On Optimizing Operational Efficiency in Storage Systems via Deep Reinforcement Learning

Sunil Srinivasa[1(✉)], Girish Kathalagiri[1], Julu Subramanyam Varanasi[2], Luis Carlos Quintela[1], Mohamad Charafeddine[1], and Chi-Hoon Lee[1]

[1] Samsung SDS America, 3655 North First Street, San Jose, CA 95134, USA
ssunil@gmail.com, girish.sk@gmail.com,
{l.quintela,mohamad.c,lee.chihoon}@samsung.com
[2] Samsung Semiconductors Inc., 3655 North First Street, San Jose, CA 95134, USA
sv.julu@gmail.com

Abstract. This paper deals with the application of deep reinforcement learning to optimize the operational efficiency of a solid state storage rack. Specifically, we train an on-policy and model-free policy gradient algorithm called the Advantage Actor-Critic (A2C). We deploy a dueling deep network architecture to extract features from the sensor readings off the rack and devise a novel utility function that is used to control the A2C algorithm. Experiments show performance gains greater than 30% over the default policy for deterministic as well as random data workloads.

Keywords: Data center · Storage system · Operational efficiency
Deep reinforcement learning · Actor-critic methods

1 Introduction

The influence of artificial intelligence (AI) continues to proliferate our daily lives at an ever-increasing pace. From personalized recommendations to autonomous vehicle navigation to smart personal assistants to health screening and diagnosis, AI has already proven to be effective on a day-to-day basis. But it can also be effective in tackling some of the world's most challenging control problems – such as minimizing the power usage effectiveness[1] (PUE) in a data center.

From server racks to large deployments, data centers are the backbone to delivering IT services and providing storage, communication, and networking to the growing number of users and businesses. With the emergence of technologies such as distributed cloud computing and social networking, data centers have an even bigger role to play in today's world. In fact, the Cisco® Global Cloud Index,

[1] PUE [1] is defined as the ratio of the total energy used by the data center to the energy delivered towards computation. A PUE of 1.0 is considered ideal.

U. Brefeld et al. (Eds.): ECML PKDD 2018, LNAI 11053, pp. 238–253, 2019.
https://doi.org/10.1007/978-3-030-10997-4_15

an ongoing effort to forecast the growth of data center and cloud-based traffic, estimates that the global IP traffic will grow 3-fold over the next 5 years [2].

Naturally, data centers are sinkholes of energy (required primarily for cooling). While technologies like virtualization and software-based architectures to optimize utilization and management of compute, storage and network resource are constantly advancing [4], there is still a lot of room to improve the utilization of energy on these systems using AI.

1.1 The Problem

In this paper, we tackle an instance of the aforementioned problem, specifically, optimizing the operation of a solid state drive (SSD) storage rack (see Fig. 1). The storage rack comprises 24 SSDs and a thermal management system with 5 cooling fans. It also has 2 100G ethernet ports for data Input/Output (I/O) between the client machines and the storage rack. From data I/O operations off the SSDs, the rack's temperature increases and that requires the fans to be turned on for cooling. Currently, the fan speeds are controlled simply based on a tabular (rule-based) method - thermal sensors throughout the chassis including one for each drive bay record temperatures and the fan speeds are varied based on a table that maps the temperature thresholds to desired fan speeds. In contrast, our solution uses a deep reinforcement learning[2] (DRL)-based control algorithm called the Advantage Actor-Critic to control the fans. Experimental results show significant performance gains over the rule-based current practices.

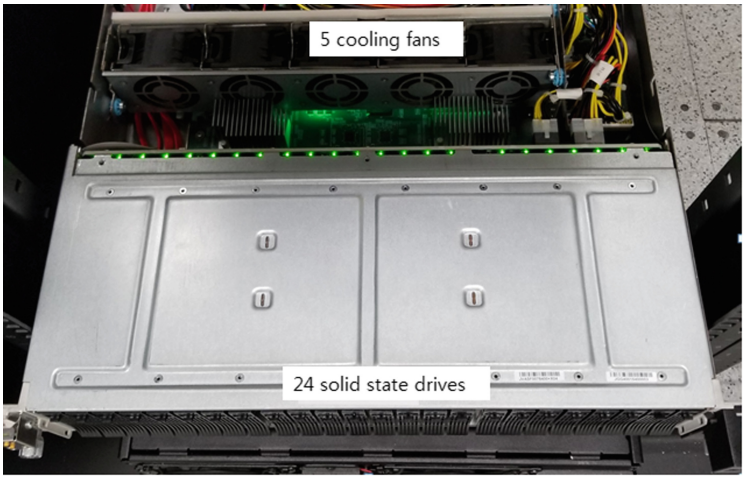

Fig. 1. The SSD storage rack we used for our experiments. It comprises 24 SSDs that are kept cool via 5 fans.

[2] DRL [3] is the latest paradigm for tackling AI problems.

Ideally, if we are able to measure the PUE of the storage rack, that will be the precise function that needs to be minimized. However, the rack did not contain any hooks or sensors for directly measuring energy. As a proxy for the PUE, we instead designed a utility (or reward) function that represents the *operational efficiency* of the rack. Our reward function is explained in detail in Sect. 2.2, and across the space of its arguments, comprises contours of good as well as contours of not-so-good values. The problem now becomes being able to learn a control algorithm that optimizes the operational efficiency of the rack by always driving it towards the reward contours with good values.

1.2 Related Work and Our Contributions

Prior work in the area of developing optimal control algorithms for data centers [4–6] build approximate models to study the effects of thermal, electrical and mechanical subsystem interactions in a data center. These model-based methods are sometimes inadequate and suffer from error propagation, which leads to sub-optimal control policies. Recently, Google DeepMind published a blog [7] on using AI to reduce Google's data centre cooling bill by 40%. In [8], the authors use the deep deterministic policy gradient technique on a simulation platform and achieve a low PUE as well as a 10% reduction in cooling energy costs.

Our novel contributions are three-fold. First, unlike prior model-based approaches, we formulate a model-free method that does not require any knowledge of the SSD server behavior dynamics. Second, we train our DRL algorithm on the real system[3], and do not require a simulator. Finally, since our SSD rack does not have sensors to quantify energy consumption, we devise a reward function that is used not only to quantify the system's operational efficiency, but also as a control signal for training.

2 Our DRL-Based Solution

In this section, we provide the details of our solution to the operation optimization problem. We first introduce the reader to reinforcement learning (RL), and subsequently explain the algorithm and the deep network architecture used for our experiments.

2.1 Reinforcement Learning (RL) Preliminaries

RL is a field of machine learning that deals with how an agent (or algorithm) ought to take actions in an environment (or system) so as to maximize a certain cumulative reward function. It is gaining popularity due to its direct applicability to many practical problems in decision-making, control theory and multi-dimensional optimization. RL problems are often modeled as a Markov Decision

[3] In fact, the SSD server rack also has a Intel Xeon processor with 44 cores to facilitate online (in-the-box) training.

Process with the following typical notation. During any time slot t, the environment is described by its *state* notated s_t. The RL agent interacts with the environment in that it observes the state s_t and takes an *action* a_t from some set of actions, according to its *policy* $\pi(a_t|s_t)$ - the policy of an agent (denoted by $\pi(a_t|s_t)$) is a probability density function that maps states to actions, and is indicative of the agent's behavior. In return, the environment provides an immediate *reward* $r_t(s_t, a_t)$ (which is evidently a function of s_t and a_t) and transitions to its next state s_{t+1}. This interaction loops in time until some terminating criterion is met (for example, say, until a time horizon H). The set of states, actions, and rewards the agent obtains while interacting (or rolling-out) with the environment, $\tau =: \{(s_0, a_0, r_0), (s_1, a_1, r_1), \ldots, (s_{H-1}, a_{H-1}, r_{H-1}), s_H\}$ forms a *trajectory*. The cumulative reward observed in a trajectory τ is called the *return*, $\mathcal{R}(\tau) = \sum_{t=0}^{H-1} \gamma^{H-1-t} r_t(s_t, a_t)$, where γ is a factor used to discount rewards over time, $0 \leq \gamma \leq 1$. Figure 2 represents an archetypal setting of a RL problem.

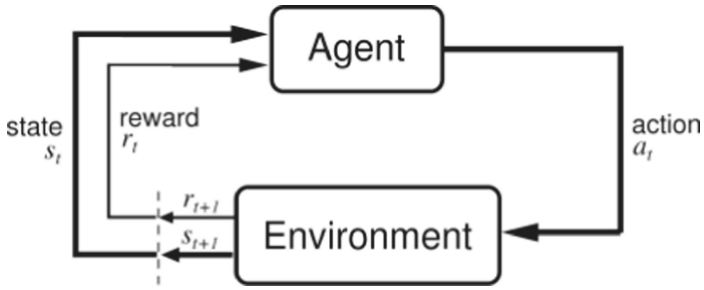

Fig. 2. The classical RL setting. Upon observing the environment state s_t, the agent takes an action a_t. This results in an instantaneous reward r_t, while the environment transitions to its next state s_{t+1}. The objective of the agent is to maximize the cumulative reward over time.

In the above setting, the goal of the agent is to optimize the policy π so as to maximize the expected return $\mathbb{E}_\tau[\mathcal{R}_\tau]$ where the expectation operation is taken across several trajectories. Two functions related to the return are (a) the *action value function* $Q^\pi(s_t, a_t)$, which is the expected return for selecting action a_t in state s_t and following the policy π, and (b) the *state value function* $V^\pi(s_t)$, which measures the expected return from state s_t upon following the policy π. The *advantage* of action a_t in state s_t is then defined as $A^\pi(s_t, a_t) = Q^\pi(s_t, a_t) - V^\pi(s_t)$.

2.2 State, Action and Reward Formulations

In order to employ RL for solving our problem, we need to formulate state, action and reward representations.

State: We use a vector of length 7 for the state representation comprising the following (averaged and further, normalized) scalars:

1. **tps** (transfers per second): the mean number of transfers per second that were issued to the SSDs. A transfer is an I/O request to the device and is of indeterminate size. Multiple logical requests can be combined into a single I/O request to the device.
2. **kB_read_per_sec**: the mean number of kilo bytes read from an SSD per second.
3. **kB_written_per_sec** (writes per second): the mean number of kilo bytes written to an SSD per second.
4. **kb_read**: the mean number of kilo bytes read from an SSD in the previous time slot.
5. **kb_written**: the mean number of kilo bytes written to an SSD in the previous time slot.
6. **temperature**: the mean temperature recorded across the 24 SSD temperature sensors on the rack.
7. **fan_speed**: the mean speed of the 5 cooling fans in revolutions per minute (rpm).

Recall there are 24 SSDs in our rack (and hence 24 different values of transfers per second, bytes read, bytes written, etc.), but we simply use the averaged (over the 24 SSDs) values for our state representation[4].

To obtain 1. through 5. above, we use the linux system command **iostat** [20] that monitors the input/output (I/O) device loading by observing the time the devices are active in relation to their average transfer rates. For 6. and 7., we use the **ipmi-sensors** [21] system command that displays current readings of sensors and sensor data repository information.

For normalizing, 1. through 7., we use the *min-max* strategy. Accordingly, for feature X, an averaged value of $\bar{x}$ is transformed to $\Gamma(\bar{x})$, where

$$\Gamma(\bar{x}) = \frac{\bar{x} - \text{minX}}{\text{maxX} - \text{minX}}, \tag{1}$$

where minX and maxX are the minimum and maximum values set for the feature X. Table 1 lists the minimum and maximum values we used for normalizing the state features, which were chosen based on empirically observed range of values.

Action: The action component of our problem is simply in setting the fan speeds. In order to keep the action space manageable[5], we use the same action

[4] We shall see shortly that we are constrained to use the same control setting (speed) on all the 5 fans. Hence, it is not meaningful to individualize the SSDs based on their slot location inside the server. Using averaged values is practical in this scenario.

[5] The action space grows exponentially with the number of fans, when controlling each fan independently. For example, with just 3 fan speed settings, the number of possible actions with 5 fans becomes $3^5 = 243$. DRL becomes ineffective when handling such a large problem. Instead, if we set all the fan speeds to the same value, the action space dimension reduces to just 5, and is quite manageable. Incidentally, setting separate speeds to the fans is also infeasible from a device driver standpoint - the ipmitool utility can only set all the fans to the same speed.

Table 1. Minimum and maximum values of the various state variables used for the min-max normalization.

Feature X	minX	maxX
tps	0.0	1000.0
kB_read_per_sec	0.0	4000000.0
kB_written_per_sec	0.0	4000000.0
kB_read	0.0	4000000.0
kB_written	0.0	4000000.0
temperature	27.0	60.0
fan_speed	7500.0	16000.0

(i.e., speed-setting) on all the 5 fans. For controlling the fan speeds, we use the **ipmitool** [22] command-line interface. We consider two separate scenarios:

- **raw action:** the action space is discrete with values 0 through 6, where 0 maps to 6000 rpm, while 6 refers to 18000 rpm. Accordingly, only 7 different rpm settings are allowed: 6000 through 18000 in steps of 2000 rpm. Note that consecutive actions can be very different from each other.
- **incremental action:** the action space is discrete taking on 3 values - 0, 1 or 2. An action of 1 indicates no change in the fan speed, while 0 and 2 refer to an decrement or increment of the current fan speed by 1000 rpm, respectively. This scenario allows for smoother action transitions. For this case, we allow 10 different rpm values: 9000 through 18000, in steps of 1000 rpm.

Reward: In this section, we design a reward function that functions as a proxy for the operational efficiency of the SSD rack. One of the most important components of a RL solution is *reward shaping*. Reward shaping refers to the process of incorporating domain knowledge towards engineering a reward function, so as to better guide the agent towards its optimal behavior. Being able to devise a good reward function is critical since it explicitly relates to the expected return that needs to be maximized. We now list some desired properties of a meaningful reward function that will help perform reward shaping in the context of our problem.

- Keeping both the devices' temperatures and fan speeds low should yield the highest reward, since this scenario means the device operation is most efficient. However, also note that this case is feasible only when the I/O loads are absent or are very small.
- Irrespective of the I/O load, a low temperature in conjunction with a high fan speed should yield a bad reward. This condition is undesirable since otherwise, the agent can always set the fan speed to its maximum value, which in turn will not only consume a lot of energy, but also increase the wear on the mechanical components in the system.

– A high temperature in conjunction with a low fan speed should also yield a poor reward. If not, the agent may always choose to set the fan speed to its minimum value. This may result in overheating the system and potential SSD damages, in particular when the I/O loads are high.
– Finally, for different I/O loads, the optimal rewards should be similar. Otherwise, the RL agent may learn to overfit and perform well only on certain loads.

While there are several potential candidates for our desired reward function, we used the following mathematical function:

$$R = -\max\left(\frac{\Gamma(\bar{T})}{\Gamma(\bar{F})}, \frac{\Gamma(\bar{F})}{\Gamma(\bar{T})}\right), \tag{2}$$

where $\bar{T}$ and $\bar{F}$ represent the averaged values of temperature (over the 24 SSDs) and fan speeds (over the 5 fans), respectively. $\Gamma(\cdot)$ is the normalizing transformation explained in (1), and is performed using the temperature and fan speed minimum and maximum values listed in Table 1. Note that while this reward function weighs F and T equally, we can tweak the relationship between F and T to meet other preferential tradeoffs that the system operator might find desirable. Nevertheless, the DRL algorithm will be able to optimize the policy for any designed reward function.

Figure 3 plots the reward function we use as a function of mean temperature $\bar{T}$ (in °C) and fan speed $\bar{F}$ (in rpm). Also shown on the temperature-fan speed plane are contours representing regions of similar rewards. The colorbar on the right shows that blue and green colors represent the regions with poor rewards, while (dark and light) brown shades the regions with high rewards. All the aforementioned desired properties are satisfied with this reward function - the reward is maximum when both fan speed temperature are low; when either of them becomes high, the reward drops and there are regions of similar maximal rewards for different I/O loads (across the space of temperature and fan speeds).

2.3 Algorithm: The Advantage Actor-Critic (A2C) Agent

Once we formulate the state, action and reward components, there are several methods in the RL literature to solve our problem. A rather classic approach is the policy gradient (PG) algorithm [9], that essentially uses gradient-based techniques to optimize the agent's policy. PG algorithms have lately become popular over other traditional RL approaches such as Q-learning [10] and SARSA [11] since they have better convergence properties and can be effective in high-dimensional and continuous action spaces.

While we experimented with several PG algorithms including Vanilla Policy Gradient [12] (and its variants [13,14]) and Deep Q-Learning [15], the most encouraging results were obtained with the **Advantage Actor-Critic (A2C)** agent. A2C is essentially a synchronous, deterministic variant of Asynchronous

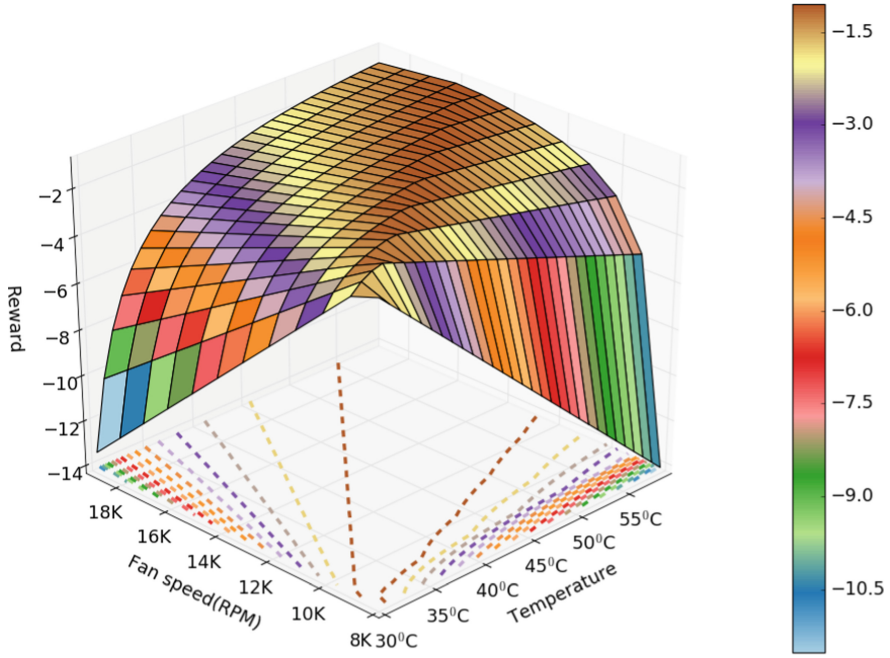

Fig. 3. Depiction of the reward (operational efficiency) versus temperature and fan speeds. The reward contours are also plotted on the temperature-fan speed surface. The brown colored contour marks the regions of optimal reward. (Color figure online)

Advantage Actor-Critic (A3C) [16], that yields state-of-the-art performance on several Atari games as well as on a wide variety of continuous motor control tasks.

As the name suggests, actor-critic algorithms comprise two components, an *actor* and a *critic*. The actor determines the best action to perform for any given state, and the critic estimates the actor's performed action. Iteratively, the actor-critic network implements generalized policy iteration [11] - alternating between a policy evaluation step and a policy improvement step. Architecturally, both the actor and the critic are best modeled via functional approximators, such as deep neural networks.

2.4 Actor-Critic Network Architecture

For our experiments, we employed a **dueling network architecture**, similar to the one proposed in [17]. The exact architecture is depicted in Fig. 4: the state of the system is a 7-length vector that is fed as input to a fully connected (FC) layer with 10 neurons represented by trainable weights θ. The output of this FC layer explicitly branches out to two separate feed-forward networks - the *policy* (actor) *network* (depicted on the upper branch in Fig. 4) and the *state-value function* (critic) *network* (depicted on the lower branch). The parameters θ are shared

between the actor and the critic networks, while additional parameters α and β are specific to the policy and state-value networks, respectively. The policy network that has a hidden layer with 5 neurons and a final softmax output layer for predicting the action probabilities (for the 7 raw or 3 incremental actions). The state-value function network comprises a hidden layer of size 10 that culminates into a scalar output for estimating the value function of the input state. The actor aims to approximate the optimal policy π^*: $\pi(a|s;\theta,\alpha) \approx \pi^*(a|s)$, while the critic aims to approximate the optimal state-value function: $V(s;\theta,\beta) \approx V^*(s)$.

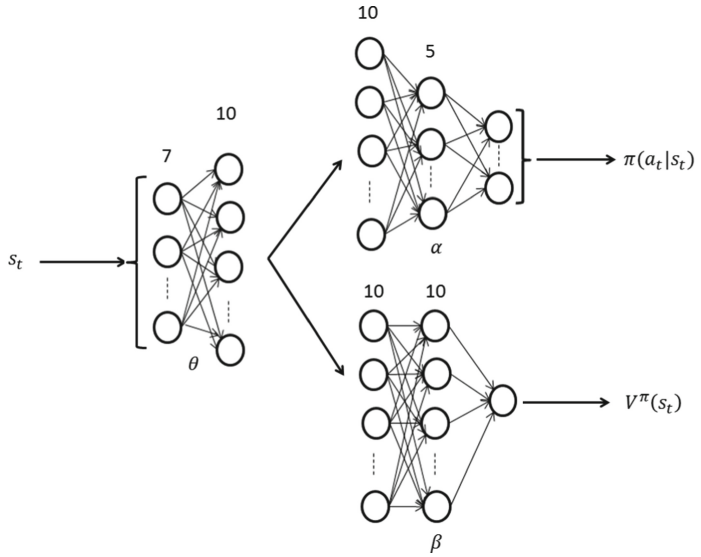

Fig. 4. The employed dueling network architecture with shared parameters θ. α and β are the actor- and critic-specific parameters, respectively. All the layers are fully connected neural networks; the numbers shown above represent the hidden layer dimensions. The policy network on the upper branch estimates the action probabilities via a softmax output layer, while the critic network on the lower branch approximates the state-value function.

Prior DRL architectures for actor-critic methods [16,18,19] employ single-stream architectures wherein the actor and critic networks do not share parameters. The advantage of our dueling network lies partly in its ability to compute both the policy and state-value functions via fewer trainable parameters vis-à-vis single-stream architectures. The sharing of parameters also helps mitigate overfitting one function over the other (among the policy and state-value functions). In other words, our dueling architecture is able to learn both the state-value and the policy estimates efficiently. With every update of the policy network parameters in the dueling architecture, the parameters θ get updated as well - this contrasts with the updates in a single-stream architecture wherein when the policy parameters are updated, the state-value function parameters

remain untouched. The more frequent updating of the parameters θ mean a higher resource allocation towards the learning process, thus resulting in faster convergence in addition to obtaining better function approximations.

The pseudocode for our A2C algorithm in the context of the dueling network architecture (see Fig. 4) is described in Algorithm 1. Note that R represents the Monte Carlo return, and well-approximates the action-value function. Accordingly, we use $R - V(s; (\theta, \beta))$ as an approximation to the advantage function.

Algorithm 1. Advantage Actor-Critic (A2C) - pseudocode

// Notate shared parameters by θ and actor- and critic-specific parameters by α and β, respectively.
// Assume same learning rates η for θ, α as well as β. In general, they may all be different.
Initialize θ, α and β via uniformly distributed random variables.
repeat
  Reset gradients $\mathrm{d}\theta = 0$, $\mathrm{d}\alpha = 0$ and $\mathrm{d}\beta = 0$.
  Sample N trajectories $\tau_1, \ldots, \tau_N$ under the (current) policy $\pi(\cdot; (\theta, \alpha))$.
  i = 1
  repeat
    $t_{\text{start}} = t$
    Obtain state s_t
    repeat
      Perform action a_t sampled from policy $\pi(a_t|s_t; (\theta, \alpha))$.
      Receive reward $r_t(s_t, a_t)$ and new state s_{t+1}
      $t \leftarrow t + 1$
    until $t - t_{\text{start}} = H$
    $i \leftarrow i + 1$
    Initialize R: $R = V(s_t; (\theta, \beta))$
    for $i \in \{t - 1, \ldots, t_{\text{start}}\}$ **do**
      $R \leftarrow r_i(s_i, a_i) + \gamma R$
      Sum gradients w.r.t θ and α: // gradient ascent on the actor parameters
      $\mathrm{d}\theta \leftarrow \mathrm{d}\theta + \nabla_\theta \log \pi(a_i|s_i; (\theta, \alpha)) (R - V(s_i; (\theta, \beta)))$
      $\mathrm{d}\alpha \leftarrow \mathrm{d}\alpha + \nabla_\alpha \log \pi(a_i|s_i; (\theta, \alpha)) (R - V(s_i; (\theta, \beta)))$
      Subtract gradients w.r.t β and θ: //gradient descent on the critic parameters
      $\mathrm{d}\theta \leftarrow \mathrm{d}\theta - \nabla_\theta (R - V(s_i; (\theta, \beta)))^2$
      $\mathrm{d}\beta \leftarrow \mathrm{d}\beta - \nabla_\beta (R - V(s_i; (\theta, \beta)))^2$
    end for
  until $i = N$
  // Optimize parameters
  Update θ, α and β: $\theta \leftarrow \theta + \eta \mathrm{d}\theta$, $\alpha \leftarrow \alpha + \eta \mathrm{d}\alpha$, $\beta \leftarrow \beta + \eta \mathrm{d}\beta$.
until convergence.

3 Experimental Setup and Results

3.1 Timelines

Time is slotted to the duration of 25 s. At the beginning of every time slot, the agent observes the state of the system and prescribes an action. The system is then allowed to stabilize and the reward is recorded at the end of the time slot (which is also the beginning of the subsequent time slot). The system would have, by then, proceeded to its next state, when the next action is prescribed. We use a time horizon of 10 slots ($H = 250$ s), and each iteration comprises $N = 2$ horizons, i.e., the network parameters θ, α and β are updated every 500 s.

3.2 I/O Scenarios

We consider two different I/O loading scenarios for our experiments.

- **Simple periodic workload:** We assume a periodic load where within each period, there is no I/O activity for a duration of time (roughly 1000 s) followed by heavy I/O loading for the same duration of time. I/O loading is performed using a combination of 'read', 'write' and 'randread' operations with varying block sizes of data ranging from 4 KBytes to 64 KBytes. A timeline of the periodic workload is depicted in Fig. 5 (left).
- **Complex stochastic workload:** This is a realistic workload where in every time window of 1000 s, the I/O load is chosen uniformly randomly from three possibilities: no load, medium load or heavy load. A sample realization of the stochastic workload is shown in Fig. 5 (right).

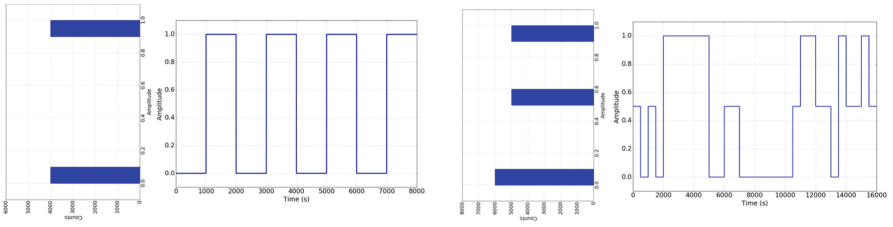

Fig. 5. The simple periodic workload (left) with a period of 2000 s, and a realization of the more realistic stochastic workload (right). Histograms of the load types are also shown for clarity. While the simple load chooses between no load and heavy load in a periodic manner, the complex load chooses uniformly randomly between the no load, medium load and heavy load scenarios.

3.3 Hyperparameters

Table 2 lists some hyperparameters used during model training.

Table 2. Table of hyperparameters.

Parameter	Value
γ: discount factor used in computing the returns	0.99
Optimizer for training θ, α and β	Stochastic gradient descent
SGD learning rate η	0.01
Entropy bonus scaling (See [23])	0.05

3.4 Results

In this section, we present our experimental results. Specifically, we consider three separate scenarios - (a) periodic load with raw actions, and the stochastic load with both (b) raw and (c) incremental actions. In each of the cases, we compare the performances of our A2C algorithm (after convergence) against the default policy (which we term the *baseline*). Recall that he baseline simply uses a tabular method to control fan speeds based on temperature thresholds.

(a) Scenario 1: Periodic Load with Raw Actions. We first experimented with the periodic I/O load shown in Fig. 5 (left). Figure 6 summarizes the results; it shows the I/O activity, normalized cumulative rewards[6], fan speeds and temperature values over time for both the baseline and our method. Compared to the baseline, the A2C algorithm provided a cumulative reward uplift of ∼33% for similar I/O activity! The higher reward was obtained primarily as a result of the A2C algorithm prescribing a higher fan speed when there was heavy I/O loading (roughly 16000 rpm versus 11000 rpm for the baseline), which resulted in

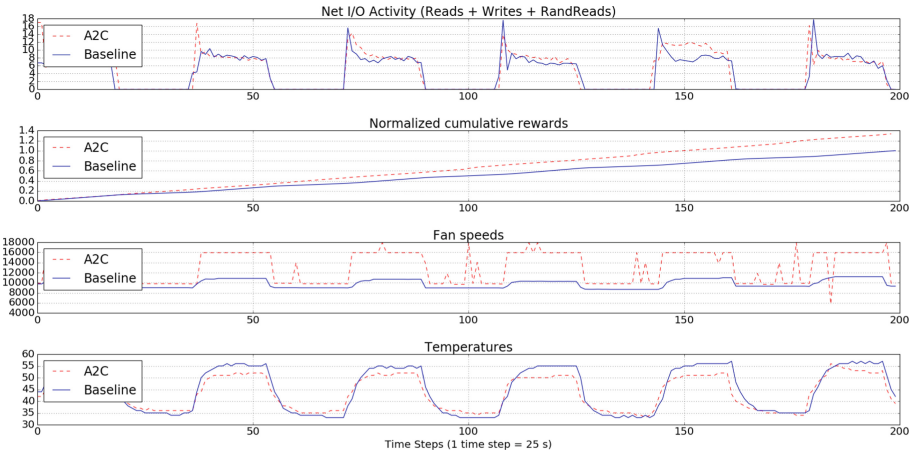

Fig. 6. Performance comparison of baseline and A2C for scenario 1.

[6] We normalize the cumulative baseline reward to 1.0, and correspondingly scale the cumulative A2C reward. This also helps quantify the reward uplift.

a lower temperature (52 °C versus 55 °C). To clarify this, Fig. 7 plots the contour regions of the temperature and fan speeds for the baseline (left) and the A2C algorithm, at convergence (right). The black-colored blobs essentially mark the operating points of the SSD rack for the two types of load. Evidently, the A2C method converges to the better reward contour as compared to the baseline.

(b) Scenario 2: Stochastic Load with Raw Actions. With the stochastic I/O load (see Fig. 8), the overall reward uplift obtained is smaller (only 12% after averaging over 3000 time steps) than the periodic load case. Again, the A2C algorithm benefits by increasing the fan speeds to keep the temperatures lower. Upon looking more closely at the convergence contours (Fig. 9), it is noted

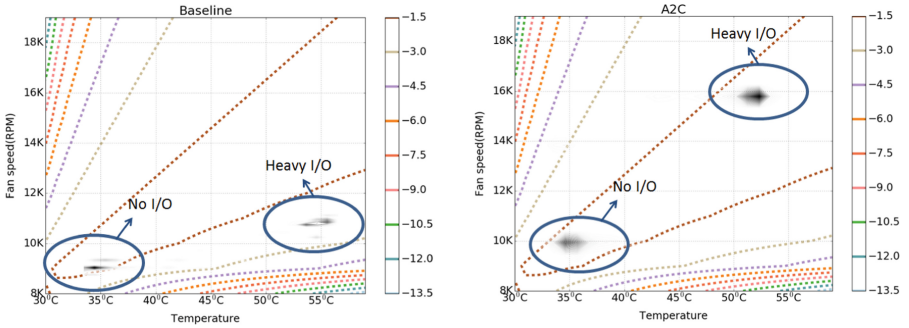

Fig. 7. For the no load scenario, the baseline policy settles to 36 °C and 9 K rpm while the A2C algorithm converges to 35 °C and 10 K rpm. With heavy I/O loading, the corresponding numbers are 55 °C and 11 K versus 52 °C and 16 K. The A2C algorithm is seen to always settle at the innermost contour, as desired.

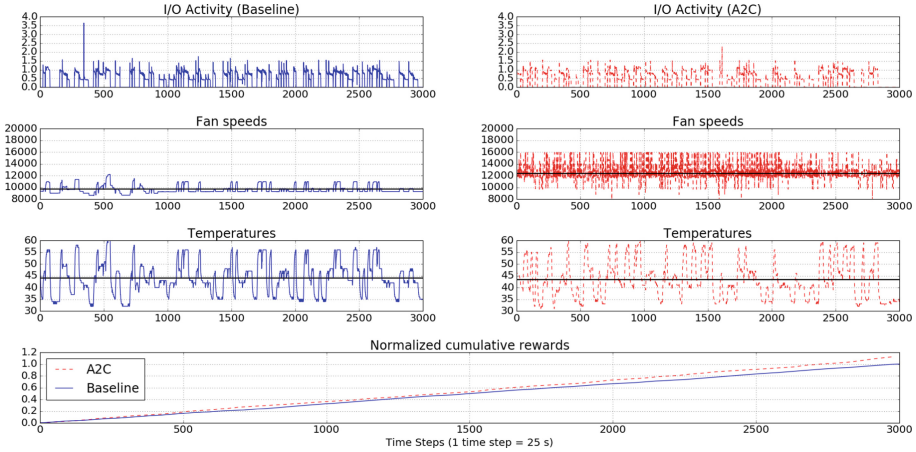

Fig. 8. Performance comparison of baseline and A2C for scenario 2. The mean values of temperatures and fan speeds are shown using the black line.

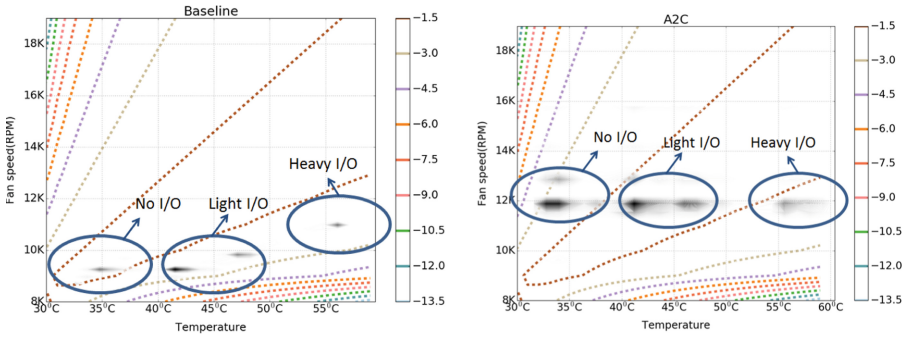

Fig. 9. Temperature and fan speed contours for the baseline (left) and the A2C method (right). With the stochastic loading with 7 actions, the A2C does not converge as well as in the periodic load case (see Fig. 9 (right)).

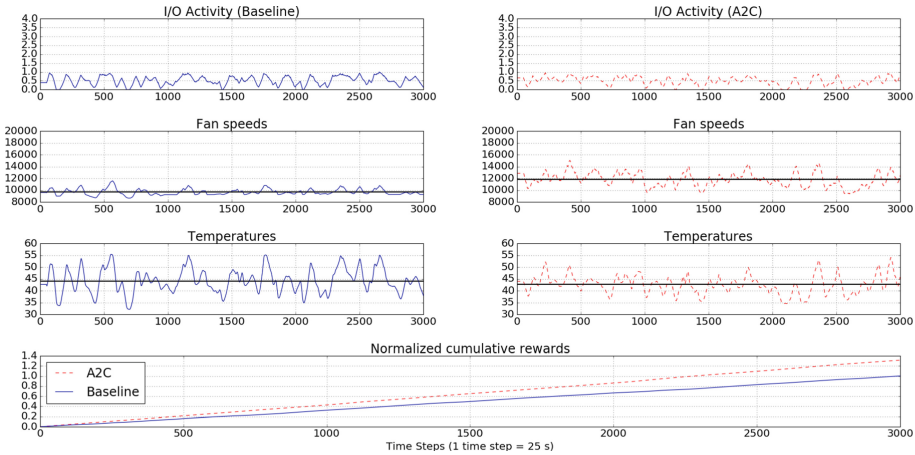

Fig. 10. Performance comparison of baseline and A2C for scenario 3.

that with the stochastic load, the A2C method sometimes settles to sub-optimal reward contours. We believe this happened due of insufficient exploration.

(c) Scenario 3: Stochastic Load with Incremental Actions. With raw actions, the action space is large to explore given the random nature of the I/O load, and this slows learning. To help the algorithm explore better, we study scenario 3. wherein actions can take on only 3 possible values (as compared to 7 values in the prior scenario). With this modification, more promising results are observed - specifically, we observed a cumulative reward uplift of 32% (see Fig. 10). In fact, the A2C algorithm is able to start from a completely random policy (Fig. 11 (left)) and learn to converge to the contour region with the best reward (Fig. 11 (right)).

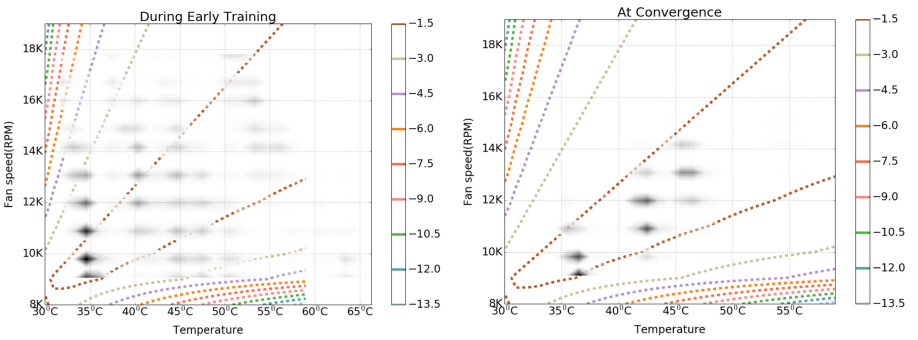

Fig. 11. Temperature and fan speed contours for the A2C method under scenario 3. The left plot is taken during early steps of training, while the plot on the right is taken at convergence. This illustrates that the A2C algorithm is able to start exploring from a completely random policy (black blobs everywhere) and learn to converge to the contour region with the best reward.

4 Concluding Remarks

In this paper, we tackle the problem of optimizing the operational efficiency of a SSD storage rack server using the A2C algorithm with a dueling network architecture. Experimental results demonstrate promising reward uplifts of over 30% across two different data I/O scenarios. We hope that this original work on applied deep reinforcement learning instigates interest in employing DRL to other industrial and manufacturing control problems. Interesting directions for future work include experimenting with other data I/O patterns and reward functions, and scaling this work up to train multiple server racks in parallel in a distributed fashion via a single or multiple agents.

Acknowledgements. We want to thank the Memory Systems lab team, Samsung Semiconductors Inc. for providing us a SSD storage rack, workload, data and fan control API for running our experiments. We also thank the software engineering team at Samsung SDS for developing a DRL framework [24] that was used extensively for model building, training and serving.

References

1. Power Usage Effectiveness. https://en.wikipedia.org/wiki/Power_usage_effectiveness
2. Cisco Global Cloud Index: Forecast and Methodology, 2016–2021 White Paper, February 2018. https://www.cisco.com/c/en/us/solutions/collateral/service-provider/global-cloud-index-gci/white-paper-c11-738085.html
3. Arulkumaran, K., Deisenroth, M.P., Brundage, M., Bharath, A.A.: A brief survey of deep reinforcement learning. IEEE Sig. Process. Mag. **34**(6), 26–28 (2017)
4. Shuja, J., Madani, S.A., Bilal, K., Hayat, K., Khan, S.U., Sarwar, S.: Energy-efficient data centers. Computing **94**(12), 973–994 (2012)

5. Sun, J., Reddy, A.: Optimal control of building HVAC systems using complete simulation-based sequential quadratic programming (CSBSQP). Build. Environ. **40**(5), 657–669 (2005)
6. Ma, Z., Wang, S.: An optimal control strategy for complex building central chilled water systems for practical and real-time applications. Build. Environ. **44**(6), 1188–1198 (2009)
7. Evans, R., Gao, J.: DeepMind AI Reduces Google Data Centre Cooling Bill by 40%, July 2016. Blog: https://deepmind.com/blog/deepmind-ai-reduces-google-data-centre-cooling-bill-40/
8. Li, Y., Wen, Y., Guan, K., Tao, D.: Transforming Cooling Optimization for Green Data Center via Deep Reinforcement Learning (2017). https://arxiv.org/abs/1709.05077
9. Peters, J., Schaal, S.: Reinforcement learning of motor skills with policy gradients. Neural Netw. (2008 Spec. Issue) **21**(4), 682–697 (2008)
10. Watkins, C.J.C.H., Dayan, P.: Q-learning. Mach. Learn. **8**(3–4), 279–292 (1992)
11. Sutton, R.S., Barto, A.G.: Reinforcement Learning: An Introduction, 2nd edn. MIT Press, Cambridge (2017)
12. Sutton, R.S., McAllester, D., Singh, S., Mansour, Y.: Policy gradient methods for reinforcement learning with function approximation. In: Advances in Neural Information Processing Systems, vol. 12, pp. 1057–1063 (2000)
13. Williams, R.J.: Simple statistical gradient-following algorithms for connectionist reinforcement learning. Mach. Learn. **8**(3–4), 229–256 (1992)
14. Greensmith, E., Bartlett, P.L., Baxter, J.: Variance reduction techniques for gradient estimates in reinforcement learning. J. Mach. Learn. Res. **5**, 1471–1530 (2004)
15. Mnih, V., et al.: Human-level control through deep reinforcement learning. Nature **518**(7540), 529–533 (2015)
16. Mnih, V., et al.: Asynchronous methods for deep reinforcement learning. In: International Conference on Machine Learning, pp. 1928–1937 (2016)
17. Wang, Z., Schaul, T., Hessel, M., Hasselt, H.V., Lanctot, M., Freitas, N.D.: Dueling network architectures for deep reinforcement learning. In: International Conference on International Conference on Machine Learning, vol. 48 (2016)
18. Silver, D., Lever, G., Heess, N., Degris, T., Wierstra, D., Riedmiller, M.: Deterministic policy gradient algorithms. In: International Conference on Machine Learning (2014)
19. Lillicrap, T.P., et al.: Continuous control with deep reinforcement learning. US Patent Application, No. US20170024643A1. https://patents.google.com/patent/US20170024643A1/en
20. iostat Man Page. https://linux.die.net/man/1/iostat
21. ipmi-sensors Man Page. https://linux.die.net/man/8/ipmi-sensors
22. ipmitool Man Page. https://linux.die.net/man/1/ipmitool
23. O'Donoghue, B., Munos, R., Kavukcuoglu, K., Mnih, V.: Combining policy gradient and Q-learning. In: International Conference on Learning Representations (2017)
24. Parthasarathy, K., Kathalagiri, G., George, J.: Scalable implementation of machine learning algorithms for sequential decision making. In: Machine Learning Systems, ICML Workshop, June 2016