Christopher Sullivan

Professor O'Neill

CS-390

3/30/23

Randomized Optimization: An Algorithmic Analysis

Introduction

Learning Problems:

Within this study, performance analysis was conducted on various random optimization algorithms, measured across three unique loss functions from the *mlrose* Python package. To begin, these optimization techniques were each applied in the development of a neural network, with the target being a state vector containing the optimal value of each weight parameter. The creation and weight optimization of these neural networks was based on a previously used dataset from 'Supervised Learning Analysis'. The file, diabetes.csv, is a set of records of Pima Indian women who were surveyed on various health factors; the binary classification variable for this set is whether or not the patient tested positive for diabetes. Outlined in the study is a brief comparison between the *scikit-learn* MLPClassifier, and the randomly optimized *mlrose* neural networks.

Following the weight optimization, the algorithms were presented with the Four Peaks problem, a maximization problem in which the target is a bit-string with the greatest amount of leading 1's and trailing 0's over a threshold T. This problem is an interesting one for comparing optimization techniques due to the search landscape which it produces. Notably, the structure of this problem results in a layout containing two local maxima, and two possible global optima.

The local maxima are strings comprised of solely homogenous bits, and the global maxima are the target strings which maximize leading and trailing bits.

Lastly, the n-Queens problem was run through each of the algorithms, with the target being the minimal number of pairs of queens which may attack each other on a theoretical (n x n) chess board. Given random queen placement states, each algorithm attempts to return the optimal placement subset which will minimize collisions between n number of queens. This problem domain contains n^n possible varying states, and thus it provides a large search landscape with many low-point valleys. Based upon the aforementioned reasons, these three problems should aptly highlight the differences of unique optimization algorithms in practice.

Algorithms:

Starting out, the Random Restart Hill Climbing algorithm, or simply Hill Climbing as it will be henceforth referred to, was applied to each problem. This algorithm is a flavor of simple hill-climbing, in which the base algorithm is performed on the dataset a given of k times. Upon each iteration, a random starting point is chosen, and a neighbor function is used to 'climb' from the current point until we reach a point that we cannot improve upon. Thus, we sacrifice time through iterations in exchange for enhanced data-exploratory function from the algorithm. Because of the random restarts feature, it is likely that this Hill Climbing technique will find peaks within a dataset given enough restarts. However, due to its inability to reduce fitness for long-term gain, it may always become stuck on local maxima, and never reach the target value.

Additionally, each problem was given to *mlrose*'s version of a Simulated Annealing algorithm. Like Hill Climbing, the goal is to find the optimal value in a search landscape, by repeatedly using a neighbor function. However, unique to this algorithm is the temperature parameter; starting at an initial value, temperature determines the algorithm's willingness to

reduce fitness or take ‘downwards steps’, and gradually decreases over iterations. Because of this calculated risk-willingness, Simulated Annealing is theoretically more suited to exploring a dataset than Hill Climbing, as it is able to cross over valleys in order to reach higher peaks.

Finally, the last algorithm applied to each problem was a Genetic Algorithm. Unlike Hill Climbing and Simulated Annealing, the Genetic Algorithm does not employ a neighbor function in order to increase fitness. Rather, it calculates the fitness of k random instances, creates new instances based off these fitness values, and presents a slight chance of ‘mutating’ any instance. Using these techniques, the algorithm eventually converges on or near a certain point, with the mutation chance present in order to increase exploration away from this point. Knowing this, it can be assumed that genetic algorithms are capable of covering a wide search landscape, and are resistant against dips in instance fitness.

Overall, given the structuring of each optimization problem and the tendencies of each algorithm, analysis commenced with some initial expectations: The Genetic Algorithm would provide the most optimal neural network weights, Simulated Annealing would be the most efficient at solving Four Peaks, and again that the Genetic Algorithm would dominate the n-Queens problem.

Neural Network Weights

In order to produce the best results, the first step in searching for the optimal weights was to limit the weight range to {-1, 1} so as to produce values fitting of a neural network. When attempting to produce the optimal weight vector, each algorithm was run a standard number of times, employing more iterations each time. While the initial range was {100-1,000} iterations across each algorithm, *mlrose’*s basic implementations for this problem caused this to be an issue when producing data. The implementations for neural networks using Hill Climbing and the

Genetic Algorithm included parameters which set the number of restarts and population size respectively. This meant that for *each* iteration, these algorithms would consider $(10 \cdot k)$ states at any given time, as 10 neighbors or genetic 'offspring' are checked per attempt.

However, a neural network using Simulated Annealing runs the algorithm k times, with only a fixed neighbor-checking rate of 10 per iteration. Therefore, although this attempt-per-iteration rate is set to 10 for each algorithm, Hill Climbing and the Genetic Algorithm had the advantage of considering ~(10(k) * *iterations*) total instances throughout the problem, while Simulated Annealing was limited to ~(10 * *iterations*) total instances. While this may be viewed as an inherent difference between the algorithms, this structure would be a limited method of analyzing their capabilities, and proved to be an apparent issue in the results; Simulated Annealing was performing significantly worse than its counterparts, despite theoretically being better suited than Hill Climb.

In an effort to remedy this anomaly, the way in which attempts were measured was modified. Rather than a constant number of iterations across the board, Simulated Annealing would iterate as many times as the other two algorithms' iteration count, multiplied by their restart or population count. In doing so, the data appears to be more consistent with the techniques given the problem. Additionally, due to the way in which these weight optimization methods are implemented in *mlrose*, timing their executions was beyond the scope of this project, and thus neural network run-times are explained relatively.

Hill Climbing:

During testing, the algorithm was run 50 times, with five executions at each pass. The first pass began with 100 iterations, and subsequently increased by 100 up to 1,000 iterations. Following incremental testing, the optimal number of restarts for this algorithm was found to be

200, considering both performance and run-time. Since this algorithm is monotonically non-decreasing, it follows that a higher number of restarts will always result in equal or greater optimization. However, beyond 200 restarts, the jump in run-time became noticeable upon execution, with each run of 300+ iterations taking a substantial amount of time locally.

Furthermore, of the three algorithms applied, the weight vector produced by Hill Climbing resulted in the least accurate neural network classifier. To keep consistent across the tests, the same number of hidden layers and perceptron nodes were used in each classifier. The highest training accuracy reported by the randomly optimized neural network was 58%, and its maximum reported test accuracy was 60%. Given the algorithm's bias towards local maxima, it makes sense that many sub-optimal weights across the vector would cause decreased performance. In comparison to the MLPClassifier used from *scikit-learn*, this neural network came relatively close in terms of accuracy, but did not do as well. As the MLPClassifier recorded a maximum test score of 73.95% through backpropagation, there is a ~14% difference between the two. However, it should be noted that the score for the MLPClassifier was calculated using a standard accuracy formula, while the *mlrose* neural network was scored using the F1 scoring formula. Pictured below is the algorithm's relative fitness curve at the median number of 500 iterations. During classification, the neural network behind this chart produced only 26% test accuracy. This showcases the random nature of the algorithm, demonstrating that one run can result in a weight vector which produces a weak learner, while the next produces a classifier which is far worse than natural chance. Additionally, the chart showcases the algorithm's monotonic traits, as its fitness value only rises or remains static across iterations.

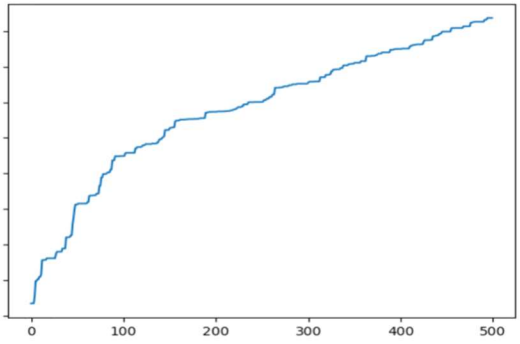
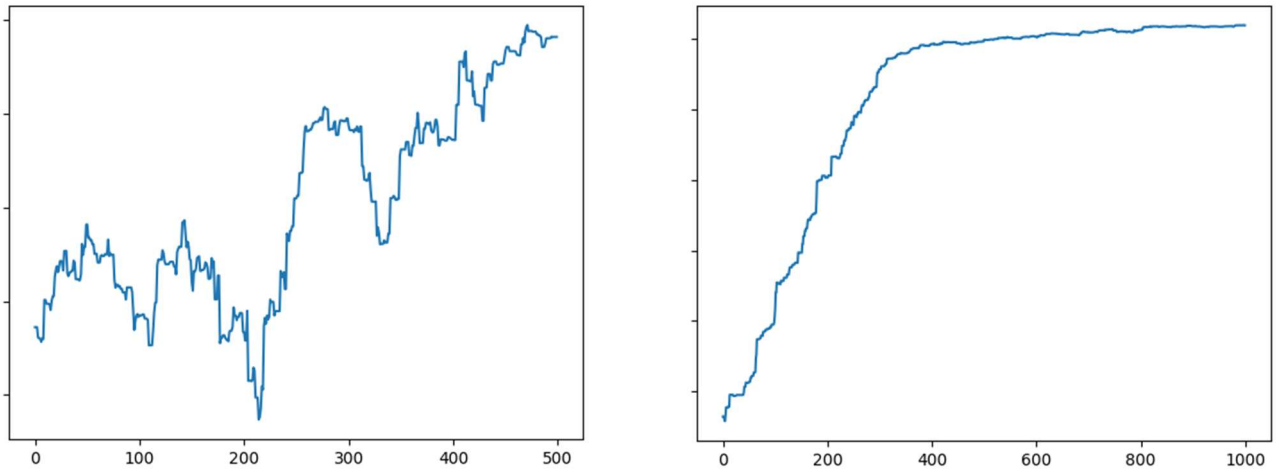


Figure 1.1: *Random Restart Hill Climb Fitness Curve*

Simulated Annealing:

Following Hill Climbing, Simulated Annealing was used to search for the optimal weights of the classifier. Of the three algorithms, this was the one which produced the most optimal classifier, resulting in a maximum recorded training accuracy of 70% and a max test accuracy of 71%. Compared to the MLPClassifier, there is only about a 3% difference in accuracy, showcasing improvement from Hill Climbing. Additionally, in comparison with the other algorithms, this implementation of Simulated Annealing runs in a fraction of the time, executing practically instantaneously at lower iteration counts.

To prepare for analysis, the algorithm was run 50 times, starting at five runs of 2,000 iterations and working up to 200,000. During coding, attempts were made to tweak the default cooling schedule of the algorithm. The initial temperature and decay factor were increased and decreased in intervals, and subsequently tested. However, nothing that was attempted appeared to produce greater accuracy in the final classifier. Overall, the success of Simulated Annealing can be attributed to its ability to take risks through the search field, allowing it to overcome local maxima and achieve higher optimization.

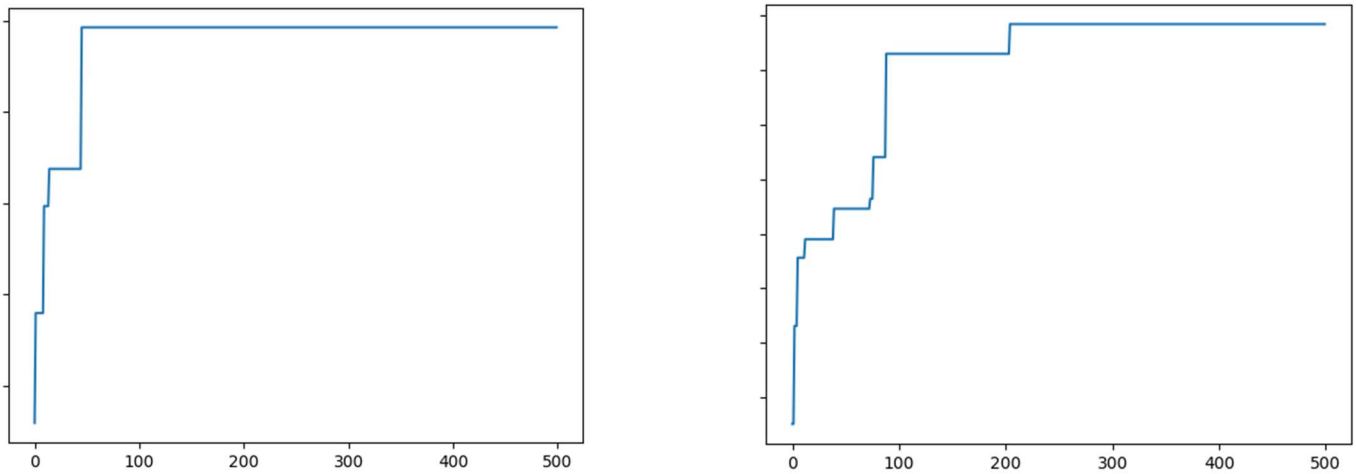


Figures 1.2, 1.3: *Simulated Annealing Fitness Curve using Default Cooling (L), Tweaked Cooling (R) Raised initial temperature, lowered decay results in a curve which is far more unwilling to decrease fitness.*

Genetic Algorithm:

Lastly, the final algorithm to attempt to optimize the neural network was the Genetic Algorithm from *mlrose*. While it was expected to produce the most optimal result, it fell short of Simulated Annealing in practice. It is likely that this low performance is due to the fact that this is a continuous optimization problem, which genetic algorithms tend to do worse at than discrete problems. Again, the algorithm was run 50 times over {100-1,000} iterations, with a population size of 200 for consistency. With a maximum training and test accuracy rate of 65%, though, the algorithm performed better than Hill Climbing. Considering the two algorithms and how many iterations they were given, this checks out as the Genetic Algorithm has better exploratory properties.

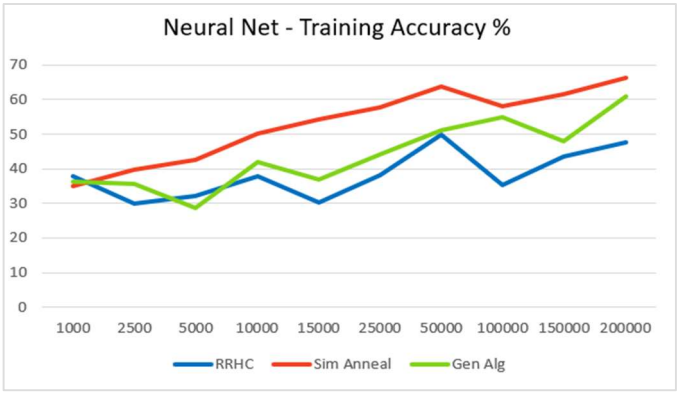
One thing to note in the code is the '*mutation_prob'* variable of this algorithm, and how it affects outcomes. Below are two relative fitness curves, one from a run with the default *mutation_prob* of 0.1, and another with a raised value of 0.475. Upon raising this value, average test accuracy increased from 53% to 62%, representing a measured 9% gain on the final supervised learning model. This is most likely due to the data exploration gained over the low number of iterations, which allowed the algorithm to break free of typical convergence points and find higher maxima.

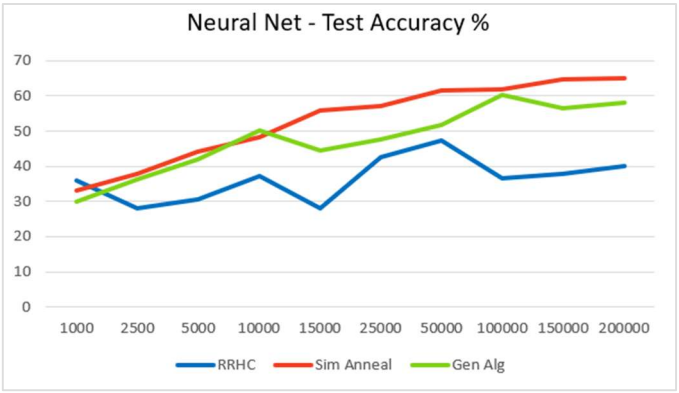


Figures 1.4, 1.5: *Change in relative fitness curve before (L) and after (R) mutation_prob adjustment. Flat lines indicate iterations without fitness gain.* ***Figure 1.5*** *illustrates the gains produced by exploratory mutation.*

Performance:

Overall, the neural network weight vector proved an interesting optimization problem for comparing these three algorithms. While Hill Climbing, the Genetic Algorithm, and arguably Simulated Annealing would all benefit from increased iterations, the base rate proved effective in contrasting the tendencies of each, and ultimately showcased weak learning by two of the three neural networks.





Figures 1.5, 1.6: *Average Training & Test Accuracies over 50 runs per algorithm.*

Unique Optimization Problems

Beyond the neural network, each algorithm was applied to two more optimization problems in order to gauge their comparative functionalities. Across both problems, the length was set to (n = 50), and each algorithm was run 10 times in order to consistently test them using the same parameters. While each technique was given 500 iterations to optimize Four Peaks, 750 iterations were afforded under n-Queens due to the vast set of possible states. On top of this, the maximum number of attempts was set to a standard 500 for testability, and the restart/population size metrics were capped at 250. Finally, the execution of each algorithm was repeatedly timed in order to produce a lower-bound run-time for each application to both problems. More information on this run-time measurement can be found in the *main.py* script.

Four Peaks:

Unlike the optimal weight vector, the Four Peaks problem presents a discrete-state, discrete-output maximization function to be optimized by the algorithms. Theoretically, this condition should allow for the Genetic Algorithm to work well under the problem domain. As stated in the introduction, the initial belief was that Simulated Annealing would produce the best results in the most efficient manner, reducing run-time between itself and the other algorithms.

Four Peaks Lower Bound Runtimes

Hill Climb	Simulated Annealing	Genetic Algorithm
5.3 sec	0.036 sec	36.977 sec

Table 1.1: *Lowest calculated run-time of each algorithm for Four Peaks.*

Although it clocked in as the fastest to converge on this problem at under a second, Simulated Annealing produced the worst results in practice, producing a high fitness of 43. For reference, the optimal state fitness for Four Peaks where (n = 50) with a T of 1/5 is 89. On the contrary, the Genetic Algorithm and Hill Climbing techniques performed relatively well, reaching respective highs of 71 and 75. While no technique converged on the global optima in testing, this is likely explained by the low iteration count necessary to reduce run-time for data collection. However, the Simulated Annealing results are of some concern, and may be influenced by its previously mentioned low state consideration value.

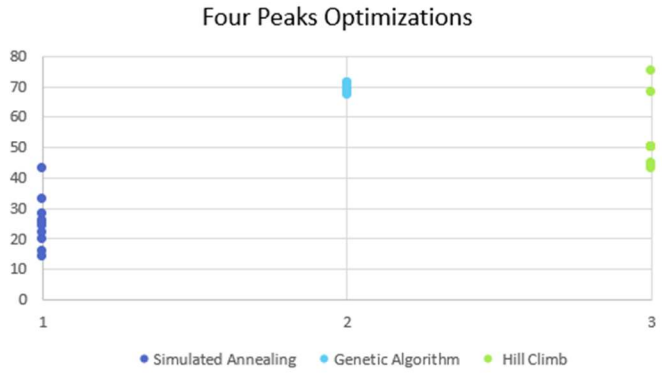


Figure 2.1: *Strip Plot of Four Peaks results over 10 runs.*

n-Queens:

Of the three problems, n-Queens produced results far different from the other two. Within this implementation, the theoretical chessboard is a 50x50 board, with 50 queens needing to be placed. However, before testing on this larger scale, small-scale tests were ran on single-digit values of n, with surprising results; each algorithm had significant difficulty in solving n-Queens, even when n was equal to values such as 4.

n-Queens Lower Bound Runtimes

Hill Climb	Simulated Annealing	Genetic Algorithm
37.08 sec	0.145 sec	65.349 sec

Table 1.2: *Lowest calculated run-time of each algorithm for n-Queens.*

While the run-time results are relatively the same as Four Peaks but on a larger scale, the Genetic Algorithm was the only technique which showed improvement over 10 runs of (n = 50). While Hill Climbing and Simulated Annealing got caught on local minima of 1225 (even following iteration raises), the Genetic Algorithm showed improvement ranging from a low fitness of 903 to slightly higher at 841. While there are a variety of factors in why this is, it is probable that it mainly has to do with *mlrose*'s implementations of these algorithms, and how they handle varying-state problems such as this one. While a safe assumption for the (n = 50) data would be that not enough iterations were afforded in order to explore the massive search space, the low n-value results lead me to believe that something else is causing these poor results, not just the large layout. Overall, the least data about the three algorithms was gained by comparing them against n-Queens.

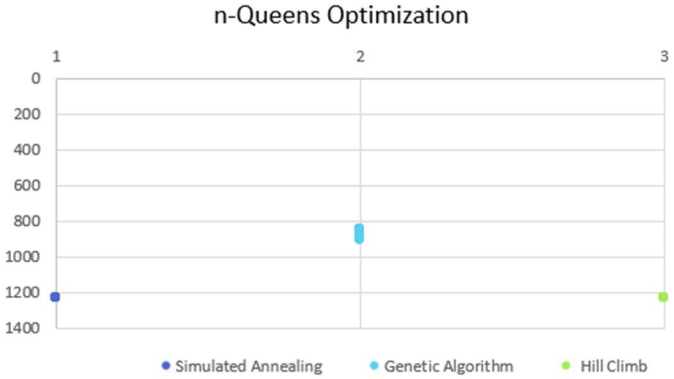


Figure 2.2: *Strip Plot of n-Queens results over 10 runs.*