Statistical Machine Learning Notes 2

Empirical Risk Minimization and Optimization

Instructor: Justin Domke

1 Empirical Risk Minimization

Empirical Risk Minimization is a fancy sounding name for a very simple concept. Supervised learning can usually be seen as picking one function f from a set of possible functions F. An obvious questions is, how can we tell a good function f from a bad one? These notes introduce a general framework that applies for

First, we introduce the concept of a **loss function** L. Given some particular pair of inputs $\mathbf{x}$ and outputs $\mathbf{y}$,

$$L(f(\mathbf{x}), \mathbf{y})$$

tells us how much it "hurts" to make the prediction $f(\mathbf{x})$ when the true output is $\mathbf{y}$.

Now, let us define the (true) **risk**

$$R_{\text{true}}(f) = E_p\big[L(f(\mathbf{x}), \mathbf{y})\big] = \int\int p(\mathbf{x}, \mathbf{y}) L(f(\mathbf{x}), \mathbf{y}) d\mathbf{x} d\mathbf{y}.$$

Here p is the *true* distribution over the inputs $\mathbf{x}$ and $\mathbf{y}$. The risk measures how much, on average, it hurts to use f as our prediction algorithm.

This can all be made clear by considering an example. Suppose we want to fit a function for predicting if it will rain or not. The input $\mathbf{x}$ will be the sky: CLEAR, CLOUDY, or MIXED. The output $\mathbf{y}$ will be either, RAIN (when it rains) or NOPE (when it doesn't rain). The loss function is now a function $L : \{\text{RAIN}, \text{NOPE}\}^2 \to \Re$.

What loss function is appropriate? It is important to realize that this cannot be answered by math. The loss function depends on the priorities of the user. For example, if you are a person who really hates getting wet, but doesn't particularly mind carrying an umbrella on a clear day, you might use a loss function like:

$L_{\text{hate-rain}}(Y_1, Y_2)$

$Y_1 \backslash Y_2$	RAIN	NOPE
RAIN	0	1
NOPE	25	0

Meaning you hate getting rained on 25 times as much as carrying an umbrella on a clear day. On the other hand, someone who loses things frequently might hate carrying an umbrella on a clear day. They might have a loss function more like:

$L_{\text{hate-umbrellas}}(Y_1, Y_2)$

$Y_1 \backslash Y_2$	RAIN	NOPE
RAIN	0	1
NOPE	1	0

Now, let's suppose that the true distribution p is given as follows:

$p(\mathbf{x}, \mathbf{y})$

$\mathbf{x} \backslash \mathbf{y}$	RAIN	NOPE
CLEAR	0	1/4
CLOUDY	1/4	0
MIXED	1/6	1/3

Let's consider two possible prediction functions

$$f_1(\mathbf{x}) = \begin{cases} \text{CLEAR} & \text{NOPE} \\ \text{CLOUDY} & \text{RAIN} \\ \text{MIXED} & \text{NOPE} \end{cases}$$

$$f_2(\mathbf{x}) = \begin{cases} \text{CLEAR} & \text{NOPE} \\ \text{CLOUDY} & \text{RAIN} \\ \text{MIXED} & \text{RAIN} \end{cases}$$

If we use $L_{\text{hate-rain}}$, it is easy to calculate that $R(f_1) = 1/6 \cdot 25$, and $R(f_2) = 1/6 \cdot 1$, and so f_2 has the lower risk. Meanwhile, if we use $L_{\text{hate-umbrellas}}$, we can calculate $R(f_1) = 1/6 \cdot 1$, and $R(f_2) = 1/3 \cdot 1$, and so f_1 has the lower risk.

So, it sounds like the thing to do is to pick f to minimize the risk. Trouble is, that is impossible. To calculate the risk, we would need to know the true distribution p. We don't have the true distribution– if we did, we wouldn't be doing machine learning. So, what should we do?

Since the data D comes from p, we should be able to get a reasonable approximation

$$E_p L(f(\mathbf{x}), \mathbf{y}) \approx \frac{1}{N} \sum_{\hat{\mathbf{x}}, \hat{\mathbf{y}}} L(f(\hat{\mathbf{x}}), \hat{\mathbf{y}}). \tag{1.1}$$

The right hand side of Eq. 1.1 is called the **empirical risk**.

$$R(f) = \frac{1}{N} \sum_{\hat{\mathbf{x}}, \hat{\mathbf{y}}} L(f(\hat{\mathbf{x}}), \hat{\mathbf{y}}).$$

Picking the function f^* that minimizes it is known as **empirical risk minimization**.

$$f^* = \arg\min_{f \in F} R(f)$$

Two points of caution here: First, how well the empirical risk approximates the true risk, obviously, will depend on the amount of data. Second, notice that empirical risk minimization has no safeguards to resist overfitting. When used in practice, it is usually necessary to perform some sort of model selection or regularization to make empirical risk minimization generalize well to new data. We will talk about these things much more later on, when discussing specific classifiers.

2 Convex Optimization

The rest of these notes discus specific algorithms for optimization. From now on, $f(\mathbf{x})$ denotes a generic function we want to optimize. Notice that this is different from the previous section, which used $f(\mathbf{x})$ to denote a possible predictor. The clash of notation is unfortunate, but both of these usages are very standard (and used in the supplementary readings).

A generic optimization problem is of the form

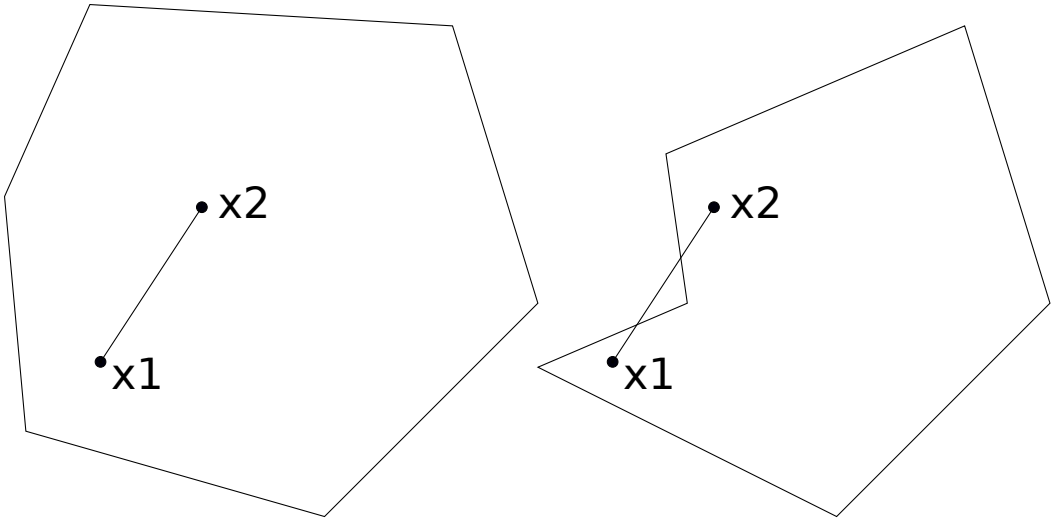
$$\begin{aligned} \min_{\mathbf{x}} . \quad & f(\mathbf{x}) \\ \text{s.t.} \quad & \mathbf{x} \in C. \end{aligned} \tag{2.1}$$

That is, one should minimize the function f, subject to the constraint that $\mathbf{x}$ is in the set C. The trouble with a problem like this is that it is very easy to write down optimization problems like in Eq. 2.1 that are essentially impossible to solve. Convex optimization is a subset of optimizations that, roughly speaking, we can solve.

A set C is **convex** if for $\mathbf{x}_1$,$\mathbf{x}_2 \in C$,

$$\theta \mathbf{x}_1 + (1 - \theta)\mathbf{x}_2 \in C, \, 0 \leq \theta \leq 1.$$

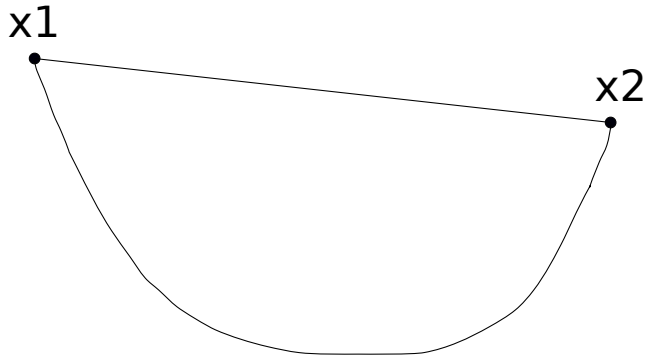
This has a simple geometrical interpretation: Given any two points in the set, the line segment joining them must be in the set. So the set below on the left is convex, while the one on the right is not.



Now, a function $f(\mathbf{x})$ is **convex** over C if for $\mathbf{x}_1$,$\mathbf{x}_2 \in C$,

$$f(\theta\mathbf{x}_1 + (1-\theta)\mathbf{x}_2) \leq \theta f(\mathbf{x}_1) + (1-\theta)f(\mathbf{x}_2),\ 0 \leq \theta \leq 1.$$

Geometrically, this means the line joining any two function values must lie above the function itself.



Notice that a linear function is convex.

A function is called **strictly convex** if the inequality is strictly satisfied for all points not at the ends.

$$f(\theta\mathbf{x}_1 + (1-\theta)\mathbf{x}_2) < \theta f(\mathbf{x}_1) + (1-\theta)f(\mathbf{x}_2),\ 0 < \theta < 1.$$

A linear function is *not* strictly convex.

Convex optimization problems can usually be solved reliably. The basic reason is that if you identify a local minima in a convex optimization problem, there is no possibility that some other, better, minima exists elsewhere.

3 Unconstrained Optimization

In this section, we discuss some standard algorithms for unconstrained problems. We also assume below that the function f we are optimizing is twice differentiable. Ignore this assumption at your peril! If you apply these methods to a seemingly innocent function like $f(\mathbf{x}) = |\mathbf{x}|_1 = \sum_i |x_i|$, they will not work.

Aside: One of the common ways of dealing with non-differentiable unconstrained optimization problems is to reformulate them as differentiable constrained optimizations. For example, instead of minimizing $f(\mathbf{x})$ above, we could minimize (with respect to both $\mathbf{z}$ and $\mathbf{x}$) $f(\mathbf{z}) = \sum_i z_i$, s.t. $\mathbf{z} \geq \mathbf{x}, \mathbf{z} \geq -\mathbf{x}$. We will see some examples of this when we talk about Kernel Methods.

3.1 Descent Methods

In these notes, we will discuss a few basic optimization algorithms. All of these algorithm can be seen as slight variants of a descent method.

Generic Descent Method

Repeat:

- Find a descent direction $\Delta\mathbf{x}$.
- Find a step size t.
- $\mathbf{x} \leftarrow \mathbf{x} + t\Delta x$

A direction $\Delta\mathbf{x}$ is called a descent direction at $\mathbf{x}$ if the angle between $\Delta\mathbf{x}$ and the gradient ∇f is less than 90 degrees, or equivalently if

$$\Delta\mathbf{x}^T \nabla f(\mathbf{x}) < 0.$$

The way to understand this is that for very small t, $f(\mathbf{x} + t\Delta x) \approx f(\mathbf{x}) + t\Delta\mathbf{x}^T \nabla f(\mathbf{x})$. So, a descent direction ensures that we can find a positive t that we make progress in reducing f.

Here, we will always choose steps t in the same way, through a...

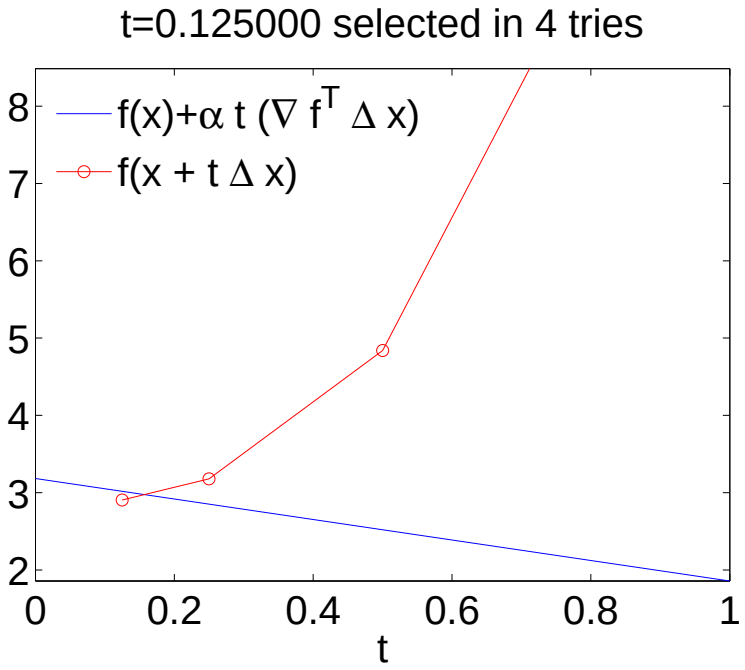
Backtracking Line Search $(0 < \alpha < \frac{1}{2}\ ,\ 0 < \beta < 1)$

- $t \leftarrow 1$
- While $f(\mathbf{x} + t\Delta\mathbf{x}) > f(\mathbf{x}) + \alpha t \nabla f(\mathbf{x})^T \Delta\mathbf{x}$:

– Set $t \leftarrow \beta t$

The intuition for this is that we start with $t = 1$, and keep making it smaller by a factor of β until we are "happy". When should we be happy? We expect that for small t, $f(\mathbf{x} + t\Delta\mathbf{x}) \approx f(\mathbf{x}) + t\Delta\mathbf{x}^T \nabla f(\mathbf{x})$. We are willing to tolerate less improvement than this, and so we add the factor of α.

Here is an example with $\alpha = \frac{1}{4}$ and $\beta = \frac{1}{2}$.



In some special circumstances, it is possible to choose steps through an exact line search, finding $\min_t f(\mathbf{x}+t\Delta\mathbf{x})$. When it can be done, this does make optimization somewhat faster.

3.2 Gradient Descent

We will discuss three algorithms in these notes. The only difference between these is in how the descent direction $\Delta\mathbf{x}$ is selected. Gradient descent takes the simplest approach, of using the negative gradient of f

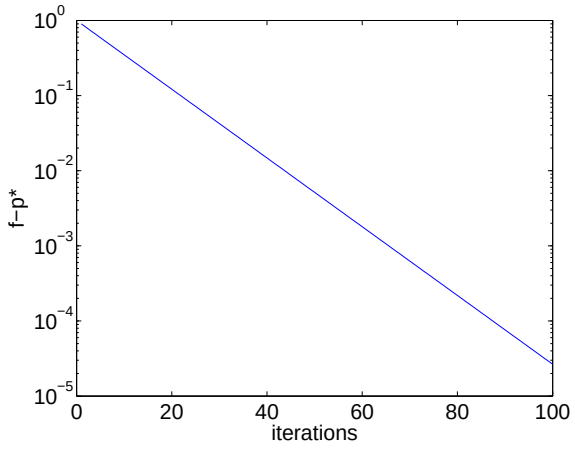
$$\Delta\mathbf{x} = -\nabla f(\mathbf{x}).$$

This is the direction that most quickly decreases f in the immediate neighborhood of $\mathbf{x}$.

Now, let the specific values of $\mathbf{x}$ that are generated by the algorithm be $\mathbf{x}^{(0)}, \mathbf{x}^{(1)}, \mathbf{x}^{(2)}$, etc. It can be shown that, for strictly convex functions, for gradient descent

$$f(\mathbf{x}^{(k)}) - p^* \leq c\big(f(\mathbf{x}^{(k-1)}) - p^*\big). \tag{3.1}$$

This is called **linear convergence**. Here, $0 < c < 1$ is a number that depends (a lot) on properties of the function f and (a little) on the parameters of the line search. To understand this, we can think of $f(\mathbf{x}^{(k)}) - p^*$ as how suboptimal the function is at iterate k. At each iteration, this gets multiplied by a number less than one, and so will eventually go to zero.



At first glance, Eq. 3.1 looks very nice. And, it is true, in many real problems, gradient descent works well. It may be that, owing to its simplicity, it is the single most widely used optimization algorithm in the world. Unfortunately, on many other problems, c is very close to 1, and so gradient descent is slow slow as to be unusable.

We can develop some intuition about gradient descent by considering some examples. Consider the quadratic function

$$f(\mathbf{x}) = \frac{1}{2}\big(x_1^2 + a x_2^2\big).$$

Imagine we start at $\mathbf{x} = (1, 1)$. The best direction to proceed towards the minimum is $(-1, -1)$. However, the negative gradient will point in the direction $\Delta\mathbf{x} = (-1, -a)$. If a is very large, this means that the gradient will be almost orthogonal to the direction that leads to the minimum, and so many iterations will be required to reach the minimum.

The above example shows that gradient descent is not invariant to linear transformations. If we set $f'(\mathbf{x}) = f(A\mathbf{x})$ for some linear transformation A, gradient descent might minimize f quickly but f' slowly.

Conclusions:

- Gradient descent is simple, and is computationally fast per iteration.
- In many cases, gradient descent converges linearly.
- The number of iterations is hugely dependent on the scaling of the problem. Often it works well. Other times, it doesn't seem to converge at all. Playing around with the scaling of the parameters can improve things a great deal.

One final note. One can often get away with running gradient descent with a *fixed* step size t. To understand this, consider all the step sizes that would be selected by the backtracking line search for all points $\mathbf{x}$. If there is a minimum, then this minimum will assure convergence of the method. In practice, this step is usually selected by running gradient descent with a few different step sizes, and picking one that converges quickly.

3.3 Newton's Method

In Newton's method, we choose the step by multiplying the gradient with the inverse Hessian.

$$\Delta\mathbf{x} = -H(\mathbf{x})^{-1}\nabla f(\mathbf{x}), \quad H = \frac{\partial^2 f(\mathbf{x})}{\partial\mathbf{x}\partial\mathbf{x}^T}$$

If H is positive definite (one of the most common ways of demonstrating convexity), it is easy to show that this is a descent direction[1].

Why this? Let's approximate our function with a second-order Taylor expansion. For small $\mathbf{v}$,

$$f(\mathbf{x}+\mathbf{v}) \approx f(\mathbf{x}) + \mathbf{v}^T\nabla f(\mathbf{x}) + \frac{1}{2}\mathbf{v}^T H(\mathbf{x})\mathbf{v}.$$

It is easy to see[2] that the minimum of this is found when $\mathbf{v} = -H(\mathbf{x})^{-1}\Delta f(\mathbf{x})$.

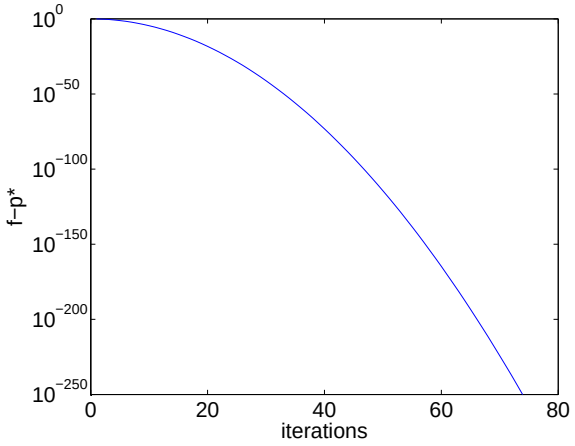
Notice that if f actually is a quadratic, the Taylor approximation is exact, and so the $\mathbf{x}+\Delta\mathbf{x}$ will be the global minimum. This intuition is extremely valuable in understanding Newton's method.

Given some technical assumptions, it can be shown that Newton's method goes through two stages. In the first stage, the line search may choose $t < 1$, and the objective is decreases linearly. Eventually, the method switches into the second mode, and takes "pure" Newton steps, with $t = 1$. Once the second condition occurs, the first does not occur again.

[1] $\Delta\mathbf{x}^T\nabla f(\mathbf{x}) = -\nabla f(\mathbf{x})^T H^{-1}\nabla f(\mathbf{x})$ The definition of positive definite is that for all $\mathbf{v}$, $\mathbf{v}^T H\mathbf{v} > 0$. Choosing $\mathbf{v} = H^{-1}\nabla f$, we have $\nabla f^T H^{-T} H H^{-1}\nabla f = \nabla f^T H^{-1}\nabla f > 0$. (Alternatively, observe that a positive definite matrix has a positive definite inverse, which was essentially proven in the previous sentence.)

[2] Take the derivative with respect to $\mathbf{v}$. This must be zero, when $\mathbf{v}$ is minimum, so $\nabla f(\mathbf{x}) + H(\mathbf{x})\mathbf{v} = 0$.

The second stage is quadratically convergent, meaning, roughly speaking, a downward slanting parabola on a plot of iterations vs. error. Roughly speaking, this means the number of correct digits doubles in each iterations. One practical implication of this is that if you bother to run Newton's method to any reasonable accuracy, it is pretty cheap to go ahead and run it all the way to machine precision.



One nice property of the Newton step is that it *is* invariant to linear changes of coordinates. If we define $f'(y) = f(Ty)$ and run Newton's method on f starting at $\mathbf{x}^{(0)}$ and f' starting at $\mathbf{y}^{(0)} = T\mathbf{x}^{(0)}$, then it can be shown that $T\mathbf{y}^{(k)} = \mathbf{x}^{(k)}$. Thus, the kind of playing around with parameters suggested above for gradient descent is totally unnecessary.

(**Optional remark**: Notice, however, that Newton's method is not invariant to, say, *quadratic* changes of coordinates. One often finds a great deal of contempt thrown in Gradient Descent's direction for not being invariant to linear changes of coordinates, with no mention that Newton's method is really just invariant to one higher order. One can design even higher-order, faster converging, methods that make of the tensor of third order derivatives $\partial^2 f/\partial x_i \partial x_j \partial x_k$– or higher. Your instructor is not aware of any examples of these methods being used in practice in machine learning, and couldn't even determine if they had an accepted name.)

So, Newton's method sounds great. Why would we ever use gradient descent? The main reason is that it can be very costly to create the Hessian matrix H and to solve the linear system $H^{-1}\nabla f(\mathbf{x})$ to get the Newton step. If $\mathbf{x}$ has N dimensions, H will be of size $N \times N$, and naive methods for solving the linear system take time $\mathcal{O}(N^3)$. In very large systems, this can be a huge cost.

Conclusions:

- Newton's method is considered the "gold standard" of optimization algorithms. If you can run it, it will usually reliably recover the solution to high accuracy in few iterations.

- The main downside of Newton's method is the expense and inconvenience of dealing with the Hessian matrix. For a dense matrix, the most reliable methods of finding the Newton step take time $\mathcal{O}(N^3)$ per iteration.

3.4 Quasi-Newton Methods

Given that Newton's method is so expensive, it is natural to try to find a cheaper and more convenient method that still converges faster than gradient descent. The basic idea of Quasi-Newton methods is to, at iteration k, approximate f by

$$f^k(\mathbf{x}+\mathbf{v}) = f(\mathbf{x}) + \Delta f(\mathbf{x})^T\mathbf{v} + \frac{1}{2}\mathbf{v}^T B_k \mathbf{v},$$

where B_k is an approximation of the Hessian matrix that we will continuously update to try to "track" the Hessian. The minimum of this approximation is at $-B_k^{-1}\nabla f(\mathbf{x})$, which we use as the descent direction.

$$\Delta\mathbf{x} = -B_k^{-1}\nabla f(\mathbf{x})$$

Now, how should we update B_k? Quasi-Newton methods do this by imposing that f^{k+1} should match the true gradient computed at the previous iteration, while remaining as "close" to B_k as possible. The condition that the gradients match turns out[3] to be equivalent to

$$B_{k+1}(\mathbf{x}^{(k+1)} - \mathbf{x}^{(k)}) = \nabla f(\mathbf{x}^{(k+1)}) - \nabla f(\mathbf{x}^{(k)})$$

(Be careful with the (k) and $(k+1)$ in this equation.) Quasi-Newton methods pick B_{k+1} by

$$\begin{array}{lll} & \min_B & ||B - B_k|| \\ \text{s.t.} & B = B^T & B_{k+1}(\mathbf{x}^{(k+1)} - \mathbf{x}^{(k)}) = \nabla f(\mathbf{x}^{(k+1)}) - \nabla f(\mathbf{x}^{(k)}) \end{array} \tag{3.2}$$

Various matrix norms can be used in Eq. 3.2, which yield different algorithms.

From this description, we still have to compute $B_k^{-1}\Delta f(\mathbf{x})$, which doesn't seem like too much of an improvement over Newton's method. However, by applying a clever identity known

[3]

$$\frac{d}{d\mathbf{x}} f^{(k+1)}(\mathbf{x}^{(k)}) = \nabla f(\mathbf{x}^{(k+1)}) - B_k(\mathbf{x}^{(k+1)} - \mathbf{x}^{(k)}) = \nabla f(\mathbf{x}^{(k)})$$

as the Sherman-Morrison formula, it is possible to do all updates directly in terms of the inverse of B_k, which we will denote by F_k. Then, the descent direction can be computed simply as a matrix multiply as $-F_k \nabla f(\mathbf{x})$ in time only $\mathcal{O}(N^2)$.

The most popular Quasi-Newton method is known as BFGS after Broyden, Fletcher, Goldfarb and Shanno, who all four independently published it in 1970!

Optional details: The exact update used in BFGS is

$$F_{k+1} = (I - \rho_k \mathbf{s}_k \mathbf{y}_k^T) F_k (I - \rho_k \mathbf{y}_k \mathbf{s}_k^T) + \rho_k \mathbf{s}_k \mathbf{s}_k^T$$

where $\mathbf{s}_k = \mathbf{x}_{k+1} - \mathbf{x}_k$, $\mathbf{y}_k = \nabla f(\mathbf{x}^{(k+1)}) - \nabla f(\mathbf{x}^{(k)})$, and $\rho_k = 1/\mathbf{y}_k^T \mathbf{s}_k$.

In general, Quasi-Newton methods converge faster than linearly, but not quadratically. However, given that they do not require computing the gradient, they are often the method of choice in practice.

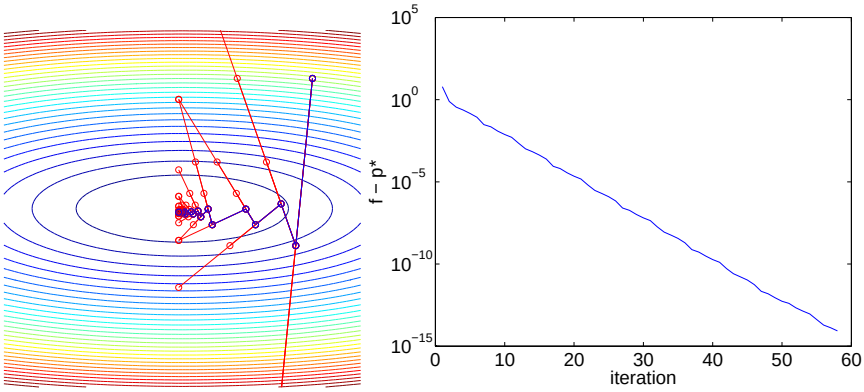
There is a variant of BFGS known as limited-memory BFGS. L-BFGS only stores the previous M gradients, and computes the descent direction $\Delta\mathbf{x}$ directly from these, using only $O(MN)$ time and space. Even using $M = 10$ often yields a significant over gradient descent in practice, and is usually affordable, even if the number of variables is very large.

3.5 Examples

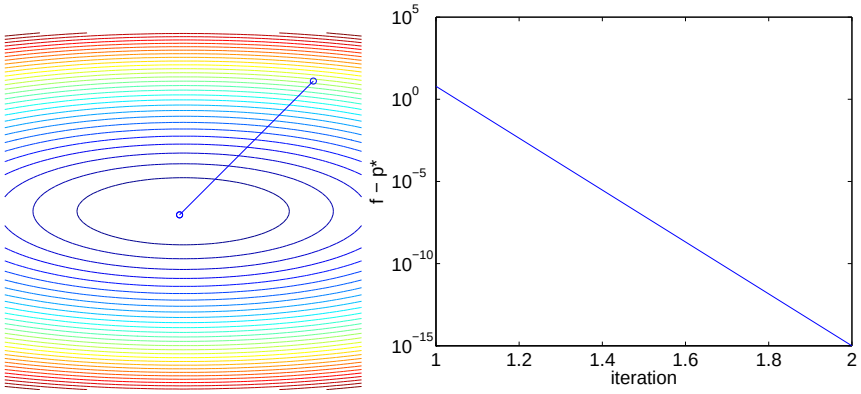
The section below shows the results of running gradient descent, Newton's method, and BFGS on three example objectives.

This is a two-dimensional optimization, of the function $f(\mathbf{x}) = x_1^2 + 10x_2^2$. Newton's method converges in a single iteration, while gradient descent is confused by the uneven scaling.

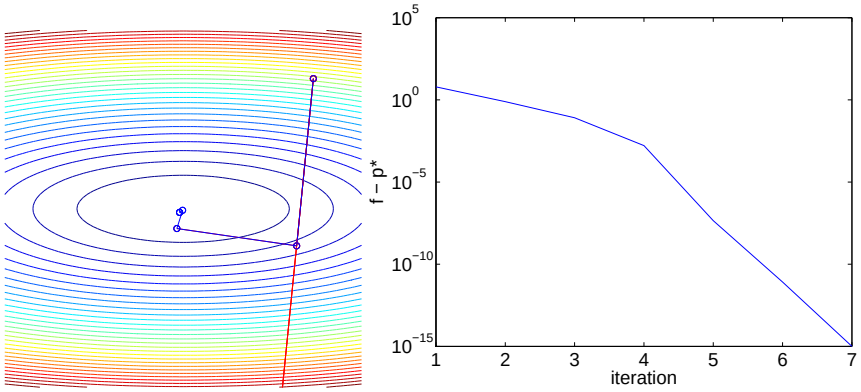
Gradient Descent



Newton's Method



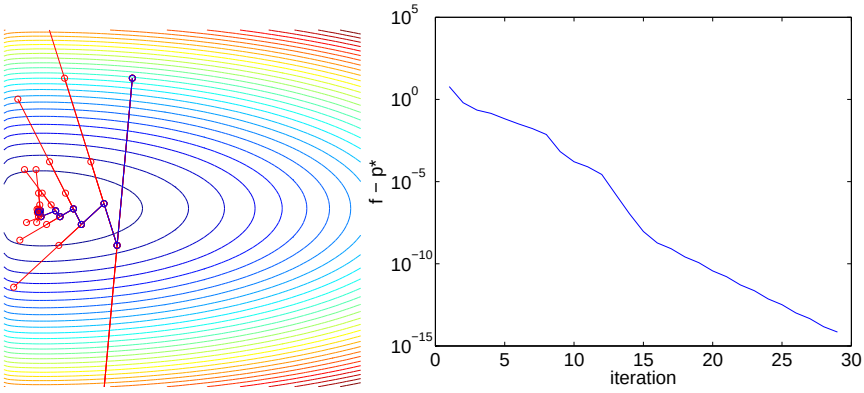
BFGS



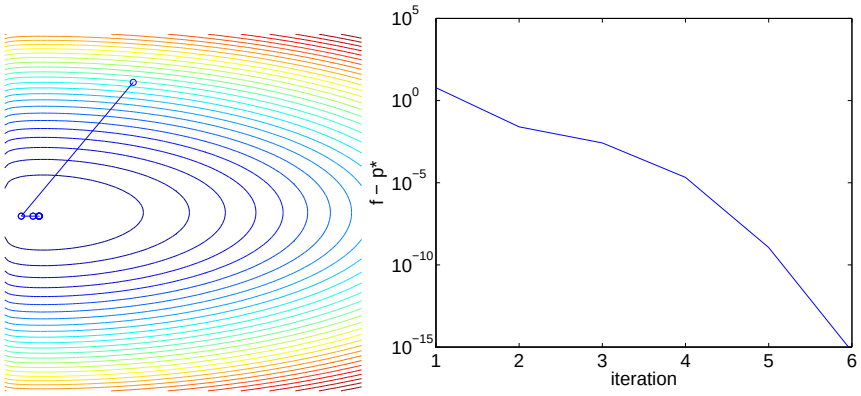
This is a two-dimensional optimization, of the function $f(\mathbf{x}) = x_1^2 + 10x_2^2 - .1 \log x_1$..

We see that because the model is non-quadratic, Newton's method no longer converges in a single iteration. BFGS is slow, but not by a great deal.

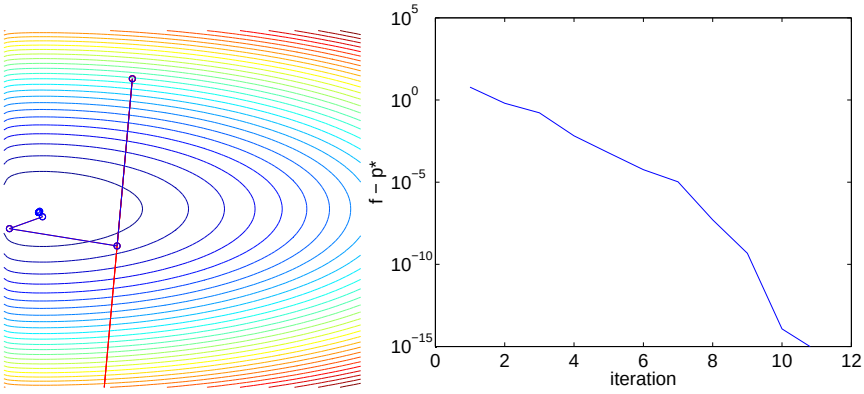
Gradient Descent



Newton's Method

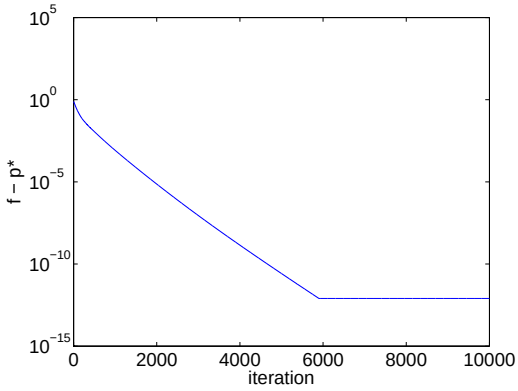


BFGS

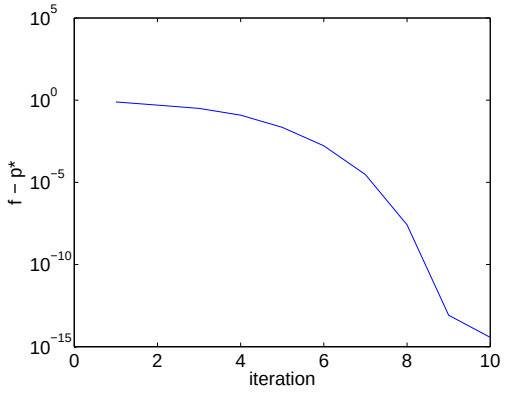


This is a 100-dimensional optimization, of a function of the form $f(\mathbf{x}) = \mathbf{c}^T\mathbf{x} - \sum \log(\mathbf{b} - A\mathbf{x})$. where A is 500x100 and $\mathbf{b}$ is of length 100. Here, Newton's method converges very quickly, BFGS takes 40 times as long, and gradient descent takes an additional factor of 25.

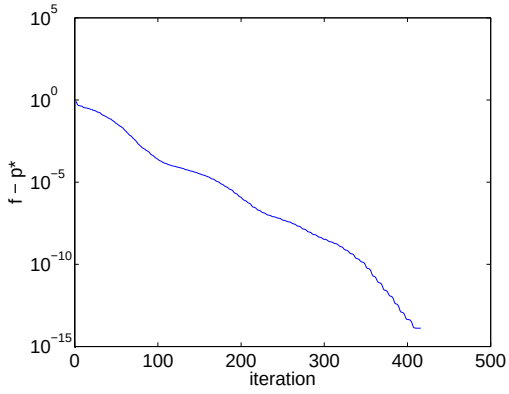
Gradient Descent



Newton's Method



BFGS



4 Constrained Optimization via Projection

Recall our definition of an optimization problem:

$$\begin{aligned} \min_{\mathbf{x}} . \quad & f(\mathbf{x}) \\ \text{s.t.} \quad & \mathbf{x} \in C. \end{aligned}$$

We can only use the algorithms above when we have no constraints, or $C = \Re^n$. What to do if C is not so trivial?

In general, constrained optimization algorithms are more complex than unconstrained ones. The most popular and powerful techniques are probably *interior-point methods*. Roughly speaking, these methods reduce the constrained problem to a *series* of unconstrained ones, in such a way that the unconstrained problems approximate the constrained ones more and more tightly. Explaining these techniques is beyond the scope of these notes.

Here, we discuss one of the simplest techniques for constrained optimization, known as **projected gradient descent**. This algorithm has many of the properties of regular gradient descent. Namely, it is simple, and works best only on well-scaled problems. Additionally, as we will see below, projected gradient descent cannot always (practically) be used. The set C needs to be simple in the sense that given an arbitrary point $\mathbf{x}^*$, there is a fast procedure to find the closest point to $\mathbf{x}^*$ in C.

The basic idea is to do gradient descent with a fixed step size but, after each iteration, project the solution into the constraint set C.

Projected Gradient Descent

Repeat:

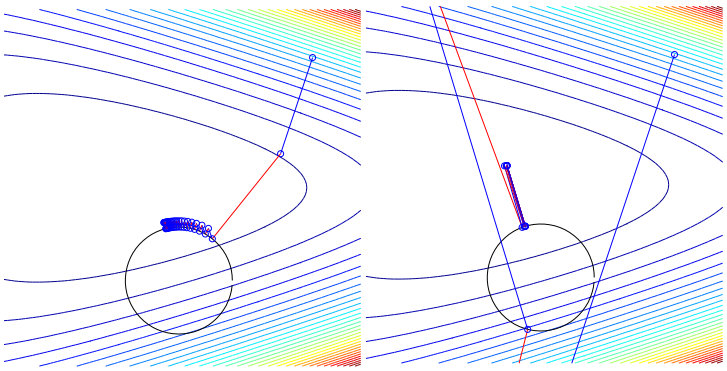
- $\mathbf{x}^* \leftarrow \mathbf{x}^k - t \nabla f(\mathbf{x}^k)$
- $\mathbf{x}^{k+1} \leftarrow \arg\min_{\mathbf{x} \in C} ||\mathbf{x} - \mathbf{x}^*||$

Instead of doing a line search, the version of projected gradient descent here uses a fixed step size t. If this is too big, the procedure will not converge. If it is too small, many iterations will be required.

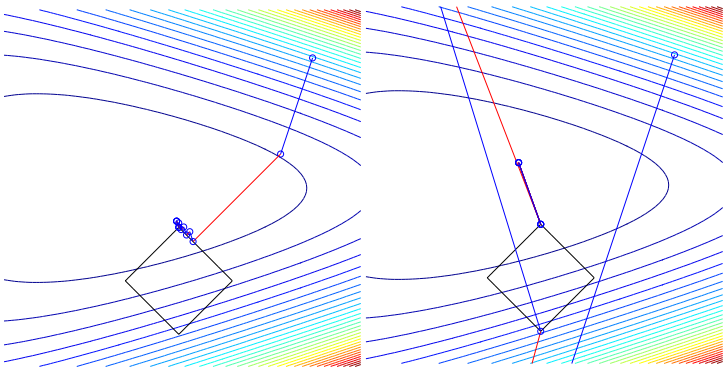
Let's consider a simple example, with $C = \{\mathbf{x} : ||\mathbf{x}|| \leq r\}$, i.e. C is the sphere of radius r centered at the origin. It is not hard to see that

$$\arg\min_{\mathbf{x} \in C} ||\mathbf{x} - \mathbf{x}^*|| = \begin{cases} \mathbf{x}^* & ||\mathbf{x}|| \leq r \\ r \frac{\mathbf{x}^*}{||\mathbf{x}^*||} & \text{otherwise} \end{cases}$$

Here are results with step sizes of .01 and .1. The blue lines show the gradient steps, while the red lines show the projection steps. The black circle contains C.



Meanwhile, here are the results of projecting onto the so-called l_1 ball. (These require a somewhat more complicated algorithm to project.) Interestingly, the exact solution is identified with 4 and 2 iterations for step sizes of .01 and .1, respectively.



The main issues with projected gradient descent are:

- As with regular gradient descent, convergence can be slow for poorly scaled problems.
- Projected gradient descent is only attractive when there is an efficient algorithm for projecting onto C. In many realistic problems, C might be very complicated, for example, $C = \{\mathbf{x} : A\mathbf{x} \leq \mathbf{b}\}$. In general, there is no efficient algorithm to project onto sets like this, and so more advanced methods must be used.

5 Stochastic Gradient Descent

If we are doing empirical risk minimization, we typically encounter objective functions of the form

$$R = \sum_{(\hat{\mathbf{x}};\hat{\mathbf{y}})\in D} L(f(\hat{\mathbf{x}}), \hat{\mathbf{y}}).$$

That is, the objective is a sum of function, one for each training element. If n is very large (e.g. $10^6 - 10^9$) it might take a huge amount of time to even evaluate this function and its gradient once. Meanwhile, for any particular datum, we can evaluate $L(f(\hat{\mathbf{x}}), \hat{\mathbf{y}})$ very quickly. Is there any way to take advantage of this special form of optimization? Can we do better than forgetting about this special structure and just using a standard optimization? One method that sometimes shows huge orders of improvement is **stochastic gradient descent**. In the context of machine learning, one can write this method like

- For $k = 1, 2, ..., \infty$:
  - Pick an element $(\hat{\mathbf{x}}, \hat{\mathbf{y}})$ of the data.
  - $\mathbf{w} \leftarrow \mathbf{w} - t_k \frac{d}{d\mathbf{w}} L(f(\hat{\mathbf{x}}), \hat{\mathbf{y}})$.

Here, t_k is a pre-determined step size. It is possible to show that for t_k of the form $a/(b+t_k)$ the algorithm will, as $k \to \infty$, converge to the optimal $\mathbf{w}$.

The advantage of stochastic gradient descent are its simplicity, and the fact that it can sometimes converge much more quickly than batch methods. To understand why, imaging taking a dataset D, and creating a new dataset D', which consists of 100 copies of all the elements in D. If we do empirical risk minimization on D', all of the above "batch" methods will slow by a factor of 100. Stochastic gradient descent, on the other hand, will operate as before. Many real-world datasets appear to display a great deal of "redundancy" like this.

The disadvantage is that the speed of convergence is often highly dependent on the values of a and b. There are no good guidelines for tuning these parameters in practice. Thus, getting good results out of stochastic gradient descent requires more manual "babysitting" than traditional optimization methods. Additionally, of course, stochastic gradient descent will only work on problems that are reasonably well-scaled.

Improving stochastic gradient descent is an active research area. Some work considers specializations of the algorithm for specific problems, while other work considers generic, problem independent modification of stochastic gradient descent, usually to try to add some second-order aspects to the optimization.