Optimization Methods - Optimization Techinques for Machine Learning: Additional Lecture Notes

Matteo Lapucci
matteo.lapucci@unifi.it

Department of Information Engineering, Università degli Studi di Firenze

Contents

1 Further results on Unconstrained Optimization

In this Section we review some additional or alternative analysis of about nonlinear optimization solvers.

1.1 Efficiency of optimization solvers

When studying optimization algorithms, the primary interest lies in studying the local and global convergence properties: it is fundamental to ensure that they effectively produce optimal solutions. However, efficiency of the algorithms is clearly also very important, especially in the large scale scenario where computing times may be really long. Efficiency of optimization algorithms is usually studied in terms of two different concepts: *convergence rate* and *iteration complexity*.

1.1.1 Convergence Rate

Convergence rate intuitively denotes how fast the sequence of objective values $\{f(x^k)\}$, or equivalently the sequence of iterates $\{x^k\}$, approaches the limit point. Formally, we have the following possible cases.

Definition 1. Let $\{f(x^k)\}$ be the sequence of objective values generated by an iterative algorithms, with $f(x^k) \to f^\star$. Then, we say that the rate of convergence is

- *sublinear* if
$$\lim_{k\to\infty} \frac{f(x^{k+1}) - f^\star}{f(x^k) - f^\star} = 1;$$
- *linear* if
$$\lim_{k\to\infty} \frac{f(x^{k+1}) - f^\star}{f(x^k) - f^\star} = \rho$$
for some $\rho \in (0, 1)$;
- *superlinear* if
$$\lim_{k\to\infty} \frac{f(x^{k+1}) - f^\star}{f(x^k) - f^\star} = 0;$$
- *quadratic* if
$$\lim_{k\to\infty} \frac{f(x^{k+1}) - f^\star}{(f(x^k) - f^\star)^2} = \rho$$
for some $\rho > 0$.

Having a sublinear convergence rate is bad: the longer you run the algorithm, the less progress it makes. In brief, linear rate is ok, superlinear is great, quadratic is amazing.

1.1.2 Iteration Complexity

Another very popular approach to measuring the efficiency of optimization algorithms is based on the concept of iteration cost. Of course, even if algorithms have asymptotic convergence properties, in practice they are stopped after a finite time. Now, the interesting question is how many iterations are required to reach a particular accuracy level ϵ.

To carry out this kind of analysis, a quite ideal setting is typically considered; we assume the objective function to be

- bounded below (global minimum $f^\star > -\infty$);
- convex;
- with Lipschitz-continuous gradient ∇f, of constant L.

Stronger results will be obtainable if the objective function satisfies the following property.

Definition 2. A function $f : \mathbb{R}^n \to \mathbb{R}$ is *strongly convex* if there exists $\mu > 0$ such that $f(x) - \mu\|x\|^2$ is a convex function, i.e., $f(x) = g(x) + \mu\|x\|^2$ being g a convex function.

We are now able to define the concept of iteration complexity.

Definition 3. Let $\{x^k\}$ be the sequence of iterates generated by an iterative method, with $f(x^k) \to f^\star$. We say that the algorithm has an *iteration error* of $\mathcal{O}(h(k))$ if:

$$f(x^k) - f^\star = \mathcal{O}(h(k));$$

equivalently, given an accuracy level ϵ, the algorithm has an *iteration complexity* of $\mathcal{O}(\hat{h}(\epsilon))$ if

$$\min\{k \mid f(x^k) - f^\star < \epsilon\} = \mathcal{O}(\hat{h}(\epsilon)).$$

As remarked in the definition, there is a correspondence between iteration error and iteration complexity; if an algorithm has an iteration error of $\mathcal{O}(\frac{1}{k})$, then in order to reach an accuracy ϵ we need a number k of iterations such that

$$f(x^k) - f^\star \approx \frac{1}{k} < \epsilon,$$

i.e.,

$$k > \frac{1}{\epsilon}.$$

In other words, if the iteration error is $\mathcal{O}(\frac{1}{k})$, the iteration complexity is $\mathcal{O}(\frac{1}{\epsilon})$; similarly, if the iteration error is $\mathcal{O}(\frac{1}{k^2})$, the iteration complexity is $\mathcal{O}(\frac{1}{\sqrt{\epsilon}})$.

Note that $\mathcal{O}(\frac{1}{\epsilon})$ is not a good iteration complexity. We can think of $\log(\frac{1}{\epsilon})$ as "number of digits of accuracy" wanted. With an $\mathcal{O}(\frac{1}{\epsilon})$ complexity, if we need 10 iterations for a digit of accuracy, then we might need 100 iterations for 2 digits, 1000 iterations for 3 digits, and so on: time is exponential.

This can be put in relation with convergence speed:

- An iteration complexity of $\mathcal{O}(\frac{1}{\epsilon})$ is equivalent to an error of $\mathcal{O}(\frac{1}{k})$, which can be substituted into the definition of convergence rate:

$$\lim_{k\to\infty} \frac{f(x^{k+1}) - f^\star}{f(x^k) - f^\star} = \lim_{k\to\infty} \mathcal{O}\left(\frac{k}{k+1}\right) = 1,$$

  the rate is sublinear!

- An error of $\mathcal{O}(\rho^k)$ with $\rho < 1$ leads to linear convergence rate, with an iteration complexity of $\mathcal{O}(\log(\frac{1}{\epsilon}))$; the cost of adding a new digit of accuracy is "polynomial".

- An error of the kind $\mathcal{O}(\rho^{2^k})$ with $\rho < 1$ leads to a superlinear convergence rate, with an iteration complexity of $\mathcal{O}(\log(\log(\frac{1}{\epsilon})))$; the cost of adding a new digit of accuracy is "constant"!

1.2 Constant stepsize Gradient Descent method

In this Section, we discuss the convergence results concerning the *gradient descent method with constant stepsize*, i.e., the iterative optimization algorithm whose iterations take the form:

$$x^{k+1} = x^k - \alpha\nabla f(x^k),$$

being $\alpha > 0$ a constant, suitably chosen, stepsize. In the general case, a constant stepsize is not able to guarantee the sufficient decrease property leading to the convergence of the

overall algorithm. However, convergence properties can be established under sufficiently strong regularity assumptions.

Here, we are going to assume that f is twice continuously differentiable and that ∇f is a Lipschitz continuous function, i.e., there exists a constant $L > 0$ such that

$$\|\nabla f(x) - \nabla f(y)\| \leq L\|x - y\| \text{ for all } x, y \in \mathbb{R}^n.$$

For twice continuously differentiable functions, Lipschitz continuity is equivalent to the following condition: there exists some constant $L > 0$ such that, for all $x \in \mathbb{R}^n$ and any vector $u \in \mathbb{R}^n$,

$$\left|u^T \nabla^2 f(x) u\right| \leq L\|u\|^2.$$

This also implies the following lemma.

Lemma 1.1. *Let $f : \mathbb{R}^n \to \mathbb{R}$ be a twice continuously differentiable function, with L-Lipschitz continuous gradient. Then, for all $x \in \mathbb{R}^n$, $d \in \mathbb{R}^n$ and $t \in \mathbb{R}$, we have*

$$f(x + td) \leq f(x) + t\nabla f(x)^T d + \frac{1}{2}t^2 L\|d\|^2.$$

Proof. By Taylor's Theorem, we have for any x, d, and t:

$$f(x + td) = f(x) + t\nabla f(x)^T d + \frac{1}{2}t^2 d^T \nabla^2 f(y) d$$

for some $y \in \mathbb{R}^n$. By the Lipschitz continuity property of the gradient we thus have

$$\begin{aligned} f(x + td) &= f(x) + t\nabla f(x)^T d + \frac{1}{2}t^2 d^T \nabla^2 f(y) d \\ &\leq f(x) + t\nabla f(x)^T d + \frac{1}{2}t^2 \left|d^T \nabla^2 f(y) d\right| \\ &\leq f(x) + t\nabla f(x)^T d + \frac{1}{2}t^2 L\|d\|^2. \end{aligned}$$

□

We know show that, if we knew the value of the Lipschitz constant L and we set $\alpha_k = \alpha = \frac{1}{L}$ for all k in the gradient descent method, the algorithm actually converges.

Proposition 1.2. *Let $f : \mathbb{R}^n \to \mathbb{R}$ a twice continuously differentiable function with Lipschitz continuous gradient with Lipschitz constant L and let $x^0 \in \mathbb{R}^n$ be a point such that the level set $\mathcal{L}_f(x^0)$ is compact. Let $\{x^k\}$ be the sequence of solutions generated by the gradient descent method with constant step size $\alpha = \frac{1}{L}$; then:*

(a) $\{f(x^k)\}$ is a strictly decreasing sequence, satisfying

$$f(x^{k+1}) - f(x^k) \leq -\frac{1}{2L}\|\nabla f(x^k)\|^2 < 0; \tag{1}$$

(b) $\{x^k\}$ has accumulation points, each one being a stationary point of f.

Proof. (a) From the Lipschitz continuity of $\nabla f(x)$ and by Lemma 1.1, being $x^{k+1} = x^k + \alpha d_k$, we have that

$$\begin{aligned} f(x^{k+1}) &\leq f(x^k) + \alpha\nabla f(x^k)^T d_k + \frac{1}{2}\alpha^2 L\|d_k\|^2 \\ &= f(x^k) - \frac{1}{L}\nabla f(x^k)^T \nabla f(x^k) + \frac{L}{2L^2}\left\|-\nabla f(x^k)\right\|^2 \\ &= f(x^k) - \frac{1}{2L}\|\nabla f(x^k)\|^2, \end{aligned} \tag{2}$$

where we have exploited the fact that $\alpha = \frac{1}{L}$ and $d_k = -\nabla f(x^k)$. Hence, we get (1).

(b) By point (a), the sequence $\{f(x^k)\}$ is decreasing and thus admits limit $f^\star$. Moreover, the sequence $\{x^k\}$ thus belongs to $\mathcal{L}_f(x^0)$, which by the assumptions is a compact set. Hence, $\{x^k\}$ admits accumulation points. In addition, by the continuity of f and Weierstrass theorem, $\{f(x^k)\}$ is bounded. Hence its limit is finite. We can therefore take the limit in (1) for $k \to \infty$ to obtain

$$\lim_{k\to\infty} -\frac{1}{2L}\|\nabla f(x^k)\|^2 \geq 0,$$

i.e., $\nabla f(x^k) \to 0$ for $k \to \infty$. This completes the proof.

□

We shall remark that information about the value of the Lipschitz constant L is in general not available, or, at least, expensive to obtain. Thus, line searches are usually needed to have convergence guarantees; however, the above theorem provides an explanation about the gradient method with constant stepsize often working properly in practice; in fact, we could show that there exist a range of values for α around $1/L$ that actually make the algorithm convergent. Also, addressing the constant stepsize case makes it easier to analyze the complexity results for gradient descent; indeed, the result that we are going to prove hereafter for the case $\alpha = \frac{1}{L}$ is analogous to what we would obtain if we allowed line searches to be employed. Before proceeding, we remark that the complexity analysis of the algorithm of course requires us to state convexity assumptions for f.

Proposition 1.3. *Let f be a convex function with Lipschitz-continuous gradient of constant L, and $\{x^k\}$, converging to $x^\star$ ($f(x^\star) = f^\star$), be the sequence produced by the gradient descent algorithm with constant stepsize $\alpha_k = \frac{1}{L}$. Then,*

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

i.e., the algorithm has an iteration error of $\mathcal{O}(\frac{1}{k})$.

Proof. By the convexity of f, we can write

$$f(x^\star) \geq f(x^k) + \nabla f(x^k)^T(x^\star - x^k)$$

and rearranging

$$f(x^k) \leq f(x^\star) + \nabla f(x^k)^T(x^k - x^\star).$$

Combining this last result with (1), which holds from Proposition 1.2, we get

$$f(x^{k+1}) \leq f(x^\star) + \nabla f(x^k)^T(x^k - x^\star) - \frac{1}{2L}\|\nabla f(x^k)\|^2,$$

and hence

$$\begin{aligned}
f(x^{k+1}) - f(x^\star) &\leq \frac{L}{2}\left(\frac{2}{L}\nabla f(x^k)^T(x^k - x^\star) - \frac{1}{L^2}\|\nabla f(x^k)\|^2\right) \\
&= \frac{L}{2}\left(\frac{2}{L}\nabla f(x^k)^T(x^k - x^\star) - \frac{1}{L^2}\|\nabla f(x^k)\|^2 - \|x^k - x^\star\|^2 + \|x^k - x^\star\|^2\right) \\
&= \frac{L}{2}\left(-\|x^k - x^\star - \frac{1}{L}\nabla f(x^k)\|^2 + \|x^k - x^\star\|^2\right) \\
&= \frac{L}{2}\left(-\|x^{k+1} - x^\star\|^2 + \|x^k - x^\star\|^2\right),
\end{aligned}$$

where we have used $\|a\|^2 + \|b\|^2 - 2a^Tb = \|a - b\|^2$. For each k, we thus have

$$f(x^{k+1}) - f(x^\star) \leq \frac{L}{2}(\|x^k - x^\star\|^2 - \|x^{k+1} - x^\star\|^2).$$

Therefore

$$\begin{aligned}\sum_{t=0}^{k} f(x^{t+1}) - f(x^\star) &\leq \sum_{t=0}^{k} \frac{L}{2}(\|x^t - x^\star\|^2 - \|x^{t+1} - x^\star\|^2) \\ &= \frac{L}{2}(\|x^0 - x^\star\|^2 - \|x^{k+1} - x^\star\|^2) \\ &\leq \frac{L}{2}\|x^0 - x^\star\|^2.\end{aligned}$$

Recalling that $\{f(x^k)\}$ is decreasing, we have $f(x^{t+1}) \geq f(x^{k+1})$ for all $t = 0, \ldots, k$, hence

$$\sum_{t=0}^{k} f(x^{t+1}) - f(x^\star) \geq (k+1)(f(x^{k+1}) - f(x^\star)).$$

Putting everything together, we get

$$(k+1)(f(x^{k+1}) - f(x^\star)) \leq \frac{L}{2}\|x^0 - x^\star\|^2,$$

i.e.,

$$f(x^{k+1}) - f(x^\star) \leq \frac{L}{2(k+1)}\|x^0 - x^\star\|^2.$$

□

Things get much better if stronger assumptions are satisfied. In particular, under strong convexity assumptions on f, we could prove that gradient descent method has an iteration complexity of $\mathcal{O}(\log(\frac{1}{\epsilon}))$, an iteration error of $\mathcal{O}(\rho^k)$ and, thus, a linear convergence rate.

1.3 Heavy-ball (Momentum) and Accelerated gradient methods

In this Section, we briefly describe two different modifications within the gradient descent update rule, exploiting the history of past iterations in order to try to speed up convergence.

Heavy-ball (Polyak) The heavy-ball method from Polyak, also known as gradient descent with *momentum*, solves unconstrained nonlinear optimization problems by the following two-steps update rule:

$$\begin{cases} y^k = x^k - \alpha_k \nabla f(x^k) \\ x^{k+1} = y^k + \beta_k (x^k - x^{k-1}) \end{cases} \tag{3}$$

where α_k and β_k are chosen by a suitable rule.

This method is often explained by a (not so accurate) physical analogy: the vector of variables can be seen as a particle, moving in the Euclidean space, which naturally tends to preserve its current trajectory, but at the same time is deflected by the acceleration (the gradient) produced by some force.

The addition of the momentum term allows to alleviate oscillation and zigzagging effects and causes an acceleration along the previous direction at low curvatures, decreasing the number of steps to convergence.

The idea of the above update is that the direction of last iteration will arguably be a good descent direction even at the current one. Partly repeating the previous step helps with controlling oscillation and providing acceleration in low curvature regions.

The main computational advantage of introducing the momentum term lies in the fact that it only exploits already computed information: no additional evaluations of the objective function or, even worse, the gradient, are required.

The momentum update can be equivalently defined by the following pair of equations:

$$v^{k+1} = \beta v^k - \alpha \nabla f(x^k),$$
$$x^{k+1} = x^k + v^{k+1}.$$

The update vector v^k can be interpreted as a speed term, which is computed as an exponentially decaying mean of past negative gradients; the gradients are thus used to modify the speed, rather than the position of the particle. The movement is then performed according to the speed computed at the current positions.

Local convergence results can be established for this algorithm. Moreover, in the case of strictly convex quadratic functions, the optimal values for the parameters α_k and β_k can be computed in closed form. The choice of this optimal parameters allows to obtain a local linear convergence rate with better constants than standard gradient descent. This result can be can be generalized under hypotheses of twice continuous differentiability, Lipschitz-continuous gradient and strong convexity.

However, the heavy-ball method was shown to be potentially unable to converge even under very strong regularity assumptions. Moreover, except for the quadratic case, the selection of parameters α and β is not trivial; α might be chosen by a line search, whereas β is usually set to a constant chosen somewhat heuristically.

Nesterov Accelerated Gradient *Nestereov Accelerated Gradient* method can be seen as a variant of Momentum. The update rule of Nesterov algorithm is given by the following pair of equation:

$$v^{k+1} = \beta_k v^k - \alpha_k \nabla f(x^k + \beta_k v^k),$$
$$x^{k+1} = x^k + v^{k+1}.$$

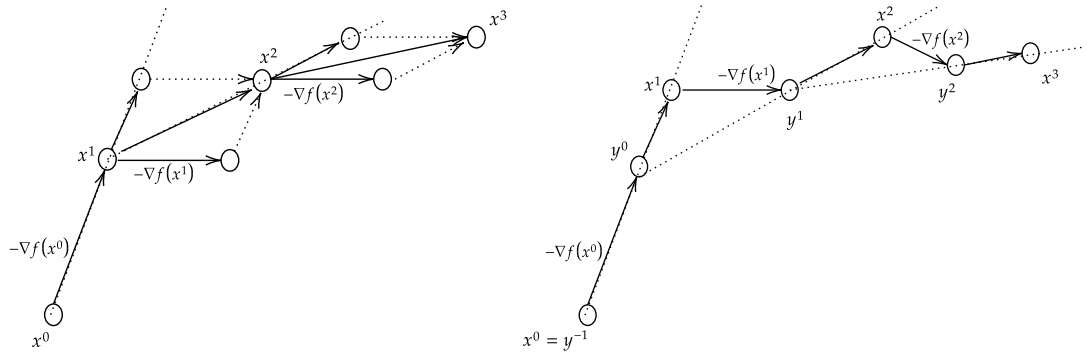
As we can see, the only difference w.r.t. Momentum lies in the fact that gradient is computed at the point that would be obtained if the last move was repeated, rather than at the current point; iterations are essentially made of two steps: first a pure momentum step (with no gradient influence) and then a pure gradient descent step. The update rule can indeed be equivalently rewritten as:

$$\begin{cases} y^k = x^k - \alpha_k \nabla f(x^k) \\ x^{k+1} = y^k + \beta_k (y^k - y^{k-1}) \end{cases} \tag{4}$$

The stepsize α_k can be computed by an Armijo line-search, whereas β_k follows a suitable schedule.

It has been proved that the the accelerated gradient algorithm achieves an iteration complexity of $\mathcal{O}(\frac{1}{\sqrt{\epsilon}})$, i.e., $\mathcal{O}(\frac{1}{k^2})$; this result is significant, since this complexity bound has also been shown to be optimal for first-order methods: using only gradients information there is no way of doing better. Note that, however, such a complexity still leads to a sublinear asymptotic convergence rate. As for the strongly convex case, the accelerated gradient method achieves the same iteration complexity and convergence rate as gradient descent, $\mathcal{O}(\rho^k)$, but with a smaller value of ρ, i.e., it has a faster linear convergence rate.

A visual comparison of the two algorithms can bee observed in Figure 1. We should also notice that, as opposed to Heavy-ball, Nesterov AG method is intrinsically a non-monotone descent method.



(a) Heavy-Ball (b) Nesterov AG

Figure 1: Visualization of the iterations of Momentum and Accelerated gradient algorithms.

1.4 Conjugate Gradient Method

The *conjugate gradient method* is an iterative algorithm which appears to be particularly efficient at solving quadratic optimization problems with strictly convex objective functions and linear systems of equations. The method is based on the following concept.

Definition 4. Directions $d_0, d_1, \ldots, d_{m-1} \in \mathbb{R}^n$ are *mutually conjugate* w.r.t. a symmetric positive definite matrix Q if $d_i^T Q d_j = 0 \, \forall \, i, j = 0, \ldots, m-1, \, i \neq j$.

Proposition 1.4. *Let $d_0, \ldots, d_{m-1}$ be conjugate directions w.r.t. a symmetric positive-definite matrix Q. Then, the directions are linearly independent.*

Proof. Let $\alpha_0, \ldots, \alpha_{m-1}$ real constants such that

$$\sum_{j=0}^{m-1} \alpha_j d_j = 0.$$

Multiplying (on the left) by $d_i^T Q$ both sides of the equation, we get

$$\sum_{j=0}^{m-1} \alpha_j d_i^T Q d_j = 0.$$

The directions are mutually conjugate, hence $d_i^T Q d_j = 0$ for all $j \neq i$; hence, we are left with

$$\alpha_i d_i^T Q d_i = 0;$$

we observe that $d_i^T Q d_i > 0$ being Q positive definite, hence $\alpha_i = 0$. Being i arbitrary, we get that $\alpha_0, \ldots, \alpha_{m-1}$ are all zero, i.e., the directions are linearly independent. □

Being mutually conjugate is something stronger than being linearly independent; indeed, let us consider the optimization problem

$$\min_x \frac{1}{2} x^T Q x - c^T x,$$

and let $d_0, \ldots, d_{n-1}$ conjugate direction w.r.t. Q (symmetric positive definite); since we are considering n linearly independently directions, we have for an optimal solution $x^\star$:

$$x^\star = \sum_{i=0}^{n-1} \alpha_i d_i, \qquad Q x^\star = \sum_{i=1}^{n} \alpha_i Q d_i.$$

Then, we derive

$$d_j^T Q x^\star = \sum_{i=0}^{n-1} \alpha_i d_j^T Q d_i = \alpha_j d_j^T Q d_j,$$

i.e.,

$$\alpha_j = \frac{d_j^T Q x^\star}{d_j^T Q d_j} = \frac{d_j^T c}{d_j^T Q d_j},$$

where we have used $\nabla f(x^\star) = Qx^\star - c = 0$. We can finally write

$$x^\star = \sum_{i=0}^{n-1} \alpha_i d_i = \sum_{i=0}^{n-1} \frac{d_i^T c}{d_i^T Q d_i} d_i.$$

If we knew n mutually conjugate directions w.r.t. Q, we would have the closed form solution for the problem.

However, we can interpret the closed form formula as the result of an iterative procedure in n steps, where each time the term $\alpha_i d_i$ is added; this is the idea of the *conjugate directions method*; let x^0 an arbitrary starting point and $d_0, \ldots, d_{n-1}$ a set of mutually conjugate directions; we have

$$x^\star = x^0 + \alpha_0 d_0 + \ldots + \alpha_{p-1} d_{n-1}, \qquad x^k = x^0 + \alpha_0 d_0 + \ldots + \alpha_{k-1} d_{k-1}$$

and thus

$$Q(x^\star - x^0) = \sum_{i=0}^{n-1} \alpha_i Q d_i, \qquad Q(x^k - x^0) = \sum_{i=0}^{k-1} \alpha_i Q d_i.$$

Since the directions are mutually conjugate, we also have

$$d_k^T Q(x^k - x^0) = \sum_{i=0}^{k-1} \alpha_i d_k^T Q d_i = 0$$

and, by similar reasonings,

$$d_k^T Q(x^\star - x^0) = \alpha_k d_k^T Q d_k.$$

Therefore, the step for direction d_k is given by

$$\begin{aligned} \alpha_k &= \frac{d_k^T Q(x^\star - x^0)}{d_k^T Q d_k} = \frac{d_k^T Q(x^\star - x^k)}{d_k^T Q d_k} + \frac{d_k^T Q(x^k - x^0)}{d_k^T Q d_k} \\ &= \frac{d_k^T Q(x^\star - x^k)}{d_k^T Q d_k} = \frac{d_k^T (c - Q x^k)}{d_k^T Q d_k} = \frac{-\nabla f(x^k)^T d_k}{d_k^T Q d_k}, \end{aligned}$$

i.e., it coincides with the optimal step along direction d_k at x^k.

Proposition 1.5 (Finite Convergence of the Conjugate Directions Method)**.** *Let $d_0, \ldots, d_{n-1}$ a set of mutually conjugate directions w.r.t. Q. Let $x^0 \in \mathbb{R}^n$ and $x^{k+1} = x^k + \alpha_k d_k$, $\alpha_k = \frac{-\nabla f(x^k)^T d_k}{d_k^T Q d_k}$. Then, letting $g_k = \nabla f(x^k)$, we have*

(i) for all k, $g_k^T d_i = 0$, for all $i = 0, \ldots, k-1$;

(ii) there exists $m \leq n-1$ such that $x^{m+1} = x^\star$.

Proof. (i) Let $i \in \{0, \ldots, k-1\}$; we have $x^k = x^i + \sum_{j=i}^{k-1} \alpha_j d_j$; thus, we get $Qx^k - c = Qx^i + \sum_{j=i}^{k-1} \alpha_j Q d_j - c$, i.e., $g_k = g_i + \sum_{j=i}^{k-1} \alpha_j Q d_j$. Therefore we can conclude that $g_k^T d_i = g_i^T d_i + \sum_{j=i}^{k-1} \alpha_j d_i^T Q d_j = g_i^T d_i + \alpha_i d_i^T Q d_i = g_i^T d_i - \frac{g_i^T d_i}{d_i^T Q d_i} d_i^T Q d_i = 0$.
(ii) Assume n iterations have been carried out; $g_n^T d_i = 0$ for all $i = 0, \ldots, n-1$. Since $d_0, \ldots, d_{n-1}$ are n linearly independent directions, we necessarily have $g_n = 0$. □

Now, the algorithmic issue with the method of conjugate directions is how to compute the set of directions. The *conjugate gradient* method comes here into play; let $g_k = \nabla f(x^k)$; let $d_0 = -g_0$ and then:

$$\alpha_k = \frac{-g_k^T d_k}{d_k^T Q d_k} = \frac{\|g_k\|^2}{d_k^T Q d_k}, \qquad \beta_{k+1} = \frac{\|g_{k+1}\|^2}{\|g_k\|^2}, \qquad d_{k+1} = -g_{k+1} + \beta_{k+1} d_k;$$

it can be shown that, at each iteration, the new direction d_{k+1} is mutually conjugate w.r.t. $d_0, \ldots, d_k$.

The scheme of the conjugate gradient method is reported in Algorithm 1. Clearly, recalling Proposition 1.5, the method is exact, but in high dimensional cases a stopping criterion $\|g_k\| \leq \varepsilon$ can be introduced, making it an iterative method, to reduce the computational time.

Algorithm 1: `Conjugate Gradient Method`

Input: $x^0 \in \mathbb{R}^n$, $d_0 = -g_0$.
$k = 0$;
while $\|g_k\| \neq 0$ **do**
  $\alpha_k = \frac{\|g_k\|^2}{d_k^T Q d_k}$;
  $x^{k+1} = x^k + \alpha_k d_k$;
  $g_{k+1} = g_k + \alpha_k Q d_k$;
  $\beta_{k+1} = \frac{\|g_{k+1}\|^2}{\|g_k\|^2}$;
  $d_{k+1} = -g_{k+1} + \beta_{k+1} d_k$;
  $k = k + 1$
Output: x^k

The conjugate gradient method can be extended to the general nonlinear case; in that case, the choice of the stepsize α_k has to be made by means of a line search procedure. As for β_{k+1}, many different update rules have been proposed. These update rules are based on different, equivalent ways of expressing, in the quadratic case, the value $\frac{\|g_{k+1}\|}{\|g_k\|^2}$. Of course, in the general case these expressions are no more equivalent, thus resulting in slightly different algorithmic schemes; the algorithm based on the update $\beta_{k+1} = \frac{\|g_{k+1}\|}{\|g_k\|^2}$ is referred to as Fletcher-Reeves conjugate gradient method and is in fact known as one of the most efficient realizations of the nonlinear conjugate gradient method.

The particular line search scheme used for selecting α_k is crucial to obtain convergent versions of the algorithm. In particular, the Armijo rule alone is not sufficient to guarantee that the direction $d_{k+1} = -g_{k+1} + \beta_{k+1} g_k$ will be a descent direction. In order to have a descent direction, the condition

$$d_{k+1}^T g_{k+1} = -\|g_{k+1}\|^2 + \beta_{k+1} g_k^T g_{k+1} < 0$$

must be guaranteed. Note that both $g_{k+1} = \nabla f(x^k + \alpha_k d_k)$ and $\beta_{k+1} = \frac{\|\nabla f(x^k + \alpha_k d_k)\|^2}{\|g_k\|^2}$ are indeed functions of α_k. It can be proved that selecting a stepsize α_k satisfying strong Wolfe conditions is enough to guarantee the descent property within the Fletcher-Reeves conjugate gradient algorithm.

1.5 Newton's method: extras

1.5.1 Example: Newton's method does not necessarily converges

Consider the problem

$$\min_{x \in \mathbb{R}} f(x) = \sqrt{1 + x^2}.$$

TThe problem has a quite regular objective function: it is strictly convex, coercive, twice continuously differentiable with

$$f'(x) = \frac{x}{\sqrt{1+x^2}}, \qquad f''(x) = \frac{1}{(1+x^2)^{3/2}}.$$

The iterations of the Newton's method applied to this problem take the form:

$$\begin{aligned} x^{k+1} &= x^k - \frac{f'(x^k)}{f''(x^k)} = x^k - \frac{x^k}{(1+(x^k)^2)^{1/2}}(1+(x^k)^2)^{3/2} \\ &= x^k - x^k(1+(x^k)^2) = -(x^k)^3. \end{aligned}$$

If we take $x^0 = 1$ or $x^0 = -1$, iterates bounce back and forth from 1 to -1, defining a sequence with two accumulation points, 1 and -1, neither being stationary; if on the other hand we take x^0 such that $|x^0| > 1$, it is easy to observe that $x^k = (-1)^k (x^0)^{3^k}$ is a divergent sequence. Hence, Newton's method does not globally converge to the minimum point, as convergence depends on the starting solution.

1.5.2 On the convergence rate and complexity of Newton's method

An interesting result is that, without further assumptions than convexity, Newton's method has the same complexity of $\mathcal{O}(\frac{1}{k^2})$ as Nesterov accelerated gradient method; this rate can be improved up to $\mathcal{O}(\frac{1}{k^3})$ only by introducing proper modifications to the procedure. Still, the rate remains sublinear. The superlinear convergence rate known for Newton's method holds for the strongly convex case. Indeed, if we are in the convex setting and assume that the Hessian matrix is invertible in neighborhood of a stationary solution, than we are basically assuming that, locally, the Hessian is positive definite. Hence, the function is locally strictly and strongly convex. In this case the iteration complexity is $\log(\log(\frac{1}{\epsilon}))$. Note that, while Newtons' method is very fast in terms of iterations, each single iteration has a high computational cost (Hessian matrix is computed and inverted).

1.6 BFGS

The BFGS algorithm is a Quasi-Newton method that operates a rank-2 update of the approximation of the inverse of the Hessian matrix. The iterations of the algorithm are of the form

$$x^{k+1} = x^k - \alpha_k B_k^{-1} \nabla f(x^k), \qquad B_k \approx \nabla^2 f(x^k) \qquad \text{(direct update)}$$

or

$$x^{k+1} = x^k - \alpha_k H_k \nabla f(x^k), \qquad H_k \approx [\nabla^2 f(x^k)]^{-1} \qquad \text{(inverse update)}$$

where B_k and H_k satisfies the Quasi-Newton equation

$$B_{k+1} s_k = y_k, \qquad H_{k+1} y_k = s_k,$$

where $y_k = \nabla f(x^{k+1}) - \nabla f(x^k)$ and $s_k = x^{k+1} - x^k$.

The BFGS update of the Quasi-Newton matrix, in the direct form, is given by

$$B_{k+1} = B_k + \frac{y_k y_k^T}{y_k^T s_k} - \frac{B_k s_k s_k^T B_k}{s_k^T B_k s_k},$$

whereas in the inverse case it is

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

Note that these update rules actually satisfiy the Quasi-newton property; indeed:

$$\begin{aligned} B_{k+1}s_k &= B_k s_k + \frac{y_k y_k^T}{y_k^T s_k} s_k - \frac{B_k s_k s_k^T B_k}{s_k^T B_k s_k} s_k \\ &= B_k s_k + y_k \frac{y_k^T s_k}{y_k^T s_k} - B_k s_k \frac{s_k^T B_k s_k}{s_k^T B_k s_k} \\ &= B_k s_k + y_k - B_k s_k = y_k, \end{aligned}$$

and similar calculations can be done for the inverse update rule.

Not only we need H_k (or B_k if we consider the direct update case) to satisfy the Quasi-Newton equation; we also need the direction $d_k = -H_k \nabla f(x^k)$ to be a descent direction at x^k, i.e., $\nabla f(x^k)^T d_k < 0$; we hence want $-\nabla f(x^k)^T H_k \nabla f(x^k) < 0$, which is guaranteed for positive definite H_k. This property can actually also be enforced by means of the BFGS rule.

Proposition 1.6. *Let H_k be positive definite and let H_{k+1} computed by the BFGS rule. Then H_{k+1} is positive definite if and only if $s_k^T y_k > 0$.*

The above proposition ensures that, if H_0 is positive definite and at each iteration $s_k^T y_k > 0$, the positive definiteness condition will continue to hold at each step. The condition $s_k^T y_k > 0$ can be imposed by properly setting the stepsize α_k; in fact, we have

$$s_k^T y_k = (\nabla f(x^{k+1}) - \nabla f(x^k))^T (x^{k+1} - x^k) > 0 \iff (\nabla f(x^{k+1}) - \nabla f(x^k))^T d_k > 0,$$

or in other words

$$\nabla f(x^{k+1})^T d_k > \nabla f(x^k)^T d_k,$$

which can be imposed by setting α_k by a Wolfe line search.

For the BFGS algorithm, results exist of global convergence under convexity hypotheses on the objective function f; moreover, if strong convexity is assumed, the method can be proved to superlinearly converge to the global minimizer.

1.7 L-BFGS

The L-BFGS algorithm is nothing but a BFGS modification specifically designed for the large scale setting (the L stands for Limited memory), but which actually shows strong performance in general settings and is widely considered the preferred choice for solving unconstrained nonlinear optimization problems.

The basic idea of the method is that, with high dimensional problems, storing the matrix H_k may be computationally prohibitive. In addition, the cost of computing the product $d_k = -H_k \nabla f(x^k)$ becomes unsustainable.

The matrix H_k can be obtained by a recursive rule requiring the only knowledge of vectors y_k and s_k:

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

where

$$\rho_k = \frac{1}{y_k^T s_k}, \qquad V_k = I - \rho_k y_k s_k^T.$$

The first key cog in L-BFGS consists in approximating H_{k+1} by stopping the recursive formula after m steps: only the last m pairs(y_k, s_k) are required together with a starting approximation of H_{k-m}. The approximation usually employed is $H_{k-m} \approx \gamma I$, which can be easily stored in memory. Experimental experience suggests that relatively small values of m (2 up to 20) are enough to obtain satisfactory approximations.

This formula can then be exploited to reduce the computation of $H_k \nabla f(x^k)$. The so-called HG recursive procedure allows to compute $d_k = -H_k \nabla f(x^k)$ by only performing $4mn$ multiplications, with the sole knowledge of (y_{k-i}, s_{k-i}), $i = 0, \ldots, m$, and the approximation of H_{k-m}.

As already discussed, the L-BFGS algorithm is really efficient in practice at solving unconstrained optimization problems. Unfortunately, the experimental strength is not coupled with strong theoretical properties: global convergence results to stationary points are not known for the algorithm, even when the most crucial safeguards are employed within the algorithm. Yet, it is possible to prove superlinear convergence in the strongly convex case under hypotheses on the sequence $\{H_k\}$.

As a final remark, we point out that an extension of the L-BFGS method, called L-BFGS-B, has been designed to address bound constrained optimization problems.

1.8 Decomposition Techniques

In both cases of constrained and unconstrained optimization, there are situations where it is useful splitting problems into smaller, easier subproblems. Cases where *decomposition techniques* are helpful include:

- The number n of variables is large ($\sim 10^4$).
- The problem can be decomposed into independent subproblems that can be solved with parallel computation techniques.
- A hard problem can be decomposed into easy subproblems.

A problem can be decomposed if either the objective function or the constraints are separable with respect to the variables. So, if we have a generic problem of the form

$$\min_{x \in X} f(x) \tag{5}$$

where $f : \mathbb{R}^n \to \mathbb{R}$ and $X \in \mathbb{R}^n$, a decomposition might consist of the problem

$$\min_{x_1 \in X_1, \ldots, x_m \in X_m} f(x_1, \ldots, x_m) \tag{6}$$

where $f : \mathbb{R}^{n_1} \times \ldots \times \mathbb{R}^{n_m} \to \mathbb{R}$, $X_1 \subseteq \mathbb{R}^{n_1}, \ldots, X_m \subseteq \mathbb{R}^{n_m}$ and $\sum_{i=1}^{m} n_i = n$.

There are plenty of techniques to generate and solve subproblems, once the blocks of variables have been defined; these algorithms can be grouped into two main classes: *sequential methods* and *parallel methods*. Subproblems are defined with respect to one block of variables.

Sequential methods solve subproblems one at a time and variables are updated at each step. An example of sequential method is *Gauss-Southwell* algorithm. Gauss-Seidel algorithm is defined by the following update rule:

$$x_i^{k+1} = \arg\min_{\xi_i \in X_i} f(x_1^{k+1}, \ldots, x_{i-1}^{k+1}, \xi_i, x_{i+1}^{k}, \ldots, x_m^{k}).$$

The algorithm simply cyclically solves the subproblems and exploits the solution to immediately update the respective block of variables.

Note however that the algorithm requires to compute a global optimizer for each subproblem: this is a strong requirement, that might be not easily satisfied.

Convergence properties of Gauss-Seidel method can be stated if the objective function is convex, or if it is component-wise strictly convex, or without assumptions on the objective function in the two-blocks case ($m = 2$).

Parallel methods solve independently and at the same time all the subproblems generated by each different block, but only the best one among these solutions is used for the variables update, i.e., at each iteration only one block of variables is updated. An example of parallel method is *Jacobi* algorithm:

$$\hat{x}_i^{k+1} \in \arg\min_{x_i} f(x_1^k, \ldots, x_i, \ldots, x_m^k), \qquad x^{k+1} \in \arg\min_{(x_1^k, \ldots, \hat{x}_i^{k+1}, \ldots, x_m^k)} f(x_1^k, \ldots, \hat{x}_i^{k+1}, \ldots, x_m^k).$$

1.8.1 Decomposition Methods with Blocks Overlapping

Up to this point, we considered decomposition approaches where $x \in \mathbb{R}^n$ is partitioned into m blocks that do not vary with the iterations: $x^k = (x_1^k, \ldots, x_m^k)$.

We now present a more flexible scheme: we introduce the notion of *working set* $W_k \subset \{1, \ldots, n\}$. At each iteration we consider the subproblem

$$\min_{x_{W_k}} \; f(x_{W_k}, x^k_{\overline{W}^k}) \tag{7}$$

where $W^K \subset \{1, \ldots, n\}$, $\overline{W}^K = \{1, \ldots, n\} \setminus W_k$ and $x^k = (x^k_{W_k}, x^k_{\overline{W}^k})$. Thus, if $x^\star_{W_k}$ is the solution of problem (7) at the k-th iteration, we have the following update rule:

$$x_i^{k+1} = \begin{cases} x_i^\star & \text{if } i \in W_k, \\ x_i^k & \text{otherwise.} \end{cases} \tag{8}$$

Global convergence of this scheme depends on the *working set selection rule*. Possible selection rules ensuring convergence include:

- *Cyclic Rule*: $\exists\, M > 0$ s.t. $\forall\, i \in \{1, \ldots, n\}$, $\forall\, k\ \exists\, \ell(k) \leq M$ s.t. $i \in W^{k+\ell(k)}$.
- *Gauss-Southwell Rule*: $\forall k$, $i(k) \in W_k$ if $|\nabla_{i(k)} f(x^k)| \geq |\nabla_i f(x^k)|\ \forall\, i \in \{1, \ldots, n\}$.

1.9 Stochastic Gradient Method for Finite-Sum Problems

A particularly relevant class of unconstrained nonlinear optimization problems is that of *finite-sum problems*, i.e.,

$$\min_{x \in \mathbb{R}^n} \; f(x) = \frac{1}{N} \sum_{i=1}^{N} f_i(x).$$

Stochastic Gradient Descent (SGD) is the prototypical stochastic optimization method, dating back to 1950s, designed to tackle this class of problems. The main idea of stochastic optimization of finite-sum problems is to update the variables at each iteration by descent steps, approximating the true gradient of the function $\nabla f(x)$ by the direction

$$\frac{1}{|B|} \sum_{i \in B} \nabla f_i(x), \tag{9}$$

where $B \subset \{1, \ldots, n\}$ is a subset of objective function terms. We thus have updates of the form

$$x^{k+1} = x^k - \alpha \frac{1}{|B|} \sum_{i \in B} \nabla f_i(x^k), \tag{10}$$

where the step size α is usually set to a constant, or at most follows a predefined sequence of values.

This approximation somehow alleviates the burden of the gradient computation, which might be particularly expensive in some applications where the number N of terms ∇f_i is very high and/or each one is expensive to compute. Indeed, at every iteration k we are only considering a small number of terms, indexed by B. In the context of machine and statistical learning, each term f_i corresponds to a sample; a subset B of examples is referred to as *minibatch*, opposed to the *full batch*, i.e., the entire data set (in this context, classical descent methods exploiting the information about the entire objective function are referred to as *batch optimizers*). Moreover, in the particular context of deep learning, the stepsize is commonly referred to as *learning rate*.

The most basic version of SGD dictates to use minibatches of size 1. In practice, it is computationally (and theoretically) more useful to use minibatches of M examples, with

M some orders of magnitude smaller than N. We will discuss this aspect in detail when talking about deep learning problems. For the moment, we will consider for the sake of simplicity the case $M = 1$.

At each step, a random index is uniformly sampled from $\{1, \dots, N\}$ to approximate gradients by

$$\nabla f(x^k) \approx \nabla f_i(x).$$

If we look at the expected value of this quantity, recalling that for a uniform distribution over the N values $\{1, \dots, N\}$ we have $p_i = \frac{1}{N}$, we get

$$\mathbb{E}[\nabla f_i(x^k)] = \sum_{i=1}^{N} p_i \nabla f_i(x^k) = \sum_{i=1}^{N} \frac{1}{N} \nabla f_i(x^k) = \frac{1}{N} \sum_{i=1}^{N} \nabla f_i(x^k) = \nabla f(x^k);$$

in other words, on average we expect to get the true gradient.

1.9.1 Theoretical Analysis of SGD

In this section, we carry out a simple convergence analysis for the basic SGD method, where a single, random sample is selected to approximate the gradient at each iteration; in other words, the algorithm we are going to analyze performs steps of the form

$$x^{k+1} = x^k - \alpha_k \nabla f_{i_k}(x^k). \tag{11}$$

Unlike gradient descent, SGD does not necessarily decrease the value of the objective function at each step. In order to rigorously study the algorithm, we first need some new assumption that characterizes how far the gradient samples can be from the true gradient. Assume that f is bounded below and that, for some constant $G > 0$, the magnitude of gradient samples are bounded, for all $x \in \mathbb{R}^n$, by

$$\|\nabla f_i(x)\| \leq G.$$

We will also assume that the objective function is twice continuously differentiable and has Lipschitz continuous gradients. Recall that, for a twice continuously differentiable functions, Lipschitz continuity is equivalent to the following condition: there exists some constant $L > 0$ such that, for all x in the space and for any vector $u \in \mathbb{R}^n$,

$$\left|u^T \nabla^2 f(x) u\right| \leq L\|u\|^2.$$

Proposition 1.7. *Let $\{x^k\}$ be the sequence generated by the SGD algorithm* (11) *with a stepsize sequence $\{\alpha_k\}$ satisfying*

$$\sum_{k=0}^{\infty} \alpha_k = \infty, \qquad \sum_{k=0}^{\infty} \alpha_k^2 < \infty.$$

Further assume that, at each iteration k, the algorithm would randomly output $z^k = x^\tau$ with probability

$$\mathbb{P}(\tau = t) = \frac{\alpha_t}{\sum_{i=0}^{k-1} \alpha_i}$$

for $t = 0, \dots, k-1$. Then,

$$\lim_{k \to \infty} E\left[\left\|\nabla f(z^k)\right\|^2\right] = 0.$$

Proof. Let k be any iteration. From Taylor's theorem, we know there exists $y_k \in \mathbb{R}^m$ such that

$$\begin{aligned} f(x^{k+1}) &= f(x^k - \alpha_k \nabla f_{i_k}(x^k)) \\ &= f(x^k) - \alpha_k \nabla f_{i_k}(x^k)^T \nabla f(x^k) + \frac{\alpha_k^2}{2} \nabla f_{i_k}(x^k)^T \nabla^2 f(y_k) \nabla f_{i_k}(x^k) \\ &\leq f(x^k) - \alpha_k \nabla f_{i_k}(x^k)^T \nabla f(x^k) + \frac{\alpha_k^2 L}{2} \|\nabla f_{i_k}(x^k)\|^2 \\ &\leq f(x^k) - \alpha_k \nabla f_{i_k}(x^k)^T \nabla f(x^k) + \frac{\alpha_k^2 L G^2}{2}. \end{aligned}$$

Now, the term $\alpha_k \nabla f_{i_k}(x^k)^T \nabla f(x^k)$ is not necessarily nonnegative, we are not necessarily making any progress in the objective function. We shall then see what happens in expectation:

$$\begin{aligned} E\left[f(x^{k+1})\right] &\leq E\left[f(x^k) - \alpha_k \nabla f_{i_k}(x^k)^T \nabla f(x^k) + \frac{\alpha_k^2 L G^2}{2}\right] \\ &= E\left[f(x^k)\right] - \alpha_k E\left[\nabla f_{i_k}(x^k)^T \nabla f(x^k)\right] + \frac{\alpha_k^2 L G^2}{2}. \end{aligned}$$

Now, the expected value of $\nabla f_{i_k}(x^k)$ given x^k is

$$E\left[\nabla f_{i_k}(x^k) \mid x^k\right] = \sum_{i=1}^{n} \nabla f_i(x^k) \cdot \mathbb{P}(i_k = i \mid x^k) = \sum_{i=1}^{n} \nabla f_i(x^k) \cdot \frac{1}{n} = \nabla f(x^k),$$

so [1]

$$E\left[f(x^{k+1})\right] \leq E\left[f(x^k)\right] - \alpha_k E\left[\left\|\nabla f(x^k)\right\|^2\right] + \frac{\alpha_k^2 L G^2}{2}.$$

Applying recursively the above inequality and noting that $E[f(x^0)] = f(x^0)$, we get

$$E\left[f(x^{k+1})\right] - f(x^0) \leq -\sum_{t=0}^{k} \alpha_t E\left[\left\|\nabla f(x^t)\right\|^2\right] + \frac{LG^2}{2} \sum_{t=0}^{k} \alpha_t^2,$$

i.e.,

$$\begin{aligned} \sum_{t=0}^{k} \alpha_t E\left[\left\|\nabla f(x^t)\right\|^2\right] &\leq f(x^0) - E\left[f(x^{k+1})\right] + \frac{LG^2}{2} \sum_{t=0}^{k} \alpha_t^2 \\ &\leq f(x^0) - f^\star + \frac{LG^2}{2} \sum_{t=0}^{k} \alpha_t^2, \end{aligned}$$

where $f^\star$ is the (finite) global optimum of f. Now, let us consider the expected value of the gradient at the "output" solution z^{k+1}:

$$\begin{aligned} E\left[\left\|\nabla f(z^{k+1})\right\|^2\right] &= \sum_{t=0}^{k} E\left[\left\|\nabla f(x^t)\right\|^2\right] \cdot \mathbb{P}(z^{k+1} = x^t) \\ &= \sum_{t=0}^{k} E\left[\left\|\nabla f(x^t)\right\|^2\right] \cdot \frac{\alpha_t}{\sum_{i=0}^{k} \alpha_i} \\ &= \frac{1}{\sum_{i=0}^{k} \alpha_i} \sum_{t=0}^{k} \alpha_t E\left[\left\|\nabla f(x^t)\right\|^2\right]. \end{aligned}$$

[1] exploiting the law of iterated expextation: $E[X] = E_Y[E_X[X|Y]]$

We hence have

$$E\left[\left\|\nabla f(z^{k+1})\right\|^2\right] \leq \frac{1}{\sum_{i=0}^{k}\alpha_i}\left(f(x^0) - f^\star + \frac{LG^2}{2}\sum_{t=0}^{k}\alpha_t^2\right).$$

Taking the limits for $k \to \infty$, recalling that $\sum \alpha_t = \infty$ and $\sum \alpha_t^2 < \infty$, we get

$$\lim_{k\to\infty} E\left[\left\|\nabla f(z^{k+1})\right\|^2\right] = 0.$$

□

A step size schedule that ensures convergence in expectation to stationary points for the SGD algorithm is given by the following rule:

$$\alpha_k = \frac{\alpha_0}{k+1}.$$

Basically, steps shall go to zero, but "slowly" enough to allow the algorithm reach a stationary point.

As for the complexity of the algorithm, the speed of convergence is lower than that of full-batch methods: SGD has $\mathcal{O}(\frac{1}{\epsilon^2})$ iteration complexity in the convex case and $\mathcal{O}(\frac{1}{\epsilon})$ in the strongly convex case (i.e., sublinear convergence, whereas batch GD has linear convergence). Moreover, acceleration does not improve the rate. However, as opposed to batch GD, SGD does not hide within the complexity constants the number N of the summation terms of the objective functions. This is one of the main reasons why minibatch GD ($1 < |B| = M \ll N$), representing the middle way between batch and stochastic GD, is in practice the most effective approach in applications. Thus, in practice, optimization can be carried out by macro-iterations, referred to as *epochs*. At each epoch the index set $\{1, \ldots, N\}$ is randomly split into minibatches of size M. N/M updates of the form (10) are then performed, one for each minibatch. Note that the random splits defining the minibatches are different at every epoch.

2 Optimization Problems in Machine Learning: Basics

2.1 Introduction

Machine learning is not optimization. Machine learning is a branch of artificial intelligence techniques, that proved to be highly successful on numerous tasks in recent years; its success is arguably attributable to the strong statistical properties possessed by learning models, that allow to properly make use of the large amount of data available nowadays; moreover, the immense ongoing advance of hardware and software tools played a crucial role in making learning systems actually, effectively employable.

Still, *mathematical optimization* is a core cog for this technology: the "engine" metaphor is often used. Indeed, the training process of a learning system consists, for the vast majority of models, in the solution of an optimization problem. Many machine learning libraries have become popular and widely employed in recent years; in each of them, hitting the "run button" does nothing else than staring an optimization algorithm.

For this reason, it is extremely important for machine learning experts and engineers to know in detail the mechanisms within these processes; this is especially true in order to be able to interpret bad behaviors and fix issues that frequently occur when designing and implementing learning systems.

Throughout the course, we will focus on the most relevant optimization algorithms employed with *supervised learning* tasks. Thus, we have a dataset

$$\mathcal{D} = \left\{(x^{(i)}, y^{(i)}) \mid x^{(i)} \in \mathcal{X},\ y^{(i)} \in \mathcal{Y},\ i = 1, \ldots, n\right\},$$

where $\mathcal{X} \subseteq \mathbb{R}^p$ and either $\mathcal{Y} = \mathbb{R}$ (*regression tasks*) or $\mathcal{Y} = \{0, 1\}$ (*binary classification tasks*). The dataset represents a sampling from some distribution, where some relation f exists between pairs (x, y), such that $f(x) = y$.

The aim in machine learning is to construct, based on the pairs in $\mathcal{D}$, a function $\hat{f}$ that captures the essence of f, being able to accurately provide values of $\hat{y} = \hat{f}(x)$ for point x that are not present in the training set $\mathcal{D}$.

Training is typically modeled as an optimization problem (*empirical risk minimization*), where a *loss function* has to be minimized w.r.t. the parameters w of the model f. The usual form of training optimization problems is

$$\min_{w} \; L(w) = \frac{1}{n} \sum_{i=1}^{n} \ell(f(x^{(i)}; w), y^{(i)}), \tag{12}$$

i.e., we want to minimize the finite sum of terms that can have different forms, depending on the specific loss employed; examples of loss functions are

- **Square loss:** $\ell(u, v) = (u - v)^2$ (regression);
- **ℓ_1 loss:** $\ell(u, v) = |u - v|$ (regression);
- **Log loss:** $\ell(u, v) = -(u \log(v) - (1 - u) \log(1 - v))$ (binary classification);
- **0-1 loss:** $\ell(u, v) = 1 - \mathbb{1}\{u = v\}$ (binary classification);
- **Hinge loss:** $\ell(u, v) = \max\{0, 1 - uv\}$ (binary classification).

Now, being able to effectively solve the optimization problem (12) is not sufficient to guarantee that the resulting model will work well on out-of-sample data; two unfortunate situations often occur:

- if data quality is poor, or if the model is too simple compared to data distribution, a good approximation of the "true" f cannot be identified even if the optimization problem is accurately solved; this problem is referred to as *underfitting* and is not addressable with mathematical optimization tools alone;
- the second problem is somewhat the converse of the first one and occurs when the optimization problem is solved "too well"; in fact, a surrogate objective function is used in problem (12): the error on the training set is minimized, whereas we would like to minimize the error on the entire data distribution, including unseen data; if the learning model is sufficiently expressive (which is often the case with large *parametric models*) a very complicated prediction function might be obtained, perfectly tailored for data in $\mathcal{D}$, but absolutely incorrect with unseen data. This issue is referred to as *overfitting*.

In order to partially mitigate the latter problem, a regularization term is usually introduced in the training problem to enhance the generalization ability of the learning model. The resulting optimization problem is given by

$$\min_{w} \; L(w) + \Omega(w), \tag{13}$$

where the regularization term $\Omega(w)$ is usually set as one among:

- $\Omega(w) = \|w\|_2^2$ (quadratic regularization);
- $\Omega(w) = \|w\|_1$ (ℓ_1 regularization);
- $\Omega(w) = \|w\|_0$ (ℓ_0 regularization; $\|w\|_0 = |\{i : w_i \neq 0\}|$).

The quadratic regularizer is the most often employed for its simplicity and its regularity properties. The ℓ_1 and ℓ_0 regularizers are sparsity-inducing penalties, the former one having much stronger regularity properties than the latter one. Sparsity is often a valuable characteristic in predictive models.

From now on, we will not focus on the statistical details of learning models, but we will only consider the pure mathematical optimization point of view.

As a matter of fact, we shall note that adding a quadratic regularization term into the optimization problem not only has a statistical value, improving the generalization properties of the trained model, but it also turns convex objective functions into *strongly convex* functions. Keeping into account the discussion on computational complexity of optimization algorithms in Section 1, we can point out that regularization should also significantly speed up the optimization process.

2.2 Linear Regression

The simplest model for regression tasks is the *linear regression* one. Linear regressors are usually obtained by solving a (regularized) *least squares problem*:

$$\min_{w \in \mathbb{R}^p} \|Aw - b\|^2 + \lambda \|w\|^2 \tag{14}$$

where $A \in \mathbb{R}^{n \times p}$ and $b \in \mathbb{R}^n$. The problem is convex and is thus equivalent to finding a solution with zero gradients. Letting

$$f(w) = \|Aw - b\|^2 + \lambda \|w\|^2 = w^T A^T A w - 2 w^T A^T b + \|b\|^2 + \lambda w^T w$$

and

$$\nabla f(w) = 2A^T A w - 2A^T b + 2\lambda w,$$

the problem is hence equivalent to solving the following linear system of equations, usually referred to as *normal equations*:

$$(A^T A + \lambda I) w = A^T b. \tag{15}$$

Proposition 2.1. *Problem* (14) *admits a unique optimal solution.*

Proof. The objective function is coercive:

$$\lim_{\|w\| \to \infty} \|Aw - b\|^2 + \lambda \|w\|^2 \geq \lim_{\|w\| \to \infty} \lambda \|w\|^2 = +\infty.$$

Hence, by Weierstrass theorem, the problem admits solution; the Hessian matrix of the objective function is given by

$$\nabla^2 f(w) = 2A^T A + 2\lambda I,$$

which is positive definite; indeed, for any $w \neq 0$, we have

$$2 w^T A^T A w + 2 w^T (\lambda I) w = 2\|Aw\|^2 + 2\lambda \|w\|^2 \geq \lambda \|w\|^2 > 0.$$

Therefore, $f(w)$ is strictly convex and the (global) minimizer is unique. □

The system of equations (15) has a unique solution, that can be computed:

- in closed form, by matrix inversion: $w^\star = (A^T A + \lambda I)^{-1} A^T b$; this approach can be used if p is relatively small (the cost of matrix inversion is $\mathcal{O}(p^3)$) and if A is not ill-conditioned;
- using an iterative method, such as gradient descent or Newton's method; in fact, the *conjugate gradient* method is often employed with linear systems.

2.2.1 Regularization-free case

The problem

$$\min_{w \in \mathbb{R}^p} \|Aw - b\|^2 \tag{16}$$

has similar properties as its ℓ_2-regularized counterpart and can be solved accordingly, but showing that the solution of the problem always exists is a little bit trickier; moreover, the solution is not always unique, if the rank of A is not guaranteed to be full; if A is not full-rank, the Hessian matrix A^TA is not strictly positive-definite and f is not necessarily coercive nor strictly convex.

Proposition 2.2. *Problem* (16) *always admits solution.*

Proof. Let us consider the problem

$$\begin{aligned} \min_z \ & \frac{1}{2}\|b - z\|^2 \\ \text{s.t.} \ & A^T z = 0. \end{aligned}$$

The objective function of the problem is coercive and the feasible set is closed, hence the problem admits a solution, that is unique being the objective function strictly convex; since the objective is quadratic and the constraints linear, KKTs are necessary and sufficient conditions of optimality: $\exists \mu^\star \in \mathbb{R}^p$ such that

$$\nabla_z L(z^\star, \mu^\star) = -(b - z^\star) + A\mu^\star = 0, \qquad \text{with } A^T z^\star = 0.$$

Hence, $b = z^\star + A\mu^\star$, with $z^\star : A^T z^\star = 0$ and $\mu^\star \in \mathbb{R}^p$. We have retrieved a basic result from geometry:

$$b = b_R + b_N,$$

$$b_R = A\mu^\star \in Im(A) \quad (\exists y : Ay = b_R), \qquad b_N = z^\star \in Ker(A^T) \quad (A^T b_N = 0).$$

Now, we can observe that

$$A^T b = (A^T z^\star + A^T A\mu^\star) = A^T A\mu^\star,$$

i.e., $b_R = A\mu^\star$ is solution of the normal equations. □

2.3 Logistic Regression

In this section, we address the problem of fitting a logistic regression model; the optimization problem for this setting has the form

$$\min_{w \in \mathbb{R}^p} \mathcal{L}(w) + \lambda\Omega(w), \tag{17}$$

where $\mathcal{L}(w)$ is the negative log-likelihood function of the logistic model, which is a convex function, and $\Omega(w)$ is a convex regularizer. Note that this setting is conceptually equivalent to other training problems with convex loss functions, such as softmax regression or ARMA models fitting in time series.

For problems of this form, an iterative algorithm is required to train the model; methods such as gradient descent or Newton's method are all viable options; however, the L-BFGS algorithm is generally considered the most efficient method for unconstrained optimization, both in the convex case and in the nonconvex case when global optimality is not actually required, with problems up to a considerably large size.

Here we show how the gradients and the Hessian matrix for the logistic regression problem can be computed. We first recall that, letting $\mathcal{Y} = \{-1, 1\}$ the loss function has the form

$$\mathcal{L}(w; X, y) = \sum_{i=1}^{n} \log(1 + \exp(-y^{(i)} w^T x^{(i)})).$$

If we set $z = Xw$ (i.e., $z_i = w^T x^{(i)}$ for all i), we have

$$\mathcal{L}(w; X, y) = \phi(z; y) = \sum_{i=1}^{n} \log(1 + \exp(-y^{(i)} z_i)).$$

We want to compute $\nabla_w \mathcal{L}(w; X, y)$; by the multivariate chain rule, we have

$$\nabla_w \mathcal{L}(w; X, y)^T = \nabla_z \phi(Xw; y)^T \frac{\partial}{\partial w}(Xw).$$

It is also easy to observe that

$$\begin{aligned}\frac{\partial}{\partial z_i}\phi(z; y) &= \frac{\partial}{\partial z_i}(\log(1 + \exp(-y^{(i)} z_i))) \\ &= \frac{1}{1 + \exp(-y^{(i)} z_i)} \exp(-y^{(i)} z_i)(-y^{(i)}) \\ &= -y^{(i)} \frac{1}{1 + \exp(y^{(i)} z_i)} = -y^{(i)} \sigma(-y^{(i)} z_i),\end{aligned}$$

where $\sigma(\cdot)$ is the sigmoid function. Hence,

$$\nabla_z \phi(z; y) = (-y^{(1)} \sigma(-y^{(1)} z_1), \ldots, -y^{(n)} \sigma(-y^{(n)} z_n))^T.$$

On the other hand, we have

$$\frac{\partial}{\partial w}(Xw) = X,$$

therefore

$$\nabla_w \mathcal{L}(w; X, y) = (r^T X)^T = X^T r$$

with $r \in \mathbb{R}^n$, $r_i = -y^{(i)} \sigma(-y^{(i)} w^T x^{(i)})$ for all $i = 1, \ldots, n$.

By similar calculations we can also get

$$\nabla^2 \mathcal{L}(w; X, y) = X^T D X,$$

where D is a diagonal matrix with $d_{ii} = \sigma(y^{(i)} w^T x^{(i)}) \sigma(-y^{(i)} w^T x^{(i)})$.

3 Support Vector Machines

A (linear) Support Vector Machine (SVM) model is, in brief, the classification model obtained solving the empirical risk minimization problem with the hinge loss and the ℓ_2 regularizer, i.e.,

$$\min_{w,b} \ \frac{1}{2}\|w\|^2 + C \sum_{i=1}^{n} \max\{0, 1 - y^{(i)}(w^T x^{(i)} + b)\}.$$

It is easy to realize that the optimal solutions of the above nonsmooth unconstrained optimization problem are also solution of the following smooth problem with linear constraints:

$$\begin{aligned}\min_{w,b,\xi} \ & \frac{1}{2}\|w\|^2 + C \sum_{i=1}^{n} \xi^{(i)} \\ \text{s.t.} \ & y^{(i)}(w^T x^{(i)} + b) \geq 1 - \xi^{(i)}, \\ & \xi^{(i)} \geq 0.\end{aligned} \tag{18}$$

It can be shown that, when $C = \infty$, the problem amounts to finding, among the hyperplanes perfectly separating the training data, the one maximizing the distance from the closest point (of both classes), see Figures 2 and 3. Of course, however, the problem has no feasible solution in that case if data are not linearly separable (Figure 4).

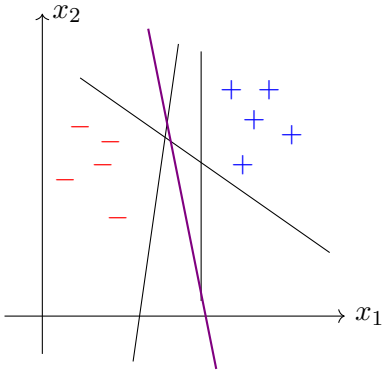


Figure 2: Classification through separating hyperplanes. Each one of the hyperplanes in figure solves correctly the classification task of training data. SVM selects amongst them the one which is equidistant from the two classes (the purple one).

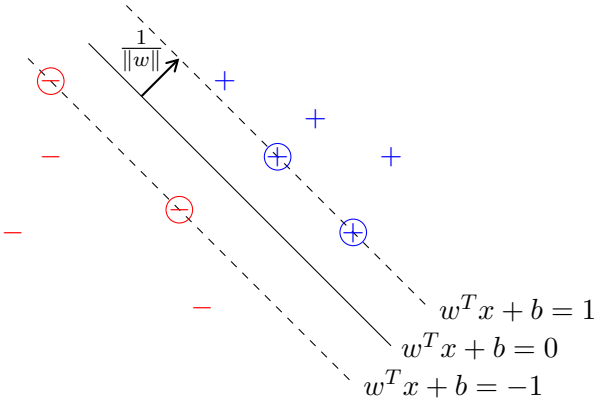


Figure 3: Maximum margin hyperplane. The circled points, that define the separation margin, are called support vectors.

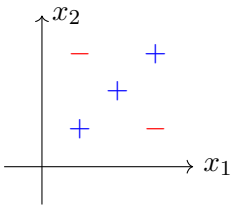


Figure 4: Example of a non linearly separable dataset.

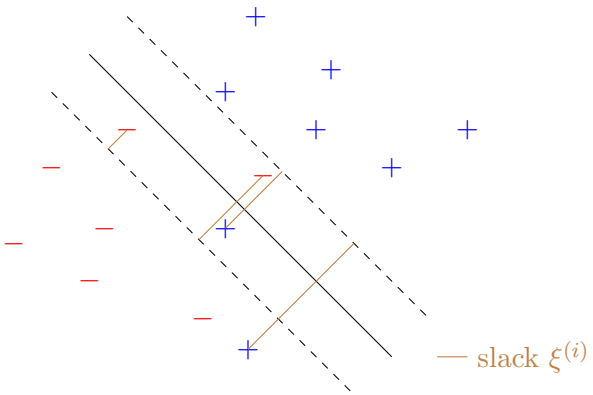


Figure 5: SVM classifier obtained allowing non null slacks.

Hence, with a finite choice of C, we accept to "pay a cost" for all points that are classified incorrectly, and even for those classified correctly but with an insufficient confidence (Figure 5). Not only this allows to guarantee the existence of a solution, but it is also useful to avoid overfitting issues.

Now, we turn to the study of the optimization problem (18). The problem is convex quadratic with linear constraints, thus it is in principle solvable by standard constrained solvers, and in particular the Frank-Wolfe algorithm. However, the number of constraints is proportional to the number of training data points, and might grow large quite fast. For this reason (and also others that we will address later), a different path shall be followed.

3.1 The dual problem

As aforementioned, problem (18) is (strictly) convex quadratic with linear constraints. Thus, KKTs are necessary and sufficient conditions of optimality. Now, without delving deep into an important but vast and complex topic, we introduce a result from *duality theory* concerning constrained convex problems.

Proposition 3.1. *Let us consider the optimization problem*

$$\begin{aligned} &\min_x f(x) \\ &s.t.\ g(x) \leq 0, \end{aligned}$$

being f and g continuously differentiable convex functions. Let $x^\star$ be an optimal solution for the problem and assume $\mu^\star$ is a vector of multipliers such that $(x^\star, \mu^\star)$ satisfies the KKT conditions. Then $(x^\star, \mu^\star)$ is an optimal solution of the so called Wolfe dual problem.

$$\begin{aligned} &\max_{x,\mu} \mathcal{L}(x,\mu) = f(x) + \mu^T g(x) \\ &s.t.\ \nabla_x \mathcal{L}(x,\mu) = 0, \\ &\quad \mu \geq 0. \end{aligned}$$

Proof. The pair $(x^\star, \mu^\star)$ satisfies KKT conditions, i.e.,

$$\nabla_x \mathcal{L}(x^\star, \mu^\star) = 0, \quad \mu^\star \geq 0, \quad g(x^\star) \leq 0, \quad \mu_i^\star g_i(x^\star) = 0 \; \forall\, i.$$

Thus, $(x^\star, \mu^\star)$ satisfies the constraints of the Wolfe dual problem. Moreover, by the complementarity condition, $\mathcal{L}(x^\star, \mu^\star) = f(x^\star) + \sum_i \mu_i^\star g_i(x^\star) = f(x^\star)$. Now, let (x, μ) be any feasible solution for the dual problem. Since $\mu \geq 0$ and $g(x^\star) \leq 0$, we get

$$\mathcal{L}(x^\star, \mu^\star) = f(x^\star) \geq f(x^\star) + \sum_i \mu_i g_i(x^\star) = \mathcal{L}(x^\star, \mu).$$

Then, by the convexity of $\mathcal{L}(x, \mu)$ w.r.t. x variables (it is the positive linear combination of the convex functions $f, g_1, \ldots, g_m$), we can also write

$$\mathcal{L}(x^\star, \mu) \geq \mathcal{L}(x, \mu) + \nabla_x \mathcal{L}(x, \mu)^T (x^\star - x).$$

Putting the pieces together and recalling that $\nabla_x \mathcal{L}(x, \mu) = 0$ being (x, μ) feasible for the dual problem, we obtain

$$\mathcal{L}(x^\star, \mu^\star) \geq \mathcal{L}(x^\star, \mu) \geq \mathcal{L}(x, \mu).$$

Being (x, μ) an arbitrary feasible solution of the dual problem, we get the thesis. □

We can now go back to the SVM problem (18). Being KKTs necessary and sufficient conditions for global optimality, for the (unique) optimal solution $(w^\star, b^\star, \xi^\star)$ there have

to exist multipliers $(\alpha^\star, \mu^\star)$ such that $(w^\star, b^\star, \xi^\star, \alpha^\star, \mu^\star)$ is a KKT point and, by the above proposition, is the solution of the dual problem

$$\max_{w,b,\xi,\alpha,\mu} \mathcal{L}(w,b,\xi,\alpha,\mu) = \frac{1}{2}\|w\|^2 + C\sum_i \xi_i + \sum_i -\mu\xi_i + \sum_i \alpha_i(1 - y^{(i)}(w^T x^{(i)} + b) - \xi_i)$$

$$\begin{aligned} \text{s.t. } & \alpha \geq 0, \quad \mu \geq 0, \\ & \nabla_w \mathcal{L}(w,b,\xi,\alpha,\mu) = 0, \\ & \nabla_b \mathcal{L}(w,b,\xi,\alpha,\mu) = 0, \\ & \nabla_\xi \mathcal{L}(w,b,\xi,\alpha,\mu) = 0. \end{aligned}$$

We can then observe that the constraints imply:

$$0 = \nabla_w \mathcal{L}(w,b,\xi,\alpha,\mu) = w - \sum_i \alpha_i y^{(i)} x^{(i)}, \quad \text{i.e.,} \quad w = \sum_i \alpha_i y^{(i)} x^{(i)},$$

$$0 = \nabla_b \mathcal{L}(w,b,\xi,\alpha,\mu) = -\sum_i \alpha_i y^{(i)}, \quad \text{i.e.,} \quad \alpha^T y = 0$$

$$0 = \nabla_\xi \mathcal{L}(w,b,\xi,\alpha,\mu) = Ce - \sum_i \mu_i e_i - \sum_i \alpha_i e_i, \quad \text{i.e.,} \quad \alpha_i = C - \mu_i \leq C \; \forall\, i.$$

Manipulating the objective function, we can first obtain

$$\begin{aligned} &\frac{1}{2} w^T w + C\sum_i \xi_i + \sum_i -\mu_i\xi_i + \sum_i \alpha_i + \\ &- \sum_i \alpha_i y^{(i)} w^T x^{(i)} - b\sum_i \alpha_i y^{(i)} - \sum_i \alpha_i \xi_i, \end{aligned}$$

then we shall recall that $\sum_i \alpha_i y^{(i)} = 0$ by the second condition above, whereas using the third one we can write $C\sum_i \xi_i - \sum_i \mu_i\xi_i = \sum_i \xi_i(C - \mu_i) = \sum_i \xi_i\alpha_i$. We therefore get

$$\frac{1}{2} w^T w + \sum_i \alpha_i\xi_i + \sum_i \alpha_i - \sum_i \alpha_i y^{(i)} w^T x^{(i)} - \sum_i \alpha_i\xi_i,$$

i.e.,

$$w^T \left(\frac{1}{2} w - \sum_i \alpha_i y^{(i)} x^{(i)} \right) + \sum_i \alpha_i.$$

Now, we can substitute $w = \sum_i \alpha_i y^{(i)} x^{(i)}$ to finally obtain that

$$\mathcal{L}(w,b,\xi,\alpha,\mu) = -\frac{1}{2}\sum_i \sum_j \alpha_i\alpha_j y^{(i)} y^{(j)} {x^{(i)}}^T x^{(j)} + \sum_i \alpha_i$$

for any feasible solution of the Wolfe dual problem. In other words, the objective is only function of multipliers α, that are solely constrained by $y^T\alpha = 0$, $\alpha \geq 0$ and $\alpha \leq C$.

Swapping the signs and turning to matrix notation, after defining the matrix Q such that $Q_{ij} = y^{(i)} y^{(j)} {x^{(i)}}^T x^{(j)}$, we end up with the dual problem

$$\begin{aligned} \min_\alpha \; & \frac{1}{2}\alpha^T Q\alpha - e^T\alpha \\ \text{s.t. } & \alpha^T y = 0, \\ & 0 \leq \alpha_i \leq C \quad \forall\, i. \end{aligned} \tag{19}$$

Once problem (19) has been solved, we can use the solution $\alpha^\star$ to retrieve the other variables:

$$w^\star = \sum_i \alpha_i^\star y^{(i)} x^{(i)}, \quad \mu_i^\star = C - \alpha_i^\star \; \forall\, i,$$

$$b^\star = \frac{y^{(i)}}{{w^\star}^T x^{(i)}} \text{ for any } i \text{ s.t. } \alpha_i^\star \in (0, C), \quad \xi_i^\star = \max\{0, 1 - y^{(i)}({w^\star}^T x^{(i)} + b^\star)\},$$

where the third equation (for $b^\star$) follows from imposing the complementarity slackness conditions form KKTs.

We shall note that, by the complementarity conditions, we have

$$\alpha_i^\star(1 - y^{(i)}(w^{\star T}x^{(i)} + b^\star) - \xi_i^\star) = 0 \text{ and } 0 = \mu_i^\star\xi^\star = (C - \alpha_i^\star)\xi_i^\star.$$

Thus, if $\alpha_i^\star \in (0, C)$, we have $\xi_i^\star = 0$ and $y^{(i)}(w^{\star T}x^{(i)} + b^\star) = 1$, i.e., the i-th data sample exactly lies on the border of the separation margin; on the other hand, if $\alpha_i^\star < C$, then $\xi_i^\star = 0$: the samples for which we pay a penalty are only those associated with multipliers $\alpha_i^\star$ with value at the upper bound. All the points associated with nonzero multipliers $\alpha_i^\star$ are called support vectors; we can observe that, since $w^\star = \sum_i \alpha_i^\star y^{(i)}x^{(i)}$, the resulting classifier is only based on support vectors.

When classifying a new data point x, what we actually do is computing

$$w^{\star T}x = \sum_i \alpha_i y^{(i)}x^T x^{(i)};$$

in other words, the outcome of the decision function is a weighted sum over training data, where only support vectors are actually taken into account; for each support vector, we carry out the dot product with the point to be classified: the higher the dot product is, the higher is the similarity between $x^{(i)}$ and x, the larger will be the contribution of that support vector towards assigning class $y^{(i)}$ to the new data point.

The dot product is not the only "similarity" measure available for comparing two data points; in fact, we might think of substituting the terms $x^T x^{(i)}$ in the decision function with *kernel* functions, $k(\cdot,\cdot) : \mathbb{R}^p \times \mathbb{R}^p \to \mathbb{R}$ (see Figure 6); in this case, the elements of matrix Q in problem (19) shall be defined as $Q_{ij} = y^{(i)}y^{(j)}k(x^{(i)}, x^{(j)})$. Without delving deep into kernel theory, we shall just recall that a function k is a valid kernel to be used if and only if:

- the matrix Q is positive semi-definite for any possible dataset $\mathcal{D}$ (we are guaranteed that the dual is actually a convex problem);
- the kernel function represents a dot product between the data points mapped to some possibly higher dimensional space.

In these cases, of course, the decision function becomes

$$h(x) = \sum_i \alpha_i^\star y^{(i)}k(x, x^{(i)}),$$

which is in general not a linear function. From the one hand, it is no more possible to express the classifier in terms of weights w; on the other hand, the classifier is nonlinear, so that more powerful classifiers can be constructed.

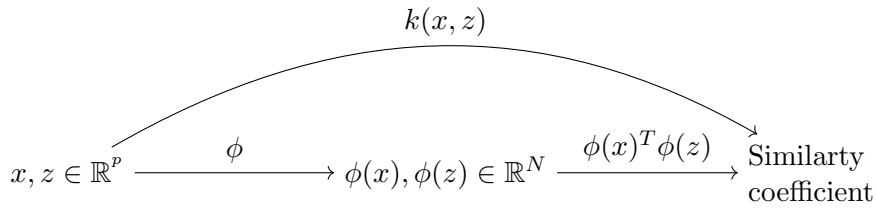
To sum up, the two major reasons to consider the dual SVM problem are:

- by the kernel trick, it is possible to obtain nonlinear classifiers;
- the problem is convex quadratic with linear constraints, like the primal, but constraints are simpler (the only "difficult" constraint is $\alpha^T y = 0$).

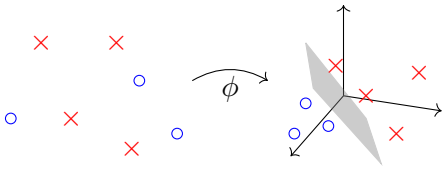
3.2 Solving the Dual Problem

In this section we show how the dual problem for nonlinear SVM training can be efficiently solved. We first recall the formulation with some notation changes to make it clearer from an optimization perspective:

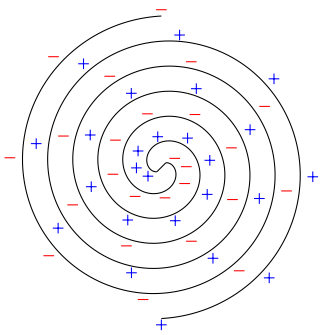
$$\begin{aligned} \min\ & \frac{1}{2}x^T Q x - e^T x \\ \text{s.t. } & a^T x = 0, \\ & 0 \le x \le C, \end{aligned} \tag{20}$$



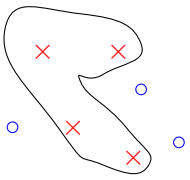
(a) Kernels concept



(b) Kernels and feature maps



(c) Example of separation surface obtainable with kernels



(d) Example of separation surface obtainable with kernels

Figure 6: Kernels

where $Q \in \mathbb{R}^{n\times n}$ is a positive semi-definite matrix (n is training set dimension) and $e = (1\dots1)^T$. Typically n is large and Q is dense; these facts contribute to problem hardness. The other source of complexity is the linear constraint $a^T x = 0$. Box constraints are easy to treat, since they are component-wise separable.

This problem is efficiently solved through a decomposition strategy: at each iteration a working set $W \subset \{1,\dots,n\}$ is selected; $\overline{W} = \{1,\dots,n\} \setminus W$ denotes its complementary. The resulting subproblem is therefore

$$\begin{aligned} \min_{x_W}\ & f(x_W, x^k_{\overline{W}}) = \frac{1}{2} x_W^T Q_{WW} x_W - (e_W - Q_{W\overline{W}} x^k_{\overline{W}})^T x_W \\ \text{s.t. }\ & a_W^T x_W = -a_{\overline{W}}^T x^k_{\overline{W}}, \\ & 0 \le x_W \le C, \end{aligned} \tag{21}$$

whose solution is $x^\star_W$. Thus, variables are updated as follows:

$$x_i^{k+1} = \begin{cases} x_i^\star & \text{if } i \in W, \\ x_i^k & \text{if } i \notin W. \end{cases}$$

At this point, two questions arise. The first one is the minimum cardinality of W: it must be $|W| \ge 2$; if it were $W = \{i\}$ it would always be $x^{k+1} = x^k$, since both x^k and x^{k+1} are feasible and satisfy $a_i x_i = -a_{\overline{W}}^T x^k_{\overline{W}}$.

Decomposition algorithms are classified according to the working set cardinality: if $|W| = 2$, we talk about *Sequential Minimal Optimization* (SMO) algorithms, that do not require a solver for the subproblem; otherwise, if $|W| > 2$, the algorithm needs a solver to find a solution to (21).

The second issue is the selection of W; we will deepen this aspect in the rest of this section.

3.2.1 Sequential Minimal Optimization

In the case $|W| = 2$, subproblem (21) becomes

$$\begin{aligned} \min\ & \frac{1}{2}\begin{bmatrix} x_i & x_j \end{bmatrix}\begin{bmatrix} q_{ii} & q_{ji} \\ q_{ij} & q_{jj} \end{bmatrix}\begin{bmatrix} x_i \\ x_j \end{bmatrix} - x_i - x_j + p_{\overline{W}}^T \begin{bmatrix} x_i \\ x_j \end{bmatrix} \\ \text{s.t. } & a_i x_i + a_j x_j = b, \\ & 0 \le x_i, x_j \le C, \end{aligned} \tag{22}$$

which is a quadratic convex problem in the two variables x_i, x_j. The solution to this problem can be computed analytically: not needing to employ any solver is the main benefit of SMO algorithms.

Now, we focus on the selection strategy of subproblem variables; we want

$$x^{k+1} = (x_1^k, \dots, x_i^{k+1}, \dots, x_j^{k+1}, \dots, x_n^k)$$

to be feasible and

$$f(x^{k+1}) < f(x^k).$$

To achieve these goals, we have to identify at each step a feasible descent direction with two and only two non-zero components.

Proposition 3.2. *The set of feasible directions for Problem* (20) *at point* $\bar{x}$ *is given by*

$$\mathcal{D}(\bar{x}) = \left\{ d \in \mathbb{R}^n \mid a^T d = 0,\ d_i \ge 0\ \forall\, i \in L(\bar{x}),\ d_i \le 0\ \forall\, i \in U(\bar{x}) \right\}$$

where

$$L(\bar{x}) = \{ i \in \{1, \dots, n\} \mid \bar{x}_i = 0 \};$$
$$U(\bar{x}) = \{ i \in \{1, \dots, n\} \mid \bar{x}_i = C \}.$$

Proof. Let's denote with $\mathcal{S}$ the *feasible set*

$$\mathcal{S} = \{ x \in \mathbb{R}^n \mid a^T x = 0,\ 0 \le x \le C \}$$

Let $\bar{x} \in \mathcal{S}$ and let $\mathcal{D}(\bar{x})$ be the set of feasible directions at $\bar{x}$; if $d \in \mathcal{D}(\bar{x})$, then $\bar{x} + td \in \mathcal{S}\ \forall\, t \in [0, \bar{t}\,]$ for $\bar{t}$ sufficiently small. Thus $a^T(\bar{x} + td) = 0$, i.e., $a^T\bar{x} + ta^T d = 0$, which implies

$$a^T d = 0. \tag{23}$$

Furthermore $0 \le \bar{x} + td \le C$, which implies

$$\begin{cases} d_i \le 0 & \text{if } \bar{x}_i = C \\ d_i \ge 0 & \text{if } \bar{x}_i = 0. \end{cases} \tag{24}$$

Putting conditions (23) and (24) together we get the thesis. □

We are looking for directions in $\mathcal{D}(\bar{x})$ with two non-zero components:

$$d^{i,j} = \begin{pmatrix} 0 & \dots & d_i & \dots & d_j & \dots & 0 \end{pmatrix}^T. \tag{25}$$

Since $a^T d^{i,j} = 0$, we have $a_i d_i + a_j d_j = 0$: we can choose

$$d_i = \frac{1}{a_i}, \qquad d_j = -\frac{1}{a_j}. \tag{26}$$

Moreover, assume $i \in L(\bar{x})$, i.e., $\bar{x}_i = 0$. Then we need $d_i \ge 0$ and thus we can only consider variables corresponding to $a_i > 0$. On the contrary, if $i \in U(\bar{x})$, then d_i must be

index	variable value	feasible direction	coeff. constraint
$i \in L(\bar{x})$	$\bar{x}_i = 0$	$d_i \geq 0$	$a_i > 0$
$i \in U(\bar{x})$	$\bar{x}_i = C$	$d_i \leq 0$	$a_i < 0$
$j \in L(\bar{x})$	$\bar{x}_j = 0$	$d_j \geq 0$	$a_j < 0$
$j \in U(\bar{x})$	$\bar{x}_j = C$	$d_j \leq 0$	$a_j > 0$

Table 1: Constraints on the selection of non-zero components of directions $d^{i,j}$, based on the sign of coefficients a.

non-positive and we can consider the i-th component only if $a_i < 0$. Similarly, if $j \in L(\bar{x})$ we need $a_j < 0$ and if $j \in U(\bar{x})$ it has to be $a_j > 0$ (Table 1).
We note that if $0 < \bar{x}_i < C$ there is no constraint on the sign of a_i; the same observation holds for a_j. We can subsequently partition sets U and L as follows:

$$L(\bar{x}) = L^+(\bar{x}) \cup L^-(\bar{x}) \qquad U(\bar{x}) = U^+(\bar{x}) \cup U^-(\bar{x})$$

where

$$L^+(\bar{x}) = \{h \in L(\bar{x}) \mid a_h > 0\} \qquad L^-(\bar{x}) = \{h \in L(\bar{x}) \mid a_h < 0\}$$
$$U^+(\bar{x}) = \{h \in U(\bar{x}) \mid a_h > 0\} \qquad U^-(\bar{x}) = \{h \in U(\bar{x}) \mid a_h < 0\}.$$

In the end, we define the sets

$$R(\bar{x}) = L^+(\bar{x}) \cup U^-(\bar{x}) \cup \{i \mid 0 < \bar{x}_i < C\}$$

and

$$S(\bar{x}) = L^-(\bar{x}) \cup U^+(\bar{x}) \cup \{i \mid 0 < \bar{x}_i < C\}.$$

Proposition 3.3. *Direction $d^{i,j}$ defined as* (25)-(26) *is feasible at $\bar{x}$ for Problem* (20) *if and only if $i \in R(\bar{x})$ and $j \in S(\bar{x})$.*

Proof. Let $d^{i,j}$ be feasible and assume by contradiction that $j \notin S(\bar{x})$. Then, either $j \in L^+(\bar{x})$ or $j \in U^-(\bar{x})$, but in the first case $d_j = -1/a_j < 0$, while in the second one $d_j = -1/a_j > 0$: both cases violate the feasibility assumption. The case $i \notin R(\bar{x})$ is similar.
On the other hand, let $i \in R(\bar{x})$ and $j \in S(\bar{x})$; in particular, assume $i \in U^-(\bar{x})$ and $j \in U^+(\bar{x})$ (the other combinations can be treated similarly). Then $a^T d^{i,j} = a_i \frac{1}{a_i} - a_j \frac{1}{a_j} = 0$; moreover, since $i \in U^-(\bar{x})$, $a_i < 0$ and therefore $d_i = 1/a_i < 0$; similarly, $j \in U^+(\bar{x})$ implies $a_j > 0$ and thus $d_j = -1/a_j < 0$. This completes the proof. □

Proposition 3.4. *Direction $d^{i,j}$ defined as* (25)-(26) *is a descent direction for Problem* (20) *if and only if*

$$\frac{\nabla_i f(\bar{x})}{a_i} < \frac{\nabla_j f(\bar{x})}{a_j}. \tag{27}$$

Proof. Since f is a quadratic convex function, d is a descent direction at $\bar{x}$ if and only if we have

$$\nabla f(\bar{x})^T d^{i,j} = \frac{1}{a_i} \nabla_i f(\bar{x}) - \frac{1}{a_j} \nabla_j f(\bar{x}) < 0,$$

i.e., (27), is a necessary and sufficient condition of descent. □

We are now able to write the general scheme of an SMO algorithm (Algorithm 2).

As we know from Propositions 3.3 and 3.4, this choice of i and j guarantees x^{k+1} feasibility and f decrease. The gradient of f at x^{k+1} is given by

$$\begin{aligned} \nabla f(x^{k+1}) &= Qx^{k+1} - e = Q(x^{k+1} - x^k) + Qx^k - e \\ &= Q_i(x_i^{k+1} - x_i^k) + Q_j(x_j^{k+1} - x_j^k) + \nabla f(x^k), \end{aligned}$$

Algorithm 2: `Sequential Minimal Optimization`

Input: Q, a, C.
$x^0 = 0$;
$k = 0$;
$\nabla f(x^0) = -e$;
while *stopping criterion not satisfied* **do**
  select $i \in R(x^k)$, $j \in S(x^k)$ such that $\frac{\nabla_i f(\bar{x})}{a_i} - \frac{\nabla_j f(\bar{x})}{a_j} < 0$
  $W = \{i, j\}$;
  solve analytically
  $$\min_{x_W} f(x_W, x^k_{\overline{W}}) \text{ s.t. } a_W^T x_W = -a_{\overline{W}}^T x^k_{\overline{W}}, \ \ 0 \le x_W \le C;$$
  let $x^\star_W$ be the solution found at the previous step;
  $$\text{set } x_h^{k+1} = \begin{cases} x_j^\star & \text{if } h = j, \\ x_i^\star & \text{if } h = i, \\ x_h^k & \text{otherwise;} \end{cases}$$
  $\nabla f(x^{k+1}) = \nabla f(x^k) + Q_i(x_i^{k+1} - x_i^k) + Q_j(x_j^{k+1} - x_j^k)$;
  $k = k + 1$;
Output: x^k

since Qx^{k+1} and Qx^k only differ by the i-th and j-th components. This expression shows that only two columns of Q are needed to update the gradients, resulting in a saving of time. Moreover, $x^0 = 0$ and thus $\nabla f(x^0) = -e$: there is no need to compute the gradient at the starting point. Matrix Q is thus never required in its entirety to get the gradient: this is an advantage also in terms of I/O time.

Feasibility and strict decrease are not sufficient to ensure global convergence; in the next section we will address convergence issues more deeply.

3.2.2 SMO Convergence Properties with First-order Selection Rule

The choice of the working set $W = \{i, j\}$ is the key to obtain convergence properties for SMO.

Proposition 3.5. *A point $x^\star$ is a global minimizer for problem (20) if and only if*

$$\max_{h \in R(x^\star)} \left\{ -\frac{\nabla_h f(x^\star)}{a_h} \right\} \le \min_{h \in S(x^\star)} \left\{ -\frac{\nabla_h f(x^\star)}{a_h} \right\} \tag{28}$$

Proof. Since problem (20) is a convex problem with linear constraints, KKTs are necessary and sufficient optimality conditions:

$$\begin{aligned} \nabla_i \mathcal{L}(x, \lambda, \xi, \hat{\xi}) = \nabla_i f(x) + \lambda a_i - \xi_i + \hat{\xi}_i = 0 \quad & \forall i = 1, \dots, n, \\ \xi_i x_i = 0 \quad & \forall i = 1, \dots, n, \\ \hat{\xi}_i (x_i - C) = 0 \quad & \forall i = 1, \dots, n, \\ \xi, \hat{\xi} \ge 0 \quad & \forall i = 1, \dots, n. \end{aligned}$$

Therefore, for optimal $x^\star, \lambda^\star, \xi^\star, \hat{\xi}^\star$, we have

$$\nabla_i f(x^\star) + \lambda^\star a_i \begin{cases} \ge 0 & \text{if } i \in L(x^\star), \\ \le 0 & \text{if } i \in U(x^\star), \\ = 0 & \text{if } 0 < x_i^\star < C, \end{cases}$$

and then

$$\frac{\nabla_i f(x^\star)}{a_i} + \lambda^\star \begin{cases} \ge 0 & \text{if } i \in L^+(x^\star) \cup U^-(x^\star), \\ \le 0 & \text{if } i \in L^-(x^\star) \cup U^+(x^\star), \\ = 0 & \text{otherwise,} \end{cases}$$

from which follows

$$\begin{cases} \lambda^\star \leq -\frac{\nabla_i f(x^\star)}{a_i} & \forall i \in L^-(x^\star) \cup U^+(x^\star), \\ \lambda^\star \geq -\frac{\nabla_i f(x^\star)}{a_i} & \forall i \in L^+(x^\star) \cup U^-(x^\star), \\ \lambda^\star = -\frac{\nabla_i f(x^\star)}{a_i} & \forall i : 0 < x_i^\star < C. \end{cases}$$

Recalling

$$R(x^\star) = L^+(x^\star) \cup U^-(x^\star) \cup \{i \mid 0 < x_i^\star < C\},$$
$$S(x^\star) = L^-(x^\star) \cup U^+(x^\star) \cup \{i \mid 0 < x_i^\star < C\},$$

we finally obtain (28). □

Corollary 3.6. *Let x^k be the current (feasible) solution and assume it is not optimal. Then there exist $i \in R(x^k)$ and $j \in S(x^k)$ such that*

$$-\frac{\nabla_i f(x^k)}{a_i} > -\frac{\nabla_j f(x^k)}{a_j}.$$

Proof. This Corollary is a direct consequence of Proposition 3.5. □

Definition 5. The *most violating pair* at the feasible, non optimal solution x^k of (20) is the pair of indices $(i^\star, j^\star)$ defined as

$$i^\star \in \arg\max_{h \in R(x^k)} \left\{ -\frac{\nabla_h f(x^k)}{a_h} \right\}, \qquad j^\star \in \arg\min_{h \in S(x^k)} \left\{ -\frac{\nabla_h f(x^k)}{a_h} \right\}. \tag{29}$$

We can think of selecting in the SMO scheme a Most Violating Pair, i.e., the pair $(i^\star, j^\star)$ that violates the most the optimality condition (28). We thus obtain the famous SVM$^{\text{light}}$ Algorithm 3, which is a SMO decomposition algorithm with convergence properties historically employed in software libraries to solve non-linear SVM training problem.

Algorithm 3: SVM$^{\text{light}}$

Input: Q, a, C.
$x^0 = 0$;
$k = 0$;
$\nabla f(x^0) = -e$;
while *stopping criterion not satisfied* **do**
  identify the most violating pair $(i^\star, j^\star)$
  $W = \{i^\star, j^\star\}$;
  solve analytically
    $$\min_{x_W} f(x_W, x_{\overline{W}}^k) \text{ s.t. } ; a_W^T x_W = -a_{\overline{W}}^T x_{\overline{W}}^k, \;\; 0 \leq x_W \leq C;$$
  let $x_W^\star$ be the solution found at the previous step; set
    $$x_h^{k+1} = \begin{cases} x_h^\star & \text{if } h \in W, \\ x_h^k & \text{otherwise;} \end{cases}$$
  $\nabla f(x^{k+1}) = \nabla f(x^k) + Q_{i^\star}(x_{i^\star}^{k+1} - x_{i^\star}^k) + Q_{j^\star}(x_{j^\star}^{k+1} - x_{j^\star}^k)$;
  $k = k + 1$;
Output: x^k

By a quite complex proof, the following proposition can be stated.

Proposition 3.7. *The sequence $\{x_k\}$ generated by algorithm SVM^{light} has limit points, each one being a solution of problem (20).*

Last, we have to discuss about the stopping criterion. The optimality condition is (28); a reasonable stopping criterion is based on the quantities

$$m(x) = \max_{h \in R(x)} \left\{ -\frac{\nabla_h f(x^k)}{a_h} \right\}, \qquad M(x) = \min_{h \in S(x)} \left\{ -\frac{\nabla_h f(x^k)}{a_h} \right\}. \tag{30}$$

From Proposition 28, $x^\star$ is an optimal solution if and only if $m(x^\star) \leq M(x^\star)$. Then we can think of defining a stopping criterion based on the following proposition.

Proposition 3.8. *Let $m(x)$ and $M(x)$ be defined as in (30). Then, for any $\epsilon > 0$, algorithm $\texttt{SVM}^{\texttt{light}}$ produces a solution x^k satisfying*

$$m(x^k) \leq M(x^k) + \epsilon \tag{31}$$

within a finite number of iterations.

The proof of the above property again requires a quite complicated reasoning, mainly because of the fact the functions $m(x)$ and $M(x)$ are not continuous.

3.2.3 Working Set Selection using Second Order Information

The choice of the most violating pair as working set is directly related to the first order approximation of f. $W = \{i^\star, j^\star\}$ indeed can be proven to solve following problem:

$$\begin{aligned} \min_{W:|W|=2} \; & \min_{d_W} \nabla_W f(x^k)^T d_W \\ & \text{s.t. } a_W^T d_W = 0, \\ & \quad d_t \geq 0 \text{ if } x_t^k = 0,\ t \in W, \\ & \quad d_t \leq 0 \text{ if } x_t^k = C,\ t \in W, \\ & \quad -1 \leq d_t \leq 1,\ t \in W. \end{aligned} \tag{32}$$

Recalling our definition of d, the first order approximation of f at x^k is

$$f(x^k + d) \approx f(x^k) + \nabla f(x^k)^T d = f(x^k) + \nabla_W f(x^k)^T d_W$$

which is exactly the objective of (32). The linear constraint comes from $a^T(x^k + d) = 0$ and $a^T x^k = 0$. The second and third constraints come from $0 < x^k + d < C$. The box constraint prevents the linear objective to go to $-\infty$. Looking for the maximal violating pair is an efficient way to solve this problem (time required is $\mathcal{O}(n)$).

We might think of extending this technique to exploit second order information. In fact, this is somehow what is implemented to select the working set in the current version of LIBSVM. Since f is quadratic, we have the reduction of the objective value can be exactly expressed as

$$\begin{aligned} f(x^k + d) - f(x^k) &= \nabla f(x^k)^T d + \frac{1}{2} d^T \nabla^2 f(x^k) d \\ &= \nabla_W f(x^k)^T d_W + \frac{1}{2} d_W^T \nabla^2_{WW} f(x^k) d_W. \end{aligned} \tag{33}$$

Substituting the objective function in (32) with this quantity and removing the box constraint (which is no more necessary since the new objective function is bounded below), we get the following problem:

$$\begin{aligned} \min_{W:|W|=2} \; & \min_{d_W} \nabla_W f(x^k)^T d_W + \frac{1}{2} d_W^T \nabla^2_{WW} f(x^k) d_W \\ & \text{s.t. } a_W^T d_W = 0, \\ & \quad d_t \geq 0 \text{ if } x_t^k = 0,\ t \in W, \\ & \quad d_t \leq 0 \text{ if } x_t^k = C,\ t \in W. \end{aligned} \tag{34}$$

However, solving this problem requires $\mathcal{O}(n^2)$ operations (all $n(n-1)/2$ possible working sets should be checked). Then, the way of proceeding is checking only several Ws heuristically: one component of W is selected as before (e.g. $i^\star \in \arg\max_{h \in R(x^k)} \left\{-\nabla_h f(x^k)/a_h\right\}$); now, only $\mathcal{O}(n)$ Ws are left to be checked to decide $j^\star$.

It has been shown that $j^\star$ solving (34) fixed $i^\star$ is such that

$$j^\star \in \arg\min_t \left\{ -\frac{b_{it}^2}{a_{it}} \middle| \, t \in S(x^k), -\nabla_t f(x^k)/a_t < -\nabla_i f(x^k)/a_i, \right\}$$

where $a_{it} = Q_{ii} + Q_{tt} - 2Q_{it}$ and $b_{it} = \nabla_t f(x^k)/a_t - \nabla_i f(x^k)/a_i > 0$.

In fact, this formula holds assuming the kernel matrix is positive definite; in the positive semi-definite case some corrections would be required, but we will not address them here.

It is important to remark that with this working set selection rule the theoretical convergence properties of `SVM`$^{\texttt{light}}$ continue to hold.

3.3 Algorithms for Linear SVMs

With particularly large datasets, linear classifiers are often the preferable solution, for two main reasons; first, data dimensionality may be so high that mapping points to higher dimensional spaces by the kernel trick may not be necessary; secondly, if the linear kernel is used, special features of the problem can be exploited to make the training process much faster and thus allowing to carry out training with larger amounts of data.

Problem (18) can be equivalently rewritten as

$$\min_w \ \frac{1}{2}\|w\|^2 + C\sum_{i=1}^{n} \max\{0, 1 - y^{(i)} w^T x^{(i)}\}; \tag{35}$$

here, the bias term b is not explicitly introduced in the model (we talk about *unbiased formulations*); this is not a restriction from a statistical perspective: bias can be implicitly set into the model by adding a constant feature across all examples in the dataset. However, we will see that this simple change has interesting consequences from an optimization point of view.

A similar problem that can be addressed is the following one

$$\min_w \ \frac{1}{2}\|w\|^2 + C\sum_{i=1}^{n} \max\{0, 1 - y^{(i)} w^T x^{(i)}\}^2, \tag{36}$$

where the hinge loss terms are squared.

These two problems share the dual formulation:

$$\begin{aligned} &\min_\alpha \ \frac{1}{2}\alpha^T \bar{Q}\alpha - e^T\alpha \\ &\text{s.t. } 0 \le \alpha \le U, \end{aligned} \tag{37}$$

where $\bar{Q} = Q + D$ and D is a diagonal matrix; for problem (35) we have $U = C$ and $D_{ii} = 0 \, \forall i$, while for problem (36) we have $U = \infty$ and $D_{ii} = \frac{1}{2C}$. We can observe that the removal of the bias term leads to a bound constrained dual, without the linear equality constraint.

A *dual coordinate descent* (DCD) algorithm has been proposed to efficiently solve problem (37) when the linear kernel is employed. The algorithm has the following characteristics:

- it is a decomposition method where the variables are updated one at a time:

$$\alpha_i^{k+1} = \arg\min_{\alpha_i \in [0,U]} f(\alpha_1^{k+1}, \ldots, \alpha_{i-1}^{k+1}, \alpha_i, \alpha_{i+1}^k \ldots, \alpha_n^k),$$

  which can be done in closed form if $\nabla_i f(\alpha_1^{k+1}, \ldots, \alpha_{i-1}^{k+1}, \alpha_i^k, \alpha_{i+1}^k \ldots, \alpha_n^k)$ is available;

- variables are updated in cyclic order;
- randomly selecting the variables to be updated experimentally proves to be more efficient;
- solving the subproblems requires the knowledge of $\nabla_i f(\alpha^k)$, which in turn requires the i-th column of $\bar{Q}$;

In Table 2, we report the cost of the main operations of the DCD and the SMO algorithms, in order to carry out a comparison. If we assume the cost of computing a kernel matrix element to be $\mathcal{O}(p)$, we have that SMO spends $\mathcal{O}(np)$ to keep the gradients updated, but then it can smartly select the variables and then solve the subproblems with basically no additional cost; on the other hand, the DCD requires to compute a kernel column at each update; the single iteration of both algorithms approximately has the same cost, but since SMO selects the variables in a much more effective manner, exploiting first-order information, the number of total iterations is usually much lower for this latter algorithm.

However, in the special case of linear kernels, the relation $w = \sum_{i=1}^{n} y^{(i)} \alpha_i x^{(i)}$ can be exploited; in fact, the following equality holds

$$\nabla_i f(\alpha^k) = y^{(i)} w^T x^{(i)} - 1 + D_{ii} \alpha_i^k,$$

making the variables update cost $\mathcal{O}(p)$, i.e., the iteration cost is much lower than SMO and does not depend on the number of the training samples.

	SMO	**DCD**	**DCD-linear**
variables update	$\mathcal{O}(1)$	$\mathcal{O}(np)$	$\mathcal{O}(p)$
gradients update	$\mathcal{O}(np)$	NA	NA

Table 2: Cost of main operations in SMO and DCD.

The DCD algorithm is thus well suited for large scale linear classification. However, as the size of the problems keeps increasing, this approach may also become computationally unsustainable.

For these cases, a specialized Newton type method has been proposed to directly tackle the continuously differentiable primal problem (36).

These approaches have been implemented in the popular LIBLINEAR library for large scale linear classification.

4 Large Scale Optimization for Deep Models

One of the most relevant optimization problems nowadays is that of *training artificial neural networks*. ANNs are a powerful machine learning model that has been used in a number of applications with impressive results.

The *empirical risk minimization* task of a network with weights $w \in \mathbb{R}^n$ given the data set (X, Y) of N samples takes the form of problem (13), but possesses a number of very peculiar features w.r.t. classical unconstrained nonlinear optimization problem. More in detail:

1. As with other machine learning problems, we are not actually interested in minimizing the empirical risk, which is a surrogate objective function, but rather in finding an effective model in terms of generalization capabilities. In fact, the highly nonconvex training problem typically has plenty of local minima, most of which with a loss value pretty close to the global optimizer; however, the out-of-sample performance of these solution may dramatically vary from one to another. There is thus no point in seeking the global optimum; rather, good out-of-sample solutions often correspond to local optima that can be reached even without low-precision solvers.

2. Thanks to the *backpropagation* algorithm, the gradients of the loss function $\nabla\mathcal{L}(w)$ can be computed efficiently, to the point that the cost of gradients computation is only twice the cost of evaluating the loss function itself. More details on backpropagation and automatic differentiation are reported in Section 4.2.

3. On the other hand, computing the value of $\mathcal{L}(w)$ is computationally expensive, as the entire data set has to be used to determine all the elements of the sum defining the function.

SGD, and more in general stochastic optimization algorithms, appear to be well suited for ANN training problems for many reasons, which we try to summarize hereafter:

1. Data is often redundant, so using all the available information at every iteration is in fact inefficient.

2. The computational experience gathered by the machine learning community through the years teaches that, if a proper step size α is selected, stochastic methods are indeed much faster than batch ones, especially at the early stages of the optimization process.

3. The rate of convergence to ϵ-optimality of SGD is slower than that of batch GD ($\mathcal{O}(1/\epsilon)$ vs. $\mathcal{O}(N\log(\epsilon))$), but is in fact independent of the training set size, which is typically huge in applications. Thus the asymptotic lower cost of GD starts to be observed only for very low values of ϵ.

4. Since it is difficult for SGD to end up in "sharp minima", an intrinsic regularization operation is somehow carried out, leading to benefits in terms of generalization properties.

Note that, however, batch approaches possess some intrinsic advantages. In particular:

- the use of full gradient information at each iteration opens the door for many deterministic gradient-based optimization methods;

- due to the sum structure of the empirical risk, a batch method, as opposed to SG, can easily benefit from parallelization since the bulk of the computation lies in evaluations of the objective function and gradients;

- if we were to consider a larger number of epochs, then one would see the batch approach eventually overtake the stochastic method and yield a lower training error.

The above comments, both of theoretical and empirical nature, should be convincing about minibatch SGD being the most appropriate approach to solving deep neural nets problem. Choosing $1 < M \ll N$ is a middle way that allows to take advantage of the best features of both stochastic and full batch methods.

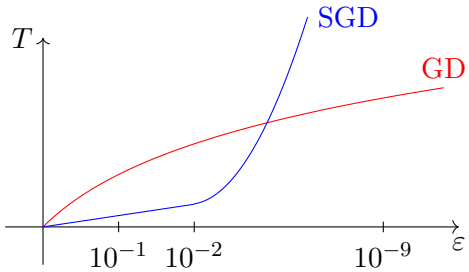


Figure 7: Tipico andamento delle prestazioni di SGD e GD in problemi di addestramento di deep networks.

4.1 Improvements to SGD for Deep Networks Training

In this section, we describe some modifications to the SGD algorithm that have proven to be practically useful for solving NNs training problems.

4.1.1 Acceleration

Momentum or Nesterov acceleration terms, discussed in Section 1.3 for the standard unconstrained optimization case, are particularly impactful with stochastic optimization algorithms for deep learning. There are two main reasons:

- Momentum terms are based on averages of the previous steps; these averages somehow filter out strong oscillations due to the stochastic nature of the considered "descent" directions; thus, adding momentum or acceleration terms allows to heavily alleviate the zigzagging behavior that is typical of SGD-type algorithms.
- In the deep learning case, the computation of both the objective function and its gradient is expensive; on the other hand, the information provided by terms $(x^k - x^{k-1})$ is obtainable very cheaply; thus, acceleration can be seen as a way to improve the effectiveness of each iteration without basically any additional cost.

4.1.2 Adaptive stepsizes

Momentum and Nesterov AG exploit information about past iterations to change and improve the quality of search directions. The other road that has been followed (successfully) for the last years with deep learning problems in the field of nonlinear nonconvex stochastic optimization is that of using adaptive learning rates: the stepsize changes at each iteration, based on the progress of the process; moreover, since weights gradients have different dynamics at different layers, each variable is associated with a different step size.

Several strategies to update the learning rate have been proposed; in the following we report the most relevant ones. For the sake of simplicity, we will ignore the fact that in stochastic optimization gradients are computed using only a portion of the objective function. We will describe the upcoming methods with "batch optimization" notation, but they are stochastic methods indeed.

AdaGrad One of the first proposed methods following this idea is *AdaGrad*. In AdaGrad, the running sums s^k of the squares of directional derivatives is stored:

$$s_i^{k+1} = s_i^k + (\nabla_i f(x^k))^2.$$

These quantities are used to scale the gradients, leading to the following update formula

$$x^{k+1} = x^k - \alpha(\text{diag}(s^{k+1}) + \epsilon I)^{-1/2}\nabla f(x^k), \tag{38}$$

where ϵ is a is a smoothing term that avoids division by zero; the update rule can be equivalently written in a component-wise form which is easier to interpret

$$x_i^{k+1} = x_i^k - \frac{\alpha}{\sqrt{s_i^{k+1} + \epsilon}}\nabla_i f(x^k).$$

Note that the components of s are increasing through the iterations. This is ok in principle, as SG theory suggests to diminish learning rates, scaling them with $\mathcal{O}(1/k)$. However, the behavior of AdaGrad is often too aggressive in practice. For this reason, variants have been proposed in the literature, having great impact in the deep learning field.

RMSprop The *RMSprop* (Root Mean Square Propagation) method tries to overcome the limitations of AdaGrad replacing the running sum with an exponentially decaying average of past gradients:

$$s_i^{k+1} = \rho s_i^k + (1-\rho)(\nabla_i f(x^k))^2.$$

Then, parameters are updated by equation (38), exactly as with AdaGrad. With this simple modification, the most recent steps have the largest influence. Also, the sequence s^k is no more divergent, with an advantage in terms of numerical stability.

AdaDelta *AdaDelta* is another variant of AdaGrad, trying again to reduce the aggressive behavior of the latter algorithm. In fact, even though AdaDelta has been developed independently from RMSprop, it can be seen as an extension of the latter. Along with the exponential average of the squared gradients, AdaDelta requires to store the running average of the squared displacements:

$$r^{k+1} = \gamma r^k + (1-\gamma)(x^k - x^{k-1})^2.$$

The update rule of AdaDelta is then

$$x_i^{k+1} = x_i^k - \frac{\sqrt{r_i^{k+1} + \epsilon}}{\sqrt{s_i^{k+1} + \epsilon}} \nabla_i f(x^k).$$

We can note there is no more need to set the base learning rate α.

Adam *Adaptive Moment Estimation*, or simply *Adam*, is another adaptive learning rate algorithm. It can be seen as an evolution of RMSprop and AdaDelta and is at present the most widely employed optimization algorithm for deep networks training.

In addition to the exponentially decaying average of past squared gradients, Adam also keeps, similarly as Momentum, the exponentially decaying average of gradients:

$$\begin{aligned} m_i^{k+1} &= \beta_1 m_i^k + (1-\beta_1)\nabla_i f(x^k), \\ v_i^{k+1} &= \beta_2 v_i^k + (1-\beta_2)(\nabla_i f(x^k))^2 \end{aligned}$$

Vectors m^k and v^k can be seen as estimates of the first moment (the mean) and the second moment (the uncentered variance) of the gradients, hence the name of the method. These vectors are initialized to 0, generating a bias towards zero which is particularly strong at initial iterations and when the decay rates are small. Indeed:

$$m_i^{k+1} = (1-\beta_1) \sum_{i=0}^{k} \beta_1^{k-i} \nabla f(x^{k-i})$$

and thus

$$E[m_i^{k+1}] = (1-\beta_1)E[\nabla f(x)] \sum_{i=0}^{k} \beta_1^{k-i} = (1-\beta_1)E[\nabla f(x)] \frac{1-\beta_1^k}{1-\beta_1} = E[\nabla f(x)](1-\beta_1^k).$$

This bias can be countered by computing bias-corrected first and second moment estimates:

$$\hat{m}^k = \frac{m^k}{1-(\beta_1)^k} \tag{39}$$

$$\hat{v}^k = \frac{v^k}{1-(\beta_2)^k} \tag{40}$$

Then, the variables are updated by the following update rule:

$$x^{k+1} = x_i^k - \frac{\alpha}{\sqrt{\hat{v}_i^k} + \epsilon} \hat{m}_i^k$$

Some variants of the Adam algorithm have been proposed in recent years. One of them is *AdaMax*, which was proposed along with Adam. AdaMax replaces the update formula (40) for the second momentum with the (empirically) more stable

$$u^{k+1} = \max\{\beta_2 u^k, |\nabla f(x^k)|\},$$

which also doesn't need correction for the initialization bias.

Another popular variant of Adam is *Nadam*, which stands for Nesterov-accelerated Adaptive Moment Estimation. The core idea of Nadam is that of incorporating Nesterov's acceleration into Adam.

4.2 Automatic Differentiation and Backpropagation Algorithm

One of the key issues arising to train neural networks concerns the computation of the gradients of the loss function. Indeed, the objective of the optimization problem is (ignoring the regularization term) the finite sum of functions of (typically) millions of variables, each one being the cascaded composition of elementary functions.

Therefore, numerical differentiation techniques are out of question: the cost is too high (a large number of function evaluations is required to obtain gradient approximation) and they also suffer from significant approximation errors; thus, finite difference approximation is generally used only to check gradients implementation correctness, at low accuracy.

On the other hand, deriving explicit expressions for the gradients is too complex; even the employment of symbolic differentiation tools, i.e., software to manipulate expressions for obtaining expressions of the derivatives, leads to very long and complicated formulae and consequently to massive computations.

Neural networks training was actually made possible by the development of *Automatic Differentiation* (AD) techniques. By automatic differentiation, we refer to a set of approaches that smartly exploit the multivariate chain rule:

$$f = f(g(x)) \qquad \frac{\partial f}{\partial x_j} = \sum_i \frac{\partial f}{\partial g_i} \frac{\partial g_i}{\partial x_j}.$$

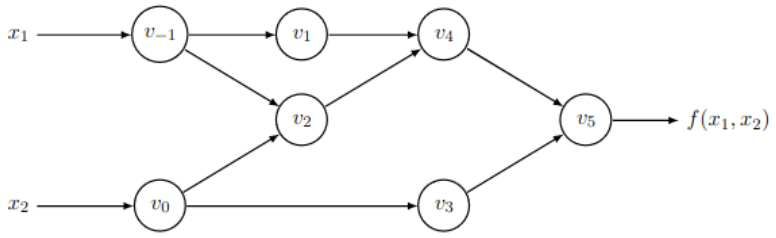
Since evaluating a mathematical function, no matter how complicated, consists in computing out a sequence of elementary operations and functions (exp, log, sin, cos, etc.), by repeatedly applying the chain rule to these operations, partial derivatives can be obtained automatically.

In the context of deep learning, in particular, the AD algorithm typically employed is the famous *backpropagation* algorithm, which is, in technical terms, automatic differentiation in a so called *reverse mode*.

We will not delve deep into the technical details of AD and the backpropagation algorithm. We will just get an idea of its mechanisms by means of a simple example, see Figure 8.

First, all elementary operations are mapped into a *Computation Graph* (Figure 8a), which is a structure that allows to link quantities that depend one from the other. This way, computed quantities that will be needed to compute other terms can be stored in memory, avoiding duplicate computations.

The actual computation first proceeds by feeding the inputs to the function and by computing intermediate products up until the function output; then, the graph is traversed backward, to compute all the partial derivatives. We shall note that, in order to compute the gradients, we obtain as an intermediate side product the function value; thus, in



(a) Computation Graph

Forward Primal Trace			Reverse Adjoint (Derivative) Trace		
v_{-1}	$= x_1$	$= 2$	$\bar{\boldsymbol{x}}_1$	$= \bar{\boldsymbol{v}}_{-1}$	$= \mathbf{5.5}$
v_0	$= x_2$	$= 5$	$\bar{\boldsymbol{x}}_2$	$= \bar{\boldsymbol{v}}_0$	$= \mathbf{1.716}$
v_1	$= \ln v_{-1}$	$= \ln 2$	$\bar{v}_{-1}$	$= \bar{v}_{-1} + \bar{v}_1 \frac{\partial v_1}{\partial v_{-1}}$	$= \bar{v}_{-1} + \bar{v}_1 / v_{-1} \quad = 5.5$
v_2	$= v_{-1} \times v_0$	$= 2 \times 5$	$\bar{v}_0$	$= \bar{v}_0 + \bar{v}_2 \frac{\partial v_2}{\partial v_0}$	$= \bar{v}_0 + \bar{v}_2 \times v_{-1} \quad = 1.716$
			$\bar{v}_{-1}$	$= \bar{v}_2 \frac{\partial v_2}{\partial v_{-1}}$	$= \bar{v}_2 \times v_0 \quad = 5$
v_3	$= \sin v_0$	$= \sin 5$	$\bar{v}_0$	$= \bar{v}_3 \frac{\partial v_3}{\partial v_0}$	$= \bar{v}_3 \times \cos v_0 \quad = -0.284$
v_4	$= v_1 + v_2$	$= 0.693 + 10$	$\bar{v}_2$	$= \bar{v}_4 \frac{\partial v_4}{\partial v_2}$	$= \bar{v}_4 \times 1 \quad = 1$
			$\bar{v}_1$	$= \bar{v}_4 \frac{\partial v_4}{\partial v_1}$	$= \bar{v}_4 \times 1 \quad = 1$
v_5	$= v_4 - v_3$	$= 10.693 + 0.959$	$\bar{v}_3$	$= \bar{v}_5 \frac{\partial v_5}{\partial v_3}$	$= \bar{v}_5 \times (-1) \quad = -1$
			$\bar{v}_4$	$= \bar{v}_5 \frac{\partial v_5}{\partial v_4}$	$= \bar{v}_5 \times 1 \quad = 1$
y	$= v_5$	$= 11.652$	$\bar{v}_5$	$= \bar{y}$	$= 1$

(b) Reverse AD

Figure 8: Automatic differentiation (reverse mode) example; $f(x_1, x_2) = \ln(x_1) + x_1 x_2 - \sin(x_2)$. We use the notation $\bar{v}_t = \partial y / \partial v_t$. Credits: `https://www.jmlr.org/papers/volume18/17-468/17-468.pdf`

a gradient descent iteration, we can obtain the function value at the current point, by carrying out a forward pass through the computation graph, and then the gradients, after a backward pass through the graph.

As aforementioned, some terms appear multiple times throughout computation; in order to avoid duplicate calculations, we can store these values to reuse them when needed; of course, it shall not be hard to imagine that in deep and complex networks terms repeat in a much more massive manner than in the simple example at hand.

4.3 What's more?

A number of relevant themes could not fit into this short course. We report here a list of topics that may be useful to know about, or, at least, be aware of their existence:

- **Handling sparsity-inducing regularizers:** ℓ_1 and ℓ_0 regularizers require specialized tools; in the deep learning setting, ℓ_1 regularization is usually tackled by the *proximal gradient* method and suitable acceleration techniques; other tailored approaches exist to handle ℓ_1 terms in least squares regression; dedicated exact and inexact approaches exist to handle ℓ_0 terms in convex and nonconvex problems.

- **Making NNs training practical:** the training of deep network results in a very complex optimization problem: the surface of the objective function is very irregular and to carry out a proper training, particular care has to be taken with details; for example, the choice of the starting point (*network initialization*) is crucial to avoid ill-posed areas of the space where optimization is impractical; in particular, a robust initialization, coupled with suitable *gradient clipping* techniques and stepsize selection rules, might be necessary to overcome *exploding/vanishing gradients* issues (some partial derivatives might be too large/small).

- **Treating nonsmoothness:** some popular activations employed in NNs architectures are actually *not differentiable* (e.g., the ReLU); from the one hand, ReLU are useful to reduce the vanishing gradients problem; on the other hand, in order to properly study training when nondifferentiability elements have to be taken into account, *nonsmooth optimization* theory should be considered, where *subgradients* and *subderivatives* are employed instead of gradients.

- **Missing theory behind training deep networks** There are important "patches" employed with deep learning algorithms that proved to be very useful (sometimes crucial); however, many of these fixes are empirically proved to work well, but are not fully understood from a theoretical perspective; these tricks have often been investigated and explained years after their introduction, and the comprehension of some of them is still lacking to the present days.