EECS 553: Machine Learning (ECE) **University of Michigan**

Iterative Algorithms for Continuous Optimization

Winter 2023 **Clayton Scott**

Optimization algorithms is a vast topic and in these notes we overview some key iterative methods that are commonly used in machine learning. These methods are general in that they can be applied in a number of different settings. We consider an unconstrained optimization problem of the form

$$\min_{\boldsymbol{\theta} \in \mathbb{R}^k} \; J(\boldsymbol{\theta}). \tag{1}$$

We assume that J is a continuous function of $\boldsymbol{\theta}$. In a machine learning context, $\boldsymbol{\theta}$ might be the weights of a linear predictor or the weights of a neural network. J is a function that reflects some machine learning goal and is usual derived from statistical or geometric principles.

1 First-order methods

This section covers gradient descent and its variants.

1.1 Steepest ascent property

Assume J is differentiable. Suppose $\boldsymbol{\theta}_0$ is a fixed point. Let $\boldsymbol{u} \in \mathbb{R}^k$ be a unit vector, and define the function

$$\phi(t) = J(\boldsymbol{\theta}_0 + \boldsymbol{u}t).$$

The *directional derivative* of J at $\boldsymbol{\theta}_0$, if it exists, is defined to be

$$\phi'(0) = \lim_{t \searrow 0} \frac{J(\boldsymbol{\theta}_0 + \boldsymbol{u}t) - J(\boldsymbol{\theta}_0)}{t}.$$

By the multivariable chain rule, this is equal to $\langle \nabla J(\boldsymbol{\theta}_0), \boldsymbol{u} \rangle$. A natural question is which direction $\boldsymbol{u}$ maximizes the directional derivative. This direction is referred to as the direction of *steepest ascent*. By the Cauchy-Schwartz inequality (condition for equality), $\langle \nabla J(\boldsymbol{\theta}_0), \boldsymbol{u} \rangle$ is maximized when $\boldsymbol{u}$ is a scalar multiple of $\nabla J(\boldsymbol{\theta}_0)$, namely $\boldsymbol{u} = \frac{\nabla J(\boldsymbol{\theta}_0)}{\|\nabla J(\boldsymbol{\theta}_0)\|}$. Therefore the gradient of J at $\boldsymbol{\theta}_0$ is in the direction of steepest ascent. See Figure 1.

1.2 Gradient descent

Still assuming J is differentiable, by similar reasoning to the above, $-\nabla J(\boldsymbol{\theta}_0)$ is in the direction of steepest *descent* at $\boldsymbol{\theta}_0$. This motivates the following simple iterative algorithm for (1), known as *gradient descent*.

> Initialize $\boldsymbol{\theta}_0$
> For $t = 1, \ldots,$ `max_iter`
> $\quad \boldsymbol{\theta}_t \leftarrow \boldsymbol{\theta}_{t-1} - \eta \nabla J(\boldsymbol{\theta}_{t-1})$
> $\quad$ If convergence condition satisfied, exit
> End

η is called the *learning rate* or *step-size parameter*. If η is too large, $\boldsymbol{\theta}_t$ might keep oscillating around the minimizer. See Fig. 2. On the other hand, choosing η to be too small causes the algorithm to converge slowly. As a compromise, typically the step size is gradually decreased as t increases. Gradient descent is an example of a *descent method*, which means that if η is chosen appropriately, the sequence of objective function values $J(\boldsymbol{\theta}_t)$ in nonincreasing.

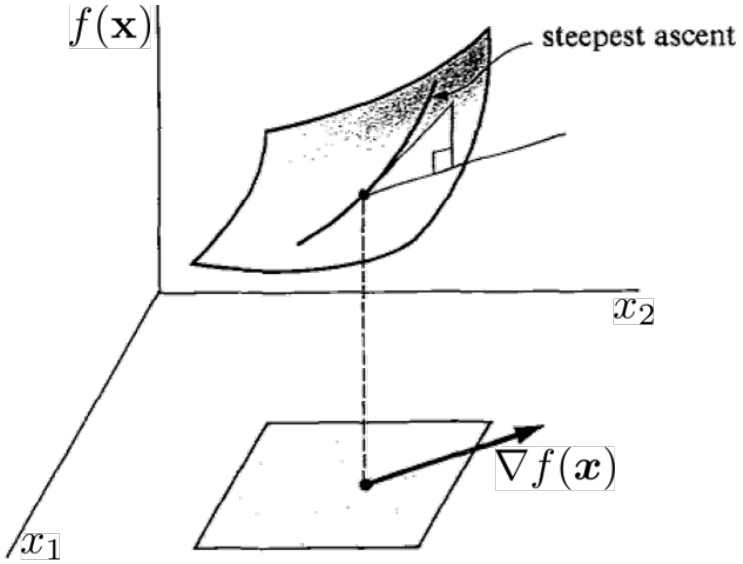


Figure 1: The gradient points in the direction of steepest ascent.

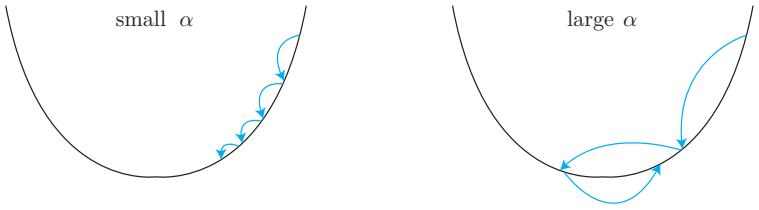


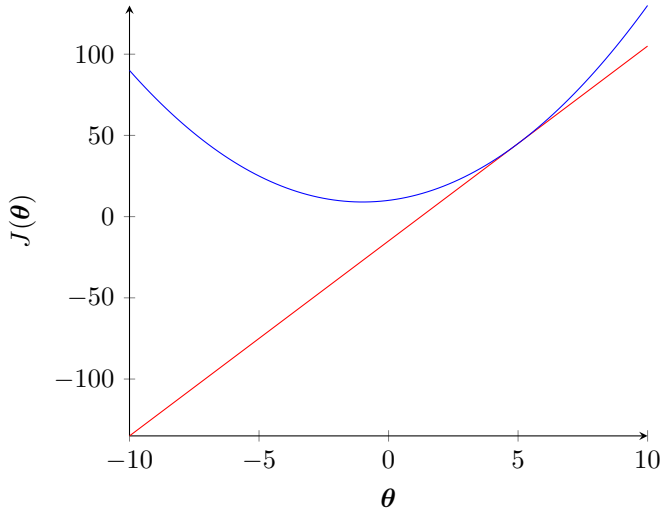
Figure 2: Gradient descent converges to a local minimizer for small η. For large η, oscillation may occur. (α should be η in the figure.)

1.3 The subgradient method

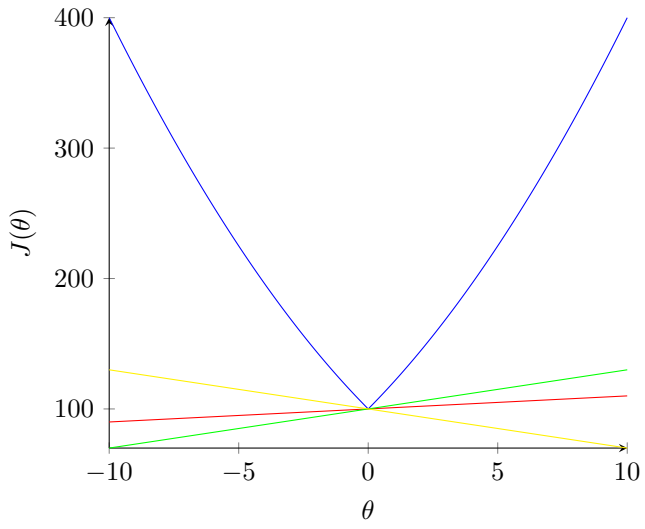
The *subgradient method* is an extension of gradient descent that applies to *nondifferentiable*, *convex* functions, like empirical risk minimization (ERM) with hinge loss, or ERM with squared error loss and ℓ_1-norm penalty.

Let J be convex, and let $\boldsymbol{\theta} \in \mathbb{R}^k$. If J is differentiable, then $\boldsymbol{u} = \nabla J(\boldsymbol{\theta})$ is the only vector such that

$$J(\boldsymbol{\theta}') \geq J(\boldsymbol{\theta}) + \boldsymbol{u}^T(\boldsymbol{\theta}' - \boldsymbol{\theta}) \quad \forall \boldsymbol{\theta}'. \tag{2}$$



If J is convex but *not* differentiable, then for some $\boldsymbol{\theta}$, there may be many $\boldsymbol{u}$ satisfying (2). Define the *subdifferential* of J at $\boldsymbol{\theta}$, denoted $\partial J(\boldsymbol{\theta})$, to be the set of all $\boldsymbol{u}$ satisfying (2). A *subgradient* is any element of the subdifferential.



In the figure above where J has a scalar input, the subdifferential is the interval $\left[J'_-(\theta), J'_+(\theta)\right]$ where $J'_-(\theta), J'_+(\theta)$ denote the left and right derivatives.

In the subgradient method, we update the parameter just as in gradient descent, but where the gradient is replaced by *any* subgradient. Here's the pseudo-code for minimizing $J(\boldsymbol{\theta})$.

Initialize $\boldsymbol{\theta}_0$
$t \leftarrow 0$
Repeat
  select $\boldsymbol{u}_t \in \partial J(\boldsymbol{\theta}_t)$
  $\boldsymbol{\theta}_{t+1} \leftarrow \boldsymbol{\theta}_t - \eta_t \boldsymbol{u}_t$
  $t \leftarrow t + 1$
Until stopping criterion satisfied

Remark 1. While subgradients are only guaranteed to exist for convex objectives, the subgradient method is nonetheless applied to minimize nonconvex objectives in practice, such as a neural network with ReLU activation.

Remark 2. Unlike gradient descent, where one can always find a step-size such that the objective decreases (unless you're at a local min), the objective will occasionally increase as you iterate a subgradient method. Therefore the subgradient method is not a descent method, although it is often referred to as "subgradient descent."

1.4 Stochastic Gradient Descent

Suppose it is possible to write $J(\boldsymbol{\theta}) = \sum_{i=1}^{n} J_i(\boldsymbol{\theta})$, where $J_i(\boldsymbol{\theta})$ involves only the i^{th} training pair $(\boldsymbol{x}_i, y_i)$. This situation arise often in practice, such as with any empirical risk minimization problem. *Stochastic gradient descent* (SGD) is the following variation on gradient descent:

Initialize $\boldsymbol{\theta}_0$
$t \leftarrow 0$
For $j = 1, \ldots,$ `max_iter`
  For $i = 1, \ldots, n$ *in random order*
    $\boldsymbol{\theta}_{t+1} \leftarrow \boldsymbol{\theta}_t - \eta_t \nabla J_i(\boldsymbol{\theta}_t)$
    If convergence condition satisfied, exit
    $t \leftarrow t + 1$
  End
End

The basic idea is that $\nabla J(\boldsymbol{\theta}) = \sum_i \nabla J_i(\boldsymbol{\theta})$. Instead of waiting to sum over all i to compute the full gradient, which is what GD does, SGD starts taking steps as each term $\nabla J_i(\theta)$ is computed. On one hand, the steps are not necessarily in the steepest descent direction, and may not even be descent directions. On the other hand, however, the SGD steps point *on average* in a useful direction (because their average is proportional to the gradient). Therefore, by the time we get to the end of the inner loop, SGD has taken several mostly useful steps and is able to advance further toward the solution than a single step of GD. This is confirmed in practice, where SGD converges with far fewer computations compared to GD.

Remark 3. The reason for using a random order for the inner loop is based on some theory that guarantees convergence of SGD. Intuitively, a random order helps avoid cycles (endlessly tracing the same path in $\boldsymbol{\theta}$ space). Randomization is commonly used in practice, although deterministic schedules can also produce fine results.

An import generalization of both GD and SGD is called *mini-batch SGD*. The basic idea is, instead of updating $\boldsymbol{\theta}$ based on a single term, the update is based on a few terms. If b is the batch size, mini-batch SGD loops through different subsets $\Omega_j \subset \{1, \ldots, n\}$ of size b, and the update step becomes

$$\boldsymbol{\theta}_{t+1} \leftarrow \boldsymbol{\theta}_t - \eta_t \sum_{i \in \Omega_j} \nabla J_i(\boldsymbol{\theta}_t).$$

The batch size is a tuning parameter. Mini-batch SGD offers a nice compromise between SGD and GD. This is because the updates are more likely to be near the steepest descent direction, which results from averaging $\nabla J_i(\boldsymbol{\theta}_t)$ over the mini-batch.

All of these ideas extend naturally to the subgradient method, yielding the stochastic subgradient method.

2 Second Order Methods

First order methods can be very slow to converge. Second order methods use both first and second order derivatives, and are able to converge in far fewer iterations than first order methods. Intuitively, second order methods use information about the curvature of J to take steps aimed closer to the global minimizer. The drawback of second order methods is that they are more expensive per iteration.

2.1 Newton's Method

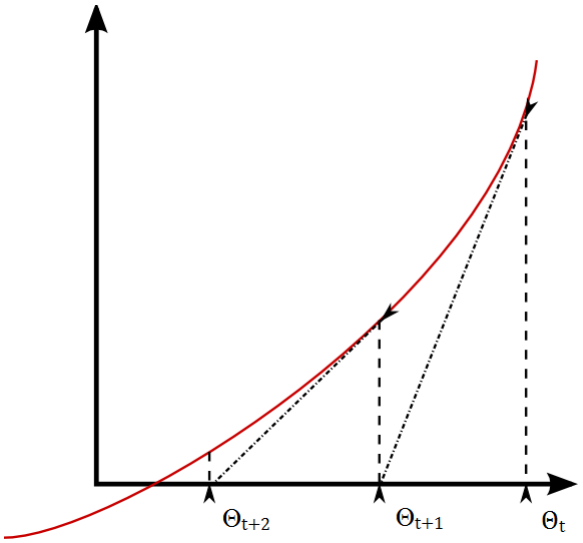
Newton's method, also known as the *Newton-Raphson* method, iterates

$$\boldsymbol{\theta}_{t+1} = \boldsymbol{\theta}_t - (\nabla^2 J(\boldsymbol{\theta}_t))^{-1} \nabla J(\boldsymbol{\theta}_t).$$

Newton's method can be viewed as successively approximating J using the second order approximation

$$J(\boldsymbol{\theta}) \approx J(\boldsymbol{\theta}_t) + \nabla J(\boldsymbol{\theta}_t)^T (\boldsymbol{\theta} - \boldsymbol{\theta}_t) + \frac{1}{2} (\boldsymbol{\theta} - \boldsymbol{\theta}_t)^T \nabla^2 J(\boldsymbol{\theta}_t) (\boldsymbol{\theta} - \boldsymbol{\theta}_t), \tag{3}$$

and jumping to the critical point of that approximation. Newton's method requires far fewer iterations than gradient descent, but each step takes longer to compute, because of the need to invert the Hessian. For high-dimensional parameter spaces, Newton's method is impractical.



Note that Newton's method converges to a critical point, which in general could be a local minimum, local maximum, or saddle point. If the objective function is nonconvex, Newton's method may need to be initialized carefully.

2.2 Quasi-Newton Methods

Quasi-Newton methods are iterative methods for continuous optimization that aim to approximate the inverse Hessian in some sense, striving for the benefits of Newton's method without the computational expense.

Important examples are the BFGS algorithm, name after its developers Broyden–Fletcher–Goldfarb–Shanno, and L-BFGS, the "limited memory" verison of BFGS. L-BFGS is scalable and general purpose, and should be strongly considered when attempting to solve a continuous optimization problem for which no efficient solver has already been developed. An efficient implementation is provided by Python's SciPy package.

3 Majorize-Minimize Algorithms

Majorize-minimize algorithms represent a different class of iterative algorithms that do not rely on gradients or Hessians. Here is the basic algorithm:

Initialize $\boldsymbol{\theta}_0$
$t \leftarrow 0$
Repeat
Majorize: Find a function $J_t(\boldsymbol{\theta})$ such that

$$J(\boldsymbol{\theta}_t) = J_t(\boldsymbol{\theta}_t)$$
$$J(\boldsymbol{\theta}) \leq J_t(\boldsymbol{\theta}) \qquad \forall \boldsymbol{\theta}$$

Minimize: Solve

$$\boldsymbol{\theta}_{t+1} \leftarrow \arg\min_{\boldsymbol{\theta}} J_t(\boldsymbol{\theta})$$

$t \leftarrow t + 1$
Until convergence

See Fig. 3 for an illustration.

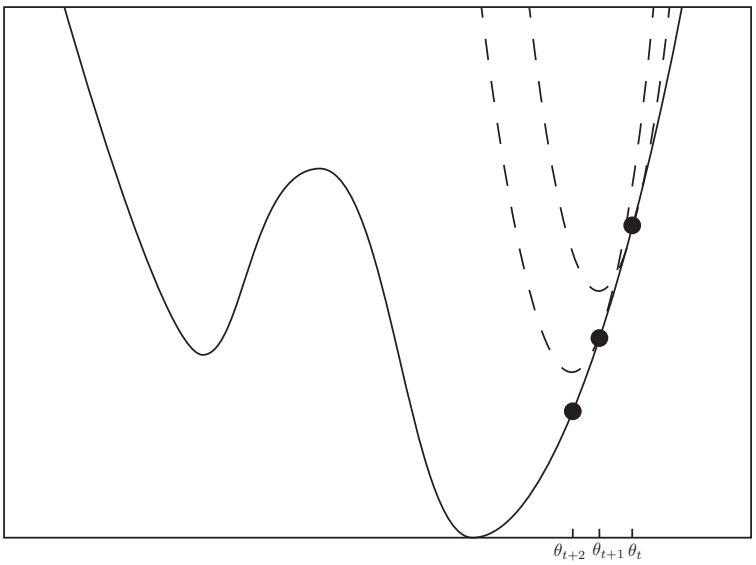


Figure 3: Majorize/minimize method.

The function $J_t(\boldsymbol{\theta})$ is called a *majorizer* or *majorizing function*. The majorizer upper bounds $J(\boldsymbol{\theta})$, and is equal to $J(\boldsymbol{\theta})$ at the current iterate $\boldsymbol{\theta}_t$. The MM algorithm alternates between finding a majorizer, and minimizing that majorizer to obtain the next iterate. The point of MM algorithms is that for many optimization problems, the objective $J(\boldsymbol{\theta})$ may be complex, but the majorizer is simpler and computationally easy to optimize.

Prominent examples of MM algorithms in machine learning include the iteratively re-weighted least squares algorithm for robust regression, the expectation-maximization algorithm, and the MM algorithm for stress-based multi-dimensional scaling. Unfortunately, there is no single recipe for deriving majorizing functions. While there are a few common techniques, in general majorizers must be derived anew for each new optimization problem considered.

MM algorithms are attractive because they are descent algorithms.

Theorem 1. *MM algorithms produce a sequence of iterates $\boldsymbol{\theta}_t$ satisfying $J(\boldsymbol{\theta}_{t+1}) \leq J(\boldsymbol{\theta}_t)$ for all t.*

Proof.

$$J(\boldsymbol{\theta}_{t+1}) \leq J_t(\boldsymbol{\theta}_{t+1}) \leq J_t(\boldsymbol{\theta}_t) = J(\boldsymbol{\theta}_t),$$

where the first step is because J_t is a majorizer, the second step is by the definition of the minimize step, and the third step is again because J_t is a majorizer. □

Exercises

1. (★★) Show that Newton's method results from finding a critical point of the right-hand side of (3). You may assume the Hessian is invertible.

2. (★★) Let $J(\boldsymbol{\theta})$ be a twice differentiable function and suppose that there exists a symmetric and positive definite matrix $\boldsymbol{B}$, not depending on $\boldsymbol{\theta}$, such that for all $\boldsymbol{\theta}$, $\boldsymbol{B} - \nabla^2 J(\boldsymbol{\theta})$ is positive semi-definite. Consider an iterative algorithm for minimizing J that produces a sequence of estimates $\boldsymbol{\theta}_t$, $t = 1, 2, \ldots$.

   (a) Argue that

   $$J_t(\boldsymbol{\theta}) = J(\boldsymbol{\theta}_t) + \nabla J(\boldsymbol{\theta}_t)^T(\boldsymbol{\theta} - \boldsymbol{\theta}_t) + \frac{1}{2}(\boldsymbol{\theta} - \boldsymbol{\theta}_t)^T \boldsymbol{B}(\boldsymbol{\theta} - \boldsymbol{\theta}_t)$$

   is a majorizing function for J. *Hint*: Use one of the multivariable versions of Taylor's theorem found in the exercises on unconstrained optimization.

   (b) Determine a formula for

   $$\boldsymbol{\theta}_{t+1} = \arg\min_{\boldsymbol{\theta}} \; J_t(\boldsymbol{\theta}).$$

   (c) Devise a majorize-minimize algorithm for regularized logistic regression. What is a computational advantage of the MM algorithm compared to Newton's method?