# CAPES: Unsupervised Storage Performance Tuning Using Neural Network-Based Deep Reinforcement Learning

Yan Li
University of California, Santa Cruz
yanli@ucsc.edu

Kenneth Chang
University of California, Santa Cruz
kchang44@ucsc.edu

Oceane Bel
University of California, Santa Cruz
obel@ucsc.edu

Ethan L. Miller
University of California, Santa Cruz
Pure Storage
elm@ucsc.edu

Darrell D. E. Long
University of California, Santa Cruz
darrell@ucsc.edu

## ABSTRACT

Parameter tuning is an important task of storage performance optimization. Current practice usually involves numerous tweak-benchmark cycles that are slow and costly. To address this issue, we developed CAPES, a model-less deep reinforcement learning-based unsupervised parameter tuning system driven by a deep neural network (DNN). It is designed to find the optimal values of tunable parameters in computer systems, from a simple client-server system to a large data center, where human tuning can be costly and often cannot achieve optimal performance. CAPES takes periodic measurements of a target computer system's state, and trains a DNN which uses Q-learning to suggest changes to the system's current parameter values. CAPES is minimally intrusive, and can be deployed into a production system to collect training data and suggest tuning actions during the system's daily operation. Evaluation of a prototype on a Lustre file system demonstrates an increase in I/O throughput up to 45% at saturation point.

## CCS CONCEPTS

• **Information systems** → **Storage management**; *Distributed storage*; • **Computing methodologies** → Neural networks;

## KEYWORDS

performance tuning, deep learning, q-learning

## 1 INTRODUCTION

Performance tuning is a common task where human experts analyze a system's historical and real-time performance indicators, and tweak values of parameters in order to increase certain performance metrics for running certain workloads. Tuning is challenging for a variety of reasons: First, the effect of adjusting a system's parameter can be influenced by factors such as system hardware, operating system, and running workloads. This correlation of variables makes predicting the effects of changes difficult at best, at worst humans may not have a clue what might happen if something were to change in a complex system. This issue arises from the fact that computers are nonlinear systems and there is no known method for quantifying and modeling a complex system to the level of precision required by performance tuning; in practice, all performance tuning has to be done on the actual system. Second, the delay between an action and the resulting change in performance makes it even harder to correlate the relationship between system input and output. Third, the available parameter space is huge, often including thousands of parameters and each parameter can take a wide range of values. Humans can only propose and evaluate a few commonly accepted parameter values derived from past experience, and they tend to reuse those same values across many systems for simplicity, leaving a larger, more diverse, parameter space unexplored. Fourth, assigning human experts to monitor dynamic workloads 24x7 is simply too costly in practice.

In machine learning practice, we can approach this problem as a game where the goal is to find an appropriate setting that will render the system more efficient. By observing some indicators in the system, the player can maximize a cumulative performance reward such as energy usage, operations per second, or data transfer throughput. Recent advancements in machine learning showed that Deep Reinforcement Learning (DRL) can perform unsupervised learning well on diverse data featuring long delays between action and reward [21]. Many techniques are being used in tandem to make this possible, such as Deep Q-learning and experience replay [19].

We developed CAPES (Computer Automated Performance Enhancement System) and demonstrated that it can increase the throughput of the Lustre high performance storage system [22] up to 45% under heavy workloads. The training is online and requires around 12 to 24 hours, which can be done during the system's daily operation. This is a significant increase in efficiency, especially for large and expensive system installations.

In comparison to earlier automatic tuning systems, CAPES has several advantages:

- It requires no prior knowledge of the target system.
- It requires little change to the target system and little downtime for setting up.
- It can run continuously to adapt to dynamically changing workloads.
- It can dynamically choose optimal values for parameters that used to be set statically.

## 2  BACKGROUND

*Automatic Performance Tuning.* Modern computer systems are highly complex and often have many tunable parameters to accommodate a wide range of workloads. Such parameters can include the number of worker threads, inbound/outbound buffer sizes, congestion window size, packet size, etc. All field engineers know that the different in performance for a customer's workload can be prominent before and after tuning. It is in the user's best interest to keep high cost systems, such as data centers or supercomputers, running at peak performance. Smaller enterprise computing systems can also see a considerable performance boost when parameters are tuned to values that best suit the user's environment.

Fine tuning a computer system is considered to be a form of art that requires extensive knowledge and experience, thus this service is often limited to a few high-paying customers. Often, for a service provider, there are simply not enough domain experts that understand all the quirks of a product. Most users simply have to fall back to following an inflexible, untailored performance tuning guide. Even for experts, a lengthy trial-and-error process is needed to obtain enough understanding about the customer's workloads, and this process can last weeks to months.

Yet even with the best experts, it is nigh impossible to achieve optimal performance. Using static parameter values is not ideal for handling dynamic workloads. Depending on how the system is used, workloads can change over time or be cyclical, but they rarely stay stable. The best a human expert can do is to come up with several sets of optimal setting values for several typical workloads at a coarse grade, and allowing the system to pick a set of values when a certain condition is detected, hoping these sets can cover most of the customer's workloads. It is prohibitively expensive and time consuming to find optimal values for all possible workloads. In addition to this difficulty, it would be very hard to exhaustively test all possible combinations of parameter values even for a static workload. Furthermore, modern systems may behave differently under peak traffic. When systems are pushed to the limit, the efficiency of many components can drop rapidly. This phenomenon is usually called "congestion collapse", and it is a common curse among network and storage researchers.

In all, it is clear that the existing, human driven, tune-benchmark-tune cycle does not fit the evolving nature and scalability of new technology. An automated hands-free solution to performance tuning is ever more important in an era of widespread high performance computing.

*The Challenge of Parameter Tuning.* Parameter tuning focuses on finding optimal parameter values that make a target system or application perform better. Better performance usually means lower energy consumption, higher throughput, lower latency, or an objective function that combines these.

Automated parameter tuning faces several challenges in practice:

(1) No high-precision model has been successfully constructed for a complex distributed system.
(2) Workloads are often dynamic and affect each other.
(3) The parameter space can be large, and sweeping through the entire space would be prohibitively slow.
(4) Tuning has to be responsive: when the workloads change, the system should also adjust quickly.
(5) Distributed systems can be large and the tuning system has to be extremely scalable.
(6) Tuning for multiple objectives should be possible, such as tuning for throughput and latency at the same time.

Theoretically, automatic dynamic parameter tuning can be constructed as a Decentralized Partially Observable Markov Decision Process (POMDP), and Bernstein et al. have proven that it is a NEXP-hard problem [5]. A perfect tuning system would need access to information of the entire history of observations, including the start times of all previous system resource requests, their finish times, and the start times of all future requests; this is simply impossible to construct in practice. Therefore, all existing tuning approaches have adopted some sort of approximation in their policies.

Designing these policies is inherently difficult because there is no known method that can perfectly establish the cause and effect between a tuning action taken and the system's reaction. First, the delay between applying a modification and its consequences makes it difficult to say whether a performance increase was due to a recent modification, or the effects of a modification done several minutes prior. Second, the delays could vary in length, and the length itself could be affected by the system, the workload, and many other factors. Third, any improvements measured could be the effects of several modifications taken in a specific order, which could be difficult to trace back. This is commonly known as the "Credit Assignment Problem" [26].

*Hyperparameters.* In the machine learning community, the parameters of machine learning algorithms are referred to as hyperparameters [6] to distinguish them from the parameters of the target system. Hyperparameters can greatly affect the efficiency of the machine learning algorithm, and need to be chosen carefully. Common methods for hyperparameter optimization include Bayesian optimization, random search, and gradient-based optimization [10].

*Q-learning.* Reinforcement Learning (RL) [26] is a branch of machine learning concerned with how an agent ought to take actions within an environment in order to maximize a certain reward. Reinforcement learning has seen successes in many areas, from robotics to game play [20]. An environment can be modeled as a stochastic finite state machine with inputs (actions) and outputs (observation and reward). The interaction between the environment and the agent is usually constructed as a Markov Decision Process (MDP) because the outcomes are partly random and partly under the control of the agent.

In the context of CAPES, we treat the target system as the environment. The tuning module is added to the environment as an agent, which observes the state of the target system. Actions are

then calculated by the tuning module and issued to the target system to carry out. We use the output of an objective function, whose input is the target system's performance, as the reward. Using an objective function provides the flexibility to tune the system for one or more objectives. The agent's goal is to find a policy to maximize the expected sum of all future rewards. Let $r_t$ be the reward we expect at time $t$, the expected sum of future rewards at time $t$ can be expressed as:

$$R_t = \sum_{i=t}^{n} r_i$$

Because of the random nature of the process, the further into the future we predict, the less precise the prediction becomes. Thus, it is better to discount future rewards so they are less important than immediate rewards. Also the workload could change, making current modifications possibly ineffective. Let $\gamma$ be the discount rate, the new expected sum of the rewards is:

$$R_t = \sum_{i=t}^{n} \gamma^{i-t} r_i$$

Let $s_t$ be the system's state and $a_t$ be an action at time $t$. Assuming we continue optimally from time $t$, we define an action-value function $Q(s, a)$ that represents the best expected sum of reward we can get. This is also why this method is referred to as Q-learning:

$$Q(s_t, a_t) = \max_{\pi} R_{t+1}$$

Here $\pi$ represents the policy that dictates what action to choose in each future decision step.

If function $Q$ exists, the decision making step can be written as:

$$\pi(s) = \max_{a} Q(s, a)$$

Bellman proved that $Q$ can be solve iteratively for any action $a'$ at state $s'$:

$$Q(s, a) = r + \gamma \max_{a'} Q(s', a')$$

This is the Bellman equation [26]. It is one solution to the credit assignment problem because iteratively solving this equation does not require the delay between action and reward to be known.

Solving the Q-function is the core task of Q-learning. Because the state space is usually prohibitively large, generalizing from known experience is important, and a nonlinear action-value function approximator is often used to express the Q-function, such as a neural network. Recent advances in deep neural networks (DNN) [4, 21] have shown that it can effectively learn concepts directly from raw sensory data. DNN employs several layers of neurons to build up progressively more abstract representations of the data. When DNN is used to approximate the $Q$ function it is often called a $Q$-network, and its weights are often referred to as $\theta$.

However, reinforcement learning is known to be unstable or even diverge when a nonlinear function approximator is being used. Many techniques have been developed to solve these challenges. Of these, experience replay is one of the most important methods.

*Experience Replay.* The transitions of states, actions, and rewards received from each step can be kept in a database so that we can *replay* these experiences later in different orders to break temporal correlation introduced by traditional training process that uses sequential system status. This kind of training is for overfitting

prevention and is generally referred to as experience replay [19]. During a training step $i$, we randomly pick samples from the database and pack them into a minibatch ($D$). Thus, the goal of the training step is to adjust $\theta_i$ to reduce the average mean-square error in the Bellman equation for samples within the minibatch:

$$L_i(\theta_i) = \mathbf{E}_D[(r_i + \gamma \max_{a'} Q(s', a'; \theta_i^-) - Q(s, a; \theta_i)^2] \qquad (1)$$

*Target Network.* Another important technique that helps the DNN to converge is using a target network [21]. Another $Q$-network of parameter, $\theta_i^-$, is used to approximate the optimal target value, and this network is called the target $Q$-network. We can use two methods to get the weights of the target network: getting the weights from a previous iteration, or using a slowly updated $Q$-network (the update rate has to be limited to a small value). Both of these methods help reduce the chance of overfitting the network to a specific workload.

Experience replay helps prevent overfitting, and limiting the update rate of the target $Q$-network has proven to increase the stability and efficiency of the training process [21].

*The Exploration-Exploitation Tradeoff.* Because DNN-based reinforcement learning is often used in environments that have prohibitively large state spaces or when the models are too hard to construct, a Monte Carlo approximation is often used. We only update the $Q$ function for states that the agent has actually visited in the environment, usually using a temporal difference method. This means that for unseen states the $Q$ function can only extrapolate from known experiences. Thus, it is important for the agent to "experience" as many states as possible during the training process (exploration). During the initial exploration process, we usually generate random actions to explore as many states as possible.

However, in a complex environment, performing totally random actions usually would not take the agent far. A standard practice is to initially use a mix of random and DNN-calculated actions, and gradually increase the chances of taking a calculated action further into the training process.

## 3 ALGORITHM

Figure 1 shows the architecture of CAPES and illustrates one possible way to use CAPES with a target system that contains several servers and application nodes. CAPES assumes little of the target system and only requires an interface to periodically extract states of the system and a way to change parameter values. The architecture shown in Figure 1 is not meant to limit how the system can be deployed. For instance, if the sampling of performance indicators is already implemented and the data is stored on a central monitoring node, CAPES can tap into that information without the need to deploy Monitoring Agents.

Each node has a Monitoring Agent and a Control Agent running on them. The Monitoring Agents gather Performance Indicators and Rewards from the target system's nodes and send them to the Interface Daemon. Performance Indicators are system measurements that are related to the system's operating status (§ 3.1). Rewards vary based on current tuning efforts, and reflect the successfulness of the current tuning (§ 3.2). The Interface Daemon (§ 3.3) relays the incoming Performance Indicators into the Replay Database (Replay DB). The DRL Engine reads the Performance Indicators from the
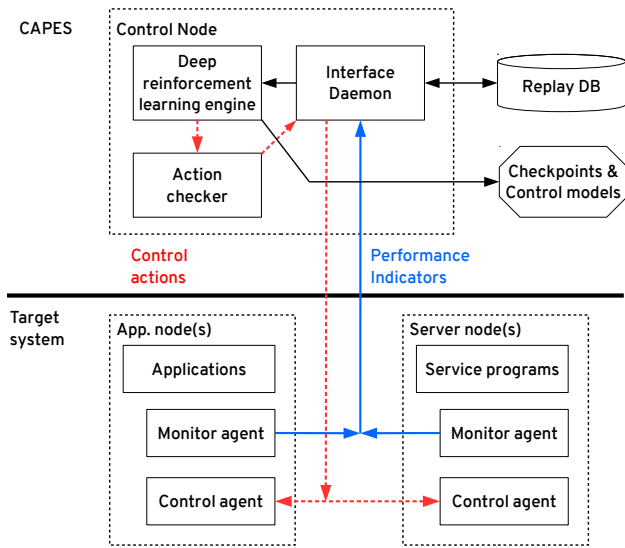
**Figure 1: CAPES architecture. Solid blue lines represent the flow of performance indicators. Dashed red lines represent the flow of actions.**

Replay DB to do training steps (§ 3.4). At a fixed interval, the DRL Daemon sends back an Action via the Interface Daemon, which will broadcast the action to the action's targeted Control Agents. These actions are also stored within the Replay DB, as part of Experience Replay. Finally, the Control Agents make the appropriate changes on the target nodes when actions are received (§ 3.7).

In a production environment, the Interface Daemon, Replay DB, and DRL Engine can run on one or more dedicated nodes to prevent interference with other nodes in the target system. For efficiency, the DRL Engine can be run on a separate node with a GPU for faster DNN computation. This is strictly for performance gains. In addition, the node that the Replay DB runs on should have plenty of RAM, ideally to keep the whole database in memory.

## 3.1 Choosing Performance Indicators

Performance Indicators are important for tuning the system because DNN relies on analyzing them to understand how the system is running. We should include system states that are related to the metric we wish to tune. This is a feature selection problem, which was deemed to be one of the most important steps for successfully applying almost any machine learning algorithm. However, advances in DNN has rendered this step less important because DNN is good at picking out useful data among noisy, raw inputs.

Therefore, we can be quite liberal on choosing performance indicators; any system statuses that are likely related to the performance of the system should be included. Both raw and secondary system statuses, derived from raw system status, can be included. Samples of raw system status include number of CPUs, CPU utilization, free memory, separate read/write I/O rate of each storage device, and buffer size. Samples of secondary system status could

be the total number of active threads, which needs to be calculated by counting the number of threads that are running.

All inputs to the Deep Neural Network should be converted into floating point numbers. This is easy for integers. Nonnumerical statuses, such as which I/O scheduler the system is using and what power status the system is in, should be converted into numerical values, such as 0, 1, or 2. Examples of such could be 0 for a round robin scheduler or 1 for a lottery scheduler.

System statuses that are accumulative in nature should generally be excluded unless they are known to be related to system performance. Such accumulative statuses include system uptime, total sent/received bytes over the network, total read/write bytes of I/O devices, etc. The rates of change of these statuses can be useful for indicating a system's operational status, but the total sums are generally not useful. For instance, it is unlikely that the system's optimal settings should rely on how many bytes the system has received since it started up.

Date and time should also be included if the workload is known to be cyclical, such as many enterprise workloads, however we should not include it as a single representation. Instead, it is easier for the DNN to understand if we include the month, day of the week, hour, and minute as separate performance indicators. By doing this, the DNN can discover any relationship between changes in the workload and the hour that it changed.

There may be a potential problem by picking too many performance indicators, which may result in increased computational cost. We can safely overlook this concern because the throughput of a modern desktop GPU can offset the increased computational costs given that we do not grossly overload the system (see Table 2 for the measured training speed in the evaluation).

## 3.2 Reward

Reward plays an important role and guides the direction of the whole tuning process. After performing each action on the target system, CAPES measures an immediate reward. For instance, after changing the congestion window size, we can measure the change of I/O throughput at the next second to use it as the reward. Only the immediate reward is necessary because there is no need to worry about the delay between an action and a reward since the Q-function will ultimately converge to the optimal oracle-like function after iterative trainings, according to Bellman's proof [26].

We use the output of an objective function as the reward. For single-objective tuning, the objective function equals the tuning objective measurement, such as throughput or latency. It is also common to use an objective function that combines multiple objectives, such as tuning for both higher throughput and fairness [17].

## 3.3 Monitoring Agents and The Interface Daemon

A Monitoring Agent runs on each node that needs to be monitored. At a predesignated sampling frequency, it collects Performance Indicators and sends them to the Interface Daemon for processing. We call each of these actions a *sampling tick*. In order to minimize both CPU utilization and network communication, we use a differential communication protocol designed to only send out a performance indicator when its data is different from the value of the previous

sampling tick. In addition, all network communications are compressed. If the target system uses different networks for data and control, the monitoring agents should use the control network to communicate with the Interface Daemon.

The Interface Daemon is a lightweight daemon that receives incoming messages from all Monitoring Agents. It also receives suggested actions from the DRL Engine, and broadcasts them to the Control Agents. Introducing the Interface Daemon into the system has several benefits. First, it decouples the network communication code from other parts of the system code. Second, it is the only component that needs to write to the Replay DB (the DRL Engine only needs to read from it), greatly reducing the overhead of locking the Replay DB. Third, this enables independent control of the Monitoring Agent and the DRL Engine so we can choose to do solely monitoring or training on demand.

### 3.4 Modeling and Training the Deep Neural Network

Mathematically, the purpose of the training step is to minimize the prediction error for the training data. Prediction error is the different between the neural network's predicted performance after observing the system's status and the actual system performance one second later. The prediction reflects how "well" CAPES understands the target system, and a lower prediction error leads to better tuning results. The Deep Reinforcement Learning (DRL) Engine retrieves uniformly random observations from the Replay DB and feeds them into the deep $Q$-network for training (experience replay). Because the performance indicators of one sampling tick cannot reflect the moving trend of these indicators, it is common to use a stack of multiple consecutive snapshots in the DNN training process [21]. Let $d_{i,j}$ be the output of the objective function of node $i$ at time $j$, $N$ be to total number of nodes, and $S$ be the number of sampling ticks. We construct the observation at time $t$ as a matrix:

$$s_t = \begin{vmatrix} d_{1,t-S+1} & d_{2,t-S+1} & \cdots & d_{N,t-S+1} \\ d_{1,t-S+2} & d_{2,t-S+2} & \cdots & d_{N,t-S+2} \\ \cdots\cdots\cdots\cdots\cdots\cdots\cdots\cdots\cdots\cdots\cdots \\ d_{1,t} & d_{2,t} & \cdots & d_{N,t} \end{vmatrix}$$

A set of random observations from the Replay DB is packed together as one minibatch and fed to the DNN trainer. Batching minimizes data movement overhead between the main memory and GPU memory, and is highly efficient because all computation can be done as matrix manipulation in the GPU.

The Q function can be parameterized using a neural network in many ways that differ in terms of the number, size, and type of hidden layers, and how the Q-value (e.g. the predicted reward) for candidate actions are calculated. There are primarily two methods for calculating the Q-values: the first type maps an observation-action pair to scalar estimates, and the second type maps an observation to an array of Q-values of each action [21]. The first type requires a separate forward pass to compute the Q-value of each candidate action, resulting in a cost that scales linearly with the number of actions. The main advantage of the second type is the ability to compute Q-values for all possible actions in a given state with only a single forward pass through the network. We have chosen the second type for CAPES because of its lower computational cost.

Because the observations are floating point numbers that represent system statuses and are usually not related by locality (adjacent numbers in observations are not necessarily related), we choose to use a multi-layered perceptron (MLP) network to construct the DNN. MLP is a mature method that can learn to classify any linearly separable and non-separable set of inputs. It can represent boolean functions, such as AND, OR, NOT, and XOR, and can allow a user to get approximate solutions for complex problems. In CAPES, we use a standard two-hidden-layer MLP with a hyperbolic tangent (tanh) nonlinear activation function. The two hidden layers are of the same size as the input array. The final output layer is a fully-connected linear layer with a single output for each valid action. According to the Universal Approximation Theorem, a feedforward network with a single hidden layer is enough to approximate any mathematical function [12]. But the learnability of a single hidden layer is still not clear. In practice, it is common to use two or more fully connected layers. We chose to begin with two hidden layers as Mnih et al. did for DRL [20]. Adding more layers becomes a problem of diminishing returns, with each additional layer adding significantly more computation time while returning lower gains in training successes.

We use the Adam optimizer [15] for training the DNN. Adam is accepted by the machine learning community as virtually the best stochastic gradient descent optimization algorithm. It has high convergence speed and good at escaping from saddles and certain local minima [23]. The DRL Engine is a separate process, and always runs during the training step using different random minibatches. For each minibatch, we update the target network's $\theta_i^-$ using $\theta_i$:

$$\theta_i^- = \theta_i^- \times (1 - \alpha) + \theta_i \times \alpha$$

Where $\alpha$ is the target network update rate.

### 3.5 Replay Database

One training step ($w$) needs the transition of system status from $t$ to $t + 1$, the action performed, and the reward after performing the action: $w_t = (s_t, s_{t+1}, a_t, r_t)$. In CAPES, we store system status and actions in two tables that are indexed by $t$ in the Replay Database. CAPES uses this algorithm to construct a minibatch for training, which is shown in Algorithm 1.

CAPES checks that the Replay DB contains enough data for each sampled timestamp.

### 3.6 Exploration Period

As we have stated in the background section, it is important for the agent to experience as many states as possible during the training process. The initial training period uses a standard $\epsilon$-greedy policy, in which the tuning agent takes the estimated optimal action with probability $1 - \epsilon$, and randomly picks an action for the other cases. We let $\epsilon$ to anneal linearly from 1.0 to 0.05 (100% to 5%) during the training period. $\epsilon$ here is an example of hyperparameter. Additionally, the Interface Daemon has a controlling program that has access to the scheduling of the workload. Whenever a new workload is started on the system, the Interface Daemon notifies the DRL Engine to bump up $\epsilon$ to 0.2 (or 20% of random actions) so that the tuning agent can do some exploration while avoiding local maximums.

**Algorithm 1** Constructing a minibatch of size $n$ from data in the Replay DB.

---
1: **procedure** CONSTRUCTMINIBATCH($n$)
2:     $samplesNeeded \leftarrow n$
3:     **while** True **do**
4:         Uniformly generate $samplesNeeded$ timestamps
5:         **for** each timestamp $t_i$ **do**
6:             **if** Replay DB contains enough data at $t_i$ **then**
7:                 Get $s_t, s_{t+1}, a_t$ from Replay DB
8:                 $r_i \leftarrow$ CalcReward($s_t, s_{t+1}$)
9:                 $W+ = (s_t, s_{t+1}, a_t, r_t)$
10:            **end if**
11:        **end for**
12:        **if** $W$ has $n$ samples **then return** $W$
13:        **end if**
14:        $samplesNeeded \leftarrow n - \text{len}(W)$
15:    **end while**
16: **end procedure**
---

## 3.7   Performing Actions

Actions dictate what a target system's parameters should be, and CAPES can tune many parameters at the same time. At a fixed rate (every action tick), CAPES decides on an action that either increases or decreases one parameter by a step size. The valid range and tuning step size are customizable for each target system. For instance, one can say that we need to tune the I/O size, which has a valid range from 1 KB to 256 KB, and a tuning step size of 1 KB. We also include a NULL action that performs no action for a step. The DNN can choose to do the NULL action if it sees no need to change any parameter. Thus, the total number of actions we are training the DNN for is

$$2 \times \text{number\_of\_tunable\_parameters} + 1.$$

The same observation data format is used in both training and action steps. The DRL Engine always uses the observation of the current $t$ to calculate the candidate action. Before broadcast, the Interface Daemon will call an Action checker to rule out egregiously bad actions, such as setting the CPU clock rate to 0. This step is optional, and we have not used it in our evaluations, but if there are known bad parameter values, they can be shielded from the target system. We do acknowledge that this adds an extra step for the user of a real CAPES system to define what a bad action is prior to running CAPES, however we consider it reasonable that the user has some general knowledge of what the system should never do. The Interface Daemon then determines which Action Message should be sent to which Control Agent. A Control Agent will listen for inbound Action Messages from the Interface Daemon and will change the system parameters accordingly.

## 4   IMPLEMENTATION AND EVALUATION

We chose the Lustre file system as the target system because it is a high performance distributed file system that can distribute I/O requests from every node to many servers in parallel. It can also generate a huge amount of I/O to stress the system. The purpose of the evaluation is to test whether CAPES can improve the throughput of the workload during peak times and to understand its effectiveness on a variety of workloads.

## 4.1   Implementation

We implemented a CAPES prototype to evaluate this design. The majority of the system is written in Python, with the DNN implemented using Google TensorFlow [1]. We carefully profiled all code and optimized all hotspots to ensure minimal resource use of the Monitoring and Control Agent, in order to maximize the training speed. The Replay DB is a SQLite database using Write-Ahead-Logging for optimal concurrent write/read performance. The whole system has about 6,000 lines of code.

Each Lustre client maintains one Object Storage Client (OSC) for a server it talks to. We have four servers, and are using stripe count of four so each client has four OSCs. Each OSC's Performance Indicators are calculated independently. We collect the following Performance Indicators:

(1) max_rpc_in_flight: Lustre congestion window size.
(2) Read throughput.
(3) Write throughput.
(4) Dirty bytes in write cache.
(5) Maximum size of write cache.
(6) Ping latency from each client to each server.
(7) Ack EWMA: exponentially weighted moving average (EWMA) of gaps between server replies.
(8) Send EWMA: EWMA of gaps between the original sent times of the corresponding requests of the replies received by the client.
(9) Process Time (PT) ratio: current Process Time / shortest Process Time seen so far. Process Time is the time needed by the server to process one I/O request.

The last three indicators are secondary indicators. They were well known to reflect the congestion state of networks and distributed storage systems [17] so we patched the Lustre client to track them.

We tune the following two parameters of each Lustre Object Storage Client. All clients use the same parameter values for all connections.

(1) max_rpc_in_flight: Lustre congestion window size.
(2) I/O rate limit: how many outgoing I/O requests are allowed per second.

## 4.2   System Setup

The evaluation system contains four dedicated servers and five dedicated clients. All nodes use the same hardware: an Intel Xeon CPU E3-1230 V2 @ 3.30 GHz, 16 GB RAM, and one Intel 330 SSD for the OS. The network is gigabit ethernet with measured peak aggregated throughput of ~500 MB/s. Each storage server node uses one 7200 RPM HGST Travelstar Z7K500 hard drive, of which raw I/O performance is measured at 113 MB/s for sequential read and 106 MB/s for sequential write. We used Lustre's default stripe count of four and 1 MB stripe size. No workload is memory intensive, so all server and clients nodes have plenty of memory for buffering and running worker threads. The cache policies of read and write are both Lustre default – write cache is write-through; the server replies a write completion when data hits the disk. We specifically picked this storage and network hardware so the whole system has a 1:1 network to storage bandwidth ratio, matching other larger supercomputers [7], in order to study and test CAPES on a system

**Table 1: List of hyperparameters and their values used in CAPES evaluation**

| Hyperparameter | Value | Description |
|---|---|---|
| action tick length | 1 | One action is performed every second. |
| $\epsilon$ initial value | 1 | Initial value of $\epsilon$ (100% actions are random at the beginning of training). |
| $\epsilon$ final value | 0.05 | Final value of $\epsilon$ (5% actions are random after the training process). |
| discount rate ($\gamma$) | 0.99 | The discount rate as used in Equation 1. |
| hidden layer size | 600 | 8 + 4 PIs per client ×5 clients ×10 ticks per observation. |
| initial exploration period | 2 h | The duration of which the initial value of $\epsilon$ (random action percentage) is linearly annealed to the final value. |
| minibatch size | 32 | Number of observations over which each stochastic gradient descent update is performed. |
| missing entry tolerance | 20% | For each observation, as much as 20% missing data is tolerated. |
| number of hidden layers | 2 | The number of hidden layers beside the input and output layers. The size of the hidden layers is the same as the input. |
| Adam learning rate | 0.0001 | The learning rate of Adam. |
| sampling tick length | 1 s | One sample is taken every second. |
| sampling ticks per observation | 10 | The number of sampling ticks to be included in one Observation. This effectively pack 10 seconds information leading to $t$ into one observation. |
| target network update rate ($\alpha$) | 0.01 | For each minibatch, the target network's $\theta_i^-$ is updated as $\theta_i^- = \theta_i^- \times (1 - \alpha) + \theta_i \times \alpha$. |

that mimics typical real world resource-constrained environments. The storage workloads were generated using Filebench [25] running on all clients in parallel.

The Monitoring and Control Agents only run on Lustre clients, and we do not tune anything on the server node for this prototype. All other components of CAPES run on another dedicated node. Our CAPES node has an Intel Xeon CPU E5-2637 @ 3.00 GHz, 128 GB RAM, an SSD RAID, and one nVidia GTX 1080 GPU.

It is worth noting that the whole evaluation system is not located on an isolated network due to the IT requirements of our department, and we have observed network traffic interference from time to time, such as the routine network scanning of the IT department and machine status queries from the cluster monitoring system. We did not isolate the whole system because we consider this kind of noise as beneficial to the evaluation, because more noise makes the training and tuning process challenging, and a tuning system works only within a perfect environment is not pragmatically interesting.

The hyperparameters used in the evaluation are listed in Table 1. We chose those values after several informal trials, so it is conceivable that better tuning results and/or a shorter training duration can be achieved by using better values. It is within our future work to perform a systematic search on these hyperparameters.

## 4.3 Evaluation Workloads and Performance Increase

We evaluated the following synthetic workloads:
- Random read and write with various read to write ratios: $9:1, 4:1, 1:1, 1:4, 1:9$;
- Filebench file server; and
- Filebench five-stream concurrent sequential write.

*Random read and write workloads.* In these random read and write workloads, each client has five threads doing the same random read and write with a fixed ratio. We have evaluated various
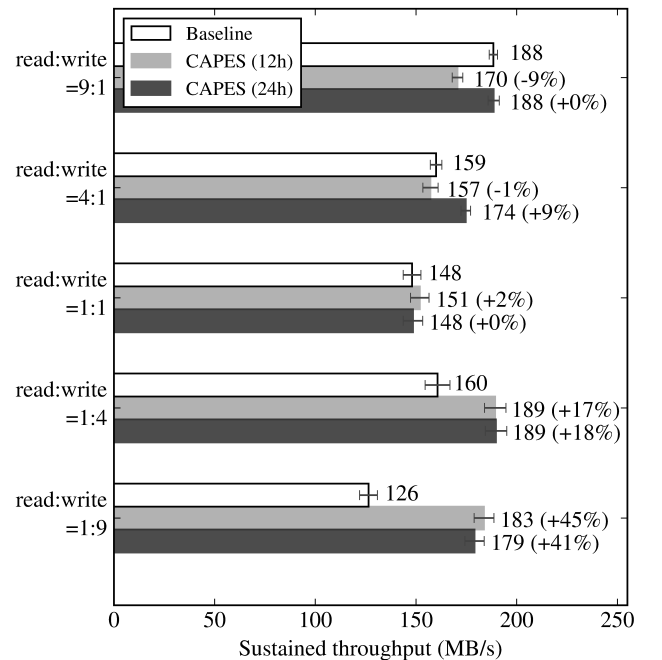


**Figure 2: Overview of random read write workloads evaluated with CAPES. Throughput before, after 12 hours training, and after 24 hours training are shown. Baseline uses default Lustre settings. Error bars show 95% confidence intervals.**

different read to write ratios to mimic a broad range of real applications. We conducted training processes of 12 and 24 hours with the goal of optimizing the aggregated read/write throughput. After training, we evaluated the effects of CAPES's tuning.

It can be seen in Figure 2 that CAPES works best with workloads that are dominated by writes; it increased the performance of the workload with 1:9 read:write ratio by 45%. CAPES did not show obvious effect on read-heavy workloads. This is expected because tuning the number of allowed outstanding I/O requests (congestion window size) of Lustre does have a bigger impact on write than read. The evaluation used storage servers that have hard disk drives as the underlying storage device, which need to spend a majority of I/O time doing seeks for random reads and would not be affected much by the number of outstanding read requests. In contrast, outstanding random write requests can be merged and handled more efficiently if there are more requests in the I/O queue, thus tuning the number of allowed outstanding write requests has a bigger impact on the efficiency of the merge, and in turn the performance.

We also measured the performance after different training duration to understand how long the training sessions needed to be. We can see that training for 24 hours had slightly better results than training for 12 hours only for read-heavy workloads, and had little effect on other workloads. This is likely due to that changing the congest window size has a non-obvious effect on the read performance, and that small changes in the read performance cannot be easily discerned from noise. Therefore, it is understandable that the training would need a longer duration to converge.
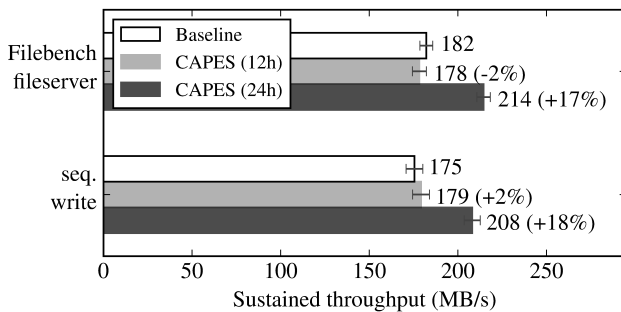


**Figure 3: Overview of Filebench file server and sequential write workload evaluated with CAPES. Throughput before and after CAPES tuning are shown. Baseline uses default Lustre settings. Error bars show 95% confidence intervals.**

*Filebench file server workload.* In addition to the random read write workloads, we have also evaluated the Filebench file server and a sequential write workload, as shown in Figure 3. Filebench file server is a synthetic workload that simulates the I/O pattern that is usually seen on busy file servers, which is one of the most common and important workloads among data centers and enterprise storage servers. Each instance of the workload includes read, write, and metadata operations. It loops through the following I/O operations using a prepopulated set of files:

(1) Create a file and write the file to 100 MB,
(2) Open another file and append random sized data (mean at 100 MB),
(3) Open a randomly picked file and read 100 MB,
(4) Delete a random file, and
(5) Stat a random file.

Each node ran 32 instances (160 instances in total for five nodes) that simulates I/O-bound applications that are competing with each other for the file server. They generated enough traffic to saturate the server nodes.

The second workload is the sequential write workload, which has five sequential write instances on each client (25 instances in total). Each instance does sequential write with 1 MB write size. This benchmark simulates both HPC checkpoint and video survellance workloads. Both the file server and sequential write workloads measure the aggregated throughput of all instances.

We observed that 12 hours training is not enough to find the optimal policy for optimizing the file server workload. The file server workload is especially challenging for Q-learning because, unlike other random read/write workloads, it involves a wide range of read, write, and metadata operations. This inevitably introduces more noise into the measurement process: the aggregated throughput has more fluctuations, and, from CAPES perspective, a good action might not lead to a higher throughput every time, and the delay between action and reward varies too due to different types of operations involved. It required about 24 hours of training to converge to a good policy that can lead to 17% increase in throughput.

Some existing parameter optimization and congestion control systems suffer from the overfitting problem: the effectiveness of the trained model diminishes quickly when there are changes to seemingly related properties of the workload, such as on-disk data location, file fragmentation, allocation of files among servers, and the amount of free space. To test if our trained DNNs also suffer from overfitting, we tested a DNN in three sessions that were spread out over two weeks, with numerous unrelated file operations between the sessions. Each session is four hours long, including two hours for measuring the baseline throughput (using default parameter values without tuning) and two hours for tuned throughput. The results are shown in Figure 4. The CAPES DNN has increased the throughput of all three sessions by from 13% to 36%. Rigorous statistical checks have been done using the Pilot tool [16]: throughput was measured every second, autocorrelation of the samples are checked to ensure they are independent and identically distributed and not temporally correlated, and confidence intervals are calculated at 95% confidence level. The results show that there is no obvious overfitting problem.

### 4.4 Training Efficiency

Figure 5 shows how the prediction error changes over time during the whole training process. The prediction error shows the difference between the DNN's predicted performance and the real performance. It is an important metric of training efficiency: the lower prediction error it gets, the better the DNN can know which action to take to get a desired performance boost. We can see that the prediction error decreases steadily as the training session continues after an initial warm up period.

### 4.5 Training Session's Impact on the Workload

The training session includes carrying out random actions on the target system, therefore it is important to understand the training's impact on the target system's performance. Because we used an $\epsilon$-greedy policy that anneals from 100% random action to 5% action,

**Table 2: List of technical measurements of the CAPES evaluation (9 Monitoring Agents in Total)**

| Measurement | Value | Description |
|---|---|---|
| duration of training step (CPU) | $\approx 0.1$ s | One training step of a 32-observation minibatch on CPU. |
| duration of training step (GPU) | $\approx 0.01$ s | One training step of a 32-observation minibatch on GPU. |
| number of records of the Replay DB | 250 k | One record per second. 70 hours in total. |
| size of the DNN model | 84 MB | The size of the deep neural network in memory. |
| total size of the Replay DB on disk | 0.5 GB | The size of the SQLite database on disk (no compression). |
| total size of the Replay DB in memory | 1.5 GB | The size of the whole Replay DB in memory when being used by the training session. |
| performance indicators per client | 44 | Every client collects this many performance indicators per second (float numbers). |
| observation size | 1760 | One observation contains this many float numbers. |
| average message size per client | $\approx 186$ B | Every second one client sends out about this many bytes to the Interface Daemon. This is the compressed size of all 44 performance indicators. |
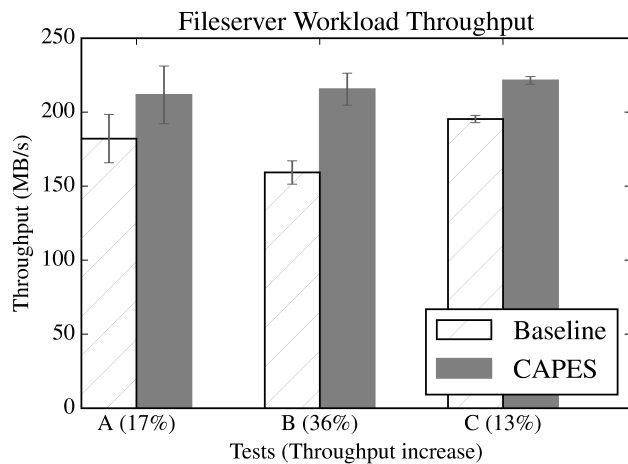


Figure 4: Fileserver workload throughput with and without CAPES tuning. Baseline uses default Lustre settings. Error bars show the confidence interval at 95% confidence level.
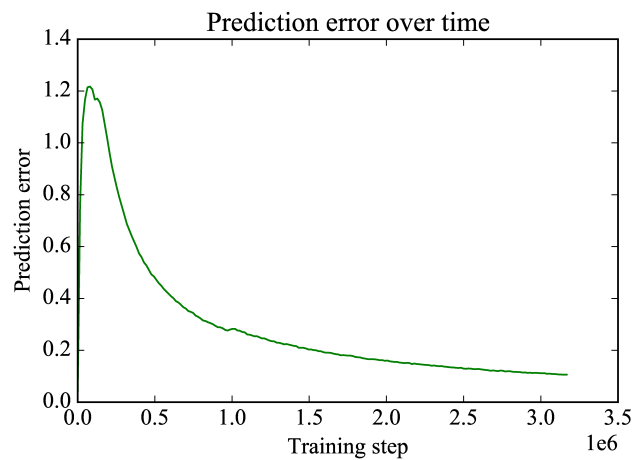


Figure 5: Predicted error during the training process. Prediction error is the different between the neural network's predicted performance after observing the system's status and the actual system performance one second later. The prediction reflects how "well" CAPES understands the target system, and a lower prediction error leads to better tuning results.

the DNN should be able to "mitigate" the impact of the suboptimal random actions when it has a chance to deliver a calculated action, except for the beginning of the training session. Figure 6 confirms this speculation and shows that the overall throughput of a 70-hour training session is comparable to the three baseline throughputs we measured at three different times.

### 4.6 Other Measurements

We provide other related measurements we have collected during the evaluation process in Table 2. They are useful for understanding the computational cost of CAPES for planning to build a trainer for a larger system. It can be seen that the messages sent out by the Monitoring Agents used a small amount of network traffic, and the Replay DB could be easily stored in a modern computer's memory. Using a GPU can achieve a 10 fold increase in training performance comparing to CPU.

## 5 RELATED WORK

Parameter optimization is a challenging research question. The optimal values of parameters can be affected by every aspect of the workloads and the system, such as the I/O request size, randomness, and network topology. Different software versions can also have different quirks, causing their performance to vary. Existing solutions can be classified by whether a model is required and whether the tuning is a one-time process or a continuous process that can be used in production.

Feedback control theory is commonly used in model-based approaches and are often combined with slow-start, fast fallback heuristics [9, 27, 32]. There are other more complex models as well [14, 29]. Model-based approaches work well when the system and workloads are relatively simple and well understood. Most of these solutions still require the administrator to choose values

for critical parameters. For instance, if the start is too slow or the falling back is too fast, the system's capacity is wasted; if the speed increases too fast or the falling back is not fast enough, the system becomes unstable under peak workloads.

Model-less, general purpose approaches usually treat the target system as a black box with knobs and adopt a certain search algorithm, such as hill climbing or evolutionary algorithms [13, 24, 30]. These search-based solutions are often designed as a one-time process to find the optimal parameter values for a certain workload running on a certain system. The search process usually requires a simulator, a small test system, or the target system to be in a controlled environment where the user workload can be repeated again and again, testing different parameter values. ASCAR [17] directly tunes the whole target system and can automatically find optimal traffic control policies to improve peak hour performance. Most of these search methods are a one-time process: if the status of the target system or workloads do not match what the optimizer saw during the bootstrap tuning process, it would fail to improve the system. This inflexibility limited their use in real world environments. There are also domain specific solutions that tunes the parameters of a certain application [3, 11, 28].

The efficiency of search-based algorithms depends on the size of the parameter space, and many of them suffer from overfitting because search algorithms do not provide generalization; when the system or workload changes, the search process needs to be redone. Zhang et al. proposed a method that used neural network to accelerate a traditional search method and to add a certain degree of generalization [31]. Chen et. al. created an early attempt at using neural network-based reinforcement learning to tune a single server, however it's tuning was limited to that single server. [8] CAPES is a more complete system that works on a larger scale, and has taken advantage of the recent rapid progress of deep learning techniques.

There are other optimization solutions that change the architecture of the system automatically, like Hippodrome [2]. They require intrusive and radical modifications to the whole system.
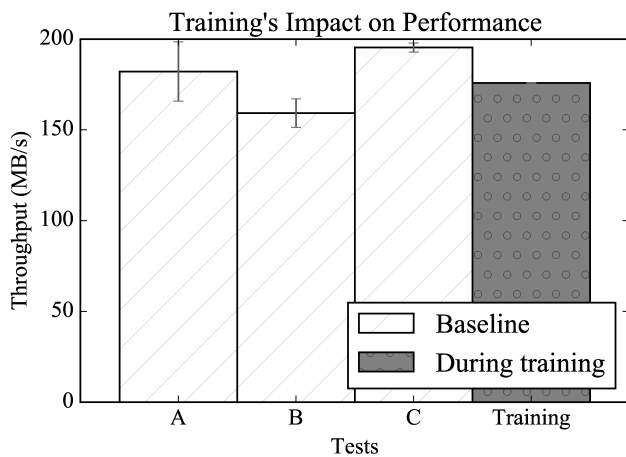


**Figure 6: Baseline throughputs and training session overall throughput. Error bars show the confidence interval at 95% confidence level.**

There are also tools such as [33] that can manage parameters of a large number of nodes. CAPES can work in tandem with such systems to achieve more comprehensive coverage of performance optimization in addition to parameter tuning.

## 6 CONCLUSION AND FUTURE WORK

CAPES is capable of finding optimal values for the congestion window size and I/O rate limit of a distributed storage system in a noisy environment. The optimal values reduces peak time congestion and increases overall throughput by up to 45% in different heavy mixed read/write/metadata workload tests. Compared to manual parameter tuning, CAPES is superior in that it does not require to be supervised, it does not require prior knowledge of the system, it can always run during normal operations, and it can dynamically change parameters. We maintain that automated tuning system could play an important role for future complex distributed systems, such as data centers and supercomputers, to both reduce management costs and increase performance.

The design is general purpose and does not assume anything except that a target system has parameters that can be tuned during run time. With an early prototype, we have demonstrated that it can tune a Lustre file system with minimal human intervention. Theoretically, CAPES can work with a wide range of complex systems, and we plan to evaluate it on more systems in production environments.

DNN-based reinforcement learning does have a disadvantage in that it can be difficult to explain how the trained model works. Usually this is not a compelling problem for performance tuning problems, but can be problematic if the target system is mission critical and suboptimal actions need to be absolutely avoided. That is why we introduced the action checker component (see Figure 1). New deep learning techniques are being invented on an almost daily basis and sometimes can greatly increase the training efficiency. These new techniques, such batch normalization and continuous Deep Q learning [18], need be systematically evaluated and added to CAPES to make it more intelligent and generate better results. We will also need to use a systematic approach to hyperparameter optimization, such as using grid search.

On the Lustre-specific evaluation system, there are many more things can be done. For instance, we can collect information from server nodes in addition to client nodes. It is also possible to tune for two performance indices, such as throughput and latency, at the same time. More performance indices can be merged into a single reward score using an objective function [17]. We can also tune more parameters in addition to the congestion window size and a hard rate limit; DNN is known to be quite effective at handling 20 or more candidate actions [21], which maps to at least 10 tunable parameters.

CAPES needs to be evaluated on larger systems with more features, more parameters, and/or more nodes. There should be no need to do manual feature selection for PIs or to change the structure of the DNN, because DNNs are good at filtering through raw input data [4, 21]. Increasing the size of the network alone should be enough to scale up CAPES considerably.

It also would be interesting to compare CAPES' best results with the best results from other automatic tuning methods. To

further promote research on this topic, we released CAPES and our modified Lustre system at https://github.com/mlogic/capes-oss and https://github.com/mlogic/ascar-lustre-2.9-client.

## REFERENCES

[1] Martín Abadi, Paul Barham, Jianmin Chen, Zhifeng Chen, Andy Davis, Jeffrey Dean, Matthieu Devin, Sanjay Ghemawat, Geoffrey Irving, Michael Isard, Manjunath Kudlur, Josh Levenberg, Rajat Monga, Sherry Moore, Derek G. Murray, Benoit Steiner, Paul Tucker, Vijay Vasudevan, Pete Warden, Martin Wicke, Yuan Yu, and Xiaoqiang Zheng. 2016. TensorFlow: A System for Large-Scale Machine Learning. In *Proceedings of the 12th Symposium on Operating Systems Design and Implementation (OSDI '16)*. USENIX Association, Savannah, GA.
[2] Eric Anderson, Michael Hobbs, Kimberly Keeton, Susan Spence, Mustafa Uysal, and Alistair Veitch. 2002. Hippodrome: running circles around storage administration. In *Proceedings of the Conference on File and Storage Technologies (FAST)*. Monterey, CA. http://www.ssrc.ucsc.edu/PaperArchive/anderson-fast02.pdf
[3] Mona Attariyan, Michael Chow, and Jason Flinn. 2012. X-ray: Automating Root-cause Diagnosis of Performance Anomalies in Production Software. In *Proceedings of the 10th USENIX Conference on Operating Systems Design and Implementation (OSDI'12)*. USENIX Association, Berkeley, CA, USA, 307–320. http://dl.acm.org/citation.cfm?id=2387880.2387910
[4] Yoshua Bengio. 2009. Learning Deep Architectures for AI. *Foundations and Trends® in Machine Learning* 2, 1 (Jan. 2009), 1–127. DOI: https://doi.org/10.1561/2200000006
[5] Daniel S. Bernstein, Robert Givan, Neil Immerman, and Shlomo Zilberstein. 2002. The complexity of decentralized control of Markov decision processes. *Mathematics of Operations Research* 27, 4 (Nov. 2002), 819–840. DOI: https://doi.org/10.1287/moor.27.4.819.297
[6] Christopher M. Bishop. 2007. *Pattern Recognition and Machine Learning* (1st ed.). Springer.
[7] Julian Borrill, L. Oliker, J. Shalf, and Hongzhang Shan. 2007. Investigation of leading HPC I/O performance using a scientific-application derived benchmark. In *Proceedings of SC07*. 1–12. DOI: https://doi.org/10.1145/1362622.1362636
[8] Haifeng Chen, Guofei Jiang, Hui Zhang, and Kenji Yoshihira. 2009. Boosting the Performance of Computing Systems Through Adaptive Configuration Tuning. In *Proceedings of the 2009 ACM Symposium on Applied Computing (SAC '09)*. ACM, New York, NY, USA, 1045–1049. DOI: https://doi.org/10.1145/1529282.1529511
[9] Y. Diao, J. L. Hellerstein, S. Parekh, and J. P. Bigus. 2003. Managing Web Server Performance with AutoTune Agents. *IBM Systems Journal* 42, 1 (Jan. 2003), 136–149. DOI: https://doi.org/10.1147/SJ.2003.5386833
[10] Katharina Eggensperger, Matthias Feurer, Frank Hutter, James Bergstra, Jasper Snoek, Holger Hoos, and Kevin Leyton-Brown. 2013. Towards an empirical foundation for assessing bayesian optimization of hyperparameters. In *NIPS workshop on Bayesian Optimization in Theory and Practice*. 1–5.
[11] Adem Efe Gencer, David Bindel, Emin Gün Sirer, and Robbert van Renesse. 2015. Configuring Distributed Computations Using Response Surfaces. In *Proceedings of the 16th Annual Middleware Conference (Middleware '15) (Middleware '15)*. ACM, New York, NY, USA, 235–246. DOI: https://doi.org/10.1145/2814576.2814730
[12] Kurt Hornik. 1991. Approximation Capabilities of Multilayer Feedforward Networks. *Neural Network* 4, 2 (March 1991), 251–257. DOI: https://doi.org/10.1016/0893-6080(91)90009-T
[13] Pooyan Jamshidi and Giuliano Casale. 2016. An Uncertainty-Aware Approach to Optimal Configuration of Stream Processing Systems. In *Proceedings of the 24th International Symposium on Modeling, Analysis, and Simulation of Computer and Telecommunication Systems (MASCOTS '16)*.
[14] Magnus Karlsson, Christos Karamanolis, and Xiaoyun Zhu. 2005. Triage: Performance Differentiation for Storage Systems Using Adaptive Control. *ACM Transactions on Storage* 1, 4 (2005), 457–480. http://www.ssrc.ucsc.edu/PaperArchive/karlsson-tos05.pdf
[15] Diederik P. Kingma and Jimmy Ba. 2015. Adam: A Method for Stochastic Optimization. (2015). arXiv:cs.LG/1412.6980
[16] Yan Li, Yash Gupta, Ethan L. Miller, and Darrell D. E. Long. 2016. Pilot: A Framework that Understands How to Do Performance Benchmarks the Right Way. In *Proceedings of the 24th International Symposium on Modeling, Analysis, and Simulation of Computer and Telecommunication Systems (MASCOTS '16)*.

[17] Yan Li, Xiaoyuan Lu, Ethan L. Miller, and Darrell D. E. Long. 2015. ASCAR: Automating Contention Management for High-Performance Storage Systems. In *Proceedings of the 31th IEEE Conference on Mass Storage Systems and Technologies*.
[18] Timothy Lillicrap, Jonathan Hunt, Alexander Pritzel, Nicolas Heess, Tom Erez, Yuval Tassa, David Silver, and Daan Wierstra. 2016. Continuous control with deep reinforcement learning. (2016). arXiv:cs.LG/1509.02971
[19] Long-Ji Lin. 1993. *Reinforcement learning for robots using neural networks.* Technical Report. DTIC Document.
[20] Volodymyr Mnih, Koray Kavukcuoglu, David Silver, Alex Graves, Ioannis Antonoglou, Daan Wierstra, and Martin Riedmiller. 2013. Playing atari with deep reinforcement learning. *arXiv preprint arXiv:1312.5602* (2013).
[21] Volodymyr Mnih, Koray Kavukcuoglu, David Silver, Andrei A. Rusu, Joel Veness, Marc G. Bellemare, Alex Graves, Martin Riedmiller, Andreas K. Fidjeland, Georg Ostrovski, Stig Petersen, Charles Beattie, Amir Sadik, Ioannis Antonoglou, Helen King, Dharshan Kumaran, Daan Wierstra, Shane Legg, and Demis Hassabis. 2015. Human-level control through deep reinforcement learning. *Nature* 518, 7540 (26 02 2015), 529–533. http://dx.doi.org/10.1038/nature14236
[22] Open Scalable File Systems, Inc. 2014. The Lustre® file system. http://www.opensfs.org/. (2014).
[23] Sebastian Ruder. 2017. An overview of gradient descent optimization algorithms. http://sebastianruder.com/optimizing-gradient-descent/. (2017).
[24] A. Saboori, G. Jiang, and H. Chen. 2008. Autotuning Configurations in Distributed Systems for Performance Improvements Using Evolutionary Strategies. In *The 28th International Conference on Distributed Computing Systems (ICDCS '08)*. 769–776. DOI: https://doi.org/10.1109/ICDCS.2008.11
[25] SUN Microsystems, File system and Storage Lab (FSL) at Stony Brook University, and Other Contributors. 2016. Filebench. https://github.com/filebench/filebench. (2016).
[26] Richard S. Sutton and Andrew G. Barto. 1998. *Reinforcement Learning: An Introduction.* MIT Press, Cambridge, MA.
[27] Andrew S. Tanenbaum. 2010. *Computer Networks (5th Edition).* Prentice Hall.
[28] K. Wang, X. Lin, and W. Tang. 2012. Predator – An experience guided configuration optimizer for Hadoop MapReduce. In *IEEE 4th International Conference on Cloud Computing Technology and Science (CloudCom '12)*. 419–426. DOI: https://doi.org/10.1109/CloudCom.2012.6427486
[29] Mengzhi Wang, Kinman Au, Anastassia Ailamaki, Anthony Brockwell, Christos Faloutsos, and Gregory R. Ganger. 2004. Storage device performance prediction with CART models. In *Proceedings of the 12th International Symposium on Modeling, Analysis, and Simulation of Computer and Telecommunication Systems (MASCOTS '04)*. 588–595. DOI: https://doi.org/10.1109/MASCOT.2004.1348316
[30] Keith Winstein and Hari Balakrishnan. 2013. TCP ex Machina: computer-generated congestion control. In *Proceedings of the Conference on Applications, Technologies, Architectures, and Protocols for Computer Communication (SIGCOMM '13)*. Hong Kong, China, 123–134.
[31] F. Zhang, J. Cao, L. Liu, and C. Wu. 2011. Performance improvement of distributed systems by autotuning of the configuration parameters. *Tsinghua Science and Technology* 16, 4 (Aug 2011), 440–448. DOI: https://doi.org/10.1016/S1007-0214(11)70063-3
[32] Jianyong Zhang, Anand Sivasubramaniam, Qian Wang, Alma Riska, and Erik Riedel. 2006. Storage Performance Virtualization via Throughput and Latency Control. *ACM Transactions on Storage* 2, 3 (Aug. 2006), 283–308. DOI: https://doi.org/10.1145/1168910.1168913
[33] Wei Zheng, Ricardo Bianchini, and Thu D. Nguyen. 2007. Automatic Configuration of Internet Services. In *Proceedings of the 2Nd ACM SIGOPS/EuroSys European Conference on Computer Systems 2007 (EuroSys '07)*. ACM, New York, NY, USA, 219–229. DOI: https://doi.org/10.1145/1272996.1273020

## A    ARTIFACT DESCRIPTION

We have released the source code of CAPES under the 3-clause BSD license as an artifact to promote research in related areas.

## A.1    Abstract

This artifact contains the source code of CAPES, a deep reinforcement learning-based parameter tuning system. It can be used to tune virtually any parameters as long as an adapter function is provided for collecting the observation from the target system and for setting the parameters to the target system. A sample adapter function for the Lustre file system is included. Hardware and software requirements are described, and installation instructions are provided.

## A.2 Description

The CAPES system collects a target system's states (observation, also called Performance Indicators, or PIs) and performance measurements (rewards), and uses Deep Q-Learning to train a deep neural network (DNN) for generating parameter values that can be use to increase the target system's performance. CAPES is designed to be non-invasive and can work with most existing distributed systems. The current prototype includes an adapter for working with the Lustre distributed file system.

### A.2.1 Check-list (artifact meta information).
- **Algorithm: CAPES**
- **Compilation: No need**
- **Run-time environment: Red Hat Enterprise Linux/CentOS 7, Python 3.5**
- **Publicly available?: Yes**

### A.2.2 How software can be obtained.
The artifact can be download from the git repository https://github.com/mlogic/capes-oss.

### A.2.3 Hardware dependencies.
CAPES is designed to be flexible and can work with both small and large target systems. Here we describe the hardware dependencies of the CAPES system, and it should not be confused with the target system's hardware requirement. For instance, it is possible to run CAPES on a powerful server to tune one or more small, embedded systems.

CAPES consists of several daemons:
- Interface Daemon for receiving, storing, and aggregating observations from the Monitoring Agent.
- Deep Q-Learning (DQL) Daemon for running the core neural network algorithm.
- Monitoring Agent for collecting observation and setting parameter valuse for the target system.

The Interface Daemon and the DQL Daemon can be deployed to any computer node in or out of the target system. But it is recommended to deploy them to one or more dedicated nodes to avoid negatively impacting the performance of the target system. It is usually adequate to deploy them on the same node for evaluation purpose. For large target systems, different daemons should be deployed on different nodes for better scalability.

The minimal hardware requirement for running the Interface Daemon and the DQL Daemon is the same as the hardware requirement of Red Hat Enterprise Linux(RHEL)/CentOS 7 (https://access.redhat.com/articles/rhel-limits) and Python 3.5. The default setting of the DQL daemon caches the whole replay database in memory for optimal training performance and requires plenty of RAM. The cache data is stored in a memory-efficient manner using NumPy arrays. The total amount of RAM required depends on the observation size and number of entries in the replay database. The observation size depends on how many frames of data are included in one observation, number of Performance Indicators in each frame, and the number of nodes in the system. For our test cluster that has about 10 nodes, the largest cache size was about 3 GB, so we expect any modern hardware that has no less than 16 GB of RAM should be enough for evaluation purpose.

A GPU that is supported by TensorFlow (https://www.tensorflow.org/install/install_linux#NVIDIARequirements) is recommended for the neural network's training process. For evaluation purpose with a small target system, using GPU is not necessary, but we expect GPUs to be used in any non-trivial deployment.

The Monitoring Agent needs to be be deployed to every node of the target system where observation or setting parameters are needed. The Monitoring Agent does not have special hardware requirement and should run on any systems that support Python 3.5.

### A.2.4 Software dependencies.
CAPES has been developed and tested in Python 3.5 on RHEL/CentOS 7. CAPES should also work on other modern Linux systems and later versions of Python with minimal modification. CAPES also requires TensorFlow v1.0, which can be downloaded from www.tensorflow.org. Other dependencies are listed in `setup.py` in the top level directory.

It is possible to port CAPES to other OSes, such as Windows or macOS, as long as they support Python 3.5 (and TensorFlow for the DQL Daemon only). The major parts that need porting are `_run_as_service.py`, which implements running a Python program as a system service (daemon), and all shell scripts.

The CAPES source code repository includes a Lustre adapter for observing and controlling a Lustre file system. The default setting controls only `max_rpcs_in_flight` and should work with any version of Lustre. If you want to take advantage of the secondary performance indicators as introduced in the paper or choose to enable the controlling of I/O rate limit, you would need to apply our patch to the Lustre 2.9 client. The patch can be found at the git repository: https://github.com/mlogic/ascar-lustre-2.9-client.

## A.3 Installation

CAPES is designed to optimize a target system, so in general you should have the target system deployed before setting up CAPES. The following instructions assume that you already have a working Lustre system. The default set up of CAPES needs a Monitoring Agent running on each client and does not require changes to the server. It is also possible to evaluate CAPES using only a subset of clients of the cluster.

Python 3.5 need to be set up first. Python 3.5 can be easily installed on RHEL/CentOS 7 from either the Red Hat Software Collections www.softwarecollections.org or the IUS https://ius.io/ repository. Please refer to TensorFlow's website for methods to install TensorFlow.

For simplicity, all daemons (Interface Daemons, DQL Daemon, and Monitoring/Controlling Agents) share the same source code tree. After deploying the CAPES source code to the node, run:

```
python setup.py install
```

to install Python dependencies.

The final step of installation is configuring CAPES. All CAPES configuration settings are in the file `conf.py` in the top level directory. It includes explanation and sample default values. Each node can have a different `conf.py` file if needed, but for simplicity you can share the same `conf.py` among all CAPES services.

All parameters and their valid ranges of values must be set correctly in `conf.py`. Read the comments and samples in `conf.py` for detailed instructions.

### A.3.1 Interface Node.
The Interface Node runs the Interface Daemon, which receives, stores, and aggregates observations from

all Monitoring Agents. The IP address of the Interface Node and the database connection information should be set in `conf.py`. The Interface Daemon does not have extra Python dependencies and can be started and stopped by running:

```
intfdaemon_service.sh conf.py {start|stop}
```

*A.3.2 DQL Node.* The DQL Node runs the Deep Q-Learning Daemon and requires TensorFlow. The installation instruction for TensorFlow can be found at https://www.tensorflow.org/install/. The DQL Daemon can be started and stopped by running:

```
dqldaemon_service.sh conf.py {start|stop}
```

*A.3.3 Client Nodes.* The client nodes read the IP address of the Interface Node from `conf.py`. You will also need to provide a collector function for collecting observation and a controller function for setting the parameter values if you are not using the provided Lustre adapter. These two functions are Python functions that can be defined anywhere and imported in `conf.py`, which includes function interface declarations and samples. The Monitoring Agent Daemon can be started and stopped by running:

```
ma_service.sh conf.py {start|stop}
```

Many clusters have existing telemetry function for diagnose and monitoring purposes, and as long as they can provide high resolution state information of the system (observing the state at a frequency no smaller than 1 Hz is recommended), CAPES can take advantage of these existing information. In these cases, you will need to develop adapters to channel the existing information into the Interface Daemon.

*A.3.4 Debugging.* CAPES can output detailed debugging information for monitoring the status of the system. The location of log files and debugging level are also controlled by settings in `conf.py`.

## A.4 Experiment workflow

Evaluation of CAPES usually involves plugging CAPES into the target system, letting CAPES run for at least 12 hours, and measuring the change in performance. We believe it is generally safe to evaluate CAPES on a production system provided that bad parameter values (or combinations) are excluded in the settings. For instance, we knew that the `max_rpcs_in_flight` (RPC congestion window size for Lustre) should not be smaller than eight, then the valid range for the congestion window should start from nine. For extra safety, CAPES can be started at the least busy hour of a production system, such as at night.

Existing performance monitoring mechanisms can be used to monitor the change in system performance before and after CAPES is started. Please refer to the Computational Results Analysis appendix for our analytic method. There is no special training period needed and the performance should increase gradually over time, with notable increase happening usually after 12 hours. The setting of exploration period can also affect the system's performance in the first few hours after CAPES is plugged in and can be set in `conf.py`.

For optimal performance, you can also make use of the job information if the workloads on the target system change by schedule. For instance, if at any given moment only one kind of workload

is running on the system, we can use different CAPES session for each different workload so that they are trained and tuned respectively. CAPES automatically checkpoints and stores the trained model when being stopped, and loads the saved model when being started next time, and different sessions can use different saved model file names and replay database locations, which are set in the `conf.py` file. Therefore, the controlling of the Interface Daemon and DQL Daemon can be added to the job scheduler to synchronize them with the workload. On the other hand, if there are always many workloads running at the same time on the target system, the mixed I/O workload is usually considered a noisy random read-/write workload, and separating them into sessions usually does not make sense.

For evaluation purpose, the workflow is usually like:
(1) Set up the test environment.
(2) Turn on CAPES and do a training session (12 to 24 hours).
(3) Turn off CAPES and measure the baseline performance.
(4) Turn on CAPES and measure the tuned performance.

## B COMPUTATIONAL RESULTS ANALYSIS

This appendix describes the steps and analytic methods we use to improve the trustworthiness of our benchmark results.

## B.1 Abstract

We take the trustworthiness of our evaluation results seriously and have developed complex methods to improve the accuracy, precision, comparability, and repeatability of the results. We have developed the Pilot benchmark framework to process our results. The Pilot benchmark framework automates statistical validations, and can help to reduce human error and shorten measurement time. All of our evaluation results are provided with confidence interval, which is calculated after all performance measurements are validated to be independent and identically distributed (*i.i.d.*).

## B.2 Results Analysis Discussion

Our results analysis focused on improving the following aspects of the measurement:

- **Accuracy** reflects whether the results actually measure what we want to measure. A benchmark usually involves many components of the system, so when we need to measure a certain property of the system, such as I/O bandwidth, the benchmark needs to be designed in such a way that no other components, such as CPU or RAM, are limiting the measured performance. We measured the usage of related components while the benchmark is running and checked that which component is limiting the measured performance.
- **Precision** is related to accuracy but is a different concept. Precision is the difference between the measured value and the real value we need to measure. In statistical terms, precision is the difference between a sample mean and its corresponding population mean. Precision can be described by confidence interval (CI). The CI of a sample mean describes the range of possible population mean at certain likelihood. For instance, if the CI of a throughput mean ($\mu$) is $C$ at the 95% confidence level, we know that

there is a 95% chance that the system's real mean throughput is within interval $[\mu - \frac{C}{2}, \mu + \frac{C}{2}]$. In practice, CIs are typically stated at the 95% confidence level.

- **Repeatability** is critical to a valid performance measurement because the goal of most performance benchmark is to predict the performance of future workloads, which means that we want the measurement results to be repeatable. In addition to accuracy and precision, random errors in the measurement can have a negative impact on repeatability if not handled properly. Random errors are affected by noise outside our control, and can result in non-repeatable measurements if the sample size is not large enough or samples are not independent and identically distributed (*i.i.d.*).

Knowing the sample mean of a performance measurement without knowing the confidence interval (CI) is not very useful, especially when talking about small performance changes. For instance, we would not be able to know if a throughput mean of CI $150 \pm 50$ MB/s is faster or slower than a throughput mean of CI $180 \pm 5$ MB/s, because the actual mean of the first system can be anywhere between 100 MB/s and 200 MB/s and the actual mean of the second system can be anywhere between 175 MB/s and 185 MB/s. The CI of a mean is critical for comparing system performance, and it must be tight (a small value).

Taking all these aspects into consideration, we took periodic measurement of the I/O performance during the benchmark and calculated the mean and CI of mean using the student's $t$-distribution. A naïve way of applying the student $t$ distribution is to directly calculate the CI using the measured sample. However, one major challenge for applying the student's $t$-distribution is that all measurement samples must be i.i.d., and this aspect is often overlooked in many published results. If the samples are not i.i.d., or in other words, autocorrelated, the calculated CI would be falsely tight.

Validating the i.i.d. of results is related to the question of deciding the optimal sampling frequency during the measurement. If the sampling frequency is too high, the measurement samples would be highly correlated due to the discrete nature of computer algorithms (many computer jobs are handled by schedulers, which allocate time in the unit of slices); if the sampling frequency is too low, it would take a very long time to accumulate enough samples to calculate a CI that is narrow enough.

To solve this problem, we take the measurement every second and calculate the autocorrelation of the samples. Autocorrelation is a good indication of the degree of independence of the samples. If the absolute value of the calculated autocorrelation is higher than 0.1 (autocorrelation has a range of $[-1, 1]$), we use the subsession analysis method to treat the samples. In subsession analysis, adjacent samples in a time series are merged by taking the mean, and this can reduce the autocorrelation of the samples but would also result in fewer samples. This merging process is repeated until the autocorrelation is brought down to below the threshold, and the subsession results are used to calculate the CI. For certain measurements, we have to merge hundreds of adjacent samples before their autocorrelation is low enough. This illustrates the importance of verifying the i.i.d. of samples before applying the student's $t$-distribution.

Another factor that may affect the accuracy and precision of the measurement results is the warm-up and cool-down phases. We want to measure the sustainable I/O performance of a certain workload, but at the start and end of the workloads the performance would have significant fluctuation caused by caching effect. These warm-up and cool-down phases are not what we want to measure and negatively impact the accuracy and precision of our results. We used a changepoint detection algorithm to detect these non-stable phases and removes them from the result calculation.

We implemented these methods in the Pilot benchmark framework [16] and also released it at https://ascar.io/pilot/.

## B.3 Summary

We take the statistical validity and trustworthiness of the results seriously. The bottleneck of the sample is analyzed to make sure the measured I/O performance is accurate. The autocorrelation of the measurement samples is calculated, and subsession analysis is applied to make sure that the samples are i.i.d. before calculating the confidence intervals of all results. The warm-up and cool-down phases of the samples are detected and removed to improve the precision of the results. We have included CI at 95% confidence level for all measurement results in the paper.