Lecture Notes:
Some notes on gradient descent

Marc Toussaint
Machine Learning & Robotics lab, FU Berlin
Arnimallee 7, 14195 Berlin, Germany

May 3, 2012

I'll briefly cover the following topics about gradient descent:

- "Steepest descent"
- How the "size" of the gradient might be misleading. This leads to methods for stepsize adaptation.
- How to guarantee monotonous convergence.
- Reconsideration of what "steepest descent" should mean in the case of a non-Euclidean metric. This leads to the so-called covariant or natural gradient.
- A brief comment on co- and contra-variance (of the gradient) under linear transformations.
- The relation of the covariant gradient to the Newton method.
- Rprop.

Simple gradient descent is a very "handy" method for optimization. That is, while gradient descent is often not the most efficient method, it is an absolutely essential tool to prototype optimization algorithms and for preliminary testing of models. Once the model formulation is stable one might want to invest more in considering better optimization methods, especially 2nd order methods, iterated line search, existing professional optimizers (e.g. SNOPT), mathematical programming (e.g. CPLEX), etc. Nonetheless, in these notes I cover some points that lead to simple yet efficient variants of gradient descent, very useful for prototyping. Points I am not covering but should be on the list is:

- stochastic gradient descent
- iterative line search
- many more...

1 Gradient descent

Given a scalar function $f(x)$ with $x \in \mathbb{R}^n$. We want to find its minimum

$$\min_x f(x) \; . \tag{1}$$

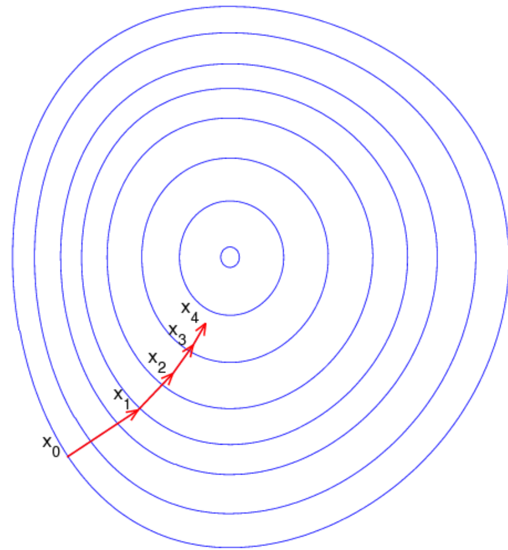

Figure 1: Illustration of steepest descent.

The gradient $\frac{\partial f(x)}{\partial x}$ at location x points toward a direction where the function increases. The negative $-\frac{\partial f(x)}{\partial x}$ is usually called *steepest descent direction*—in section 3 we will discuss in which sense this really is the steepest descent direction and in section 4 whether $\frac{\partial f(x)}{\partial x}$ really is a "vector".

The plain "gradient descent method" to find the minimum of $f(x)$ starts from an initial point x_0, then iteratively takes a step along the steepest descent direction (optionally scaled by a stepsize), until convergence. The algorithm and an illustration are given in Figure 1.

Algorithm 1 Plain gradient descent

Input: starting point $x \in \mathbb{R}^n$, a function $\frac{\partial f(x)}{\partial x}$, stepsize α, tolerance θ
Output: some x hopefully minimizing f
1: **repeat**
2: $\quad x \leftarrow x - \alpha \frac{\partial f(x)}{\partial x}^\top$
3: **until** $\Delta x < \theta$ for 10 iterations in sequence

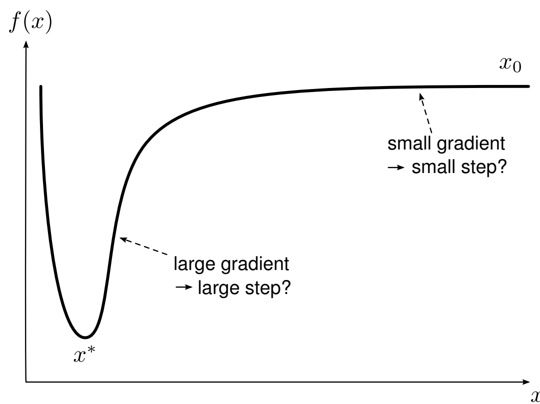

Figure 2: Should a small gradient really imply a small stepsize? And a large gradient a large stepsize? No! Instead robust online stepsize adaptation is necessary. Algorithm 2 is one version; α determines the absolute stepsize since $g/|g|$ is normalized.

2 The stepsize issue and monotonicity

The first pitfall with gradient descent is the stepsize, which in Algorithm 1 is proportional to the gradient size $|\frac{\partial f(x)}{\partial x}|$. First, be aware that the scaling of the x-axis is really arbitrary. Ideally an optimization algorithm should be invariant under rescaling of the x-axis. The same holds for the y-axis. The plain gradient $\frac{\partial f(x)}{\partial x}$ is of course not rescaling/transformation invariant (detailed in section 4). Furthermore, functions may have very different characteristics in different regions: Consider the function in Figure 2; while it is near flat (=small gradient) in one region, it is very steep (=large gradient) in other regions. Should a small (large) gradient really always imply a small (large) step?

The more one thinks about such examples, the more one might come to the following conclusion: *never trust (interpret too much into) the size of the gradient*. The direction of the gradient is ok, but the size questionable. Therefore, robust gradient methods must permanently rescale the stepsize empirically depending on local properties of the function. There are two simple heuristics:

- When you've done a step and the function value increases, your step was too large; undo the step and decrease the stepsize.
- When you've done a successful step and the function value decreases, perhaps your step could have been larger; increase the stepsize.

Algorithm 2 presents one version of such a robust stepsize adaptation.

The "undo" in the first heuristic might seem overly conservative and a waste—one is tempted to not fully undo the step but do something more heuristic (half the step, etc, etc). In my experience this ends up in overly complicated and eventually non-robust hacks. The undo step has the great advantage of guaranteeing monotonicity! Which is great! Never underestimate the relieve of a programmer (esp. in a prototyping context) of having a method that is really guaranteed to converge.

Algorithm 2 Monotonous gradient descent with stepsize adaptation

Input: starting point $x \in \mathbb{R}^n$, a scalar function $f(x)$ and its gradient $\frac{\partial f(x)}{\partial x}$, initial stepsize α, tolerance θ
Output: some x hopefully minimizing f

1: initialize $f_x = f(x) \in \mathbb{R}$, $g = \frac{\partial f(x)}{\partial x}^\top \in \mathbb{R}^n$
2: **repeat**
3: $\quad y \leftarrow x - \alpha\, g/|g|\,, \quad f_y \leftarrow f(y)$
4: $\quad$**if** $f_y < f_x$ **then**
5: $\quad\quad x \leftarrow y\,, \quad f_x \leftarrow f_y$
6: $\quad\quad g \leftarrow \frac{\partial f(x)}{\partial x}^\top$
7: $\quad\quad a \leftarrow 1.2a$ *// increase stepsize*
8: $\quad$**else**
9: $\quad\quad a \leftarrow 0.5a$ *// decrease stepsize*
10: $\quad$**end if**
11: **until** $|y - x| < \theta$ for 10 iterations in sequence

3 Steepest descent and the covariant [natural] gradient

The origin of the word "covariant" will be explained in the next section—here let's simply ask: What is really the direction of *steepest*?

Consider the following definition:

Definition 1 (Really steepest descent)**.** Given $f(x)$ and a point x_0. Define $B = \{x \in \mathbb{R}^n : d(x, x_0) = \epsilon\}$ as the set of points with *distance* ϵ to x_0. Here we presume the existence of a distance function $d(x, x_0)$. Let $x^* = \text{argmin}_{x \in B} f(x)$ be the point with smallest f-value and distance ϵ to x_0. As the steepest descent direction we define the direction from x_0 to x^*, that is, $(x^* - x_0)/\epsilon$ in the limit $\epsilon \to 0$.

I think that definition is very reasonable. It makes explicit that it is "fair" to compare the function decrease along different directions only when we walk the same *distance* along these directions. Note that the partial gradient $\frac{\partial f(x)}{\partial x}$ does not acknowledge any notion of distance in the space—how can it make a statement about steepest descent then?

Let us assume that the distance is (at least locally...) given in terms of a metric A:

$$d(x, x_0) = (x - x_0)^\top A (x - x_0) \tag{2}$$

Since, by definition of the partial gradient $g^\top = \frac{\partial f(x_0)}{\partial x}$, the function $f(x)$ can be locally approximated as

$$f(x_0 + \delta) \doteq f(x_0) + g^\top \delta \tag{3}$$

the $x^* \in B$ with minimal f-value and distance ϵ to x_0 is given as

$$x^* - x_0 = \underset{\delta}{\text{argmin}}\, g^\top \delta \quad \text{s.t.} \quad \delta^\top A \delta = \epsilon^2 \tag{4}$$

$$\Big[\text{let } A = \epsilon^2 B^\top B \text{ and } z = B\delta\Big] \tag{5}$$

$$= \underset{\delta}{\text{argmin}}\, \delta^\top g \quad \text{s.t.} \quad z^\top z = 1 \tag{6}$$

$$= B^{-1} \underset{z}{\text{argmin}} (B^{-1} z)^\top g \quad \text{s.t.} \quad z^\top z = 1 \tag{7}$$

$$= B^{-1} \underset{z}{\text{argmin}}\, z^\top B^{-\top} g \quad \text{s.t.} \quad z^\top z = 1 \tag{8}$$

$$\propto B^{-1}[-B^{-\top} g] \propto -A^{-1} g \tag{9}$$

(The B^{-1} in front of the argmin in equation (7) is because the *return value* of argmin_z has to be transformed to become a δ!!) This tells us that the "real" steepest descent direction is $-A^{-1}g$ (we neglected constants because, as discussed, one should not trust the gradient size anyway.)

In conclusion, if a metric A is given, the covariant gradient descent is given as

$$\Delta x \propto -A^{-1} g\ . \tag{10}$$

The next section will give an intuitive example. The covariant gradient is also called *natural gradient* – especially in the context when x describes a probability distribution and one uses the so-called Fisher metric in the space of distributions (literature: Amari).

4 Invariance under transformation: the gradient is a covariant thing

Consider the following example: We have a function $f(x)$ over a two dimensional space, $x \in \mathbb{R}^2$. We express the coordinates in this space as $x = (x_1, x_2)$, and let's simply assume $f(x) = x_1 + x_2$. The function's gradient is of course $\frac{\partial f}{\partial x} = (1 \ \ 1)$.

Now let's transform the coordinates of the space: we introduce new coordinates $(z_1, z_2) = (2x_1, x_2)$ or $z = Bx$ with $B = \begin{pmatrix} 2 & 0 \\ 0 & 1 \end{pmatrix}$. The same function, written in the new coordinates, is $f(z) = \frac{1}{2} z_1 + z_2$. The gradient of that same function, written in these new coordinates, is $\frac{\partial f}{\partial z} = (\frac{1}{2} \ \ 1)$.

Let's summarize what happened:

- We have a transformation of a vector space described by a coordinate transformation matrix B.
- Coordinate vectors of course transform as $z = Bx$.
- However, the partial gradient of a function w.r.t. the coordinates transforms as $\frac{\partial f}{\partial z}^\top = B^{-\top} \frac{\partial f}{\partial x}^\top$.
- Therefore, there seems to exist one type of mathematical objects (e.g. coordinate vectors) which transform with B, and a second type of mathematical objects (e.g. the partial gradient of a function w.r.t. coordinates) which transform with $B^{-\top}$.

These two types are called *contra-variant* and *co-variant*. This should at least tell us that indeed the so-called "gradient-vector" is somewhat different to a "normal vector": it behaves inversely under coordinate transformations.

Ordinary gradient descent of the form $x \leftarrow x + \alpha \frac{\partial f}{\partial x}$ adds objects of different types: a contra-variant coordinate vector x with a co-variant coordinate gradient $\frac{\partial f}{\partial x}$. Clearly, adding two such different types leads to an object who's transformation under coordinate transforms is strange—and indeed the ordinary gradient descent is not invariant under transformations.

So **why is the covariant gradient descent called covariant**? Let's check how $A^{-1}g$ transforms under a linear transformation B: As we showed, the new "vector" of partial coordinate derivatives will be $g' = B^{-\top} g$. The new metric matrix in the new coordinate system will be $A' = B^{-\top} A B^{-1}$ (a metric matrix is doubly co-variant, which we didn't show). Therefore the new covariant gradient in the new coordinates is $A'^{-1} g' = (B^{-\top} A B^{-1})^{-1} B^{-\top} g = B(A^{-1} g)$, which is the old covariant gradient in the old coordinate simply transformed forward as a normal vector. Therefore the inverse metric converts the co-variant gradient to become contra-variant. In that sense, the covariant gradient is "the only sensible thing to do" from the view of correct behavior under coordinate transforms.

In the remainder, just for the fun, I'll try to give a slightly more detailed explanation of the meaning of co- and contra-variant.

Suppose we have two (mathematical) objects and if we multiply them together this gives a scalar. The scalar shouldn't depend on any choice of coordinate system and is therefore invariant against coordinate transforms. Then, if one of the objects transforms in a co-variant ("transforming *with* the transformation") manner, the other object must transform in a contra-variant ("transforming *contrary* to the transformation") manner to ensure that the resulting scalar is invariant. This is a general principle: whenever two things multiply together to give a scalar, one should transform co- the other contra-variant.

The gradient $\frac{\partial f}{\partial x}$ of a function can be defined as an object (a so-called 1-form) that multiplies to a point variation δx to give the function variation $\delta f = \frac{\partial f}{\partial x} \delta x$. Therefore, if coordinate vectors are contra-variant, the gradient must be co-variant. Normal (contra-variant) coordinate vectors are written as columns. They multiply with row vectors to give a scalar; row vectors are co-variant.[1] It

[1] There is more confusion to add: A vector itself (as a coordinate-free algebraic object) is a co-variant object and should be distinguished from its coordinate vector (given a basis), which is a contra-variant object (it must be because coordinate coefficients multiply to basis vectors). A coordinate-free gradient itself (a 1-form) is a contra-variant object and should be distinguished from its coordinate representation (given a basis), which is a co-variant object. A metric (or 2-form) is "double-contravariant" since it multiplies to two vectors to give a scalar; its inverse is "double-covariant"; and again, moving from algebraic objects to coordinate representations this is flipped.

therefore makes sense to write a gradient as a row vector. However, note that row/column notations are just a convention anyway—and the wast majority of researchers have the convention to write gradients also as column vector.

5 Relation between covariant gradient and the Newton method

The Newton method computes the 2nd order approximation (called 2nd order Taylor expansion) of the function:

$$f(x_0 + \delta) \approx f(x_0) + \frac{\partial f(x_0)}{\partial x} \delta + \delta^\top H \delta \tag{11}$$

The matrix H is called Hessian and, intuitively, describes the local n-dimensional parabola curvature that approximates f. If H is positive-definite, the parabola is indeed positively curved, meaning that it goes *up* in all directions. (Would H have a negative eigenvalue it would describe a saddle-like function, going down along the eigen-vectors.) If H is positive-definite the parabola has a definite minimum

$$\delta^* = -H^{-1} \frac{\partial f(x_0)}{\partial x}^\top . \tag{12}$$

The Newton method iterates exactly this step: It computes a local 2nd order approximation of f, jumps to the minimum of this parabola, and iterates from there.

This is exactly the same as covariant gradient descent, but with the local Hessian H replacing the metric A. A difference is that the Hessian H of a function is *local* and typically different for any point in space, whereas the metric is usually assumed constant throughout the space.

6 Rprop

Rprop stands for "Resilient Back Propagation"; where "back propagation" refers to traditional methods of training neural networks and "resilient", in my interpretation, for "robustness by undoing steps". (Literature: Riedmiller, Igel, Hsken.) At first sight the algorithm might seem a terrible hack—but it is simple, robust and efficient. It takes the heuristic of "don't trust the gradient size" to an extreme and maintains a separate online-adapted stepsize for each input dimension. By rescaling the gradient in each dimension separately it also ignores the precise gradient direction—which could be justified by not knowing the true underlying metric: instead of assuming a known metric and using it for a covariant gradient method, it heuristically and locally rescales each dimension via an online stepsize adaptation. In fact, Rprop is fully invariant against separate linear transformations of each dimension ($\leftrightarrow$ a diagonal metric) and, empirically, also pretty robust against any linear transformation of the problem. It could therefore be interpreted as a (very heuristic) attempt to mimic covariance gradient descent.

Algorithm 3 An Rprop variance (iRprop$^-$, Igel, Hsken)

Input: functions $f(x)$ and $\frac{\partial f(x)}{\partial x}$, starting point x_0, initial stepsize α, tolerance θ
Output: some x hopefully minimizing f

1: initialize $x = x_0$, all $\alpha_i = \alpha$, all $g'_i = 0$
2: **repeat**
3: $\quad g \leftarrow \frac{\partial f(x)}{\partial x}^\top$
4: $\quad x' \leftarrow x$
5: $\quad$ **for** $i = 1 : n$ **do**
6: $\quad\quad$ **if** $g_i g'_i > 0$ **then** *// same direction as last time*
7: $\quad\quad\quad \alpha_i \leftarrow 1.2\alpha_i$
8: $\quad\quad$ **else if** $g_i g'_i < 0$ **then** *// change of direction*
9: $\quad\quad\quad \alpha_i \leftarrow 0.5\alpha_i$
10: $\quad\quad\quad g_i \leftarrow 0$ *// force $g_i g'_i = 0$ in next iteration*
11: $\quad\quad$ **end if**
12: $\quad\quad$ optionally: cap $\alpha_i \in [\alpha_{\text{min}}, \alpha_{\text{max}}]$
13: $\quad\quad x_i \leftarrow x_i - \alpha_i \ \text{sign}(g_i)$
14: $\quad$ **end for**
15: $\quad g' \leftarrow g$ *// store the gradient for next iteration*
16: **until** $|x' - x| < \theta$ for 10 iterations in sequence