Proximal Algorithms

The subgradient algorithm is one generalization of gradient descent. It is simple, but the convergence is typically very slow (and it does not even converge in general for a fixed stepsize). The essential reason for this was that there are plenty of subgradients that are large near and even at the solution.

In this section, we will look at another iteration that finds the solution to an unconstrained convex program. The **proximal algorithm** or **proximal point method** can be understood in many different ways, but we will start by thinking about as a regularizer that naturally damps its influence as the iterations proceed, and as a twist on gradient descent.

Our goal is to solve the unconstrained optimization program

$$\underset{\boldsymbol{x}\in\mathbb{R}^N}{\text{minimize}}\, f(\boldsymbol{x}) \tag{1}$$

where f is convex but not necessarily smooth. Proximal point methods use the following iteration:

$$\boldsymbol{x}^{(k+1)} = \arg\min_{\boldsymbol{x}\in\mathbb{R}^N} \left(f(\boldsymbol{x}) + \frac{1}{2t_k}\|\boldsymbol{x}-\boldsymbol{x}^{(k)}\|_2^2 \right). \tag{2}$$

When f is convex, $f(\boldsymbol{x})+\alpha\|\boldsymbol{x}-\boldsymbol{z}\|_2^2$ is strictly convex for all $\alpha>0$ and $\boldsymbol{z}\in\mathbb{R}^N$, so the mapping from $\boldsymbol{x}^{(k)}$ to $\boldsymbol{x}^{(k+1)}$ is well-defined. We will sometimes use the "prox operator" to denote this mapping:

$$\text{prox}_{tf}(\boldsymbol{z}) = \arg\min_{\boldsymbol{x}\in\mathbb{R}^N} \left(f(\boldsymbol{x}) + \frac{1}{2t}\|\boldsymbol{x}-\boldsymbol{x}^{(k)}\|_2^2 \right).$$

At this point, you would be forgiven for wondering what all the fuss is about. We have taken the unconstrained problem (1) and turned

it into a series of constrained problems (2). For this to make sense, this program would have to be easier to solve for some reason. This can certainly be the case, and we will will see an example in the next section. One way to think about the additional $\frac{1}{2t_k}\|\boldsymbol{x} - \boldsymbol{x}^{(k)}\|$ is as a *regularizer* whose influence naturally disappears as we approach the solution, even for a fixed "step size" $t_k = t$. Computationally, the smoothed problem can be much easier to solve.

But past this, we will use our discussion here to build up to the so-called *proximal gradient* algorithm, discussed in the next set of notes, which is a great way to minimize an unconstrained function that can be broken into a smooth part and a nonsmooth part. A very nice a detailed review of proximal algorithms can be found in [PB14].

Least squares

Suppose we want to solve the standard least-squares problem

$$\underset{\boldsymbol{x}\in\mathbb{R}^N}{\text{minimize}} \|\boldsymbol{y} - \boldsymbol{A}\boldsymbol{x}\|_2^2.$$

When $\boldsymbol{A}$ has full column rank, we know that the solution is given by $\hat{\boldsymbol{x}}_{\text{ls}} = (\boldsymbol{A}^{\text{T}}\boldsymbol{A})^{-1}\boldsymbol{A}^{\text{T}}\boldsymbol{y}$. However, we also know that when $\boldsymbol{A}^{\text{T}}\boldsymbol{A}$ is not well-conditioned, this inverse can be unstable to compute, and iterative descent methods (gradient descent and conjugate gradients) can take many iterations to converge.

Consider the proximal point iteration (with fixed $t_k = t$) for solving this problem:

$$\boldsymbol{x}^{(k+1)} = \arg\min_{\boldsymbol{x}\in\mathbb{R}^N} \left(\frac{1}{2}\|\boldsymbol{y} - \boldsymbol{A}\boldsymbol{x}\|_2^2 + \frac{1}{2t}\|\boldsymbol{x} - \boldsymbol{x}^{(k)}\|_2^2 \right).$$

Here we have the closed form solution

$$\begin{aligned}\boldsymbol{x}^{(k+1)} &= (\boldsymbol{A}^{\mathrm{T}}\boldsymbol{A} + \delta\mathbf{I})^{-1}(\boldsymbol{A}^{\mathrm{T}}\boldsymbol{y} + \delta\boldsymbol{x}^{(k)}), \quad \delta = \frac{1}{t} \\ &= \boldsymbol{x}^{(k)} + (\boldsymbol{A}^{\mathrm{T}}\boldsymbol{A} + \delta\mathbf{I})^{-1}\boldsymbol{A}^{\mathrm{T}}(\boldsymbol{y} - \boldsymbol{A}\boldsymbol{x}^{(k)}).\end{aligned}$$

Now each step is equivalent to solving a least-squares problem, but this problem can be made well-conditioned by choosing δ (t) appropriately. The iterations above will converge to $\hat{\boldsymbol{x}}_{\mathrm{ls}}$ for any value of t; as we decrease t (increase δ), the number of iterations to get within a certain accuracy of $\hat{\boldsymbol{x}}_{\mathrm{ls}}$ increases, but the least-squares problems involved are all very well conditioned. For t very small, we are back at gradient descent (with stepsize t).

This is actually a well-known technique in numerical linear algebra called *iterative refinement*.

Implicit gradient descent ("backward Euler")

The proximal point method can also be interpreted as a variation on gradient descent. To see this, let us return for a moment to the differential equations for the "gradient flow" of f:

$$\boldsymbol{x}'(t) = -\nabla f(\boldsymbol{x}), \quad \boldsymbol{x}(0) = \boldsymbol{x}_0. \tag{3}$$

The equilibrium points for this system are the $\boldsymbol{x}$ such that $\nabla f(\boldsymbol{x}) = \mathbf{0}$, which are precisely the minimizers for $f(\boldsymbol{x})$.

As we first saw in the notes on the heavy ball method, we can interpret gradient descent as a first-order numerical method for tracing the path from $\boldsymbol{x}_0$ to a solution $\boldsymbol{x}^\star$. This comes from discretizing the derivative on the right using a forward finite difference:

$$\frac{\boldsymbol{x}(t+h) - \boldsymbol{x}(t)}{h} \approx -\nabla f(\boldsymbol{x}(t)) \quad \text{for small } h.$$

Thus the gradient descent iterations

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

approximate the solution at equispaced times spaced h seconds apart — the stepsize in gradient descent can be interpreted as the time scale to which we are approximating the derivative. This is known as the *forward Euler method* for discretizing (3).

But now suppose we used a *backward difference* to approximate the derivative:

$$\frac{\boldsymbol{x}(t) - \boldsymbol{x}(t-h)}{h} \approx -\nabla f(\boldsymbol{x}(t)) \quad \text{for small } h.$$

Now the iterates must obey

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

This is known as the *backward Euler method* for discretizing (3). Computing the iterates is not as straightforward — we can just compute the gradient at the current point, we have to find the next point by finding a $\boldsymbol{x}^{(k+1)}$ that obeys the equation above.

This is exactly what the proximal operator does. If f is differentiable, then

$$\begin{aligned} \boldsymbol{x}^{(k+1)} &= \arg\min_{\boldsymbol{x}\in\mathbb{R}^N} \left(f(\boldsymbol{x}) + \frac{1}{2t}\|\boldsymbol{x} - \boldsymbol{x}^{(k)}\|_2^2 \right) \\ &\Updownarrow \\ \mathbf{0} &= \nabla f(\boldsymbol{x}^{(k+1)}) + \frac{1}{t}(\boldsymbol{x}^{(k+1)} - \boldsymbol{x}^{(k)}). \end{aligned} \tag{4}$$

So the proximal point method can be interpreted as a backward Euler discretization for gradient flow.

Note that we assumed the differentiability of f above purely for illustration; we can compute the prox operator whether or not f has a gradient.

Convergence

We have yet to show that the iteration

$$\boldsymbol{x}^{(k+1)} = \text{prox}_{tf}(\boldsymbol{x}^{(k)})$$

actually converges to a minimizer of f. We will do this now, and see that the convergence guarantees (at least in terms of number of iterations) are much nicer than they are for the subgradient method.

First question: If we are at a solution $\boldsymbol{x}^\star$, does applying the proximal operator move us off of this solution?

Answer: No. $\boldsymbol{x}^\star$ is a minimizer of convex $f(\boldsymbol{x})$ if and only if

$$\boldsymbol{x}^\star = \text{prox}_{tf}(\boldsymbol{x}^\star)$$

for every $t > 0$. That is, minimizers of f are fixed points of the proximal operator. Just like applying another iteration of gradient descent at the minimizer of a smooth function moves you nowhere (since the gradient is zero), applying the prox operator to the minimizer of a general convex function does nothing. This is in stark contrast to the subgradient method, where we can definitely move off a solution.

The fixed-point property can be established with a quick argument. If $\boldsymbol{x}^\star$ is a minimizer of f, then

$$\begin{aligned} \mathbf{0} &\in \partial f(\boldsymbol{x}^\star) \\ &= \partial f(\boldsymbol{x}^\star) + \frac{1}{t}(\boldsymbol{x}^\star - \boldsymbol{x}^\star) \end{aligned}$$

and so $\mathbf{0}$ is also in the subdifferential of $f(\boldsymbol{x})+\frac{1}{2t}\|\boldsymbol{x}-\boldsymbol{x}^\star\|_2^2$ at $\boldsymbol{x}=\boldsymbol{x}^\star$. The converse is now also clear, as if $\mathbf{0}$ is in the subdifferential of $f(\boldsymbol{x})+\frac{1}{2t}\|\boldsymbol{x}-\boldsymbol{x}^\star\|_2^2$ at $\boldsymbol{x}=\boldsymbol{x}^\star$, then

$$\begin{aligned} \mathbf{0} &\in \partial f(\boldsymbol{x}^\star) + \frac{1}{t}(\boldsymbol{x}^\star - \boldsymbol{x}^\star) \\ &\Downarrow \\ \mathbf{0} &\in \partial f(\boldsymbol{x}^\star), \end{aligned}$$

and so $\boldsymbol{x}^\star$ is a minimizer of f.

We can also bound the number of steps it takes for the proximal point method to achieve a certain accuracy, just as we have for the other unconstrained minimization algorithms that we have encountered. Under the conditions that f is a closed convex function (so that the prox function is well-defined), and that there exists a minimizer $\boldsymbol{x}^\star$, we will show below that the iterations $\boldsymbol{x}^{(k+1)} = \mathrm{prox}_{t_k f}(\boldsymbol{x}^{(k)})$ obey

$$f(\boldsymbol{x}^{(k)}) - f^\star \;\leq\; \frac{\|\boldsymbol{x}^{(0)} - \boldsymbol{x}^\star\|_2^2}{2\sum_{i=1}^k t_i} \quad \text{for all} \;\; k \geq 1, \tag{5}$$

where $f^\star = f(\boldsymbol{x}^\star)$. From here we can see that

$$t_k = t \quad \Rightarrow f(\boldsymbol{x}^{(k)}) - f^\star = O(1/k),$$

that is, we are guaranteed to get to a point with $\boldsymbol{x}^{(k)}$ such that $f(\boldsymbol{x}^{(k)}) - f^\star \leq \epsilon$ in $O(1/\epsilon)$ iterations.

In looking at (5), it seems that we can make the convergence as fast as we want by making the t_k very large. This is true, but remember that each step itself involves solving an optimization program, and the cost of solving this program might rely critically on the t_k. As t_k gets large, we are effectively solving the original program itself.

We will establish (5) in three steps:

1. We will show that $f(\boldsymbol{x}^{(i)})$ is non-increasing
2. We will also show that

$$f(\boldsymbol{x}^{(i)}) - f^\star \;\leq\; \frac{1}{2t_i}\left(\|\boldsymbol{x}^{(i-1)} - \boldsymbol{x}^\star\|_2^2 - \|\boldsymbol{x}^{(i)} - \boldsymbol{x}^\star\|_2^2\right)$$

3. Combining the two facts above gives us

$$\begin{aligned}\left(\sum_{i=1}^k t_i\right)\left(f(\boldsymbol{x}^{(k)}) - f^\star\right) &\leq \sum_{i=1}^k t_i\left(f(\boldsymbol{x}^{(i)}) - f^\star\right)\\ &\leq \frac{1}{2}\|\boldsymbol{x}^{(0)} - \boldsymbol{x}^\star\|_2^2\end{aligned}$$

We will start by looking at what happens at a single iteration. Define

$$\boldsymbol{g}^{(i)} = \frac{1}{t}\left(\boldsymbol{x}^{(i)} - \mathrm{prox}_{tf}(\boldsymbol{x}^{(i)})\right).$$

Thus $\boldsymbol{g}^{(i)}$ is the (negative of) the direction we move in going from $\boldsymbol{x}^{(i)}$ to $\boldsymbol{x}^{(i+1)}$:

$$\boldsymbol{x}^{(i+1)} = \boldsymbol{x}^{(i)} - t\boldsymbol{g}^{(i)}.$$

Since we are just going to study this one iteration, we will make the notation cleaner by dropping the superscript, taking $\boldsymbol{g} := \boldsymbol{g}^{(i)}$ and $t = t_i$ below.

Since

$$\boldsymbol{x}^{(i+1)} = \arg\min_{\boldsymbol{x}\in\mathbb{R}^N}\left(f(\boldsymbol{x}) + \frac{1}{2t}\|\boldsymbol{x} - \boldsymbol{x}^{(i)}\|_2^2\right),$$

the optimality conditions for the program on the right tell us that

$$\boldsymbol{0} \in \partial f(\boldsymbol{x}^{(i+1)}) + \frac{1}{t}\left(\boldsymbol{x}^{(i+1)} - \boldsymbol{x}^{(i)}\right),$$

which means

$$\boldsymbol{g} \in \partial f(\boldsymbol{x}^{(i+1)}).$$

By the definition of subgradient, then,

$$\begin{aligned} f(\boldsymbol{x}^{(i+1)}) &\leq f(\boldsymbol{z}) + \boldsymbol{g}^{\mathrm{T}}(\boldsymbol{x}^{(i+1)} - \boldsymbol{z}) \\ &= f(\boldsymbol{z}) + \boldsymbol{g}^{\mathrm{T}}(\boldsymbol{x}^{(i)} - \boldsymbol{z} - t\boldsymbol{g}) \\ &= f(\boldsymbol{z}) + \boldsymbol{g}^{\mathrm{T}}(\boldsymbol{x}^{(i)} - \boldsymbol{z}) - t\|\boldsymbol{g}\|_2^2 \quad \text{for all } \boldsymbol{z} \in \mathbb{R}^N. \end{aligned} \tag{6}$$

In particular, if we evaluate the above at $\boldsymbol{x} = \boldsymbol{x}^{(i)}$, then

$$f(\boldsymbol{x}^{(i+1)}) \;\leq\; f(\boldsymbol{x}^{(i)}) - t\|\boldsymbol{g}\|_2^2.$$

Thus $f(\boldsymbol{x}^{(i+1)}) \leq f(\boldsymbol{x}^{(i)})$, and property 1 above holds.

To get property 2, we evaluate (6) at $\boldsymbol{z} = \boldsymbol{x}^\star$ to get

$$f(\boldsymbol{x}^{(i+1)}) - f^\star \leq \boldsymbol{g}^{\mathrm{T}}(\boldsymbol{x}^{(i)} - \boldsymbol{x}^\star) - t\|\boldsymbol{g}\|_2^2.$$

Now we "complete the square" on the right. Since $t > 0$,

$$\begin{aligned} f(\boldsymbol{x}^{(i+1)}) - f^\star &\leq \boldsymbol{g}^{\mathrm{T}}(\boldsymbol{x}^{(i)} - \boldsymbol{x}^\star) - \frac{t}{2}\|\boldsymbol{g}\|_2^2 \\ &= \frac{1}{2t}\left(2t\boldsymbol{g}^{\mathrm{T}}(\boldsymbol{x}^{(i)} - \boldsymbol{x}^\star) - t^2\|\boldsymbol{g}\|_2^2\right) \\ &= \frac{1}{2t}\left(-\|\boldsymbol{x}^{(i)} - \boldsymbol{x}^\star - t\boldsymbol{g}\|_2^2 + \|\boldsymbol{x}^{(i)} - \boldsymbol{x}^\star\|_2^2\right) \\ &= \frac{1}{2t}\left(\|\boldsymbol{x}^{(i)} - \boldsymbol{x}^\star\|_2^2 - \|\boldsymbol{x}^{(i+1)} - \boldsymbol{x}^\star\|_2^2\right). \end{aligned}$$

Thus property 2 above holds.

Accelerated proximal points algorithms

We can "accelerate" the proximal point method in the same way we accelerated gradient descent: by adding "feedback" from previous iterations. This result is essentially contained in [BT09]; we will discuss it more in the next section when we talk about the proximal gradient algorithm.

The essential iteration is as follows:

$$\begin{aligned}\boldsymbol{x}^{(1)} &= \text{prox}_{t_1 f}(\boldsymbol{x}^{(0)}),\\ \boldsymbol{x}^{(k)} &= \text{prox}_{t_k f}\left(\boldsymbol{x}^{(k-1)} + s_k(\boldsymbol{x}^{(k-1)} - \boldsymbol{x}^{(k-2)})\right),\end{aligned}$$

where the t_k are given as before, and the s_k are computed as

$$s_k = \theta_k\left(\frac{1}{\theta_{k-1}} - 1\right), \quad \text{where } \theta_k \text{ obeys } \quad \frac{\theta_k^2}{\theta_{k-1}^2} = (1-\theta_k)\frac{t_k}{t_{k-1}}.$$

So we are adding a little bit of momentum to $\boldsymbol{x}^{(k-1)}$ before we compute the prox mapping. This makes sense intuitively, and we have seen that this type of idea leads to real gains in standard gradient descent. Deriving the step sizes above, though, is non-trivial, and we will omit it here.

The convergence results for the accelerated proximal method say that the iteration converges if the "step sizes" t_k are chosen such that $\sum_i \sqrt{t_i} \to \infty$. For constant step size, we have $O(1/k^2)$ convergence, which is a significant improvement over the standard proximal point method.

Moreover, this acceleration comes at practically zero additional computational cost. All we have to do is store an extra iterate in memory.

References

[BT09] A. Beck and M. Teboulle. A fast iterative shrinkage-thresholding algorithm for linear inverse problems. *SIAM J. Imaging Sci.*, 2(1):183–202, 2009.

[PB14] N. Parikh and S. Boyd. Proximal algorithms. *Foundations and Trends in Optimization*, 1(3):123–231, 2014.