Automated performance tuning of distributed storage system based on deep reinforcement learning

Lu Wang, Wentao Zhang, Yaodong Cheng
Computing Center, Institute of High Energy Physics, Chinese Academy of Sciences, Beijing, P.O.Box 100049, China
wanglu@ihep.ac.cn

Abstract. Automated performance tuning is a tricky task for a large scale storage system. Traditional methods highly rely on experience of system administrators and cannot adapt to changes of working load and system configurations. Reinforcement learning is a promising machine learning paradigm which learns an optimized strategy from the trials and errors between agents and environments. Combining with the strong feature learning capability of deep learning, deep reinforcement learning has showed its success in many fields. We implemented a performance parameter tuning engine based on deep reinforcement learning for Lustre file system, a distributed file system widely used in HEP data centres. Three reinforcement learning algorithms: Deep Q-learning, A2C, and PPO are enabled in the tuning engine. Experiments show that, in a small test bed, with IOzone workload, this method can increase the random read throughput by about 30% compared to default settings of Lustre. In the future, it is possible to apply this method to other parameter tuning use cases of data centre operations.

1. Introduction
Automated performance tuning is an important and tricky task for a large scale storage system. Heuristic methods require experience of system administrators. A blind search in high dimensional parameter space is time consuming. Neither of these methods can adapt to changes of workload and system configurations. Besides, it is difficult for these methods to model and calculate the long term impact of a specific action in a sequence of tuning actions.

Reinforcement Learning [1] (RL) is a kind of machine learning which learns to make a decision by sequentially interacting with environment. Different from simple trial and error, in the process of RL, it is possible to model and calculate the long term impact of an action, to have tradeoffs between exploration and exploitation and to adapt to the environment changes. Combining with the powerful feature-learning capability of deep learning, deep reinforcement learning (DRL), has seen successful applications in domains of games, robotics, and so on. Some of the DRL algorithms also support stochastic policies and/or continuous action value spaces.

The Lustre file system [2] is the major disk storage system at Institute of High Energy Physics, (IHEP). Currently, the system is consisted of 60+ servers and 2000+ clients, and provides 14 PB capacity for multiple HEP experiments. Performance tuning of Lustre is a tricky task: first, there is a large difference between the IO patterns of different workloads, while on a computing node running Lustre client, the workload is mixed and unpredictable. Secondly, there are tens of tuning parameters distributed in different Lustre kernel modules, but there is no “one-fits-all” configuration of these parameters.
As the most popular file system in HPC, Lustre supports online parameter tuning and provides a rich collection of monitoring metrics. That is to say, on the Lustre client, its running state is observable, and tuning actions can take effect immediately after they are executed. These features provide the premise to transfer the Lustre performance tuning task into a DRL problem.

2. Deep Reinforcement Learning
Deep reinforcement learning learns from the sequential interaction between an agent and the environment. As shown in Figure 1, at each step $t$, after the learning agent receives an observed state $S_t$, it will recommend an action $A_t$ according to present policy $\pi$. When the environment receives and executes the action $A_t$, it will transfer to a new state $S_{t+1}$ and generate a reward $R_{t+1}$. Policy $\pi$ defines which action to take (deterministic policy) or the probability of taking an action (stochastic policy) on a certain state.

The goal of the agent is to learn a policy $\pi^*$, which can maximize average cumulative reward $R$ defined in equation (1), of possible action trajectories $\tau = (s_0, a_0, r_0, s_1, a_1, r_1, \ldots)$. The $\gamma \in (0, 1)$ is used to decrease the weight of later received rewards. In deep reinforcement learning, the policies are parameterized and can be modeled by neural networks. With Temporal-Difference [1] learning, policies can be updated at a certain step size, and it is not a necessity to be updated at the termination state. Value function $V^\pi$ and $Q^\pi$ define how good a state and a (state, action) pair is under current policy, see equation (2). There are three kinds of DRL algorithms: value based, policy gradient based and Actor Critic based.

$$R = E \left[ \sum_{t=0} R_t \right]$$

$$V^\pi = E \left[ \sum_{t=0} \gamma^t R_t \mid S_0 = s, \pi \right]$$

$$Q^\pi = E \left[ \sum_{t=0} \gamma^t R_t \mid S_0 = s, A_0 = a, \pi \right]$$

2.1. Value-based DRL algorithm
Value based DRL algorithm learns parameterized Q functions, and acts according to an implicit policy: for state $s$, it selects the action $a$ which has the largest Q value, as shown in equation (3). During model training process, $\epsilon$-greedy policy is always used to balance the exploration and exploitation, that is to say, use $1 - \epsilon$ probability to choose action defined by equation(3) and $\epsilon$ probability to choose a random action. $\epsilon$ can be decayed with time step.

$$a = \underset{a \in A}{\text{arg max}} Q(s, a)$$

$$Q(s, a) = E \left[ r + \gamma \max_{a'} Q(s', a') \mid s, a \right]$$
where \( y_i = E \left[ r + \gamma \max_{a'} Q(s', a') | s, a \right] \) (5)

\[
Loss_i = (y_i - Q(s, a; \theta))^2
\] (6)

According to Bellman Equation defined in equation (4), \( y_i \) and \( loss_i \) can be computed with batch of tuning samples as \((s,a,s',r)\). By minimizing \( loss_i \) with gradient descent, function \( Q \) can be finally be approximated. Traditional DRL with a nonlinear function approximator has shortcomings as instability or divergence. By means of experience replay, target network and Exploration-Exploitation Tradeoff, Deep Q-learning[3] largely overcome these problems.

2.2. Policy Gradient Base Agent

Policy gradient based DRL learns an explicit parameterized policy function in form of \( \pi(a) \), or \( P(a|\pi) \) to maximize the expected reward \( J(\theta) \) of trajectories, as shown in equation (7).

\[
J(\theta) = E \left[ \sum_{t=0}^{T} y^t r_t | \pi_\theta \right] = E_{t\sim p(\tau; \theta)} [r(\tau)] = \int r(\tau)p(\tau; \theta) d\tau
\] (7)

\[
V_{\theta} J(\theta) = \int r(\tau)V_\theta \log p(\tau; \theta) d\tau = E_{t\sim p(\tau; \theta)} [r(\tau)V_\theta \log p(\tau; \theta)]
\]

\[
\approx E_{t\sim p(\tau; \theta)} \left[ \sum_{t=0}^{T} V_\theta \log \pi_\theta(a_t | s_t) \sum_{t'=t}^{T} R(s_{t'}, a_{t'}, s_{t'+1}) \right]
\] (8)

Vanilla policy gradient (VPG) algorithm maximized \( J(\theta) \) by gradient ascend, shown in equation (8). Recently, algorithms TRPO, PPO[4] have improved VPG on data efficiency and robustness by different formulation of function \( \theta \).

2.3. Actor Critic based Agent

An Actor-Critic based algorithm learns a parameterized value function \( Q \) and a parameterized value function \( V \). It can be proved that if the function \( b \) only depends on state \( s \), equation (9) equals 0.

\[
E_{a_t \sim \pi_\theta} [V_\theta \log \pi_\theta(a_t | s_t) b(s_t)] = 0
\] (9)

Therefore, policy gradient can be written as equation (10).

\[
V_{\theta} J(\pi_\theta) = E_{t\sim \pi_\theta} \left[ \sum_{t=0}^{T} V_\theta \log \pi_\theta(a_t | s_t) \left( \sum_{t'=t}^{T} R(s_{t'}, a_{t'}, s_{t'+1}) - b(s_t) \right) \right]
\] (10)

The rightmost part of (8), \( A^\pi = \sum_{t'=t}^{T} R(s_{t'}, a_{t'}, s_{t'+1}) - b(s_t) \), is called an advantage function. It defines how much an action was better than expected. Algorithm which uses an advantage function in form of \( A^\pi(s, a) = (Q^\pi(s, a) - V^\pi(s)) \) is called an Actor-Critic based algorithm. It updates these two functions alternately in the procedure as described in Figure 2[5]. Typical Actor-Critic based algorithm includes A2C, A3C and Soft Actor-Critic[6].

3. Implementation

We implemented a DRL based framework [7] for automated Lustre client performance tuning, as shown in Figure 3. The daemon process running on Lustre client (target node) has two function components: Monitor and Actor. At each time step, “Monitor” collects Lustre performance metrics, builds a “state” (details in 3.1) vector, and sends it to the policy node. With the received “state” as input, the policy node, which runs the DRL agent, will select best tuning “action” with present policy network and then send the “action” number (details in 3.2 )to target node. Then the “Actor” will act according to the number it received, and compute the present “reward” (details in 3.3). Then “Actor” will send the “reward” and new
state back to the policy node. After receiving the “reward “and new state, the policy node will build a new training sample with episode (state, action, reward, state\textit{new}), and insert into its sample buffer. After a certain time gap, the policy node will optimize its model with stochastic batched samples. The policy node communicates with target nodes through ZeroMQ\[8\].

\begin{align*}
\text{Initialize policy parameters } & \theta, \text{ critic parameters } \emptyset \\
\text{For iteration}=1,2 \ldots \text{ do} & \\
\text{Sample } m \text{ trajectories under current policy} & \\
\Delta \theta & \leftarrow 0 \\
\text{For } i=1,\ldots,m \text{ do} & \\
\text{For } t=1,\ldots,T \text{ do} & \\
A_t & = \sum_{t'\geq t} \gamma^{t'-t} \cdot r_t - V_{\emptyset}(s_t) \\
\Delta \emptyset & \leftarrow \Delta \emptyset + A_t \cdot \nabla_{\emptyset} \log (a_t|s_t) \\
\Delta \theta & \leftarrow \sum_i \sum_t \nabla_{\theta} [A_t]^2 \\
\theta & \leftarrow \alpha \Delta \theta \\
\emptyset & \leftarrow \beta \Delta \emptyset \\
\text{End for} & \\
\end{align*}

Figure 2. Pseudocode of Actor-Critic based RL algorithm.

3.1. State
The state vector is built with 17 performance metrics under /proc/fs/lustre/* as:
- \textit{cur\_read\_throughput}, \textit{write\_throughput}, \textit{dirty\_pages\_hits}, \textit{dirty\_pages\_misses}, \textit{read\_IOPS}, \textit{write\_IOPS}, \textit{used\_mb}, \textit{unused\_mb}, \textit{recliam\_count}, \textit{cur\_dirty\_bytes}, \textit{cur\_grant\_bytes\_in\_flight}, \textit{timeouts}, \textit{avg\_waittime}, \textit{osc\_cached\_mb}, \textit{req\_waittime}, \textit{req\_active}

For those metrics which has a number for each Lustre OSC, an averaged number will be reported.

3.2. Action
Currently, we only tune 7 Lustre parameters on client as:
- \textit{max\_dirty\_mb}, \textit{max\_pages\_per\_rpc}, \textit{max\_rpcs\_in\_flight}, \textit{max\_rpcs\_in\_flight}, \textit{max\_cached\_mb}, \textit{max\_read\_ahead\_mb}, \textit{max\_read\_ahead\_per\_file\_mb}, \textit{max\_read\_ahead\_whole\_mb}

At each tuning step, one of these 7 parameters will be tuned according to the action number send back from policy node. There is 14 actions, the first 7 actions will tune up corresponding parameter with a predefined GAP\_SIZE, while the second 7 actions will tune them down. If the value after tune is located outside the ranger predefined by [MIN, MAX], the agent will keep the parameter unchanged. Different parameter has different GAP\_SIZE, MIN and MAX.

3.3. Reward
Currently, reward is defined by the increase of read and write throughput as defined in equation (9).

\[ R_t = \text{Read\_throughput}_t + \text{Write\_throughput}_t - \text{Read\_throughput}_{t-1} - \text{Write\_throughput}_{t-1} \]

3.4 DRL algorithms
We implemented three DRL models based on Deep Q-learning, PPOand A2C algorithms separately. The first model only use an actor network, while the rest two use both actor networks and critic networks. All the networks adapt a 3-layers MLP network architecture. In the actor networks the input
layer uses the “state” tensors, while the output layer has the size of the action number and action number will be chose through a softmax operation. For critic network, the input of second hidden layer is a concatenated result of the first hidden layer output and the action tensor. The action tensor is the one-hot result of the action network. Generally, the number of hidden units is twice the size of the input layer. The framework is developed with PyTorch [9] version 0.4, and the activation function is ReLU, the optimization algorithm of MLP is Adam. During the training, we try to balance the exploration of new policy and exploitation of current policy by switch the output of action network with a randomly chose action number with a probably of epsilon. The epsilon decays with the number of training step from ~0.015 to 0.01.

3.5 Improvements on previous work
Compared with previous work [10], this work has improvements as:
1. The state is consisted of metrics of a single client instead of metrics from all the clients. It will make the system more flexible and scalable.
2. This work has implemented two policy gradient based algorithms which present recent progresses in the DRL field while the preceded work has only implemented Deep Q-learning.

4. Experiments

4.1. Testbed
We validated the tuning framework in an isolated Lustre file system. It consists of 1 Metadata server (MDS), 1 object storage server (OSS), 2 Lustre clients and a policy node. The interconnection is 10Gb Ethernet. The data storage of OSS is a DELL MD3860 disk array with 60 SATA disks configured in DDP redundancy mode. The policy node is configured with a NVIDIA Tesla K80 GPU to accelerate the MLP model training.

4.2. Benchmark
The test workload is generated by IOzone [11] (version 3.4.9). We tested 5 different workloads: sequential write and rewrite, read and reread, random write and read, stride read with parameters as 8 GB file size, 1M record size, 2M read stride. Each Lustre client ran 16 test processes in the test.

4.3. Experiment Process
Before performance test of the DRL engine, we ran three rounds of IOzone test with default setting to record the average performance of each workload as baseline. After that, for each algorithm, we first ran 24 hours pre-training to train a model from scratch, and then we reset the parameters on Lustre clients, and then ran three rounds of 10-hour on line training and evaluation to record average performance for comparison.

4.4. Result
As shown in table 1, write, stride read, random read and write are the four workloads which need to be optimized most. For these four workloads, policy gradient based algorithms (PPO and A2C) can achieve a performance boost up to 32% while Deep Q-learning based algorithms provided a lower performance than the baseline. The DRL based method shown its promise in larger storage systems.

5. Conclusion and Future Work
We implemented a DRL based framework to automate the performance tuning task of Lustre. In a small testbed, policy gradient based RL algorithms perform better than the default settings. DRL is a fast developing domain, in the future, we will widen and deepen our exploration in flowing directions:
1. To train a recursive neural network to abstract input features in multiple time windows to make sure the input state has Markov property,
2. To implement algorithms which support continuous action space to better adapt to the parameter tuning scenarios,
3. To implement algorithms which are off-policy and therefore more data efficient.

Table 1. Average Throughput (MB/s) with different DRL based parameter tuning algorithms.

| Workload   | Algorithm     | Base     | Deep Q-learning | A2C      | PPO      |
|------------|---------------|----------|-----------------|----------|----------|
| Write      | Base          | 38.53    | 37.06 (-4%)     | 36.63 (-5%) | 37.97 (-1%) |
|            | Deep Q-learning |         |                 |          |          |
| Read       | 1206.66       | 953.28 (-21%) | 1155.63 (-4%) | 1278.88 (+6%) |
| Stride read| 109.16        | 111.56 (+2%)  | 126.59 (+16%)  | 123.16 (+13%) |
| Random read| 59.16         | 48.93 (-17%)  | 78.28 (+32%)   | 75.22 (+27%) |
| Random write| 17.50       | 17.66 (+1%)   | 18.16 (+4%)    | 17.59 (+1%) |

Acknowledgments
This work is supported by Youth Innovation Promotion Association of CAS (No. 201512)

References
[1] Szepesvari, “Algorithms for Reinforcement Learning”, free digital version available from http://www.ualberta.ca/~szepesva/papers/RLAlgsInMDPs.pdf (last update: June 25, 2018)
[2] Home of Lustre file system. “Lustre”[software]. Available from http://lustre.org/ [accessed 2019-12-20]
[3] Mnih V, Kavukcuoglu K, Silver, D et al 2015 Human-level control through deep reinforcement learning Nature 518 529–533
[4] John S, Filip W, Prafulla D, Alec R, and Oleg K 2017 Proximal policy optimization algorithms arXiv:1707.06347v2
[5] CS231n Convolutional Neural Networks for Visual Recognition, Avalible from http://cs231n.github.io[accessed 2019-12-20]
[6] Mnih, V et al. 2016 Asynchronous methods for deep reinforcement learning JMLR (New York, NY, USA) 48 1928-37
[7] Zhang W and Wang L, “Automated parameter tuning with deep reinforcement learning”[software], commit 9587b302feb89a9f38db757b043392ca5ad89f47, available from https://github.com/maiwen/DeepTuning
[8] “ZeroMQ”[software], available from http://zeromq.org [accessed 2019-12-20]
[9] “PyTorch”[software], available from https://pytorch.org [accessed 2019-12-20]
[10] Li, Y et al 2017 Proceedings of the International Conference for High Performance Computing, Networking, Storage and Analysis ( Denver, Colorado, ACM ) 1-14
[11] IOZone Filesystem Benchmark,”IOzone”[software], available from https://iozone.org[accessible 2019-12-20]