Research Article

Action-Based Load Balancing Technique in Cloud Network Using Actor-Critic-Swarm Optimization

Arabinda Pradhan, Sukant Kishoro Bisoy, and Mangal Sain

1Department of Computer Science & Engineering, CV Raman Global University, Bhubaneswar, Odisha, India
2Division of Computer Engineering, Dongseo University, Busan, Republic of Korea

Correspondence should be addressed to Mangal Sain; mangalsain1@gmail.com

Received 21 April 2022; Accepted 16 June 2022; Published 30 June 2022

Academic Editor: Kalidoss Rajakani

Copyright © 2022 Arabinda Pradhan et al. This is an open access article distributed under the Creative Commons Attribution License, which permits unrestricted use, distribution, and reproduction in any medium, provided the original work is properly cited.

Increasing scale of task in cloud network leads to problem in load balancing and its improvement in parameters. In this paper, we proposed a hybrid scheduling policy which is a hybrid of both Particle Swarm Optimization (PSO) algorithm and actor-critic algorithm named as Hybrid Particle Swarm Optimization Actor Critic (HPSOAC) to solve this issue. This hybrid scheduling policy helps to each agent to improve an individual learning as well as learning through exchanging information among other agents. An experiment is carried out by the help of Python simulator with TensorFlow. Outcome shows that our proposed scheduling policy reduces 5.16% and 10.86% in energy consumption, reduces 7.13% and 10.04% in makespan time, and has marginally better resource utilization over Deep Q-network (DQN) and Q-learning based on Modified Particle Swarm Optimization (QMPSO) algorithm, respectively.

1. Introduction

Nowadays, cloud network is a popular technology that serves various services to its clients over the internet. Therefore, the number of client requests increases but the server or physical machine (PM) in the datacenter is limited. To handle large number requests, virtualization concept is used, through which one server is logically divided into number of virtual machine (VM) and tries to handle all incoming user requests [1]. Due to dynamic nature of cloud network, loads are fluctuated with respect to time for allocating a suitable VM. It shows load balancing problem within VM that put a direct impact on PM. This problem will lead to show high cost, minimize the profit of an organization, and degrade system performance. Each PM consumes electric power for work. That means if a server has number of workloads, then it consumes more energy [2]. To overcome this problem, a better scheduling algorithm is required that can handle the load among the VMs and execute all the incoming tasks within less execution time and consume less energy on the datacenter. Therefore, our objective is to transfer the extra task to the underloaded machine to complete the task execution within less time period and handle the load of the server to minimize the consumption of energy but maximize the utilization of resources.

Basic framework of cloud network is shown in Figure 1, where it contains two layer such as task layer and datacenter layer. These two layers store needed information to get our objective. In task layer, all needed information of incoming task such as expected finishing time (EFT), transfer speed of task (T_{tt}), task length (T_{len}), and file size (T_{file}) are stored in task queue to find best VMs from datacenter. Similarly, datacenter layer contains all the required information of both servers and VMs, such as storage unit, processing unit, data transfer capacity, and processing speed. This data is forwarded to the scheduler where it will be applying best scheduling policy to accomplish the objective and getting optimum result.

Due to arrival of enormous number of jobs at the datacenter, it makes complex and dynamic environment. If the environment cannot be adequately predicted, then the reasonable and effective scheduling algorithm cannot be
successfully applied. Due to the unpredictability of upcoming activities and the dynamic environment, it is too difficult to represent the environment correctly in practice. There are no patterns that can be used to predict how the task will arrive at the datacenter. It means the number and size of tasks that will be assigned in the future are unclear. As a result, the algorithm has to schedule the tasks at once, with no prior experience or information. For example, network bandwidth, speed, processing performance of multiple machines, capacity, and load of the required resources to support task execution all affect task execution time. The demand for data center services or jobs fluctuates depending on the time period, environmental conditions, and other factors. When a server is used for an extended period of time, it consumes more energy. It is influenced by the server’s status and can have a direct impact on service costs. Based on changing demand, the scheduling algorithm must automatically optimize execution time, resource use, and energy consumption.

Almost all publications have attempted to reduce makespan, increase resource utilization, reduce energy consumption, and balance the load of all VMs inside datacenter. But still load balancing problem is arising because if a machine is heavy load, then it takes more time to transfer the extra task. Also, if a machine is not used for long time, then it consumes energy that degrades system performance. Similar objectives are taken in this paper and try to solve these objectives with the help of machine learning as well as swarm optimization algorithm. Various types of machine learning approaches such as reinforcement learning (RL), deep reinforcement learning (DRL), and actor-critic (AC) are used in the field of cloud computing. In these methodologies, an agent selects a suitable action to perform in the state of the environment to get the reward. This learning interaction proceeds until an agent receives maximum reward from the environment. In cloud networking, these methods are used to drive the action of cloud state to get the result [3]. But whenever we deal with high-dimensional problem, RL such as Q-learning represents the limitation of state space and it is hard to apply to get the objective outcome [4, 5]. In DRL strategy, for example, Deep Q-network (DQN), it may not manage the problem in continuous nature regularly [6, 7].

To take care of the issue in RL and DRL, in this paper we developed a hybrid scheduling policy which depends on actor-critic learning method and PSO-based population method, known as HPSOAC (Hybrid Particle Swarm Optimization Actor Critic) scheduling policy. This policy finds the better solution where the problem is continuous and dynamic in nature. In our proposed method, the number of agents learns at the same time that depends on two learning methodologies such as individual learning and learning from group by passing their data to achieve the goal. Individual best of each agent is obtained from the current information of the state, and the worldwide best is found out from the whole agent. After getting the result, each agent updates its actor-critic parameter value by utilizing the update conditions of PSO. In our proposed method, we assume VM as an agent. Server within the datacenter is
represented as an environment. The state of the server is active (both running and idle) and sleep. Operation within the server is known as action. Whether all the objectives are achieved or not can be defined as rewards. To train our agent, we retrieve an important data from the environment such as capacity, bandwidth, RAM, CPU, network, and storage.

Cloud requests have stochastic properties which cause problems with balancing the load and increasing the cost of services. Proper decision can be taken in the cloud environment (datacenter) to allocate suitable resources to the incoming requests. AC can train the agent to learn the optimal decision through interaction with environments without requiring any prior knowledge. AC acquires knowledge of the environment to choose the best actions based on the parameters of the environment. It should handle the extra task and migrate into the underloaded VM. This decision on the actions is to maximize the reward of each agent. PSO is chosen as the metaheuristic optimization technique to enhance the system performance and reduce search time even while improving the fitness value. Based on the fitness value, agents are exchanging their information among themselves to get the suitable VM to allocate the incoming task. The fundamental commitments of our proposed work can be given as follows: (1) To solve load balancing problem and its parameters, we developed an effective scheduling policy. (2) This scheduling policy is a combination of PSO and AC. (3) If one VM is heavily loaded, then we take an appropriate action to transfer extra task to underloaded VM that solves load balancing problem.

To improve the makespan and energy consumption of the average work, the proposed DQN algorithm in [3] is compared to three different algorithms: random, round robin, and Modified PSO (MoPSO). During training, the author used the DQN method with a weight parameter of 0.9. Two types of experiments were carried out to obtain the results in energy consumption and makespan time. In comparison to its competitors, the DQN algorithm reduces energy consumption by 2.6% to 11.7% and reduces makespan time by 8.4% to 21.1%. In [10], the proposed Modified PSO and Q-learning algorithm (QMPSO) method are compared to two existing algorithms: Q-learning and Modified PSO. The algorithm’s performance was evaluated in terms of makespan, energy consumption, and throughput, which are indicators of the external services’ effectiveness. The fitness function and the weight of the fitness value are used to achieve these goals. The fitness value’s weight ranges from 0.5 to 1.0. Three types of experiments were carried out to determine the system’s energy usage, makespan time, and throughput. In comparison to competitors, the QMPSO algorithm saves 3.5% to 7.4% and 5.3% to 18.6% makespan time while increasing throughput by 60%. In [14], the proposed hybrid scheduling algorithm is compared with the Cloud Workflow Scheduling Algorithm (CWSA), Dynamic Power Saving Resource Allocation (DPRA), Ant Colony Optimization and Particle Swarm Optimization (ACOPSO), and a heuristic approach. The algorithm’s performance was evaluated in terms of makespan time, power consumption, and resource utilization. In comparison to competitors, hybrid algorithm reduces 8.3% makespan as compared to CWSA and the heuristic approach. Similarly, it reduces power consumption by 10% as compared to the DPRA approach. Finally, it increases resource utilization by 10% as compared to the ACOPSO and CWSA approaches. In [33], the proposed improved weighted round robin (IWRR) algorithm is compared with two heuristic-based algorithms, such as the round robin and the weighted round robin (WRR) algorithm. In comparison to competitors, the IWRR algorithm reduces 5% and 10% of makespan time as compared to the WRR and RR algorithms. All three methods, except IWRR, have two common objectives, i.e., makespan time and energy consumption. To achieve the goal, there is no suitable action defined to handle the extra task that can increase the reward in a dynamic environment. Therefore, in this paper, we define a suitable action that can handle the extra task and give the maximum reward as compared to the above methods. Comparison between various scheduling policies is presented in Table 1.

The rest of the paper is composed as follows: related work, objective formulation, actor-critic method, HPSOAC scheduling policy, pseudocode, experimental analysis, and conclusion of the paper.

2. Related Work

There are an enormous number of scheduling policies that have already developed that are used on static policy, dynamic policy, and hybrid policy and focused on either a single objective, biobjective, or multiojective. These policies have concentrated their research work on maintaining the load balance between machines as well as various parameters of load balancing, such as makespan time, energy consumption, and resource usage. For this purpose, researchers proposed various machine learning and population-based optimization scheduling strategies for the cloud environment to solve the above issue. To improve user service quality necessities, authors in [3] proposed a technique dependent on the Deep Q-network (DQN) calculation to diminish energy utilization and makespan time by changing the extent of the reward of various streamlining objectives. This algorithm solves difficult multiobjective optimization problems and provides a sensible and efficient resource allocation and task scheduling strategy. It modifies the weight of rewards to make a trade-off in the link between energy consumption and task makespan. The authors in [4] divided the complicated cloud scheduling problem into two subscheduling problems: job scheduling and resource allocation. Heterogeneous Distributed Deep Learning (HDDL) is utilized to handle job scheduling, while Deep Q-network (DQN) is employed to solve resource allocation. This method is commonly used to increase energy efficiency. Furthermore, the suggested framework has strong scalability and low computation time, as well as the potential to obtain a global near-optimal by attaining local optimization at each level. Authors in [6] presented an advantage actor-critic-based reinforcement learning (RL) paradigm for resource allocation in cloud datacenter. First, depending on the critic’s evaluations (evaluating actions), the actor...
parameterizes the policy (allocating resources) and determines continuous actions (scheduling jobs). The policy is then updated using gradient ascent, and using the advantage function, the variation of the policy gradient can be greatly reduced. Furthermore, in terms of work latency, the suggested method beats previous resource allocation algorithms and provides a faster convergence time than the typical policy gradient method. [8] described a machine learning-based adaptive resource management technique that allows a cloud storage system to self-manage and give higher performance in the face of a wide range of workloads and resource bottlenecks. A stochastic policy gradient-based reinforcement learning technique was also used to detect performance concerns in cloud storage and take the required actions, such as load balancing and data movement, to improve storage performance. Authors in [9] used design Deep Deterministic Policy Gradient (DDPG) task scheduling approach which depends on deep reinforcement learning (DRL) method to diminish the reaction time and keep up the load balancing. Without any prior knowledge, the proposed algorithm may learn straight from its experience and make the proper scheduling decision for VMs for ongoing online task requests. The agent’s status is determined by the information about VMs and tasks received by the scheduler. The agent distributes each task to the appropriate VM adaptively using the defined states, and the task response time is rewarded as the agent’s activity. To improve the exhibition, keep up load adjusting and increment the throughput, and [10] developed a hybrid technique which combines both Modified PSO and Q-learning algorithm known as QMPSO. This hybridization process is used to change the MPPO’s velocity using the gbest and pbest functions, which are based on the best action obtained by improved Q-learning. This aids in achieving a goal. [11] proposed Q-learning-based foresighted task scheduling to limit reaction time and makespan however increment the resource effectiveness. With the help of the Q-learning algorithm, the scheduler allocates all the incoming tasks to the suitable VMs in a first-come-first-serve manner. The scheduler takes all the necessary information about the task and the VM before allocating the task, such as task length, entering time, and VM capacity. [12] proposed QL-HEFT task scheduling approach that consolidates Q-learning and Heterogeneous Earliest Finish Time (HEFT) calculation to diminish the makespan. The algorithm employs the upward ranked value of HEFT as the immediate reward in Q-learning. The algorithm then sorts the original task order using the converged Q-table to arrive at the best result. The entire process employs the Earliest Finish Time (EFT) allocation approach, which assigns the best processor to each task in the most efficient order. [13] introduced the advantage actor-critic (A2C) algorithm, which is based on deep reinforcement learning and is used to improve job scheduling performance and datacenter resource usage. The actor network is used to pick actions in A2C, whereas the critic network is used to evaluate them. Without any prior information about the upcoming jobs, the proposed technique is used to reduce average task completion time and improve resource use efficiency. [14] developed an approach that is dependent on hybrid Ant Colony Optimization (ACO) and deep reinforcement learning (DRL) to limit task execution time and improve the usage of resources. In a cloud environment, the DRL is used to increase the utilization of idle resources by dividing them into action space and state space. DRL-based resource management reduces resource consumption in large-scale cloud environments with a large number of servers receiving a significant number of requests per day. Once the criterion is met, ACO is used to map jobs to relevant virtual machines. [15] developed energy-efficient resource allocation method which is the combination of reinforcement learning (RL) and fuzzy logic to reduce energy consumption and increase CPU utilization. At the datacenter level, reinforcement learning (RL) is used to recommend the best allocation policy based on Power Usage Effectiveness (PUE), Datacenter Infrastructure Efficiency (DCIE), and CPU utilization. Fuzzy logic, on the other hand, is used in the allocation process that leads to a green datacenter. [16] presented the Asynchronous Particle Swarm Optimization (APSO) algorithm, which is an asynchronous multithreaded parallel PSO algorithm. APSO is a basic, easy-to-implement algorithm that solves the problem of the classic PSO algorithm falling into a local minimum. Then, using APSO in an asynchronous reinforcement learning algorithm called Backward Q-learning, based on State Action Reward State Action (SARSA), a novel asynchronous reinforcement learning method called Backward Q-learning and Asynchronous Particle Swarm Optimization (APSO-BQSA) is suggested. APSO-BQSA uses the APSO method to optimize the parameters while updating them with the improved BQSA. The proposed approach outperforms the existing reinforcement learning algorithm in terms of convergence speed and performance, and it may be used to solve a variety of situations. [17] proposed an algorithm to combine policy gradient (PG) with Particle Swarm Optimization parameter exploration (PG-PSOPE) algorithm to avoid the problem in continuous nature and improve convergence rate. PG is a solution that involves parameterizing the policy and updating the parameters via gradient estimation. The gradients of policy can be

Table 1: Comparison between various scheduling policies with their objectives.

<table>
<thead>
<tr>
<th>Ref. No.</th>
<th>Scheduling policy</th>
<th>Makespan time</th>
<th>Energy consumption</th>
<th>Throughput</th>
<th>Resource utilization</th>
</tr>
</thead>
<tbody>
<tr>
<td>[3]</td>
<td>DQN</td>
<td>Medium</td>
<td>High</td>
<td></td>
<td></td>
</tr>
<tr>
<td>[10]</td>
<td>QMPSO</td>
<td>High</td>
<td>Low</td>
<td>Medium</td>
<td></td>
</tr>
<tr>
<td>[14]</td>
<td>Hybrid ACO and DRL</td>
<td>Low</td>
<td>Medium</td>
<td>Medium</td>
<td></td>
</tr>
<tr>
<td>[33]</td>
<td>IWRR</td>
<td>Low</td>
<td>Low</td>
<td></td>
<td></td>
</tr>
<tr>
<td>Proposed algorithm</td>
<td>HPPOAC</td>
<td>High</td>
<td>High</td>
<td>High</td>
<td></td>
</tr>
</tbody>
</table>
unbiasedly assessed using the normal PG by sampling various interactive trajectories. To circumvent this issue, the mapping between PG and PSO was introduced initially. Following that, guidelines for exploring and updating are devised, including the mutation probability, to prevent local optima and excessive variance caused by gradient estimation. As a result, the suggested PG-PSOPE explores the parameter space rather than the action space. In [18], Deep Q-network (DQN) method is proposed, which follows deep reinforcement learning (DRL) approach to reduce energy consumption and average response time while increasing the success rate. The DRL approach is utilized in this case to schedule jobs in the queue on a first-come, first-served (FCFS) basis. The reward is derived using the reward function, and the scheduling is based on the $q$ values evaluated by a deep neural network. A job will be given to the best VM from the cloud resource in terms of meeting job criteria and consuming the least amount of energy. Multiple jobs are scheduled to different VMs in the cloud resource, and the proposed technique is designed for online learning for a public cloud. Each job is self-contained and must be scheduled to execute in a virtual machine (VM) within a server, with each server containing exactly one virtual machine. Furthermore, during the job execution, each VM only has one instance. If a job is not at the first in the queue, it has to wait for all the jobs before it to be executed.

2.1. Objective Formulation. In this part, we will expound the objectives of cloud framework, including task makespan time, resource utilization, and energy consumption. Based on each objective, we found the reward function that helps us to reach the optimization of the system.

2.2. Makespan Time. Makespan time is referred to as most extreme time to finish or execute the task on a particular VM. If the task takes maximum execution time, then it shows poor in load balance and minimum execution time shows the better load balancing. Suppose datacenter contains $i$ and $j$ number of tasks and VMs, respectively, such as $T_j = \{ T_1, T_2, \ldots, T_m \}$ and $VM_j = \{ VM_1, VM_2, \ldots, VM_n \}$. In any case, the condition for execution of such task is greater than number of VMs, i.e., $i > j$ and task length $T_{len}$ with VM speed is dealing with millions of instructions (MI) and millions of instructions per second (MIPS), respectively. To accomplish our objective, we figure the processing rate of VM utilizing Equation (1), which depends on the properties of VM, such as MIPS, CPU, and memory.

$$\sum_{j=1}^{n} VM_{pr} = \sum_{j=1}^{n} (VM_{mips} + VM_{cpu} + VM_{mem}). \quad (1)$$

The expected finishing time (EFT) of task $T_i$ on virtual machine $VM_j$ can be address as the following equation:

$$EFT_{ij} = \frac{\sum_{j=1}^{n} T_{len}}{\sum_{j=1}^{n} VM_{pr}}. \quad (2)$$

When task $T_i$ is allocated on $VM_j$, then it takes some time, which is called as task allocation time ($TAT_{ij}$) and it is determined by the task file size with respect to bandwidth of VM ($VM_b$), represented in the following equation:

$$TAT_{ij} = \frac{T_{filesize}}{VM_b}. \quad (3)$$

Total finishing time ($FT_{ij}$) of all entering task to execute on VMs is calculated as the addition of expected finishing time ($EFT_{ij}$) and task allocation time ($TAT_{ij}$), represented in Equation (4) where $DV_{ij}$ is the decision variable and represented in Equation (5).

$$FT_{ij} = EFT_{ij} + TAT_{ij} + DV_{ij}, \quad (4)$$

$$DV_{ij} = \begin{cases} 1, \text{if } T_i \text{ is allocated to } VM_j, \\ 0, \text{otherwise}. \end{cases} \quad (5)$$

Makespan time (MST) can be diminishing when all tasks are completed their execution effectively. It can be determined as the maximum of $FT_i$ [19], and it can be calculated by the following equation:

$$MST = \max \{ FT_{ij} \}. \quad (6)$$

Thus, our first objective function can be formulated as in the following equation:

$$O_1 = \min (MST). \quad (7)$$

2.3. Resource Utilization. Next enhancement method is to build the utilization of resource ($R_{util}$) which is determined by the division of total finishing time with respect to makespan time and available in Equation (8). A high utilization rate shows that cloud provider can acquire greatest benefit.

$$R_{util} = \frac{\sum_{i=1}^{m} FT_{ij}}{MST}. \quad (8)$$

Therefore, our second objective function can be formulated as in the following equation:

$$O_2 = \max (R_{util}). \quad (9)$$

2.4. Energy Consumption. Excessive demanded for cloud services has expanded energy utilization in data centers. High energy utilization makes more expensive to operate as well as prompts carbon emissions. The energy received by a datacenter is mainly distributed to functioning servers or PMs, for cooling frameworks and for organizational components. Among these three sections, PMs are the greatest energy consumption [2]. In datacenters, VMs play an important role in reducing energy consumption. If we reduce energy in VMs, then it automatically reduces the energy in the associate server. Then, the third optimization target is to reduce energy consumption of VMs in datacenter. Each VM has two states, such as active (running and idle) and sleep state.
Let datacenter $d$ having $s$ number of server and each server having $j$ number of VMs and each VM having number of resources such as CPU, MIPS, memory, and bandwidth. We take $S_{ij}$ physical servers in datacenter $DC_{d}$, expressed as $\{DC_{d} | S_{1}, S_{2}, \ldots, S_{n}\}$. Then, the total number of servers in datacenter is $\sum_{i=1}^{n} S_{ij}$, and each server contains the total number of VMs, such as $\sum_{j=1}^{n} VM_{ij}^{total}$. Among all VMs, some of servers are in active state and rest are in sleep state. Let $VM_{active}^{n}$ be the total number of active VM, which contains all running mode and ideal mode. Total running VMs on the server at time $t$ is $\{VM_{run}^{1}, VM_{run}^{2}, \ldots, VM_{run}^{n}\}$. Then, the number of VM in in idle mode is $VM_{active}^{n} - VM_{run}^{n}$. The number of VM is in sleep condition within server is $VM_{sleep}^{n}$. Utilization of total resources among each VM is $VM_{run}^{n}$. Total active server ($S_{active}^{n}$) in a datacenter can be expressed as in the following equation:

$$\sum_{i=1}^{n} \sum_{j=1}^{n} \left( S_{run}^{1} \times VM_{run}^{n} + S_{idle}^{1} \times VM_{idle}^{n} \right). \tag{10}$$

In datacenter, running server ($S_{run}^{1}$) as well as idle server ($S_{idle}^{1}$) still consume energy. Then, total energy consumption in active state ($E_{cons}^{active}$) is represented in Equation (11). Decision variable $D_{f,s}$ is shown in Equation (12).

$$E_{cons}^{active} = S_{idle}^{1} \times VM_{idle}^{n} + \left( S_{run}^{1} \times VM_{run}^{n} \right) \times VM_{run}^{n} \times D_{f,s}. \tag{11}$$

$$DV_{f,s} = \begin{cases} 1, & \text{if } VM_{f,s} \text{ is allocated to } S_{i}, \\ 0, & \text{otherwise}. \end{cases} \tag{12}$$

To reduce energy consumption in datacenter, we should turn off the idle server that is no longer use but it maintains QoS. Whenever extra load come, then we activate the sleep server [20, 21]. To accomplish our objective, we should follow the two conditions that are described as bellows.

**Step 1.** Rest down inactive servers: In each time slot, if there may exist an idle or inactive server with no loads are assigned, then we put them into sleep mode. It is depending upon the following equilibrium:

$$\sum_{j=1}^{m} T_{i} \times VM_{u} \times D_{ij}^{f} < \sum_{n=1}^{n} S_{n} \times DV_{f,s}^{t}. \tag{13}$$

where $D_{ij}^{f} = 0$ which means no task is allocated to VM at time period $t$ and $DV_{f,s}^{t} = 1$ which means VM is allocated to server at time period $t$. $T_{i}$ is the number of incoming task or load. If all available VM utilization is less than total server utilization $S_{i}$ for the time period $t$ and this condition exists for longer time period, then we transfer this type of server into sleep mode, which is represented in the following equation:

$$\sum_{i=1}^{m} T_{i} \times VM_{u} \times D_{ij}^{f} > \sum_{n=1}^{n} S_{n} \times DV_{f,s}^{t}. \tag{15}$$

If Equation (15) is satisfied, then we should wake up the sleep server. It is represented in the following equation:

$$S_{w}^{t} = S_{w}^{t-1} \times DV_{w,s}^{t-1} \times transition{t-1}.$$

For handling such large amount of task, we wake up the sleep server into active server and it consumes some energy, which is denoted as $E_{cons}^{sleep}$. As per [21], sleep state server may consume approximate $1/3$ energy as compared to active state server. So, overall energy consumption ($E_{cons}^{total}$) by the datacenter is represented in the following equation:

$$E_{cons}^{total} = E_{cons}^{active} + E_{cons}^{sleep}. \tag{17}$$

Finally, our third objective function can be formulated as in the following equation:

$$O_{3} = \min \left( \sum_{i=1}^{n} E_{cons}^{total} \right). \tag{18}$$

2.5. Reward Function. The performance of cloud computing depends on the server, where it assigns the user request to provide required services, consume energy to run, and fully utilize all its resources. Due to the continuous nature of tasks in cloud computing, load balancing problem is figured as Markov decision process (MDP). In this process, VM is an agent and server is the environment, where VM takes an action by interacting with the server at each cycle. MDP can be represented as four-tuple ($S, A, P, R$), and these are described as follows [22]:

$S$ representing as state space: $s_{i} \in S$ denotes the state of the server at time period $t$.

$A$ representing as action space: $a_{i} \in A$ is the action through which VM takes at time period $t$.

$P:S \times A \rightarrow S$ is defined as the transition function. $P(s_{i}, a_{i}, s_{i+1})$ means when we take an action $a_{i}$ on state $s_{i}$, then we are reaching at next state $s_{i+1}$.

$R:S \times A \rightarrow R$ represents reward function. At the time period $t$, VM locates at current server state $s_{i}$ and takes an action $a_{i}$ to reach next state $s_{i+1}$ while receiving a reward $r_{i}$ which is considered in the following equation:

$$r_{i} = \min \left( O_{i} + \frac{O_{1} + O_{2}}{O_{2}} \right). \tag{19}$$

In RL, there are two important factors through which an agent can get the maximum reward, such as policy and value function. In policy function method, policy is set as $\Pi : S$
A and agent find the best policy \( (\Pi^* ) \) to take an action \( A \) on state \( S \) to get highest rewards in future. But finding the best policy, we need a value function. Value function shows that our policy is best or not to get the maximum reward. There are two kinds of value function: state value function and state-action value function [23, 24]. Common type of state value function \( (V^\Pi(s)) \) can be defined in the following equation:

\[
V^\Pi(s) = \mathbb{E} [R | s, \Pi] = \mathbb{E} \left\{ \sum_{t=0}^{\infty} \gamma^t r_t | s_t = s \right\},
\]

where \( \gamma \) is the discount factor, \( \gamma \in [0, 1] \), and the current state is \( s_t \). Commonly, this value function shows the expected discounted reward which depends on a policy \( \Pi \) and it is the combination of current reward with discounted future reward, known as Bellman condition [25]. Optimal state value function is represented in the following equation:

\[
V^*(s) = \max_\Pi V^\Pi(s) \forall s \in S.
\]

As like as state value function, state-action value function \( Q^\Pi(s, a) \) presents a value for selecting an action \( a \) on current state \( s \) that follows a policy \( \Pi \). State-action value is represented in the following equation:

\[
Q^\Pi(s, a) = \mathbb{E} [R | s, a, \Pi] = \mathbb{E} \left\{ \sum_{t=1}^{\infty} \gamma^t r_t | s_t = s, a_t = a \right\}.
\]

\( Q^\Pi(s, a) \) is similar to \( V^\Pi(S) \) except that by following the policy \( \Pi \), an initial action \( a \) is taken to reach the successive next state. Optimal state-action value function is shown in the following equation:

\[
Q^*(s, a) = \max_\Pi Q^\Pi(s, a) \quad \forall s \in S, \forall a \in A.
\]

To manage the continuous nature with getting the most elevated reward, we will probably track down the ideal policy \( \Pi \) which is based on probability distribution function [6, 26] and represented in the following equation:

\[
J(\Pi) = \mathbb{E} \{ Q^\Pi(s, a) \} = \sum_{s \in S} d^\Pi(s) \sum_{a \in A} \Pi(s, a) Q^\Pi(s, a) \text{dads},
\]

where \( d^\Pi(s) \) denotes the stationary distribution function under policy \( \Pi(s, a) \). The same as the state-action value function \( (Q^\Pi(s, a)) \), an advantage function \( (A^\Pi(s, a)) \) represents relative state-action values, which shows the selected action is valuable or not. Frequently, it is simpler to discover the suitable action for getting highest reward than other [27]. Advantage function can be represented in the following equation:

\[
A^\Pi(s, a) = Q^\Pi(s, a) - V^\Pi(s),
\]

where \( V^\Pi(s) \) shows the state value function or baseline function. An action is performed using an advantage function in value-based RL algorithms [28, 29]. Similarly in our methodology, we take an advantage value to take an action on each iteration.

2.6. Actor-Critic Method. Actor-critic is an advance version of RL algorithms in which both the value-based (i.e., Q-learning) and policy-based (i.e., policy gradient) methods are used. In AC method, each agent made out of two parts, i.e., actor part which is responsible to calculate the ideal policy by using policy gradient method and another is critic part, which is responsible to calculate the value function and calculate TD-error that demonstrates if the reward is good or not as the expectation. According to the ideal policy actor, select the action to get the reward, while critic is criticizing the current approach is better or not to get maximum reward from an environment.

2.7. Actor Part. Policy gradient (PG) method [26] is generally received in the actor part to refresh the defined policy for improve the target work in Equation (24). The policy \( \Pi(s, a) \) can be at first worked by utilizing the parameter vector \( \theta = \{ \theta_1, \theta_2, \ldots, \theta_T \} \), which is indicated by \( \Pi_\theta(s, a) = \text{pr}(s | a, \theta) \). Then, at that point, the gradient of the policy w.r.t \( \theta \) as far as the target work in Equation (24) is represented in the following equation:

\[
\nabla_\theta J(\Pi_\theta) = \sum_{s \in S} d(S) \sum_{a \in A} \nabla_\theta \Pi_\theta(s, a) Q^\Pi_\theta(s, a) \text{dads}.
\]

2.8. Critic Part. Objective of critic part is to find the approximate value function. This function can be used by actor to improve the policy [25]. The value function \( Q_\nu(s, a) \) is parameterized by \( \nu \) and represented in the following equation:

\[
Q_\nu(s, a) = \nu^T \Psi(s, a) = \sum_{j=1}^{n} \nu_j \Psi_j(s, a),
\]

where \( \Psi(s, a) = (\Psi_1(s, a), \Psi_2(s, a), \ldots, \Psi_n(s, a))^T \) is the basis function vector when agent chooses the action \( a \) at the state \( s \) and the parameter vector of weights is represented as \( \nu \).

To calculate the approximate value function, first of all, critic finds temporal difference (TD) error of Bellman equation [25, 26]. TD error is measured as distinction between the approximated value function and the actual value function at a current state-action situation. It is represented in the following equation:

\[
\delta_t = r_{t+1} + \gamma Q_\nu(s_{t+1}, a_{t+1}) - Q_\nu(s_t, a_t),
\]

where \( Q_\nu(s, a) \) is state-action value function and transition function is \( \{ s_t, a_t, r_{t+1}, s_{t+1}, a_{t+1} \} \). By using TD error, the critic parameter \( \nu \) can be updated by the gradient-descent update rule [30] which is represented in the
following equation:

$$v_{t+1} = v_t + \beta c_1 \frac{dQ_c(s,a)}{dv} s = s_t, a = a_t, \quad (29)$$

where $\beta e [0, 1]$ is learning rate of critic value function and the feature of value function is represented as $dQ_c(s,v) / dv$. After optimizing the parameter $v$ in Equation (29), actor updates its value function in Equation (27).

2.9. HPSOAC Scheduling Policy for Parameter Updating. This segment shows the essential system of our proposed HPSOAC scheduling policy method. In this method, each agent can get their individual reward by taking appropriate action in each state of an environment. After getting their individual best, they want to get the global best reward by exchanging their information with their neighbours by applying PSO algorithm. Thus, our proposed method selects the action to optimize both the policy and value function to get the personal best reward for each VM. After that, each VM can find the global best by sharing their information among others. Finally, all agents update their parameters $\{\theta_j, v_j\}$ according to the respective iteration. Figure 2 shows our proposed HPSOAC scheduling policy. To improve the convergence speed for searching for the global best value in actor-critic algorithm, we combine PSO with actor-critic method to investigate policy parameters.

PSO is a swarm-based optimization computation procedure through which numbers of particles with various positions and velocities are built to discover solutions [31, 32]. Similarly, all agents in HPSOAC algorithm have different weights of load and different velocities of task transfer rate for getting the optimum result. The velocities of the task transfer rate determine the weight of the load. All learning parameters are updated as follows:

2.10. Action Selection. In cloud computing, all incoming tasks should be allocated to suitable VMs for executing in minimum time. It depends upon the capacity of the VM that is represented in Equation (30). Load of the VM is calculated in Equation (31). After allocating the task, some of VMs may be overloaded or underloaded. If the load of a VM exceeds the capacity of VM, then it is overloaded. To balance the load among VM, HPSOAC scheduling policy finds the best VM, i.e., either under load or ideal VM and then it transfers the extra task by taking the selected action. Task transfer speed can be calculated by using Equation (32). Thus, action selection depends upon current load of VM $(VM_{load})$ and task transferred $(T_{tt})$ to give maximum reward. Then, the selected action can be calculated by using Equation (33).

$$VM_{cap} = VM_{pr} + VM_{b}, \quad (30)$$

$$VM_{load} = \frac{T_i \times T_{len} \times T_{filesize}}{VM_{pr}}, \quad (31)$$

$$T_{tt} = \sum_{i=1}^{m} \sum_{b=1}^{n} \sum_{c=1}^{n} (1 - DV_{ib,c} DV_{ic}) \frac{ETT_{bc}}{BW_{bc}}, \quad (32)$$

where $ETT_{bc}$ is the extra task transfer between VM b and VM c. $BW_{bc}$ is the required bandwidth between two VM b and VM c in MIPS. $DV_{ib,c} = 1$ means ith task has taken from VM b and assigned to the VM c.

$$a_i = P \times VM_{load} + V \times T_{tt}, \quad (33)$$

where $P$ and $V$ can be defined as the weight of VM load and task transfer speed in which value of $V = 1 - P$. Thus, action selection is depending the set of weight $P_j$ and $V_j$ of an agent for getting maximum reward, i.e., mentioned in sub-section Update Velocity and Position.
HPSOAC Algorithm

Input: Server set as $S = \{S_1, S_2, \cdots, S_x\}$, VM set as $VM = \{VM_1, VM_2, \cdots, VM_n\}$ and Task set as $T = \{T_1, T_2, \cdots, T_m\}$ with $\gamma, \lambda, \beta_c, \beta_a$.

Output: Reduce makespan Time, reduce energy consumption, increase usage of resource and balanced VM load.

Assumption:
- Particle or Agent VM
- Position Load of VM allocated on PM
- Velocity Speed of task transfer
- Pbest Individual VM performance
- Gbest Optimal result
- Iteration or Episode Time period
- Information of Server and VM MIPS, Memory, CPU and Bandwidth
- Information of Task Length and File size

Start
For $i = 1$ to $m$
For $j = 1$ to $n$
Incoming task is allocated on VM
  // For Makespan
  Calculate processing rate of VM, expected finishing time and task allocation time by applying Eq. (1), (2) and (3).
  $VM_{pr} = VM_{mips} + VM_{cpu} + VM_{mem}$
  $EFT_{i,j} = \frac{T_{len}}{VM_{pr}}$
  $TAT_{i,j} = \frac{T_{filesize}}{VM_{bw}}$
  Find Finishing Time of each task on VM by the help of Eq. (4):
  $FT_{i,j} = EFT_{i,j} + TAT_{i,j}$
  Calculate Makespan Time by utilizing Eq. (6):
  $MST = \max \{FT_{i,j}\}$
  // For Resource Utilization
  Compute Resource Utilization by using Eq. (8):
  $R_{util} = \frac{FT_{i,j}}{MST}$
  End of for
End of for
  // For Energy Consumption
  Initialize active and sleep VM, utilization of both VM and server.
  For $j = 1$ to $n$
  For $s = 1$ to $x$
  Compute active server by using Eq. (10):
  $\gamma_{run} \times VM_{run}^{s} + \gamma_{idle} \times VM_{idle}^{s}$
  Compute energy consumption of active server by using Eq. (11):
  $E_{cons}^{active} = \gamma_{idle} \times VM_{idle}^{s} + (\gamma_{run} \times VM_{run}^{s}) \times VM_{u}$
  If ($VM_{u} T_{i,j} < S_{s}$)
    Put the server into sleep mode
  Else
    Wake-up the sleep server
  End of if
  Compute total energy consumption by using Eq. (17):
  $E_{total}^{cons} = E_{cons}^{active} + E_{cons}^{sleep}$
  End of for
End of for
  // Applying HPSOAC
  // Initialize AC parameters
  Initialize $j, c, Q_i, Q_j, Q_o(s, a)$ and $\Pi_i(s, a)$
  // Initialize PSO parameters
  Initialize the $w, \epsilon_1, \epsilon_2$, population size, the number of iterations, $r_1$ and $r_2$
  For $j = 1$ to $n$
  Initialize individual best ($R_{pbest,j}$)0, current position and velocity of agent $j (VM_j)$
  End of for
  Initialize individual best ($R_{gbest,j}$)0, target position
  For $e = 1$ to $e_{max}$
  Pseudocode 1: Continued.
2.11. Critic Parameter Update. Due to late response of the RL approach, it will impact on the result of the current reward and aggregate reward. Therefore, Equation (29) is not sufficient to produce an optimum result. Therefore, eligibility trace technique is used to accelerate the convergence rate [26]. In this technique, only qualified states and actions are taking interest in increasing the speed. The updated eligibility trace vector $e_{t+1}$ is represented by the following equation:

$$e_{t+1}(s, a) = \begin{cases} 
1 & \text{if } s = s_t \text{ and } a = a_t, \\
\lambda \lambda e_t(s, a) + \Psi(s, a) & \text{otherwise,}
\end{cases}$$  
(34)

where $\lambda \in [0, 1]$ is trace-decay rate. This eligibility trace technique is used to update critic parameter vector $v$ can

<table>
<thead>
<tr>
<th>Term</th>
<th>Meaning</th>
<th>Term</th>
<th>Meaning</th>
</tr>
</thead>
<tbody>
<tr>
<td>$j$</td>
<td>Agent</td>
<td>$Q_v(s, a)$</td>
<td>State-value function</td>
</tr>
<tr>
<td>$s_t$</td>
<td>Initial state</td>
<td>$\Pi_\theta(s, a)$</td>
<td>Policy function</td>
</tr>
<tr>
<td>$s_{t+1}$</td>
<td>Next state</td>
<td>$\beta_a$</td>
<td>Actor learning rate</td>
</tr>
<tr>
<td>$e$ &amp; $t$</td>
<td>Episode &amp; iteration</td>
<td>$\beta_c$</td>
<td>Critic learning rate</td>
</tr>
<tr>
<td>$\theta_t$</td>
<td>Actor parameter</td>
<td>$\delta_t$</td>
<td>TD error</td>
</tr>
<tr>
<td>$v_t$</td>
<td>Critic parameter</td>
<td>$A^\Pi(s, a)$</td>
<td>Advantage function</td>
</tr>
<tr>
<td>$\epsilon_t$</td>
<td>Eligibility trace</td>
<td>$\nabla_\theta(\Pi_\theta)$</td>
<td>Policy gradient</td>
</tr>
<tr>
<td>$\lambda$</td>
<td>Trace decay rate</td>
<td>$\gamma$</td>
<td>Discount factor</td>
</tr>
<tr>
<td>$R_t$</td>
<td>Reward</td>
<td>$R_{j}$</td>
<td>Fitness value</td>
</tr>
<tr>
<td>$R_{\text{best},i}$</td>
<td>Personal best</td>
<td>$R_{\text{best},i}$</td>
<td>Global best</td>
</tr>
</tbody>
</table>
be expressed as in the following equation:

\[ v_{t+1} = v_t + \beta \delta_t \epsilon_{t+1}(s,a). \]  

(35)

2.12. Actor Parameter Update. Some error and bias are still present in approximate policy gradient method that is not give an optimum result. But the error and bias can be removed by the help of compatible approximation function \[ f(s,a) \] is zero, which is represented in Equation (38).

\[ \nabla_{s} Q_{\theta}(s,a) = \frac{1}{2} \left[ Q_{\theta}(s,a) - Q_{\theta}(s,a) \right]^2, \]  

(37)

Then, \( Q_{\theta}(s,a) \) can substitute \( Q_{\theta}(s,a) \) directly into Equation (26) and gradient of policy with compatible approximation function is became as in Equation (39), to reduce bias and error.

\[ \nabla_{\theta} j(\Pi_{\theta}) = \sum_{s \in S} \sum_{a \in A} \nabla_{\theta} \Pi_{\theta}(s,a) Q_\theta(s,a) d a d s. \]  

(39)

After reducing the error in Equation (37), actor updates its state-value function as in Equation (40).

\[ Q_\theta(s,a) = v^T \Psi(s,a). \]  

(40)

Table 3: Task, server, and VM properties.

<table>
<thead>
<tr>
<th>Task properties Parameters</th>
<th>Value</th>
<th>Server properties Parameters</th>
<th>Value</th>
<th>VM properties Parameters</th>
<th>Value</th>
</tr>
</thead>
<tbody>
<tr>
<td>Task range</td>
<td>10000-50000</td>
<td>Server range</td>
<td>10-50</td>
<td>VM range</td>
<td>50-500</td>
</tr>
<tr>
<td>Length</td>
<td>100000-600000</td>
<td>Processing speed (MIPS)</td>
<td>30000</td>
<td>Processing speed (MIPS)</td>
<td>2000-3000</td>
</tr>
<tr>
<td>File size</td>
<td>300</td>
<td>Memory</td>
<td>15000</td>
<td>Memory</td>
<td>256-512</td>
</tr>
<tr>
<td></td>
<td></td>
<td>CPU</td>
<td>10</td>
<td>CPU</td>
<td>5</td>
</tr>
<tr>
<td></td>
<td></td>
<td>Bandwidth</td>
<td>8000</td>
<td>Bandwidth</td>
<td>5000</td>
</tr>
</tbody>
</table>

Table 4: PSO and AC properties.

<table>
<thead>
<tr>
<th>PSO properties Parameters</th>
<th>Value</th>
<th>AC properties Parameters</th>
<th>Value</th>
</tr>
</thead>
<tbody>
<tr>
<td>Number of particle</td>
<td>30</td>
<td>Maximum episodes (T)</td>
<td>1000</td>
</tr>
<tr>
<td>Maximum iteration (T)</td>
<td>1000</td>
<td>Actor’s learning rate</td>
<td>0.02</td>
</tr>
<tr>
<td>c1 &amp; c2</td>
<td>2</td>
<td>Critic’s learning rate</td>
<td>0.02</td>
</tr>
<tr>
<td>r1 &amp; r2</td>
<td>[0.1]</td>
<td>Discount factor</td>
<td>0.95</td>
</tr>
<tr>
<td>ω</td>
<td>0.73</td>
<td>Trace decay factor</td>
<td>0.3</td>
</tr>
</tbody>
</table>

2.13. Update Velocity and Position. To maintain the load balance and achieve the objective function, each agent searches for the optimum result within the environment. Each agent can adjust their load by depending on their own arrangement as well as their neighbours. If an agent is heavily loaded, then it looks at its neighbours and transfers the extra task to the next suitable agent. Therefore, the velocity of task transfer depends on the weight of load. Suppose there are \( n \) number of VMs or agents in an environment, and \( t \) number of iterations called episode is available. Let \( P_j \) and \( V_j \) indicate the weightage of load, and task transfer speed or velocity of each VM is represented as \( P_j = \{ P_{j1}, P_{j2}, \ldots, P_{jn} \} \) and \( V_j = \{ V_{j1}, V_{j2}, \ldots, V_{jn} \} \). At each iteration \( t \), an agent communicates with the environment and updates their weightage of load with velocity according to their neighbours’ best value. The updated load and velocity of each agent is represented in the following equations:

\[ P_j(t+1) = P_j(t) + V_j(t+1), \]  

(43)

\[ V_j(t+1) = \omega V_j(t) + c_1 r_1 (P_{best} - P_j(t)) + c_2 r_2 (G_{best} - P_j(t)). \]  

(44)

By the help of state action value functions parameter \( \nu \), actor is searching the best parameter \( \theta \) to increase the expected reward \( J(\Pi_{\theta}) \) under the given policy \( \Pi_{\theta} \) [25]. Finally, the update policy parameter for maximizing the reward is represented in Equation (42), where \( \beta_a \) denotes the learning rate of the actor.

\[ \theta_{t+1} = \theta_t + \beta_a \nabla_{\theta} j(\Pi_{\theta}) \]  

(42)
2.14. Pseudocode of HPSOAC Algorithm. Pseudocode of our proposed algorithm is represented in Pseudocode 1. Terms and meaning are shown in Table 2.

2.15. Experimental Analysis. Based on simulation results, this section evaluates the performance of our proposed HPSOAC scheduling policy compared with other existing policy with respect to different load balancing parameters such as energy consumption, makespan time, and resource utilization. All simulation experiments were conducted in a Python environment with TensorFlow. The detailed information of our experiment is mentioned below.

2.16. Parameters Used for Simulation. This section provides necessary information for carrying out the simulation. For the test, we take continuous and independent tasks to get maximum reward from the environment. Tasks are distributed between number of VMs, and these VMs are assigned to a number of servers in a datacenter. Here we take 10000 to 50000 tasks that are distributed among 50 to 500 virtual machines in a datacenter. These tasks have independent length and file size. Our proposed scheduling policy is aimed at equally distributing the whole task among the available virtual machines, which can reduce the makespan time as well as energy consumption but increase the resource utilization. If a virtual machine is heavily loaded, then the extra task should be managed easily. That makes all the machines balanced. We take these objectives as our reward rate. Initially, our proposed scheduling policy is to rapidly increase to get the higher reward, but whenever it reaches iteration number 600, it is a little bit fluctuated and remains unchanged until the last iteration 1000. In our experiment, we take a total number of iterations of 1000 because after 1000 iterations, there is no change in our reward percentage. All the properties of task, server, and VM are shown in Table 3. Table 4 shows different properties of PSO and AC approach.

3. Result and Analysis

This section shows the performance of HPSOAC scheduling policy against two other optimization policies such as Deep Q-network (DQN) and Q-learning based on Modified Particle Swarm Optimization (QMPSO) algorithm. All simulation results are shown in Figures 3–8.

Figure 3 shows the total reward percentage per iteration under the different scheduling policy in our environment where the number of servers is 50 and tasks are entered into the environment at different lengths and speeds. From this figure, we can find that our proposed HPSOAC scheduling policy shows the best reward performance with a faster convergence speed and better performance as compared to other two policies. To get the optimum result, both DQN and QMPSO policies required more learning steps. As compared with QMPSO, DQN has better convergence speed but it is not sufficient. At the start of iteration, the percentage of reward of all algorithms is very similar but from the iteration number 20 to till the last iteration, our proposed scheduling algorithm gives better reward as compared to its competitors. When it reaches iteration 600, then the percentage of reward is near about 80% and it remains almost the same at the end of iteration 1000. But in the case of both DQN and QMPSO algorithms, the percentage of reward is near about 60% and 39%, respectively.

Figure 4 represents energy consumed by different scheduling policies where different numbers of tasks are taken with a fixed number of VMs. From this figure, it is concluded that when the number of tasks is increases, then
energy consumption of DQN and QMPSO grows faster as compared to HPSOAC. At a particular time period, HPSOAC scheduling policy shows in marginally increase the energy consumption when number of tasks increases to 4000 to 5000. Finally, at the end of iteration, our proposed algorithm takes approximately 5.16% and 10.86% less energy consumed as compared to DQN and QMPSO, respectively. Table 5 shows the dataset for Figure 4.

Figure 5 represents energy consumed by different servers in a datacenter where each server can be virtualized by 10 number of VMs. In this figure, we take the total energy consumption that we get from the simulation for our proposed scheduling policy with respect to number of servers. From this figure, we found that a smaller number of servers can consume more energy as compared with a greater number of servers. Table 6 shows the dataset for Figure 5.

Table 7 represents the datasets for the total amount of energy consumed by our proposed scheduling policy that can be shared between two different states of the server, i.e., active state and sleep state. Figure 6 shows the energy consumption at different states of the server. From this figure, we take the total energy that can be consumed by...
HPSOAC scheduling policy for both active and sleep state of the server and found that majority of the energy is utilized by the servers in active state as compared to sleep state.

The outcome displayed in Figure 7 is obtained by various algorithms having 1000 to 5000 number of tasks, allocated to available VMs. From this figure, our proposed HPSOAC scheduling policy takes less makespan time than other two calculations. Percentage comparison of the lowest makespan time is approximately 3.02% and 5.04%, and percentage comparison of the highest makespan time is 7.13% and 10.04%, respectively, as compared with DQN and QMPSO policies. Dataset for Figure 7 is displayed in Table 8.

Total utilization of resources depends on the active number of running servers in datacenter, and the utilization result is shown in Figure 8. From the figure, it is clear that our proposed HPSOAC scheduling policy is increased resource utilization as compared to DQN and QMPSO. In this experiment, we take different sets of VMs, such as 50
to 500 and executed over maximum load. From 50 to 300 VMs, our proposed approach is marginally more but when the number of VMs increases with maximum load, then our proposed scheduling policy shows better resource utilization compared with two other techniques.

4. Conclusion

Balancing load in a cloud environment has been a challenging issue due to its dynamic nature. This paper proposes a hybrid scheduling policy which is the combination of both PSO algorithm and actor-critic algorithm named Hybrid Particle Swarm Optimization Actor Critic (HPSOAC). This scheduling policy is proposed to achieve improvement in various parameters of load balancing such as makespan, utilization of resource, and energy consumed in continuous state-action space. Our simulation experiment is done with the help of Python simulator with TensorFlow. Experimental results show that HPSOAC scheduling policy is used to reduce 5.16% and 10.86% energy consumption, reduce 7.13% and 10.04% makespan time, and have marginally better resource utilization than DQN and QMPSO algorithms, respectively.
In future, improving the resource allocation and resource management concept cloud be focused on cloud datacenter. To achieve this, a hybrid algorithm cloud be proposed which is the combination of two different machine learning algorithms. This algorithm will provide real-time analytics in the complex and dynamic cloud network.

**Data Availability**

The data used to support the findings of this study are included within the article.

**Conflicts of Interest**

The authors declare that there is no conflict of interest regarding the publication of this paper.

**Acknowledgments**

This work was supported by Dongseo University, “Dongseo Cluster Project” Research Fund of 2022 (DSU-20220006).

**References**


