Proximal Policy Optimization Based Job Scheduling Algorithm for Cloud Data Centers

Aman Singh

Dept. of Computer Science and Engineering
IIIT Naya Raipur, India
aman20100@iiitnr.edu.in

Kshitij Kumar Singh Chauhan

Dept. of Computer Science and Engineering
IIIT Naya Raipur, India
kshitij20100@iiitnr.edu.in

ABSTRACT

This paper presents an algorithm for job scheduling in cloud data centers based on deep reinforcement learning. This approach is in contrast to traditional heuristics, which often struggle to adapt to changing environments and optimize for specific workloads. Our algorithm operates within the bin packing problem framework and automatically learns a fitness calculation method to minimize makespan and maximize throughput based on experience. Through a trace-driven simulation, we have shown that our algorithm converges and generalizes well, while also highlighting the underlying learning mechanisms. Compared to traditional heuristic-based job scheduling algorithms, our approach outperforms them in terms of performance, indicating the effectiveness of our work. We believe that our algorithm has the potential to revolutionize job scheduling in cloud data centers by automatically adapting to changing workloads and achieving optimal performance.

KEYWORDS: Job-Scheduling, DRL, Proxy-Policy Optimization

1. INTRODUCTION

Our project focuses on addressing the resource management problem in cloud data centers through a job scheduling algorithm based on deep reinforcement learning. Traditional heuristics prioritize generality and ease of implementation over achieving ideal performance on specific workloads, often requiring careful testing and adjustment. However, our approach, based on reinforcement learning, can learn decision-making policies directly from experience and adapt to the dynamic environment of cloud data centers.

To solve the problem of job scheduling in cloud data centers which may require dynamic decisions, we need to consider multiple resource types and assign tasks to the fewest number of machines to maximize throughput and minimize makespan, while also keeping average

completion and average slowdown in check. This is analogous to the multi-dimensional bin packing problem, which is known to be computationally complex and typically requires the use of heuristic algorithms.

In our work, reinforcement learning has been used to automatically learn a fitness calculation method for optimizing performance **[7]**. By designing a suitable reward function that reflects our optimization goals, the reinforcement learning model can learn how to assign tasks to machines based on their resource requirements in a way that minimizes makespan. This approach is more flexible and adaptable than handcrafted fitness calculation methods, which require manual adjustment of parameters and may not always result in optimal solutions.

Our algorithm has been built within the bin packing problem framework **[12]**, where fitness is calculated to maximize throughput by matching tasks to machines. Our algorithm automatically obtains a fitness calculation method through deep reinforcement learning, aiming to minimize the makespan of a set of jobs. We tested our algorithm on a dataset of 31,756 jobs and found it to be successful, outperforming traditional heuristic-based job scheduling algorithms.

Our approach differs from previous attempts to design job scheduling algorithms based on reinforcement learning, such as DeepRM and other multi-resource multi-machine environments **[10] [15]**, as we embed our algorithm in the bin packing problem framework. Through trace-driven simulations, we have shown that our algorithm converges and generalizes well while reducing the average completion and average slowdown. We have also improved upon DeepJS using Proximal Policy Optimization instead of Policy Gradients and a modified reward function that takes both average completion and average slowdown into consideration.

2. MOTIVATION

The job scheduling problem is traditionally addressed using heuristic algorithms such as first-fit and best-fit, which are fast but often do not have an optimal solution **[13]**. For example, Tetris is an algorithm that projects task requirements and machine available resources into Euclidean space and picks the pair with the highest dot product value. However, only tasks whose requirements are satisfiable are considered. While the dot product calculation is reasonable, it may not always result in the optimal solution **[8]**.

These heuristic algorithms prioritize generality, ease of understanding, and straightforward implementation over achieving ideal performance on a specific workload. They are designed to work reasonably well on average, but may perform poorly on certain workloads.

In industry, even simpler fitness calculation methods are often applied, which are general for the workload **[17]**. Although some parameters can be adjusted to adapt to workloads with different characteristics, the adjustment of the parameters relies on the operator's experience. Due to the complexity of the relationship between parameters and optimization goals, this process can be cumbersome, and the improvement is not guaranteed.

Reinforcement learning offers an alternative approach to traditional heuristic methods, as it can automatically learn a fitness calculation method for a specific workload **[9]**. By designing a suitable reward function that reflects the goal of performance optimization, the reinforcement learning model can learn the optimal decision-making policy directly from experience and

adapt to the dynamic environment. Compared to traditional heuristic methods, reinforcement learning can achieve better performance by exploiting workload-specific patterns that heuristic algorithms may overlook.

3. LITERATURE SURVEY

AGH+QL **[1]**, a novel revised Q-learning-based model, takes hash codes as input states with a reduced size of state space. It uses anchor graph hashing which can accelerate training. Hash codes have been used to reduce the size of state space. However, lack of a DRL based feedback leads to a compromised adaptability to dynamic workloads.

QEEC **[2]** is a Q-learning based task scheduling framework for energy-efficient Cloud computing using Q-value table to express the decision maker of action. It reduces the average waiting time of task using dynamic task ordering strategy to promote the quality of Cloud services. However, it lacks queueing model to satisfy realistic scenes.

The DeepRM_Plus **[3]** uses a neural network that has convolution neural network (CNN) of six layers to describe the mapper of decision based on the great success of DNN in image processing. Data centre cluster, waiting queues, and backlog queue compose the state of environment which is represented by image. It uses imitating learning to accelerate convergence and CNN to capture the state of resource. It lacks analytical capability for state-recognition. Alternative policies such as Actor-Critic network and DDPG can be explored.

DERP **[4]** uses three different approaches of a DRL agent to handle the multi-dimensional state and to provide elastic VM resource. It does not demand space Partitioning. It requires a intercommunicate framework of simple DRL, full DRL and DDRL for performance boosting.

DPM framework **[5]** based on RL, adopts the long short-term memory (LSTM) network Deep Reinforcement Learning-based Methods for Resource Scheduling in Cloud Computing: A Review and Future Directions to capture the prediction results and uses DRL to train the strategy of resource allocation aimed at reducing energy consumption in Cloud environment. It uses LSTM Network to predict workload which can eliminate the vanishing gradient problem. Energy efficiency remains an issue.

DDQN **[6]**, Duelling deep Q-network, contains a set of convolutional neural networks and full connected layer to achieve higher efficiency of data processing, lower network cost, and better security of data interaction. It can keep stability of reward however it is not efficient in terms of energy.

In context of designing a DRL-based scheduling algorithm following challenges were identified from the literature survey:

1. **High computational complexity:** DRL requires large computing power and can be very complex, especially for multi-cluster or large-scale systems. This computational complexity needs to be optimized to make it practical for real-world applications.
2. **Unpredictable results:** The scheduling results based on DRL can still be unpredictable, making it difficult to evaluate worst-case performance.
3. **Dynamic task prediction:** Real scheduling often depends on the prediction of dynamic tasks without prior knowledge or pre-emption, making it challenging to design effective DRL algorithms.
4. **Local optimization:** Gradient descent algorithm used in DRL or Bellman Equation used in Q-learning have inherent limitations that can lead to local optimization rather than global optimization.
5. **Lack of explainability:** The training process in DRL can be challenging to explain using mathematical techniques, especially when dealing with high-dimensional continuous state space. This makes it difficult to model and derive theoretical insights from DRL algorithms.

4. METHODOLOGY

4.1. Dataset

An open-source data consisting of 31755 job instances was used for simulation and testing. Simulation involved multiple clusters that worked upon dataset attributes vis. A vi, submit time, duration, cpu, memory, job_id, task_id, instances number to obtain an optimal scheduling of for jobs in the dataset.

4.2. Proximal-Policy Optimization

Policy gradients are a popular approach in RL-based job scheduling, but they suffer from significant drawbacks that hinder their effectiveness:

- Firstly, sample inefficiency is a significant concern, as samples are only used once, and the policy is updated, making sampling expensive. The problem is further compounded as old samples become irrelevant after a large policy update, which can be prohibitive.
- Secondly, inconsistent policy updates are another critical issue. Policy updates often overshoot or miss the reward peak, or stall prematurely, particularly in neural network architectures **[11]**. This can lead to poor convergence and difficulty in achieving optimal performance.
- Thirdly, high reward variance is a common problem with policy gradients. Since policy gradient is a Monte Carlo learning approach that takes into account the entire reward trajectory (i.e., a full episode), the variance of the reward can be high, making convergence difficult.

Overall, these issues significantly impact the performance of policy gradients in RL-based job scheduling, and alternative approaches must be considered to mitigate their effects.

Proxy policy optimization (PPO) is a popular policy-based reinforcement learning algorithm that addresses many of the issues associated with traditional policy gradient methods, such as sample inefficiency, inconsistent policy updates, and high reward variance. In our work on job scheduling optimization, we use PPO to improve the performance of our RL-based approach. One of the main benefits of PPO is its use of a revised objective that prevents the policy from changing frequently between updates. Specifically, PPO constrains the policy update by limiting the ratio between the new and old policy probabilities, ensuring that the updated policy does not deviate too far from the old policy **[14]**. This helps to prevent inconsistent policy updates and ensures that the algorithm can recover from poor updates.

In addition, PPO may also be complemented with the use of a clipped objective that further improves stability and reduces reward variance. The clipped objective constrains the policy update to a fixed range around the old policy, effectively limiting the impact of any given update. This helps to reduce the impact of high-variance rewards and ensures that the algorithm converges more reliably.

Furthermore, PPO uses a form of importance sampling that re-weights the gradient updates based on the probability of the actions under the new and old policies. This helps to reduce sample inefficiency by re-using data from previous trajectories, which can significantly speed up the learning process and reduce the total number of samples required. By using PPO in our RL-based job scheduling approach, we are able to improve the stability, efficiency, and reliability of the learning process. This allows us to more effectively optimize our policy, reducing makespan and improving overall system performance.

4.3. Design of Algorithms: Proximal Policy Optimization (PPO) Training Loop for Reinforcement Learning Agent

Proposed algorithm initializes an Agent class that is used to train an RL agent using the Proximal Policy Optimization (PPO) algorithm. PPO improves on the vanilla policy gradient by using a clipped surrogate objective function to prevent the new policy from diverging too far from the old policy in each update **[16].**

The Agent class has several parameters that can be customized for training, such as the discount factor gamma, whether to use reward-to-go or not, whether to use a neural network baseline or not, and whether to normalize advantages or not. The train method of the Agent class takes an environment as input and uses it to train the agent. The training proceeds for a maximum number of episodes specified by the max_episodes parameter. In each episode, the agent interacts with the environment by taking actions based on the current policy, storing the transition tuple (observation, action, reward, next observation, done) in a replay buffer, and updating the policy using the PPO algorithm.

Input: name, brain, gamma, max_kl, lr, reward_to_go, nn_baseline, normalize advantages, epsilon, epochs, batch_size, model_save_path, summary_path, max_episodes, target_reward

Table 1 provides a description of each of the listed parameter.

Input Parameter	Remark
name	name of the agent
brain	neural network model for the agent's policy
gamma	discount factor for future rewards
max_kl	maximum KL divergence between old and new policy
lr	learning rate for policy and value network
reward_to_go	flag to use reward-to-go instead of for calculating advantages
nn_baseline	flag to use a neural network baseline for advantage normalization
normalize_advantages	flag to normalize advantages
epsilon	clipping parameter for PPO loss
epochs	number of epochs to update the policy and value network
batch_size	number of transitions to sample per update
model_save_path	path to save trained model
summary_path	path to save TensorBoard summaries
max_episodes	maximum number of episodes to train for
target_reward	target average reward to stop training early

Table 1: Description of Input Parameters

Output: trained agent

Algorithm has been divided into six modules for better comprehension.

4.3.1 Module a: Initialize and Setup

This module sets up the necessary components for the reinforcement learning agent. It initializes a summary writer for logging to TensorBoard, creates an empty list to store episode rewards, initializes the policy and value networks with the provided brain, and initializes the optimizer for the policy and value networks with the provided learning rate.

1. **Create** a summary writer
2. **Set** empty list **->** episode rewards
3. **Set** policy, value network
4. **Set** optimizer for the policy

4.3.2 Module b: Main Loop

The main loop is where the agent interacts with the environment. It repeats for a specified number of episodes, each time resetting the environment to obtain the initial observation, setting the done flag to False, initializing the episode reward to 0, and initializing an empty list to store transitions for the current episode. It then repeats until the environment signals that the episode is done.

```
5.  Repeat for i in range(max_episodes):
6.      Reset env Collect initial observation
7.      Set done flag -> False
8.      Set episode reward -> 0
9.      Create empty list Load transitions_current_episode
```

4.3.3 Module c: Execute Agent Actions

In this module, the agent executes actions in the environment based on the current state of the policy and value networks. It predicts an action and value estimate using the agent's policy, takes the action, and obtains the next observation, reward, and done flag. It stores the transition in the current episode's list of transitions, updates the observation to the next observation, and adds the reward to the episode reward.

```
10.     Repeat While not done:
11.         Estimate action, value Calling agent's policy
12.         Collect next observation, reward, done flag
13.         Append transition to current_episode_list
14.         Update observation -> the next observation
15.         episode_reward = episode_reward + reward
```

4.3.4 Module d: Update Policy and Value Networks

The policy and value networks are updated in this module based on the collected transitions from the current episode. If the current episode number is a multiple of the batch size, the advantages and value targets for the collected transitions are computed, the transitions are flattened and their log probabilities under the old policy are computed, and the policy is updated using the clipped PPO loss. If nn_baseline is True, the value network is updated using the mean squared error loss. The list of transitions for the current episode is then cleared.

```
16.         Append episode reward to episode_reward_list
17.         If reward > target reward
18.             Print message
19.             Break
20.         If current_episode_number % batch_size = 0:
21.             Load advantages, value targets from collections
22.             Flatten transitions
23.             Store log_probabilities
24.             Update policy -> clipped PPO loss
25.         If Boolean(nn_baseline) = True:
```

```
            Update value network -> mean squared error loss
        Clear list_of_transitions
```

4.3.5 Module e: Print and Log Results

This module calculates the average reward over the last 100 episodes and prints the current episode number, episode reward, and average reward. If the current episode number is a multiple of 100, it saves the current model to the model save path (if specified) and logs the current episode reward and average reward to TensorBoard.

```
    For episodes in range 1 to 100:
        Compute average_reward
    Print current_episode_number, episode_reward, and average_reward
    If current episode number % 100 = 0:
        If find(model_save_path):
            Save current_model -> model_save_path
        Log current_episode_reward, average_reward
```

4.3.6 Module f: Stop Condition and Clean-up

This module checks if the maximum number of episodes has been reached and prints a message if it has. It then closes the environment.

```
    If current_episode_number = maximum_episode_number:
        Print message
        Exit
Close env
```

4.4. Design of reward function

The ModifiedRewardGiver function is hierarchically designed, as in Fig. 1, to provide a reward signal to a reinforcement learning agent based on the performance of a simulation. In particular, the reward function is designed to incorporate both average completion and average slowdown in the feedback signal, which can help the agent learn to minimize both completion time and slowdown.

Fig 1: Hierarchical design of reward function

The reward function calculates a reward value based on the negative reciprocal of each unfinished task's duration. This calculation effectively penalizes longer-running tasks, incentivizing the agent to complete tasks quickly. Additionally, the reward value is adjusted based on the number of unfinished tasks and a constant factor of -0.5. This adjustment ensures that the agent is incentivized to complete tasks even when there are a large number of unfinished tasks.

ModifiedRewardGiver function aids training reinforcement learning agents in simulated environments, allowing them to learn to balance completion time and slowdown in a way that maximizes performance.

4.6. Design of neural network

Neural network has 5 hidden layers with 9, 9, 18, 36, and 9 neurons respectively, and an output layer with 1 neuron. It uses the hyperbolic tangent (tanh) activation function in all hidden layers except the output layer, which has no activation function.

Layer	Output Shape	Parameter #	Activation Function
Input Layer	(None, 3)	0	None
Hidden Layer 1	(None, 9)	3 * 9 + 9	tanh
Hidden Layer 2	(None, 9)	9 * 9 + 9	tanh
Hidden Layer 3	(None, 18)	9 * 18 + 18	tanh
Hidden Layer 4	(None, 36)	18 * 36 + 36	tanh

Hidden Layer 5	(None, 9)	36 * 9 + 9	tanh
Output Layer	(None, 1)	9 * 1 + 1	None

Table 1: Neural network architecture

5. RESULTS

Algorithm proposed in this paper outperforms existing peer algorithms on slowdown and completion metrics. Makespan reduction is also improved in comparison to heuristic algorithms as evident from Table 2.

Algorithm	Average Makespan (seconds)
Ours	626
First Fit	680
Tetris	689

Table 2: Makespan comparison against heuristic algorithms

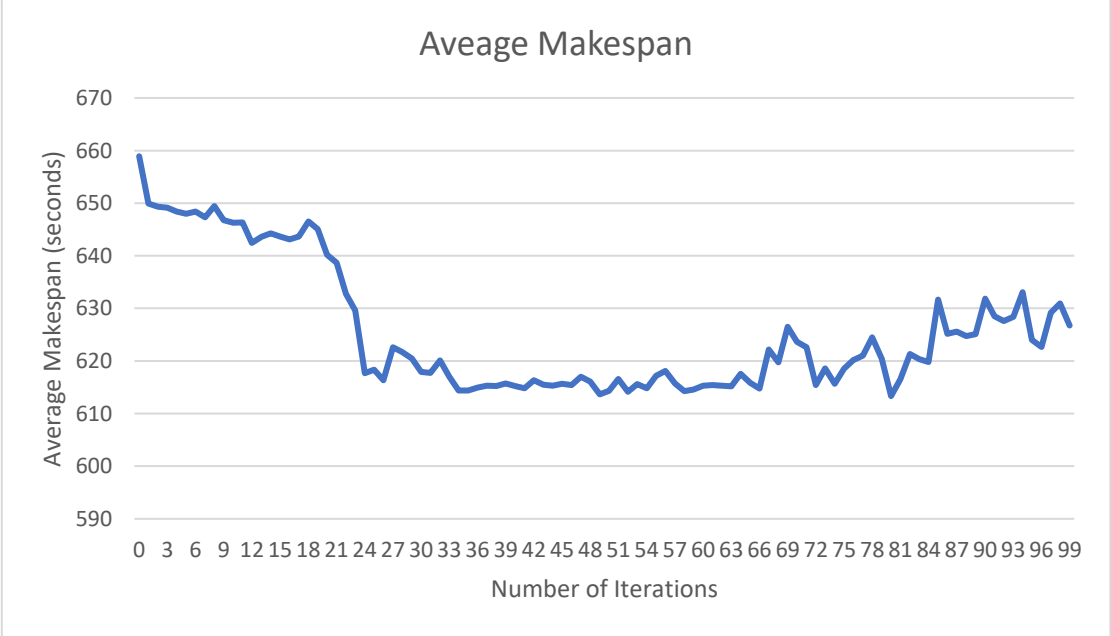

Fig. 2: Average Makespan vs Number of Iterations

Fig. 2 shows the variation in average makespan with number of iterations. Convergence of algorithm on this metric can be easily observed. Though minimum value of average makespan is achieved as 613 on iteration 80, resulting average makespan on iteration 100 stands to be 626. This deviation from minimum makespan can be seen as a result of trade-offs required to strike a balance among makespan, completion (Fig. 3) and slowdown (Fig. 4).

Figures 3 and 4 present the results of the job scheduling task in terms of average completion time and average slowdown, respectively.

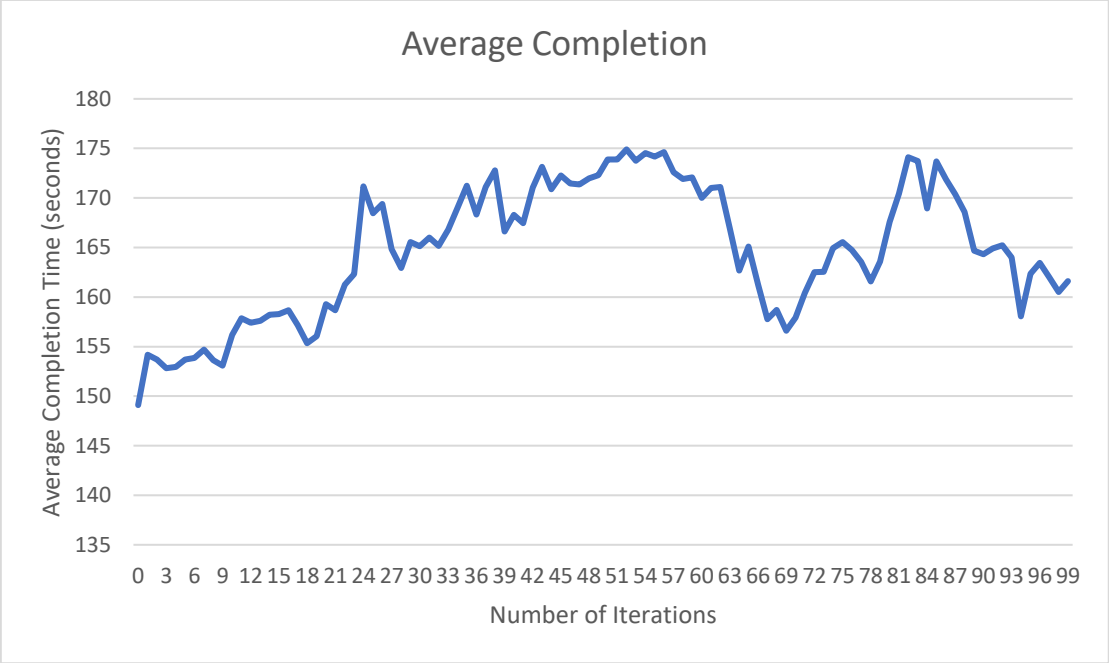

Fig. 3: Average Completion Time vs Number of Iterations

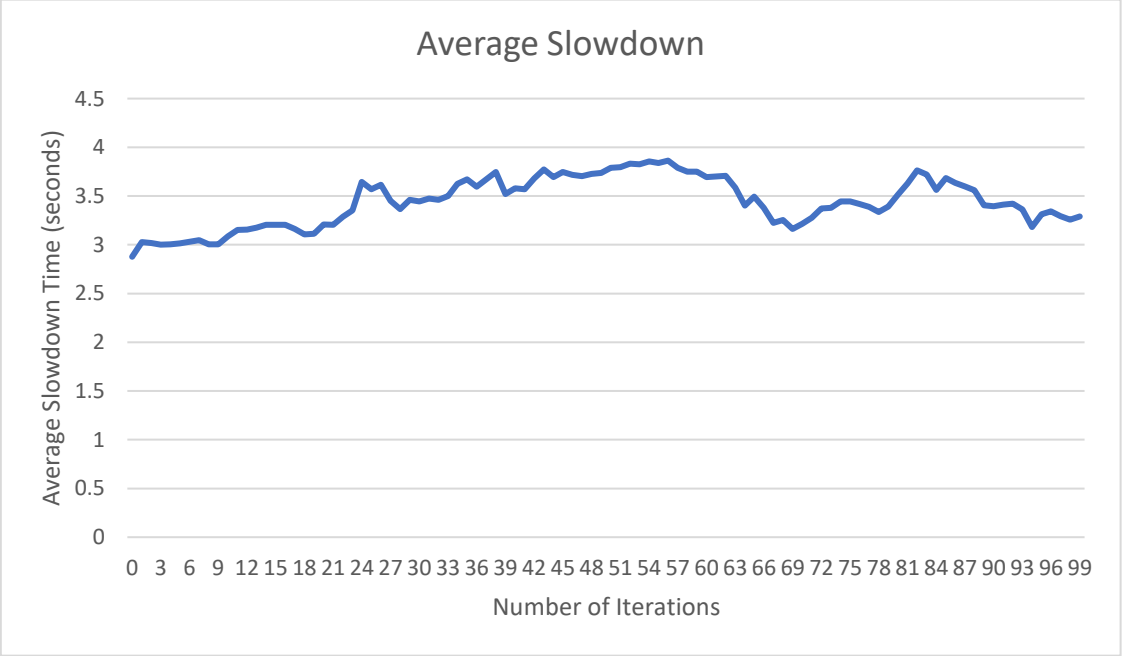

Fig. 4: Average Slowdown Time vs Number of Iterations

The figures show that the average completion time is restricted to 161 and the average slowdown is restricted to 3.2. The reason for completion time fluctuating more frequently across iterations compared to slowdown can be explained by the way in which the two metrics are calculated. Completion time is a direct measure of the time taken to complete a job, which can vary significantly depending on factors such as job size and available resources. As a result, the completion time metric can be more sensitive to changes in the scheduling policy and can fluctuate more frequently across iterations. On the other hand, slowdown is calculated as the ratio of job-time with wait to job-time without wait. This metric takes into account the waiting time experienced by jobs and provides a more comprehensive measure of the scheduling policy's performance. Since slowdown is calculated based on a ratio, it is less sensitive to changes in individual job sizes and resource availability, and as a result, may fluctuate less frequently across iterations compared to completion time.

The restriction on average completion time and average slowdown indicates that the scheduling policies generated by the DRL models are able to maintain a certain level of performance consistency. While completion time may fluctuate more frequently across iterations, the use of the slowdown metric provides a more stable and comprehensive measure of the scheduling policy's performance.

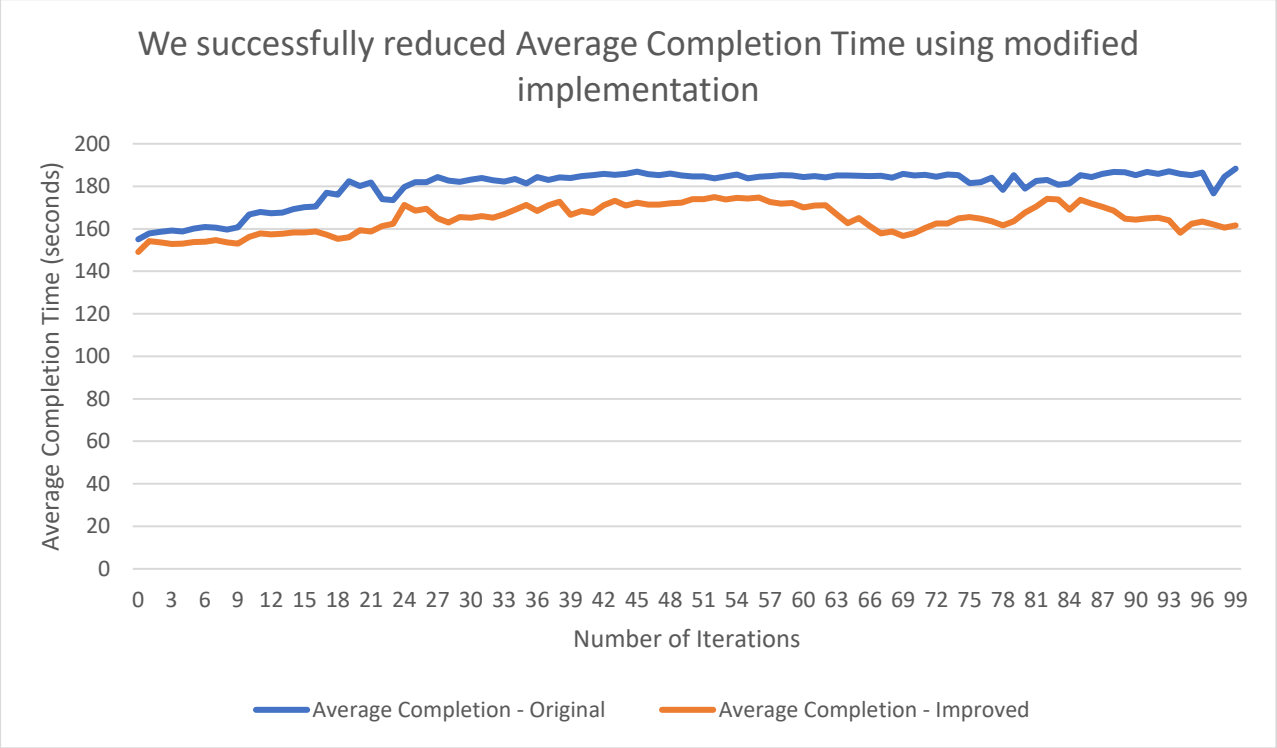

Fig. 5: Average Slowdown Time vs Number of Iterations

In Fig. 5, a comparison is presented between DRL-based models that use policy gradients and those that use Proximal Policy Optimization (PPO) with modified reward functions over completion time metrics. The results show that the models that use PPO outperform the policy

gradient-based algorithms in terms of completion time metrics. Moreover, the difference in performance between these two types of models increases across iterations, which signifies better convergence of the PPO-based models. The improved performance of the PPO-based models can be attributed to the use of the modified reward function, which enables the agent to learn better policies that take into account the completion time metric. PPO is also known to be more stable and efficient than policy gradient-based algorithms, which may explain its superior performance. Additionally, the better convergence of the PPO-based models indicates that they are able to learn faster and more efficiently than the policy gradient-based algorithms.

6. CONCLUSION

In conclusion, the proposed algorithm for deep reinforcement learning-based job scheduling is capable of learning from experience and optimizing a fitness function that directly minimizes completion time and slowdown while also keeping the makespan in check. The evaluation of the algorithm using simulation shows that it outperforms heuristic-based job scheduling algorithms and reduces the average completion time and slowdown. Furthermore, the use of a proximal policy optimization-based algorithm instead of policy gradients-based approaches has been implemented to ensure greater stability and faster convergence. This algorithm not only provides superior performance but also addresses the challenges of convergence that are typically associated with policy gradient-based algorithms. The results obtained in this work demonstrate that the proposed algorithm has the potential to improve job scheduling across various industrial spheres. It is expected that this algorithm can provide significant benefits in scenarios where job scheduling is critical to the overall performance of the system, such as in manufacturing, logistics, and data centers. The ability to optimize job scheduling policies in real-time can potentially lead to significant improvements in efficiency and productivity, ultimately resulting in cost savings and improved energy efficiency.

REFERENCES

1. G. Sun, T. Zhan, G. O. Boateng, D. Ayepah-Mensah, G. Liu, and W. Jiang, “Revised reinforcement learning based on anchor graph hashing for autonomous cell activation in cloud-rans,” Future Gener. Comput. Syst., vol. 104, pp. 60–73, 2020.
2. D. Ding, X. Fan, Y. Zhao, K. Kang, Q. Yin, and J. Zeng, “Q-learning based dynamic task scheduling for energy-efficient cloud computing,” Future Gener. Comput. Syst., vol. 108, pp. 361–371, 2020.
3. W. Guo, W. Tian, Y. Ye, L. Xu, and K. Wu, “Cloud resource scheduling with deep reinforcement learning and imitation learning,” IEEE Internet Things J., vol. 8, no. 5, pp. 3576–3586, 2021.
4. Bitsakos, I. Konstantinou, and N. Koziris, “DERP: A deep reinforcement learning cloud system for elastic resource provisioning,” in 2018 IEEE International Conference on Cloud

Computing Technology and Science, CloudCom 2018, Nicosia, Cyprus, December 10-13, 2018. IEEE Computer Society, 2018, pp. 21–29

5. N. Liu, Z. Li, J. Xu, Z. Xu, S. Lin, Q. Qiu, J. Tang, and Y. Wang, “A hierarchical framework of cloud resource allocation and power management using deep reinforcement learning,” in 37th IEEE International Conference on Distributed Computing Systems, ICDCS 2017, Atlanta, GA, USA, June 5-8, 2017. IEEE Computer Society, 2017, pp. 372–382.
6. Li, F. R. Yu, P. Si, W. Wu, and Y. Zhang, “Resource optimization for delaytolerant data in blockchain-enabled iot with edge computing: A deep reinforcement learning approach,” IEEE Internet Things J., vol. 7, no. 10, pp. 9399–9412, 2020.
7. *S. Yasuda et al., "An Adaptive Cloud Bursting Job Scheduler based on Deep Reinforcement Learning," 2021 International Conference on High Performance Big Data and Intelligent Systems (HPBD&IS), Macau, China, 2021, pp. 217-224, doi: 10.1109/HPBDIS53214.2021.9658447.*
8. *P. Song et al., "A Deep Reinforcement Learning-based Task Scheduling Algorithm for Energy Efficiency in Data Centers," 2021 International Conference on Computer Communications and Networks (ICCCN), Athens, Greece, 2021, pp. 1-9, doi: 10.1109/ICCCN52240.2021.9522309.*
9. *Y. Feng et al., "Flexible Job Shop Scheduling Based on Deep Reinforcement Learning," 2021 5th Asian Conference on Artificial Intelligence Technology (ACAIT), Haikou, China, 2021, pp. 660-666, doi: 10.1109/ACAIT53529.2021.9731322.*
10. *Song et al. "A reinforcement learning based job scheduling algorithm for heterogeneous computing environment." Computers and Electrical Engineering 107 (2023): 108653.*
11. *Engstrom, Logan, et al. "Implementation matters in deep policy gradients: A case study on ppo and trpo." arXiv preprint arXiv:2005.12729 (2020).*
12. *Li et al.. "Deepjs: Job scheduling based on deep reinforcement learning in cloud data center." Proceedings of the 4th International Conference on Big Data and Computing. 2019.*
13. *Sheng, Shuran, et al. "Deep reinforcement learning-based task scheduling in iot edge computing." Sensors 21.5 (2021): 1666.*
14. *Schulman, John, et al. "Proximal policy optimization algorithms." arXiv preprint arXiv:1707.06347 (2017).*
15. *Che, Haiying, et al. "A deep reinforcement learning approach to the optimization of data center task scheduling." Complexity 2020 (2020): 1-12.*
16. *Hsu, Chloe Ching-Yun, Celestine Mendler-Dünner, and Moritz Hardt. "Revisiting design choices in proximal policy optimization." arXiv preprint arXiv:2009.10897 (2020).*
17. *Rajkumar et al. "Performance and cost-efficient spark job scheduling based on deep reinforcement learning in cloud computing environments." IEEE Transactions on Parallel and Distributed Systems 33.7 (2021): 1695-1710.*