Report on Learning Search Space Partition for Black-box Optimization using Monte Carlo Tree Search

Xin Li
Department of Computer Science
University of California
San Diego, CA 92037
xil104@ucsd.edu

1 Summary and Review

1.1 Introduction

The problem that the original paper **Learning Search Space Partition for Black-box Optimization using Monte Carlo Tree Search** try to solve is fairly straightforward, it wants to find the global optimum of a deterministic black-box function which is Lipschitz continuous. This is called the Black-box Optimization problem.

Black-box optimization has many applications in various of topics, such as planning in robotics, hyper-parameter tuning of Neural Networks etc.

Without knowing the formula or structure of the function, gradient based methods are not viable, the only way to search for the optimum is by sampling. Sometimes simply random sampling or uniform sampling would be good enough, but for harder problems, it will take exponential time, and in those cases where the time cost or financial cost of a single black-box function call is very high, we want to reduce the number of function calls as much as possible. Basically the only reasonable track we can follow is to utilize or assume that the black-box function is Lipschitz continuous, that means we can potentially find small values near small values, and big values near big values.

Take the famous Bayesian Optimization (BO) method for example, it first "guesses" what the black-box function looks like based on the existing samples, by training a surrogate regressor(usually gaussian process regressor). And then, with the information of the trained regressor, the so called Acquisition Function propose new samples that are likely to get optimal values. Specifically, when propose, Acquisition Function will calculate the expected improvement of each sample and propose samples with big expected improvements.

The paper invents a technique called Latent Action Monte Carlo Tree Search (LA-MCTS), which basically utilize Monte Carlo Tree Search to search for a promising sub-area and sampling in that area using previous methods like Bayesian Optimization (BO)[1, 4] or TuRBO[3], so the authors call it a "meta-algorithm".

1.2 Related Work

The original paper does cover most of the popular methods that are used in black-box optimization, mainly BO related methods and tons of its variants, or Evolutionary Algorithms (EA)[5]. And after that, they also list out some MCTS papers, and explain its success in various of areas.

I'll say this part is generally objective, they are not trying to brag too much about their paper, they basically just list out the related works, and point out some limitations of the previous works, and those limitations are indeed there. For instance, BO tends to over explore the boundary area of the searching region due to the Euclidean geometry, especially when the dimension of the function gets larger.

1.3 Technical Results

The original paper provides their code in the link below, it is implemented in python:

`https://github.com/facebookresearch/LaMCTS`

Specifically, what they do in LA-MCTS is that every time LA-MCTS wants new samples, it builds a Partition tree based on the existing samples. For instance, if we're trying to find the maximum, it will

1. Create a root node that contains all the existing samples.

2. Do a 2-class Kmeans clustering on the containing samples, and take the cluster with a higher average value as the "good" cluster, and the other one is the "bad" cluster.

3. Train a SVM classifier to try to learn the "good" region and the "bad" region based on the labels that Kmeans creates.

4. Build a left child node contains the "good" samples, and a right child node contains the "bad" samples.

5. Do 2,3 and 4 recursively on the child nodes until there are less samples than some threshold.

and then, it will search the tree using this Upper Confidence Bound (UCB) score to balance exploration and exploitation:

$$ucb_j = \frac{v_j}{n_j} + 2C_p * \sqrt{2\log\left(n_p\right)/n_j}$$

that's what they put in the original paper, but I think there's actually some inconsistent symbols here, by $\frac{v_j}{n_j}$ what they actually mean is the average value of current samples according to the context here and the code they provide, however, as they defined earlier in Methodology, v_j already means $\frac{1}{n_j}\sum f\left(\mathbf{x}_j\right)$ which is the average.

C_p is a hyperparameter used to balance the scale of average function value and the degree of exploration. n_p is the number of samples the parent node of current node has, and n_j is the number of samples the current node contains. Again, they're actually not using this logarithm in their code, instead, they use $2C_p * \sqrt{2\sqrt{(n_p)}/n_j}$ for whatever reason, maybe they think square root is already a good approximation to natural logarithm.

So, it searches the tree using this UCB score, and gets a leaf node representing a promising region define by all the SVM boundaries along the searching path. Finally, it propose new samples in the region using previous methods. At least that's the idea, in the real implementation however, it is hard to just "propose samples in that region" directly, given the fact that it is hard to define this region constrained by a bunch of SVM boundaries. So I think for this reason, the authors first propose random samples using sobol sequence[8], expand the samples to make sure certain ratio of samples are out of the region, and then reject those that are out of boundary instead. At last they calculate expected improvement for the rest of the samples, and propose those with big expected improvements.

Next time LA-MCTS wants new samples, it rebuild a entire new tree based on the expanded existing samples, and search for the next promising region.

1.4 Review

Generally speaking, this is a simple idea but it does introduce an improvement of the searching performance. As a fact, two teams in The Black-Box Optimization challenge (NeurIPS 2020), JetBrains[7] and KAIST[6] won the 3rd place and 8th place, using their independent re-implemented versions of LA-MCTS.

So, in my review, this work is indeed practical, however, there's some inconsistencies of symbols in the paper, besides the UCB score that is mentioned in Technical Results, the paper also mix up

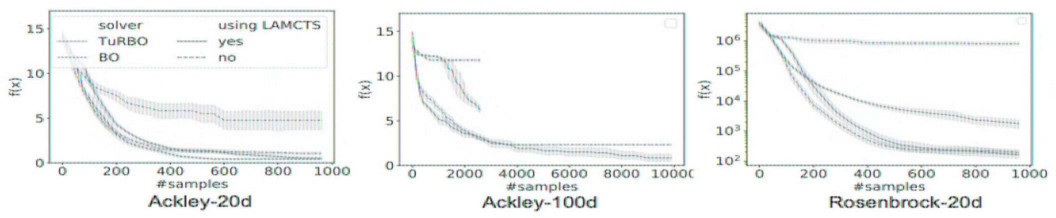


Figure 1: LA-MCTS improvement

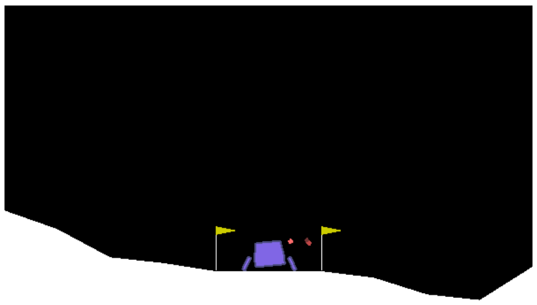


Figure 2: Lunarlanding

with maximizing and minimizing occasionally, at the beginning, they state they're going to find the maximum, but in the promising area that they find, they are actually minimizing the function. But no big deal, it's just some typos.

The major confusion in my head however, is the use of Kmeans. The authors use Kmeans to tell "good" samples from "bad" samples, it takes $(\vec{x}, f(\vec{x}))$ as the feature vector, where $\vec{x}$ is the sample coordinates, and $f(\vec{x})$ is the function value. The coordinates and function value, they don't even have the same scale, and there's no proof that Kmeans will separate the samples as we expected, but the authors just take the cluster with a bigger average value as "good", and it all works somehow. I guess maybe the accuracy of separation is not that important after all, it can tolerate some randomness.

As for the algorithm complexity, it might seems a little time consuming to rebuild a entire tree each time of sampling, and the time of building a tree increase as the number of samples increase, but it's acceptable because we don't want to sample too much after all.

For experiments, the paper first illustrate some results on the MuJoCo[9] locomotion tasks, LA-MCTS uses TuRBO to collect new samples in promising areas in this task. Compared with many other methods, LA-MCTS+TuRBO do perform very well in terms of reward gain per sample. This is maybe because MCTS is good at motion planning.

After that, the paper compares the performance of BO with LA-MCTS+BO, and TuRBO with LA-MCTS+TuRBO on a task of optimizing Ackley function and Rosenbrock function, results shown in Figure 1, they then conclude that LA-MCTS provides performance boost for both BO and TuRBO consistently, which I think is fair for BO, but the improvement shown in LA-MCTS+TuRBO is really not that significant. Maybe LA-MCTS provides more significant improvements in certain kind of tasks.

2 Experiments

In this section, I perform some experiments like Lunarlanding from MuJoCo tasks, in which we need to control a robot to land in between two flags as shown in Figure 2

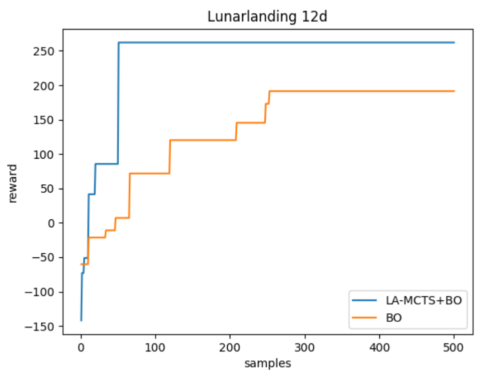
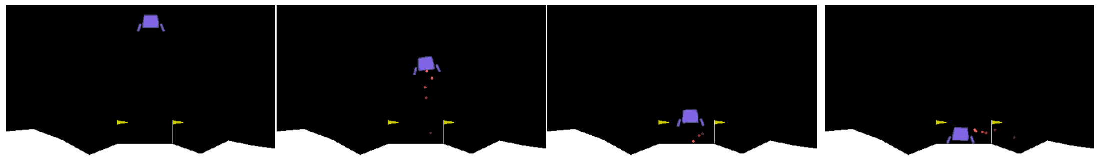


Figure 3: comparison of LA-MCTS+BO and BO on Lunarlanding

I use LA-MCTS+BO instead of LA-MCTS+TuRBO to see how LA-MCTS work with BO for motion planning, since the original paper shows that LA-MCTS+TuRBO beats other state of art algorithms. Result shown in Figure 3.



(a) frame 34. (b) frame 67. (c) frame 137. (d) frame 192.

Figure 4: landing process

The result shows that LA-MCTS+BO indeed gain higher reward than just BO, and visualization of the landing process in Figure 4 shows that the robot lands more steady with LA-MCTS+BO. One more thing to notice is that LA-MCTS+BO reach its bottleneck in less than 100 samples, while BO reach its bottleneck in 300 samples, so LA-MCTS+BO also saves function calls. But it's still not comparable with LA-MCTS+TuRBO, this indicates that as a "meta-algorithm", the choice of sampler in promising areas has a big influence on the searching performance.

Apart from motion planning, Ackley and Levy function are two functions that are widely used in validation of optimization algorithms, the global minimum of Ackley function is

$$Ackley(\vec{x}) = 0, \vec{x} = \vec{0}$$

and the global minimum of Levy function is

$$Levy(\vec{x}) = 0, \vec{x} = \vec{1}$$

2-d Ackley and Levy function are shown in Figure 6[1] ,the original paper does not show how dimension of the functions affects the performance of LA-MCTS, so I sample 500 times using LA-MCTS+BO for Ackley and Levy function with dimensions from 1 to 40, and record the minimum value the algorithm gets in the 500 samples. Result shown in Figure 5.

For Ackley function, dimension almost doesn't matter, LA-MCTS+BO always find a value near 0 after 500 samples, it sure better get a few more samples when dimension grows, but samples required grows slowly with dimension. On the other hand, when dimension grows for Levy function, LA-MCTS+BO fails to maintain the minimum value found at a low level, in fact, the minimum value found in 500 samples grows exponentially. This sharp contrast may be result from the shape of these two functions. The results show that the performance of LA-MCTS is still not satisfying for higher dimension Levy function.

[1]https://www.sfu.ca/ ssurjano/levy.html, https://www.sfu.ca/ ssurjano/ackley.html

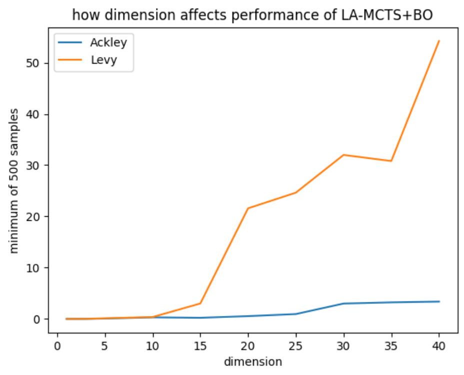
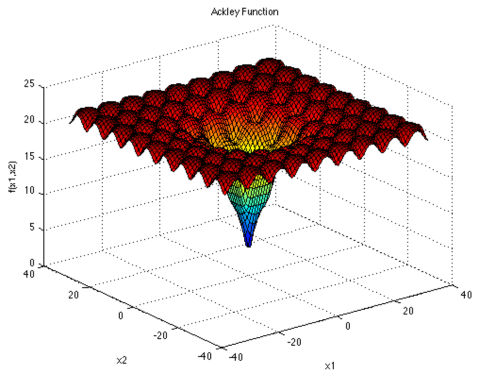
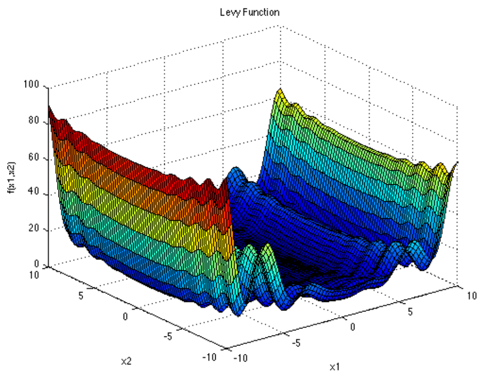


Figure 5: influence of dimension



$$f(\mathbf{x}) = -a \exp\left(-b\sqrt{\frac{1}{d}\sum_{i=1}^{d} x_i^2}\right) - \exp\left(\frac{1}{d}\sum_{i=1}^{d}\cos(cx_i)\right) + a + \exp(1)$$



$$f(\mathbf{x}) = \sin^2(\pi w_1) + \sum_{i=1}^{d-1}(w_i - 1)^2\left[1 + 10\sin^2(\pi w_i + 1)\right] + (w_d - 1)^2\left[1 + \sin^2(2\pi w_d)\right], \text{ where}$$

$$w_i = 1 + \frac{x_i - 1}{4}, \text{ for all } i = 1, \ldots, d$$

Figure 6: 2d Ackley and Levy function

Table 1: Dependencies

module	repository
Kmeans	KMeansRex
Gaussian process regression	GPR
SVM	thundersvm[10]
vector operation	Eigen
multithread support	OpenMP

3 Extension

When attempt to do extension, I re-implement this LA-MCTS in c++ for the convenience and efficiency of my tests, and also to prove that it can be reproduced.

But after weeks of trying, there's still something wrong, I just cannot get comparable results with the original python version, I think this is because of the difference of the dependencies. So, I then switch back to python code, and finally I see some significant improvements using my own extensions!

Both of my c++ code and python code are in the repository below:

`https://github.com/winie-the-pooh/LA-MCTS-in-CPP`

At the beginning, because I'm using C++, I need to use a totally different set of dependencies, listed in table 1[2].

I tried a lot of things that I think maybe a fix, but to be honest, it's all rather unsuccessful, the running time is reduced by huge amount though, due to the efficiency of C++ and the hardware optimization conducted. I highly suspect that there's something wrong with some of the dependencies. :-(

Sadly, I tried to figure out where gets wrong for C++ for many days, but it is still unfixed till now, so I have to switch back to python to test my ideas, and the results show both improvements in running speed and searching performance!

At this point, I don't have much time to run more full tests or tune my hyper parameters, so I just choose my parameters empirically, the performance can be even better with carefully chosen parameters. But don't trust me, my python codes are in the repository above, you can test for yourself. I just modify the original code that the paper provides like I do in Experiments. The following is my extension ideas:

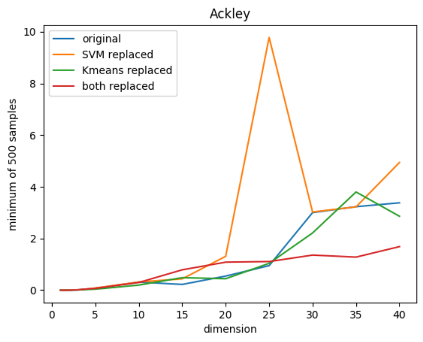
I get mainly 2 ideas, the first idea that come across my head is to replace SVM with something else. Since if you think of this, what SVM actually do in LA-MCTS is only to constrain the searching space to a smaller region for the samplers like BO. There's a lot of more efficient ways to do that, I come up with an idea that first find the weighted center of the current samples(weighted according to the function value, sample coordinates with better value gets bigger weight), and then sample using a normal distribution around that center, sigma of the distribution gets smaller when the depth of the leaf gets larger in order to constrain the searching space to a smaller area when there are more samples. Specifically, I use this center, again it's all empirical:

$$\vec{c} = \frac{\sum_i \vec{x_i} \times e^{-f(\vec{x})/4}}{\sum_i e^{-f(\vec{x})/4}}$$

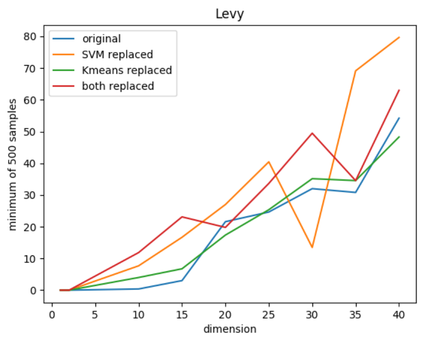
and for sigma, I just use $\frac{1}{depth\ of\ current\ node}$. After that, the Acquisition Function proposes samples with bigger expected improvements as usual. I get this idea from meanshift[2].

And my second idea is to replace Kmeans, as I stated in Review, Kmeans wouldn't necessarily separate good samples from bad, and the judgement of good or bad should be made purely based on function values, so I want to first calculate the average function value of all the current samples, and mark those with a function value bigger than the average as good, and the rest as bad.

[2]many of the code in dependencies are modified to integrate with my code



(a) Ackley

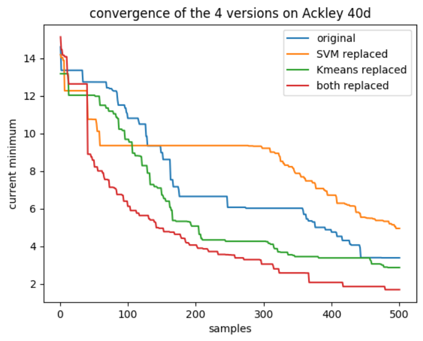


(b) Levy

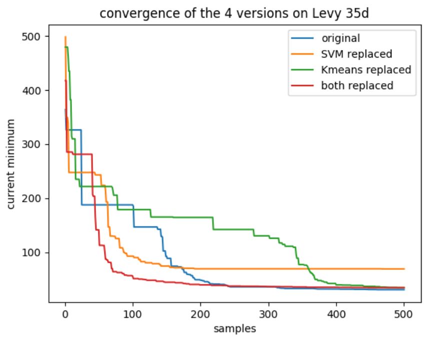
Figure 7: searching performance comparison

In the experiments, I compare the performance of the original version, SVM replaced version, Kmeans replaced version, and both SVM and Kmeans replaced version in the task of finding the minimum of Ackley and Levy function of different dimension. Results shown in Figure 7.

The extend versions achieve comparable or better performance than the original version already, except that only replace SVM acts a little strange at some dimensions, this maybe because as I said, Kmeans alone doesn't necessarily tells good from bad or it can be due to the randomness in the algorithm. With more carefully chosen center and sigma, the performance can be further improved. for instance, we can make sigma bigger when the function dimension gets bigger, this will help with the searching performance of high dimension functions. Figure 8 shows how the 4 versions converge for 40-d Ackley function and 35-d Levy function, note that the extend versions also converge faster.



(a) Ackley 40d



(b) Levy 35d

Figure 8: the convergence process of the 4 versions

Apart from searching performance, they also provide significant speed boost since there are no more time consuming SVM or Kmeans. So these are really some promising extensions.

References

[1] Eric Brochu, Vlad M Cora, and Nando De Freitas. A tutorial on bayesian optimization of expensive cost functions, with application to active user modeling and hierarchical reinforcement learning. *arXiv preprint arXiv:1012.2599*, 2010.

[2] Dorin Comaniciu and Peter Meer. Mean shift: A robust approach toward feature space analysis. *IEEE Transactions on pattern analysis and machine intelligence*, 24(5):603–619, 2002.

[3] David Eriksson, Michael Pearce, Jacob R Gardner, Ryan Turner, and Matthias Poloczek. Scalable global optimization via local bayesian optimization. *arXiv preprint arXiv:1910.01739*, 2019.

[4] Peter I Frazier. A tutorial on bayesian optimization. *arXiv preprint arXiv:1807.02811*, 2018.

[5] Yaochu Jin and Jürgen Branke. Evolutionary optimization in uncertain environments-a survey. *IEEE Transactions on evolutionary computation*, 9(3):303–317, 2005.

[6] Taehyeon Kim, Jaeyeon Ahn, Nakyil Kim, and Seyoung Yun. Adaptive local bayesian optimization over multiple discrete variables. *arXiv preprint arXiv:2012.03501*, 2020.

[7] Mikita Sazanovich, Anastasiya Nikolskaya, Yury Belousov, and Aleksei Shpilman. Solving black-box optimization challenge via learning search space partition for local bayesian optimization. *arXiv preprint arXiv:2012.10335*, 2020.

[8] Il'ya Meerovich Sobol'. On the distribution of points in a cube and the approximate evaluation of integrals. *Zhurnal Vychislitel'noi Matematiki i Matematicheskoi Fiziki*, 7(4):784–802, 1967.

[9] Emanuel Todorov, Tom Erez, and Yuval Tassa. Mujoco: A physics engine for model-based control. In *2012 IEEE/RSJ International Conference on Intelligent Robots and Systems*, pages 5026–5033. IEEE, 2012.

[10] Zeyi Wen, Jiashuai Shi, Qinbin Li, Bingsheng He, and Jian Chen. ThunderSVM: A fast SVM library on GPUs and CPUs. *Journal of Machine Learning Research*, 19:797–801, 2018.