CS 7641 Assignment 2 Report

Tan Cheng

Introduction

Four Randomized Optimization algorithms are explored in this assignment. 1) Random Hill Climbing (RHC) is a hill climbing approach. It finds the optima by exploring a solution space and moving in the direction such that the fitness score is increased per iteration. RHC randomly restarts its position to not get stuck at local optimums. 2) Simulated Annealing (SA) focuses more on the exploration of a solution space. It randomly chooses sub-optimal next steps with some probability. It leads to an eternal balance between exploration and exploitation. This increases the likelihood of finding global optima instead of getting stuck in local optima. 3) Genetic Algorithm (GA) is an evolutionary algorithm that produce new generations based on fitness of prior generations. It selects the most fit individuals, and replace individuals with poor fitness score via mutation, crossover etc. 4) Mutual Information Maximizing Input Clustering (MIMIC) is an RO approach that attempts to exploit the underlying "structure" of a problem and therefore might be a good choice for complicated problems. Dependency Trees are normally used to present the conditional probability information and sample future candidate hypothesis in the algorithm.

Part 1. Neural Network with Randomized Optimization Algorithms

In the first part of the assignment, we will use three randomized optimization algorithms - RHC, SA and GA to find good weights for a neural network. We will use them instead of backpropagation for the neural network that is used in assignment #1.

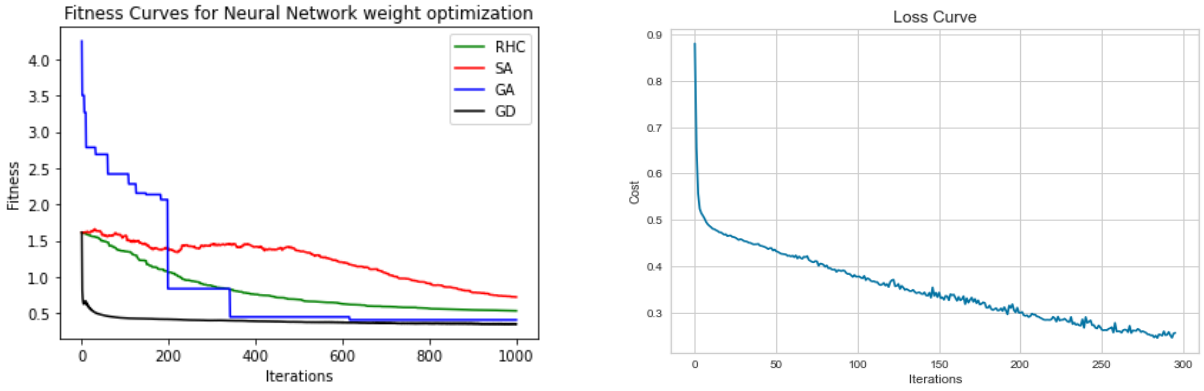
The Indian Liver Patient from assignment 1 will be used to investigate and compare the performance of the three randomized algorithms. The data set contains 416 liver patient records and 167 non-liver patient records. The column "is_patient" is a class label used to divide into groups - liver patient or not. Label "1" stands for positive cases (is patient), while label "2" stands for negative cases (not a patient). For convenience label "2" will be transformed to "0" to represent negative cases in data preprocessing. There are 10 attributes in total for this dataset. The data preprocessing is the same as assignment 1. The dataset is shuffled and split into training set and test set with a dividing ratio of 80%: 20%. And data standardization is performed using sklearn library. Besides, the balanced accuracy is used as the performance metric since the dataset is imbalanced. Balanced accuracy is defined as the average of recall obtained on each class such that overwhelming prediction accuracy of majority class can be eliminated. This keeps the performance metric the same as assignment 1.

MLROSE library is used for applying different randomized optimization algorithms to the neural network weight optimization. The number of hidden layer nodes of the neural network keeps the same as assignment 1 for all different algorithms, such that the neural network architecture is invariant for this study. In addition to the three randomized optimization algorithms

mentioned above, gradient descent algorithm (GD) is also performed using mlrose library in order to have a more fair comparison with the other three randomized optimization algorithms. The learning_rate parameter in the mlrose NeuralNetwork function (learning rate for gradient descent or step size for randomized optimization algorithms) is tuned for each algorithm. Besides, some other hyperparameters such as "restarts" for RHC, "schedule" function for SA, "pop_size" and "mutation_prob" for GA, are tuned until the best performance is reached.

After the hyperparameter tuning from the table below it is seen that each algorithm has its specific optimal learning rate or step size according to the selected dataset. The optimal step sizes for RHC and SA are about 0.1. GA has a smaller optimal step size of 0.01. Gradient descent (GD) algorithm is very sensitive to learning rate, and its optimal learning rate turns out to be 0.001 for this problem. GD results the highest training and test balanced accuracy, and it's actually running very fast, nearly as fast as RHC and SA. On the other hand, GA results the second highest balanced accuracy score, but its running time is much slower than the rest of algorithms. The results indicate that GA is not a very efficient algorithm to find out the optimal weight of neural network, it takes much longer time than other algorithms.

	Best Learnig Rate or Step Size	Training Balanced Accuracy	Test Balanced Accuracy	Fitting Time	Final Loss
Random Hill Climbing	0.1	56.50%	55.30%	2 sec	0.53
Simulated Annealing	0.1	53.40%	50.70%	2.56 sec	0.72
Genetic Algorithm	0.01	62.70%	59.60%	304.43 sec	0.41
Gradient Descent	0.001	85.20%	63.80%	3.41 sec	0.35



(a) (b)

Figure 1. (a) Fitness curves (loss curves) of RHC, SA, GA and GD algorithms applying to the neural network weight optimization with the Indian Liver Patient dataset. The curves are obtained by mlrose library. (b) The loss curve of neural network training that is copied from assignment 1. The curve is obtained by sklearn MLPC gradient descent algorithm.

The problem of fitting the parameters (or weights) of a machine learning model can also be viewed as a continuous-state optimization problem, where the loss function takes the role of the fitness function, and the goal is to minimize this function. So the fitness function is

equivalent to the loss function by default if use the mlrose library. Figure 1 shows the fitness curves for neural network weight optimization by applying RHC, SA, GA and GD. The plot on the right-hand side is copied from the loss curve that is obtained by sklearn MLPC library in assignment 1. The loss curve (fitness curve) of GD by using mlrose library has consistent trend with that from assignment 1. The losses are dramatically reduced in in the initial iterations for both curves, and converge fast after about 250 iterations. However, the randomized algorithms take much more iterations for the loss curve to converge. Therefore, the results show the advantages of gradient descent (GD) algorithm over the randomized algorithms (RHC, SA and GA) regarding the neural network weight optimization. In overall GD achieves higher prediction accuracy, competitive fitting time and takes less iteration to converge for this problem. In Contrast, GA is not priority choice for such problem because of the expensive computation cost.

Lastly the confusion matrix is obtained for each algorithm as shown in Figure 2. The confusion matrix tells detailed information of classification results for the test set. It shows the neural network model fitted by gradient descent (GD) algorithm can predict more true negative cases but at a cost of less true positive cases. We see the trade off in prediction accuracy between majority class and minority class. Even though one model could have higher balanced accuracy score like GD in this case, whether choose it or not still depends on how we want to distribute the weight on each class. And this may depend on applications in different real problems.

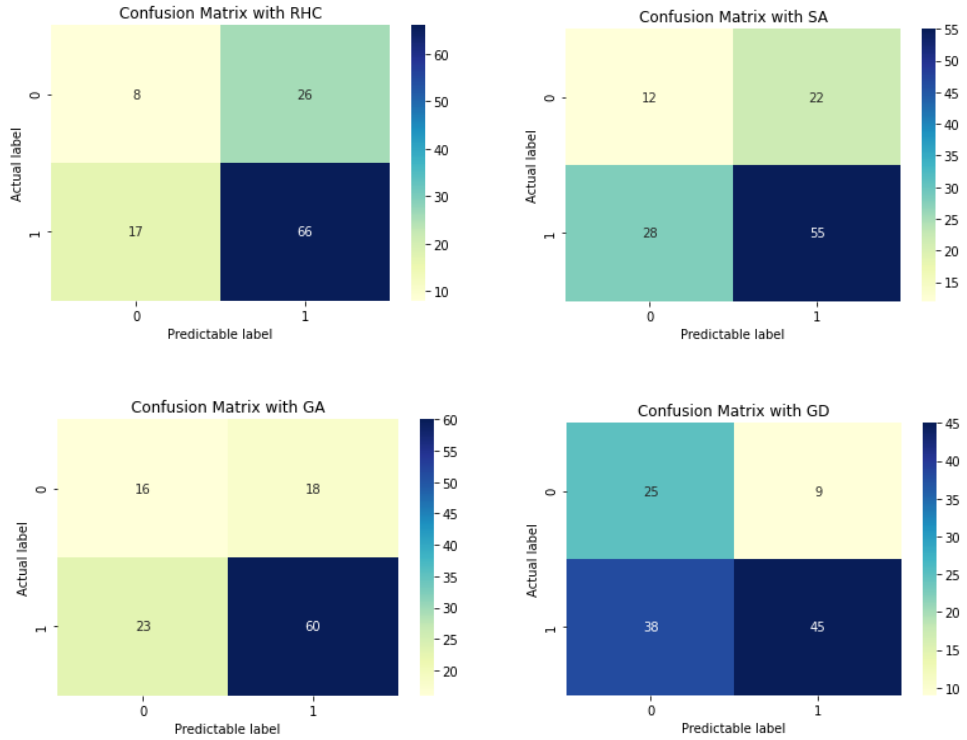


Figure 2. The confusion matrices of the test set that are evaluated by different neural network models. The weight optimization of the four neural network models are derived respectively from RHC, SA, GA and GD algorithms.

Part 2. RO Algorithms on Other Optimization Problems

In this part we will investigate the performance of different RO algorithms on three optimization problems. Specifically, Four Randomized algorithms - RHC, SA, GA, and MIMIC are applied to three optimization problems – Flip Flop, Four Peak and Continuous Peals respectively. These three classical problems are representative examples to show the advantages of different RO algorithms in different problems.

a. Four Peaks

In the Four Peaks Problem the fitness is simply defined as follows. It consists of counting the number of 0s at the start, and the number of 1s at the end and returning the maximum. If both the number of 0s and the number of 1s are above some threshold value T then the fitness function gets a bonus of reward. The name "four peaks" means that there are two small peaks where there are lots of 0s, or lots of 1s, and then there are two larger peaks, where the bonus is included. The two local optima with wide basins of attraction are designed to catch simulated annealing and random hill climbing. Genetic Algorithms are more likely to find these global optima than other methods.

The fitness function uses the default fitness function from mlrose library. The fitness function of an n-dimensional state vector x, given parameter T, is evaluated as:

$$Fitness(x, T) = \max(tail(0, x), head(1, x)) + R(x, T)$$

Where: tail (b,x) is the number of trailing b's in x; head(b,x) is the number of leading b's in x; R(x,T) = r , if tail(0, x) >T and head (1, x)> T ; R(x, T)=0, otherwise; T=t_pct * n.

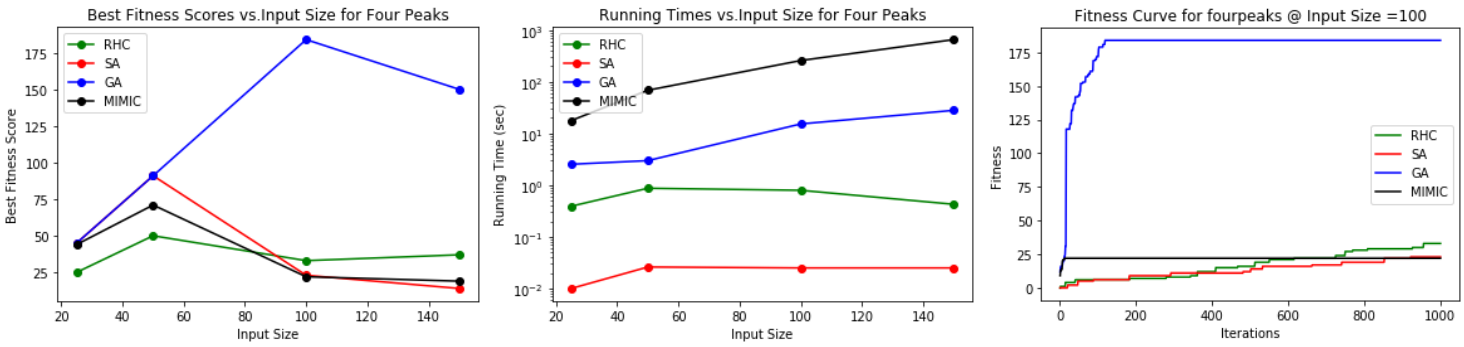
The objective is to maximize the fitness function as optimization problem. In all the experiments I choose t_pct = 0.15, Maximum number of attempts = 200 and Maximum number of iterations = 1000 as constant values. The input size varies from n= 25 up to n= 150, so that the performance of each algorithm can be analyzed as the problem complexity scales up. The below tables show part of the hyperparameter tuning results for each algorithm. It is seen that RHC is not very sensitive to the hyperparameter tuning in this problem. The fitness score of GA tends to increase as the population size increases. Also, the optimal mutation probability for GA and keep percentage for MIMIC tends to be around 0.2-0.4.

	Input Size N=25	Input Size N=50	Input Size N=100	Input Size N=150
	(Restarts, Fitness)	(Restarts, Fitness)	(Restarts, Fitness)	(Restarts, Fitness)
	((50,), 25.0)	((50,), 50.0)	((50,), 34.0)	((50,), 36.0)
RHC	((80,), 25.0)	((80,), 50.0)	((80,), 36.0)	((80,), 33.0)
	((100,), 25.0)	((100,), 50.0)	((100,), 36.0)	((100,), 36.0)
	((150,), 25.0)	((150,), 50.0)	((150,), 36.0)	((150,), 36.0)

	Input Size N=25	Input Size N=50	Input Size N=100	Input Size N=150
SA	((exp_const, min_temp), fitness)	((exp_const, min_temp), fitness)	((exp_const, min_temp), fitness)	((exp_const, min_temp), fitness)
	((0.0001, 0.001), 45.0)	((0.0001, 0.001), 38.0)	((0.0001, 0.001), 20.0)	((0.0001, 0.001), 20.0)
	((0.0001, 0.01), 40.0)	((0.0001, 0.01), 17.0)	((0.0001, 0.01), 0.0)	((0.0001, 0.01), 6.0)
	((0.0001, 0.1), 41.0)	((0.0001, 0.1), 18.0)	((0.0001, 0.1), 9.0)	((0.0001, 0.1), 9.0)
	((0.001, 0.001), 45.0)	((0.001, 0.001), 25.0)	((0.001, 0.001), 31.0)	((0.001, 0.001), 12.0)
	((0.001, 0.01), 45.0)	((0.001, 0.01), 38.0)	((0.001, 0.01), 11.0)	((0.001, 0.01), 8.0)
	((0.001, 0.1), 25.0)	((0.001, 0.1), 22.0)	((0.001, 0.1), 14.0)	((0.001, 0.1), 11.0)
	((0.01, 0.001), 45.0)	((0.01, 0.001), 35.0)	((0.01, 0.001), 29.0)	((0.01, 0.001), 14.0)
	((0.01, 0.01), 45.0)	((0.01, 0.01), 34.0)	((0.01, 0.01), 5.0)	((0.01, 0.01), 22.0)
	((0.01, 0.1), 25.0)	((0.01, 0.1), 50.0)	((0.01, 0.1), 20.0)	((0.01, 0.1), 11.0)
	((0.05, 0.001), 45.0)	((0.05, 0.001), 25.0)	((0.05, 0.001), 17.0)	((0.05, 0.001), 12.0)
	((0.05, 0.01), 25.0)	((0.05, 0.01), 44.0)	((0.05, 0.01), 20.0)	((0.05, 0.01), 24.0)
	((0.05, 0.1), 25.0)	((0.05, 0.1), 75.0)	((0.05, 0.1), 39.0)	((0.05, 0.1), 20.0)

	Input Size N=25	Input Size N=50	Input Size N=100	Input Size N=150
GA	((pop_size, mutation_prob), fitness)	((pop_size, mutation_prob), fitness)	((pop_size, mutation_prob), fitness)	((pop_size, mutation_prob), fitness)
	((200, 0.2), 45.0)	((200, 0.2), 91.0)	((200, 0.2), 100.0)	((200, 0.2), 150.0)
	((200, 0.4), 45.0)	((200, 0.4), 91.0)	((200, 0.4), 100.0)	((200, 0.4), 150.0)
	((400, 0.2), 45.0)	((400, 0.2), 91.0)	((400, 0.2), 100.0)	((400, 0.2), 150.0)
	((400, 0.4), 45.0)	((400, 0.4), 91.0)	((400, 0.4), 100.0)	((400, 0.4), 150.0)
	((600, 0.2), 45.0)	((600, 0.2), 91.0)	((600, 0.2), 184.0)	((600, 0.2), 276.0)
	((600, 0.4), 45.0)	((600, 0.4), 91.0)	((600, 0.4), 184.0)	((600, 0.4), 150.0)

	Input Size N=25	Input Size N=50	Input Size N=100	Input Size N=150
MIMC	((pop_size, keep_pct), fitness)	((pop_size, keep_pct), fitness)	((pop_size, keep_pct), fitness)	((pop_size, keep_pct), fitness)
	((200, 0.2), 45.0)	((200, 0.2), 73.0)	((200, 0.2), 25.0)	((200, 0.2), 18.0)
	((200, 0.4), 45.0)	((200, 0.4), 74.0)	((200, 0.4), 18.0)	((200, 0.4), 30.0)
	((200, 0.6), 45.0)	((200, 0.6), 15.0)	((200, 0.6), 16.0)	((200, 0.6), 17.0)



(a) (b) (c)

Figure 3. (a) Best fitness score vs. Input size for each randomized algorithm in Four Peaks problem. (b) Running time in log scale vs. Input size for each randomized algorithm in Four Peaks problem. (c) Fitness curve vs iterations for each randomized algorithm in Four Peaks problem at given input size n = 100.

Figure 3 shows GA dominates the performance in the four peaks problem. It has something to do with the nature of the problem. The crossover on a population of strings will occasionally create higher fitness individuals, for example receiving the extra bonus of reward. Then this individual has higher fitness than its parent. GA can find these high fitness individuals by combing important building blocks that are contained in those lower fitness individuals. Figure 3 (a) also shows GA algorithm work much better than the other three algorithms when the size of the problem scales up. Figure 3 (c) shows that GA does not need many iterations to converge. It

needs more iterations to converge than MIMIC, but much less iterations than RHC and SA. The only concern of GA is that it can be very slow compared to RHC and SA. But considering its obvious advantage in finding the highest fitness score, GA is still the best algorithm to consider in the Four Peaks problem.

MIMC converges using the fewest iterations, but the running time is much longer than the other algorithms. Notice that the running time is plotted in log scale. Since this problem does not benefit much from remembering the internal structure or underlying probability distribution, MIMIC is not preferred for this problem. SA running time is the fastest among all, but the fitness score dramatically decays when the input size increases, which indicates it suffers when input size scales up.

b. Flip Flop

Flip Flop is a problem that counts the number of times of bits alternation in a bit string. The fitness function uses the default fitness function from mlrose library, basically it looks to find the total number of consecutive bit alternations within a bit string. Formally, fitness is a state vector x as the total number of pairs of consecutive elements of x, (x_i and x_i+1), where x_i is not equal to x_i+1. In that sense, a string with maximum fitness bit would be the one that is composed of entirely alternating digits.

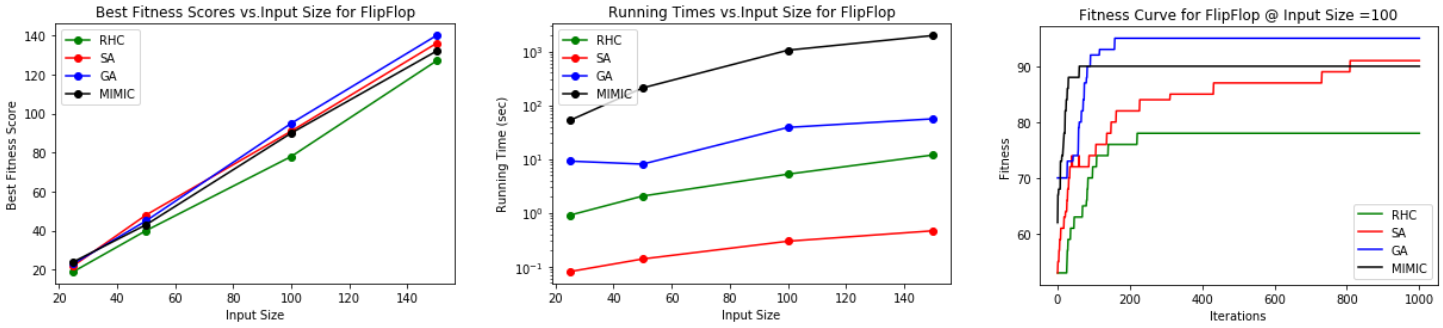
The objective is to maximize the fitness function as optimization problem. In all the experiments I choose Maximum number of attempts = 200 and Maximum number of iterations = 1000 as constant values. The decay schedule of SA algorithm uses exponentially decaying. The input size varies from 25 to 150, so that the performance of each algorithm can be analyzed as the problem complexity scales up. The below table shows part of the hyperparameter tuning results for each algorithm. It is seen that all the algorithms are not very sensitive to the hyperparameter tuning which is a bit surprising. It's probably because Flip Flop is a really a simple one, all the algorithms have no difficulty to achieve the maximum fitness value even within relatively loose hyperparameter ranges. The best fitness scores are quite close for all the algorithms for a given input size. From the SA hyperparameter tuning table, even though SA is not very sensitive to the hyperparameter tuning, exponential constant still has some impact on SA performance. Higher exponential constant is preferred since it results higher fitness score.

	Input Size N=25	Input Size N=50	Input Size N=100	Input Size N=150
	(Restarts, Fitness)	(Restarts, Fitness)	(Restarts, Fitness)	(Restarts, Fitness)
RHC	((50,), 25.0)	((50,), 40.0)	((50,), 78.0)	((50,), 127.0)
	((80,), 25.0)	((80,), 40.0)	((80,), 78.0)	((80,), 127.0)
	((100,), 25.0)	((100,), 40.0)	((100,), 78.0)	((100,), 127.0)
	((150,), 25.0)	((150,), 40.0)	((150,), 78.0)	((150,), 127.0)

	Input Size N=25	Input Size N=50	Input Size N=100	Input Size N=150
SA	((exp_const, min_temp), fitness)	((exp_const, min_temp), fitness)	((exp_const, min_temp), fitness)	((exp_const, min_temp), fitness)
	((0.0001, 0.001), 17.0)	((0.0001, 0.001), 37.0)	((0.0001, 0.001), 78.0)	((0.0001, 0.001), 110.0)
	((0.0001, 0.01), 17.0)	((0.0001, 0.01), 37.0)	((0.0001, 0.01), 78.0)	((0.0001, 0.01), 110.0)
	((0.0001, 0.1), 17.0)	((0.0001, 0.1), 37.0)	((0.0001, 0.1), 78.0)	((0.0001, 0.1), 110.0)
	((0.001, 0.001), 22.0)	((0.001, 0.001), 46.0)	((0.001, 0.001), 89.0)	((0.001, 0.001), 125.0)
	((0.001, 0.01), 22.0)	((0.001, 0.01), 46.0)	((0.001, 0.01), 89.0)	((0.001, 0.01), 125.0)
	((0.001, 0.1), 22.0)	((0.001, 0.1), 46.0)	((0.001, 0.1), 89.0)	((0.001, 0.1), 125.0)
	((0.01, 0.001), 23.0)	((0.01, 0.001), 49.0)	((0.01, 0.001), 92.0)	((0.01, 0.001), 135.0)
	((0.01, 0.01), 23.0)	((0.01, 0.01), 49.0)	((0.01, 0.01), 92.0)	((0.01, 0.01), 135.0)
	((0.01, 0.1), 23.0)	((0.01, 0.1), 49.0)	((0.01, 0.1), 92.0)	((0.01, 0.1), 135.0)

	Input Size N=25	Input Size N=50	Input Size N=100	Input Size N=150
GA	((pop_size, mutation_prob), fitness)	((pop_size, mutation_prob), fitness)	((pop_size, mutation_prob), fitness)	((pop_size, mutation_prob), fitness)
	((200, 0.2), 23.0)	((200, 0.2), 45.0)	((200, 0.2), 86.0)	((200, 0.2), 139.0)
	((200, 0.4), 23.0)	((200, 0.4), 47.0)	((200, 0.4), 93.0)	((200, 0.4), 138.0)
	((200, 0.6), 23.0)	((200, 0.6), 46.0)	((200, 0.6), 90.0)	((200, 0.6), 134.0)
	((400, 0.2), 23.0)	((400, 0.2), 46.0)	((400, 0.2), 91.0)	((400, 0.2), 136.0)
	((400, 0.4), 23.0)	((400, 0.4), 47.0)	((400, 0.4), 94.0)	((400, 0.4), 139.0)
	((400, 0.6), 24.0)	((400, 0.6), 45.0)	((400, 0.6), 92.0)	((400, 0.6), 140.0)

	Input Size N=25	Input Size N=50	Input Size N=100	Input Size N=150
MIMC	((pop_size, keep_pct), fitness)	((pop_size, keep_pct), fitness)	((pop_size, keep_pct), fitness)	((pop_size, keep_pct), fitness)
	((200, 0.2), 24.0)	((200, 0.2), 49.0)	((200, 0.2), 86.0)	((200, 0.2), 133.0)
	((200, 0.4), 22.0)	((200, 0.4), 45.0)	((200, 0.4), 87.0)	((200, 0.4), 128.0)
	((200, 0.6), 24.0)	((200, 0.6), 44.0)	((200, 0.6), 89.0)	((200, 0.6), 129.0)



(a) (b) (c)

Figure 4. (a) Best fitness score vs. Input size for each randomized algorithm in Flip Flop problem. (b) Running time in log scale vs. Input size for each randomized algorithm in Flip Flop problem. (c) Fitness curve vs iterations for each randomized algorithm in Flip Flop problem at given input size n = 100.

Figure 4 (a) shows that all the algorithms achieve very close maximum fitness value for a given size of problem. And all the algorithms work well when the problem size scales up. However, figure 4 (b) shows the running time are still significant different among the four algorithms. Running time can be another performance metric other than the best fitness score. In this sense SA is clearly the winner considering both the fitness score and the running time. Figure 4 (c) shows the fitness curves for all algorithms when the input size = 100. In this problem RHC and SA converge fast to a fairly high fitness score without using too many iterations. This is different

from the previous Four Peaks problem in which the SA and RHC seem to take forever to converge. This shows why SA and RHC are better choices than GA and MIMIC in simpler problems like Flip Flop.

Flip Flop is such a typical problem that shows advantage of simulated annealing. when the problem is not very complicated and evaluation functions are inexpensive to compute, SA can solve the optimization problem usually faster than the other RO algorithms, and achieve nearly optimal fitness score as well. In this problem RHC takes longer time than SA, which indicates it might be easier for RHC to get stuck at local maxima. In RHC when the algorithm restarts, the state restarts randomly in the search space instead of choosing some neighbors nearby. On the contrary, the SA algorithm picks a neighbor based on some probability distribution that is determined by the temperature value. When the temperature is very high, the algorithm has big chance to move to better neighbors, acting like Random Walk algorithm. But it also allows moving to some worse neighbors given a certain acceptance probability. As the temperature cools, the acceptance probability decreases such that the algorithm behaves more like RHC in the end.

It turns out GA and MIMIC are much more time consuming while the fitness scores don't show much improvement in this problem. They are not efficient choices for this kind of simple problem and can hardly beat SA in the Flip Flop problem.

c. Continuous Peaks

The Fitness function for Continuous Peaks optimization problem is defined as follows . For an n-dimensional state vector x, given parameter T, the fitness function is calculated as:

$$Fitness(x, T) = \max(max\_run(0, x), max\_run(1, x)) + R(x, T)$$

Where max_run(b,x) is the length of the maximum run of b's in x, R(x,T) = n if max_run(0,x) > T and max_run(1,x) > T. R(x, T) = 0, otherwise.

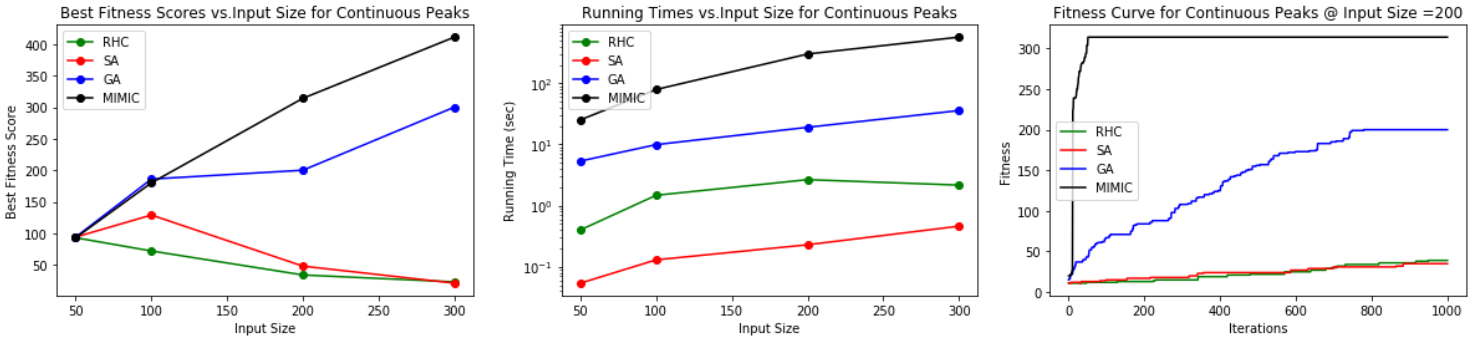
The objective is to maximize the fitness function as optimization problem. In all the experiments I choose t_pct = 0.15, Maximum number of attempts = 200 and Maximum number of iterations = 1000 as constant values. The input size varies from n= 50 up to n= 300, so that the performance of each algorithm can be analyzed as the problem complexity scales up. The below tables show part of the hyperparameter tuning results for each algorithm. MIMIC is shown to be very sensitive to the hyperparameter tuning in the Continuous Peaks problem, while RHC, SA and GA are not. And increasing the population size is necessary for MIMIC to keep up good performance when the problem size increases, for example the number of bits in a string. So the performance of MIMIC is highly relying on hyperparameter tuning. Enough population size should be guaranteed for the algorithm to work, even though it is at cost of running time.

	Input Size N=50	Input Size N=100	Input Size N=200	Input Size N=300
RHC	(Restarts, Fitness)	(Restarts, Fitness)	(Restarts, Fitness)	(Restarts, Fitness)
	((20,), 93.0)	((20,), 53.0)	((20,), 38.0)	((20,), 31.0)
	((60,), 93.0)	((60,), 54.0)	((60,), 38.0)	((60,), 32.0)
	((100,), 93.0)	((100,), 54.0)	((100,), 39.0)	((100,), 34.0)

	Input Size N=50	Input Size N=100	Input Size N=200	Input Size N=300
SA	((exp_const, min_temp), fitness)	((exp_const, min_temp), fitness)	((exp_const, min_temp), fitness)	((exp_const, min_temp), fitness)
	((0.0001, 0.001), 85.0)	((0.0001, 0.001), 126.0)	((0.0001, 0.001), 21.0)	((0.0001, 0.001), 31.0)
	((0.0001, 0.01), 85.0)	((0.0001, 0.01), 126.0)	((0.0001, 0.01), 21.0)	((0.0001, 0.01), 31.0)
	((0.0001, 0.1), 85.0)	((0.0001, 0.1), 126.0)	((0.0001, 0.1), 21.0)	((0.0001, 0.1), 31.0)
	((0.001, 0.001), 76.0)	((0.001, 0.001), 128.0)	((0.001, 0.001), 20.0)	((0.001, 0.001), 31.0)
	((0.001, 0.01), 76.0)	((0.001, 0.01), 128.0)	((0.001, 0.01), 20.0)	((0.001, 0.01), 31.0)
	((0.001, 0.1), 76.0)	((0.001, 0.1), 128.0)	((0.001, 0.1), 20.0)	((0.001, 0.1), 31.0)
	((0.01, 0.001), 94.0)	((0.01, 0.001), 52.0)	((0.01, 0.001), 35.0)	((0.01, 0.001), 31.0)
	((0.01, 0.01), 94.0)	((0.01, 0.01), 52.0)	((0.01, 0.01), 35.0)	((0.01, 0.01), 31.0)
	((0.01, 0.1), 94.0)	((0.01, 0.1), 52.0)	((0.01, 0.1), 35.0)	((0.01, 0.1), 31.0)
	((0.1, 0.001), 94.0)	((0.1, 0.001), 150.0)	((0.1, 0.001), 35.0)	((0.1, 0.001), 31.0)
	((0.1, 0.01), 94.0)	((0.1, 0.01), 150.0)	((0.1, 0.01), 35.0)	((0.1, 0.01), 31.0)
	((0.1, 0.1), 94.0)	((0.1, 0.1), 150.0)	((0.1, 0.1), 35.0)	((0.1, 0.1), 31.0)

	Input Size N=50	Input Size N=100	Input Size N=200	Input Size N=300
GA	((pop_size, mutation_prob), fitness)	((pop_size, mutation_prob), fitness)	((pop_size, mutation_prob), fitness)	((pop_size, mutation_prob), fitness)
	((100, 0.2), 94.0)	((100, 0.2), 189.0)	((100, 0.2), 200.0)	((100, 0.2), 196.0)
	((100, 0.4), 94.0)	((100, 0.4), 168.0)	((100, 0.4), 200.0)	((100, 0.4), 222.0)
	((200, 0.2), 94.0)	((200, 0.2), 189.0)	((200, 0.2), 200.0)	((200, 0.2), 208.0)
	((200, 0.4), 94.0)	((200, 0.4), 189.0)	((200, 0.4), 200.0)	((200, 0.4), 300.0)

	Input Size N=50	Input Size N=100	Input Size N=200	Input Size N=300
MIMIC	((pop_size, keep_pct), fitness)	((pop_size, keep_pct), fitness)	((pop_size, keep_pct), fitness)	((pop_size, keep_pct), fitness)
	((100, 0.2), 76.0)	((100, 0.2), 127.0)	((100, 0.2), 35.0)	((100, 0.2), 39.0)
	((100, 0.4), 76.0)	((100, 0.4), 136.0)	((100, 0.4), 49.0)	((100, 0.4), 52.0)
	((200, 0.2), 86.0)	((200, 0.2), 155.0)	((200, 0.2), 255.0)	((200, 0.2), 97.0)
	((200, 0.4), 90.0)	((200, 0.4), 178.0)	((200, 0.4), 248.0)	((200, 0.4), 43.0)
	((400, 0.2), 93.0)	((400, 0.2), 180.0)	((400, 0.2), 314.0)	((400, 0.2), 443.0)
	((400, 0.4), 91.0)	((400, 0.4), 179.0)	((400, 0.4), 286.0)	((400, 0.4), 46.0)



(a) (b) (c)

Figure 5. (a) Best fitness score vs. Input size for each randomized algorithm in Continuous Peaks problem. (b) Running time in log scale vs. Input size for each randomized algorithm in Continuous Peaks problem. (c) Fitness curve vs iterations for each randomized algorithm in Continuous Peaks problem at given input size n = 100.

Figure 5 (a) shows that MIMIC and GA clearly outperform RHC and SA. MIMIC achieves highest best fitness score meaning that it is the most effective algorithm to solve the Continuous Peaks

problem. When input size is small, GA performs equally well as MIMIC. When input size is large, GA performs slightly worse than MIMIC. However, RHC and SA don't work well especially when problem size scales up. Again, Figure 5 (b) shows MIMIC takes much longer running time than the other algorithms. Even though RHC and SA save a lot of running time, they cannot find the global optima effectively. The reason could be the Continuous Peaks problem contains more complex structure. It is not simple problem like finding peaks any more, the history information like what are the values of max_run(0, x) and max_run(1, x) that have seen so far is quite important in this problem. Knowing this information could be greatly helpful to find the global maxima. MIMIC has nature advantage in solving this problem since it learns the space, stores history information and underlying probability distribution.

Figure 5 (c) again shows that MIMC requires much less iteration to converge to the best fitness score, while RHC and SA need a lot more iterations to converge. This characteristic of MIMIC is very helpful if fitness function evaluation computational cost is high, MIMIC can reduce evaluations relative to the other algorithms.

Summary

From the results above, it shows depending on the nature, the complexity, and the structure of the problem, one algorithm may perform better than the others. For example, SA and RHC have a lot of similarity but also have obvious difference in the searching strategies. SA annealing seems to be a more intelligent way to do the search as the decay schedule and definition of temperature are used as the control factors. In all the problems above, SA runs much faster than the other algorithms. Especially for simpler problems like four peaks and flip flop, SA not only runs fast, but also achieves competitive fitness score. In summary SA is a superior randomized optimization problem for simple structured search problems where the fitness function estimation is inexpensive as well.

MIMIC and GA outperformed RHC and SA when the problems are more complex. The complexity involves the input size, the problem nature and problem structure. MIMIC and GA are more capable to handle large size problems. RHC and SA show some limitation when the problem size scales up as seen for example in the continuous peaks problem shown above. This is possibly because the search space increases significantly when problems scale up, which makes RHC and SA much more difficult to find the global optima. Another advantage of MIMIC and GA is that they normally require less iterations on fitness function call. This is important when the fitness function evaluation is expensive. GA works well in a wide range of optimization problems and is a very useful toolbox in machine learning problems. Comparing to GA, RHC and SA, MIMIC is suitable when there is structure information or relationship between different features that we need to care about. The drawback is that MIMIC normally take much longer running time than the other algorithms.

Finally, the results of neural network weight optimization study shows the advantages of gradient descent (GD) algorithm over the randomized algorithms (RHC, SA and GA) regarding the neural network weight optimization. In overall GD achieves higher prediction accuracy, competitive fitting time and takes less iteration to converge for this problem. In Contrast, GA is not priority choice for such problem because of the expensive computation cost.