Monte Carlo Optimization

Junyi Zhou

April 8, 2019

Contents

1 Introduction **2**

2 Stohastic Search **3**
- 2.1 Numerical Solutions . . . 3
  - 2.1.1 Gradient Descent Based Methods . . . 3
  - Stochastic Gradient Descent Based Methods . . . 4
  - Newton-Raphson Method . . . 5
  - Momentum Based Algorithms . . . 6
  - Adaptive Learning Rate Based Algorithms . . . 7
- 2.2 Stochastic Solutions . . . 8
  - 2.2.1 A Basic Solution . . . 8
  - 2.2.2 Stochastic Gradient . . . 9
  - 2.2.3 Simulated Annealing . . . 10

3 Stochastic Approximation **12**
- 3.1 Monte Carlo EM (MCEM) . . . 13

1 Introduction

In this article, we propose two separate uses of Monte-Carlo simulation method to solve optimization problems. The first use, described in Section 2, is to produce stochastic search techniques to reach the maximum (or minimum) of a function, introducing random exploration techniques on the surface of target function that avoid being trapped in local maxima (or minima) and have certain chance of reaching global optimum. The second use, as seen in Section 3, simulation is used to approximate the function to be optimized.

The entire paper addresses the issue of finding either maxima or minima using different stochastic approaches. For simplicity, we only focus on the maximization problem

$$\max_{\theta \in \Theta} h(\theta) \tag{1}$$

since a minimization problem could be handled as a maximization problem by substituting $-h(\theta)$ or $1/h(\theta)$ for $h(\theta)$. Besides, maximize a specific function, such as a log-likelihood function, is a common request in most statistical based problems. Note that, the techniques we introduce in this paper almost always focusing on the problems with continuous domains, Θ. This means that optimization problems over a finite set such as *traveling salesman problem* (see, e.g. Spall, 2003, Robert and Casella, 2004) are not considered here.

To better compare the results among different approaches, we adopt a global example of maximizing the likelihood function associated with the following mixture model

$$\pi \mathcal{N}(\mu_1, \sigma_1^2) + (1-\pi)\mathcal{N}(\mu_2, \sigma_2^2), \tag{2}$$

with explicit log-likelihood function

$$logL(\theta|x) \propto \sum_i log\{\pi \exp(-\frac{(x_i-\mu_1)^2}{2\sigma_1^2}) + (1-\pi)\exp(-\frac{(x_i-\mu_2)^2}{2\sigma_2^2})\}. \tag{3}$$

Assume that we already know the true value of π, which is 0.25, σ_1, which is 0.5, and σ_2, which is 1. We generate a simulated sample of 800 observations from this mixture with $\mu_1 = 0$ and $\mu_2 = 2.5$, then our purpose is to find out the MLE of μ_1 and μ_2 according to different methods of achieving the global maximum point on the likelihood surface $h(\theta|X)$. The corresponding density, likelihood of this mixture model are shown in Figure 1.

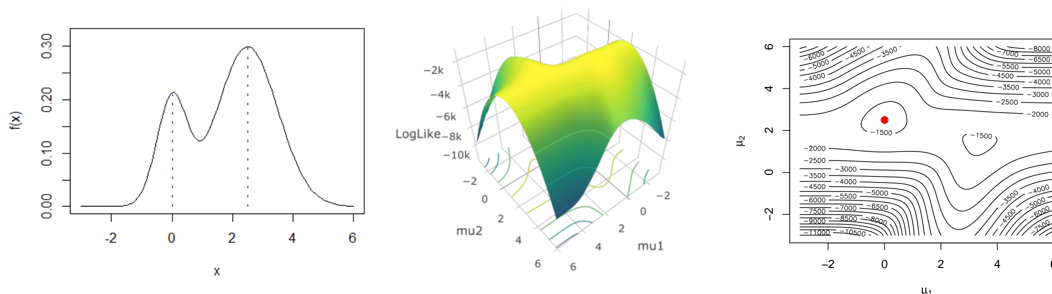

Figure 1: (left) Mixture Model Density Plot; (middle) log-Likelihood Surface Plot; (right) log-Likelihood Contour Plot (red dot indicating the global maxima point)

2 Stohastic Search

The optimization problem (1) can be processed by either numerical or stochastic solutions. In the numerical perspective, e.g. Newton-Raphson method, performance is highly dependent on the analytical properties of the target function $h(\theta)$ (such as convexity, boundedness, and smooothness). However, those properties of $h(\theta)$ play a lessor role in stochastic-based approaches (we will see that in the remainder of this section). Therefore, if $h(\theta)$ is too complex to allow an analytic study or if the domain Θ is too irregular, stochastic solution is a more appropriate method of choice.

2.1 Numerical Solutions

Among numerical solutions, the original optimization problem is usually transformed into finding the root of first derivative of $h(\theta)$, i.e.

$$\{\theta : \nabla h(\theta|X) = 0\}. \tag{4}$$

As aforementioned, if $h(\theta|X)$ is of proper attributions, then the root finding could be very efficient. While if there are multiple roots with negative or positive second derivatives (means maxima or minima) of this equation, then numerical solutions will lead to different optimums, either global or local, based on initial values. Hence, for numerical approaches, change different starting points is a commonly used technique to avoid being trapped by local optimum or saddle points.

2.1.1 Gradient Descent Based Methods

Gradient descent is one of the most popular algorithms to perform optimization, and by far, is also the most commonly used underlying algorithm to train a neural network. This section aims at providing intuitions towards the behavior of different algorithms sharing the idea of gradient descent. Both pros and cons will be illustrated so that helps to put them into use.

Gradient descent is an optimization algorithm totally relies on the first derivatives of the target function. The target function $h(\theta)$ increases fastest if one goes from current position $\theta^{(j)}$ in the direction of the positive gradient of $h(\theta)$ at $\theta^{(j)}$, noted as

$$\theta^{(j+1)} = \theta^{(j)} + \alpha_j \nabla h(\theta^{(j)}|X), \tag{5}$$

where α_j is the step that take on such direction, usually called learning rate. If α_j does not change with the iterations, then it becomes a vanilla version of gradient descent, also named as batch gradient descent,

$$\theta^{(j+1)} = \theta^{(j)} + a \nabla h(\theta^{(j)}|X), \tag{6}$$

which has been widely used due to its simplicity. However, choosing a proper learning rate a is not trivial. A learning rate that is too small leads to painfully slow convergence, while a too large learning rate can hinder convergence or even cause divergence.

To balance the efficacy and accuracy of such method, a variety of related algorithms are proposed and will be introduced in the following sections. Some works on adjusting for the directions, e.g. momentum based algorithms, some adopts different learning rate at each step, e.g. AdaGrad, and others uses different amount of data during the process, e.g. mini-batch gradient descent.

Stochastic Gradient Descent Based Methods

Unlike batch gradient descent, which update parameters based on all available data, stochastic gradient descent based methods only use part of the training data to generate updates. The idea is coming from removing redundant computations in batch gradient descent, as gradients for similar examples have been heavily recomputed before each parameter update. It is therefore much faster.

Stochastic gradient descent (SGD) usually refers to performing parameter updates only rely on single observation, X_i:

$$\theta^{(j+1)} = \theta^{(j)} + a\nabla h(\theta^{(j)}|X_i). \tag{7}$$

Mini-batch gradient descent, on the other hand, takes the best of both worlds and performs an update for every mini-batch of n training examples:

$$\theta^{(j+1)} = \theta^{(j)} + a\nabla h(\theta^{(j)}|X_{(i:i+n)}). \tag{8}$$

This way, it reduces the variance of the parameter updates occurs in SGD, and could lead to more stable convergence. Besides, this method is also way efficient than batch gradient descent since n is usually smaller than 100, a tiny fraction compared to the whole dataset.

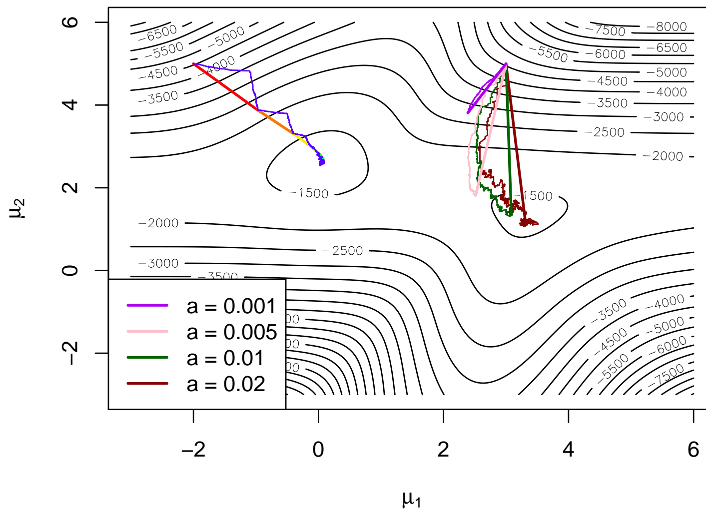

Figure 2: Stochastic gradient descent solutions in first iteration with different learning rate

Table 1: Probability of reaching global maximum start from $(3, 5)$ according to 200 simulated paths

Learning Rate	0.1	0.05	0.02	0.005	0.002
Mini-Batch	NA	82.5%	63.5%	14.5%	0.0%
SGD	32.5%	29.0%	20.5%	5.0%	0.0%

Contrast to the other gradient descent related methods introduced in this section, stochastic gradient descent based methods does not provide strictly deterministic solutions. The results are no longer depends on the initial points but also the order of data

inputs. But once both are given, the path is then fixed. As shown in Table 1, start from same initialization $(3,5)$, both SGD and mini-batch are able to find out global maximum with certain probability, but not 0 or 1. In practice, we suggest to re-shuffle the order of data at each iteration in order to avoid potential patterns in the original data (see the rainbow color path in Figure 2). Besides, a proper selection of learning rate a is essential for being an efficient algorithm.

Vanilla gradient descent methods, expressed in (6), (7), and (8) that share a constant learning rate, usually have trouble navigating ravines or ridges, i.e. areas where the surface curves much more steeply in one dimension than in another, which are common around local optima. In these scenarios, gradients oscillate across the slopes of the ridge while only take hesitant progress along the top towards the maxima as shown in Figure 3. Hence, there are a lot of adjusted gradient descent approaches, which will be discussed in the following sections, developed to overcome the problem.

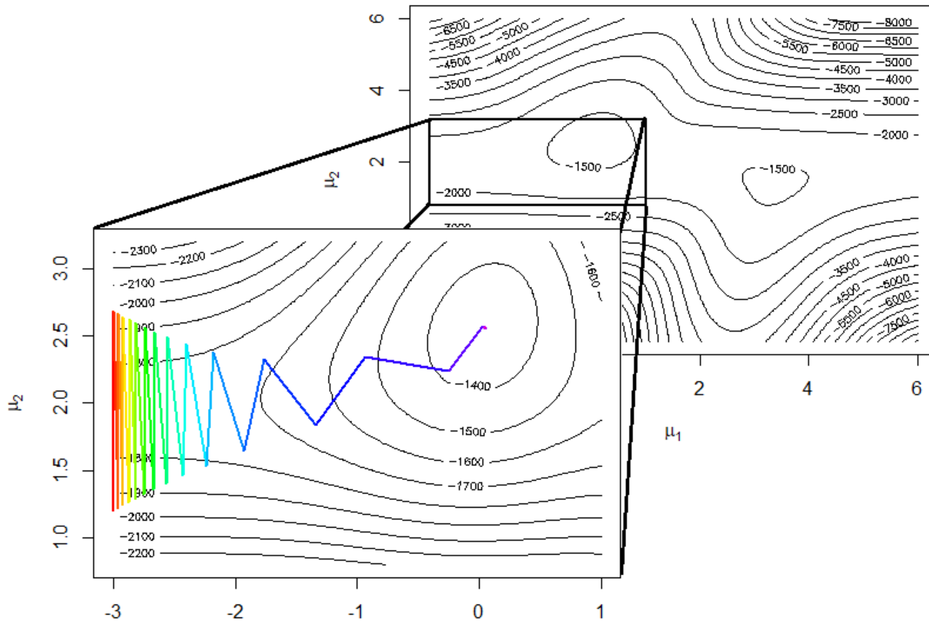

Figure 3: Example of navigating at ridge starting at $(-3,1.2)$

Newton-Raphson Method

The well-known Newton-Raphson method is a specific case of gradient descent method. The learning rate α_j in (5) is defined by the inverse of Hessian matrix, i.e.

$$\theta^{(j+1)} = \theta^{(j)} - [\nabla^2 h(\theta^{(j)}|X)]^{-1} \nabla h(\theta^{(j)}|X). \tag{9}$$

Therefore, Newton-Raphson is the fastest among all alternatives in detecting the root of $\nabla h(\theta|X) = 0$ as long as $h(\theta|X)$ is convex. However, if convexity is not satisfied, Newton-Raphson is an aggressive method comparing to the rest approaches that easily get diverged, or trapped by local optimum and saddle points. Furthermore, the requirement of second-order derivatives makes Newton's method infeasible for high-dimensional data sets in practice, no matter analytically or numerically.

The Newton-Raphson sequences represented in Figure 4 all end up in one of the optimums, but with highly nonlinear patterns. For instance, the starting point $(-2,5)$ corresponds to a steep gradient and thus bypasses the global maxima to end up at a local maxima. Note that, Newton-Raphson is a deterministic solution, i.e. the final results are determined once the initial points are given.

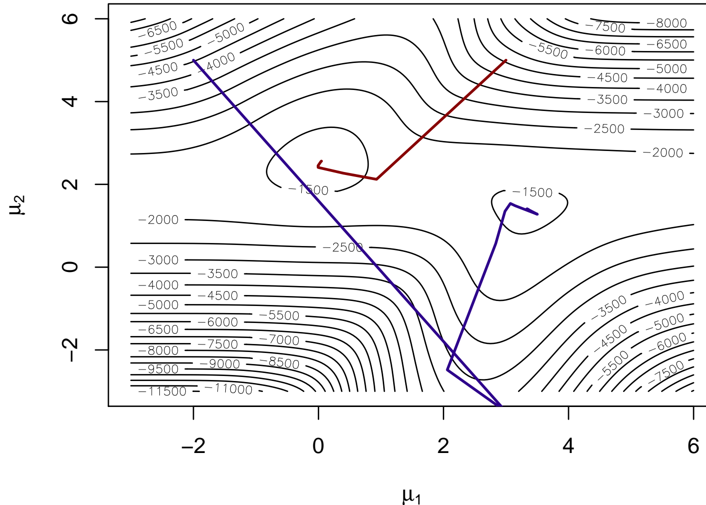

Figure 4: 2 Newton-Raphson sequences starting from $(-2, 5)$ and $(3, 5)$

Momentum Based Algorithms

Momentum based algorithms is an adjusted version of the gradient descent that helps to accelerate vanilla gradient descent in the relevant direction and dampens oscillations as can be seen in Figure 5. This is accomplished by adding a fraction γ of the update vector of the past time step to the current update vector, i.e.

$$v_j = \gamma v_{j-1} + a\nabla h(\theta^{(j)}), \tag{10}$$

$$\theta^{(j+1)} = \theta^{(j)} + v_j, \tag{11}$$

which is the most traditional **momentum method**. Compared to the vanilla gradient descent method, we simply introduce an additional term γv_{j-1} here, which is the direction coming from the last update, so-called momentum. Note that the fraction parameter γ is always suggested to take 0.9 or even 0.99, that means it takes into account a great proportion of previous step. Larger γ it takes, better performance near ridges/ravines, but worse performance on bowl-shaped areas (see optimum region in Figure 5). This is because momentum term increases for dimensions whose gradients point in the same directions and reduces updates for dimensions whose gradients change directions. As a result, it works perfectly well to cancel out back-and-forth updates but also no longer able to directly point to the optimum. Therefore, a properly tuned γ is critical and from my past experience, larger γ is suitable when the target function is irregular and extreme complex.

To improve the performance on areas other than ridges/ravines, **Nesterov Accelerated Gradient (NAG)** method is proposed, which is also based on the idea of using momentum. The difference is current update is not only taking into account the momentum from last step but also considering the direction after adjusted for the momentum. In other words, it could effectively look ahead by calculating the gradient not with respect to the current parameters but to the approximate future positions of the parameters:

$$v_j = \gamma v_{j-1} + a\nabla h(\theta^{(j)} + \gamma v_{j-1}). \tag{12}$$

Explore it by yourself if get interested.

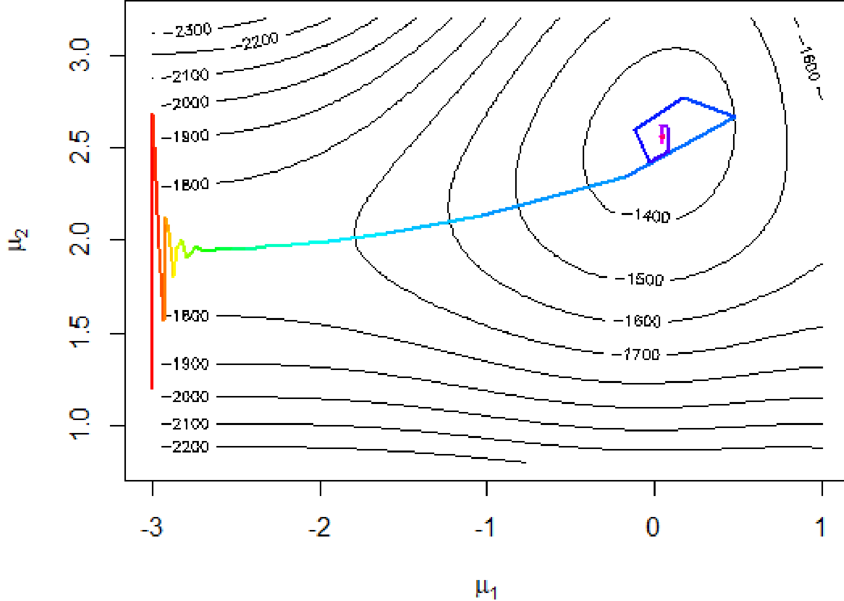

Figure 5: Momentum solution starting at same point $(-3, 1.2)$ with fixed learning rate $a = 0.0025$ and $\gamma = 0.5$

Adaptive Learning Rate Based Algorithms

Previously, we mentioned Momentum based algorithms which adjust direction term $\nabla h(\theta^{(j)})$ in (5). In this section, we propose adaptive learning rate based algorithms, which, on the other hand, focus on adjusting the learning rate term α_j according to the current stage parameters.

Adagrad is a basic algorithm among all adaptive learning rate based methods. It adapts the learning rate to the parameters, performing smaller updates, i.e. low learning rates, for parameters associated with frequently occurring features, and larger updates, i.e. high learning rate, for parameters associated with infrequent features. Obviously, unlike vanilla gradient descent that use same learning rate a at each iteration for each parameter, Adagrad uses different learning rate for every parameter θ_i at every time step j. Denote $\alpha_i^{(j)}$ as learning rate at j^{th} iteration for i^{th} parameter, and $G^{(j)}$ is a diagonal matrix where each diagonal element $G_{ii}^{(j)}$ is the sum of the squares of the gradients with respect to parameter θ_i up to iteration j, then the updating rule is

$$\theta^{(j+1)} = \theta^{(j)} + \alpha(G^{(j)} + \epsilon I_p)^{-1/2}\nabla h(\theta^{(j)}), \tag{13}$$

where $G_{ii}^{(j)} = \sum_{t=1}^{j}(\nabla_i h(\theta^{(t)}))^2$, and ϵ is a smoothing term that avoids numerical issues in inverse (usually on the order of $1e-8$).

One of Adagrad's main benefits is that it eliminates the need to manually tune the learning rate. Most implementations use a default value of 0.01 and leave it at that. Besides, it also avoid to oscillate on the mountain ridge as shown in Figure 6.

The main weakness of Adagrad is its accumulation of the squared gradients in the denominator. Since $G^{(j)}$ is a monotonous increasing function, it in turn causes the learning rate to shrink with iterations and eventually, become infinitesimally small, at which point the algorithm is no longer able to acquire additional knowledge. In practice, Adagrad always converges but does not guarantee to stop at any local optimum. Therefore, a lot of adaptive learning rate based methods are proposed aiming to resolve this flaw, e.g. Adadelta, RMSprop, Adam, AdaMax, etc. Dig through them if interested.

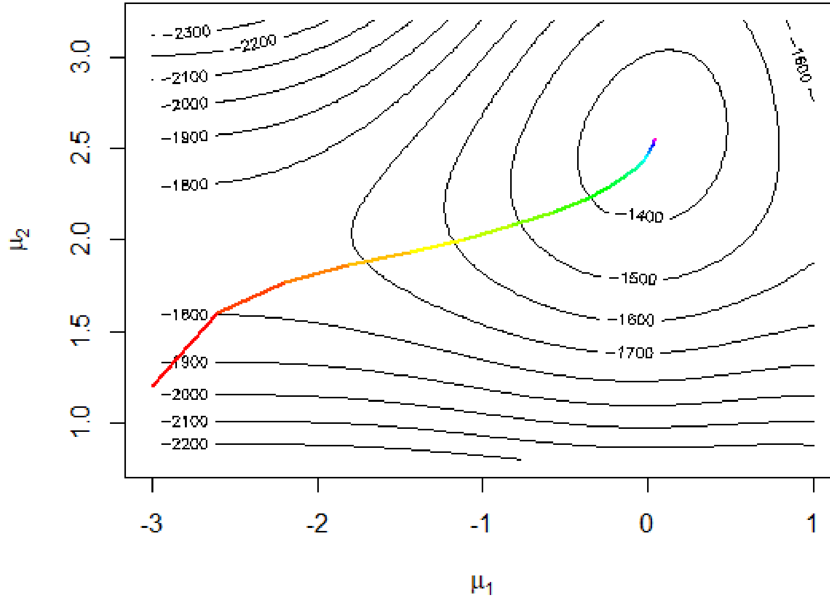

Figure 6: Adagrad solution starting at same point $(-3, 1.2)$ with $a = 0.4$ and $\epsilon = 1e-8$

2.2 Stochastic Solutions

Unlike numerical solutions, most stochastic methods directly find the optimum on the surface of target function over the domain Θ. Since no longer depends on the first derivatives, this simulation-based method has certain chance to directly find out the global optimum without the need of changing initial points, i.e. even start at the same point, the final results could be different. But on the other side, the embedded randomness causes a longer time of reaching convergence.

2.2.1 A Basic Solution

An intuitive way of using simulation to obtain an approximate solution of the optimization problem (1) is to simulate points over Θ according to an arbitrary distribution f with density covers everywhere on Θ, e.g. uniform distribution for every single parameters, until a sufficiently high value of target function $h(\theta)$ is observed. This solution can be extremely inefficient if f is not chosen in connection with target function h, but given an infinite number of simulations and some regularity requirements on the problem, it is bound to converge (Spall, 2003). So this is the last method that can rely on if all other methods failed.

Recall the example described in (3), we simulate 2000 pair of (μ_1, μ_2) from $(u_1, u_2) \sim \mathcal{U}(-3, 6)$ and record the $h = \max\left(logL(u_1^{(1)}, u_2^{(1)}), ..., logL(u_1^{(2000)}, u_2^{(2000)})\right)$ as as approximation to the solution of (1). Multiple sequences are simulated to evaluate the variability of the solutions, and the result is shown in Figure 7. While the starting value $logL(u_1^{(1)}, u_2^{(1)})$ of the sequence is highly random, the range reduces very quickly and, after 2000 iterations, the worst sequence among 500 parallel sequences is within a reasonable distance from the true maximum.

There are some properties of this basic stochastic approach. Comparing to grid search algorithm, which is a non-stochastic version of this solution, it can stop earlier with a reasonable high $h(\theta)$, but the accuracy, which can be carefully controlled in grid search method, is unknown. Most obvious drawback is this blind solution-blind in the sense that it does not take target function h into account-quickly impractical as the

dimension of h increases. Hence, all the coming stochastic solutions rely more on less on the target function h to become more efficient.

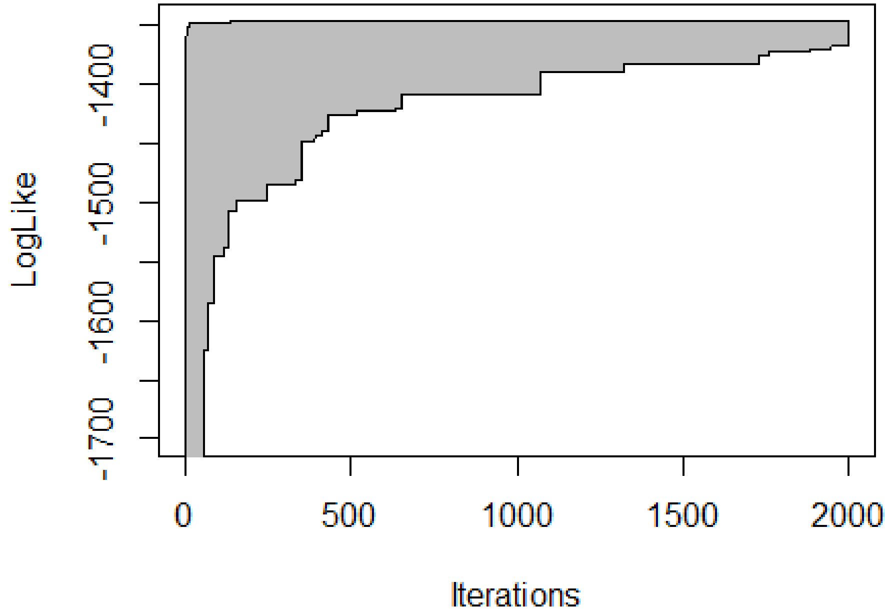

Figure 7: Range of 500 sequences of successive maxima found by random uniform sampling over 2000 iterations. The true maximum value is identified by the horizontal line on the top of the graph

2.2.2 Stochastic Gradient

Due to the inefficiency blind explore on the surface of $h(\theta)$, a different stochastic approach to maximize h is to search in a local manner, i.e. the updated θ_{j+1} is dependent on θ_j linearly:

$$\theta^{(j+1)} = \theta^{(j)} + \epsilon_j, \tag{14}$$

where ϵ_j is the local perturbation of the current value. Note that this perturbation ϵ_j could be simulated from an arbitrary distribution such as a $\mathcal{N}_p(0, \sigma^2 I_p)$ distribution. However, there will not be a major improvement from basic stochastic search algorithm. Besides, given that we are specifically searching for the maximum of h, using some information about h in constructing the distribution of the perturbation is bound to increase the efficiency of the method. In particular, it makes a lot of sense to prefer moves increasing in h over moves decreasing h, even though the latter should not be impossible if you want to avoid trapping at local maxima. An intuitive way of involving h into updating is to use the gradient of h, i.e. $\nabla h(\theta)$. Then this turns to be a deterministic way as we have seen in (5).

Take advantage of this idea and also involve stochasticity into perturbation, stochastic gradient method is proposed. Note that this name is similar to stochastic gradient descent method, only without descent, but they are fundamentally different. Previous one is purely stochastic, but latter one is determined once sequence of data is given. In stochastic gradient methods, the direction of the move is randomly chosen over the unit sphere $||\zeta|| = 1$, which is the stochastic part. But the step size along the direction is determined by the finite-difference of h, which borrows information from h. In other words, the perturbation ϵ_j keeps the size of $\nabla h(\theta^{(j)})$ but take a random direction ζ_j,

which expressed as

$$\theta^{(j+1)} = \theta^{(j)} + k\Delta h(\theta^{(j)}, \beta_j \zeta_j)\zeta_j, \tag{15}$$

where

$$\Delta h(\theta^{(j)}, \beta_j \zeta_j) = h(\theta^{(j)} + \beta_j \zeta_j) - h(\theta^{(j)} - \beta_j \zeta_j). \tag{16}$$

In contrast to the deterministic approach, the stochastic gradient method does not proceed along the steepest slope of $h(\theta^{(j)})$, but this property is generally a plus in the sense that it may avoid being trapped in local optimum or in saddlepoints of h (see Figure 8).

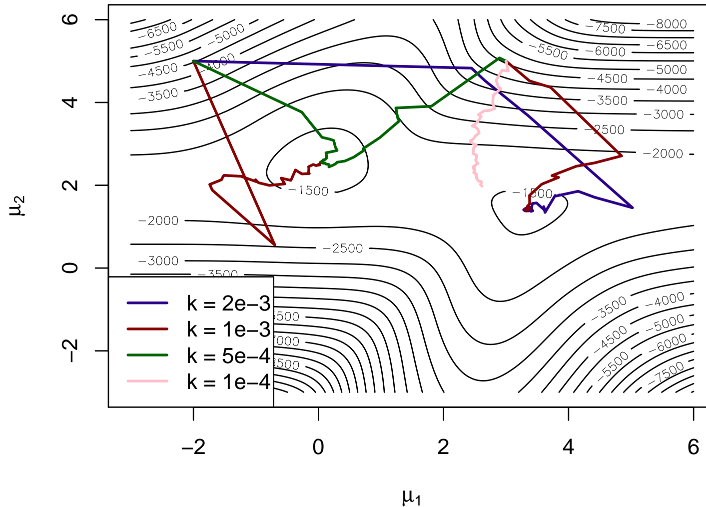

Figure 8: Stochastic gradient paths for four different choices of k either starting from $(-2, 5)$ or $(3, 5)$, given $\beta_j = 1/\sqrt{log(1+j)}$

The fixed learning rate k and monotonous decreasing sequence β_j should be carefully tuned simultaneously according to the target function h. The size of the update is jointly depend on k and β_j. Since k is usually fixed, a properly decreasing β_j with the number of iteration helps to increase the accuracy when searching in the neighborhood of the optimum. In particular, if β_j decreases too fast or k is too small, then it might not be able reach the targeted neighborhood before step size becoming infinitesimally small. On the other hand, if β_j decreases too slow or k is too large, then it not only takes long time to converge, but also has greater chance to diverge. For instance, in our example, we simulate 500 paths at same starting point $c(3, 5)$ with different k to check the distribution of final result, which is shown in Figure 9. From the leftmost bar in each plot, approximately more than 50% paths achieve the global maxima with proper k. Furthermore, even a lot of randomness is involved, the algorithm still converges very fast, usually in 25 iterations.

2.2.3 Simulated Annealing

This alternative method construct the sequence in (14) by simulating the perturbation ϵ_j in a very different manner. Rather than only take the move towards a larger target function value, which is the conventional rule that adopted in all previous methods, simulated annealing accepts some updated $\theta^{(j+1)}$ that seems to decrease h with certain

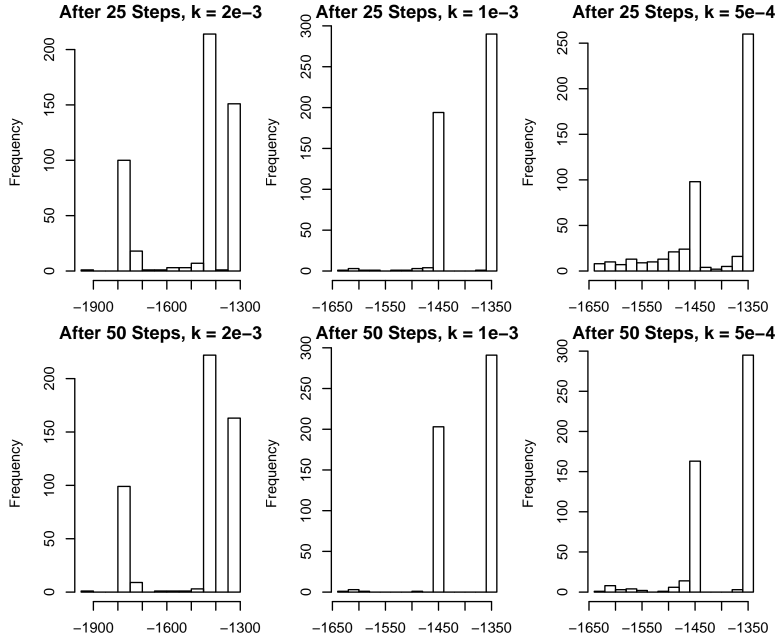

Figure 9: Distribution of maximum found after 25 and 50 steps with different k values using stochastic gradient methods 500 times starting from $(3, 5)$

probability ρ_j. As aforementioned, this mechanism helps to escape from potential local optimum after getting trapped.

The construction of the sequence of ρ_j is obviously the central issue when designing a simulated annealing algorithm. The most standard choice is based on the Boltzman-Gibbs transforms of h,

$$\rho_j(\theta) = \min\{\exp(\Delta h(\theta^{(j)})/T_j), 1\}, \tag{17}$$

where the sequence of temperatures, $\{T_j\}$, is decreasing with the number of iteration j; and $\Delta h(\theta^{(j)}) = h(\theta^{(j+1)}) - h(\theta^{(j)})$. It indicates that if the next move is on the right direction, i.e. $\Delta h(\theta^{(j)}) > 0$, then $rho_j(\theta) = 1$ and new value is automatically accepted. While even if new move decreases h, it still may be accepted with probability $\exp(\Delta h(\theta)/T_j)$. Higher the temperature T_j, it is more likely a 'wrong' step would be taken given the same $\Delta h(\theta)$, but when iteration last long, i.e. temperature is very low, the probability of accepting a reverse update is almost 0. Therefore, convergence is guaranteed when iteration is large enough.

This updating rule, which is related to Metropolis-Hastings algorithm, can be represented by

$$\theta^{(j+1)} = \begin{cases} \theta^{(j)} + \beta_j\zeta_j & \text{with probability } \rho = \min\{exp(\Delta h(\theta^{(j)})/T_j), 1\}, \\ \theta^{(j)} & \text{with probability } 1 - \rho, \end{cases}$$

where the perturbation $\epsilon_j = \beta_j\zeta_j$ is jointly driven by a decreasing sequence $\{\beta_j\}$, which quantify the step size, and a randomly picked direction over a unit sphere, $||\zeta|| = 1$. Selection of either β_j and T_j are two practical issues that hinder the implementation of this otherwise attractive algorithm. Usually we take $T_j = 1/\sqrt{log(1+j)}$, $1/log(1+j)$, or $1/(1+j)$, and $\beta_j = \sqrt{T_j}$. But they are quite problem-dependent and need to be carefully tuned.

The performance and result of simulated annealing applied to our example is shown in Figure 10 and Table 2. In the figure, slow-cool adopts $T_j = 1/log(1+j)$, middle-cool

Table 2: Probability of Reaching Global Maximum Start from $(3,5)$ using simulated annealing algorithm

Cool-Down Speed	Slow	Median	Fast
Small Scale	55.2%	50.6%	36.0%
Large Scale	54.8%	53.4%	46.2%

adopts $T_j = 1/(1+j)$, and fast-cool adopts $T_j = 1/(1+j)^2$. In the table, small scale uses $\beta_j = T_j^{0.5}$, and large scale uses $\beta_j = T_j^{0.3}$. If cool down too fast, then it is likely that algorithm converges before reaching to any local optimum. On the other hand, too slow in cooling down is inefficient. Meanwhile, this algorithm also has the power of detecting global maxima with a relatively considerable probability.

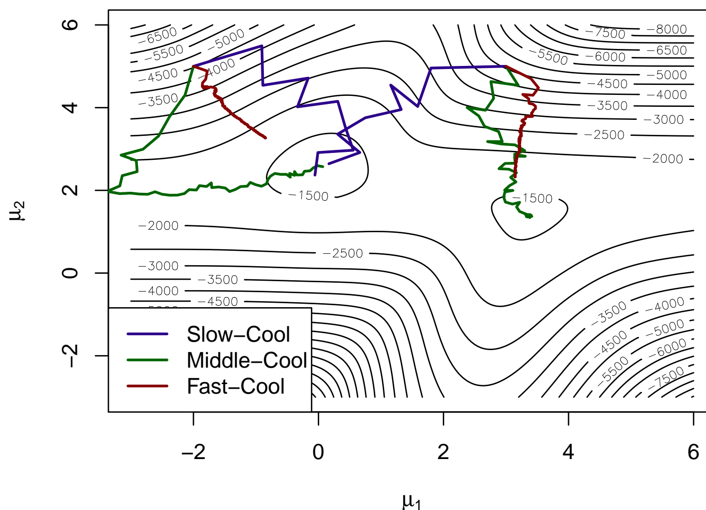

Figure 10: Simulated annealing paths for different choice of $\{T_j\}$ either starting from $(-2,5)$ or $(3,5)$, given $\beta_j = T_j^{0.5}$

3 Stochastic Approximation

Before introduction of stochastic approximation, let's first give a closer look to our example in (2). It is a mixture distribution problem that as long as the group indicator δ_i for each observation is observed, the question could be easily and precisely solved. For such missing information problem, EM algorithm can easily be implemented. The main reason is because the surface of observed log-likelihood function

$$l^o(\theta|x) \propto \sum_i log\{\pi exp(-(x_i-\mu_1)^2/2\sigma_1^2) + (1-\pi)exp(-(x_i-\mu_2)^2/2\sigma_2^2)\} \qquad (18)$$

is always flat around the global optimum so that has a non-identifiable issue. While the surface of complete log-likelihood function

$$l^c(\theta|x,\delta) \propto -\sum_i \delta_i(x_i-\mu_1)^2/\sigma_1^2 - \sum_i (1-\delta_i)(x_i-\mu_2)^2/\sigma_2^2 \qquad (19)$$

is well-shaped and easily to find out the global optimum, as shown in Figure 3. So EM algorithm first estimates the expectations of missing information, $E(\delta_i|x, \theta^{(k)})$, based on current parameters (E-step), and then plug those expectations in to the completed log-likelihood function to obtain the updated parameters $\theta^{(k+1)}$ (M-step). Repeat E- and M-step iteratively until convergence. Therefore, the good property of complete log-likelihood function is borrowed into the analysis even without knowing δ_i. In our example, with simple algebra, we obtained that

$$E(\delta_i|x, \theta^{(k)}) = \frac{\pi\phi((x_i - \mu_1)/\sigma_1)}{\pi\phi((x_i - \mu_1)/\sigma_1) + (1-\pi)\phi((x_i - \mu_2)/\sigma_2)}, \tag{20}$$

and by M-step, the updating rule for $\theta = (\mu_1, \mu_2)$ is

$$\mu_1^{(k+1)} = \sum_i x_i E^{(k)}(\delta_i) / \sum_i E^{(k)}(\delta_i); \tag{21}$$

$$\mu_2^{(k+1)} = \sum_i x_i [1 - E^{(k)}(\delta_i)] / \sum_i [1 - E^{(k)}(\delta_i)]. \tag{22}$$

The result of applying EM algorithm to our example is represented in Figure 12. EM is a deterministic approach that is guaranteed to achieve either local or global optimum. So try EM multiple times with different initial points is recommended if one wants to avoid local optimum.

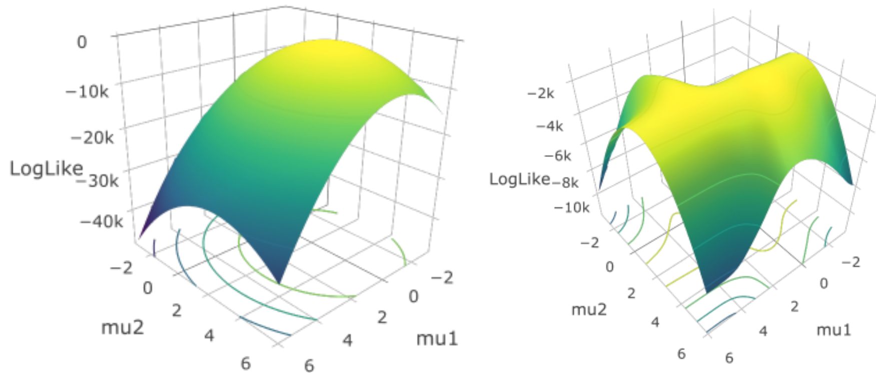

Figure 11: (left) Complete log-likelihood surface; (right) Observed log-likelihood surface

3.1 Monte Carlo EM (MCEM)

A difficulty with the implementation of the EM algorithm is that each E-step requires the computation of the expected complete log-likelihood function, namely $Q(\theta|\theta_0, x) = E_\theta(l^c(\theta_0|x))$. In some cases, $Q(\theta|\theta_0, x)$ may not have a closed-form solution or the calculation could be complex. But considering Q is naturally expressed as an expectation, it can be directly approximated by using Monte Carlo method instead. For instance, if we know the conditional distribution $f(\delta|x, \hat{\theta}^{(k)})$ in previous example, we then are able to directly estimate $E(\delta_i|x, \theta^{(k)})$ by averaging Q by plugging in the sampled δ_i from f:

$$\hat{Q}(\theta|\theta_0, x) = E(l^c(\theta_0|x, \delta)) = \frac{1}{m}\sum_{i=1}^{m} l^c(\theta_0|x, \delta_i). \tag{23}$$

Such method is suggested by Wei and Tanner (1990) under the name of Monte Carlo EM (MCEM).

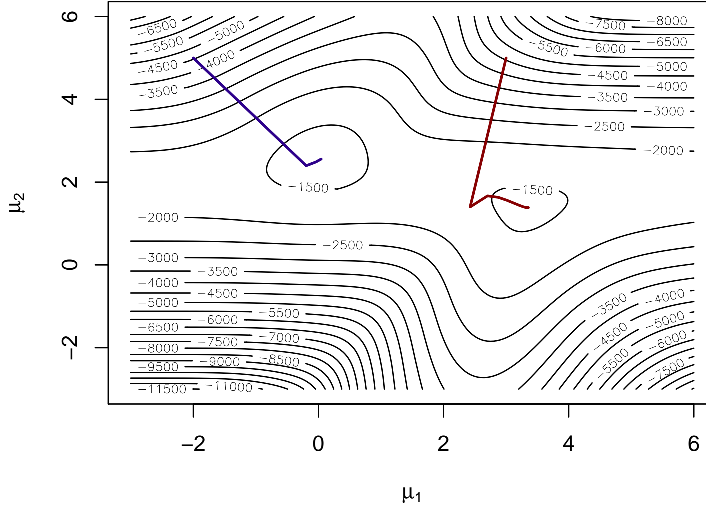

Figure 12: EM paths either starting from $(-2, 5)$ or $(3, 5)$

In our example, for illustration purpose, MCEM solution would replace the expectation in (19) with the empirical average

$$E(\delta_i|x, \theta^{(k)}) = \frac{1}{m}\sum_{i=1}^{m}\delta_i, \tag{24}$$

where δ_i is sampled from its binomial distribution

$$\delta_i \sim Binom(1, \frac{\pi\phi((x_i - \mu_1)/\sigma_1)}{\pi\phi((x_i - \mu_1)/\sigma_1) + (1-\pi)\phi((x_i - \mu_2)/\sigma_2)}). \tag{25}$$

The Monte Carlo step number m can be chosen arbitrarily. A larger m will give better approximation of Q but also take longer time to convergence. Note that the MCEM algorithm no longer enjoys the fundamental EM monotonicity property. It is therefore important to assess that the sequence of parameters produced by the MCEM algorithm converges to an approximate maximum of the model likelihood.