Randomized Optimization Report

Zhihua Jin

zjin80@gatech.edu

1. Introduction to randomized optimization algorithms

In this project, I investigated the following randomized optimization algorithms:

Randomized hill climbing: RHC randomized goes to the next feasible solution with a higher gain (or lower cost) than the present solution.

Simulated annealing: SA monitored the temperature annealing procedure. It accept a new solution according to a probability of $\exp\left(-\frac{\Delta E}{T}\right)$. When the temperature is high, it is more possible to accept worse solution, in order to enable a broader search. As the procedure going on, the temperature becomes lower, and the algorithm convergences to an optimal.

Genetic algorithm: GA monitored the behavior of animals. At each iteration, it populates a group of solutions and chooses two best individual solutions from it. After that, it performs cross-over and generates a new solution.

MIMIC: MIMIC uses the mutual information and is able to directly estimate the probability of a solution. Besides, it can convey the structure.

I implemented them on continuous and discrete problems. The key components including two aspects: first, how to define an operation which generates a neighborhood solution? Secondly, how to fine-tune the hyper-parameters in each algorithm. In the following, I will discuss these things in detail.

2. Neural network optimization

a) Dataset

I reused the “Gender Recognition by Voice” dataset (Kaggle.com, 2019) I chose in Assignment 1, with 3168 samples, each of 20 features. It is a balanced dataset. The task is to predict whether the voice is from a woman or a man. I used 5-fold cross validation,which means I splitted the dataset into 5 folds. In each time, I used 4 folds for training and the remaining 1 for testing. The final performance is averaged over 5 time running results.

b) Description of experiments

I implemented a fully connected neural network (without any deep learning toolbox or packages, see nn.py, util.py for detail). It allows users to construct a network by adding hidden layers one by one as below:

```
nn = neural_network(x.shape[1], len(np.unique(y)))
nn.add_layer(50, ReLU)
nn.add_output_layer(activation=softmax)
```

In this problem, I used cross entropy as loss function.

Baseline: stochastic gradient descent (SGD) is used as baseline to train a neural network. In each iteration, it samples a batch of data, calculate gradients over them and apply weight updates. The learning curve, f1 score and training accuracy are as below. It could be seen it performs well.

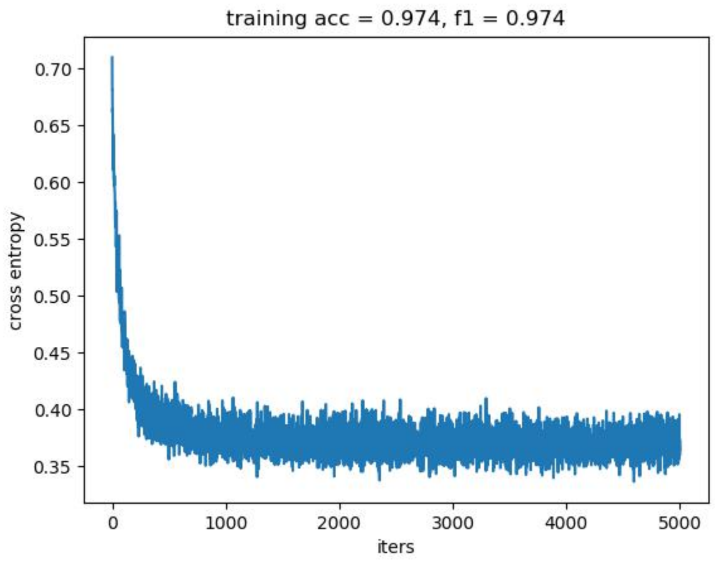

Fig. 1 cross entropy v.s. learning epochs of SGD and relevant parameters.

c) Apply randomized optimization to neural network

Randomized hill climbing: see learn_by_hill_climbing() method in nn.py.

I assumed that by "randomized" hill climbing, the assignment was asking for random selection of candidate weight updates (neighbors). At each time step, my network chose a set of gradients (instead of weights) that neighbors the best gradients (obtained by SGD), evaluates the network with the new set of weights. If the training loss decreases, the best set of weights is updated. All evaluations are done with respect to the entire training set.

In this task, I chose the neighbor gradients by adding Gaussian noise (with derivation σ) on it. σ is first set to be 0.01, and decrease with a factor 0.99 each time RHC fails to find a better solution with attempts of 10 times. Also σ is cast to be no smaller than 1e-6. I set the total iterations over the entire training set to be 500 times. However, I set the network to allow early stopping.

As a result, the learning curve (cross entropy v.s. iterations) of neural network with RHC , f1 score and training accuracy are as below:

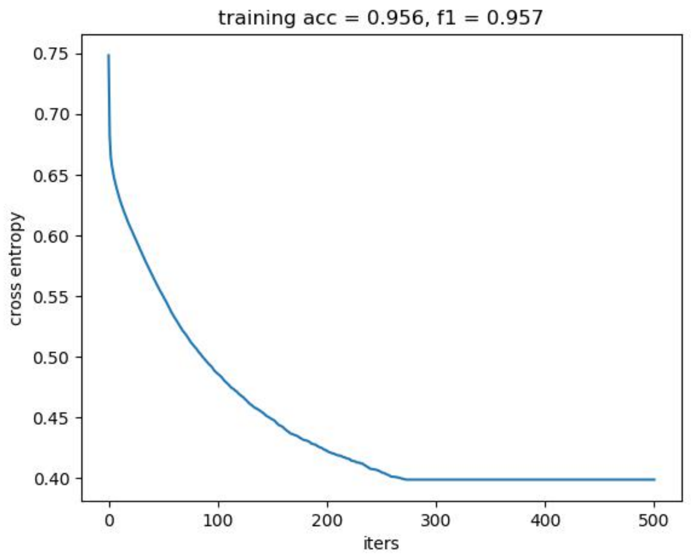

Fig. 2 cross entropy v.s. learning epochs of RHC and relevant parameters.

Simulated Annealing: see learn_by_simulated_annealing() method in nn.py.

Candidate weight increments at each step were determined according to a random vector (with normally distributed components). Unlike with RHC, where a gradient bias was necessary in order to achieve a good fit, SA with proper initial temperature and annealing rate is able to fit the training data well.

In this task, I first set the initial temperature to be 0.1, with a discount factor of 0.9. On a fix temperature, the algorithm search 100 times to find a better solution. After that, it anneals. When temperature reaches 1e-5, the algorithm stops. Again, I allowed early stopping. As a result, the learning curve (cross entropy v.s. iterations) of neural network with RHC , f1 score and training accuracy are as below:

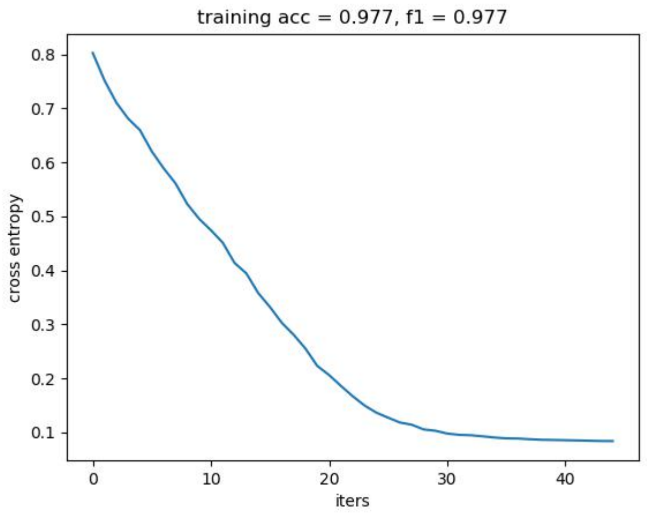

Fig. 3a cross entropy v.s. learning epochs of SA and relevant parameters.

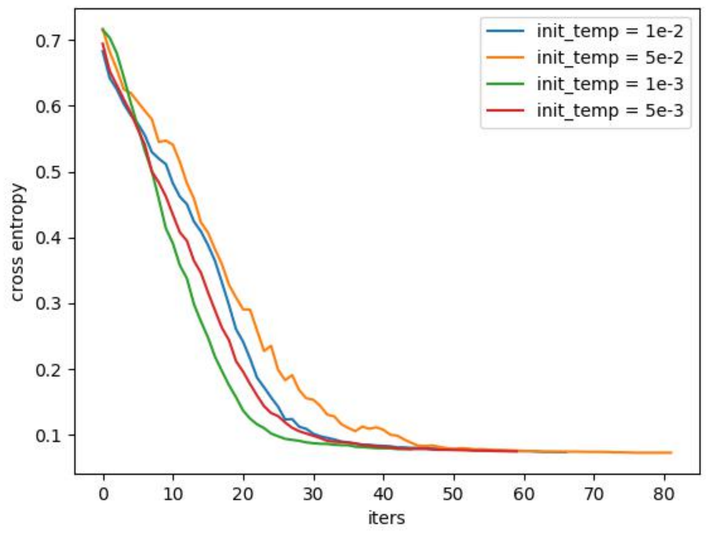

Fig. 3b cross entropy v.s. learning epochs of SA
with different initial temperature

According to Fig. 3b, we can see that, when the initial temperature is different, the loss will converge to the same value, while the convergence rate differs. But with too high or too low temperature, the convergence rate differs. We do not want the temperature to decrease too fast or too slow since the first one leads to similar situation in RHC and the latter one may mostly take uphill steps but few downward steps. So we must take care of the initial temperature carefully, a moderate temperature shall be fine.

Genetic Algorithm: see learn_by_genetic_algorithm() method in nn.py.

This GA-based optimization works by a “survival of the fittest” principle. At each iteration, I first populated a groups of weights by adding Gaussian noise onto the present weight. Then, I selected the two best weights with lower cross entropy. Next, a new solution is called their ‘son’ inherit their weights by doing average of them, this is the property brought by continuous optimization problem. At last, the new solution mutated by adding Gaussian noise, which enabled a broader search of parameter space.

In this task, I set the number of iterations to be 100. Each time, the group is of the size 100. And the Gaussian noise is with $\sigma = 0.1$, then decreased with a factor of 0.9 each time. The ratio of mutation is 0.1, i.e., only 10% percent of weights will be changed each time after inheritance.

As a result, the learning curve (cross entropy v.s. iterations) of neural network with

RHC , f1 score and training accuracy are as below:

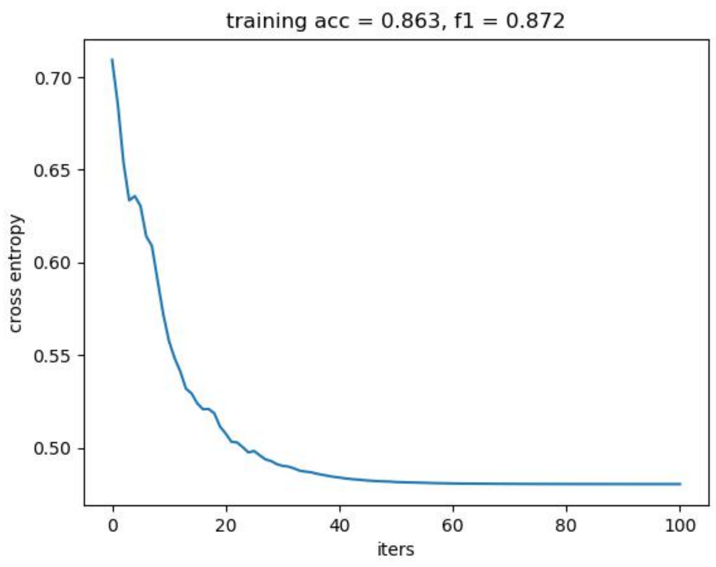

Fig. 4 cross entropy v.s. learning epochs of GA and relevant parameters.

d) **Results**

Execute voice_nn.py. The average of 5-fold accuracy and running time are listed below:

	acc	f1	auc	time/iter (s)	Converge epochs
RHC	96.46%	0.9647	0.9926	0.0193	273
SA	**97.13%**	**0.9713**	**0.9945**	0.0132	47
GA	86.14%	0.8700	0.8998	1.835	52

Note that although GA has a much higher computational cost, it spends most of time generating a group of solutions. While SA and RHC will try many iterations during one epoch, meanwhile, they spend less time per iteration.

It can be seen that SA can obtain the highest performance. I wished to improve the performance of these algorithms one by one, but it seems that RHC and GA are not the most suitable ones for optimizing neural network. I suggest the reason to be: *first,* neural network has analytical gradients when optimizing, so a random gradient is not leading to optimal results. Indeed, when sampling a batch of examples randomly to calculate the gradient, randomness is implied in it, so a neural network can explore the parameter space with SGD, not with RHC; *second,* the cross-over operation in GA is designed for discrete optimization problem, not for neural network. In this task, simulated annealing with proper parameters works best.

e) Summary

In continuous optimization problem like neural network, GA shows poorer performance. In the following, we will analyze that GA is more suitable for discrete problems. It is amazing that SA without any outer information performs better than the baseline SGD. It suggests that SA is good for this problem. RHC needs the gradient information to help convergence, so although it shows good performance, I think that its usage still needs further investigation.

3. Design problems for the randomized optimization algorithms

a) Problem 1: sum and parity

Question: Given a pre-defined length, output a bitstring with highest sum plus a parity bonus when the sum is even.

For first problem, I intentionally designed a problem that may work well using simulated annealing. So although it seem to be an easy problem, we may see the difference among these algorithms.
Execute sum_and_parity.py, it will output the following histograms, which records statistics of the best solutions after attempts of 50 times:

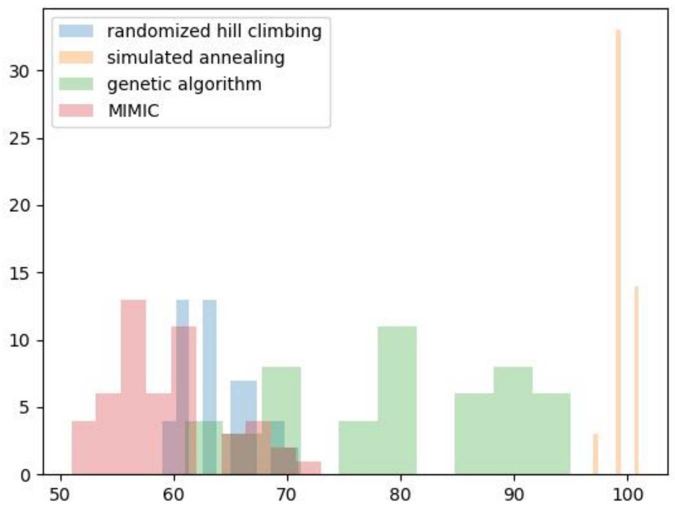

Fig. 5 histogram of best solutions found by the 4 randomized optimization algorithms.

The fitness of 50 testing times are listed as below, the metrics used include maximum, average and minimum solution theses algorithms output, the standard deviation and clock time used and time for each iteration.

	max	ave	min	std	time	time/iters(s)
RHC	71	63.80	59	3.15	0.3	0.000022

SA	101	99.44	97	1.08	1.1	0.000029
GA	95	79.76	61	9.98	0.8	0.000659
MIMIC	73	59.52	51	4.83	0.7	0.000498

With a length of 100, the optimal solution is a vector of all elements to be 1, reaching a goal of 101. We can see that SA can approximate this value very nearly all the times. While, as expected, RHC and MIMIC gets easily stuck in local maxima. GA works not bad in this problem, too. To conclude, the algorithms from best to worst in this problem is SA>GA>RHC>MIMIC.

b) **Problem 2: sum of product**

In this problem, I would like to maximize the product of the sum of consecutive ones in a bitstring. For example, fitness of [1,0,1,1] is 1*2=2, and fitness of [1,0,1,1,0,1,1,1] is 1*2*3=6.

Inspired by example provided in Machine learning lecture, the question is worth probing because genetic algorithm can inherit good parts from the 'father' and 'mother' solutions, I expect that it is suitable for this problem.

Execute sum_product.py, it will the following histograms, which statistics the best solutions of 50 times:

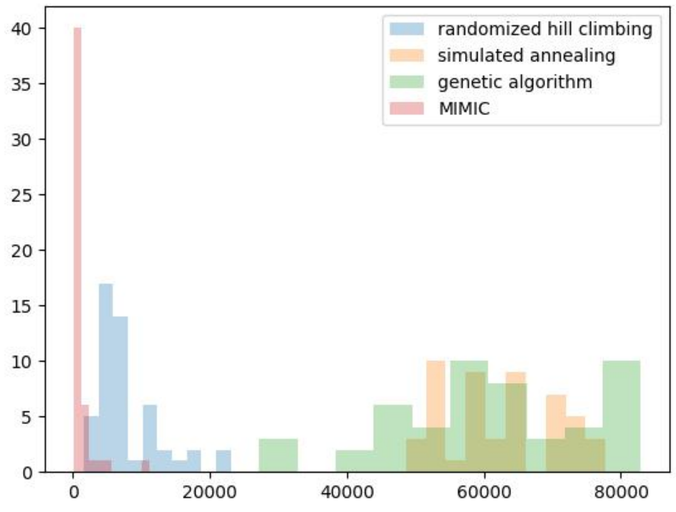

Fig. 6 histogram of best solutions found by the 4 randomized optimization algorithms

With a length of 50, I cannot solve the optimal solution manually. According to the experiments, we can see that GA can solve the best results frequently. While, as expected, RHC and MIMIC gets easily stuck in local maxima. SA works not bad in this problem, too.

The fitness of 50 testing times are listed as below:

	max	ave	min	std	time/iters(s)
RHC	23040	7734.16	1620	4795.68	0.000080
SA	77760	62094.60	48600	48600.00	0.000089
GA	**82944**	61526.88	27216	27216.00	0.003133
MIMIC	11088	834.06	39	1734.22	0.000266

To conclude, The algorithms from best to worst in this problem is GA≥SA>RHC>MIMIC.

c) **Problem 3: recover**.

In this problem, first, a vector is randomly generated according to uniform distribution U[-0.05, 0.05]. Then, I want to optimize another vector to obtain highest inner product with it.

Because MIMIC optimizes by estimating probability, so this problem is suitable for MIMIC. And also I expect GA to work well, since it can be applied to discrete optimization problem and achieve satisfactory results.

Execute recover.py, it will the following histograms, which statistics the best solutions of 50 times:

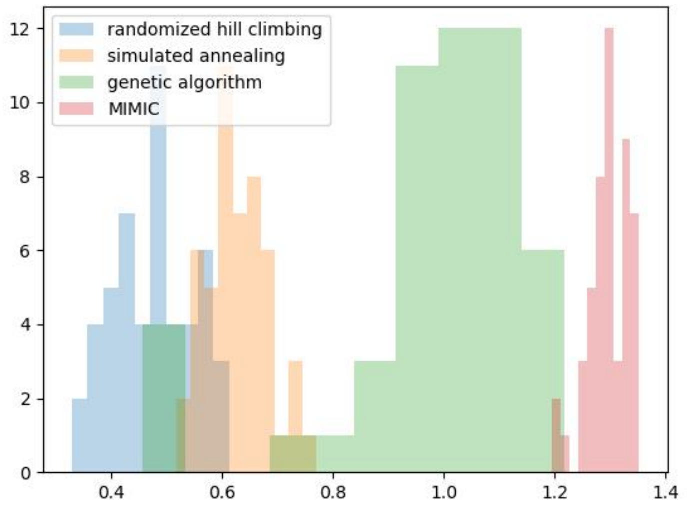

Fig. 6 histogram of best solutions found by the 4 randomized optimization algorithms.

When the vector is fixed, we can know the optimal solution is to place 1 at all the positions that the vector’s item is positive, and 0 for the others. So we can calculate the optimal solution. Here is the 50 times running results for a case, which of optimal fitness is 1.397. We can see that MIMIC reaches the optimal value, following by GA. So the algorithms from best to worst in this problem is GA⩾SA>RHC>MIMIC.

	max	ave	min	std	time/iters(s)
RHC	0.614	0.475	0.329	0.072	0.000016
SA	0.772	0.630	0.519	0.055	0.000019
GA	1.218	0.985	0.458	0.180	0.000276
MIMIC	1.353	1.296	1.195	0.035	0.000479

d) Summary

In this part, we studied three discrete optimization problem. They are all in bit-string form. We have noticed that MIMIC, SA and GA has its own advantages and disadvantages.

1) GA spends more time each iter, but can convergent in fewer iterations; GA is suitable for the problem that crossover operation will preserve optimality;

2) MIMIC is suitable for estimating probability. But in problem 3, all dimensions in the bitstring are independent, so the estimation is simpler than the common case. Using MIMIC costs too much computational resources;

3) SA also performs not bad in the three problem, and is best in the neural network. In conclusion, I suggest this is the best overall strategy for my problems.

References

1) Kaggle.com. (2019). *Default of Credit Card Clients Dataset.* [online] Available at: https://www.kaggle.com/uciml/default-of-credit-card-clients-dataset

2) Udacity.com. (2019). *Machine Learning | Udacity.* [online] Available at: https://www.udacity.com/course/machine-learning--ud262 [Accessed 13 Oct. 2019].