HW2: Randomized Optimization

Jiun-Yu Lee GTID: 903435223

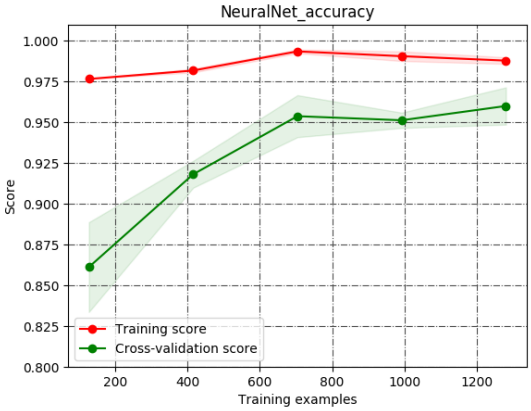
1 Randomized Optimization for Neural Network Weights

1.1 Recap HW1 - MNIST Classification

For the first part of this assignment, I will reuse the MNIST classification problem I created in the previous homework, that is, the binary classification between 4 and 9 hand-written digits. The dataset and the problem definition remain the same. However, the original feature space (784 dimensions) with the optimal network structure I found will result in around 52000 weights in the network, which is way to large for ABAGAIL to compute (Genetic Algorithm will spend over a day to compute 5000 iterations for this network). It becomes impossible to do the experiments with this structure, and we have to first reduce the feature. From the previous analysis, we already show that applying PCA before our model can reduce our feature dimension significantly without damaging the performance too much. Knowing that, we will apply the same PCA transformation to reduce the dimensions to 115 before we conduct the experiments.

In order to make a fair comparison, I rerun the HW1 experiment on neural network with the new dimension-reduced dataset. The following is the new optimal network structure I get after running grid search:

activation: logistic,
alpha: 1e-07,
hidden_layer_sizes: (16, 16),
learning_rate: adaptive
learning_rate_init: 1e-03,
max_iter: 1000,
solver": “adam



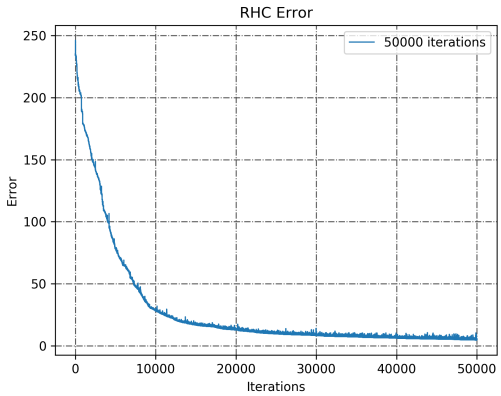
The right figure shows the learning curve of this structure. In fact, this curve is similar to the figure under the original (64, 32, 16) structure, on both the original and reduced dataset. Here, we can see a potential drawback of using grid search to find optimal hyper parameters. That is, it only considers the cross-validation score and ignores model complexity. According to Occam’s Razor, if two models perform similarly, we should choose the simpler one. As a result, we choose (16, 16) with logistic activation as our new network instead of the original (64, 32, 16) one.

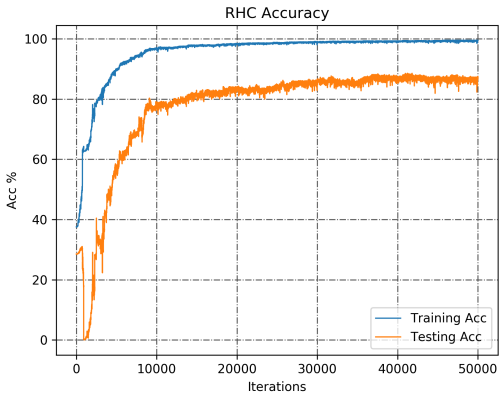
After obtaining the new feature space and optimal network structure, we are able to conduct experiments with respect to the three randomized optimizations. We will use three kinds of randomized optimization algorithm to update the weight of my new optimal neural network instead of backpropagation, compare and analyze the results and also with the one get from backpropagation. For the following experiment, all Error will refer to the sum of square error. All the randomized optimization implementation are based on ABAGAIL. Since ABAGAIL does not provide a cross-validation function and in order to avoid an excessive run time, we use a simple training-testing split with 0.2 ratios instead.

1.2 Randomized Hill Climbing (RHC)

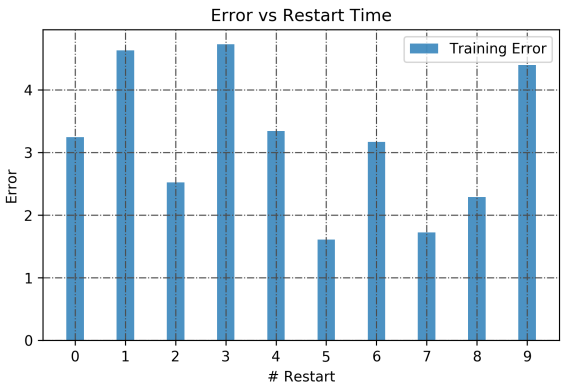
Randomized Hill Climbing is one of the simplest randomized optimization algorithms. It starts by picking one random point, calculating the fitness of its neighbors, update the point to the neighbors with the best fitness score if it is

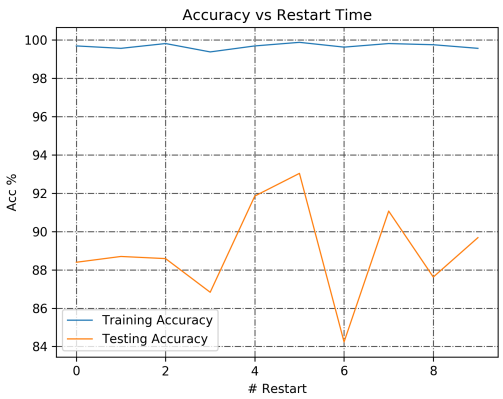
greater than the score of itself. Then it keeps doing that until there is no neighbor that has a better fitness score. To deal with the large and continuous weight space, ABAGAIL using a simple variation of the algorithm. Instead of calculating all the neighbors, it simply chooses a random dimension of the weight, adds a small value between [-0.5, 0.5] to form a neighbor, and evaluate the fitness of that point. If it has a greater fitness score, update the point to this neighbor.





First, we look at the error and accuracy of RHC with respect to the number of iterations. we can see clearly that RHC start to converge after 20000th iterations and end up with an error of 4.802. However, the accuracy figure shows that there is a gap between training and testing accuracy, indicating there is overfitting happening. We know that RHC is prone to get stuck in the local optima since it always follows the point with a lower error without exploring other paths. The possible reason for this result is that RHC gets stuck in a local optimum which makes it not able to generalize well to unseen data. For this reason, we experiment with multiple restarts to see whether it can mitigate this situation.





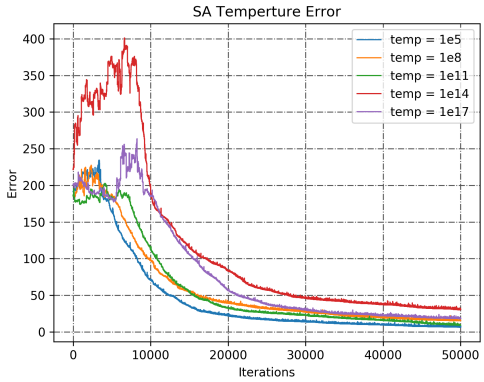
The improvement in the error is relatively small consider that the error in this problem can range up to 250. However, picking the restart number with the lowest error (#restart = 6) result in a big improvement in test accuracy (from 87% to 93%). As we can see, RHC with random restart has some ability to combat local optima.

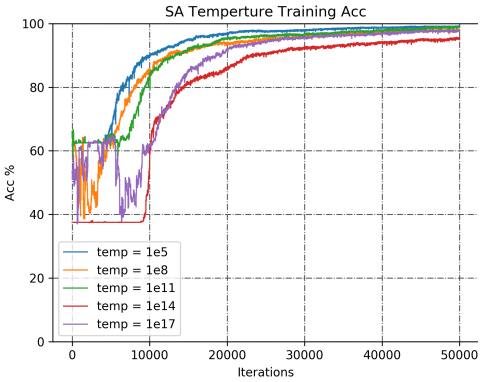
1.3 Simulated Annealing (SA)

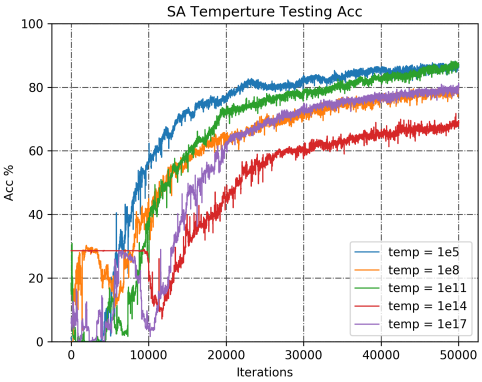
Simulated Annealing is a randomized optimization algorithm inspired by annealing in metallurgy. Unlike Randomized Hill Climbing, SA will start at a “high temperature”. In this state, SA is more willing to explore other neighbors instead of following only the one with the lowest fitness score, after exploring sufficient points, SA will gradually cool down and converge to an optimal answer. The algorithm is

much like a Randomized Hill Climbing. The only difference is that when you visit a neighbor with a lower fitness score, there is a probability with respect to T that SA will still update to that neighbor. The higher the temperature is, the more likely SA will do this operation. After each iteration, SA will multiply the temperature with a cool down rate so that the temperature decrease as we do more iterations. Eventually, SA will act just like a hill climbing algorithm.

Theoretically, SA is said to be better than RHC at avoiding getting stuck in the local optima, but it highly depends on how we choose the two hyperparameters of this algorithm, the starting temperature, and the cool down rate. For the following experiments, we first fixed the cool down rate to determine the optimal starting temperature, and then we fixed the starting temperature to that value to find the best cool down rate.



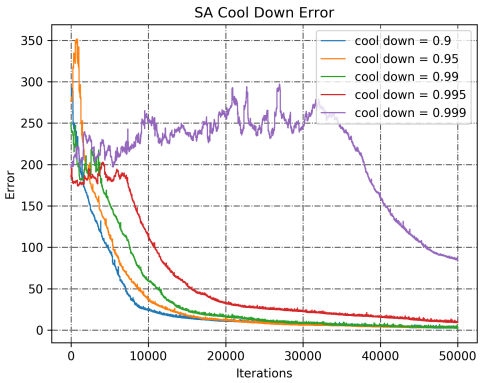


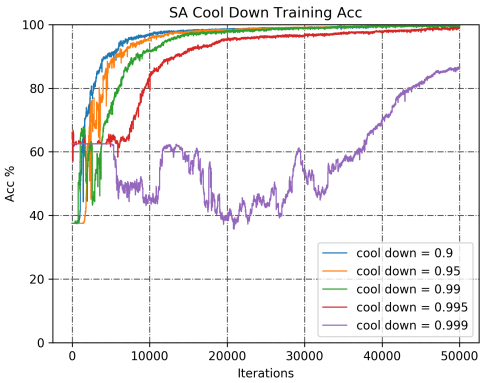


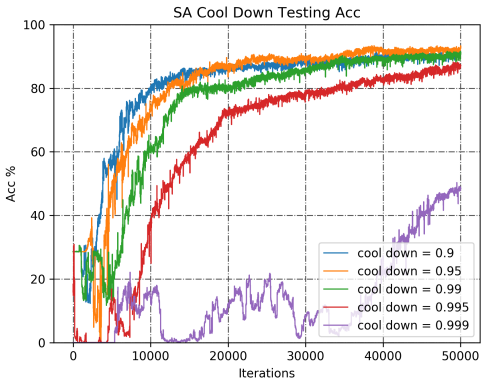
The figures show the result of five different temperature with cool down rate fixed at 0.995. We can see that there is a fluctuation at the beginning of the optimization. The higher the temperature is, the longer the fluctuation will be before it starts to converge. It indicates that higher starting temperature will give SA a longer time to explore the point, and have a higher chance to converge to the global optimal. However, using a higher starting temperature also means you will converge more slowly. In fact, we can also see that SA generally converge slower than RHC (RHC has an error lower than 30 in the 10000th iteration, while the error of SA with temp = 1e5 are still above 50).

There is another interesting finding in this result that is worth to discuss. When you look at the error line of temp = 1e8 and 1e11, you can see that 1e8 start to converge earlier than 1e11. However, at around 15000th iteration, the error of 1e11 becomes lower than 1e8. Eventually, 1e11 has a higher accuracy than 1e8 in both training and testing accuracy. This gives us an insight that using a high temperature and spending time exploring in the beginning do help SA to converge to a better optimal. For this result, 1e5 and 1e11 both give us a similar best performance. We choose 1e11 to be the optimal starting temperature to do the following experiment since it gives us a higher exploring ability.

After fixing the starting temperature to 1e11 and conduct the experiment with respect to cool down rate, we get the result at the top of the next page. Typically the effect of cool down rate is similar to starting temperature. Larger cool down will make SA explore longer and take more time to converge, but the effect is more significant compared with the starting temperature (SA with cool down rate of 0.999 is not even able to converge within 50000 iterations). In this result, we again see a phenomenon that the line with cool down rate of 0.95 starts to converge later than the 0.9 line, but surpasses 0.9 at around 15000th iteration, proving that allowing SA to explore in the beginning can help to prevent us from getting stuck in





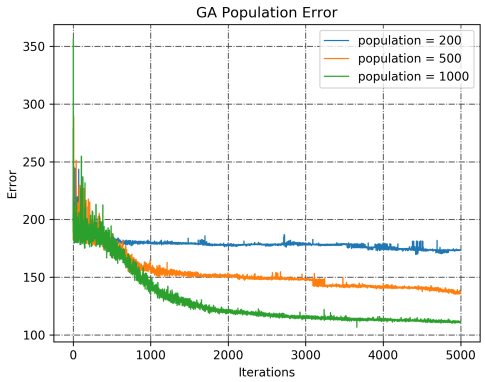


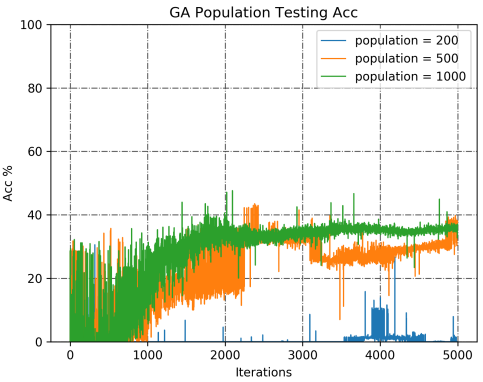
a local optimum. Although that 0.995 and 0.999 line are likely to perform better with more iteration and even beats 0.95, we still pick 0.95 here since it can give us the best result within a limited training time.

1.4 Genetic Algorithm (GA)

Genetic Algorithm, on the other hand, is not quite the same as Randomized Hill Climbing or Simulated Annealing. It is an algorithm consist of many operators that are inspired by the process of natural selection, such as crossover, mutation, and selection. Weights are considered to be an individual in a population, and the fitness score now represents how likely this individual will survive in the next generation. For each iteration, we select individuals with the best fitness score in the population and "mate" them together, crossover their weights (can be seen as gene here) to produce a potentially better individual. GA also allows us to "mutate" some of the individuals to increase the diversity of the population instead of keeping searching in a limited space of weights, which lead us to stuck in local optima.

To be more specific, we use uniform crossover here to create new individuals. For each mate, we will randomly select a weight from one of the individuals for each dimension to create their child. As for mutation, we randomly select one dimension in the individual and add a small number between [-0.5, 0.5]. There are three hyperparameters in this algorithm, population size, number of mates, and number of mutations. The following experiments will explore them one by one to determine the best parameters for this problem. Since one iteration of GA takes much longer to run compared to RHC and SA, we set the number of iterations to 5000 instead of 50000.

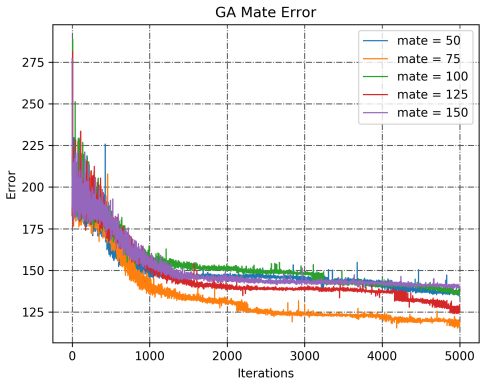


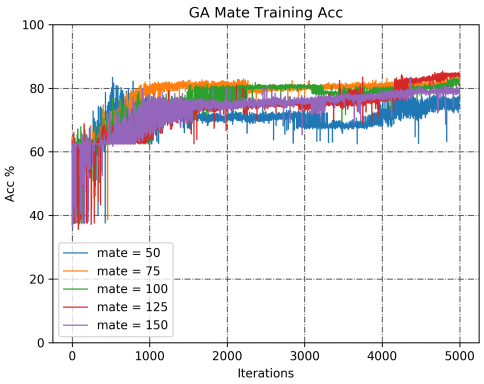


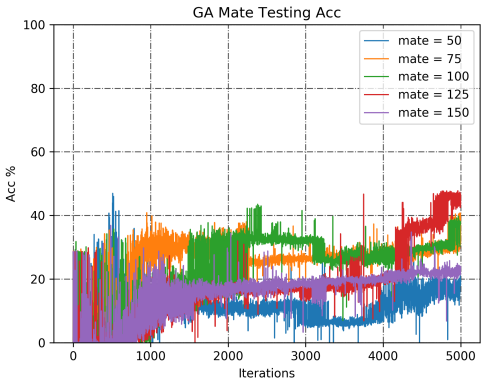
For the first experiment, we fixed the number of mates to 100, and the number of mutations to 10, which are the default values provides by ABAGAIL. The figures above show the result. It may not seems like a good result at first glance,

since they all converge at a high error, and none of them has a testing accuracy better than 40%. However, we should notice that the other two parameters are still default value, which may not suitable for this problem. A finding is that GA tends to converge more quickly than RHC and SA with respect to the iteration. Therefore, using 5000 iterations seems to be a reasonable choice.

The error figure shows that higher population gives us a lower error. In the training and testing accuracy, however, the difference between the size of 1000 and size of 500 is small. When choosing the size of the population, we don't want to have a very small size since it will decrease the diversity of the population and hence cause GA to stuck in the local optima. On the other hand, using a very large size of population is not very proper either. A large population will make it hard to narrow down to the global optima region, and therefore be much more difficult to search through the population and possibly slow down the training time. Since the train and test accuracy between size 1000 and 500 doesn't differ very much, we pick 500 as our population parameter for the following experiments.

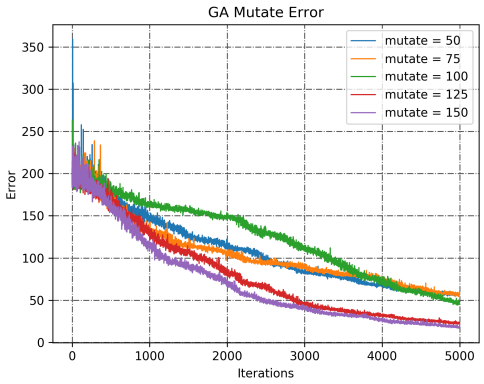


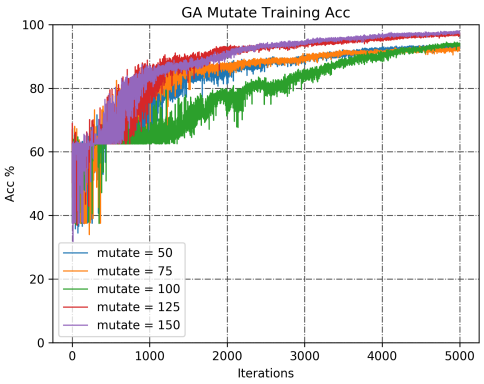


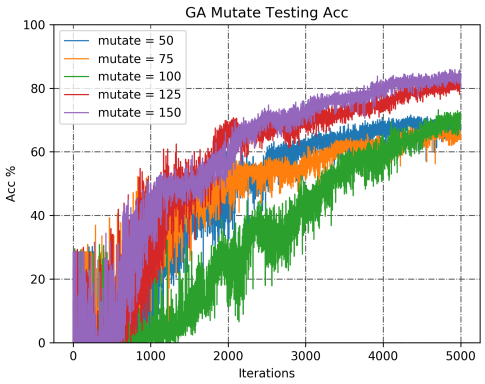


For the next experiment, we start to explore the effect of the mate parameter on Genetic Algorithm. Again, there is no significant improvement among the mate value that we have tried. This may due to that we are still using the default value 10 for mutation, which restricts the population to a small space and therefore is not even able to find an optimum. As we continue to the next experiment, we will see that changing the number of mutation does lead to some improvement.

Since mate 75 gives us the lowest error, we use this value for our final experiment.







In the final experiment we conducted to find the best value for the number of mutations, we can see clearly that increase the number of mutations does help us to search the weight space well. Mutation can let GA generate points outside our

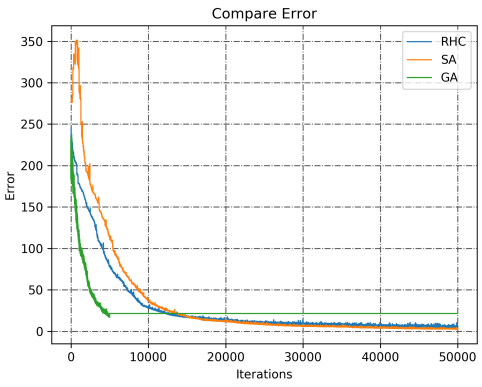
current weight space, covering a wider range so that GA can search the space more completely instead of searching in a restricted space. We can also see that as the number of mutations getting bigger, the error line goes down more steeply, indicating that GA tends to converge faster. However, one should know that crossover and mutation are costly operators that will make each iteration become much slower as the value increase. Therefore, we should not use very big values when we consider these two parameters. Since mute = 150 still has a reasonable training time, and it provides the best performance among all the parameter we have tried, we include this to our final parameter set.

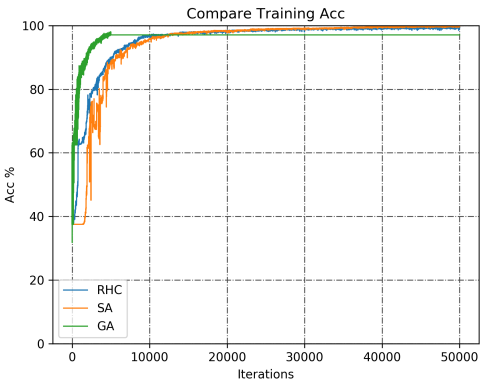
As we can tell in this result, the error line is not converged within 5000 iterations. There is still a downward trending in the error line and upward trending on both training and testing accuracy. Perhaps with more iterations, GA could perform better. However, one should notices that running the final parameter setting 5000 iterations already cost us around 3 hours. Spending another three hours to get one to two percents improvement on accuracy may not be cost-effective.

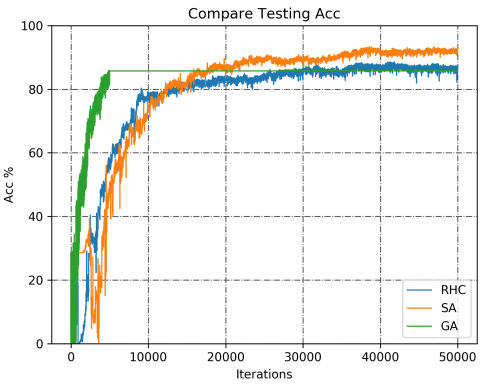
1.5 Conclusion

After discussing through the three optimization algorithms, we now present a comparison between each algorithm and backpropagation. The following table shows the optimal parameters and performance for each algorithm, and the figures show the comparison of each algorithm with respect to the number of iteration. Note that the table shows the RHC result with 6 restarts, while the figures show the result with only a single restart:

Algorithm	Optimal parameter	Train Acc	Test Acc	Training Time (Sec)
RHC	Iters = 50000, restart = 6	99.875	93.035	9842.764
SA	Iters = 50000, temp=1e11, cool = 0.95	99.750	92.822	2044.737
GA	Iters = 5000, pop = 500, mate = 75, mutate = 150	97.875	85.714	11071.954
Backprop	As listed in 1.1	98.875	**96.750**	**1.072**







As we can see in the figures, both RHC and SA converged to an optimum, while GA still has not reached a convergence. We can see clearly that GA converge faster (in terms of iterations) than the other two algorithms, and we can possibly get the best performance from GA if we give it more iterations to train. However, as said before, we should also consider training time while choosing the algorithm and parameter, and we already spent 3 hr to train this GA. Both SA and RHC have a gap between their training and testing line. There are two possible reasons for this

situation, one is that we are overfitting to the data, and the other is that we stuck in a local optimum and can not generalize well to the unseen data. I argue that it is more likely to be the latter since there are no turning around in the testing line. Overfitting will make models fit to the noise, and creating a downward trending on there testing line since the more iterations it trains, the more it will overfit. Here, however, the testing does not change much after several thousands of iterations, acting like it is getting stuck at somewhere and cannot be improved. We can see that RHC suffer more seriously from local optima, while SA eventually surpassed RHC and result in a better performance. Using random restart on RHC can help to mitigate this problem and end up with a slightly better result than SA. Nevertheless, the training time also grows significantly as the number of restarts increase.

To sum up, SA has a better performance than RHC, and can achieve nearly the same result of RHC with restart. Given more iterations, GA can potentially do better than SA. When it comes to combating local optima, SA is better than RHC, and we can determine from the trending of GA that it can potentially do better than SA. In terms of training time RHC without restart converges the fastest, following by SA, and then GA. Depend on the restart time, RHC can have a training time even longer than GA. However, none of these results can outperform backpropagation. backprop can achieve the best result in only 1 second. The difference at the training time may be influenced by the use of different language and library, but there is still a huge gap here. The explanation is that backpropagation makes use of gradient descent to find the direction that will lead to good optimum instead of randomly searching through the weight space. Moreover, it uses several tricks like adam optimizer and adaptive learning rate to prevent ending up at a local optimum. Therefore, it will give us better performance within a much smaller time. But it does not mean that randomized optimization is completely useless. In the next part, we will present three interesting problems that can actually be well approximated by these three algorithms.

2 Three Optimization Problem

1.1 Introduction

In the first part of the homework, we have applied three randomized optimization algorithms to find the optimal weight for the neural network we used in the MNIST problem we created in HW1. Now, we will further explore three interesting optimization problem to demonstrate the strength and weakness of each of the four algorithms (those three algorithms we experiment in the previous part plus MIMIC.) Specifically, we will analyze the performance of these algorithms on Traveling Salesman Problem, Four Peaks Optimization, and the Max-K-Coloring Problem. The first one will highlight the advantage of Genetic Algorithm, the second will highlight Simulated Annealing, and the third will highlight MIMIC. In order to make a complete analysis on each algorithm and reduce the length of the report, the rest of the section will be focused on the discussion of the result rather than the searching process for the best hyperparameter. We will use the strategy we used and the knowledge we gain in the previous part to determine the best hyperparameter before we discuss the result of each problem.

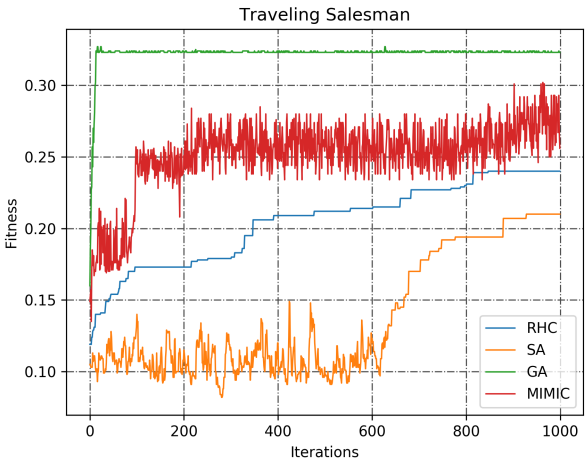
1.2 Traveling Salesman Problem

Traveling Salesman problem is a well-known NP-complete optimization problem in the field of Computer Science. The question is as follow: Given a graph where each node in the graph represents a city, and each of the edges represents the distance between two cities, what is the shortest possible distance for a salesman to travel each of the cities exactly once and end up in the city where he starts his travel. For such a complex NP-complete problem, there should exist many

local optimums in the searching space. We want to analyze the ability of tackling this problem for each of the four algorithms. Here, we use a randomized graph with 20 cities and the performance of each algorithm on this problem. The fitness function is the reciprocal of the minimum distance calculated. After finding the optimal parameter for each parameter by experiment, the result is shown as follow:

Algorithm	Optimal parameter	Fitness	Run Time (Sec)
RHC	Iters = 1000	0.239	0.030
SA	Iters = 1000, temp = 1e12, cool = 0.95	0.210	0.014
GA	**Iters = 1000, pop = 200, mate = 150, mutate = 20**	**0.323**	**0.248**
MIMIC	Iters = 1000, pop = 200, keep = 100	0.272	2.585

Here, we use iterations = 1000 for RHC and SA since we found out that they both converge to an optimum within 1000 iterations. After that, there will be no improvement made by RHC and SA. From this result, we can clearly see that GA achieves an outstanding performance compared with the other three algorithms. It converged in very few iterations and get the best fitness score. MIMIC comes in as the second place, with its fitness score fluctuating at around 0.25 and increasing at a slow pace. On the other hand, It seems like RHC and SA were getting stuck in the local optima since we found out that they are never improved after 1000th iteration.

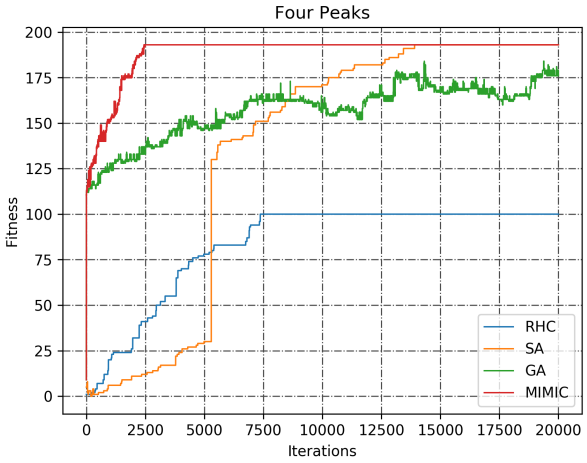


What may be the reason for this result? For a complex NP-complete problem like traveling salesman, there may be many local optimums across its entire searching space. It is almost certain that using an RHC here will make us stuck in one of them. Although SA has a stronger ability to fight against local optima, it only explores its neighbors instead of taking a big jump. If the global optima is very far away from the starting point and there exist many local optimums in the middle, we may not able to make it to the global optima before we cool down. However, the crossover operator allows GA to make a big jump and potentially cover a large enough space that includes the global optima. In conclusion, if the optimization problem is very complex and has many local optimum, using GA should be a better choice compared with other randomized optimization algorithms.

1.3 Four Peaks Optimization

Four Peaks Optimization is a relatively straightforward problem. This problem contains exactly four optimums, two local optimums with value of N (a string with entire 0 or 1) and two global optimums with value of 2N - T - 1(if both head and tail are larger than T, we will add additional N to the score and the maximum of max(head, tail) will be N - T - 1). Here, we choose N = 100 and T = 6, so that the global optimums are 193 and the local optimums are 100. This problem is interesting because it contains relatively few optima, and we know exactly what each of the value is. We then analyze which algorithm can get to the global optima quickly, and which one will still get stuck in the local optima even if this problem only have two of them. After finding the optimal parameter for each parameter by experiment, the result is shown as follow:

Algorithm	Optimal parameter	Fitness	Run Time (Sec)
RHC	Iters = 20000	100	0.114
SA	**Iters = 20000, temp = 1e11, cool = 0.95**	**193**	**0.069**
GA	Iters =20000, pop = 200, mate = 100, mutate = 50	176	2.096
MIMIC	Iters = 10000, pop = 200, keep = 20	193	59.443



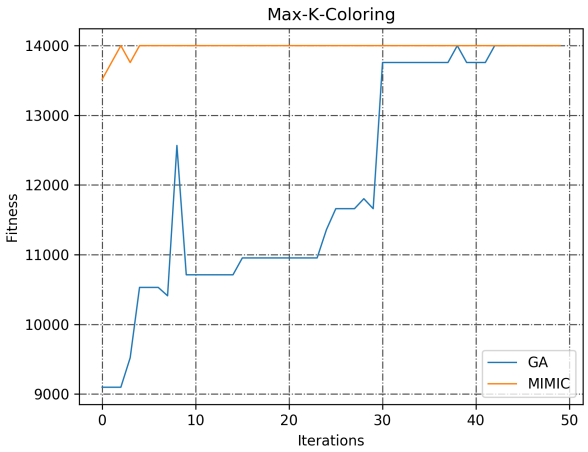
Although I use 10000 iterations to run the MIMIC algorithm, it only takes around 2500 to reach 193. Both SA and MIMIC reach the global optima 193. However, if we see run time, we will know that MIMIC takes about 15 secs to converge to 193, and a total of around 60 secs to finish 10000 iterations, while SA only takes 0.07 secs to finish all operations, which is way faster than MIMIC. For this reason, SA should be the best algorithm for this problem, following by MIMIC. The third place is GA, with the value in the halfway toward the global optima. We can see that compared to SA and MIMIC, GA grows at a slow pace and fluctuating. The last one is RHC, and we can clearly tell that it is getting stuck in the local optima since its value remains unchanged in 100.

From this result, we know that SA is very suitable for the problem with only a few optimums. The initial high temperature allows it to identify the region with global optima, and the following hill climbing allows it to quickly reach the peak. While MIMIC can model the actual probability distribution and eventually reach the global optima too, the training time is too slow compared with SA and make it not very efficient in this problem. We can see another potential disadvantage for GA is that it will take quite a long time to reach the exact global optima. It can quickly narrow down its population to cover the global optima, but since it doesn't have the hill climbing ability such as SA, it will spend a long time to reach the peak by only doing the random crossover and mutation. Finally, we can see a phenomenon that, even with a few local optimums (2 here), RHC can still get itself stuck in it. Perhaps the small basin of the global optima significant reduces the chance of RHC to get to it (We should have a least 7 consecutive 1's and 0's in both head and tail, which narrow down the space quite a lot.)

1.4 Max-K-Coloring Problem

Max-K-Coloring is another famous NP-complete problem in graph theory. The problem is defined as the following: Given a graph with N node with each node has L adjacent nodes, determine whether every node can be colored with one of the K colors and no node connected with another node with the same color. I thought this problem is interesting to analyze since each variable are strongly dependent on each other, and there is a clear structure lies in this problem. We want to determine which algorithm perform better in terms of whether it can find the solution and how quickly it can find it. The fitness is also based on this criteria. The graph setting we use here is as the following: N = 2000, L = 6, K = 8. After finding the optimal parameter for each parameter by experiment, the result is shown as follow:

Algorithm	Optimal parameter	Fitness	Run Time (Sec)
RHC	Iters = 20000	Unable to find	4.838
SA	Iters = 20000, temp = 1e12, cool = 0.95	Unable to find	4.977
GA	Iters =50, pop = 200, mate = 10, mutate = 60	14000	0.440
MIMIC	**Iters = 5, pop = 200, keep = 100**	**14000**	**0.374**



You will notice that I did not include a line for RHC and SA in the left figure of showing fitness score. This is because we are unable to find a solution based on RHC and SA, no matter how I tune the hyperparameter. Therefore, including those lines are not meaningful and can potentially interfere our analysis. For this reason, we only compare the performance between GA and MIMIC. Both of the two algorithms can find the answer and achieve the best fitness score. However, as we can see in the left graph, MIMIC reach the optima in only 5 iterations, whereas GA needs about 42 iterations to find the answer. We can also see that MIMIC also has a better run time compared with GA. Since MIMIC beats GA in iterations, run time, and convergence time, we conclude that MIMIC is more suitable for this problem.

Why MIMIC can outperform other algorithms while RHC and SA are not even able to find the answer. The reason is that we should take the “structure” of this problem into account. Both RHC and SA care only about the current point and completely ignore the structure of the problem itself. However, MIMIC learn a dependency tree to model the probability distribution of the variable directly, which is the reason that it can achieve a high fitness score in the very beginning and find the answer very quickly. Although GA does not model the structure directly, it can build the information by keeping updating its population for every generation. Therefore, GA is still able to find the answer, but much slower than MIMIC. As a result, when it comes to a problem with a clear and strong structure, MIMIC should be the algorithm we should use first.

1.5 Conclusion

In the previous sections, we conduct three experiments on three different optimization problem to highlight the strength and weakness of the four randomized optimization algorithms. It turns out that these algorithms are still useful for solving some NP-complete problems, despite that they are not able to compete with backpropagation when optimizing the weight of a neural network. In conclusion, GA can perform better if we want to optimize a problem with many local optimums distributed in the entire variable space. On the other hand, if we have a problem with only a few optimums value, SA can probably give you the global optimal value in the shortest among of time. When it comes to a problem with strong underly structure and dependence between each variable, MIMIC is probably the best choice since it directly models the structure using a dependency tree. As for RHC, we can see it is very prone to local optima. Even if a problem contains 2 local optimums can lead it to stuck. Random restart can help to mitigate this issue, as we discussed in part 1. However, we also see that it could slow down the algorithm dramatically. Perhaps the most suitable problem is the one with few optimum values and also a large basin for its global optimal.