Chapter 2

This lecture covers *smooth continuous optimization* and provides the background knowledge before we delve into *convex optimization*. In order to explore these topics, we will require several basic definitions such as gradients, Hessian matrices, Taylor Series, etc. This chapter also "scratches" the surface of properties of optimization functions: Taylor expansion is reviewed and types of stationary points are introduced. Several special conditions that benefit optimization, including Lipschitz continuity and Lipschitz gradient continuity, are introduced. The main algorithm for this chapter will be *gradient descent (GD)*, as well as *projected GD*. Additionally, these notes explain *convergence rates*. We will see how further global assumptions lead to improved convergence guarantees.

Lipschitz conditions | Gradient Descent

This course mostly covers general smooth optimization, where the objective function can be pictured as a continuous curve in high dimensions. You can easily picture it: A continuous landscape parameterized by a set of unknowns, and the goal is to find the global minimum/maximum. However, there are other important classes of optimization problems, not covered in this course, that follow this description, as shown in the figure 3. Some of them are typically explored as a special topic, for example discrete optimization and integer programming. This course is restricted only to smooth functions. The *smoothness* will be defined later on in text. For now, one way to describe smoothness is by saying that we can compute gradients on these functions.

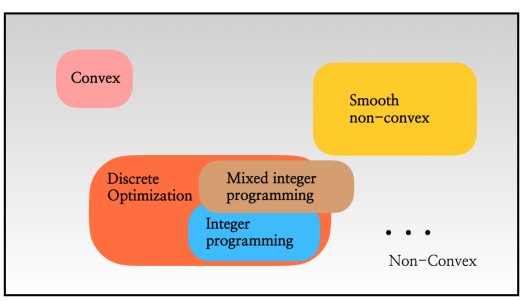


Fig. 3. Landscape of optimization

Derivatives, gradients and Hessians. Algorithms and heuristics in optimization often involve derivatives as a means of approaching an optimal solution. Put shortly, the derivative tells you the direction (and, in some way the magnitude) of steepest ascent (or descent).

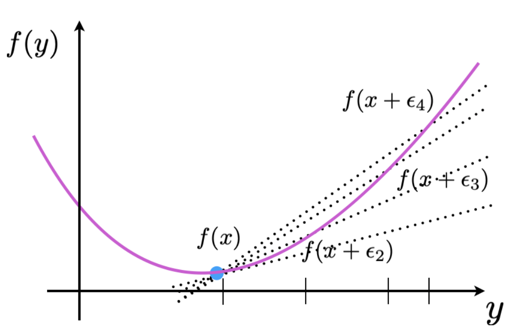


Fig. 4. Graphical illustration of first-order derivative

Definition 1. (First-order Derivative) *The derivative of a univariate function $f : \mathbb{R} \to \mathbb{R}$ at a point x is defined as:*

$$\frac{\partial f}{\partial x} = f'(x) = \lim_{\epsilon \to 0} \frac{f(x+\epsilon) - f(x)}{\epsilon}.$$

The derivative of f represents the slope f in a neighborhood of a point x. That is, it gives information about how much f changes within a very small area, when we perturb around a given point.

This in turn suggests the second-order derivative, which is recursively defined as the derivative of the derivative and describes how rapidly the derivative changes.

Definition 2. (Second-order Derivative) *The second-order derivative of a univariate function $f : \mathbb{R} \to \mathbb{R}$ at a point x is defined as:*

$$\frac{\partial^2 f}{\partial x^2} = f''(x) = \lim_{\epsilon \to 0} \frac{f'(x+\epsilon) - f'(x)}{\epsilon}.$$

The second-order derivative represents the *local curvature* of f, i.e. how much the slope of the function changes around a given point.

Some differentiation rules are:

- $(f(x) \cdot g(x))' = f'(x) \cdot g(x) + f(x) \cdot g'(x)$ (Product rule)
- $\left(\frac{f(x)}{g(x)}\right)' = \frac{f'(x) \cdot g(x) - f(x) \cdot g'(x)}{g^2(x)}$ (Quotient rule)
- $(f(x) + g(x))' = f'(x) + g'(x)$ (Sum rule)
- $(f(g(x)))' = (f \circ g)'(x) = f'(g(x)) \cdot g'(x)$ (Chain rule)

The notions of derivatives have a natural generalization to higher dimensional cases. In particular, we will start by introducing the notion of a gradient.

Definition 3. (Gradient of f) *The gradient of a multivariate function $f : \mathbb{R}^p \to \mathbb{R}$ is*

$$\nabla f(x) = \begin{bmatrix} \frac{\partial f}{\partial x_1} \\ \vdots \\ \frac{\partial f}{\partial x_p} \end{bmatrix} \in \mathbb{R}^p$$

where

$$\begin{aligned} \frac{\partial f}{\partial x_i} &= \lim_{\epsilon \to 0} \frac{f(\dots, x_{i-1}, x_i + \epsilon, x_{i+1}, \dots) - f(\dots, x_{i-1}, x_i, x_{i+1}, \dots)}{\epsilon} \\ &= \frac{f(x + \epsilon e_i) - f(x)}{\epsilon} \end{aligned}$$

The following definition computes the rate at which a function f changes at a point x, in the direction of an arbitrary vector y. This relates linear forms of the gradient (i.e. inner product) to one-dimensional derivative, evaluated at zero.

Definition 4. (First-order Directional Derivative) *Let $f : \mathbb{R}^p \to \mathbb{R}$ be a differentiable function. For two points $x, y \in \mathbb{R}^p$ and for scalar γ, we have:*

$$\nabla_y f(x) = \nabla f(x)^\top y = \lim_{\gamma \to 0} \frac{f(x+\gamma y) - f(x)}{\gamma}$$

$\nabla_y f(x)$ is called the directional derivative of f at x in the direction of y.

To verify this formula, let us define first the "helper" function:

$$\varphi(\gamma) := f(x + \gamma y) = f(\psi(\gamma)),$$

where $\psi(\gamma) := x + \gamma y$. Computing the gradient of $\varphi(\gamma)$ wrt γ is equivalent to computing the *gradient of f along the direction y*, for infinitesimal γ. In particular, by applying the chain rule, we obtain:

$$\begin{aligned}\varphi'(\gamma) &= \sum_{i=1}^{p} \frac{\partial f(\psi(\gamma))}{\partial \psi_i} \cdot \nabla \psi_i(\gamma) \\ &= \sum_{i=1}^{p} \frac{\partial f(\psi(\gamma))}{\partial \psi_i} \cdot y_i \\ &= \langle \nabla f(\psi(\gamma)), y\rangle \\ &= \langle \nabla f(x+\gamma y), y\rangle\end{aligned}$$

Then, we obtain the definition of the directional derivative when we set $\gamma = 0$.

The directional derivative is also often written in the notation:

$$\nabla_y f(x) = y_1 \cdot \frac{\partial f}{\partial x_1} + y_2 \cdot \frac{\partial f}{\partial x_2} + \cdots + y_p \cdot \frac{\partial f}{\partial x_p} = \sum_{i=1}^{p} y_i \cdot \frac{\partial f}{\partial x_i}$$

Next, we will define derivative for a multivariate vector function.

Definition 5. (Jacobian of a function f) *The Jacobian of a multivariate vector function $f : \mathbb{R}^p \to \mathbb{R}^m$ is given by:*

$$Df(x) = \begin{bmatrix} \frac{\partial f_1}{\partial x_1} & \frac{\partial f_1}{\partial x_2} & \cdots & \frac{\partial f_1}{\partial x_p} \\ \vdots & \vdots & & \vdots \\ \frac{\partial f_m}{\partial x_1} & \frac{\partial f_m}{\partial x_2} & \cdots & \frac{\partial f_m}{\partial x_p} \end{bmatrix} \in \mathbb{R}^{m\times p}$$

Loosely speaking, taking the Jacobian of the gradient yields the Hessian which contains the second-order local information about f:

Definition 6. (Hessian matrix of f) *The Hessian of a multivariate function $f : \mathbb{R}^p \to \mathbb{R}$ is*

$$\nabla^2 f(x) = \begin{bmatrix} \frac{\partial^2 f}{\partial x_1^2} & \frac{\partial^2 f}{\partial x_1 \partial x_2} & \cdots & \frac{\partial^2 f}{\partial x_1 \partial x_p} \\ \vdots & \vdots & & \vdots \\ \frac{\partial^2 f}{\partial x_1 \partial x_p} & \frac{\partial^2 f}{\partial x_2 \partial x_p} & \cdots & \frac{\partial^2 f}{\partial x_p^2} \end{bmatrix}$$

The Hessian matrix of a continuous function is symmetric. The Hessian matrix provides information about *the curvature* of the function f. For example, given a point $x^\star$, when $\nabla^2 f(x^\star) \succ 0$ holds, then $x^\star$ is (at least) a strict local minimizer of f. Alternatively, when $\nabla^2 f(x^\star) \prec 0$, then $x^\star$ is a strict local maximizer of f. See figure 7 for a geometric interpretation of the facts stated above.

Similarly to gradients, we can relate quadratic forms of the Hessian matrix to one-dimensional derivatives.

Definition 7. (Second-order Directional Derivative) *Let $f : \mathbb{R}^p \to \mathbb{R}$ be a twice-differentiable function. Let $x, y \in \mathbb{R}^p$ and γ a scalar. Then:*

$$\langle \nabla^2 f(x+\gamma y) \cdot y,\ y\rangle = \lim_{\gamma \to 0} \frac{\nabla f(x+\gamma y)^\top y - \nabla f(x)^\top y}{\gamma} = \frac{\partial^2 f(x+\gamma y)}{\partial \gamma^2}.$$

Taylor expansion of a function f. Now that we have an idea of what derivates, gradients and Hessians are, how can we use them in practice? The answer to this question will come from answering the following question: *Are there any intuitive ways of approximating the behavior of a function, even locally?* The answer is Yes: the *Taylor expansion* of the function may be used to approximate the function locally.

Definition 8. (Taylor Series) *Assuming that f is n-times differentiable, then the Taylor series of f centered at x_0 is*

$$\begin{aligned}T_\alpha(x) &= \sum_{k=0}^{\infty} \frac{f^{(k)}(\alpha)(x-\alpha)^k}{k!} \\ &= \frac{f(\alpha)}{0!} + \frac{f'(\alpha)}{1!}(x-\alpha) + \frac{f''(\alpha)}{2!}(x-\alpha)^2 + \cdots\end{aligned}$$

The k-th order Taylor approximation is the above series truncated at the k^{th} term in the sum.

Here, f is assumed to be differentiable as many times as we would like. In general, for the rest of this course we will assume that our functions are differentiable, unless stated otherwise. Most often than not, we will focus on the up-to-2nd order Taylor approximation of functions. We note that the Taylor expansion gives a good (local) estimate of the function. When we keep only the first two terms, we call it a linear approximation of the function near α, as is illustrated in figure 5.

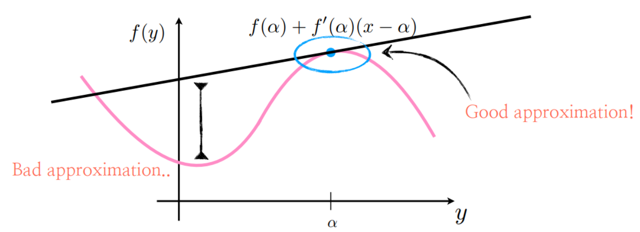


Fig. 5. The first-order Taylor expansion provides a good estimation of the function near the point α, but easily drifts away when we move a little bit away from it.

When we keep the first three terms, we obtain a quadratic approximation of f, as is illustrated in figure 6.

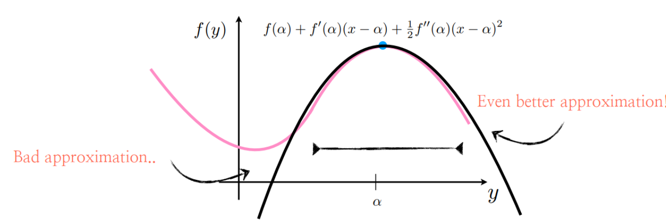


Fig. 6. The second-order Taylor expansion estimates a function better near point α.

Adding more terms provides a more accurate approximation, and for a univariate function, this is attainable. However, the complexity increases significantly in high-order Taylor expansion of multivariate functions.

Definition 9. *The Taylor expansion of a multivariate function $f : \mathbb{R}^p \to \mathbb{R}$ at point $\alpha \in \mathbb{R}^p$ is*

$$f(x) \approx f(\alpha) + \langle \nabla f(\alpha), x-\alpha\rangle + \frac{1}{2}\langle \nabla^2 f(\alpha)(x-\alpha), (x-\alpha)\rangle + \ldots$$

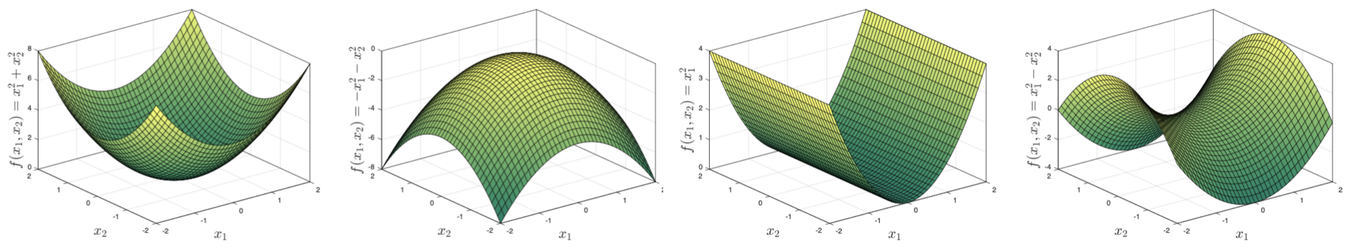
This is a natural generalization of the one dimensional version. For a first-order Taylor expansion approximation, we obtain:

$$f(x) \approx f(\alpha) + \langle \nabla f(\alpha), x-\alpha\rangle, \quad \alpha \in \mathbb{R}^p,$$

while for a second-order one, we obtain:

$$f(x) \approx f(\alpha) + \langle \nabla f(\alpha), x-\alpha\rangle + \frac{1}{2}\langle \nabla^2 f(\alpha)(x-\alpha),\ x-\alpha\rangle, \quad \alpha \in \mathbb{R}^p.$$

For our further discussions, the following *fundamental theorem of calculus* (part II) is useful: it will help showing that the differentiation in the multivariate setting can be expressed as integrals of univariate functions. The fundamental theorem reads as follows:



$\nabla^2 f(\cdot) \succ 0$ $\quad$ $\nabla^2 f(\cdot) \prec 0$ $\quad$ $\nabla^2 f(\cdot) \succeq 0$ $\quad$ Indefinite

Fig. 7. How Hessian looks like around interesting points of a two-dimensional function f (z-axis).

Definition 10. (Fundamental theorem of calculus, part II) *Let $f : [\alpha, \beta] \to \mathbb{R}$ be a continuously differentiable function. Then:*

$$\int_\alpha^\beta \tfrac{d}{dt} f(t) dt = f(\beta) - f(\alpha).$$

Based on the above, Taylor's expansion implies the following:

Lemma 1. *Let $f : \mathbb{R}^p \to \mathbb{R}$ be a differentiable function. Let two points $x, y \in \mathbb{R}^p$. Then:*

$$f(y) = f(x) + \langle \nabla f(x),\ y - x\rangle + \int_0^1 (1-\gamma) \tfrac{\partial^2 f(x+\gamma(y-x))}{\partial \gamma^2} d\gamma$$

The above provide an idea of a local approximation of a function. This leads to the Taylor's theorem, which is often called the multivariate mean-value theorem. Taylor's theorem below allows the approximation of smooth functions by simple polynomials.

Theorem 1. (Taylor's theorem)

- *If f is continuously differentiable, then:*
  $f(w) = f(w_0) + \langle \nabla f(tw + (1-t)w_0), \rangle, \quad$ *for some $t \in [0, 1]$.*
- *If f is twice differentiable, then:*
  $$\nabla f(w) = \nabla f(w_0) + \int_0^1 \nabla^2 f(tw + (1-t)w_0) \cdot (w - w_0) dt.$$
- *Further, if f is twice differentiable, then, for some $t \in [0, 1]$:*

$$\begin{aligned} f(w) =& f(w_0) + \langle \nabla f(w_0), w - w_0\rangle \\ &+ \tfrac{1}{2} \left\langle \nabla^2 f(tw + (1-t)w_0) \cdot (w - w_0), w - w_0 \right\rangle. \end{aligned}$$

But how the above are useful in optimization?. Consider that some one gives you the following problem $\min_x f(x)$ for some function f. Further, we are told that computing gradients $\nabla f(\cdot)$ and Hessians $\nabla^2 f(\cdot)$ is fairly easy. Then, assuming we start from a point x_0, instead of worrying about f itself, one can do the following steps:

- Compute gradient $\nabla f(x_0)$; name this as the h vector.
- Compute Hessian $\nabla^2 f(x_0)$; name this as the H matrix.
- Form the second-order Taylor approximation: $f(x) \approx f(x_0) + \langle \nabla f(x_0), x - x_0\rangle + \frac{1}{2}\langle \nabla^2 f(x_0) \cdot (x - x_0), x - x_0\rangle = f(x_0) + h^\top (x - x_0) + \frac{1}{2}(x - x_0)^\top H (x - x_0)$.

Hence, instead of optimizing directly $\min_x f(x)$, we first compute the second-order approximation around a point x_0:

$$\min_x \left\{ f(x_0) + \langle \nabla f(x_0),\ x - x_0\rangle + \tfrac{1}{2}\langle \nabla^2 f(x_0)(x - x_0),\ x - x_0\rangle \right\},$$

which in turn is just a minimization of a quadratic function:

$$\min_x \left\{ h^\top x + \tfrac{1}{2} x^\top H x \right\}.$$

Solving quadratic problems is a type of optimization that we know how to compute efficiently.

The above list suggests that *regardless how difficult f is to optimize, one can approximate it through Taylor's expansion to get to a problem that we can easily solve: that of a quadratic objective!* Of course, this does not guarantee by itself that we will get the optimum of f: e.g., if x_0 is not close to the optimal $x^\star$ and the local quadratic approximation does not follow well f, then we have no hope in optimizing the original f function. However, we can make this happen by using *iterative procedures* over the above motions for ever-improved x points.

Optima. It is never hard to spot the minimum of a function, whenever it can be drawn on a paper. For a computer though, this is a complicated problem akin to a grid search. Unfortunately, real problems are usually multidimensional, so we cannot draw the functions involved on a paper. Furthermore, a direct search on a multidimensional grid is computational prohibitive (the so-called “curse of dimensionality” issue). Consequently, we have no idea of the global shape of the function. We rely on the limited local information to search for the minimum. We want to call this as *agnostic optimization*. See Figure 8 and its solution in Figure 9.

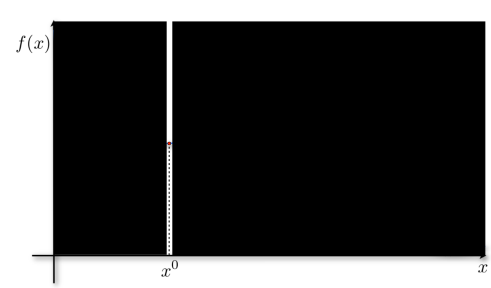


Fig. 8. Agnostic optimization. Given x_0 and $f(x_0)$ as a starting point, this is how the landscape looks like for a computer program: there is no clear path to move from x_0 to a point with a better objective value.

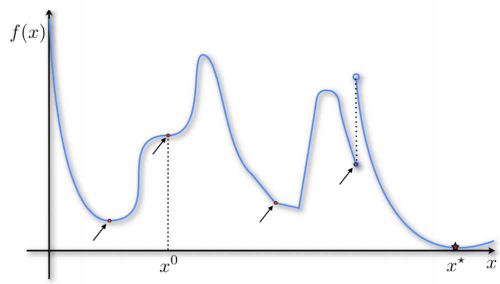


Fig. 9. However, the whole picture is unpredictable. Is it a minimum?

To refer to the optima of a function, we use a set of notations. Without loss of generality, we only discuss minimization.

Definition 11. *The global minimizer $x^\star$ of a function f satisfies*

$$f(x^\star) \leq f(x), \quad \forall x \text{ in the domain of } f.$$

Definition 12. *A local minimizer $\hat{x}$ of a function f satisfies*

$$f(\hat{x}) \leq f(x), \quad \forall x \in \mathcal{N}_{\hat{x}},$$

where $\mathcal{N}_{\hat{x}}$ defines a very small neighborhood around $\hat{x}$.

We can recognize that a solution is local minimum by the following *necessary* conditions:

- 1st order optimality condition: $\nabla f(\hat{x}) = 0$.
- 2nd order optimality condition: $\nabla f(\hat{x}) = 0$ and $\nabla^2 f(\hat{x}) \succeq 0$.

Intuitively, the above state that *i*) the function is flat at the point of the minimum, and *ii*) the function looks like a "bowl" at this point, when both conditions are satisfied. The last point relates to the notion of convexity: *this will be defined later on in the class.*

Note that these are only necessary conditions, with $f(x) = x^3$ as a simple counterexample at point $x = 0$, which satisfies the two conditions but is not a local minimum.

Lipschitz Conditions

In figures 8 and 9, there are different points where the gradient is zero. At some of these points the gradient is not unique, and at some points the function is discontinuous. In such a general case, finding the global minima seems to be a difficult task, unless we start making some assumptions about the objective f. Often, many of the objectives f we want to optimize in practice satisfy a form of *Lipschitz continuity.*

Definition 13. *A function f is called Lipschitz continuous, when*

$$|f(x) - f(y)| \leq M \cdot \|x - y\|_2, \quad \forall x, y,$$

for some constant $M > 0$.

This basically means a function should not be too steep, where the steepness is controlled by the constant M. Lipschitz continuity is a stronger condition than uniform continuity and continuity, and weaker one than convexity. Basically, a Lipschitz continuous function may not have abrupt changes.

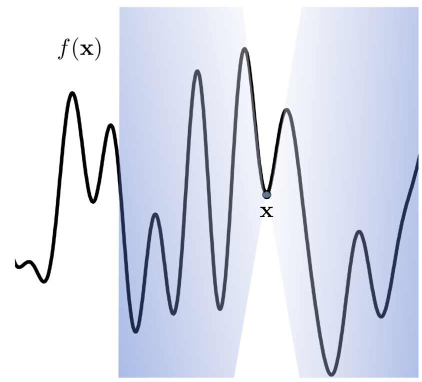


Fig. 10. Illustration of a Lipschitz continuous function, where the width of the cone in white is controlled by M. In a way, Lipschitz continuity states that a function cannot abruptly change so that it will not "appear" inside the white cone in the picture above.

A seemingly similar but quite different assumption is that of Lipschitz gradient continuity, where we apply the Lipschitz condition on the gradients of the function.

Definition 14. *A function f has Lipschitz continuous gradients, when*

$$\|\nabla f(x) - \nabla f(y)\|_2 \leq L\|x - y\|_2, \quad \forall x, y,$$

where $L > 0$ is a constant scalar. Often, such a function is also called L-smooth.

Such a condition forbids sudden changes in the gradient. Using Taylor's expansion, we can prove that

$$f(y) \leq f(x) + \langle \nabla f(x), y - x \rangle + \tfrac{L}{2}\|x - y\|_2^2,$$

which means the function is upper-bounded by a quadratic function (there is also a lower quadratic bound). There are several equivalent characterizations of *Lipschitz gradient continuity* to be aware of:

$$f(y) \leq f(x) + \langle \nabla f(x), y - x \rangle + \tfrac{L}{2}\|x - y\|_2^2,$$
$$\nabla^2 f(x) \preceq L \cdot I, \text{ where } I \text{ is the identity matrix}$$
$$\|\nabla^2 f(x)\|_2 \leq L.$$

Comparison of Lipschitz conditions:

- Lipschitz continuity implies that f should not be too steep.
- Lipschitz gradient continuity implies that changes in the slope of f should not happen suddenly.

Example: Linear regression. In linear regression, the objective $f(x) = \frac{1}{2}\|Ax - b\|_2^2$ is not Lipschitz continuous—it gets arbitrarily steep when approaching infinity in x—however, it is Lipschitz gradient continuous as

$$\begin{aligned}\|\nabla f(x) - \nabla f(y)\|_2 &= \|A^\top(Ax - b) - A^\top(Ay - b)\|_2 \\ &\leq \|A^\top A\|_2 \cdot \|x - y\|_2,\end{aligned}$$

where $\|A^\top A\|_2$, the largest singular value, serves as the parameter L. This also justifies the equivalent condition:

$$\nabla^2 f(x) \preceq L \cdot I.$$

But how can we use the Lipschitz gradient continuity in optimization?

A key product of its definition is the inequality:

$$f(y) \leq f(\alpha) + \langle \nabla f(\alpha), y - \alpha \rangle + \tfrac{L}{2}\|y - \alpha\|_2^2.$$

Therefore, at a chosen point α, we can upper bound the curve of f (for any y) with a quadratic function, evaluated around α. For a one-dimensional simple example, one can depict this as in figure 11.

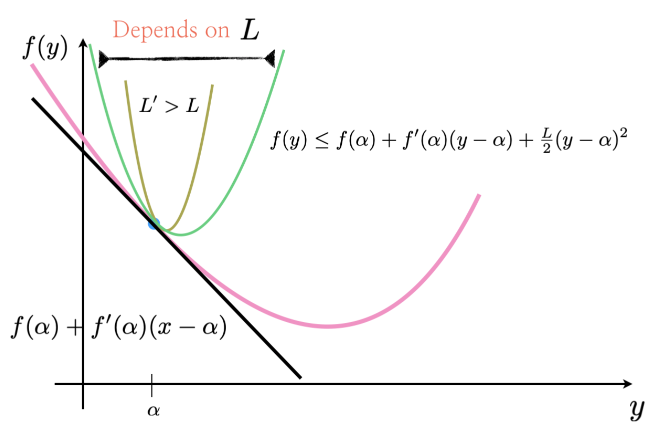


Fig. 11. Illustration of how Lipschitz gradient continuity has algorithmic implications. Here, we want to minimize the one-dimensional $f(y)$ (pink curve). Instead of minimizing f directly—in fact it could be a very complicated function to directly minimize—we will successively construct quadratic (upper-bound) approximations around the current putative solutions, and minimize those approximations. In the figure, we are at point $f(\alpha)$; one can construct the linear local approximation of f around α (black curve); Lipschitz gradient continuity goes further and introduces a quadratic term, "weighted" by the Lipschitz gradient continuity constant L (green curve). Minimizing this quadratic approximation will provide a new point, around which we can form another quadratic approximation, etc. Key observation regarding L is that the larger L is, the steepest the quadratic approximation around the current point is (compare green with khaki curves). The steepest these quadratic approximations are, the smaller the learning rate/step size in algorithms need to be in order to guarantee provable performance; more details later on.

Lipschitz gradient continuity expression $f(y) \leq f(x) + \langle \nabla f(x), y - x \rangle + \frac{L}{2}\|x - y\|_2^2$ can also be proved via Taylor's expansion + other properties of Lipschitz gradient continuous functions. We know from Taylor's expansion that:

$$f(y) = f(x) + \langle \nabla f(x), y - x \rangle + \tfrac{1}{2}\langle \nabla^2 f(z)(y - x), y - x \rangle,$$

for some z. Knowing that for a Lipschitz gradient continuous function we have:

$$\nabla^2 f(x) \preceq L \cdot I \Rightarrow \|\nabla^2 f(x)\|_2 \leq \|L \cdot I\|_2 \Rightarrow \|\nabla^2 f(x)\|_2 \leq L.$$

Then,

$$\begin{aligned} \tfrac{1}{2}\langle \nabla^2 f(z)(y - x), y - x \rangle &\leq \tfrac{1}{2}\left\|\nabla^2 f(z)(y - x)\right\|_2 \cdot \|y - x\|_2 \\ &\leq \tfrac{1}{2}\left\|\nabla^2 f(z)\right\|_2 \cdot \|y - x\|_2^2 \\ &\leq \tfrac{L}{2}\|y - x\|_2^2. \end{aligned}$$

Combining this with the initial Taylor's expansion expression, we get:

$$f(y) \leq f(x) + \langle \nabla f(x), y - x \rangle + \tfrac{L}{2}\|y - x\|_2^2.$$

Gradient Descent for Lipschitz continuous gradient f

With Lipschitz gradient continuity, we can establish the convergence of an iterative optimization method, such as gradient descent. In fact, gradient descent can be derived as the method of successively minimizing the quadratic approximations around the current point.

Let us elaborate a bit more before we present gradient descent as the basic algorithm for smooth optimization. Let $\min_{x \in \mathbb{R}^p} f(x)$ be the problem we are interested to solve. We assume that f is differentiable, and we can approximate it by Taylor's expansion as:

$$f(x + \delta) = f(x) + \langle \nabla f(x), \delta \rangle + o(\|\delta\|_2).$$

Minimizing f locally, a good direction δ is such that the quantity $\langle \nabla f(x), \delta \rangle$ is as small as possible. Given that for now, we are interested in finding a good direction (and not how far in that direction to move to), it is easy to see that a good *normalized* direction is:

$$\delta = -\tfrac{\nabla f(x)}{\|\nabla f(x)\|_2}$$

or a direction with controllable step:

$$\delta = -\eta \nabla f(x).$$

Given the above, gradient descent is defined as follows:

Definition 15. *Let f be a differentiable objective with gradient $\nabla f(\cdot)$. The gradient descent method optimize f iteratively, as in:*

$$x_{t+1} = x_t - \eta_t \nabla f(x_t), \quad t = 0, 1, \dots,$$

where x_t is the current estimate, and η_t is the step size or learning rate.

The idea behind gradient descent is simple: given the current point x_t, we can compute the negative gradient $-\nabla f(x_t)$ as direction that f has the steepest slope (locally). Following that direction, we carefully select η_t, the step size, to dictate how far in that direction we will move.

While gradient descent is quite simple, there are three actions needed in order to make it work in practice: i) how to choose step size η_t, ii) initial point x_0, and iii) when to terminate the algorithm.

—Step size: several approaches are known, some more practical than others, including:

- i) $\eta_t = \eta$; i.e., the step size is fixed to a value by the user, and stays fixed for all the iterations;
- ii) $\eta_t = O\left(\frac{c}{t}\right)$ or $\eta_t = O\left(\frac{c}{\sqrt{t}}\right)$ for a constant $c > 0$; i.e., the step size keeps decreasing as we go on with the iterations. It starts aggressively (e.g., for $t = 1$ it can be c), but very fast decreases;
- iii) $\eta_t = \text{argmin}_\eta f(x_t - \eta \nabla f(x_t))$; i.e., find the step size that minimizes our objective along the direction of gradient descet. This approach makes (computationally) sense only for a narrow set of problems, where solving the above problem has i) a closed-form solution, and ii) it is not difficult to compute that closed-form solution. In the majority of cases, finding the best η_t is computationally prohibited to perform per iteration, and often it requires the same effort as finding the solution to the original problem.
- iv) Fixed step size procedures, such as the Goldstein-Armijo rule; these are out of the scope of this course, and they are often used in classical numerical analysis.

—Initial value x_0: because we know little about the function, we usually start from points that make sense (e.g., unless the data involved in the function definition have abruptly large or small values, starting from $x_0 = 0$ makes sense for some prroblems) or we just pick a random value. How to initialize is (almost) irrelevant for some classes of problems (e.g., convex optimization), but it is extremely important for broader class of problems. By important we mean that carefully selecting the starting point either leads to some theory—but in practice several starting points lead to the same performance—or that is required to get good performance in practice.

—Termination criterion: there are various standard criterions, like "killing" the execution after T iterations (irrespective of whether we converged or not), checking how much progress we make per iteration through $\|x_{t+1} - x_t\|_2$ or $f(x_{t+1}) - f(x_t)$, or by checking if the norm of the gradient is below a threshold, $\|\nabla f(x_t)\|_2 \leq \varepsilon$.

Performance of gradient descent under smoothness assumptions.

Claim 1. *Assume that i) f is differentiable, and ii) that f has L-Lipschitz continuous gradients. Consider the gradient descent iterate: $x_{t+1} = x_t - \eta_t \nabla f(x_t)$. Then:*

$$f(x_{t+1}) \leq f(x_t) - \eta_t \left(1 - \tfrac{\eta_t L}{2}\right) \cdot \|\nabla f(x_t)\|_2^2.$$

Proof: By using the assumption of Lipschitz gradients, we have:

$$\begin{aligned}
f(x_{t+1}) &\leq f(x_t) + \langle \nabla f(x_t), x_{t+1} - x_t \rangle + \tfrac{L}{2}\|x_{t+1} - x_t\|_2^2 \\
&= f(x_t) + \langle \nabla f(x_t), x_t - \eta_t \nabla f(x_t) - x_t \rangle \\
&\quad + \tfrac{L}{2}\|x_t - \eta_t \nabla f(x_t) - x_t\|_2^2 \\
&= f(x_t) - \eta_t \|\nabla f(x_t)\|_2^2 + \tfrac{\eta_t^2 L}{2}\|\nabla f(x_t)\|_2^2 \\
&= f(x_t) - \eta_t \left(1 - \tfrac{\eta_t L}{2}\right)\|\nabla f(x_t)\|_2^2
\end{aligned}$$

□

The above result indicates that, *i)* as long as $\eta_t \left(1 - \frac{\eta_t L}{2}\right)$ is positive, by performing gradient descent steps, we decrease the objective value by a non-positive quantity $-\eta_t \left(1 - \frac{\eta_t L}{2}\right)\|\nabla f(x_t)\|_2^2$; *ii)* we can maximize the decrease by maximizing the quantity $\eta_t \left(1 - \frac{\eta_t L}{2}\right)$.

Define $g(\eta) := \eta \left(1 - \frac{\eta L}{2}\right)$. Knowing that $\eta > 0$, we first require $1 - \frac{\eta L}{2} > 0 \Rightarrow \eta < \frac{2}{L}$. Thus, for $0 < \eta < \frac{2}{L}$, we observe that the $g(\eta)$ is maximized when we require the gradient satisfies:

$$g'(\eta) = 0 \quad \Rightarrow \quad 1 - \eta L = 0 \quad \Rightarrow \quad \eta = \tfrac{1}{L}.$$

For the rest of our theory, we will use $\eta_t = \eta = \frac{1}{L}$. Observe that this step size requires the knowledge of L; for some objectives this is easy to find, e.g. linear regression, logistic regression, but for others it is not.

Claim 2. *Gradient descent $x_{t+1} = x_t - \eta_t \nabla f(x_t)$, with $\eta_t = \eta = \frac{1}{L}$, satisfies:*

$$f(x_{t+1}) \leq f(x_t) - \tfrac{1}{2L}\|\nabla f(x_t)\|_2^2$$

Proof: This is true by substituting $\eta_t = \frac{1}{L}$ in the result of claim 1. □

The above characterize the drop in function values at the t-th iteration. The idea of convergence is based on the idea of relaxation.

Definition 16. *A sequence of real numbers $\{\alpha_t\}_{t=0}^{\infty}$ is called a relaxation sequence if $\alpha_{t+1} \leq \alpha_t, \;\; t \geq 0$.*

Combining all the iterations together, for T iterations, we have:

$$\begin{aligned}
f(x_{T+1}) &\leq f(x_T) - \tfrac{1}{2L}\|\nabla f(x_T)\|_2^2 \\
f(x_T) &\leq f(x_{T-1}) - \tfrac{1}{2L}\|\nabla f(x_{T-1})\|_2^2 \\
&\;\vdots \\
f(x_1) &\leq f(x_0) - \tfrac{1}{2L}\|\nabla f(x_0)\|_2^2
\end{aligned}$$

Summing all these inequalities, and under the observation that $f(x^\star) \leq f(x_{T+1})$, we get the following claim.

Claim 3. *Over T iterations, gradient descent generates a sequence of points $x_1, x_2, \ldots,$ such that:*

$$\tfrac{1}{2L}\sum_{t=0}^{T}\|\nabla f(x_t)\|_2^2 \leq f(x_0) - f(x^\star).$$

First, observe that the right hand side is a constant quantity, as it does not depend on the number of iterations. Subsequently, the above result implies that, even if we continue running gradient descent for many iterations, the sum of gradient norms is always bounded by a constant. This indicates that the gradient norms that we add at the very end of the execution have to be small, which further implies convergence to a stationary point (also known as critical point: a point that has gradient zero, implying that it could be a local minima).

However, the above say nothing about the convergence rate. For that we have the following claim.

Claim 4. *Assume we run gradient descent for T iterations, and we obtain T gradients, $\nabla f(x_t)$, for $t \in \{0, \ldots, T\}$. Then,*

$$\min_{t \in \{0,\ldots,T\}} \|\nabla f(x_t)\|_2 \leq \sqrt{\tfrac{2L}{T+1}}\left(f(x_0) - f(x^\star)\right)^{\frac{1}{2}} = O\left(\tfrac{1}{\sqrt{T}}\right).$$

Proof: We know that

$$(T+1)\cdot \min_t \|\nabla f(x_t)\|_2^2 \leq \sum_{t=0}^{T}\|\nabla f(x_t)\|_2^2.$$

Then,

$$\begin{aligned}
\tfrac{T+1}{2L}\cdot \min_t \|\nabla f(x_t)\|_2^2 &\leq \tfrac{1}{2L}\sum_{t=0}^{\top}\|\nabla f(x_t)\|_2^2 \leq f(x_0) - f(x^\star) \Rightarrow \\
\min_t \|\nabla f(x_t)\|_2^2 &\leq \tfrac{2L}{T+1}\cdot\left(f(x_0) - f(x^\star)\right) \\
\min_t \|\nabla f(x_t)\|_2 &\leq \sqrt{\tfrac{2L}{T+1}}\cdot\left(f(x_0) - f(x^\star)\right)^{\frac{1}{2}} \\
&= O\left(\tfrac{1}{\sqrt{T}}\right).
\end{aligned}$$

□

This is called *sublinear convergence rate*. Just to provide a perspective of what it means, focus on Figure 12. In general, this is a rather pessimistic result. However, have in mind that we made no assumptions other than differentiability of f, and f being a L-smooth function. We will see that making more assumptions helps improving the convergence radically.

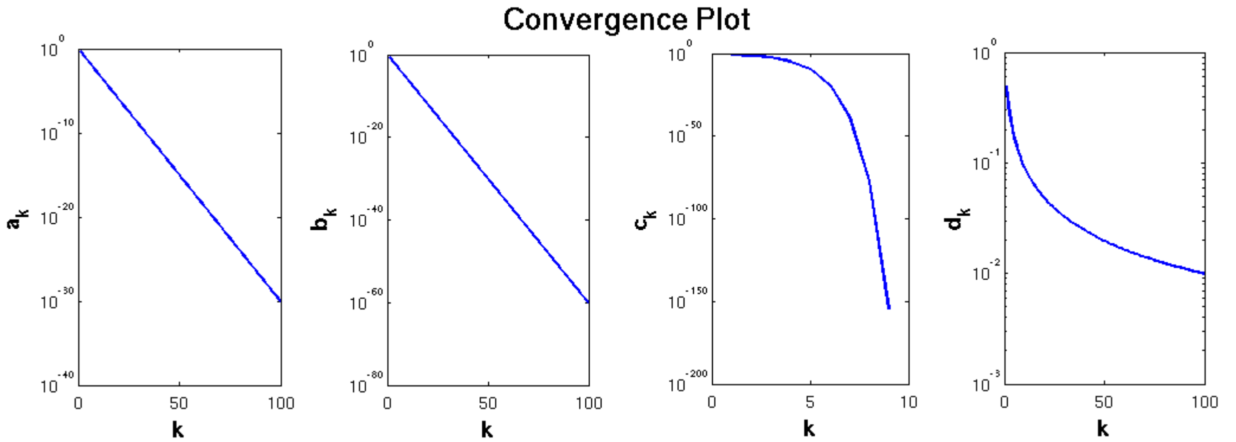


Fig. 12. Borrowed from Wikipedia. Illustration of different convergence rates. Note that y-axis is in logarithmic scale for all the plots, while the x-axis has a linear scale. The y-axis denotes a metric that dictates to the optimum point; think for example $\|x_k - x^\star\|_2$. The x-axis represents the iteration count k. The first two plots represent *linear* convergence rates: it is called linear as a convention to match the linear curve in the *logarithmic* y-axis scale. While the second plot depicts a more preferable behavior, in the big-Oh notation, the two plots are equivalent. For an error level ε, linear convergence rate implies $O\left(\log \frac{1}{\varepsilon}\right)$. The third plot depicts a *quadratic* convergence rate. For an error level ε, linear convergence rate implies $O\left(\log\log \frac{1}{\varepsilon}\right)$. Finally, the fourth plot represents the *sublinear* convergence rate; much slower than the linear rate. Some typical rates are: $O\left(1/\varepsilon^2\right)$, $O\left(1/\varepsilon\right)$, $O\left(1/\sqrt{\varepsilon}\right)$.

Side note on convergence rates. There are two notations for convergence rate, one using an error level ε that our stopping criterion is based on, and the other using the number of iterations T. For the moment, we know that gradient descent has a convergence rate, with respect to the norm of the gradients, $O\left(1/\sqrt{T}\right)$. Pick a small ε, and assume we require $\min_t \|\nabla f(x_t)\|_2 \leq \varepsilon$. This translates into:

$$\sqrt{\tfrac{2L}{T+1}} \cdot (f(x_0) - f(x^\star))^{\frac{1}{2}} \leq \varepsilon \Rightarrow$$
$$T + 1 \geq \tfrac{2L}{\varepsilon^2} \cdot (f(x_0) - f(x^\star)) \Rightarrow$$
$$T \geq \left\lceil \tfrac{2L}{\varepsilon^2} \cdot (f(x_0) - f(x^\star)) - 1 \right\rceil$$

Usually, in order for our convergence rates to make sense, we pick a small value for ε, e.g. let $\varepsilon = 10^{-3}$. Our result dictates that in order to get a solution with $\min_t \|\nabla f(x_t)\|_2 \leq 10^{-3}$, we will need approximately $O\left(1/\varepsilon^2\right) = O\left(10^6\right)$ iterations (hiding all other constants). This is the meaning of a sublinear convergence rate: in order to get ε accuracy in some sense, we require $1/\varepsilon^2$ iterations. In this course, we will discuss how to achieve better sublinear rates, or even better rates than linear.

Example: Logistic regression. We already discussed the case of linear regression, where the objective $f(x) = \frac{1}{2}\|Ax - b\|^2$ has Lipschitz continuous gradients, with constant $L := \|A^\top A\|_2$. Here, we consider another famous—and less straightforward—objective: that of logistic regression. We know that logistic regression is based on the following premise for binary classification:

Given a sample feature vector $\alpha_i \in \mathbb{R}^p$ and a binary class $y_i \in \{\pm 1\}$, define the conditional probability of y_i given α_i as:

$$\mathbb{P}\left[y_i \mid \alpha_i, x^\star\right] \propto \frac{1}{1 + \exp(-y_i \alpha_i^\top x^\star)}.$$

The above generative assumption leads to the following objective:

$$\min_{x \in \mathbb{R}^p} \left\{ f(x) = \frac{1}{n} \sum_{i=1}^n \log(1 + \exp(-y_i \alpha_i^\top x)) \right\}.$$

Following the same recipe with linear regression, one can compute the gradient and Hessian as

$$\begin{aligned}\nabla f(x) &= \frac{1}{n}\sum_{i=1}^n \nabla \left[\log(1 + \exp(-y_i\alpha_i^\top x)\right] \\ &= \frac{1}{n}\sum_{i=1}^n \tfrac{1}{1+\exp(-y_i\alpha_i^\top x)} \cdot \nabla_x \left[\exp(-y_i\alpha_i^\top x)\right] \\ &= \frac{1}{n}\sum_{i=1}^n \tfrac{\exp(-y_i\alpha_i^\top x)}{1+\exp(-y_i\alpha_i^\top x)} \cdot \nabla_x \left[-y_i\alpha_i^\top x\right] \\ &= \frac{1}{n}\sum_{i=1}^n \frac{-y_i}{1+\exp(y_i\alpha_i^\top x)} \alpha_i^\top\end{aligned}$$

and

$$\begin{aligned}\nabla^2 f(x) &= \frac{1}{n}\sum_{i=1}^n \tfrac{y_i}{(1+\exp(y_i\alpha_i^\top x))^2} \cdot \nabla \left[1 + \exp(y_i\alpha_i^\top x)\right] \cdot \alpha_i^\top \\ &= \frac{1}{n}\sum_{i=1}^n \tfrac{y_i^2}{(1+\exp(y_i\alpha_i^\top x))^2} \cdot \exp(y_i\alpha_i^\top x) \cdot \alpha_i\alpha_i^\top \\ &= \frac{1}{n}\sum_{i=1}^n \underbrace{\tfrac{1}{(1+\exp(y_i\alpha_i^\top x))^2} \cdot \exp(y_i\alpha_i^\top x)}_{\text{scalar}} \cdot \underbrace{\alpha_i\alpha_i^\top}_{\in\mathbb{R}^{p\times p}}\end{aligned}$$

Observe that, for $\beta \in \mathbb{R}$,

$$\tfrac{1}{(1+\exp(\beta))^2} \cdot \exp(\beta) = \tfrac{1}{1+\exp(\beta)} \cdot \tfrac{\exp(\beta)}{1+\exp(\beta)} = \tfrac{1}{1+\exp(\beta)} \cdot \tfrac{1}{1+\exp(-\beta)}$$

Define $h(\beta) = \frac{1}{1+\exp(-\beta)}$, and observe that h maps to $(0, 1)$. Also observe that $h(-\beta) = 1 - h(\beta)$. Then, one can check that $h(\beta) \cdot h(-\beta) \leq \frac{1}{4}$.

Going back to our Hessian derivations:

$$\nabla^2 f(x) = \frac{1}{n}\sum_{i=1}^n h\left(y_i\alpha_i^\top x\right) \cdot h\left(-y_i\alpha_i^\top x\right) \cdot \alpha_i\alpha_i^\top.$$

Thus, taking spectral norm on both sides:

$$\|\nabla^2 f(x)\|_2 \leq \frac{1}{4n}\left\|\sum_{i=1}^n \alpha_i\alpha_i^\top\right\|_2 = \tfrac{1}{4n} \cdot \|A^\top A\|_2 := L.$$

where A accumulates all α_i's as rows.

Example: $f(x) = x^2 + 3\sin^2(x)$. This is a less practical example, but it is an example that does not satisfy some nice properties that linear regression and logistic regression satisfy. The objective looks like:

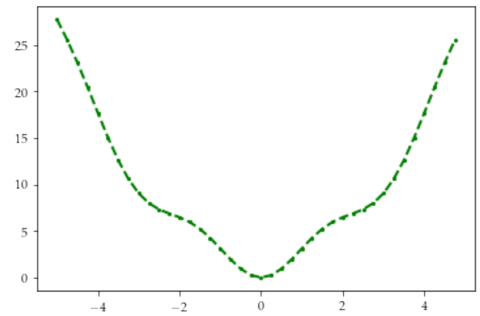


Fig. 13. $f(x) = x^2 + 3\sin^2(x)$

Let us compute the first and second derivatives of this function:

$$f'(x) = 2x + 6\sin(x) \cdot \cos(x)$$

and

$$f''(x) = 2 + 6\cos^2(x) - 6\sin^2(x)$$

Plotting the Hessian function, we obtain:

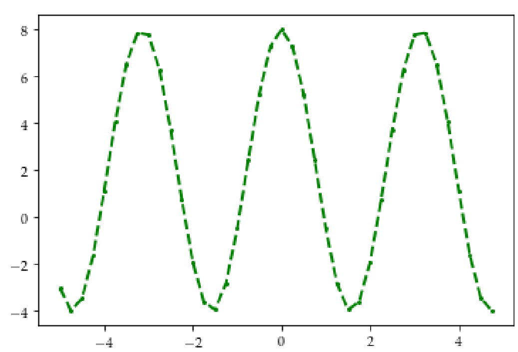


Fig. 14. $f''(x) = 2 + 6\cos^2(x) - 6\sin^2(x)$

By inspection (and based on the periodicity of the Hessian function), we can bound:

$$|f''(x)| \leq 8 := L.$$

———— ∞ ————

Deep learning, gradients and autodiff. It makes sense to open a parenthesis here and highlight the importance of (efficient) gradient calculation in modern machine learning applications. We will focus on the case of *deep learning and neural network training. (*For a deeper discussion on neural networks, there are excellent sources online; e.g., one could focus on the Deep Learning Book [11]).

Deep learning has advanced the state-of-the-art in computer vision [15–17], natural language processing [18–20] and speech recognition [21, 22]. At the time of writing this chapter, the transformer model [23] has revolutionized the NLP research, with machine translation [23], text classification [24], and image captioning [25]. Transformers are adopted for self-supervised pre-training and transfer learning, with the proposal of BERT [24]. Some popular science successes that use transformers or a variant of transformers include the GPT-3 [26], Megatron-LM [27] and T5 [28] language models, the DALLE-2 image synthesis model by OpenAI [29], and the AlphaFold2 protein-folding predictor by DeepMind [30].

Despite the impact of deep learning, the computational requirements for training such models are significant. Though pre-trained models are available online, there is need to train such models from scratch. Overall, training deep learning models is expensive (e.g., recent language models cost several million USD to train [31, 32]). For practitioners, even moderate-scale tasks can be prohibitive in time and cost due to hyperparameter tuning, which may lead to multiple iterations of model retraining, when the hyperparameter tuning process is taken into account, where it is typical to retrain models many times to achieve optimal performance.

At the core of this computational workload is the *gradient calculation*. In order to keep the discussion simple, consider the simplest version of a neural network: that of with fully connected (FC) layers, or otherwise described as *multi-layer perceptrons (MLPs)*. The mathematical description of a FC layer is as follows:

$$z_{i+1} = \sigma\left(W \cdot z_i + b\right)$$

where $z_i \in \mathbb{R}^{d_i}$ is the vector "representation" of the input at the i-th layer of a multilayer neural network, $W \in \mathbb{R}^{d_{i+1} \times d_i}$ is a trainable matrix that maps the input representation from d_i-dimensions to d_{i+1}-dimensions, and b is a bias vector (*for the rest of the discussion, we will assume that* $b = 0$ *for simplicity*). Finally, $\sigma : \mathbb{R}^{d_{i+1}} \to \mathbb{R}^{d_{i+1}}$ is a non-linear –often operating entrywise– activation function; some classical examples (either smooth or nonsmooth) are the ReLU, the sigmoid function, and the tanh function.[3]

In order to help visualize how a neural network would look like, consider the following:

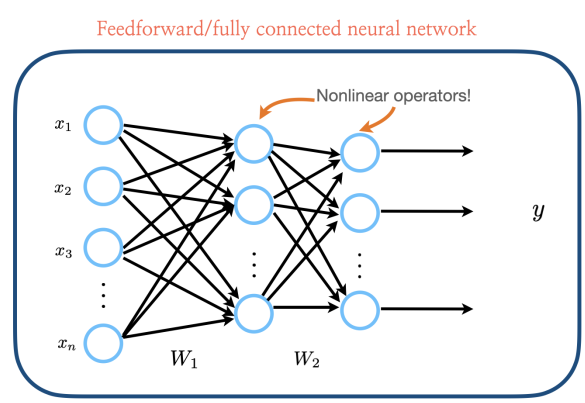


Fig. 15. 2-layer MLP

[3] In an actual neural network implementation, there might be additional layers or functions per layer that modify further the inputs at each layer (e.g., pooling layers, batch normalization layers, softmax layers), but it is out of the scope of this chapter to delve into these. At this stage, consider that a deep learning model is a *black box machine that transforms the input through a sequence of layers, each of which operates differently and based on the application at hand.*

We consider as input a vector $x = (x_1, x_2, \dots, x_n)$ (*you can think x_i as pixels of a flattened image*), and the task is to transform the input x such that the decision/output of the neural network y represents an "*answer to a question*"; e.g., the output often is an one-hot encoding where y is a probability distribution over a number of classes and the task is that of image classification. This particular toy model has two hidden layers, each one with trainable parameters W_i, represented as matrices of appropriate dimensions.

Let us go through this model and see how the input is transformed as we propagate "forward" from left to right:

- Given the input x, the first set of parameters generate the intermediate result $W_1 \cdot x$.
- Given this intermediate result, the neural network inserts that to the first set of neurons, with a nonlinear activation function, to get $z_1 = \sigma(W_1 \cdot x)$. This is the output of the first layer.
- Given z_1, we now get into the second layer: we first compute the intermediate result by applying the second set of trainable parameters, W_2: i.e., $W_2 \cdot z_1 = W_2 \cdot \sigma(W_1 \cdot x)$. *Observe the compositional formulation that a neural network takes: the input is being propagated through a series of layers that transform the initial representation in order to extract useful features.*
- Finally, this intermediate result also goes into the nonlinear activation functions of the second set of neurons to get $z_2 = \sigma(W_2 \cdot \sigma(W_1 \cdot x))$.
- (If we had more layers, this discussion would go on ...)

The goal in deep learning (and in machine learning in general) is to train these W_i tensors (here, 2-way tensors are matrices), such that the output of the model (i.e., here, the neural network) maps to the correct "labels": i.e., given a dataset $\mathcal{D}$ of images x and their corresponding correct labels $y^\star$, we want the outcome of our neural network, say $y = \text{MLP}(x)$, to be as close as possible to the ground truth $y^\star$ for all the images in the data source (which further implies we know, say, how to classify the input images).[4]

Mathematically, a way to measure the distance of the "learned" output y to the true labels $y^\star$ is by using a loss function. Here, for simplicity, we will use the ℓ_2-norm distance, and we will define the training problem as:

$$\min_{W_1, W_2} \sum_{(x, y^\star) \in \mathcal{D}} \|y^\star - y\|_2^2 = \sum_{(x, y^\star) \in \mathcal{D}} \|y^\star - \text{MLP}(x)\|_2^2$$

And, this is the optimization problem one needs to solve in order to train neural networks. I.e., if we abstract all the above (*and with the abuse of notation where we use x for the variables in this course*), the above is no different than a regular optimization problem:

$$\min_x f(x)$$

Gradient calculation in neural networks is no different than applying the chain rule of derivatives. *Automatic differentiation or autodiff* is the field that provides efficient algorithms to compute gradients of any function, especially if that is written as a composition of differentiable building blocks. E.g., in our discussion above, the building blocks are matrix-vector multiplications (e.g., $W_1 \cdot x$), nonlinear function applications, (e.g., $\sigma(W_1 \cdot x)$), etc. Autodiff is based on dynamic programming tools that choose wisely what quantities can be stored through the process and what is the optimal sequence of operations so that the gradient calculation is efficiently computed. This way, gradient calculations become an abstraction for practitioners (*that, these days, they do not need to worry about*), and this allows researchers to focus on the *modeling* part of neural networks, by defining other more informative/structured building blocks, based on the application: this has led to the creation of convolutional layers, residual layers, transformer layers, etc.

That being said, autodiff efficiently implements the *backpropagation* algorithm, which is nothing else but the calculation of the gradient of a composition of functions. In particular, given the forward output of a neural network, the backpropagation algorithm measures the discrepancy of the output with respect to the ground truth $y^\star$: i.e., $\|y^\star - y\|_2^2$. Based on this value, it is reasonable to infer the following rules:

- If the loss $\|y^\star - y\|_2^2$ is small, it means the neural network does not have to change much;
- If the loss $\|y^\star - y\|_2^2$ is large, it means that we need to update the trainable parameters in the direction to minimize this loss. And, this direction along *the negative of the gradient!* I.e., we update the parameters based on gradient-descent motions!

That being said, the following picture represents our neural network with the help of *modules*:

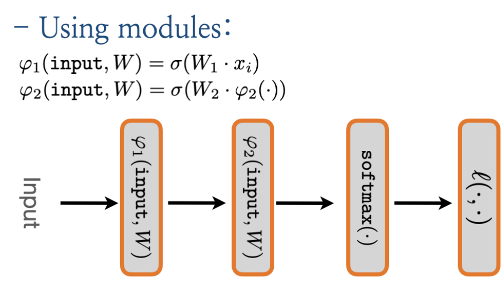


Fig. 16. Module-based representation of our toy neural network

It turns out that the backpropagation for this model is the efficient calculation and reuse of the intermediate steps, as shown in the following picture:

$$\begin{aligned}
\nabla f_i(x) &= \frac{\partial \ell(y_i, \widehat{y}_i)}{\partial W} \\
&= \frac{\partial \ell(y_i, \widehat{y}_i)}{\partial \texttt{softmax}(\cdot)} \cdot \frac{\partial \texttt{softmax}(\cdot)}{\partial W} \\
&= \frac{\partial \ell(y_i, \widehat{y}_i)}{\partial \texttt{softmax}(\cdot)} \cdot \frac{\partial \texttt{softmax}(\cdot)}{\partial \varphi(\cdot, W_2)} \cdot \frac{\partial \varphi(\cdot, W_2)}{\partial W} \\
&= \cdots
\end{aligned}$$

Fig. 17. Chain of derivatives

The idea of these calculations is that they define a *graph* of intermediate calculations that can be reused in order to complete the full gradient computation: *efficiently calculating, storing and reusing these intermediate steps is at the core of autodiff.*

[4] This is the case of *supervised learning* where $\mathcal{D}$ is a dataset that has both inputs x and the correct labels $y^\star$; these $y^\star$ will help the model update its parameters in order the output to be as close as possible to these $y^\star$.

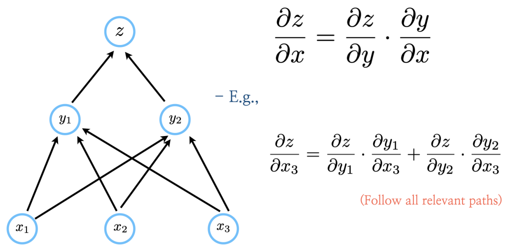


Fig. 18. Graph representation of autodiff

Appendix

1. J. Nocedal and S. Wright. Numerical optimization. Springer Science & Business Media, 2006.
2. Y. Nesterov. Introductory lectures on convex optimization: A basic course, volume 87. Springer Science & Business Media, 2013.
3. S. Boyd and L. Vandenberghe. Convex optimization. Cambridge university press, 2004.
4. D. Bertsekas. Convex optimization algorithms. Athena Scientific Belmont, 2015.
5. Sébastien Bubeck. Convex optimization: Algorithms and complexity. Foundations and Trends® in Machine Learning, 8(3-4):231–357, 2015.
6. S. Weisberg. Applied linear regression, volume 528. John Wiley & Sons, 2005.
7. T. Hastie, R. Tibshirani, and M. Wainwright. Statistical learning with sparsity: the lasso and generalizations. CRC press, 2015.
8. J. Friedman, T. Hastie, and R. Tibshirani. The elements of statistical learning, volume 1. Springer series in statistics New York, 2001.
9. M. Paris and J. Rehacek. Quantum state estimation, volume 649. Springer Science & Business Media, 2004.
10. M. Daskin. A maximum expected covering location model: formulation, properties and heuristic solution. Transportation science, 17(1):48–70, 1983.
11. I. Goodfellow, Y. Bengio, and A. Courville. Deep learning. MIT press, 2016.
12. L. Trefethen and D. Bau III. Numerical linear algebra, volume 50. Siam, 1997.
13. G. Strang. Introduction to linear algebra, volume 3. Wellesley-Cambridge Press Wellesley, MA, 1993.
14. G. Golub. Cmatrix computations. The Johns Hopkins, 1996.
15. A. Krizhevsky, I. Sutskever, and G. Hinton. Imagenet classification with deep convolutional neural networks. In Advances in neural information processing systems, pages 1097–1105, 2012.
16. K. He, X. Zhang, S. Ren, and J. Sun. Deep residual learning for image recognition. In Proceedings of the IEEE conference on computer vision and pattern recognition, pages 770–778, 2016.
17. S. Ren, K. He, R. Girshick, and J. Sun. Faster R-CNN: Towards real-time object detection with region proposal networks. In Advances in neural information processing systems, pages 91–99, 2015.
18. T. Mikolov, I. Sutskever, K. Chen, G. Corrado, and J. Dean. Distributed representations of words and phrases and their compositionality. In Advances in neural information processing systems, pages 3111–3119, 2013.
19. Dzmitry Bahdanau, Kyunghyun Cho, and Yoshua Bengio. Neural machine translation by jointly learning to align and translate. arXiv preprint arXiv:1409.0473, 2014.
20. Jonas Gehring, Michael Auli, David Grangier, Denis Yarats, and Yann N Dauphin. Convolutional sequence to sequence learning. In Proceedings of the 34th International Conference on Machine Learning-Volume 70, pages 1243–1252. JMLR. org, 2017.
21. Haşim Sak, Andrew Senior, and Françoise Beaufays. Long short-term memory recurrent neural network architectures for large scale acoustic modeling. In Fifteenth annual conference of the international speech communication association, 2014.
22. Tom Sercu, Christian Puhrsch, Brian Kingsbury, and Yann LeCun. Very deep multilingual convolutional neural networks for LVCSR. In 2016 IEEE International Conference on Acoustics, Speech and Signal Processing (ICASSP), pages 4955–4959. IEEE, 2016.
23. Ashish Vaswani, Noam Shazeer, Niki Parmar, Jakob Uszkoreit, Llion Jones, Aidan N. Gomez, Lukasz Kaiser, and Illia Polosukhin. Attention Is All You Need. page arXiv:1706.03762, 2017.
24. Jacob Devlin, Ming-Wei Chang, Kenton Lee, and Kristina Toutanova. BERT: Pre-training of Deep Bidirectional Transformers for Language Understanding. page arXiv:1810.04805, 2018.
25. Luowei Zhou, Hamid Palangi, Lei Zhang, Houdong Hu, Jason J Corso, and Jianfeng Gao. Unified vision-language pre-training for image captioning and VQA. In AAAI, pages 13041–13049, 2020.
26. Tom B Brown, Benjamin Mann, Nick Ryder, Melanie Subbiah, Jared Kaplan, Prafulla Dhariwal, Arvind Neelakantan, Pranav Shyam, Girish Sastry, Amanda Askell, et al. Language models are few-shot learners. arXiv preprint arXiv:2005.14165, 2020.
27. Mohammad Shoeybi, Mostofa Patwary, Raul Puri, Patrick LeGresley, Jared Casper, and Bryan Catanzaro. Megatron-lm: Training multi-billion parameter language models using gpu model parallelism. arXiv preprint arXiv:1909.08053, 2019.
28. Colin Raffel, Noam Shazeer, Adam Roberts, Katherine Lee, Sharan Narang, Michael Matena, Yanqi Zhou, Wei Li, and Peter J Liu. Exploring the limits of transfer learning with a unified text-to-text transformer. arXiv preprint arXiv:1910.10683, 2019.
29. Gary Marcus, Ernest Davis, and Scott Aaronson. A very preliminary analysis of DALL-E 2. arXiv preprint arXiv:2204.13807, 2022.
30. John Jumper, Richard Evans, Alexander Pritzel, Tim Green, Michael Figurnov, Olaf Ronneberger, Kathryn Tunyasuvunakool, Russ Bates, Augustin Žídek, Anna Potapenko, et al. Highly accurate protein structure prediction with AlphaFold. Nature, 596(7873):583–589, 2021.
31. Tom B. Brown, Benjamin Mann, Nick Ryder, Melanie Subbiah, Jared Kaplan, Prafulla Dhariwal, Arvind Neelakantan, Pranav Shyam, Girish Sastry, Amanda Askell, Sandhini Agarwal, Ariel Herbert-Voss, Gretchen Krueger, Tom Henighan, Rewon Child, Aditya Ramesh, Daniel M. Ziegler, Jeffrey Wu, Clemens Winter, Christopher Hesse, Mark Chen, Eric Sigler, Mateusz Litwin, Scott Gray, Benjamin Chess, Jack Clark, Christopher Berner, Sam McCandlish, Alec Radford, Ilya Sutskever, and Dario Amodei. Language models are few-shot learners, 2020.
32. Or Sharir, Barak Peleg, and Yoav Shoham. The cost of training nlp models: A concise overview. arXiv preprint arXiv:2004.08900, 2020.
33. H. Karimi, J. Nutini, and M. Schmidt. Linear convergence of gradient and proximal-gradient methods under the Polyak-Łojasiewicz condition. In Joint European Conference on Machine Learning and Knowledge Discovery in Databases, pages 795–811. Springer, 2016.
34. Philip Wolfe. Convergence conditions for ascent methods. SIAM review, 11(2):226–235, 1969.
35. Larry Armijo. Minimization of functions having Lipschitz continuous first partial derivatives. Pacific Journal of mathematics, 16(1):1–3, 1966.
36. Stephen Wright and Jorge Nocedal. Numerical optimization. Springer Science, 35(67-68):7, 1999.
37. B. Polyak. Introduction to optimization. Inc., Publications Division, New York, 1, 1987.
38. Stephen Boyd, Lin Xiao, and Almir Mutapcic. Subgradient methods. lecture notes of EE392o, Stanford University, Autumn Quarter, 2004:2004–2005, 2003.
39. Marguerite Frank, Philip Wolfe, et al. An algorithm for quadratic programming. Naval research logistics quarterly, 3(1-2):95–110, 1956.
40. M. Jaggi. Revisiting Frank-Wolfe: Projection-free sparse convex optimization. In Proceedings of the 30th international conference on machine learning, number CONF, pages 427–435, 2013.
41. J. Duchi, S. Shalev-Shwartz, Y. Singer, and T. Chandra. Efficient projections onto the ℓ_1-ball for learning in high dimensions. In Proceedings of the 25th international conference on Machine learning, pages 272–279, 2008.
42. Y. Koren, R. Bell, and C. Volinsky. Matrix factorization techniques for recommender systems. Computer, 42(8):30–37, 2009.
43. A. Mnih and R. Salakhutdinov. Probabilistic matrix factorization. In Advances in neural information processing systems, pages 1257–1264, 2008.
44. T. Booth and J. Gubernatis. Improved criticality convergence via a modified Monte Carlo power iteration method. Technical report, Los Alamos National Lab.(LANL), Los Alamos, NM (United States), 2008.
45. Kenneth Levenberg. A method for the solution of certain non-linear problems in least squares. Quarterly of applied mathematics, 2(2):164–168, 1944.
46. Donald W Marquardt. An algorithm for least-squares estimation of nonlinear parameters. Journal of the society for Industrial and Applied Mathematics, 11(2):431–441, 1963.
47. Andreas Griewank. The modification of Newton's method for unconstrained optimization by bounding cubic terms. Technical report, Technical report NA/12, 1981.
48. Yurii Nesterov and Boris T Polyak. Cubic regularization of Newton method and its global performance. Mathematical Programming, 108(1):177–205, 2006.
49. Yurii Nesterov and Arkadii Nemirovskii. Interior-point polynomial algorithms in convex programming. SIAM, 1994.
50. Ulysse Marteau-Ferey, Francis Bach, and Alessandro Rudi. Globally convergent Newton methods for ill-conditioned generalized self-concordant losses. Advances in Neural Information Processing Systems, 32, 2019.
51. Quoc Tran-Dinh, Anastasios Kyrillidis, and Volkan Cevher. Composite self-concordant minimization. J. Mach. Learn. Res., 16(1):371–416, 2015.
52. Konstantin Mishchenko. Regularized Newton method with global $o(1/k^2)$ convergence. arXiv preprint arXiv:2112.02089, 2021.
53. S. Zavriev and F. Kostyuk. Heavy-ball method in nonconvex optimization problems. Computational Mathematics and Modeling, 4(4):336–341, 1993.
54. E. Ghadimi, H. Feyzmahdavian, and M. Johansson. Global convergence of the heavy-ball method for convex optimization. In 2015 European control conference (ECC), pages 310–315. IEEE, 2015.
55. Y. Nesterov. A method for solving the convex programming problem with convergence rate $o(1/k^2)$. In Dokl. akad. nauk Sssr, volume 269, pages 543–547, 1983.
56. B. O'Donoghue and E. Candes. Adaptive restart for accelerated gradient schemes. Foundations of computational mathematics, 15(3):715–732, 2015.
57. O. Devolder, F. Glineur, and Y. Nesterov. First-order methods of smooth convex optimization with inexact oracle. Mathematical Programming, 146(1-2):37–75, 2014.
58. L. Bottou, F. Curtis, and J. Nocedal. Optimization methods for large-scale machine learning. Siam Review, 60(2):223–311, 2018.