Machine Learning Project 2: Randomized Optimization

Branden Kim
10/18/2023

Introduction

There are many different types of machine learning techniques that can be performed due to constraints of data in different scenarios or a difference in objectives. Specifically in this scenario, one of the key differences may come up when the data itself doesn't come with "true" output labels for your model to determine if a prediction is "correct". Randomized Optimization is one of the techniques that was developed to deal with scenarios where the data may not come with a "true" output value associated with the sample data. Randomized Optimization is not the only technique for these classes of problems, but this project aims to dive deeper into when and why Randomized Optimization works better in certain optimization problems and how some of the techniques utilized in these algorithms perform compared to its Supervised Learning's counterpart.

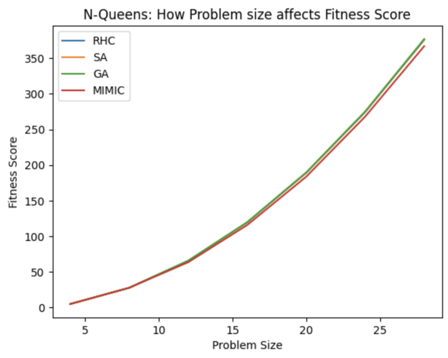
Optimization Problems

The purpose of Randomized Optimization is to find a specific input instance within the entire input space of the particular problem domain that "maximizes" or leads to the best output (assuming there is some fitness score that can map specific inputs to output values). Taking a more visual approach of an explanation, you can envision the input space on the X-axis and the fitness score on the Y-axis. In this graph, there will be some kind of function (depending on the problem domain / rules itself) that maps the specific input space values (X) to some fitness score that is quantitatively measurable (Y). Randomized Optimization is a suite of general techniques that find the specific input X that leads to some maximum value Y in the specific domain related to the problem. As there are many different types of functions, not one technique works optimally on all types of problems. Thus, I pose 3 different types of optimization problems (functions) that highlight how one Randomized Optimization technique may work better than another. Within each RO algorithm there are a specific number of hyperparameters that can be utilized to tune the performance of the RO algorithm. I performed a grid search to find the best combination of hyper parameters for each algorithm, but I wasn't able to run a full fitness score vs problem size run for each of the combinations of hyperparameters but I noticed that there wasn't any significant difference in the performance when tuning the hyper parameters other than the **max_iters** increasing the computation time which is to be expected.

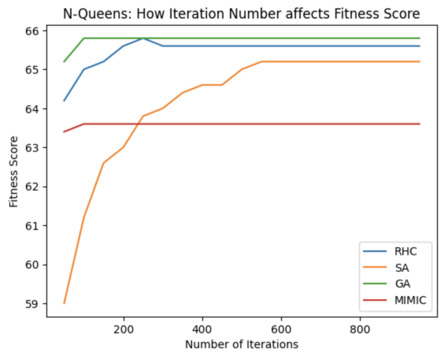
Simple Max / One Answer

The optimization problem that I am posing as "simple maximum / one answer" are the suite of problems that have some clearly defined "best" state or solution. In these problems, there is only one specific "winning" state or condition and in any specific scenario, there is always a move that can be made to move closer to that particular winning state or condition. One of the more popular examples is the OneMax problem where given a string of 0 and 1s, the score you receive is the number of 1s in the string.

In this problem, it is particularly obvious that there is only one “right” answer (all 1s) and there at any particular state, you can always make a move that leads to a better answer (switching a 0 to a 1). For these classes of problems I am calling (simple maximum) their fitness function will kind of look like a upside-down parabola as there is a clear optimal maximum with only one peak in the function. These are the simplest types of optimization problems and all the Randomized Optimization techniques should work very well with these problems, but there can be other factors that may allow certain techniques to outshine the others. Similar to the OneMax problem, I chose to first investigate the N-Queens optimization problem which falls into the class of “simple maximum” problems. In N-Queens you are trying to fit N queens in an N-N chess board such that none of the queens are attacking each other. It may not be as apparent as the OneMax problem, but within this problem there is always a method where you can make a move such that the N pairs of queens are attacking each other in a smaller amount. This means that similarly to the OneMax problem, there is always a move you can make to improve the fitness score. In these types of problems, all of the techniques maximum fitness score scale somewhat linearly with the problem size:

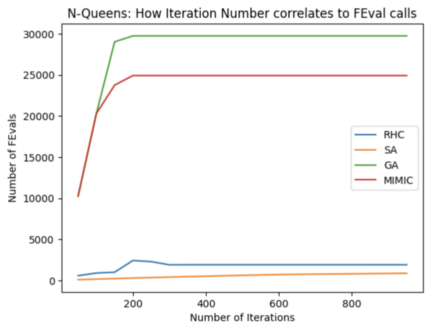


Looking at this curve, we can see that no matter what the problem size is (the problem gets more complex), the RO techniques are all able to find the “global maximum” score (the fitness score goes up as the scoring with more queens means more additions). However, the techniques differ in the way that they perform in terms of wall clock times or the number of iterations it takes to get the same fitness score. In the following graph, we fixed the problem size to 12 (complex enough to capture the relationships without running forever) and tried looking at if the number of iterations affected how quickly the technique got the “global maximum” fitness score.



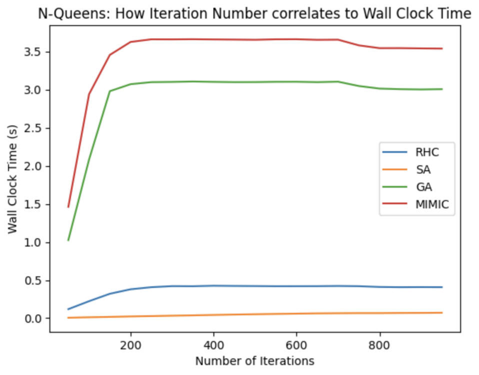
Looking at this curve we can see that the fitness score seems to level out around 300 iterations for a problem size of 12-Queens for most of the RO algorithms. It is interesting that GA and MIMIC flatline at a very small number of iterations which may suggest that it can be more efficient than that of RHC or SA as it will take a much smaller amount of iterations to converge, but we would also have to look at the number of FEval Calls per iteration. The N-Queens problem has some structure or pattern for the absolute

maximum fitness scores and maybe that is why GA and MIMIC is able to converge so quickly compared to that of the RHC and SA algorithms as it is able to pick up on that pattern through the nature of their algorithms.
All of the RO techniques have some fitness function that allows them to score a particular input example X to a fitness score. Calling this function is called a **Function Eval** or **FEval** and certain techniques can call this function more frequently than that of others. This is also another important consideration in our aim to find the “best” function as sometimes this fitness function is not a computationally trivial function. In the case of the N-Queens problem, the fitness function is a double for loop across N which means it scales with the input. If we did a 1M-Queens problem where the N is 1 Million, this would be a significant function evaluation as everytime the fitness function is called, it would be a 1M^2 instruction call. In this scenario we are seeing how the number of FEval calls changes with varying sizes of iterations run.



In this graph we can see that the number of FEvals is much higher in the GA and MIMIC algorithm compared to that of the RHC and SA algorithms. This may be because there are many more FEval calls built in within one particular iteration of the GA and MIMIC algorithms. In the case with MIMIC, in one particular iteration of a loop, there needs to be a FEval called made for each of the newly generated samples as opposed to one FEval call made per iteration for the RHC and SA algorithms. Seeing this graph, we can determine that it may be more efficient to utilize the RHC and SA algorithms and these algorithms may be a better bit for these classes of problems seeing how they achieve the same fitness score performance across all of the problem sizes and RHC and SA also utilize a significantly smaller amount of FEval calls.

Finally, we can see if the computation time of finding the global maximum fitness score for each algorithm changes with the number of iterations that we perform. This combined with the graph for FEvals can help us inform how expensive each fitness function call is in terms of pure computation time. Different RO techniques may scale differently in the number of FEval calls which means it can directly impact the computation running time of the algorithm.

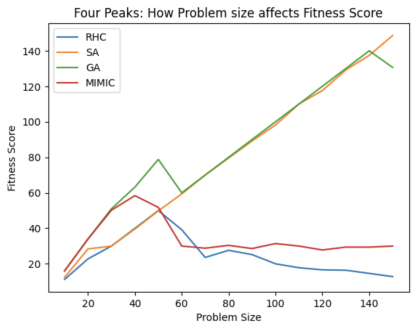


The graph above corresponds to the observations that we made on the graph for FEvals vs Iterations. It seems that the RHC and SA algorithms in terms of Wall Clock Time perform much better than the GA and

MIMIC algorithms. This makes intuitive sense knowing how the different RO algorithms work. For RHC and SA, their algorithm implementation is very simple and at each iteration only has at most 1-3 FEval calls as on each iteration the algorithm is comparing the fitness value with a constant number of neighbors. However, the GA and MIMIC algorithms have a bit of complexity in their algorithmic implementation as the GA algorithm has crossover steps which scale with the input size and the MIMIC algorithm has an expensive step to generate new samples from a distribution, calculate their fitness scores, and create a maximum spanning tree out of the new samples to form the distribution for the next iteration.
Overall, it seems for the "simple maximum / one answer problems, it is better to utilize RHC and SA as the RO algorithms to utilize as they perform just as well as the problem size scales but do so in a way more efficient manner computationally and in terms of FEval calls.

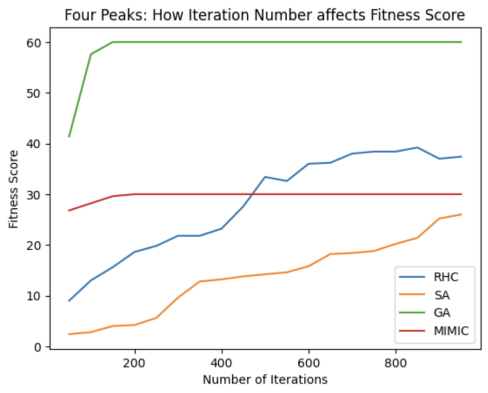
Simple Maximum / Multiple Peaks

Here is another class of optimization problems that I am calling "Simple Maximum / Multiple Peaks". Unlike the "simple maximum / one answer" class of questions, in these class of problems the fitness function has a global maximum or a peak that reaches the highest fitness score, but there are also areas within the fitness function that reach a peak that doesn't reach as high as the global maximum peak, we call these "local maximum peaks". In these class of problems there is the potential pitfall of running into scenarios where the RO technique gets "stuck" at one of these "local maximum" peaks without any method of coming out of it. One specific problem that falls within this class of problems is the **Four Peaks** problem which is basically a fitness function placed on a string of 0s and 1s where the score is dependent on the number of trailing 0s and leading 1s with some added bonus. The RO technique has no method of knowing that they aren't at the "global maximum" (a good analogy with life) and we will see that without the aid of sprinkling in some "randomness" it is hard to get out of these local peaks. Similar to the "simple maximum / one answer" class of problems, we are going to look at how the fitness score changes based on the problem size.

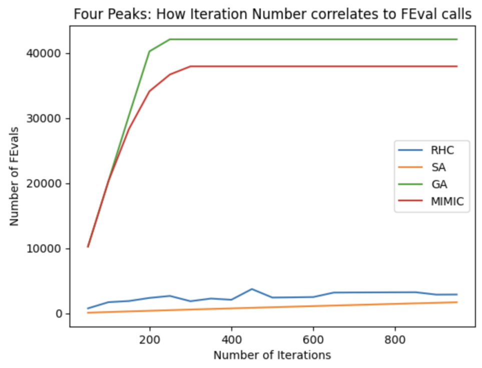


Here we can see that the SA and GA algorithm perform particularly well with increasing amounts of the problem size compared to that of RHC and MIMIC. It seems that around a bitstring length of 60 that the SA and GA algorithms converge and the RHC and MIMIC algorithms start to worsen in their performance. As I explained above, in the "simple maximum / multiple peaks" class of problems the fitness function often has a global maximum peak and several peaks with local maxima. An explanation of why the RHC and MIMIC algorithm perform worse with higher problem sizes is that as the problem size increases, the basin of attractions or ranges of input space X for the local maximum peaks increase and cover more of the input space. The RHC and MIMIC algorithms most of the time run into these basin of attractions and can't get out to the global maximum peak. For RHC, the random restart point most of the time runs into a basin with a local maxima so most of the time the maximum fitness score it outputs is the peak of that local maxima. For MIMIC, if the basin of attraction is very large for these local maxima compared to the global maxima, when sampling uniformly, there is a higher chance to retain samples that still correspond to the X values that are in the basins of the local maxima and it gets stuck there. Furthermore, in this bit string problem, there is not a particular structure or pattern to take advantage of to guide the estimated probability distributions.

Again let's take a look at the fitness function and how it changes with the number of iterations run. In this case, I ran the experiment with a problem size of 60 (bitstring of length 60) as in the above graph, it seems that there is a clear difference in performance of the RO algorithms around this size.

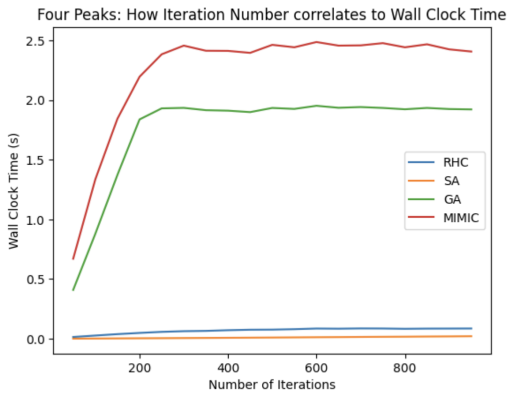


In this graph we can see that the GA and SA algorithms seem to be performing the best in terms of the fitness metric. The GA algorithm converged very early at a small number of iterations whereas the SA algorithm looks like it is increasing slowly and may need many more iterations to reach the global optimum fitness score. In contrast, the MIMIC and RHC algorithms seem to have converged and flattened out but have a fitness score lower than the global maximum which corresponds to the analysis mentioned above. The GA algorithm seems like it performs very well due to its affinity to the domain of the problem. As the underlying representation of the problem is bitstrings, it can easily modify the different parts of bitstring to create new combinations and keep the combination of bit strings that perform particularly well or maximize the fitness function and thus it can converge much quicker than that of the other algorithms. I would assume that the number of FEvals might change with a more complex function as the increase in complexity of function may lead to more retry attempts being needed to be made by the RO algorithms. Looping across the different iteration sizes at the problem size of 60, we can see the following graph:



In this graph we can see that similar to the "simple maximum / one answer" class of problems, the RHC and SA algorithms have a significantly smaller amount of FEval calls compared to that of the GA and MIMIC algorithms. Unfortunately, this is one downside of the GA algorithm as the large input size (length 60) with the amount of crossovers and mutations (the mlrose implementation performs uniform crossover) creates so many combinations that many FEval calls need to be made to select a subset of all the generated children for the next iteration.

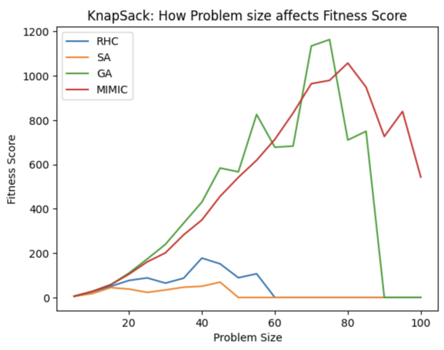
Finally, we can take a look at the Wall Clock Computation times for each RO algorithm with respect to different iteration sizes:



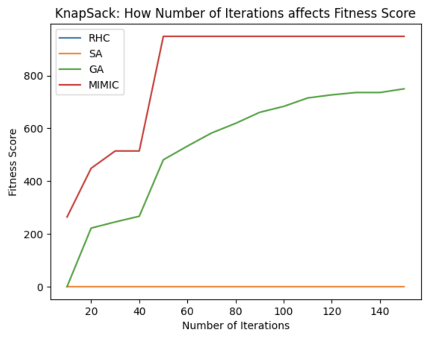
From the graph it seems that the MIMIC algorithm performs the worst and, as expected, the RHC and SA algorithms computation time doesn't scale with the number of iterations. It is interesting that the GA algorithm has a better wall clock performance even though it performs many more FEval calls than MIMIC. This might be because the GA algorithm creates much more crossover children that scales with input size and thus many more FEval calls need to be made to pick the most fit in the next generation, but each fitness score calculation itself is very inexpensive in the Four Peaks problem domain that the computation speed is very quick. However, the MIMIC algorithm has other computationally intensive processes such as creating the maximum spanning tree that the algorithm outside of the fitness function calls dominates the running time.
Overall it seems that the GA or SA algorithm is the best fit for this problem domain. The GA algorithm takes longer computationally but requires less training iterations whereas the SA algorithm performs very quickly, but requires a high number of training iterations to get the global maximum fitness score.

Structural Optimization Problems

In the final class of optimization problems, I am calling this class of problems "structural optimization problems" which include optimization problems that have some sort of special rule or ordering as constraints within the problem domain itself that make the fitness function unable to take the raw state input and calculate the output fitness score. There is some logical ruling that the fitness function and problem domain needs to keep track of. For example, the Traveling Salesman Problem (TSP) exhibits this behavior as there is a particular ordering / constraint that all of the nodes must be visited at most only once and not all nodes are connected to each other (implying there is a particular underlying structure). For this class of problems, I decided to look at the KnapSack problem. The KnapSack problem is a NP-H problem where there are N items that each have a specific weight and value. A person carrying these items has a maximum capacity that cannot be exceeded. The name of the game in this algorithm is that you want to find the combination of items to select such that you maximize the value of the items without going over capacity on the weights of the items. Similar to the TSP problem, Knapsack has a particular ordering to the weights and items that can form some sort of graph and thus has an underlying structure that can be exploited. In this case, I am not sure exactly what the fitness function curve would look like, but I am going to assume it is multi-dimensional with a bunch of local maximum peaks. Again we can look at how each of the RO algorithms performs with respect to the problem size.

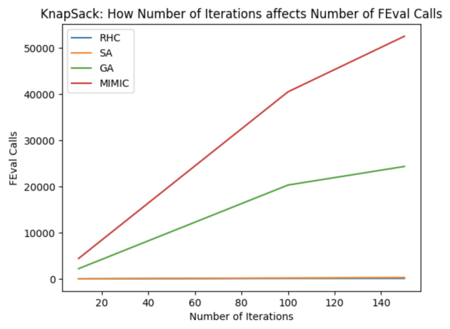


We can see here that the GA and MIMIC algorithm seems to perform the best across all of the problem sizes. There are several peculiarities here. The most notable one is that the RHC and SA algorithm eventually end up getting a fitness score of 0 as the problem size increases, however the GA and MIMIC algorithms are still able to get a maximized fitness score. This means that the RHC and SA algorithm are not able to find a valid solution to the KnapSack problem as the problem size increases. This would seem to indicate that the RHC and SA algorithms are falling into some local maxima basin in which there maximum is that there is no possible solution. Maybe what is happening is that as the problem size is increasing the basin that includes the global maximum is shrinking more and more and for randomized starting algorithms such as RHC and SA, it is unable to find this basin across the different randomized seeds that I had. This would also explain the behavior of why the GA and MIMIC algorithms eventually also start to taper off as well. When the problem size becomes so big, the likelihood of creating crossover children / mutations (GA) or picking samples from the probability distribution that have a better fitness function (MIMIC) is so low that the average overall fitness score across multiple runs starts to decrease. As expected, MIMIC seems to perform the best as it seems to learn the particular structure of the problem and performs at a high level.
Again we can see how the algorithm performs at a particular problem size () and see how the fitness scores scale with the number of iterations.



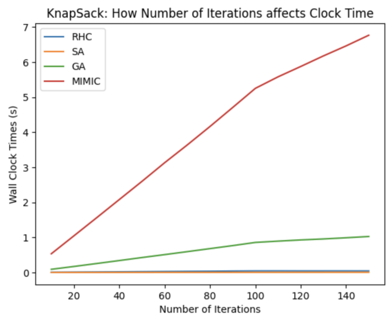
Here we can see that the fitness scores for only the GA and MIMIC algorithm go up with the higher number of iterations. We can see clearly here that the MIMIC algorithm quickly reaches the highest fitness score for the problem size in a small number of iterations. This is again possibly explained by the fact that the domain of the problem has some structure that MIMIC is exploiting / learning.

Another consideration to make is that maybe the underlying nature of the problem can lead to high number of FEval calls so we can take a look there too.



In the graph above we see that the MIMIC algorithm scales the greatest in terms of the number of FEval calls compared to the other RO algorithms. This is to be expected because as the iterations increase the number of evaluations for finding the samples for the next probability distribution increase. In this case, we are making a good tradeoff of high FEval calls for better accuracy.

Finally, let’s take a look at the Wall Clock Times for the different RO algorithms on this problem.



Similar to the FEvals, MIMIC has the highest running time out of all of the RO algorithms. For the KnapSack problem, it seems that we are making an active tradeoff between accuracy and performance in terms of computation time. It is interesting to see that only MIMIC which can pick up on the underlying structure in the problem can correctly find the global maxima in the KnapSack problem.

Neural Network Weight Optimization

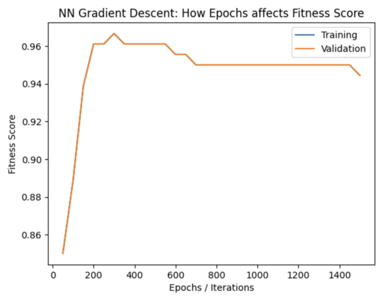
Next, we can take a look at how to utilize the RO algorithms to find the optimal network weights. In this section we aim to find a different method than calculus to tune and retrieve the optimal network weights for a neural network for the purposes of classification.

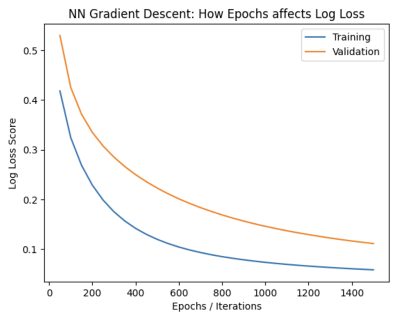
Dataset

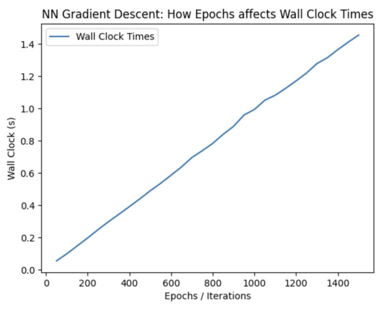
For the dataset, I decided to change up the dataset I utilized to utilize the IRIS dataset. One of the main reasons that I switched up the data set is because for some reason I couldn’t tune the Neural Network to the mlrose implementation using the datasets that I have utilized in A1. I followed the same instructions for preprocessing the data correctly and tuning the hyper parameters to ridiculous values, but it ended up not making a difference to the fitness curve and the fitness score in general as well. However, I had success with messing with some of the hyper parameters of the NN for the IRIS dataset and was able to achieve some good accuracy numbers by messing with the amount of hidden layers in the Neural Network.

Network Hyperparameters

From my personal tuning / testing, I found this list of hyperparameters to give a particularly high accuracy on the training, validation, and testing sets. For the hidden nodes, I found that a single layer of 28 hidden nodes with a learning rate of 0.00 and a clip_max of 5 seemed to give the best accuracy and best loss curve out of all of the algorithms. When training across 50 - 1500 iterations I retrieved the following training / loss curves.

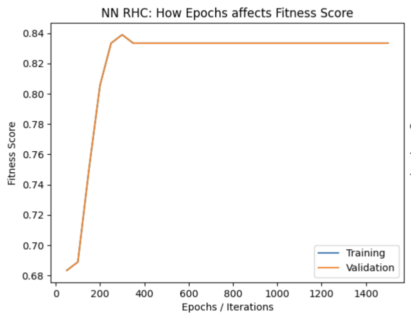


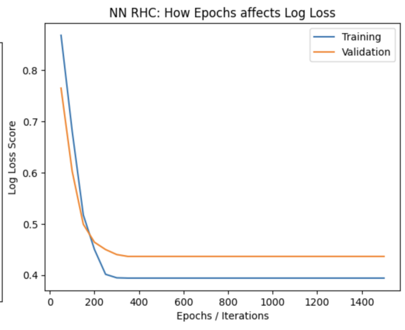


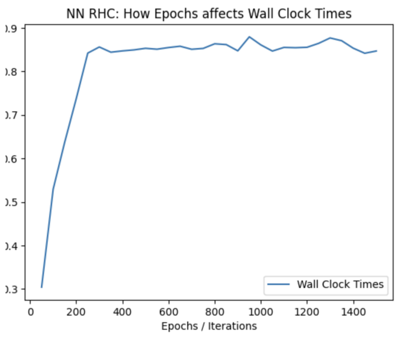


Let's compare this to the graphs we get from the other algorithms. One thing for the randomized algorithms is that I had to increase the learning rate significantly from 0.001 to 0.4 because if not it would keep getting stuck at local maxima.

Random Hill Climbing



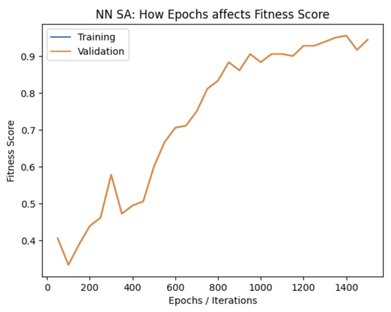


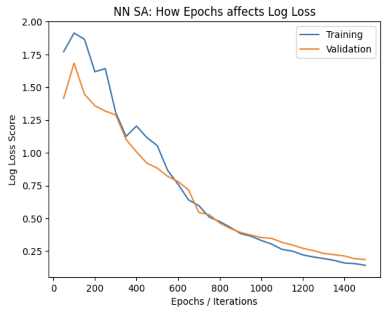


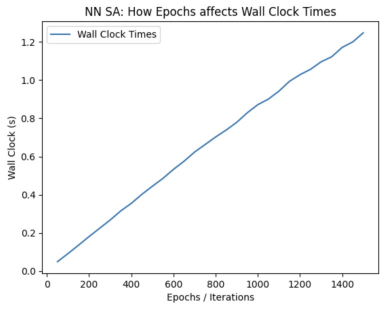
Here we can see the comparison of the Random Hill Climbing's performance with the Gradient Descent approach. In this case we see that the randomized hill climbing performs comparably in terms of accuracy and log loss as the gradient descent. The RHC algorithm also seems to converge at a much quicker iteration. One interesting thing is there is some local maximum within the dataset that the RHC algorithm falls in and thus I had to greatly increase the learning rate. Since the accuracies are comparable, let's look at the wall clock times. Interestingly, it seems that the wall clock time stays stagnant for RHC compared to GD. This means that RHC probably converges quicker to optimal weights and the algorithm performs early stopping since the convergence happens quicker. This is better since this means that the running time is not linearly scaling with the number of iterations made.

Simulated Annealing

Here we can see that there is some back and forth that goes on for the accuracy and log loss, but eventually with enough iterations it is able to find the optimal set of weights for the best fitness and log loss. SA had to utilize much more iterations to find the optimal set of weights. This had more peaks and valleys when trying to find the optimal set of weights but that is to be expected because it is an algorithm

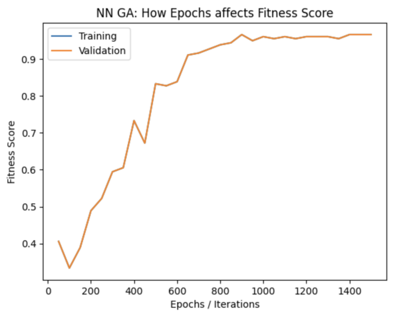


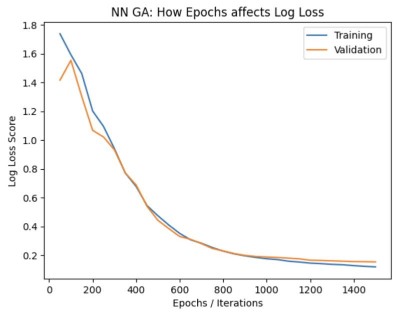


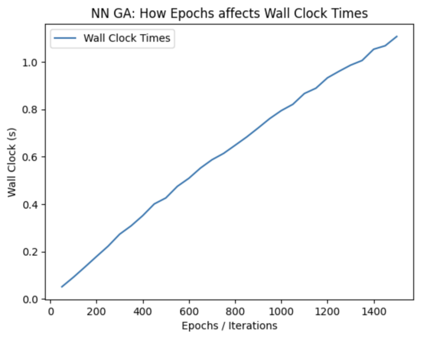


that tends to randomize directions across small local maxima. Furthermore, we can see that the running time scales linearly with the number of iterations this makes sense. This is the same as the Gradient Descent algorithm meaning that in terms of accuracy, the SA algorithm might have more optimal weights, but in terms of runtime, it performs computationally just as good as the GD algorithm. This is probably because the SA algorithm jumps around a lot of peaks and never really converges to any peak quickly and therefore there is no early stopping in the training. Since the algorithm doesn't stop early or converge early, the runtime scales linearly with the iterations.

Genetic Algorithms







In the GA algorithms, it seems to converge quicker to the optimal weights to get the best fitness in a smaller amount of iterations than that of the SA algorithm, but is slower than that of the RHC and GD algorithm. However the accuracy of the GA algorithm is the highest and the log loss curve looks good. Again the GA algorithm runtime scales linearly with the number of iterations. This is probably due the GA algorithm making the same amount of crossovers / mutations across all iterations as the crossovers it makes always introduces some amount of variance to the fitness that prevents early stopping from occurring as it never really "settles" at the true peak of the global maximum.

Conclusion

Overall with the NN weight optimizations, it seems that the RHC algorithm performs the best as it converges to the "global maximum" weights that give the best fitness in the least amount of iterations and it does so in a way that running time doesn't scale linearly with the number of iterations performed. However, this could be because the random seeds that I averaged over were just lucky to start/restart in the basin of the global maxima.