Optimization

7. Stochastic Gradient Descent

Andrew Lesniewski

Baruch College
New York

Fall 2018

Outline

1. Stochastic approximation
2. Stochastic Gradient Descent
3. Refinements of the basic GSD

Robbins-Monro algorithm

- Suppose that we are tasked with finding the solution x^* to the equation $h(x) = a$, where $x \in \mathbb{R}$. As we have seen earlier, such questions arise in optimization problems.
- This can be done by means of Newton's method:

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

  which (under suitable assumptions) converges to the desired root, $x_k \to x^*$.
- If the derivative $h'(x)$ is strictly positive near x^*, then we can instead use the following (less efficient) recursive rule:

$$x_{k+1} = x_k - \alpha\big(h(x_k) - a\big),$$

  with a constant α, chosen sufficiently small. In optimization problems, this corresponds to the gradient descent method.
- *Stochastic approximation algorithms* are recursive update rules that extend this idea to solve problems where the observations of $h(x)$ are noisy.

Robbins-Monro algorithm

- Mathematically, the goal of these algorithms is to understand the properties of a function
$$h(x) = \mathrm{E}[H(x)],$$
which is the expected value of a function, without evaluating $h(x)$ exactly. The function $h(x)$ may not be explicitly known, and one may think of $H(x)$ as noisy realizations of $h(x)$.
- We could proceed as follows. Consider m samples $x_{1,k}, \ldots, x_{m,k}$ of x, and approximate
$$h(x_k) \approx \frac{1}{m} \sum_{i=1}^{m} H(x_{i,k}).$$
This will certainly work when $m \to \infty$. The problem is that it may be too computationally expensive.
- The idea behind the *Robbins-Monro algorithm* is to use a single realization x_k to approximate x at each recursion step,
$$h(x_k) \approx H(x_k).$$

Robbins-Monro algorithm

- Then, in order to solve
$$h(x) = a \tag{1}$$
we could use the following rule:
$$x_{k+1} = x_k - \alpha\big(H(x_k) - a\big).$$
- This idea actually works and it underlies the *Robbins-Monro algorithm*. In the case the function $h(x)$ is monotone non-decreasing, the problem (1) can be solved in the following way.
- We initialize x at some value x_0, and iterate:
$$x_{k+1} = x_k - \alpha_k\big(H(x_k - a), \text{ for } k = 0, 1, \ldots, \tag{2}$$
where $\alpha_k > 0$ is the learning rate. Some conditions on α_k are required.

Robbins-Monro algorithm

- *Theorem.* The algorithm (2) converges, provided that the following conditions are satisfied:
  - (i) the random variables $H(x)$ are uniformly bounded, $|H(x)| \leq const$,
  - (ii) the function $h(x)$ is monotone non-decreasing,
  - (iii) $h'(x^*) > 0$, and
  - (iv) the sequence α_k satisfies:

$$\sum_{k=1}^{\infty} \alpha_k = \infty, \qquad \sum_{k=1}^{\infty} \alpha_k^2 < \infty. \tag{3}$$

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

Stochastic gradient descent

- *Stochastic gradient descent* (SGD), is an iterative method for minimizing a differentiable objective function, which can be viewed as an application of the idea underlying the Robbins-Monro algorithm. Unlike the standard (or "batch") gradient descent method, the data samples are selected randomly from a large set instead of the entire data set.
- In large scale optimization problems one faces the issue of minimizing an objective function that has the form of a sum:

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

  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 data set (used for training).
- The size of the data m may be very large, which could make the optimization intractable, or at least very expensive.

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 standard gradient descent method consists in generating the following iterates:

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

  where α is the learning rate (step size).
- 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 data sets, evaluating the sums of gradients in the equation above becomes prohibitively expensive.
- In order to reduce the computational cost at each iteration, GSD 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:

$$x_{k+1} = x_k - \alpha \nabla f_i(x_k). \tag{5}$$

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

- 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 GSD algorithm have been proposed and used.
- In particular, in machine learning, the key role of the learning rate has been recognized. Choosing α 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 GSD popular in machine learning.

Natural gradient descent

- 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.
- This can be corrected by taking into account the "intrinsic" geometry of the surface in $\mathbb{R}^{n+1}$ defined by

$$y = f(x). \tag{6}$$

  Here is one way this can be done.
- 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

$$\begin{aligned} df(x) &= \sum_{i=1}^{n} \frac{\partial f}{\partial x_i}\, dx_i \\ &= \nabla f(x)^{\mathrm{T}} dx. \end{aligned} \tag{7}$$

- Hence, the distance between two infinitesimally close points of the surface (6) 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

$$ds^2 = dx^{\mathrm{T}} g(x) dx, \tag{8}$$

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

$$g(x) = I + \nabla f(x) \nabla f(x)^{\mathrm{T}}, \tag{9}$$

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 (6).

Natural gradient descent

- Now, suppose we wish to express the differential $df(x)$ in terms of the distance (8) with a "natural" gradient ∇_g replacing ∇,

$$df(x) = \nabla_g f(x)^{\mathrm{T}} g(x) dx. \tag{10}$$

- Comparing with (7) yields the formula

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

Natural gradient descent

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

$$\nabla_g f(x) = \left(1 + \|\nabla f(x)\|^2\right)^{-1} \nabla f(x), \tag{11}$$

  where $\|\nabla f(x)\|^2 = \nabla f(x)^{\mathrm{T}} \nabla f(x)$ 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.

Momentum

- In the *momentum method* [3], one uses the previous value of the gradient blended with the new value.
- The rate at which the previous value of $\nabla f(x)$ is used is called the *decay rate* and is denoted by μ.
- The algorithm is formulated as follows:
  (i) select initial x_0 and $m_0 = 0$,
  (ii) while exit criteria are not met, iterate the following steps:

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

- Adding the momentum accelerates the descent along directions where the gradient is relatively stable, it tends to slows 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:

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

- Note that g_{k+1} *is not* the gradient of $f(x)$ at $x = x_k - \alpha\mu m_k$.

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:

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

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

AdaGrad

- The *adaptive gradient algorithm* (AdaGrad) [1] is a another 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:

$$
\begin{aligned}
g_{k+1} &= \nabla f(x_k), \\
G_{k+1} &= G_k + g_{k+1} g_{k+1}^{\mathrm{T}}, \\
D_{k+1} &= \operatorname{diag}(G_{k+1}), \\
x_{k+1} &= x_k - \alpha \frac{1}{\sqrt{D_{k+1}} + \epsilon} G_{k+1},
\end{aligned}
$$

  where ϵ is a small parameter introduced to prevent diving 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) [4] 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 γ.

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:

$$
\begin{aligned}
g_{k+1} &= \nabla f(x_k), \\
G_{k+1} &= (1-\gamma)G_k + \gamma g_{k+1} g_{k+1}^{\mathrm{T}}, \\
D_{k+1} &= \operatorname{diag}(G_{k+1}), \\
x_{k+1} &= x_k - \alpha \frac{1}{\sqrt{D_{k+1}} + \epsilon} G_{k+1}.
\end{aligned}
$$

Adam

- *Adaptive moment estimation* (Adam) [2] 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:
  - (i) select initial x_0, $m_0 = 0$, $\widehat{m}_0 = 0$, $G_0 = 0$, $D_0 = 0$, and $\widehat{D}_0 = 0$,
  - (ii) while exit criteria are not met, iterate the following steps:

$$
\begin{aligned}
g_{k+1} &= \nabla f(x_k), \\
m_{k+1} &= \mu m_k + (1-\mu) g_{k+1} \\
\widehat{m}_{k+1} &= \frac{m_{k+1}}{1-\mu^{k+1}} \\
G_{k+1} &= (1-\gamma) G_k + \gamma g_{k+1} g_{k+1}^{\mathrm{T}}, \\
D_{k+1} &= \operatorname{diag}(G_{k+1}), \\
\widehat{D}_{k+1} &= \frac{D_{k+1}}{1-\gamma^{k+1}}, \\
x_{k+1} &= x_k - \alpha \frac{1}{\sqrt{D_{k+1}} + \epsilon} G_{k+1}.
\end{aligned}
$$

Adam

- The rationale behind choosing the bias corrections fo 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

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

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

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


Duchi, J. Hazan, E., and Singer, Y.: Adaptive subgradient methods for online learning and stochastic optimization, *JMLR*, **12**, 2121-2159 (2011).


Kingma, D. P., and Ba, J. L.: Adam: A method for stochastic optimization, arXiv preprint arXiv:1412.6980 (2014).


Rumelhart, D. E., Hinton, G. E., and Williams, R. J.: Learning representations by back-propagating errors, *Nature*, **323**, 533–536 (1986).


Tieleman, T., and Hinton, G. Lecture 6.5-rmsprop: Divide the gradient by a running average of its recent magnitude. COURSERA: Neural Networks for Machine Learning (2012).