CSE257 Winter 2023: Assignment 2

Deadline: Feb-12 11:59pm.

- Submit the pdf file of your answers on Gradescope.
- Upload zipped source code to `https://www.dropbox.com/request/htZjTEJIiSnFcl6MqOqb`.

Question 1 (1 Point)**.** We talked about the basic Monte Carlo methods for estimating the expectation $\mathbb{E}_{x\sim p(x)}[f(x)]$, where $p(x)$ refers to the density function of some probability distribution over the state space of the random vector x. The method takes k i.i.d. samples, $x_1, ..., x_k$, and use the average of their values to estimate the real expectation, i.e.,

$$\frac{1}{k}\sum_{i=1}^{k} f(x) \longrightarrow \mathbb{E}_{x\sim p(x)}[f(x)]. \tag{1}$$

We know the estimate gets better and better as k increases, from the central limit theorem. In practice, a common problem is that $f(x)$ may have large or "important" values only at very small sets of inputs. Then by taking a limited number of random samples, it may be hard to hit those rare inputs that may contribute significantly to the overall expectation. Also, p itself may just be hard to sample from. To handle such cases, *importance sampling* techniques are crucial. The idea is to design a potentially very different distribution with density function $q(x)$, which is easy to sample from or is more consistent with f, and then use the equation:

$$\mathbb{E}_{x\sim q(x)}[\frac{p(x)}{q(x)}f(x)] = \mathbb{E}_{x\sim p(x)}[f(x)]. \tag{2}$$

This equality allows us to draw samples according to the new density function $q(x)$, and by changing $f(x)$ to $\frac{p(x)}{q(x)}f(x)$ on each of the sample, we can use the estimate of the left-hand side of the equality in (2) to estimate the original expectation in (1). This "re-parameterization trick" is the core step in many sampling-based methods.

That is just the background of the question. What you need to do is just: Prove Equation (2). It follows easily from some of the steps we took to derive the search gradient formula.

Question 2 (Implementation Required)**.** Consider the following functions:

- $f_1(x) = x_1^2 + x_2^2$
- $f_2(x) = -\dfrac{1+\cos(12\sqrt{x_1^2+x_2^2})}{0.5(x_1^2+x_2^2)+2}$ (this is called the drop-wave function)
- $f_3(x) = \sum_{i=1}^{50} x_i^2$ (yes it's a 50-dimensional function, don't freak out)

You will need to implement the following algorithms to answer the questions below.

- (GD) Gradient Descent with a fixed step size $\alpha = 0.1$.
- (SA) Simulated Annealing with initial temperature T and the annealing schedule $T_k = T/k$ for each k-th iteration (in the first iteration $k = 1$). Each Δx step should be sampled from a Gaussian around the current x (think about what that means) with the identity matrix as the covariance matrix.

- (CE) Cross-Entropy Methods with k samples in each iteration, and the elite set is formed by the top 40% of all samples in each iteration (whether "top" means higher or lower function value depends on whether you are minimizing or maximizing).

- (SG) Search gradient with k samples in each iteration, and a fixed step size of $\eta = 0.01$. Use the normalized gradient $\nabla f(x)/\|\nabla f(x)\|_2$ in each update instead of just $\nabla f(x)$, so that after multiplying with the step size, each update step is small in magnitude, which is often used to avoid divergence.

In all algorithms, whenever you need to use a distribution that has not be specified by the algorithm, always use the Gaussian distribution $\mathcal{N}(\mu, \Sigma)$ with the appropriate size, where Σ is initialized as an identity matrix. You will then need to make relevant updates to it when that's part of the steps in an algorithm.

For each function, our goal is to **minimize the function**, so make sure your updates are in the right direction, which can be different from the formulas in the slides. For all functions, the initial point is always $x_i = 2$ for each variable x_i used in the function. For each algorithm, always **run 100 iterations from the initial point**. If any algorithm diverges ($\|x\|$ gets too large), you can cut off the values at some big enough number of your own choice in your plot.

For algorithms that involve randomness (e.g. sampling), it is important to understand how randomness affects their performance. Thus, in all questions below, whenever there is sampling involved in an algorithm, you should plot 5 different runs (i.e. **5 different random seeds** of your choice) of the algorithm on the same function in the same graph. That is, starting at the same initial point, since each step will be stochastic, you should plot 5 different trajectories for each run (i.e., 5 sequences of points, use a different color for each sequence).

1. (1 Point) Plot in 3D the drop-wave function f_2 defined above over the domain $(x_1, x_2) \in [-5, 5] \times [-5, 5]$ (the vertical axis should show the value of $f_2(x_1, x_2)$ over this domain).

2. (1 Point) Perform GD on each function. Plot how the function value changes with respect to the number of iterations (x-axis: number of iterations, y-axis: function value; same for all the following plots as well).

3. (2 Points) Perform SA with two different initial temperatures $T = 1000$ and $T = 10$, for each function. Plot how the function values changes over iterations. (3 functions and 2 different temperatures, so 6 plots in total.)

4. (2 Points) Plot the performance of CE with two sample sizes, $k = 10$ and $k = 100$, for each function (so 6 plots in total). For each iteration (with each fixed random seed), you have a bunch of samples of $f(x)$ and you should use their average to be the current function value that the algorithm finds at that iteration in your plots.

5. (2 Points) Plot the performance of SG with two sample sizes, $k = 10$ and $k = 100$, for each function (6 plots in total). Plot the average of the function values of all in each iteration for each function. If you see the error of "covariance is not positive-semidefinite" explain why that is (if you didn't see it, still explain why you may see it), and you can skip the update on the covariance matrix when that happens.

6. (1 Point) Compare the performance all algorithms (and all versions) on the functions above and give some guidelines about when to use which method and why (in terms of dimensions, convexity, randomness, etc.). You should create some new plots that allow people to compare different algorithms on the same plots. There is no single correct answer, just focus on explaining your own understanding.

7. (2 Extra Points) As you may have observed, computing the covariance matrix can be the main bottleneck for using a lot of samples (N^2 elements in the matrix). A typical choice is to just use the diagonal elements in the covariance matrix to simplify the computation (N elements on the diagonal). Focus on the CE method. Design two functions such that the diagonal approximation does not affect the performance of CE on the first function, while it changes the performance on the second function quite a lot. Formulate some guidelines about when you can do this approximation. Make your own definitions of "does not affect" and "quite a lot" as long as they are reasonable. Again there is no unique correct answer, just explain your own observations.

Question 3 (1 Point)**.** Prove the Separation Property for the Dijkstra algorithm. That is: after each iteration of the algorithm (i.e., after the frontier has been fully updated in an iteration), any acyclic path from any state in the explored

set to any state in the unexplored set has to pass through some state in the frontier set. Note that the "acyclic" assumption here simplifies the proof and it is not important, because for any path that contains cycles we can always ignore the cycling part and only consider its acyclic part.

Question 4 (1 Point)**.** Construct a small undirected graph (choose your own start and goal and connectivity) and design the cost for each node (or edge, up to you) on the graph. All costs need to be positive. Design a heuristic function h such that satisfies the following conditions

- h itself is consistent (and of course, non-negative).
- By multiplying it by 3 (i.e. the heuristic value becomes $3h(s)$ for each node s), the new heuristic is inconsistent and misguides A^* search, such that it no longer returns the optimal path in the end.

Explain the computation steps of A^* in each case to explain how the second heuristic value misguides the search.

Question 5 (2 Points)**.** Consider a non-zero-sum game where you do not have min and max players because their goals are not in conflict. Instead, just Player A and Player B, each having their own objectives that define the winning or high pay-off states, but when one player wins the other doesn't need to lose. The game is still finite in length (you can assume a bound on the total number of steps that each play can take in the game). Each player still take turns and has a finite number of actions to take in each step. (Imagine a game on the chess/go board, where the players are not fighting but just to build their own patterns on the board, possibly even collaboratively sometimes.)

Design an algorithm that finds the optimal strategy for both players at all game states in this non-zero-sum setting, similar to what the minimax algorithm does. With the algorithm you design, show how it works on an example of a small game-tree. All the mathematical details are up to your own definitions.