1 22nd of November 2018 — A. Frangioni

1.1 Deflected gradient methods

The idea behind this family of algorithms, is to determine the next position x^{i+1} using the gradient and summing to it something else that gives us more information.

This kind of algorithms work also in cases in which the gradient isn't continuous.

This methods use the information about the prevous iterations without exploiting properties about the second order derivative.

1.1.1 Heavy ball gradient method

The intuition behind this algorithm may be expressed through the following metaphore: an object is moving in the space and it's subject to a force. We can observe that the heavier the object, the stronger should be the force imposed in order to make it describe a certain trajectory.

In this interpretation, we may define the $(i+1)$-th iteration as

$$x^{i+1} \leftarrow x^i - \alpha^i \nabla f(x^i) + \beta^i (x^i - x^{i-1}),$$

where $\boldsymbol{\beta^i}$ is called **momentum**, $\mathbf{x^i}$ **heavy** and $\nabla \mathbf{f}(\mathbf{x^i})$ **force**.

This isn't a descent algorithm: we are not choosing a direction and doing line search. We have no guarantee that the value of the function after one iteration will we smaller than the prevous one.

First thing to do is to choose α^i and β^i properly.

We can prove for some cases that these methods are better than gradient method, although they aren't as good as Newton or quasi Newton. Their strength resides in their simplicity though.

Notice that if the smaller eigenvalue isn't zero (i.e. quadratic case) we have a close formula to choose α and β independently from the iteration:

$$\alpha = \frac{4}{\left(\sqrt{\lambda^1} + \sqrt{\lambda^n}\right)^2}, \qquad \beta = \max\left\{\left|1 - \sqrt{\alpha\lambda^n}\right|, \left|1 - \sqrt{\alpha\lambda^1}\right|\right\}^2$$

We may observe that the step we take is something that goes like $\frac{1}{L}$, where L is the Lipshitz constant, since λ_n is very small. With these choices the rate is the following. We observe that in the gradient the rate is the same, although there aren't the square roots

$$\left\| x^{i+1} - x_* \right\} \leq \left(\frac{\sqrt{\lambda^1} - \sqrt{\lambda^n}}{\sqrt{\lambda^1} + \sqrt{\lambda^n}} \right) \cdot \left\| x^i - x_* \right\|$$

An alternative idea could be choosing β^i and finding α^i using line search. A possible issue is that we don not know if we are moving along a descending direction, but in this method it is perfectly acceptable not to make any movement at a single step (notice that in gradient method if one step has size 0 then we will not move anymore).

β^i is seen as an hyperparameter, hence its value is tuned rerunning the alogirthm several times.

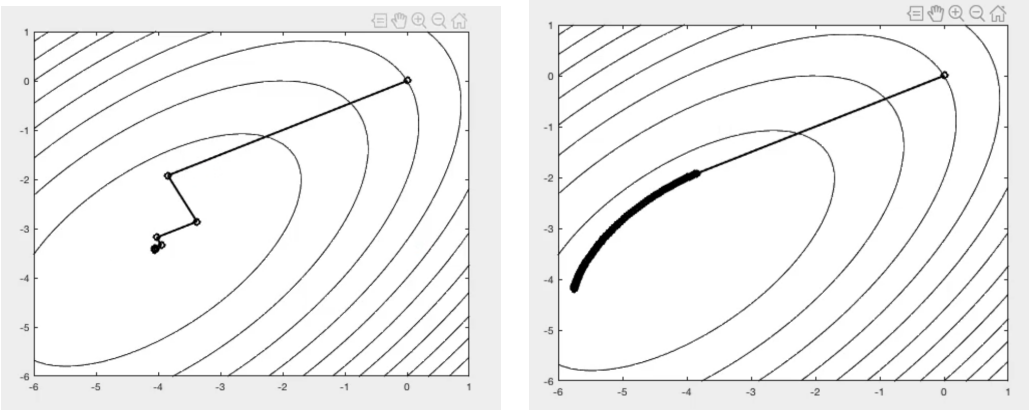

FIGURE 1.1: On the left side Newton method and on the right side the heavy ball method.

The plot of the convergenc of the heavy ball method gives a graphical idea of the fact that the direction isn't orthogonal to the one at the previous iteration, since we have the gradient plus some quantity.

In particular, the bigger β^i the "less orthogonal" the steps are. This feature allows the algorithm to have good performances on elongated functions.

1.1.2 Accelerated gradient

This method works only on convex functions, it has some similarities with heavy ball, but it's slightly different.

ALGORITHM 1.1 Pseudocode for accelerated gradient method.

procedure ACCG$(f, \nabla f, x, \varepsilon)$
  $x_- \leftarrow x$;
  $\gamma \leftarrow 1$;
  repeat
    $\gamma_- \leftarrow \gamma$;
    $\gamma \leftarrow (\sqrt{4\gamma^2 + \gamma^4} - \gamma^2)/2$;
    $\beta \leftarrow \gamma(1/\gamma_- - 1)$;
    $y \leftarrow x + \beta(x - x_-)$;
    $g \leftarrow \nabla f(y)$;
    $x_- \leftarrow x$;
    $x \leftarrow y - (1/L)g$;
  until $(\|g\| > \varepsilon)$
end procedure

The rational behind this algorithm is the following: When we are in a certain point at a certain iteration, we go on a little bit β^i and we end up in a point y. The gradient is computed in that point and used to choose the next point.

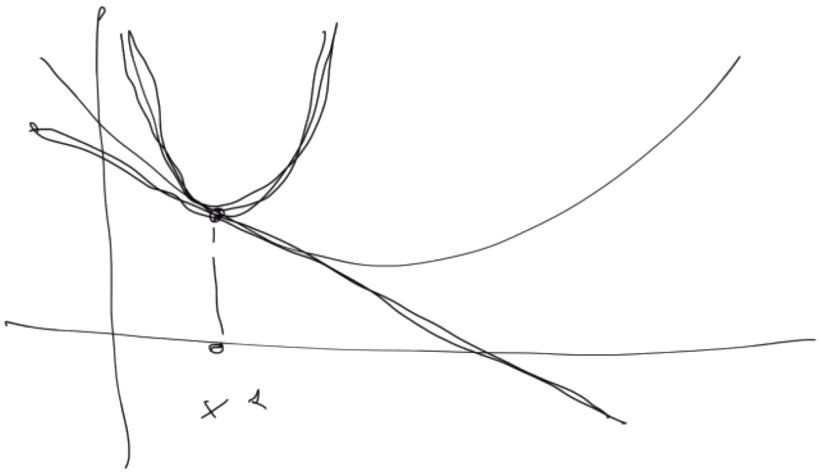

FIGURE 1.2: We build a linear model, which is a lowerbound for the function, then we build a quadratic model, which is above our function.

If we choose γ optimally then the quadratic approximation is very close to the function. We want to find the value for γs that gives best results in the worst case.

We can prove that the error $\|f(x^i) - f_*\| \leq \sigma^i(f(x^i) - f_*) \searrow 0$ is multiplied by this factor δ_i, that goes like the inverse of i^2.

Notice that if we choose α small, it will always be small.

The convergence is sublinear.

Theorem 1.1 (Optimality of accelerated gradient)**.** *If the function isn't strongly convex no algorithm has better convergence than* $|f(x^i) - f_*| = 3L\frac{\left\|x^1 - x_*\right\|^2}{32(i+1)^2}$.

Observation 1.1. *This theorem tells us that this algorithm never gets worse than* $|f(x^i) - f_*| = 3L\frac{\left\|x^1 - x_*\right\|^2}{32(i+1)^2}$*, but this doesn't imply that this method is fast on average. The state of the art provides a lot of different formulas for* β*, which of the ones leads to some theoretical results.*

From now on we will move towards a different family of functions, that aren't even differentiable, hence we can't compute the gradient.

1.2 Incremental gradient methods

This method has good performances in real world machine learning cases, where the function is differentiable but we do not want to compute the gradient.

Let $I = \{1, \dots, m\}$ be the set of observations, $X = [X^i \subset \mathcal{X}]_{i \in I}$ the set of inputs and $y = [y^i]_{i \in I}$ the set of outputs. Our goal is to explain y from X.

Since these vectors are uniformly distributed over the space (at least this is our hypotesis) when they get summed we expect some of them to cancel out.

The idea is to rewrite the problem as learning a linear function in the feature space, formally $\min \{ \sum_{i \in I} l(y^i, < \Phi(X^i), w >) \ : \ w \in \mathbb{R}^n \}$, where $l(\cdot, \cdot) =$ is called **loss function**.

We are now interested in computing the gradient, which is not hard to compute since the function is linear: $\nabla f(w) = \sum_{i \in I} \nabla f^i(w) = \sum_{i \in I} -A^i(y^i - A^i w)$.

The issue here is that computing the gradient, although it's the gradient of a very simple function, takes too long to be computed (since there are too many vectors in machine learning datasets).

To overcome this problem we choose to restrict to only a subset of observations. How can we choose such set? Randomly, of course.

At this point the algorithm isn't deterministic anymore, but it's completely **stochastic**.

The intuition behind this algorithm is to take only one observation, compute what is needed on this observation and make a small step.

An online aplication may be a sensor that produces hundreds of outputs per second and it's not possible to store each of them. They should be used to infer some information and then thrown away.

We study the converge of this kind of algorithms from a stochastic point of view.

In machine learning we always need some regularization, because the tuning of hyperparameters clearly takes into account only the error in the samples that have been seen. Let us regularize the model as follows

$$\min \{ \sum_{i \in I} l(y^i, < \Phi(X^i), w >) + \mu \Omega(w) : w \in \mathbb{R}^n \}$$

The usage of regularization may be useful, since we want to keep close to the minimum, but not reach it. It's enough to change slightly the problem and then solve it.

The regularization hyperparameter $\Omega(w)$ may be chosen as follows:

1. Lasso regularizer (best known): $\Omega(w) = \|w\|_1$;
2. In order to increase sparsity: $\Omega(w) = \|w\|^{2^2}$;
3. Leading to feature selection: many $w_j = 0$ as possible.

Ω function is not differentiable, so the function gets non differentiable, as an example look at Figure 1.3, which represents the plot of $f(w_1, w_2) = (3w_1 + 2w_2 - 2)^2 + 10(|w_1| + |w_2|)$.

1.3 Subgradient methods

These methods are thought for convex functions that are not differentiable.

Let us consider a kinky point: how can we choose between all the subgradients of that point? We assume to be able to compute some subgradients; since the function is convex we may recall

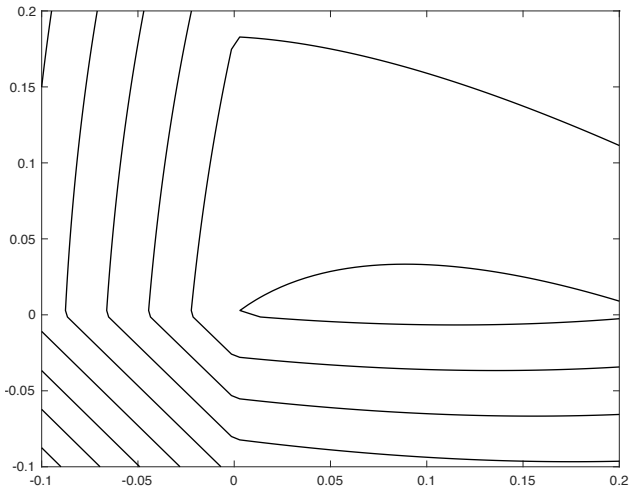

FIGURE 1.3: This function has a lot of kinky points.

Property 1.2. *Let f be a convex function, $\forall g \in \partial f(x), \forall\ y \in \mathbb{R}^n g(y) \equiv f(y) \geq f(x) + g(y - x)$.*

Let x^* be the optimum, $< x^* - x^i, g >$ is smaller than 0. This means that the angle between x and $x*$ is acute. If we knew the exact direction $x - x^*$ the line search would land on x^*. For a pictorial representation see Figure 1.4

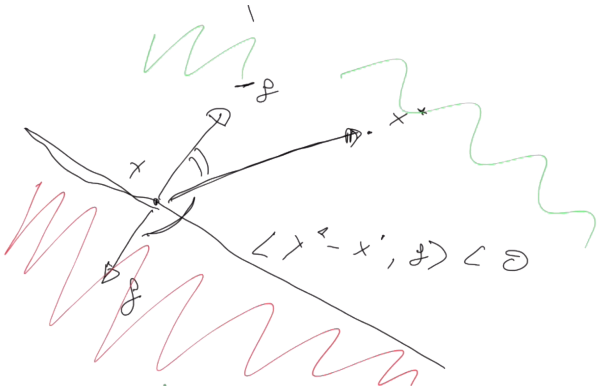

FIGURE 1.4: Since we know that $< g, x^* >$ is negative we get that the angle between g and x^* is larger than 90°. Moreover, we know where the optimum is not (red region), hence we restrict to the green region. At this point we will perform a line search on g direction. Since the function is not smooth, the line search may not succeed since the value of the function along the half line from x in g direction may remain the same, but there may be some points there that are closer to x^*.

The intuition behind this algorithm is to move in the direction of $-g$, but with a small α^i, because if the step is too large we may end up in a point which is actually further from x^* than the previous step. In that point we may find a point where perform line search, because

that point isn't kinky. In this context we won't try to minimize $\|f(x) - f(x^*)\|$, but we will minimize $\|x - x^*\|$, because the function may zigzag near that point. It goes without saying that choosing a too small value for α leads to a too slow convergence speed.