldsymbol{x}}(\nabla f(\boldsymbol{x}^{(k)})^\top \Delta \boldsymbol{x} + \frac{L}{2}\|\Delta \boldsymbol{x}\|_2^2) = \nabla f(\boldsymbol{x}^{(k)}) + L\Delta \boldsymbol{x} \overset{\text{set}}{=} \boldsymbol{0} \\ &\implies \Delta \boldsymbol{x} = -\frac{1}{L}\nabla f(\boldsymbol{x}^{(k)}). \end{aligned} \tag{9}$$

- So, we have:

$$\boldsymbol{x}^{(k+1)} := \boldsymbol{x}^{(k)} - \frac{1}{L}\nabla f(\boldsymbol{x}^{(k)}).$$

Gradient descent: step size

- Recall Eqs. (9) and (6):

$$\Delta \boldsymbol{x} = -\frac{1}{L}\nabla f(\boldsymbol{x}^{(k)}).$$

$$f(\boldsymbol{x}^{(k)} + \Delta \boldsymbol{x}) - f(\boldsymbol{x}^{(k)}) \leq \nabla f(\boldsymbol{x}^{(k)})^\top \Delta \boldsymbol{x} + \frac{L}{2}\|\Delta \boldsymbol{x}\|_2^2.$$

- Using the first equation in the second equation gives:

$$\begin{aligned} f(\boldsymbol{x}^{(k)} + \Delta \boldsymbol{x}) - f(\boldsymbol{x}^{(k)}) &\leq -\frac{1}{L}\nabla f(\boldsymbol{x}^{(k)})^\top \nabla f(\boldsymbol{x}^{(k)}) + \frac{L}{2}\|\frac{-1}{L}\nabla f(\boldsymbol{x}^{(k)})\|_2^2 \\ &= -\frac{1}{L}\|\nabla f(\boldsymbol{x}^{(k)})\|_2^2 + \frac{L}{2}\|\frac{-1}{L}\nabla f(\boldsymbol{x}^{(k)})\|_2^2 \\ &= -\frac{1}{L}\|\nabla f(\boldsymbol{x}^{(k)})\|_2^2 + \frac{L}{2}(\frac{1}{L})^2\|\nabla f(\boldsymbol{x}^{(k)})\|_2^2 \\ &= -\frac{1}{2L}\|\nabla f(\boldsymbol{x}^{(k)})\|_2^2 \leq 0, \end{aligned}$$

which satisfies Eq. (7). Eq. (9) means that it is better to move toward a scale of minus gradient for updating the solution. This inspires the name of algorithm which is *gradient descent*.

Gradient descent: step size

- The problem is that often we either do not know the Lipschitz constant L or it is hard to compute. Hence, rather than Eq. (9), $\Delta \boldsymbol{x} = -\frac{1}{L}\nabla f(\boldsymbol{x}^{(k)})$, we use:

$$\Delta \boldsymbol{x} = -\eta \nabla f(\boldsymbol{x}^{(k)}), \text{ i.e., } \boldsymbol{x}^{(k+1)} := \boldsymbol{x}^{(k)} - \eta \nabla f(\boldsymbol{x}^{(k)}), \tag{10}$$

  where $\eta > 0$ is the **step size**, also called the **learning rate** in data science literature.
- If the optimization problem is maximization rather than minimization, the step should be $\Delta \boldsymbol{x} = \eta \nabla f(\boldsymbol{x}^{(k)})$ rather than Eq. (10). In that case, the name of method is **gradient ascent**.
- Recall Eq. (6):

$$f(\boldsymbol{x}^{(k)} + \Delta \boldsymbol{x}) - f(\boldsymbol{x}^{(k)}) \leq \nabla f(\boldsymbol{x}^{(k)})^\top \Delta \boldsymbol{x} + \frac{L}{2}\|\Delta \boldsymbol{x}\|_2^2.$$

  Using Eq. (10) in this equation gives:

$$\begin{aligned} f(\boldsymbol{x}^{(k)} + \Delta \boldsymbol{x}) - f(\boldsymbol{x}^{(k)}) &\leq -\eta \|\nabla f(\boldsymbol{x}^{(k)})\|_2^2 + \frac{L}{2}\eta^2 \|\nabla f(\boldsymbol{x}^{(k)})\|_2^2 \\ &= \eta(\frac{L}{2}\eta - 1)\|\nabla f(\boldsymbol{x}^{(k)})\|_2^2 \end{aligned} \tag{11}$$

Gradient descent: step size

- We found:

$$f(\boldsymbol{x}^{(k)} + \Delta\boldsymbol{x}) - f(\boldsymbol{x}^{(k)}) \leq \eta(\frac{L}{2}\eta - 1)\|\nabla f(\boldsymbol{x}^{(k)})\|_2^2$$

- If $\boldsymbol{x}^{(k)}$ is not a stationary point, we have $\|\nabla f(\boldsymbol{x}^{(k)})\|_2^2 > 0$.
- Recall Eq. (7):

$$f(\boldsymbol{x}^{(k)} + \Delta\boldsymbol{x}) - f(\boldsymbol{x}^{(k)}) < 0.$$

- Noticing $\eta > 0$, for satisfying this equation, we must set:

$$\frac{L}{2}\eta - 1 < 0 \implies \eta < \frac{2}{L}. \tag{12}$$

Gradient descent: step size

- Recall Eq. (11):

$$f(\boldsymbol{x}^{(k)} + \Delta\boldsymbol{x}) - f(\boldsymbol{x}^{(k)}) \leq -\eta\|\nabla f(\boldsymbol{x}^{(k)})\|_2^2 + \frac{L}{2}\eta^2\|\nabla f(\boldsymbol{x}^{(k)})\|_2^2$$

- On the other hand, we can minimize this equation by setting its derivative w.r.t. η to zero:

$$\begin{aligned}\frac{\partial}{\partial\eta}(-\eta\|\nabla f(\boldsymbol{x}^{(k)})\|_2^2 + \frac{L}{2}\eta^2\|\nabla f(\boldsymbol{x}^{(k)})\|_2^2) &= -\|\nabla f(\boldsymbol{x}^{(k)})\|_2^2 + L\eta\|\nabla f(\boldsymbol{x}^{(k)})\|_2^2 \\ &= (-1 + L\eta)\|\nabla f(\boldsymbol{x}^{(k)})\|_2^2 \stackrel{\text{set}}{=} 0 \implies \eta = \frac{1}{L}.\end{aligned}$$

If we set:

$$\eta < \frac{1}{L}, \tag{13}$$

then Eq. (11) becomes:

$$\begin{aligned}&f(\boldsymbol{x}^{(k)} + \Delta\boldsymbol{x}) - f(\boldsymbol{x}^{(k)}) \leq -\frac{1}{L}\|\nabla f(\boldsymbol{x}^{(k)})\|_2^2 + \frac{1}{2L}\|\nabla f(\boldsymbol{x}^{(k)})\|_2^2 = -\frac{1}{2L}\|\nabla f(\boldsymbol{x}^{(k)})\|_2^2 < 0 \\ &\implies f(\boldsymbol{x}^{(k+1)}) \leq f(\boldsymbol{x}^{(k)}) - \frac{1}{2L}\|\nabla f(\boldsymbol{x}^{(k)})\|_2^2.\end{aligned} \tag{14}$$

Gradient descent: step size

- Recall Eqs. (13) and (14):

$$\eta < \frac{1}{L},$$
$$f(\boldsymbol{x}^{(k+1)}) \leq f(\boldsymbol{x}^{(k)}) - \frac{1}{2L}\|\nabla f(\boldsymbol{x}^{(k)})\|_2^2.$$

- Eq. (13) means that there should be an upper-bound, dependent on the Lipschitz constant, on the step size. Hence, L is still required.
- Eq. (14) shows that every iteration of gradient descent decreases the cost function:

$$f(\boldsymbol{x}^{(k+1)}) \leq f(\boldsymbol{x}^{(k)}), \tag{15}$$

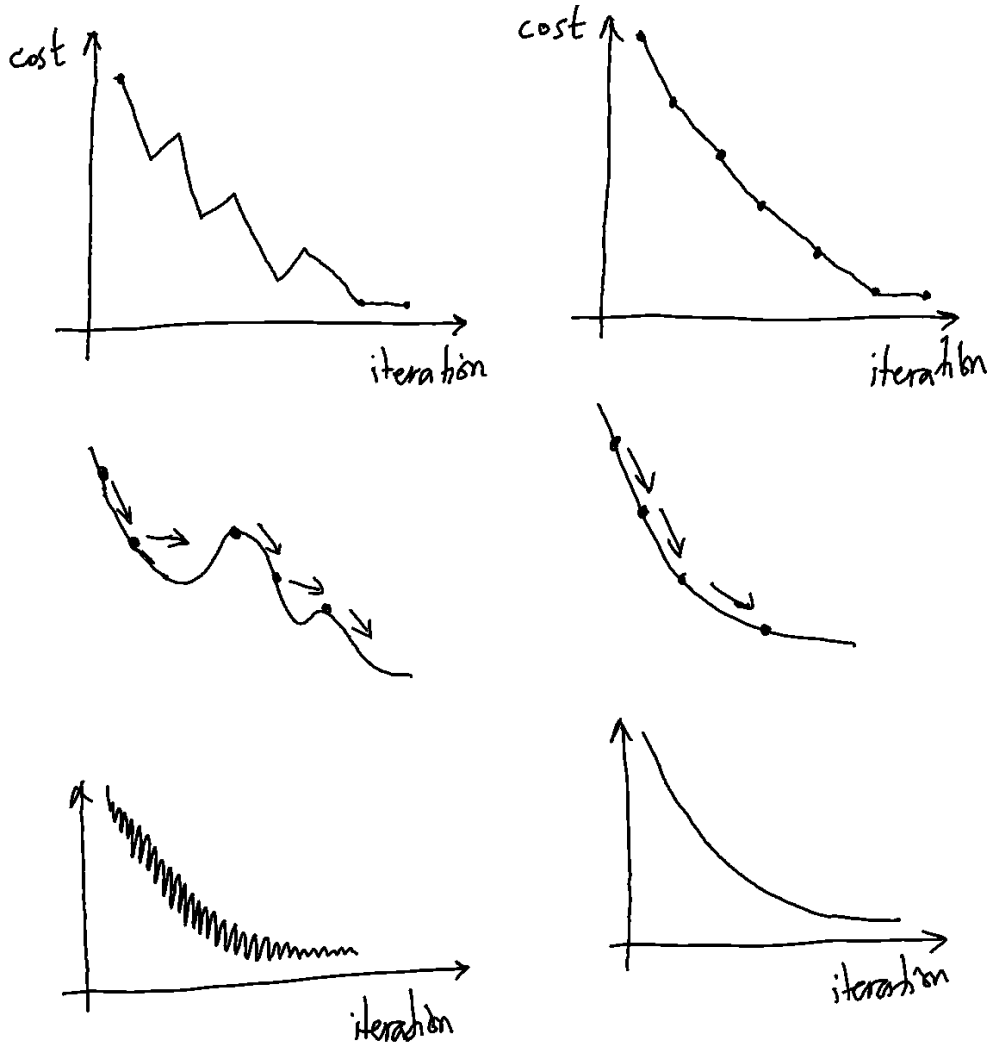
  and the amount of this decrease depends on the norm of gradient at that iteration.
- So, the series of solutions converges to the optimal solution while the function value decreases iteratively until the local minimum:

$$\{\boldsymbol{x}^{(0)}, \boldsymbol{x}^{(1)}, \boldsymbol{x}^{(2)}, \dots\} \to \boldsymbol{x}^*,$$
$$f(\boldsymbol{x}^{(0)}) \geq f(\boldsymbol{x}^{(1)}) \geq f(\boldsymbol{x}^{(2)}) \geq \cdots \geq f(\boldsymbol{x}^*).$$

- If the optimization problem is a convex problem, the solution is the global solution; otherwise, the solution is local.

Gradient descent: cost versus iterations

$$\{\boldsymbol{x}^{(0)}, \boldsymbol{x}^{(1)}, \boldsymbol{x}^{(2)}, \dots\} \to \boldsymbol{x}^*,$$
$$f(\boldsymbol{x}^{(0)}) \geq f(\boldsymbol{x}^{(1)}) \geq f(\boldsymbol{x}^{(2)}) \geq \cdots \geq f(\boldsymbol{x}^*).$$



Line-Search

Line-search

- We saw the step size of gradient descent requires knowledge of the Lipschitz constant for the smoothness of gradient. However, we may not know the exact Lipschitz constant. Hence, we can find the suitable step size η by a search which is named the **line-search**.
- In line-search of every optimization iteration, we start with $\eta = 1$ and if it does not satisfy Eq. (7) with step $\Delta \boldsymbol{x} = -\eta \nabla f(\boldsymbol{x}^{(k)})$:

$$f(\boldsymbol{x}^{(k)} + \Delta \boldsymbol{x}) < f(\boldsymbol{x}^{(k)}) \implies f(\boldsymbol{x}^{(k)} - \eta \nabla f(\boldsymbol{x}^{(k)})) < f(\boldsymbol{x}^{(k)}), \tag{16}$$

  we halve it, $\eta \leftarrow \eta/2$.
- This halving step size is repeated until this equation is satisfied, i.e., until we have a decrease in the objective function. Note that this decrease will happen when the step size becomes small enough to satisfy Eq. (13):

$$\eta < \frac{1}{L}.$$

The algorithm of gradient descent with line-search is shown in Algorithm **??**. As this algorithm shows, line-search has its own internal iterations inside every iteration of gradient descent.

Gradient descent with line-search

The algorithm of gradient descent with line-search:

```
1  Initialize x^(0)
2  for iteration k = 0, 1, ... do
3      Initialize η := 1
4      for iteration τ = 1, 2, ... do
5          Check line-search condition
6          if not satisfied then
7              η ← (1/2) × η
8          else
9              x^(k+1) := x^(k) − η∇f(x^(k))
10             break the loop
11     Check the convergence criterion
12     if converged then
13         return x^(k+1)
```

As this algorithm shows, line-search has its own internal iterations inside every iteration of gradient descent.

Backtracking line-search

- A more sophisticated line-search method is the **Armijo line-search** [4], also called the **backtracking line-search**. Rather than Eq. (16), it checks if the cost function is *sufficiently* decreased:

$$f(\boldsymbol{x}^{(k)} + \boldsymbol{p}) \leq f(\boldsymbol{x}^{(k)}) + c\, \boldsymbol{p}^\top f(\boldsymbol{x}^{(k)}), \tag{17}$$

  where $c \in (0.0.5]$ is the parameter of Armijo line-search and $\boldsymbol{p} = \Delta \boldsymbol{x}$ is the search direction for update.
- The value of c should be small, e.g., $c = 10^{-4}$ [5].
- This condition is called the **Armijo condition** or the **Armijo-Goldstein condition**.
- In gradient descent, the search direction is $\boldsymbol{p} = \Delta \boldsymbol{x} = -\eta \nabla f(\boldsymbol{x}^{(k)})$ according to Eq. (10). Hence, for gradient descent, it checks:

$$f(\boldsymbol{x}^{(k)} - \eta \nabla f(\boldsymbol{x}^{(k)})) \leq f(\boldsymbol{x}^{(k)}) - \eta\, c \|\nabla f(\boldsymbol{x}^{(k)})\|_2^2. \tag{18}$$

- Another more sophisticated line-search is **Wolfe conditions** [6]. We will learn it later in the course.

Convergence criterion

Convergence criteria

- For all numerical optimization methods including gradient descent, there exist several methods for convergence criterion to stop updating the solution and terminate optimization.
- Some of them are:
  - Small norm of gradient:

$$\|\nabla f(\boldsymbol{x}^{(k+1)})\|_2 \leq \epsilon,$$

    where ϵ is a small positive number.
    - ★ The reason for this criterion is the first-order optimality condition (recall that at the local optimum, we have $\|\nabla f(\boldsymbol{x}^*)\|_2 = 0$).
    - ★ If the function is not convex, this criterion has the risk of stopping at a saddle point.
  - Small change of cost function:

$$|f(\boldsymbol{x}^{(k+1)}) - f(\boldsymbol{x}^{(k)})| \leq \epsilon.$$

  - Small change of gradient of function:

$$|\nabla f(\boldsymbol{x}^{(k+1)}) - \nabla f(\boldsymbol{x}^{(k)})| \leq \epsilon.$$

  - Reaching maximum desired number of iterations, denoted by

$$k < \max_k .$$

Momentum

Gradient descent with momentum

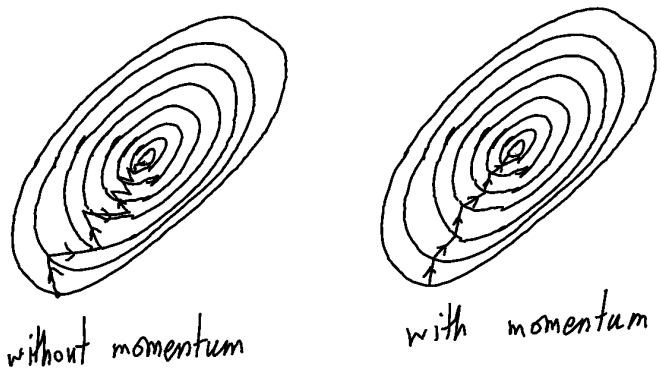
- Gradient descent and other first-order methods can have a momentum term. **Momentum**, proposed in [7], makes the change of solution $\Delta \boldsymbol{x}$ a little similar to the previous change of solution.
- Hence, the change adds a history of previous change to Eq. (10):

$$(\Delta \boldsymbol{x})^{(k)} := \alpha (\Delta \boldsymbol{x})^{(k-1)} - \eta^{(k)} \nabla f(\boldsymbol{x}^{(k)}), \tag{19}$$

  where $\alpha > 0$ is the momentum parameter which weights the importance of history compared to the descent direction.
- We use this $(\Delta \boldsymbol{x})^{(k)}$ in Eq. (5) for updating the solution:

$$\boldsymbol{x}^{(k+1)} := \boldsymbol{x}^{(k)} + (\Delta \boldsymbol{x})^{(k)}.$$

- Because of faithfulness to the track of previous updates, momentum reduces the amount of oscillation of updates in gradient descent optimization.



Steepest Descent

Steepest Descent

- **Steepest descent** is similar to gradient descent but there is a difference between them.
- In steepest descent, we move toward the negative gradient as much as possible to reach the smallest function value which can be achieved at every iteration.
- Hence, the step size at iteration k of steepest descent is calculated as [8]:

$$\eta^{(k)} := \arg\min_{\eta} f\left(\boldsymbol{x}^{(k)} - \eta \nabla f(\boldsymbol{x}^{(k)})\right), \tag{20}$$

and then, the solution is updated using Eq. (10) as in gradient descent:

$$\boldsymbol{x}^{(k+1)} := \boldsymbol{x}^{(k)} - \eta \nabla f(\boldsymbol{x}^{(k)}).$$

- Another interpretation of steepest descent [9, Chapter 9.4]: The first-order Taylor expansion of function is $f(\boldsymbol{x} + \boldsymbol{v}) \approx f(\boldsymbol{x}) + \nabla f(\boldsymbol{x})^\top \boldsymbol{v}$. Hence, the step size in the normalized steepest descent, at iteration k, is obtained as:

$$\Delta \boldsymbol{x} = \arg\min_{\boldsymbol{v}} \{\nabla f(\boldsymbol{x}^{(k)})^\top \boldsymbol{v} \mid \|\boldsymbol{v}\|_2 \leq 1\}, \tag{21}$$

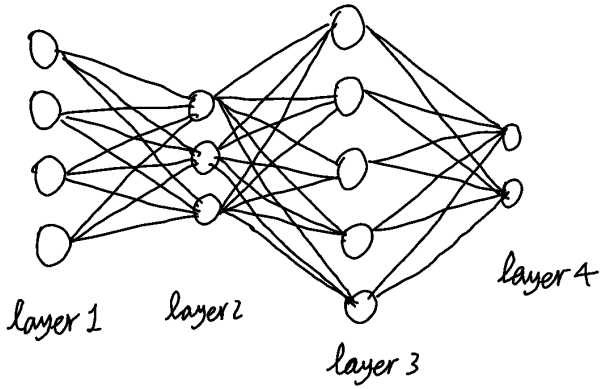
which is used in Eq. (5) for updating the solution:

$$\boldsymbol{x}^{(k+1)} := \boldsymbol{x}^{(k)} + \Delta \boldsymbol{x}.$$

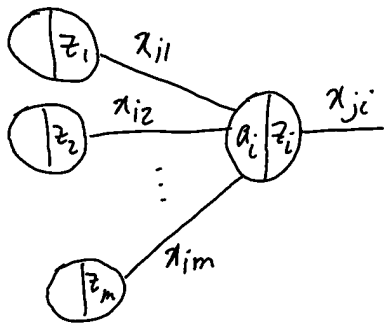
Backpropagation

Neural network

- Neural network:



- Every neuron in neural network:

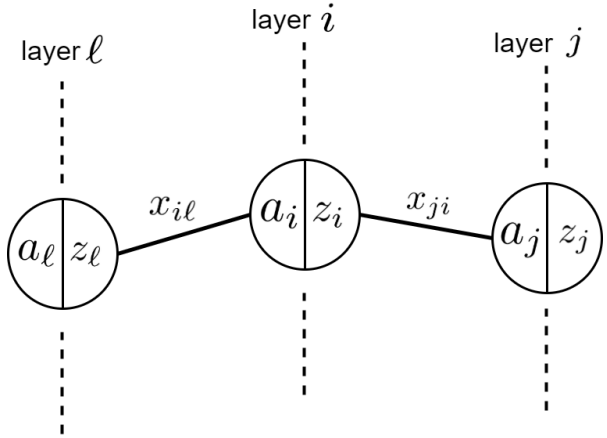


- Let x_{ji} denote the weight connecting neuron i to neuron j. Let a_i and z_i be the output of neuron i before and after applying its activation function $\sigma_i(.) : \mathbb{R} \to \mathbb{R}$, respectively.

$$a_i = \sum_{\ell=1}^{m} x_{i\ell} z_\ell, \qquad z_i := \sigma_i(a_i).$$

Backpropagation

- Consider three neurons in three layers of a network:



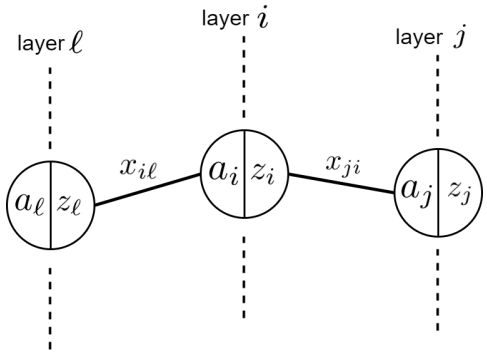
- We have $a_i = \sum_\ell x_{i\ell} z_\ell$ which sums over the neurons in layer ℓ. By chain rule, the gradient of error e w.r.t. to the weight between neurons ℓ and i is:

$$\frac{\partial e}{\partial x_{i\ell}} = \frac{\partial e}{\partial a_i} \times \frac{\partial a_i}{\partial x_{i\ell}} \overset{(a)}{=} \delta_i \times z_\ell, \tag{22}$$

where (a) is because $a_i = \sum_\ell x_{i\ell} z_\ell$ and we define:

$$\delta_i := \frac{\partial e}{\partial a_i}.$$

Backpropagation



- If layer i is the last layer, δ_i can be computed by derivative of error (loss function) w.r.t. the output.
- However, if i is one of the hidden layers, δ_i is computed by chain rule as:

$$\delta_i = \frac{\partial e}{\partial a_i} = \sum_j \left(\frac{\partial e}{\partial a_j} \times \frac{\partial a_j}{\partial a_i} \right) = \sum_j \left(\delta_j \times \frac{\partial a_j}{\partial a_i} \right). \tag{23}$$

- The term $\partial a_j / \partial a_i$ is calculated by chain rule as:

$$\frac{\partial a_j}{\partial a_i} = \frac{\partial a_j}{\partial z_i} \times \frac{\partial z_i}{\partial a_i} \overset{(a)}{=} x_{ji}\, \sigma'(a_i), \tag{24}$$

where (a) is because $a_j = \sum_i x_{ji} z_i$ and $z_i = \sigma(a_i)$ and $\sigma'(.)$ denotes the derivative of activation function. Putting Eq. (24) in Eq. (23) gives:

$$\delta_i = \sigma'(a_i) \sum_j (\delta_j\, x_{ji}).$$

Backpropagation

- We found:

$$\delta_i = \sigma'(a_i) \sum_j (\delta_j \, x_{ji}).$$

- Putting this equation in Eq. (22), $\frac{\partial e}{\partial x_{i\ell}} = \delta_i \times z_\ell$, gives:

$$\frac{\partial e}{\partial x_{i\ell}} = z_\ell \, \sigma'(a_i) \sum_j (\delta_j \, x_{ji}). \tag{25}$$

- **Backpropagation** uses the gradient in Eq. (25) for updating the weight $x_{i\ell}, \forall i, \ell$ by gradient descent:

$$x_{i\ell}^{(k+1)} := x_{i\ell}^{(k)} - \eta^{(k)} \frac{\partial e}{\partial x_{i\ell}}.$$

- This tunes the weights from last layer to the first layer for every iteration of optimization.
- Therefore, **backpropagation**, proposed in 1986 [7], is actually gradient descent with chain rule in derivatives because of having layers of parameters. It is the most well-known optimization method used for training neural networks.

Accelerated gradient method

Accelerated Gradient method

- It was shown in the literature that gradient descent is not optimal in convergence rate and can be improved.
- It was at that time that Nesterov proposed **Accelerated Gradient Method (AGM)** [10], in 1983, to make the convergence rate of gradient descent optimal [11, Chapter 2.2].
- AGM is also called the **Nesterov's accelerated gradient method** or **Fast Gradient Method (FGM)**. A series of Nesterov's papers improved AGM [10, 12, 13, 14].
- Consider a sequence $\{\gamma^{(k)}\}$ which satisfies:

$$\prod_{i=0}^{k}(1-\gamma^{(i)}) \geq (\gamma^{(k)})^2, \quad \forall k \geq 0, \quad \gamma^{(k)} \in [0,1]. \tag{26}$$

- An example sequence, satisfying this condition, is

$$\gamma^{(0)} = \gamma^{(1)} = \gamma^{(2)} = \gamma^{(3)} = 0, \gamma^{(k)} = 2/k, \forall k \geq 4.$$

- The AGM updates the solution iteratively as [10]:

$$\boldsymbol{x}^{(k+1)} := \boldsymbol{y}^{(k)} - \eta^{(k)} \nabla f(\boldsymbol{y}^{(k)}), \tag{27}$$

$$\boldsymbol{y}^{(k+1)} := (1-\gamma^{(k)})\boldsymbol{x}^{(k+1)} + \gamma^{(k)}\boldsymbol{x}^{(k)}, \tag{28}$$

until convergence.

Comparison of convergences rates

- Consider a convex and differentiable function $f(.)$, with domain $\mathcal{D}$, whose gradient is L-smooth (see Eq. (2)). Let f^* be the minimum of cost function and $\boldsymbol{x}^*$ be the minimizer. Starting from the initial point $\boldsymbol{x}^{(0)}$, after t iterations of the optimization algorithm, we will have the following.
- The convergence rate of gradient descent:

$$f(\boldsymbol{x}^{(t+1)}) - f^* \leq \frac{2L\|\boldsymbol{x}^{(0)} - \boldsymbol{x}^*\|_2^2}{t+1} = \mathcal{O}(\frac{1}{t}). \tag{29}$$

- The convergence rate of accelerated gradient method:

$$f(\boldsymbol{x}^{(t+1)}) - f^* \leq \frac{2L\|\boldsymbol{x}^{(0)} - \boldsymbol{x}^*\|_2^2}{(t+1)^2} = \mathcal{O}(\frac{1}{t^2}). \tag{30}$$

Stochastic gradient methods

Stochastic gradient descent

- Assume we have a dataset of n data points, $\{\boldsymbol{a}_i \in \mathbb{R}^d\}_{i=1}^n$ and their labels $\{l_i \in \mathbb{R}\}_{i=1}^n$.
- Let the cost function $f(.)$ be decomposed into summation of n terms $\{f_i(\boldsymbol{x})\}_{i=1}^n$. Some well-known examples for the cost function terms are:
  - Least squares error: $f_i(\boldsymbol{x}) = 0.5(\boldsymbol{a}_i^\top \boldsymbol{x} - l_i)^2$,
  - Absolute error: $f_i(\boldsymbol{x}) = \boldsymbol{a}_i^\top \boldsymbol{x} - l_i$,
  - Hinge loss (for $l_i \in \{-1, 1\}$): $f_i(\boldsymbol{x}) = \max(0, 1 - l_i \boldsymbol{a}_i^\top \boldsymbol{x})$.
  - Logistic loss (for $l_i \in \{-1, 1\}$): $\log(\frac{1}{1+\exp(-l_i \boldsymbol{a}_i^\top \boldsymbol{x})})$.
- The optimization problem becomes:

$$\underset{\boldsymbol{x}}{\text{minimize}} \quad \frac{1}{n} \sum_{i=1}^{n} f_i(\boldsymbol{x}). \tag{31}$$

In this case, the full gradient is the average gradient, i.e:

$$\nabla f(\boldsymbol{x}) = \frac{1}{n} \sum_{i=1}^{n} \nabla f_i(\boldsymbol{x}), \tag{32}$$

so Eq. (9), $\Delta \boldsymbol{x} = -(1/L)\nabla f(\boldsymbol{x}^{(k)})$, becomes $\Delta \boldsymbol{x} = -(1/(Ln)) \sum_{i=1}^{n} \nabla f_i(\boldsymbol{x}^{(k)})$. This is what gradient descent uses for updating the solution at every iteration.

Stochastic gradient descent

$$\nabla f(\boldsymbol{x}) = \frac{1}{n}\sum_{i=1}^{n} \nabla f_i(\boldsymbol{x}),$$

- Calculation of this full gradient is time-consuming and inefficient for large values of n, especially as it needs to be recalculated at every iteration.
- **Stochastic Gradient Descent (SGD)**, also called **stochastic gradient method**, approximates gradient descent stochastically and samples (i.e. bootstraps) one of the points at every iteration for updating the solution. Hence, it uses:

$$\boldsymbol{x}^{(k+1)} := \boldsymbol{x}^{(k)} - \eta^{(k)} \nabla f_i(\boldsymbol{x}^{(k)}), \tag{33}$$

  rather than Eq. (10), $\boldsymbol{x}^{(k+1)} := \boldsymbol{x}^{(k)} - \eta \nabla f(\boldsymbol{x}^{(k)})$.
- The idea of stochastic approximation was first proposed in 1951 [15]. It was first used for machine learning in 1998 [16].
- As Eq. (33) states, SGD often uses an adaptive step size which changes in every iteration. The step size can be decreasing because in initial iterations, where we are far away from the optimal solution, the step size can be large; however, it should be small in the last iterations which is supposed to be close to the optimal solution. Some well-known adaptations for the step size are:

$$\eta^{(k)} := \frac{1}{k}, \quad \eta^{(k)} := \frac{1}{\sqrt{k}}, \quad \eta^{(k)} := \eta. \tag{34}$$

Stochastic gradient descent

Theorem (Convergence rates for SGD)

Consider a function $f(\boldsymbol{x}) = \sum_{i=1}^{n} f_i(\boldsymbol{x})$ *and which is bounded below and each* f_i *is differentiable. Let the domain of function* $f(.)$ *be* $\mathcal{D}$ *and its gradient be L-smooth (see Eq. (2)). Assume* $\mathbb{E}[\|\nabla f_i(\boldsymbol{x}_k)\|_2^2 \,|\, \boldsymbol{x}_k] \leq \beta^2$ *where* β *is a constant. Depending on the step size, the convergence rate of SGD is:*

$$\mathcal{O}(\frac{1}{\log t}) \quad \text{if} \quad \eta^{(k)} = \frac{1}{k}, \tag{35}$$

$$\mathcal{O}(\frac{\log t}{\sqrt{t}}) \quad \text{if} \quad \eta^{(k)} = \frac{1}{\sqrt{k}}, \tag{36}$$

$$\mathcal{O}(\frac{1}{t} + \eta) \quad \text{if} \quad \eta^{(k)} = \eta, \tag{37}$$

where t *denotes the iteration index. If the functions* f_i*'s are* μ*-strongly convex, then the convergence rate of SGD is:*

$$\mathcal{O}(\frac{1}{t}) \quad \text{if} \quad \eta^{(k)} = \frac{1}{\mu k}, \tag{38}$$

$$\mathcal{O}((1 - \frac{\mu}{L})^t + \eta) \quad \text{if} \quad \eta^{(k)} = \eta. \tag{39}$$

Stochastic gradient descent

- Recall Eqs. (37) and (39):

$$\text{convex:} \quad \mathcal{O}(\frac{1}{t} + \eta) \quad \text{if} \quad \eta^{(k)} = \eta,$$

$$\text{strongly convex:} \quad \mathcal{O}\big((1 - \frac{\mu}{L})^t + \eta\big) \quad \text{if} \quad \eta^{(k)} = \eta.$$

- These equations show that with a fixed step size η, SGD converges sublinearly for a non-convex function and linearly for a strongly convex function in the initial iterations.
- However, in the late iterations, it stagnates to a neighborhood around the optimal point and never reaches it. Hence, SGD has less accuracy than gradient descent (whose convergence rate is $\mathcal{O}(\frac{1}{t^2})$).
- The advantage of SGD over gradient descent is that its every iteration is much faster than every iteration of gradient descent because of less computations for gradient. This faster pacing of every iteration shows off more when n is huge.
- In summary, SGD has fast convergence to low accurate optimal point.
- It is noteworthy that the full gradient is not available in SGD to use for checking convergence, as discussed before. One can use other criteria or merely check the norm of gradient for the sampled point.
- SGD can be used with the line-search methods, too. SGD can also use a momentum term.

Mini-batch stochastic gradient descent

- Gradient descent uses the entire n data points and SGD uses one randomly sampled point at every iteration. For large datasets, gradient descent is very slow and intractable in every iteration while SGD will need a significant number of iterations to roughly cover all data. Besides, SGD has low accuracy in convergence to the optimal point.
- We can have a middle case where we use a batch of b randomly sampled points at every iteration. This method is named the **mini-batch SGD** or the **hybrid deterministic-stochastic gradient** method. This batch-wise approach is wise for large datasets.
- Usually, before start of optimization, the n data points are randomly divided into $\lfloor n/b \rfloor$ batches of size b. This is equivalent to simple random sampling for sampling points into batches without replacement. We denote the dataset by $\mathcal{D}$ (where $|\mathcal{D}| = n$) and the i-th batch by $\mathcal{B}_i$ (where $|\mathcal{B}_i| = b$). The batches are disjoint:

$$\bigcup_{i=1}^{\lfloor n/b \rfloor} \mathcal{B}_i = \mathcal{D}, \tag{40}$$

$$\mathcal{B}_i \cap \mathcal{B}_j = \varnothing, \quad \forall i, j \in \{1, \dots, \lfloor n/b \rfloor\},\ i \neq j. \tag{41}$$

- Another less-used approach for making batches is to sample points for a batch during optimization. This is equivalent to bootstrapping for sampling points into batches with replacement. In this case, the batches are not disjoint anymore and Eqs. (40) and (41) do not hold.

Mini-batch stochastic gradient descent

Definition (Epoch)

In mini-batch SGD, when all $\lfloor n/b \rfloor$ batches of data are used for optimization once, an **epoch** is completed. After completion of an epoch, the next epoch is started and epochs are repeated until convergence of optimization.

- In mini-batch SGD, if the k-th iteration of optimization is using the k'-th batch, the update of solution is done as:

$$\boldsymbol{x}^{(k+1)} := \boldsymbol{x}^{(k)} - \eta^{(k)} \frac{1}{b} \sum_{i \in \mathcal{B}_{k'}} \nabla f_i(\boldsymbol{x}^{(k)}). \tag{42}$$

- The scale factor $1/b$ is sometimes dropped for simplicity.
- Mini-batch SGD is used significantly in machine learning, especially in neural networks [16, 17].
- Because of dividing data into batches, mini-batch SGD can be solved on parallel servers as a distributed optimization method.

Mini-batch stochastic gradient descent

Theorem (Convergence rates for mini-batch SGD)

Consider a function $f(\boldsymbol{x}) = \sum_{i=1}^{n} f_i(\boldsymbol{x})$ *which is bounded below and each* f_i *is differentiable. Let the domain of function* $f(.)$ *be* $\mathcal{D}$ *and its gradient be L-smooth (see Eq. (2)) and assume* $\eta^{(k)} = \eta$ *is fixed. The batch-wise gradient is an approximation to the full gradient with some error* e_t *for the t-th iteration:*

$$\frac{1}{b} \sum_{i \in \mathcal{B}_{t'}} \nabla f_i(\boldsymbol{x}^{(t)}) = \nabla f(\boldsymbol{x}^{(t)}) + e_t. \tag{43}$$

The convergence rate of mini-batch SGD for non-convex and convex functions are:

$$\mathcal{O}(\frac{1}{t} + \|e_t\|_2^2), \tag{44}$$

where t denotes the iteration index. If the functions f_i*'s are* μ*-strongly convex, then the convergence rate of mini-batch SGD is:*

$$\mathcal{O}((1 - \frac{\mu}{L})^t + \|e_t\|_2^2). \tag{45}$$

Therefore, the convergence rate of mini-batch gets closer to that of gradient descent, $\mathcal{O}(1/t)$, if the batch size increases.

Mini-batch stochastic gradient descent

- If we sample the batches without replacement (i.e., sampling batches by simple random sampling before start of optimization) or with replacement (i.e., bootstrapping during optimization), the expected error is [18, Proposition 3]:

$$\mathbb{E}[\|e_t\|_2^2] = (1 - \frac{b}{n})\frac{\sigma^2}{b}, \tag{46}$$

$$\mathbb{E}[\|e_t\|_2^2] = \frac{\sigma^2}{b}, \tag{47}$$

  respectively, where σ^2 is the variance of whole dataset.
- According to Eqs. (46) and (47), the accuracy of SGD by sampling without and with replacement increases by $b \to n$ and $b \to \infty$, respectively.
- However, this increase makes every iteration slower so there is trade-off between accuracy and speed.

Stochastic Average Gradient Methods

Stochastic average gradient

- SGD is faster than gradient descent but its problem is its lower accuracy compared to gradient descent. **Stochastic Average Gradient (SAG)**, proposed in 2012 [19], keeps a trade-off between accuracy and speed.
- Let $\nabla f_i(\boldsymbol{x}^{(k)})$ be the gradient of $f_i(.)$, evaluated in point $\boldsymbol{x}^{(k)}$, at iteration k. According to Eqs. (10) and (32), gradient descent updates the solution as:

$$\boldsymbol{x}^{(k+1)} := \boldsymbol{x}^{(k)} - \frac{\eta^{(k)}}{n} \sum_{i=1}^{n} \nabla f_i(\boldsymbol{x}^{(k)}).$$

SAG randomly samples one of the points and updates its gradient among the gradient terms. If the sampled point is the j-th one, we have:

$$\boldsymbol{x}^{(k+1)} := \boldsymbol{x}^{(k)} - \frac{\eta^{(k)}}{n} \Big(\nabla f_j(\boldsymbol{x}^{(k)}) - \nabla f_j(\boldsymbol{x}^{(k-1)}) + \sum_{i=1}^{n} \nabla f_i(\boldsymbol{x}^{(k-1)}) \Big). \tag{48}$$

- In other words, we subtract the j-th gradient from the summation of all n gradients in previous iteration $(k-1)$ by $\sum_{i=1}^{n} \nabla f_i(\boldsymbol{x}^{(k-1)}) - \nabla f_j(\boldsymbol{x}^{(k-1)})$; then, we add back the new j-th gradient in this iteration by adding $\nabla f_j(\boldsymbol{x}^{(k)})$.

Stochastic average gradient

Theorem (Convergence rates for SAG [19, Proposition 1])

Consider a function $f(\boldsymbol{x}) = \sum_{i=1}^{n} f_i(\boldsymbol{x})$ *which is bounded below and each* f_i *is differentiable. Let the domain of function* $f(.)$ *be* $\mathcal{D}$ *and its gradient be L-smooth (see Eq. (2)). The convergence rate of SAG is:*

$$\mathcal{O}(\frac{1}{t}),$$

where t *denotes the iteration index.*

- SAG has the same rate order as gradient descent; although, it usually needs some more iterations to converge.
- Practical experiments have shown that SAG requires many parameter fine-tuning to perform perfectly.

Stochastic variance reduced gradient

- Another effective first-order method is the **Stochastic Variance Reduced Gradient (SVRG)**, proposed in 2013 [20].

1 Initialize $\widetilde{\boldsymbol{x}}^{(0)}$
2 **for** *iteration* $k = 1, 2, \ldots$ **do**
3 $\quad \widetilde{\boldsymbol{x}} := \widetilde{\boldsymbol{x}}^{(k-1)}$
4 $\quad \nabla f(\widetilde{\boldsymbol{x}}) \overset{(112)}{:=} \frac{1}{n} \sum_{i=1}^{n} \nabla f_i(\widetilde{\boldsymbol{x}})$
5 $\quad \boldsymbol{x}^{(0)} := \widetilde{\boldsymbol{x}}$
6 $\quad$ **for** *iteration* $\tau = 0, 1, \ldots, m-1$ **do**
7 $\quad\quad$ Randomly sample j from $\{1, \ldots, n\}$.
8 $\quad\quad \boldsymbol{x}^{(\tau+1)} := \boldsymbol{x}^{(\tau)} - \eta^{(\tau)} \big(\nabla f_j(\boldsymbol{x}^{(\tau)}) - \nabla f_j(\widetilde{\boldsymbol{x}}) + \nabla f(\widetilde{\boldsymbol{x}}) \big)$.
9 $\quad \widetilde{\boldsymbol{x}}^{(k)} := \boldsymbol{x}^{(m)}$

- The update of solution is similar to SAG but for every iteration, it updates the solution for m times.
- SVRG is an efficient method and its convergence rate is similar to that of SAG.
- Both SAG and SVRG reduce the variance of solution to optimization [20].

Adaptive Learning Rate

Adaptive Gradient (AdaGrad)

- We can adapt the learning rate in stochastic gradient methods. Three most well-known methods for adapting the learning rate are AdaGrad, RMSProp, and Adam.
- **Adaptive Gradient (AdaGrad)** method, proposed in 2011 [21], updates the solution iteratively as:

$$\boldsymbol{x}^{(k+1)} := \boldsymbol{x}^{(k)} - \eta^{(k)} \boldsymbol{G}^{-1} \nabla f_i(\boldsymbol{x}^{(k)}), \tag{49}$$

where $\boldsymbol{G}$ is a $(d \times d)$ diagonal matrix whose (j,j)-th element is:

$$\boldsymbol{G}(j,j) := \sqrt{\varepsilon + \sum_{\tau=0}^{k} \left(\nabla_j f_{i_\tau}(\boldsymbol{x}^{(\tau)})\right)^2}, \tag{50}$$

where $\varepsilon \geq 0$ is for stability (making $\boldsymbol{G}$ full rank), i_τ is the randomly sampled point (from $\{1, \dots, n\}$) at iteration τ, and $\nabla_j f_{i_\tau}(.)$ is the partial derivative of $f_{i_\tau}(.)$ w.r.t. its j-th element (note that $f_{i_\tau}(.)$ is d-dimensional).

- Putting Eq. (50) in Eq. (49) can simplify AdaGrad to:

$$\boldsymbol{x}_j^{(k+1)} := \boldsymbol{x}_j^{(k)} - \frac{\eta^{(k)}}{\sqrt{\varepsilon + \sum_{\tau=0}^{k} \left(\nabla_j f_{i_\tau}(\boldsymbol{x}^{(\tau)})\right)^2}} \nabla f_j(\boldsymbol{x}_j^{(k)}). \tag{51}$$

- AdaGrad keeps a history of the sampled points and it takes derivative for them to use. During the iterations so far, if a dimension has changed significantly, it dampens the learning rate for that dimension (see the inverse in Eq. (49)); hence, it gives more weight for changing the dimensions which have not changed noticeably.

Root Mean Square Propagation (RMSProp)

- **Root Mean Square Propagation (RMSProp)** was first proposed in 2012 [22] which is unpublished.
- It is an improved version of **Rprop (resilient backpropagation)**, proposed in 1992 [23], which uses the sign of gradient in optimization.
- Inspired by momentum in Eq. (19):

$$(\Delta \boldsymbol{x})^{(k)} := \alpha(\Delta \boldsymbol{x})^{(k-1)} - \eta^{(k)} \nabla f(\boldsymbol{x}^{(k)}),$$

it updates a scalar variable v as [24]:

$$v^{(k+1)} := \gamma v^{(k)} + (1-\gamma)\|\nabla f_i(\boldsymbol{x}^{(k)})\|_2^2, \tag{52}$$

where $\gamma \in [0, 1]$ is the forgetting factor (e.g., $\gamma = 0.9$). Then, it uses this v to weight the learning rate:

$$\boldsymbol{x}^{(k+1)} := \boldsymbol{x}^{(k)} - \frac{\eta^{(k)}}{\sqrt{\varepsilon + v^{(k+1)}}} \nabla f_j(\boldsymbol{x}_j^{(k)}), \tag{53}$$

where $\epsilon \geq 0$ is for stability not to have division by zero.

- Comparing Eqs. (51) and (53) shows that RMSProp has a similar form to AdaGrad.

Adaptive Moment Estimation (Adam)

- **Adam optimizer** [25] improves over RMSProp by adding a momentum term.
- It updates the scalar v and the vector $\boldsymbol{m} \in \mathbb{R}^d$ as:

$$\boldsymbol{m}^{(k+1)} := \gamma_1 \boldsymbol{m}^{(k)} + (1-\gamma_1)\nabla f_i(\boldsymbol{x}^{(k)}), \tag{54}$$

$$v^{(k+1)} := \gamma_2 v^{(k)} + (1-\gamma_2)\|\nabla f_i(\boldsymbol{x}^{(k)})\|_2^2, \tag{55}$$

where $\gamma_1, \gamma_2 \in [0,1]$. It normalizes these variables as:

$$\widehat{\boldsymbol{m}}^{(k+1)} := \frac{1}{1-\gamma_1^k}\boldsymbol{m}^{(k+1)}, \quad \widehat{v}^{(k+1)} := \frac{1}{1-\gamma_2^k} v^{(k+1)}.$$

- Then, it updates the solution as:

$$\boldsymbol{x}^{(k+1)} := \boldsymbol{x}^{(k)} - \frac{\eta^{(k)}}{\sqrt{\varepsilon + \widehat{v}^{(k+1)}}}\widehat{\boldsymbol{m}}^{(k+1)}, \tag{56}$$

which is stochastic gradient descent with momentum while using RMSProp.

- The Adam optimizer is one of the mostly used optimizers in **neural networks**.

Coding a Neural Network

Neural network: importing packages

Importing packages

```
[188] # installation in Google Colab's Jupyter notebook:
      !pip install torch
```

```
Looking in indexes: https://pypi.org/simple, https://us-python.pkg.dev/colab-wheels/public/simple/
Requirement already satisfied: torch in /usr/local/lib/python3.8/dist-packages (1.13.1+cu116)
Requirement already satisfied: typing-extensions in /usr/local/lib/python3.8/dist-packages (from torch) (4.4.0)
```

```
[320] # importing packages (libraries):
      import torch
      import torch.nn as nn
      from torch.utils.data import DataLoader, Dataset
      import matplotlib.pyplot as plt
      import numpy as np
      from tqdm import tqdm
```

Neural network: defining the network

▾ Defining the network

```
[216] device = torch.device("cuda:0" if torch.cuda.is_available() else "cpu")
```

```
[339] # defining the structure of neural network:
      class NeuralNetwork(nn.Module):
          def __init__(self):
              super(NeuralNetwork, self).__init__()
              self.layer1 = nn.Linear(1, 10)
              self.layer2 = nn.Linear(10, 20)
              self.layer3 = nn.Linear(20, 10)
              self.layer4 = nn.Linear(10, 1)
              self.relu1 = nn.ReLU()
              self.relu2 = nn.ReLU()
              self.relu3 = nn.ReLU()

          def forward(self, x):
              x = self.relu1(self.layer1(x))
              x = self.relu2(self.layer2(x))
              x = self.relu3(self.layer3(x))
              x = self.layer4(x)
              return x
```

```
[340] # instantiate the class of neural network:
      net = NeuralNetwork()
      print(net)
```

```
NeuralNetwork(
  (layer1): Linear(in_features=1, out_features=10, bias=True)
  (layer2): Linear(in_features=10, out_features=20, bias=True)
  (layer3): Linear(in_features=20, out_features=10, bias=True)
  (layer4): Linear(in_features=10, out_features=1, bias=True)
  (relu1): ReLU()
  (relu2): ReLU()
  (relu3): ReLU()
)
```

Neural network: optimizer

Optimizer

```
[341] # define optimizer:
      optimizer = 'Adam'
      if optimizer == 'SGD':
          optimizer = torch.optim.SGD(net.parameters(), lr=0.02)
      elif optimizer == 'Adam':
          optimizer = torch.optim.Adam(net.parameters(), lr=0.02)

      # define the loss function:
      loss_func = torch.nn.MSELoss()
```

Neural network: data loader

Data loader

```
[342] class Data(Dataset):
          def __init__(self, x, y):
              self.data = torch.from_numpy(x.reshape(-1,1)).float()
              self.label = torch.from_numpy(y.reshape(-1,1)).float()

          def __len__(self):
              return len(self.data)

          def __getitem__(self, item):
              data_point = self.data[item]
              label_point = self.label[item]
              return data_point, label_point
```

```
[343] batch_size = 16
      def load_dataset(x_train, y_train, x_test, y_test):
          # data loader for training data:
          train_ds = Data(x_train, y_train)
          train_loader = DataLoader(train_ds, batch_size=batch_size, shuffle=True)

          # data loader for test data:
          test_ds = Data(x_test, y_test)
          test_loader = DataLoader(test_ds, batch_size=batch_size, shuffle=False)

          return train_loader, test_loader
```

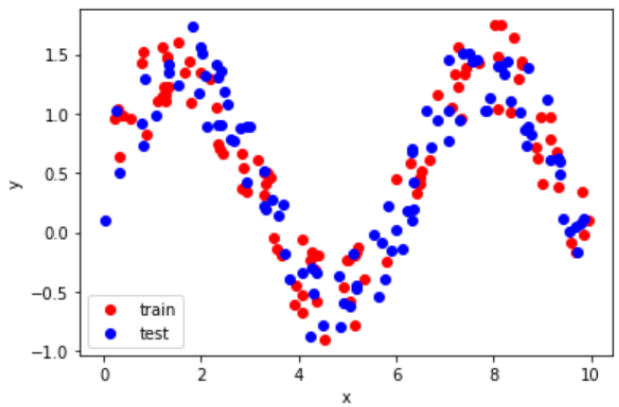
Neural network: dataset

```
dataset_type = 'nonlinear'

if dataset_type == 'linear':
    # almost linear dataset:
    x_train = np.random.rand(100)
    y_train = np.sin(x_train) * (x_train**3) + 3*x_train + np.random.rand(100)*0.8
    x_test = np.random.rand(100)
    y_test = np.sin(x_test) * (x_test**3) + 3*x_test + np.random.rand(100)*0.8
elif dataset_type == 'nonlinear':
    # dataset:
    x_train = np.random.rand(100) * 10
    y_train = np.sin(x_train) + np.random.rand(100)*0.8
    x_test = np.random.rand(100) * 10
    y_test = np.sin(x_test) + np.random.rand(100)*0.8

# reshape to have samples in rows:
x_train = x_train.reshape((-1, 1))
x_test = x_test.reshape((-1, 1))

# visualize data:
plt.scatter(x_train, y_train, c='r', label='train')
plt.scatter(x_test, y_test, c='b', label='test')
plt.xlabel('x')
plt.ylabel('y')
plt.legend()
plt.show()
```



Neural network: training

Training neural network

```
[345] # load dataset:
      train_loader, test_loader = load_dataset(x_train, y_train, x_test, y_test)
```

```
[346] n_epochs = 1000
      loss_list = []
      for epoch in tqdm(range(n_epochs), desc='epochs'):
          loss_list_in_epoch = []
          for step, (data_batch, label_batch) in enumerate(train_loader):
              data_batch, label_batch = data_batch.to(device), label_batch.to(device)
              prediction = net(data_batch)
              loss = loss_func(prediction, label_batch)
              loss_list_in_epoch.append(loss.cpu().detach().item())
              optimizer.zero_grad()
              loss.backward()
              optimizer.step()
          loss_list.append(np.mean(loss_list_in_epoch))
```

```
epochs: 100%|██████████| 1000/1000 [00:12<00:00, 81.84it/s]
```

Neural network: test (evaluation)

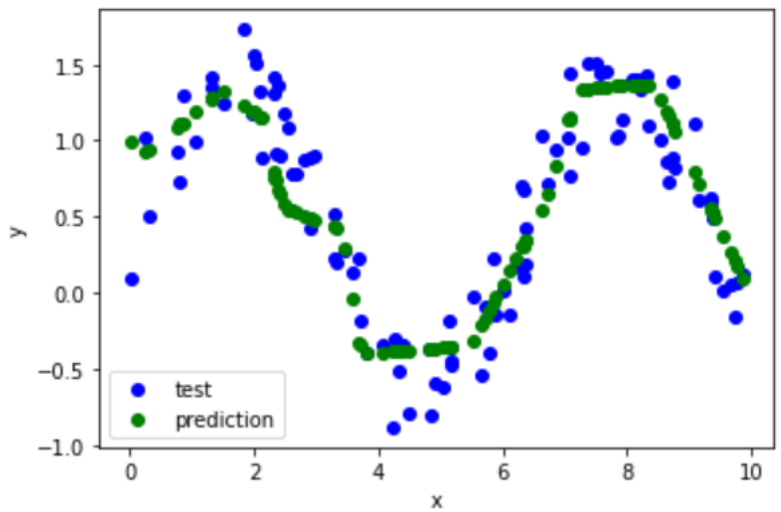
▾ Test (evaluation) phase

```
prediction_list = []
with torch.no_grad():
    for step, (data_batch, label_batch) in enumerate(test_loader):
        prediction = net(data_batch)
        prediction_list.extend(prediction)
```

0s

```
[350] # visualize the predicted and actual data:
      plt.scatter(x_test, y_test, c='b', label='test')
      plt.scatter(x_test, prediction_list, c='g', label='prediction')
      plt.xlabel('x')
      plt.ylabel('y')
      plt.legend()
      plt.show()
```

0s



Proximal Methods

Proximal Mapping

Definition (Proximal mapping/operator [26])

The proximal mapping or proximal operator of a convex function $g(.)$ is:

$$\mathbf{prox}_g(\boldsymbol{x}) := \arg\min_{\boldsymbol{u}} \left(g(\boldsymbol{u}) + \frac{1}{2}\|\boldsymbol{u} - \boldsymbol{x}\|_2^2 \right). \tag{57}$$

In case the function $g(.)$ is scaled by a scalar λ, the proximal mapping is defined as:

$$\mathbf{prox}_{\lambda g}(\boldsymbol{x}) := \arg\min_{\boldsymbol{u}} \left(g(\boldsymbol{u}) + \frac{1}{2\lambda}\|\boldsymbol{u} - \boldsymbol{x}\|_2^2 \right). \tag{58}$$

The Moreau-Yosida regularization

- The proximal mapping:

$$\mathbf{prox}_g(\boldsymbol{x}) := \arg\min_{\boldsymbol{u}} \left(g(\boldsymbol{u}) + \frac{1}{2}\|\boldsymbol{u} - \boldsymbol{x}\|_2^2 \right),$$

  is related to the Moreau-Yosida regularization defined below.

Definition (Moreau-Yosida regularization or Moreau envelope [27, 28])

The Moreau-Yosida regularization or the Moreau envelope of function $g(.)$ is:

$$M_{\lambda g}(\boldsymbol{x}) := \inf_{\boldsymbol{u}} \left(g(\boldsymbol{u}) + \frac{1}{2}\|\boldsymbol{u} - \boldsymbol{x}\|_2^2 \right). \tag{59}$$

This Moreau-Yosida regularized function has the same minimizer as the function $g(.)$ [29].

The Moreau decomposition

Lemma (Moreau decomposition [30])

We always have the following decomposition, named the Moreau decomposition:

$$\boldsymbol{x} = \mathbf{prox}_{g}(\boldsymbol{x}) + \mathbf{prox}_{g^*}(\boldsymbol{x}), \tag{60}$$

$$\boldsymbol{x} = \mathbf{prox}_{\lambda g}(\boldsymbol{x}) + \lambda\, \mathbf{prox}_{\frac{1}{\lambda} g^*}(\frac{\boldsymbol{x}}{\lambda}), \tag{61}$$

where $g(.)$ is a function in a space and $g^*(.)$ is its corresponding function in the dual space (e.g., if $g(.)$ is a norm, $g^*(.)$ is its dual norm or if $g(.)$ is projection onto a cone, $g^*(.)$ is projection onto the dual cone).

Projection onto set

- In optimization, **indicator function** $\mathbb{I}(.)$ is zero if its condition is satisfied and is infinite otherwise.

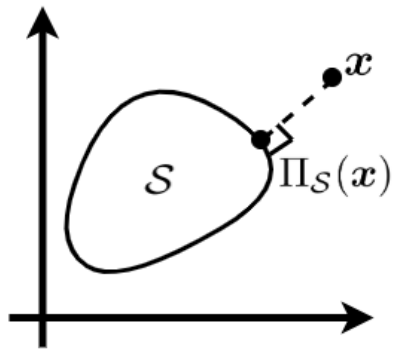
$$\mathbb{I}(\boldsymbol{x} \in \mathcal{S}) = \begin{cases} 0 & \text{if } \boldsymbol{x} \in \mathcal{S} \\ \infty & \text{if } \boldsymbol{x} \notin \mathcal{S}. \end{cases} \tag{62}$$

Lemma (Projection onto set)

The proximal mapping of the indicator function to a convex set $\mathcal{S}$, i.e. $\mathbb{I}(\boldsymbol{x} \in \mathcal{S})$, is projection of the point $\boldsymbol{x}$ onto the set $\mathcal{S}$. Hence, projection of $\boldsymbol{x}$ onto set $\mathcal{S}$, denoted by $\Pi_{\mathcal{S}}(\boldsymbol{x})$, is defined as:

$$\Pi_{\mathcal{S}}(\boldsymbol{x}) := \textbf{prox}_{\mathbb{I}(.\in\mathcal{S})}(\boldsymbol{x}) = \arg\min_{\boldsymbol{u}\in\mathcal{S}} \left(\frac{1}{2}\|\boldsymbol{u}-\boldsymbol{x}\|_2^2\right). \tag{63}$$

- This projection simply means projecting the point $\boldsymbol{x}$ onto the closest point of set from the point $\boldsymbol{x}$. Hence, the vector connecting the points $\boldsymbol{x}$ and $\Pi_{\mathcal{S}}(\boldsymbol{x})$ is orthogonal to the set $\mathcal{S}$.



Projection onto set

Lemma:

$$\Pi_{\mathcal{S}}(\boldsymbol{x}) := \mathbf{prox}_{\mathbb{I}(.\in\mathcal{S})}(\boldsymbol{x}) = \arg\min_{\boldsymbol{u}\in\mathcal{S}} \left(\frac{1}{2}\|\boldsymbol{u}-\boldsymbol{x}\|_2^2\right).$$

The proximal mapping:

$$\mathbf{prox}_{g}(\boldsymbol{x}) := \arg\min_{\boldsymbol{u}} \left(g(\boldsymbol{u}) + \frac{1}{2}\|\boldsymbol{u}-\boldsymbol{x}\|_2^2\right),$$

Proof.

$$\mathbf{prox}_{\mathbb{I}(.\in\mathcal{S})}(\boldsymbol{x}) \overset{(57)}{=} \arg\min_{\boldsymbol{u}} \left(\mathbb{I}(\boldsymbol{x}\in\mathcal{S}) + \frac{1}{2}\|\boldsymbol{u}-\boldsymbol{x}\|_2^2\right)$$
$$\overset{(a)}{=} \arg\min_{\boldsymbol{u}\in\mathcal{S}} \left(\frac{1}{2}\|\boldsymbol{u}-\boldsymbol{x}\|_2^2\right),$$

where (a) is because $\mathbb{I}(\boldsymbol{x}\in\mathcal{S})$ becomes infinity if $\boldsymbol{x}\notin\mathcal{S}$ (see Eq. (62)). □

Moreau decomposition for norm

Recall Eq. (61) from the Moreau decomposition:

$$\boldsymbol{x} = \mathbf{prox}_{\lambda g}(\boldsymbol{x}) + \lambda\, \mathbf{prox}_{\frac{1}{\lambda} g^*}(\frac{\boldsymbol{x}}{\lambda}) \implies \mathbf{prox}_{\lambda g}(\boldsymbol{x}) = \boldsymbol{x} - \lambda\, \mathbf{prox}_{\frac{1}{\lambda} g^*}(\frac{\boldsymbol{x}}{\lambda}).$$

Corollary (Moreau decomposition for norm)

If the function is a scaled norm, $g(.) = \lambda\|.\|$, we have from this equation:

$$\mathbf{prox}_{\lambda\|.\|}(\boldsymbol{x}) = \boldsymbol{x} - \lambda\, \Pi_{\mathcal{B}}(\frac{\boldsymbol{x}}{\lambda}), \tag{64}$$

where $\mathcal{B}$ is the unit ball of dual norm.

- Derivation of proximal operator for various $g(.)$ functions are available in [31, Chapter 6]. Here, we review the proximal mapping of some mostly used functions.
- If $g(\boldsymbol{x}) = 0$, proximal mapping becomes an identity mapping:

$$\mathbf{prox}_{\lambda 0}(\boldsymbol{x}) \overset{(58)}{=} \arg\min_{\boldsymbol{u}} \left(0 + \frac{1}{2\lambda}\|\boldsymbol{u} - \boldsymbol{x}\|_2^2\right) = \arg\min_{\boldsymbol{u}} \left(\frac{1}{2\lambda}\|\boldsymbol{u} - \boldsymbol{x}\|_2^2\right) = \boldsymbol{x}.$$

Proximal mapping of ℓ_2 norm

Lemma (Proximal mapping of ℓ_2 norm [31, Example 6.19])

The proximal mapping of the ℓ_2 norm is:

$$\mathbf{prox}_{\lambda\|.\|_2}(\boldsymbol{x}) = \Big(1 - \frac{\lambda}{\max(\|\boldsymbol{x}\|_2, \lambda)}\Big)\boldsymbol{x} = \begin{cases} (1 - \frac{\lambda}{\|\boldsymbol{x}\|_2})\boldsymbol{x} & \textit{if } \|\boldsymbol{x}\|_2 \geq \lambda \\ 0 & \textit{if } \|\boldsymbol{x}\|_2 < \lambda. \end{cases} \tag{65}$$

Proof.

Recall Eq. (64): $\mathbf{prox}_{\lambda\|.\|}(\boldsymbol{x}) = \boldsymbol{x} - \lambda\,\Pi_{\mathcal{B}}(\frac{\boldsymbol{x}}{\lambda})$.

Let $g(.) = \|.\|_2$ and $\mathcal{B}$ be the unit ℓ_2 ball because ℓ_2 is the dual norm of ℓ_2. We have:

$$\Pi_{\mathcal{B}}(\boldsymbol{x}) = \begin{cases} \boldsymbol{x}/\|\boldsymbol{x}\|_2 & \text{if } \|\boldsymbol{x}\|_2 \geq 1 \\ \boldsymbol{x} & \text{if } \|\boldsymbol{x}\|_2 < 1. \end{cases}$$

$$\implies \mathbf{prox}_{\lambda\|.\|_2}(\boldsymbol{x}) \overset{(64)}{=} \boldsymbol{x} - \lambda\,\Pi_{\mathcal{B}}(\frac{\boldsymbol{x}}{\lambda}) = \begin{cases} (1 - \frac{\lambda}{\|\boldsymbol{x}\|_2})\boldsymbol{x} & \text{if } \|\boldsymbol{x}\|_2 \geq \lambda \\ \boldsymbol{x} - \lambda(\boldsymbol{x}/\lambda) = 0 & \text{if } \|\boldsymbol{x}\|_2 < \lambda. \end{cases}$$

□

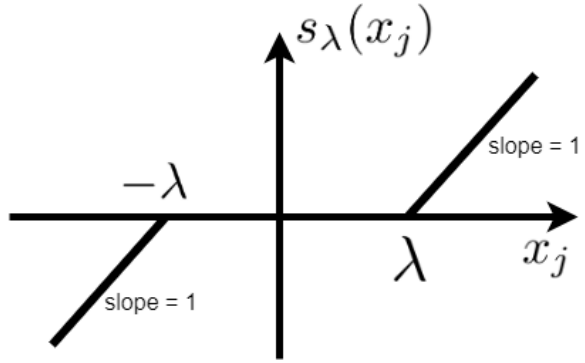
Proximal mapping of ℓ_1 norm

Lemma (Proximal mapping of ℓ_1 norm [31, Example 6.8])

Let x_j denote the j-th element of $\boldsymbol{x} = [x_1, \ldots, x_d]^\top \in \mathbb{R}^d$ and let $[\textbf{prox}_{\lambda\|.\|_1}(\boldsymbol{x})]_j$ denote the j-th element of the d-dimensional $\textbf{prox}_{\lambda\|.\|_1}(\boldsymbol{x})$ mapping. The j-th element of proximal mapping of the ℓ_1 norm is:

$$[\textbf{prox}_{\lambda\|.\|_1}(\boldsymbol{x})]_j = \max(0, |x_j| - \lambda)\,\textbf{sign}(x_j) = s_\lambda(x_j) := \begin{cases} x_j - \lambda & \text{if } x_j \geq \lambda \\ 0 & \text{if } |x_j| < \lambda \\ x_j + \lambda & \text{if } x_j \leq -\lambda, \end{cases} \tag{66}$$

for all $j \in \{1, \ldots, d\}$. Eq. (66) is called the **soft-thresholding** *function, denoted here by $s_\lambda(.)$.*



Proximal mapping of ℓ_1 norm

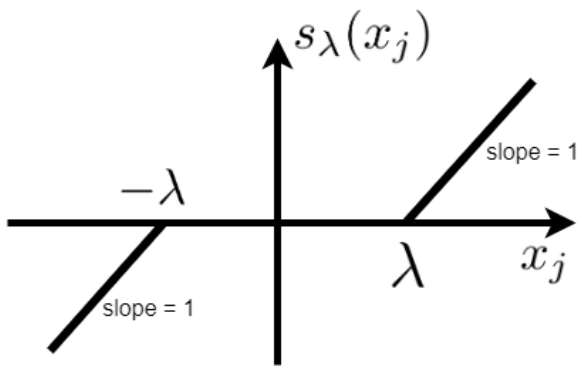
Proof.

Let $g(.) = \|.\|_1$ and $\mathcal{B}$ be the unit ℓ_∞ ball because ℓ_∞ is the dual norm of ℓ_1. The j-th element of projection is:

$$[\Pi_{\mathcal{B}}(\boldsymbol{x})]_j = \begin{cases} 1 & \text{if } x_j \geq 1 \\ x_j & \text{if } |x_j| < 1 \\ -1 & \text{if } x_j \leq -1 \end{cases}$$

$$\implies [\mathbf{prox}_{\lambda\|.\|_1}(\boldsymbol{x})]_j \overset{(64)}{=} x_j - \lambda\,[\Pi_{\mathcal{B}}(\frac{\boldsymbol{x}}{\lambda})]_j = \begin{cases} x_j - \lambda & \text{if } x_j \geq \lambda \\ 0 & \text{if } |x_j| < \lambda \\ x_j + \lambda & \text{if } x_j \leq -\lambda. \end{cases}$$

□



Proximal point algorithm

Recall Eq. (58):

$$\mathbf{prox}_{\lambda g}(\boldsymbol{x}) := \arg\min_{\boldsymbol{u}} \left(g(\boldsymbol{u}) + \frac{1}{2\lambda} \|\boldsymbol{u} - \boldsymbol{x}\|_2^2 \right).$$

The term $g(\boldsymbol{u}) + 1/(2\lambda)\|\boldsymbol{u} - \boldsymbol{x}\|_2^2$ in this equation is strongly convex; hence, the proximal point, $\mathbf{prox}_{\lambda g}(\boldsymbol{x})$, is unique.

Lemma

The point $\boldsymbol{x}^*$ minimizes the function $f(.)$ if and only if $\boldsymbol{x}^* = \mathbf{prox}_{\lambda f}(\boldsymbol{x}^*)$.

Proximal point algorithm

- Consider the following optimization problem:

$$\underset{\boldsymbol{x}}{\text{minimize}} \quad f(\boldsymbol{x}).$$

- **Proximal point algorithm**, also called **proximal minimization**, was proposed in 1976 [32]. It finds the optimal point of this problem by iteratively updating the solution as:

$$\boldsymbol{x}^{(k+1)} := \mathbf{prox}_{\lambda f}(\boldsymbol{x}^{(k)}) \overset{(58)}{=} \arg\min_{\boldsymbol{u}} \left(f(\boldsymbol{u}) + \frac{1}{2\lambda} \|\boldsymbol{u} - \boldsymbol{x}^{(k)}\|_2^2 \right), \tag{67}$$

  until convergence.

- λ can be seen as a parameter related to the step size.
- In other words, proximal gradient method applies gradient descent on the Moreau envelope $M_{\lambda f}(\boldsymbol{x})$, recall Eq. (59):

$$M_{\lambda g}(\boldsymbol{x}) := \inf_{\boldsymbol{u}} \left(g(\boldsymbol{u}) + \frac{1}{2} \|\boldsymbol{u} - \boldsymbol{x}\|_2^2 \right),$$

  rather than on the function $f(.)$ itself.

Composite problems

Definition (Composite objective function [14])

In optimization, if a function is stated as a summation of two terms, $f(\boldsymbol{x}) + g(\boldsymbol{x})$, it is called a **composite function** and its optimization is a **composite optimization problem**.

- Consider the following optimization problem:

$$\underset{\boldsymbol{x}}{\text{minimize}} \quad f(\boldsymbol{x}) + g(\boldsymbol{x}), \tag{68}$$

  where $f(\boldsymbol{x})$ is a smooth function and $g(\boldsymbol{x})$ is a convex function which is not smooth necessarily. This is a composite optimization problem.

- Composite problems are widely used in machine learning and **regularized** problems because $f(\boldsymbol{x})$ can be the cost function to be minimized while $g(\boldsymbol{x})$ is the **penalty or regularization** term [33].

Proximal gradient method

- For solving problem (68), $\underset{\boldsymbol{x}}{\text{minimize}} \quad f(\boldsymbol{x}) + g(\boldsymbol{x})$, we can approximate the function $f(.)$ by its quadratic approximation around point $\boldsymbol{x}$ because it is smooth (differentiable):

$$f(\boldsymbol{u}) \approx f(\boldsymbol{x}) + \nabla f(\boldsymbol{x})^\top (\boldsymbol{u} - \boldsymbol{x}) + \frac{1}{2\eta} \|\boldsymbol{u} - \boldsymbol{x}\|_2^2,$$

where we have replaced $\nabla^2 f(\boldsymbol{x})$ with scaled identity matrix, $(1/\eta)\boldsymbol{I}$.

- Hence, the solution of problem (68) can be approximated as:

$$\begin{aligned}
\boldsymbol{x} &= \arg\min_{\boldsymbol{u}} \big(f(\boldsymbol{u}) + g(\boldsymbol{u})\big) \\
&\approx \arg\min_{\boldsymbol{u}} \big(f(\boldsymbol{x}) + \nabla f(\boldsymbol{x})^\top (\boldsymbol{u} - \boldsymbol{x}) + \frac{1}{2\eta} \|\boldsymbol{u} - \boldsymbol{x}\|_2^2 + g(\boldsymbol{u})\big) \\
&\overset{(a)}{=} \arg\min_{\boldsymbol{u}} \big(\|\nabla f(\boldsymbol{x})\|_2^2 + \nabla f(\boldsymbol{x})^\top (\boldsymbol{u} - \boldsymbol{x}) + \frac{1}{2\eta} \|\boldsymbol{u} - \boldsymbol{x}\|_2^2 + g(\boldsymbol{u})\big) \\
&\overset{(b)}{=} \arg\min_{\boldsymbol{u}} \big(\frac{1}{2\eta} \|(\boldsymbol{u} - \boldsymbol{x}) + \eta \nabla f(\boldsymbol{x})\|_2^2 + g(\boldsymbol{u})\big) \\
&= \arg\min_{\boldsymbol{u}} \big(\frac{1}{2\eta} \|\boldsymbol{u} - (\boldsymbol{x} - \eta \nabla f(\boldsymbol{x}))\|_2^2 + g(\boldsymbol{u})\big), \qquad (69)
\end{aligned}$$

where (a) is because $f(\boldsymbol{x})$ is a constant in minimization w.r.t. $\boldsymbol{u}$, so it can be simply replaced with another constant $\|\nabla f(\boldsymbol{x})\|_2^2$, and (b) is because of writing the three first terms as the binomial square term.

Proximal gradient method

- We found Eq. (69):

$$\boldsymbol{x} = \arg\min_{\boldsymbol{u}} \big(\frac{1}{2\eta}\|\boldsymbol{u} - (\boldsymbol{x} - \eta\nabla f(\boldsymbol{x}))\|_2^2 + g(\boldsymbol{u})\big).$$

- The first term in this equation keeps the solution close to the solution of gradient descent for minimizing the function $f(.)$ (see Eq. (10)) and the second term in this equation makes the function $g(.)$ small.
- **Proximal gradient method**, also called **proximal gradient descent**, uses Eq. (69) for solving the composite problem (68). It was first proposed in 2013 [14] and also in [34] for $g = \|.\|_1$. It finds the optimal point by iteratively updating the solution as:

$$\begin{aligned}\boldsymbol{x}^{(k+1)} &\overset{(69)}{:=} \arg\min_{\boldsymbol{u}} \big(\frac{1}{2\eta^{(k)}}\|\boldsymbol{u} - (\boldsymbol{x}^{(k)} - \eta^{(k)}\nabla f(\boldsymbol{x}^{(k)}))\|_2^2 + g(\boldsymbol{u})\big) \\ &\overset{(58)}{=} \textbf{prox}_{\eta^{(k)}g}\big(\boldsymbol{x}^{(k)} - \eta^{(k)}\nabla f(\boldsymbol{x}^{(k)})\big),\end{aligned} \tag{70}$$

  until convergence, where $\eta^{(k)}$ is the step size which can be fixed or found by line-search.
- In Eq. (68), $\underset{\boldsymbol{x}}{\text{minimize}}\, f(\boldsymbol{x}) + g(\boldsymbol{x})$, the function $g(.)$ can be a regularization term such as ℓ_2 or ℓ_1 norm. In these cases, we use Lemmas 12 and 13 for calculating Eq. (70).

Constrained First-order Optimization

Projected gradient method

- **Projected gradient method**, proposed in 2003 [35], also called **gradient projection method** and **projected gradient descent**, considers $g(\boldsymbol{x})$ to be the indicator function $\mathbb{I}(\boldsymbol{x} \in \mathcal{S})$ in problem (68), $\underset{\boldsymbol{x}}{\text{minimize}}\, f(\boldsymbol{x}) + g(\boldsymbol{x})$.
- This optimization problem is a constrained problem which can be restated to:

$$\underset{\boldsymbol{x}}{\text{minimize}} \quad f(\boldsymbol{x}) + \mathbb{I}(\boldsymbol{x} \in \mathcal{S}), \tag{71}$$

because the indicator function becomes infinity if its condition is not satisfied.

- Recall Eq. (70) in proximal gradient method:

$$\boldsymbol{x}^{(k+1)} := \mathbf{prox}_{\eta^{(k)} g}\left(\boldsymbol{x}^{(k)} - \eta^{(k)} \nabla f(\boldsymbol{x}^{(k)})\right).$$

According to this equation, the solution is updated as:

$$\begin{aligned} \boldsymbol{x}^{(k+1)} &\overset{(70)}{:=} \mathbf{prox}_{\eta^{(k)} \mathbb{I}(. \in \mathcal{S})}\left(\boldsymbol{x}^{(k)} - \eta^{(k)} \nabla f(\boldsymbol{x}^{(k)})\right) \\ &\overset{(63)}{=} \Pi_{\mathcal{S}}\left(\boldsymbol{x}^{(k)} - \eta^{(k)} \nabla f(\boldsymbol{x}^{(k)})\right). \end{aligned} \tag{72}$$

- So, projected gradient method performs a step of gradient descent and then projects the solution onto the set of constraint. This procedure is repeated until convergence of solution.

Projected gradient method

- Although most often projected gradient method is used for Eq. (72), there are few other variants of projected gradient methods such as the following proposed in 2004 [36]:

$$\boldsymbol{y}^{(k)} := \Pi_{\boldsymbol{S}}\left(\boldsymbol{x}^{(k)} - \eta^{(k)} \nabla f(\boldsymbol{x}^{(k)})\right), \tag{73}$$

$$\boldsymbol{x}^{(k+1)} := \boldsymbol{x}^{(k)} + \gamma^{(k)}(\boldsymbol{y}^{(k)} - \boldsymbol{x}^{(k)}), \tag{74}$$

  where $\eta^{(k)}$ and $\gamma^{(k)}$ are positive step sizes at iteration k.

- In this alternating approach, we find an additional variable $\boldsymbol{y}$ by gradient descent followed by projection and then update $\boldsymbol{x}$ to get close to the found $\boldsymbol{y}$ while staying close to the previous solution by line-search.

Projection onto the cone of orthogonal matrices

- **Singular Value Decomposition (SVD)** decomposes the matrix $\boldsymbol{X} \in \boldsymbol{X} \in \mathbb{R}^{d_1 \times d_2}$ as:

$$\boldsymbol{X} = \boldsymbol{U}\boldsymbol{\Sigma}\boldsymbol{V}^\top,$$

  where $\boldsymbol{U} \in \mathbb{R}^{d_1 \times d_1}$ and $\boldsymbol{V} \in \mathbb{R}^{d_2 \times d_2}$ are the matrices of left and right singular vectors of $\boldsymbol{X}$, respectively.

- Consider the constraint for projection onto the cone of orthogonal matrices, i.e., $\boldsymbol{X}^\top \boldsymbol{X} = \boldsymbol{I}$. In this constraint, the constraint deals with the singular values of $\boldsymbol{X}$, because:

$$\begin{aligned} \boldsymbol{X} \overset{\text{SVD}}{=} \boldsymbol{U}\boldsymbol{\Sigma}\boldsymbol{V}^\top \implies \boldsymbol{X}^\top\boldsymbol{X} = \boldsymbol{U}\boldsymbol{\Sigma}\boldsymbol{V}^\top\boldsymbol{V}\boldsymbol{\Sigma}\boldsymbol{U}^\top &\overset{(a)}{=} \boldsymbol{U}\boldsymbol{\Sigma}^2\boldsymbol{U}^\top \overset{\text{set}}{=} \boldsymbol{I} \\ \implies \boldsymbol{U}\boldsymbol{\Sigma}^2\boldsymbol{U}^\top\boldsymbol{U} = \boldsymbol{U} &\overset{(b)}{\implies} \boldsymbol{U}\boldsymbol{\Sigma}^2 = \boldsymbol{U} \implies \boldsymbol{\Sigma} = \boldsymbol{I}, \end{aligned}$$

  where (a) and (b) are because $\boldsymbol{U}$ and $\boldsymbol{V}$ are orthogonal matrices.

- Therefore, the constraint $\boldsymbol{X}^\top \boldsymbol{X} = \boldsymbol{I}$ (i.e., projecting onto the cone of orthogonal matrices) can be modeled by setting all singular values of $\boldsymbol{X}$ to one:

$$\textbf{prox}_{\lambda,g}(\boldsymbol{X}) = \Pi_{\mathcal{O}} = \boldsymbol{U}\boldsymbol{I}\boldsymbol{V}^\top, \tag{75}$$

  where $\boldsymbol{I} \in \mathbb{R}^{d_1 \times d_2}$ is a rectangular identity matrix and $\mathcal{O}$ denotes the cone of orthogonal matrices.

- If the constraint is scaled orthogonality, i.e. $\boldsymbol{X}^\top \boldsymbol{X} = \lambda \boldsymbol{I}$ with λ as the scale, the projection is setting all singular values to λ by $\boldsymbol{U}(\lambda \boldsymbol{I})\boldsymbol{V}^\top = \lambda \boldsymbol{U}\boldsymbol{I}\boldsymbol{V}^\top$.

Projection onto convex sets (POCS)

- Assume we want to project a point onto the intersection of c closed convex sets, i.e., $\bigcap_{j=1}^{c} \mathcal{S}_j$.
- We can model this by an optimization problem with a fake objective function:

$$\begin{aligned} \underset{\boldsymbol{x}}{\text{minimize}} \quad & \boldsymbol{x} \in \mathbb{R}^d \\ \text{subject to} \quad & \boldsymbol{x} \in \mathcal{S}_1, \ldots, \boldsymbol{x} \in \mathcal{S}_c. \end{aligned} \tag{76}$$

- **Projection Onto Convex Sets (POCS)** solves this problem, similar to projected gradient method, by projecting onto the sets one-by-one [37]:

$$\boldsymbol{x}^{(k+1)} := \Pi_{\boldsymbol{S}_1}(\Pi_{\boldsymbol{S}_2}(\ldots \Pi_{\boldsymbol{S}_c}(\boldsymbol{x}^{(k)}) \ldots)), \tag{77}$$

  and repeating it until convergence.
- Another similar method for solving problem (76) is the **averaged projections** which updates the solution as:

$$\boldsymbol{x}^{(k+1)} := \frac{1}{c}\Big(\Pi_{\boldsymbol{S}_1}(\boldsymbol{x}^{(k)}) + \cdots + \Pi_{\boldsymbol{S}_c}(\boldsymbol{x}^{(k)})\Big). \tag{78}$$

Acknowledgement

- Some slides of this slide deck are inspired by the lectures of Prof. Kimon Fountoulakis at the University of Waterloo.
- Some slides of this slide deck are inspired by the lectures of Prof. Stephen Boyd at the Stanford University.
- Our tutorial also has the materials of this slide deck: [38]

References

[1] C. Lemaréchal, "Cauchy and the gradient method," *Doc Math Extra*, vol. 251, no. 254, p. 10, 2012.

[2] J. Hadamard, *Mémoire sur le problème d'analyse relatif à l'équilibre des plaques élastiques encastrées*, vol. 33.
Imprimerie nationale, 1908.

[3] H. B. Curry, "The method of steepest descent for non-linear minimization problems," *Quarterly of Applied Mathematics*, vol. 2, no. 3, pp. 258–261, 1944.

[4] L. Armijo, "Minimization of functions having Lipschitz continuous first partial derivatives," *Pacific Journal of mathematics*, vol. 16, no. 1, pp. 1–3, 1966.

[5] J. Nocedal and S. Wright, *Numerical optimization*.
Springer Science & Business Media, 2 ed., 2006.

[6] P. Wolfe, "Convergence conditions for ascent methods," *SIAM review*, vol. 11, no. 2, pp. 226–235, 1969.

[7] D. E. Rumelhart, G. E. Hinton, and R. J. Williams, "Learning representations by back-propagating errors," *nature*, vol. 323, no. 6088, pp. 533–536, 1986.

[8] E. K. Chong and S. H. Zak, *An introduction to optimization*.
John Wiley & Sons, 2004.

References (cont.)

[9] S. Boyd and L. Vandenberghe, *Convex optimization*. Cambridge university press, 2004.

[10] Y. Nesterov, "A method for solving the convex programming problem with convergence rate O(1/k^2)," in *Dokl. Akad. Nauk SSSR*, vol. 269, pp. 543–547, 1983.

[11] Y. Nesterov, *Introductory lectures on convex optimization: A basic course*, vol. 87. Springer Science & Business Media, 2003.

[12] Y. Nesterov, "On an approach to the construction of optimal methods of minimization of smooth convex functions," *Ekonomika i Mateaticheskie Metody*, vol. 24, no. 3, pp. 509–517, 1988.

[13] Y. Nesterov, "Smooth minimization of non-smooth functions," *Mathematical programming*, vol. 103, no. 1, pp. 127–152, 2005.

[14] Y. Nesterov, "Gradient methods for minimizing composite functions," *Mathematical Programming*, vol. 140, no. 1, pp. 125–161, 2013.

[15] H. Robbins and S. Monro, "A stochastic approximation method," *The annals of mathematical statistics*, pp. 400–407, 1951.

[16] L. Bottou *et al.*, "Online learning and stochastic approximations," *On-line learning in neural networks*, vol. 17, no. 9, p. 142, 1998.

References (cont.)

[17] I. Goodfellow, Y. Bengio, and A. Courville, *Deep learning*. MIT press, 2016.

[18] B. Ghojogh, H. Nekoei, A. Ghojogh, F. Karray, and M. Crowley, "Sampling algorithms, from survey sampling to Monte Carlo methods: Tutorial and literature review," *arXiv preprint arXiv:2011.00901*, 2020.

[19] N. L. Roux, M. Schmidt, and F. Bach, "A stochastic gradient method with an exponential convergence rate for finite training sets," in *Advances in Neural Information Processing Systems*, vol. 25, 2012.

[20] R. Johnson and T. Zhang, "Accelerating stochastic gradient descent using predictive variance reduction," *Advances in neural information processing systems*, vol. 26, pp. 315–323, 2013.

[21] J. Duchi, E. Hazan, and Y. Singer, "Adaptive subgradient methods for online learning and stochastic optimization," *Journal of machine learning research*, vol. 12, no. 7, 2011.

[22] T. Tieleman and G. Hinton, "Lecture 6.5-rmsprop: Divide the gradient by a running average of its recent magnitude," *COURSERA: Neural networks for machine learning*, vol. 4, no. 2, pp. 26–31, 2012.

[23] M. Riedmiller and H. Braun, "Rprop-a fast adaptive learning algorithm," in *Proceedings of the International Symposium on Computer and Information Science VII*, 1992.

References (cont.)

[24] G. Hinton, N. Srivastava, and K. Swersky, "Neural networks for machine learning lecture 6a overview of mini-batch gradient descent," tech. rep., Department of Computer Science, University of Toronto, 2012.

[25] D. P. Kingma and J. Ba, "Adam: A method for stochastic optimization," *arXiv preprint arXiv:1412.6980*, 2014.

[26] N. Parikh and S. Boyd, "Proximal algorithms," *Foundations and Trends in optimization*, vol. 1, no. 3, pp. 127–239, 2014.

[27] J. J. Moreau, "Proximité et dualité dans un espace hilbertien," *Bulletin de la Société mathématique de France*, vol. 93, pp. 273–299, 1965.

[28] K. Yosida, *Functional analysis*.
Springer Berlin Heidelberg, 1965.

[29] C. Lemaréchal and C. Sagastizábal, "Practical aspects of the Moreau–Yosida regularization: Theoretical preliminaries," *SIAM journal on optimization*, vol. 7, no. 2, pp. 367–385, 1997.

[30] J. J. Moreau, "Décomposition orthogonale d'un espace Hilbertien selon deux cônes mutuellement polaires," *Comptes rendus hebdomadaires des séances de l'Académie des sciences*, vol. 255, pp. 238–240, 1962.

[31] A. Beck, *First-order methods in optimization*.
SIAM, 2017.

References (cont.)

[32] R. T. Rockafellar, "Monotone operators and the proximal point algorithm," *SIAM journal on control and optimization*, vol. 14, no. 5, pp. 877–898, 1976.

[33] B. Ghojogh and M. Crowley, "The theory behind overfitting, cross validation, regularization, bagging, and boosting: tutorial," *arXiv preprint arXiv:1905.12787*, 2019.

[34] A. Beck and M. Teboulle, "A fast iterative shrinkage-thresholding algorithm for linear inverse problems," *SIAM journal on imaging sciences*, vol. 2, no. 1, pp. 183–202, 2009.

[35] A. N. Iusem, "On the convergence properties of the projected gradient method for convex optimization," *Computational & Applied Mathematics*, vol. 22, pp. 37–52, 2003.

[36] L. G. Drummond and A. N. Iusem, "A projected gradient method for vector optimization problems," *Computational Optimization and applications*, vol. 28, no. 1, pp. 5–29, 2004.

[37] H. H. Bauschke and J. M. Borwein, "On projection algorithms for solving convex feasibility problems," *SIAM review*, vol. 38, no. 3, pp. 367–426, 1996.

[38] B. Ghojogh, A. Ghodsi, F. Karray, and M. Crowley, "KKT conditions, first-order and second-order optimization, and distributed optimization: Tutorial and survey," *arXiv preprint arXiv:2110.01858*, 2021.