

# Optimization Techniques in Finance

## 7. Large scale optimization methods and machine learning. II

Andrew Lesniewski

Baruch College  
New York

Fall 2019

# Outline

- 1 Stochastic approximation
- 2 Stochastic gradient descent and its refinements

# Robbins-Monro algorithm

- The problem of finding the minimum  $x^*$  of a function  $f(x)$ , where  $x \in \mathbb{R}$ , leads to the equation  $f'(x) = 0$ .
- More generally, consider the problem of finding the solution  $x^*$  to the equation  $g(x) = a$ , where  $x \in \mathbb{R}$ .
- This can be done by means of Newton's method:

$$x_{k+1} = x_k - \frac{g(x_k) - a}{g'(x_k)},$$

which (under suitable assumptions) converges to the desired root,  $x_k \rightarrow x^*$ .

- If the derivative  $g'(x)$  is strictly positive near  $x^*$ , then we can instead use the following (less efficient) recursive rule:

$$x_{k+1} = x_k - \alpha_k (g(x_k) - a),$$

with a suitable sequence  $\alpha_k > 0$  (usually chosen to be constant).

# Robbins-Monro algorithm

- In the original optimization problem,  $g(x) = f'(x)$ , this corresponds to the gradient descent method.
- *Stochastic approximation algorithms*, introduced first in the landmark paper [4] by Robbins and Monro, are recursive update rules that extend this idea to solve problems where the observations of  $g(x)$  are noisy.
- One can think of these algorithms as a way of solving an optimization problem with noisy observations of the gradient of the objective function.
- Mathematically, the goal is to understand the properties of a function

$$g(x) = E(G(x)),$$

which is the expected value of a function, without evaluating  $g(x)$  exactly.

- The function  $g(x)$  may not be explicitly known, and one may think of  $G(x)$  as noisy realizations of  $g(x)$ .

# Robbins-Monro algorithm

- We could proceed as follows.
- Consider  $m$  samples  $G_1(x), \dots, G_m(x)$  of  $G(x)$ , and approximate

$$g(x) \approx \frac{1}{m} \sum_{i=1}^m G_i(x).$$

- This will certainly work when  $m \rightarrow \infty$ . The problem is that it may be computationally too expensive.
- The idea behind the *Robbins-Monro algorithm* is to use a single realization  $G(x)$  to approximate  $x$  at each recursion step,

$$g(x_k) \approx G(x_k).$$

# Robbins-Monro algorithm

- Then, in order to solve

$$g(x) = a \tag{1}$$

we could use the following rule:

$$x_{k+1} = x_k - \alpha_k (G(x_k) - a).$$

- This idea actually works and it underlies the *Robbins-Monro algorithm*.
- In the case the function  $g(x)$  is monotone non-decreasing, the problem (1) can be solved in the following way.
- We initialize  $x$  with some value  $x_0$ , and iterate:

$$x_{k+1} = x_k - \alpha_k (G(x_k) - a), \text{ for } k = 0, 1, \dots \tag{2}$$

Some conditions on  $\alpha_k$  are required.

# Robbins-Monro algorithm

- *Theorem.* Assume that equation (1) has a unique solution  $x^*$ . The algorithm (2) converges, provided that the following conditions are satisfied:
  - the random variables  $G(x)$  are uniformly bounded,  $|G(x)| \leq \text{const}$ ,
  - the function  $g(x)$  is monotone non-decreasing,
  - $g'(x^*) > 0$ , and
  - the sequence  $\alpha_k$  satisfies:
 
$$\sum_{k=1}^{\infty} \alpha_k = \infty, \tag{3}$$

$$\sum_{k=1}^{\infty} \alpha_k^2 < \infty.$$

- For example, one could choose  $\alpha_k = \alpha/k$ . In fact, one achieves the fastest rate of convergence of the algorithm with the choice  $\alpha_k = g'(x^*)/k$ .
- The Robbins-Monro algorithm, even if on its own of little practical use, is the mother of all stochastic optimization methods!

# Empirical loss function

- In large scale machine learning optimization problems the objective function (frequently termed *loss function*) has the form of a sum:

$$f(x) = \frac{1}{m} \sum_{i=1}^m f_i(x), \quad (4)$$

where the parameter  $x \in \mathbb{R}^n$  which minimizes  $f(x)$  is to be estimated.

- Each summand function  $f_i$  is typically associated with the  $i$ -th observation in the dataset (used for training).
- The size of the data  $m$  may be very large, which could make the optimization intractable, or at least very expensive.
- One can think of (4) as an empirical observation of an unknown loss function, which is the expected value of individual loss functions over an unknown probability space
- It is called *empirical loss function*.



# Stochastic gradient descent

- Broadly speaking, optimization methods in machine learning fall into two categories:
  - (i) batch methods, and
  - (ii) stochastic gradient descent methods (SGD).
- In the batch version of the gradient descent method, the  $x$  updates use the full dataset:

$$x_{k+1} = x_k - \frac{\alpha_k}{m} \sum_{i=1}^m \nabla f_i(x). \quad (5)$$

- SGD is an iterative method, which can be viewed as an application of the idea underlying the Robbins-Monro algorithm. Unlike the batch gradient descent method, the data samples are selected randomly from the full dataset instead of using the entire dataset.

# Stochastic gradient descent

- We have already discussed examples of such sum-minimization problems in the context of
  - (i) (nonlinear) least squares problems,
  - (ii) MLE estimation (for independent observations).
- Except in convex cases, finding the global minimum of  $f(x)$  may be a tall order.
- For this reason, many data scientists argue that in some MLE problems requiring a global (or even local) minimum of  $f(x)$  may be too restrictive.
- Therefore, one often considers stationary points of the likelihood function as the desired solutions to the data fitting problems.

# Basic iterative method

- The batch gradient descent method consists in generating the following iterates:

$$\begin{aligned}x_{k+1} &= x_k - \alpha_k \nabla f(x) \\ &= x_k - \frac{\alpha_k}{m} \sum_{i=1}^m \nabla f_i(x),\end{aligned}$$

where  $\alpha_k$  is the step size (or learning rate).

- In cases of moderate data sizes and / or problem dimensionality the batch approach may work.
- However, in cases of large scale machine learning problems with huge training datasets, evaluating the sums of gradients in the equation above becomes prohibitively expensive.
- In order to reduce the computational cost at each iteration, SGD samples only a small subset of summand functions at every step.

# Basic iterative method

- In the SGD (“online”) method, the full gradient of  $f(x_k)$  is approximated by a gradient calculated at a single, randomly selected example:

$$x_{k+1} = x_k - \alpha \nabla f_i(x_k). \quad (6)$$

- The algorithm can be summarized as follows:
  - (i) Select an initial set of parameters  $x_0$  and learning rate  $\alpha$ .
  - (ii) While stopping criteria not met, iterate the following step. Select  $i \in \{1, \dots, m\}$  at random and set

$$x_{k+1} = x_k - \alpha \nabla f_i(x_k).$$

- Note that we do not include the line search step discussed before: it is almost never used.
- A way of smoothing out and speeding up the convergence might be the mini batch approach: instead of a single example  $i$ , one selects a (small) set of randomly selected examples.

# Refinements of the basic SGD

- As we have already discussed, gradient descent methods tend to be slow and often do not converge.
- In order to remedy this situation, many refinements of the basic SGD algorithm have been proposed and used.
- In particular, the key role of the learning rate has been recognized: choosing  $\alpha$  too large can cause the algorithm to diverge, while choosing it too small slows down the convergence.
- We will discuss several variants of the basic SGD popular in machine learning.

# Momentum

- In the *momentum method* [5], one blends the current gradient with the most recent increment:

$$x_{k+1} = x_k - \alpha \nabla f(x_k) + \mu(x_k - x_{k-1}).$$

- The rate  $\mu$  at which the previous increment is used is called the *decay rate*.
- This can be algorithmically formulated as follows:
  - (i) select initial  $x_0$  and  $m_0 = 0$ ,
  - (ii) while stopping criteria are not met, iterate the following steps:

$$\begin{aligned}g_{k+1} &= \nabla f(x_k), \\m_{k+1} &= \mu m_k + g_{k+1}, \\x_{k+1} &= x_k - \alpha m_{k+1}.\end{aligned}$$

- Adding the momentum accelerates the descent along directions where the gradient is relatively stable, and it tends to slow down the descent along directions with oscillating gradient.

# Nesterov's accelerated gradient descent

- *Nesterov's accelerated gradient descent* is an improvement on the momentum method.
- Rather than compute the gradient at the current solution, we compute it at the location we would end up if we continued moving in the same direction.
- We write the momentum iteration in the form:

$$\begin{aligned}x_{k+1} &= x_k - \alpha m_{k+1} \\ &= x_k - \alpha \mu m_k - \alpha g_{k+1}.\end{aligned}$$

- Note that  $g_{k+1}$  is *not* the gradient of  $f(x)$  at  $x = x_k - \alpha \mu m_k$ , while the equation above suggests that it should be evaluated there.

# Nesterov's accelerated gradient descent

- The accelerated method is formulated as follows:
  - (i) select initial  $x_0$  and  $m_0 = 0$ ,
  - (ii) while exit criteria are not met, iterate the following steps:

$$g_{k+1} = \nabla f(x_k - \alpha \mu m_k),$$

$$m_{k+1} = \mu m_k + g_{k+1},$$

$$x_{k+1} = x_k - \alpha m_{k+1}.$$

- One can prove that Nesterov's method is indeed faster than the gradient descent method.



# AdaGrad

- At the core of the convergence issues of the gradient descent method for “difficult” objective functions is the fact that at each iteration one takes a step of the same size, regardless of the magnitude of the gradient.
- The *adaptive gradient algorithm* (AdaGrad) [2] is a modified stochastic gradient descent which attempts to increase the learning rate in the direction of slowly changing parameters, and decrease the learning rate in the direction of faster changing ones.
- This is achieved by keeping track of the values of gradients of the objective function at the previous iteration.

# AdaGrad

- The method is formulated as follows:
  - (i) select initial  $x_0$ ,  $G_0 = 0$ , and  $D_0 = 0$ ,
  - (ii) while exit criteria are not met, iterate the following steps:

$$g_{k+1} = \nabla f(x_k),$$

$$G_{k+1} = G_k + g_{k+1}g_{k+1}^T,$$

$$D_{k+1} = \text{diag}(G_{k+1}),$$

$$x_{k+1} = x_k - \alpha \frac{1}{\sqrt{D_{k+1}} + \epsilon} g_{k+1},$$

where  $\epsilon$  is a small parameter introduced to prevent dividing by zero.

- All squares and square root taking above is done elementwise.
- Notice that each component of  $D_{k+1}$  is the Euclidean norm of the previous gradient. The learning rate is thus adapted for each of the dimensions.

# RMSProp

- One of the issues with AdaGrad is that  $D_k$  keeps growing in the long term.
- It causes the descent to slow down in all dimensions.
- The idea underlying the *root mean square propagation* (RMSProp) [6] method is to use a weighted running average of the squared gradients instead of  $D_k$ .
- The decay rate at which the running average is blended with the current value of  $D_k$  is denoted by  $\gamma$ .

# RMSProp

- The RMSProp method is formulated as follows:
  - (i) select initial  $x_0$ ,  $G_0 = 0$ , and  $D_0 = 0$ ,
  - (ii) while exit criteria are not met, iterate the following steps:

$$g_{k+1} = \nabla f(x_k),$$

$$G_{k+1} = (1 - \gamma)G_k + \gamma g_{k+1} g_{k+1}^T,$$

$$D_{k+1} = \text{diag}(G_{k+1}),$$

$$x_{k+1} = x_k - \alpha \frac{1}{\sqrt{D_{k+1} + \epsilon}} g_{k+1}.$$

# Adam

- *Adaptive moment estimation* (Adam) [3] is a refinement of the RMSProp optimizer. It combines it with the momentum method.
- The key ideas of the method are:
  - (i) Use the momentum method, but with a decaying mean instead of a decaying sum.
  - (ii) Combine it with RMSProp.
  - (iii) Update the gradient proportionally to the normalized average gradient.
  - (iv) Add initialization bias correction to offset potential instabilities.

## Adam

- The method is formulated as follows:
  - select initial  $x_0$ ,  $m_0 = 0$ ,  $\hat{m}_0 = 0$ ,  $G_0 = 0$ ,  $D_0 = 0$ , and  $\hat{D}_0 = 0$ ,
  - while exit criteria are not met, iterate the following steps:

$$g_{k+1} = \nabla f(x_k),$$

$$m_{k+1} = \mu m_k + (1 - \mu)g_{k+1}$$

$$\hat{m}_{k+1} = \frac{m_{k+1}}{1 - \mu^{k+1}}$$

$$G_{k+1} = (1 - \gamma)G_k + \gamma g_{k+1} g_{k+1}^T,$$

$$D_{k+1} = \text{diag}(G_{k+1}),$$

$$\hat{D}_{k+1} = \frac{D_{k+1}}{1 - \gamma^{k+1}},$$

$$x_{k+1} = x_k - \alpha \frac{1}{\sqrt{\hat{D}_{k+1} + \epsilon}} \hat{m}_{k+1}.$$

# Adam

- The rationale behind choosing the bias corrections to  $m$  and  $D$  above is as follows.
- From the definition of  $m_k$ ,

$$\begin{aligned}m_{k+1} &= \mu m_k + (1 - \mu)g_{k+1} \\ &= (1 - \mu) \sum_{i=1}^{k+1} \mu^{k+1-i} g_{k+1}.\end{aligned}$$

- Taking the expected value yields

$$E(m_{k+1}) = (1 - \mu^{k+1})E(g_{k+1}).$$

- Hence  $m_k$  is a biased estimator of  $g_k$ . Dividing by  $(1 - \mu^{k+1})$  we obtain an unbiased estimator  $\hat{m}_k$ .
- An identical argument applies to  $D_k$ .

# Natural gradient descent

- *Natural gradient descent* [1] is a yet different approach based on the “intrinsic” geometry of the surface in  $\mathbb{R}^{n+1}$  defined by

$$y = f(x). \quad (7)$$

- This approach is particularly effective in MLE problems, where the intrinsic geometry is given by the Fisher information matrix.
- Here we present a toy version of this algorithm, which contains the main ideas.
- The distance  $ds$  between two infinitesimally close points in  $\mathbb{R}^{n+1}$  is given by

$$ds^2 = dy^2 + \sum_{i=1}^n dx_i^2.$$



# Natural gradient descent

- Notice that

$$\begin{aligned}df(x) &= \sum_{i=1}^n \frac{\partial f}{\partial x_i} dx_i \\ &= \nabla f(x)^\top dx.\end{aligned}\tag{8}$$

- Hence, the distance between two infinitesimally close points of the surface (7) is given by

$$ds^2 = \sum_{i=1}^n \left(1 + \left(\frac{\partial f}{\partial x_i}\right)^2\right) dx_i^2 + \sum_{i,j=1, i \neq j}^n \frac{\partial f}{\partial x_i} \frac{\partial f}{\partial x_j} dx_i dx_j.$$

# Natural gradient descent

- This can be compactly written as

$$ds^2 = dx^\top g(x) dx, \quad (9)$$

where the symmetric matrix  $g(x)$  is given by

$$g(x) = I + \nabla f(x) \nabla f(x)^\top, \quad (10)$$

and where  $I$ , as usual, is the ( $n$ -dimensional) identity matrix.

- This expression defines the intrinsic distance between two infinitesimally close points on the surface (7).

# Natural gradient descent

- Now, suppose we wish to express the differential  $df(x)$  in terms of the distance (9) with a “natural” gradient  $\nabla_g$  replacing  $\nabla$ ,

$$df(x) = \nabla_g f(x)^\top g(x) dx. \quad (11)$$

- Comparing with (8) yields the formula

$$\begin{aligned} \nabla_g f(x) &= g(x)^{-1} \nabla f(x) \\ &= (I + \nabla f(x) \nabla f(x)^\top)^{-1} \nabla f(x). \end{aligned}$$

# Natural gradient descent

- A quick algebra shows that this can be written in a simpler form,

$$\nabla_g f(x) = (1 + \|\nabla f(x)\|^2)^{-1} \nabla f(x), \quad (12)$$

where  $\|\nabla f(x)\|^2 = \nabla f(x)^\top \nabla f(x)$  is the squared magnitude of the vector  $\nabla f(x)$ .

- Notice that:
  - (i) the natural gradient penalizes directions for which  $\nabla f(x)$  is large,
  - (ii) it is essentially equal to  $\nabla f(x)$  for directions in which  $\nabla f(x)$  is small.
- The algorithm is formulated as follows:
  - (i) select initial  $x_0$ ,
  - (ii) while stopping criteria are not met, iterate the following steps:







$$g_{k+1} = \nabla f(x_k),$$

$$x_{k+1} = x_k - \alpha (1 + \|g_{k+1}\|^2)^{-1} g_{k+1}.$$

# Performance comparison

- For comparison, we applied some of the methods described above to the Rosenbrock function introduced in Lecture Notes #1. The initial point is  $x_0 = (0.5, 0.3)$  and the exit criterion is  $\|f\|_\infty < 10^{-3}$ .
  - (i) Gradient descent diverges (for this and any other choice of initial guess, except for  $x_0 = (0, 0)$ ).
  - (ii) Natural gradient descent stops after 6,430 iterations with  $x^* = (0.9991, 0.9983)$ .
  - (iii) AdaGrad stops after 25,307 iterations with  $x^* = (0.9988, 0.9975)$ .
  - (iv) RMSProp does not converge within  $max\_iters = 1,000,000$  iterations, and exits with  $(0.9902, 0.9656)$ .
  - (v) Adam stops after 859 iterations with  $x^* = (1.0002, 0.9996)$ .
- Adam wins, which explains why it is the method of choice in modern machine learning!

# References

-  [1] Amari, S.: Natural Gradient Works Efficiently in Learning, *Neural Computation*, **10**, 251 - 276 (1998).
-  [2] Duchi, J. Hazan, E., and Singer, Y.: Adaptive subgradient methods for online learning and stochastic optimization, *JMLR*, **12**, 2121 - 2159 (2011).
-  [3] Kingma, D. P., and Ba, J. L.: Adam: A method for stochastic optimization, arXiv preprint arXiv:1412.6980 (2014).
-  [4] Robbins, H., and Monro, S.: A Stochastic Approximation Method, *The Annals of Mathematical Statistics*, **22**, 400 - 407 (1951).
-  [5] Rumelhart, D. E., Hinton, G. E., and Williams, R. J.: Learning representations by back-propagating errors, *Nature*, **323**, 533 - 536 (1986).
-  [6] Tieleman, T., and Hinton, G. E.: Lecture 6.5-rmsprop: Divide the gradient by a running average of its recent magnitude, COURSERA: Neural Networks for Machine Learning (2012).