UNIVERSITÄT PADERBORN
Die Universität der Informationsgesellschaft
Faculty for Computer Science, Electrical Engineering and Mathematics
Department of Computer Science
Research Group Computer Networks

Employing Reinforcement Learning Algorithms to solve Cluster Scheduling Problem

AICON - Project Group

Authors:
Aluri JaganMohini - 6868807
Kunal Pratap Singh Sisodia - 6882887
Tejas Ravindra Dhawale - 6882910

Supervisors:
Prof. Dr. Holger Karl | Asif Hasnain, M.Sc.

Contents

List of Figures

1 Introduction

1.1 Problem Statement

Resource management is one of the ever-present problems in the networking and computer systems domain. One such example is cluster scheduling and assigning sufficient resources for the execution of the jobs. Developing a solution for such a problem requires cleverly designed algorithms optimized for good performance. In the real world though, there are several complications that make designing such an algorithm a challenging task. One such challenge is accurately modeling a complex underlying system in which there are multiple tasks associated with a job. These tasks interact with each other and might lead to interference on computing resources. Furthermore, taking scheduling decisions with noisy inputs is another challenge that needs to be dealt with.

This report is based on the work done in [6]. We mainly focus on cluster scheduling problem where we have incoming jobs which need multiple resources for their execution. The machine consists of two resources namely CPU and memory. The scheduler should pick the jobs from the input pending job queue and then assign sufficient resources for its execution. The phenomenon is shown in Figure 1.1. In our problem definition, the scheduler must also optimise

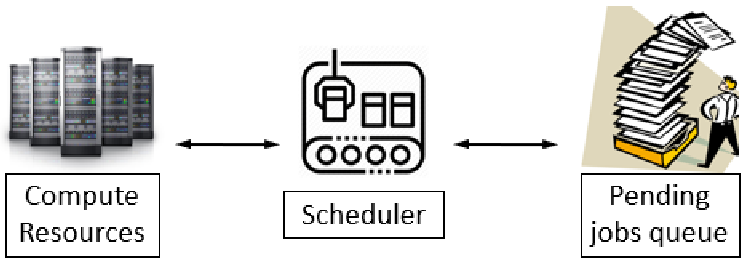


Figure 1.1: Job Scheduling.

some objective function like minimizing the average job slowdown and minimizing the average job completion time. Maintaining a balance between scheduling and optimizing objective function is challenging and makes the problem, in general, an NP-hard problem. There are various existing heuristics like Shortest-Job-First (SJF), Tetris, and Packer which address this problem of resource management. However, the majority of the existing heuristics often do not accord to optimal performance. Human-generated heuristics also require a lot of testing and tuning of the parameters so that they satisfy the working needs. The work done in [6] uses a machine learning paradigm called Reinforcement Learning so that systems can learn to manage the resources on their own and without any human intervention.

2 Goals

In this chapter, we primarily discuss the main objectives linked with the scheduling of the incoming jobs. Along with optimal resource management, the main challenge is to cater to different performance objective like minimizing the average job slowdown and minimizing the average job completion time.

2.1 Average Job Slowdown

The goal of the agent is to minimize the slowdown associated with the scheduling of the jobs. For job j in the input jobs, the slowdown is calculated using Equation 2.1, where C_j represents the time difference between the completion time and the arrival time of the job j, and T_j represents the ideal duration for the execution of the job j [6].

$$\mathbf{S}_j = \mathbf{C}_j | \mathbf{T}_j \tag{2.1}$$

2.2 Average Job Completion time

The goal of the agent is to minimize the total time taken for the execution of the jobs. For job j in the input jobs, the completion time is calculated using Equation 2.2, where C_j represents the time difference between the completion time and the arrival time of the job j .

$$\mathbf{CT}_j = \mathbf{C}_j \tag{2.2}$$

3 Approach Overview

In this chapter, let us discuss in depth the solution for solving the problem discussed in chapter 1. We use Reinforcement learning (RL) which helps the system to learn automatically from its experience. The working principle of the RL-based solution is shown in Figure 3.1. The environment is the representation of the problem that the agent must explore. At each timestep **t**, the agent observes the current state $\mathbf{s}_t$ of the environment and accordingly takes actions $\mathbf{a}_t$ in the environment. Depending on the action taken, the agent receives reward $\mathbf{r}_t$. The transition of the states and the received rewards are assumed to follow the Markov property, i.e., the state transition and the rewards totally depend on the current state $\mathbf{s}_t$ of the environment and the action $\mathbf{a}_t$ taken by the agent at timestep $\boldsymbol{t}$ [6]. Initially, the agents start with no knowledge about the environment but it gradually learns the tasks to do depending on the received reward. The primary objective of the agent is to maximize the cumulative reward received over time, which is mathematically represented as $\mathbb{E}\left[\sum_{t=0}^{\infty} \gamma^t r_t\right]$, where $\gamma \in (0, 1]$ is a factor discounting future rewards [6].

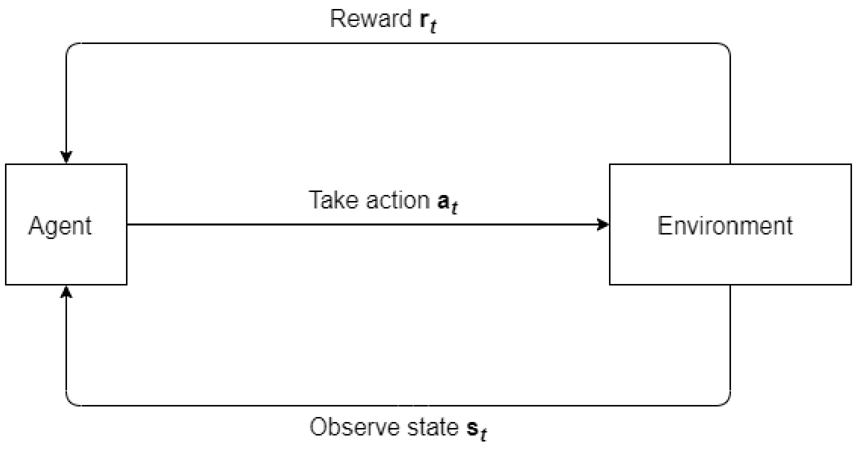


Figure 3.1: Reinforcement Learning [6].

3.1 Environment Details

In our approach to tackle the problem described in 1, we created two types of environments: **Discrete** environment and **Multi-Binary** environment. The environments only differ in the way the actions are taken. We consider the machine with two resources which are the CPU and the memory. Our implementation is based on the assumption that the resource requirements and the duration of each job are known on arrival. For each job $\boldsymbol{j}$, we define the resource profile by $\mathbf{T_j}$ and a vector $\mathbf{r_j}$. $\mathrm{T_j}$ determines the job's duration, while the vector $\mathbf{r_j = (r_{j,1}, r_{j,2})}$ determines the resource demands. Here $\mathrm{r_{j,1}}$ represents the number of CPU units, while $\mathrm{r_{j,2}}$ represents the number of memory units required by the job j. We specify the limit for the maximum duration of the job and the maximum resource units required for each job. There are $\mathbf{M}$ job slots and a backlog queue for storing the input jobs. The length of the job slots and the size of the backlog queue is also fixed. We describe both the environment using Markov Decision Process (MDP) in the following sections.

3.1.1 Discrete Environment

The Discrete environment makes use of discrete action space (§3.3.1) to allocate computing resources to the input jobs. With discrete action space, the agent selects one job from the $\mathbf{M}$ input available jobs. The current time is frozen and the agent is allowed to take multiple actions until the agent takes an invalid action or a void action. The resulting consequence is that the agent becomes capable of taking multiple actions in a single timestep.

3.1.2 Multi-Binary Environment

The Multi-Binary environment makes use of Multi-Binary action space (§3.3.2) for job scheduling. With Multi-Binary action space, the agent selects a subset from the $\mathbf{M}$ input jobs to schedule and schedules the entire subset of jobs in a single step function call if there are enough resources available. The timestep incrementation happens when the job allocation fails.

3.2 Observation Space

We implemented the environment with two observation spaces. Initially, we started with the observation space in which the job slots are stacked on each other. The representation is shown in Figure 3.2. The current status of the machine which contains CPU and memory resources is shown at the top. This is followed by the job slots in which jobs waiting to be scheduled are stored. For illustration in Figure 3.2, there are 5 job slots with a fixed size which indicates the maximum length of the job that can be contained in the job slot. Each job slot holds a job specifying the job's resource profile, for example, the job in job slot 1 requires one unit of CPU and two units of memory for three timesteps, while the job in job slot 2 requires three units of CPU and three units of memory for six timesteps.

Another representation of the observation space is shown in Figure 3.3. The system is represented like an image, similar to the one specified in [6]. Each job has a separate representation for each of its resources. The current CPU status of the machine is specified at the beginning followed by the CPU requirements of all the jobs in the job slots. In Figure 3.3, there are 5 job slots following the machine's current CPU resource status. The job in job slot 1 requires one unit of CPU for three timesteps, and the job in job slot 2 requires three units of CPU for six timesteps. After the CPU profile representation, the current memory status of the machine and the jobs in the job slots are depicted. The job in job slot 1 requires two units of memory

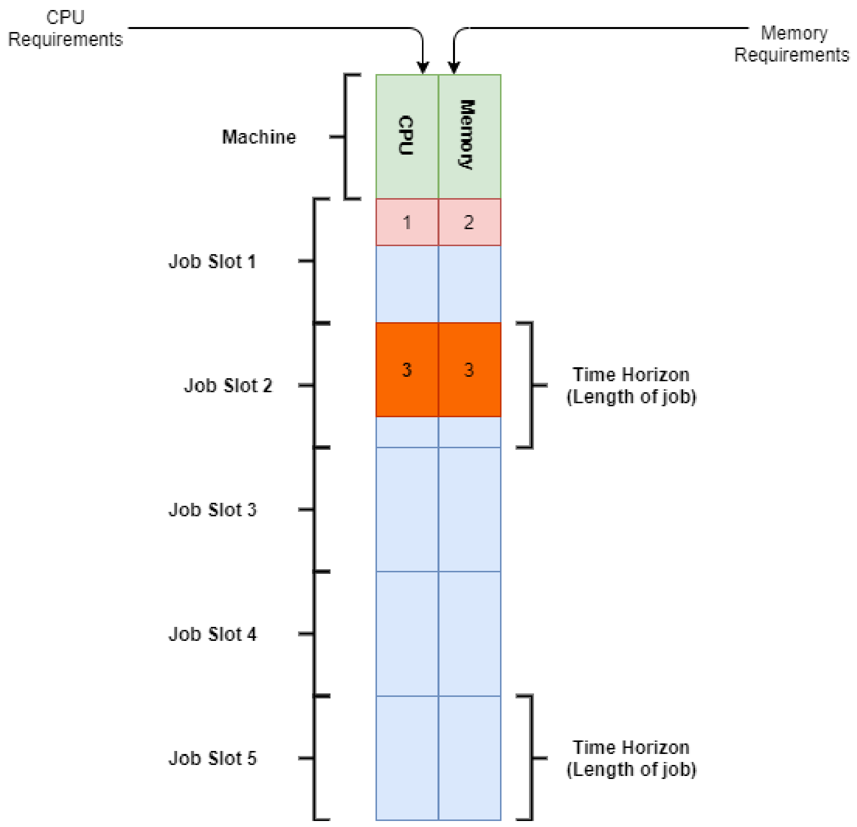


Figure 3.2: Observation Space with stacked job slots.

for three timesteps while the job in job slot 2 requires three units of memory for six timesteps. Similar to 3.2, there is a limit to the length of the job slot which is equal to the maximum length of the job. The details of the jobs past the first **M** jobs are summarized and then stored in the backlog. The backlog element contains the summarized information of the jobs that are present in the input queue but not in the job slots. For training purposes, it is sufficient if we initially limit the scope to the earlier-arriving jobs since plausible policies most likely favor the longer waiting jobs [6].

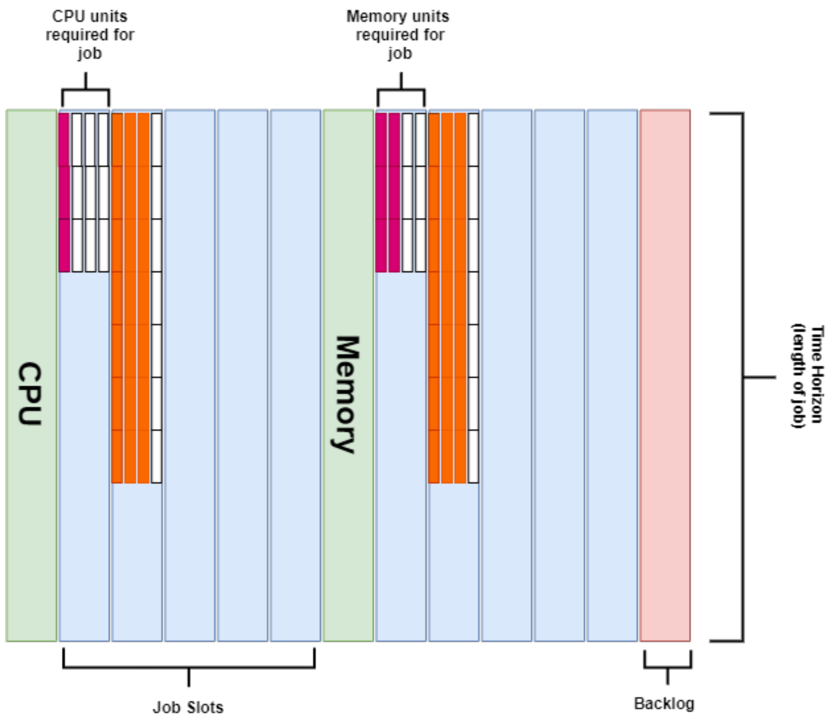

CPU units required for job
Memory units required for job
CPU
Memory
Time Horizon (length of job)
Job Slots
Backlog

Figure 3.3: Observation Space with image like representation.

3.3 Action Space

In this section, we will discuss the details of the different types of action spaces used by the agents.

3.3.1 Discrete

The Discrete environment makes use of the discrete action space. At any given point in time, the agent may wish to schedule a job from the **M** job slots. With the help of the technique specified in [6], at each timestep, the agent can take more than one action. For **M** job slots, the size of the action space is given by **M+1** and can be represented as $\{0,1,2,.....,M,\phi\}$. Here $action = i$ specifies that the agent must schedule the job from i^{th} job slot. Also, action $= \phi$ represents a void action which means that the agent does not intend to schedule any further job in the present timestep. To allow the agent to take multiple actions at the same timestep, the time is frozen until the agent chooses action $= \phi$, i.e., a void action or an invalid action. An invalid action is an action in which the agent is trying to schedule a job from the job slot, but there are no sufficient machine resources available for the execution of the job. The agent continues scheduling the job with a valid action and when the agent picks a void action or an invalid action, the time is incremented. This keeps the size of the action space comparatively small allowing the agent to schedule more than one jobs in a single timestep.

3.3.2 Multi-Binary

In the multi-binary action space, we use a vector of **N** bits instead of a single discrete value. So, for each index of the job slot, the agent will predict a binary digit. If the predicted bit is 0, then it indicates an invalid action or void action and if it is 1, then it indicates a valid action meaning that a particular job will be scheduled or allocated to the cluster, to get entertained by the cluster. As we iterate over all the job slots in a single method call, if there are multiple 1's in a vector, multiple jobs will be scheduled in a single timestep if there are enough resources available. It is still possible to allocate multiple jobs in the same timestep even in a discrete environment by freezing the current time. However, it requires multiple method calls or iterations and that is how multi-binary action space differs from the discrete action space specified in §3.3.1.

3.4 Rewards

In reinforcement learning, the agent is rewarded for every action it takes. The reward function is designed to help the agent to learn a good policy. Since the time is frozen until the agent does not take an invalid or void action, we do not reward the agent for the intermediate actions taken during a single timestep [6]. The reward function for the objective specified in chapter 2 is given in §3.4.1 and §3.4.2.

3.4.1 Average Job Slowdown

In this objective, the aim is to minimize the average job slowdown for each incoming job. We give the reward to an agent at every timestep t [6]. The formula to calculate average job slowdown is given by Equation 3.1.

$$\mathbf{Reward} = \sum_{\mathbf{i=1}}^{\mathbf{i=x}} \mathbf{-1\ /\ (job\ length)_i} + \sum_{\mathbf{j=1}}^{\mathbf{j=y}} \mathbf{-1\ /\ (job\ length)_j} + \sum_{\mathbf{k=1}}^{\mathbf{k=z}} \mathbf{-1/(job\ length)_k} \tag{3.1}$$

where,
x = number of jobs running on the machine,
y = number of jobs in the waiting queue,
z = number of jobs in the backlog.

3.4.2 Average Job Completion

In this objective, the aim is to minimize the average job completion time for each job. The agent is rewarded at every timestep using Equation 3.2.

$$\mathbf{Reward} = \mathbf{-1} * |\mathbf{X}| \tag{3.2}$$

Here, X is the sum of all the jobs present in the system which includes the machine, the job wait queue and the job backlog. It is given by Equation 3.3.

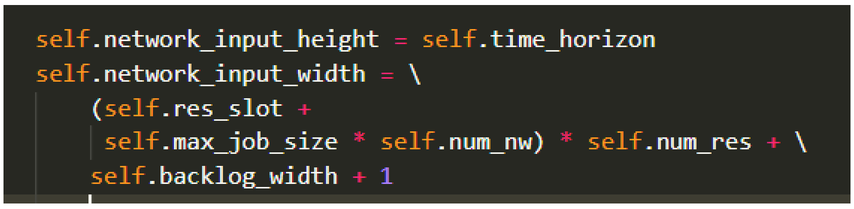
$$\mathbf{X} = \sum_{\mathbf{i=1}}^{\mathbf{i=x}} (\mathbf{jobs})_{\mathbf{i}} + \sum_{\mathbf{j=1}}^{\mathbf{j=y}} (\mathbf{jobs})_{\mathbf{j}} + \sum_{\mathbf{k=1}}^{\mathbf{k=z}} (\mathbf{jobs})_{\mathbf{k}} \tag{3.3}$$

where,
x = number of jobs running on the machine,
y = number of jobs in the waiting queue,
z = number of jobs in the backlog.

4 Implementation logic

4.1 Environment creation

The configurable parameters like the number of job slots, the length of the input job queue, and so on are specified in the file **parameters.py**. The details of some of the important configurable parameters are shown in table 4.1. Based on these parameters we calculate the input network height and input network width (Figure 4.1). This calculated network dimension determines the size of the observation space (§3.2).



```
self.network_input_height = self.time_horizon
self.network_input_width = \
    (self.res_slot +
     self.max_job_size * self.num_nw) * self.num_res + \
    self.backlog_width + 1
```

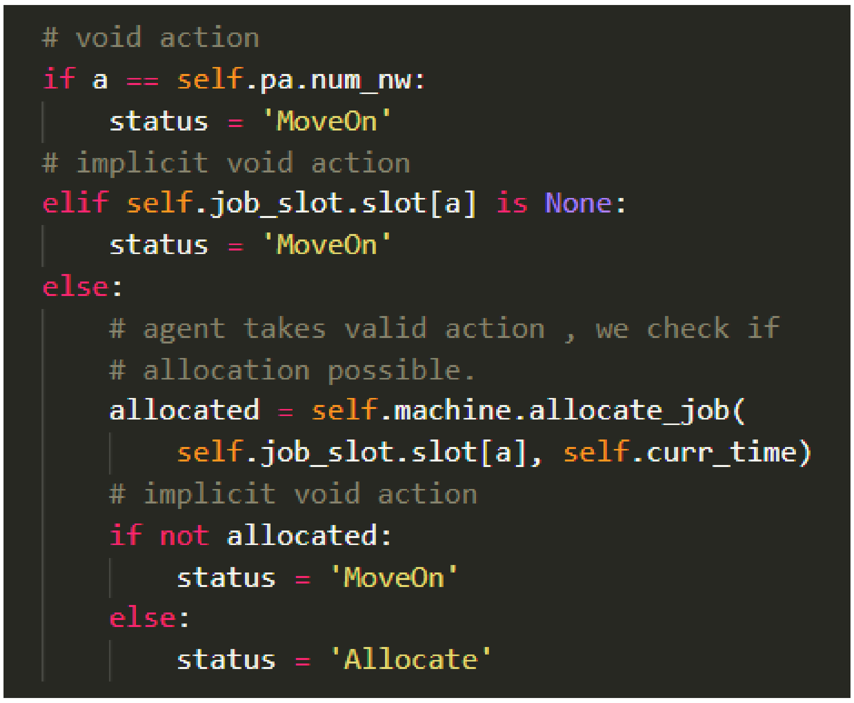
Figure 4.1: Network Dimension Calculation.

The machine consists of two resources, i.e., CPU and memory. The resource requirement of each job is known when the job arrives and then the jobs are assigned to the machine for its

Name	Description
simu_len	The length of the busy cycle that repeats itself
num_ex	The number of input job sequences
num_res	The number of resources in the system (e.g. CPU and memory)
num_nw	The number of job slots
time_horizon	The number of timesteps in the graph
res_slot	The maximum number of available reqource slots per time horizon
max_job_len	The maximum duration of the new jobs
max_job_size	The maximum resource request for a new job
backlog_size	The backlog queue size
new_job_rate	lambda in new job arrival Poisson Process

Table 4.1: Current list of parameters.

execution. When an agent takes an invalid action or a void action (ϕ) we set the current status as '**MoveOn**' which specifies that no job should be further scheduled in the current timestep. Also, a job is pulled from the input job queue and then stored in the first free available job slot. On the other hand, when an agent takes a valid action, we first check if the allocation of a job to the machine is possible. If there are enough resources available for the execution of the job, then we allocate the job to the machine and set the current status as '**Allocate**'. If even after taking a valid action, a job cannot be scheduled, then we set the current status as '**MoveOn**'. The code snippet for setting the status value is shown in Figure 4.2. Depending on the current status value we decide the further steps to be performed.



```
# void action
if a == self.pa.num_nw:
    status = 'MoveOn'
# implicit void action
elif self.job_slot.slot[a] is None:
    status = 'MoveOn'
else:
    # agent takes valid action , we check if
    # allocation possible.
    allocated = self.machine.allocate_job(
        self.job_slot.slot[a], self.curr_time)
    # implicit void action
    if not allocated:
        status = 'MoveOn'
    else:
        status = 'Allocate'
```

Figure 4.2: Steps to take based on the status.

When the current status is '**MoveOn**' we increment the current time. Along with it, a new job is drawn from the input queue and we also shift up the current observation space representation by one timestep. Depending on the availability of job slots, the new job is stored in the first available free job slot. If no free job slot is available, the new job will be stored in the backlog. When the current status is '**Allocate**', we assign necessary computing resources to the job and calculate the job slowdown and the time required for the completion of the job. Also, the job slot in which the newly allocated job was stored is freed up so that the job slot will be available for other jobs. If the backlog is not empty and contains jobs, then whenever the status will be '**Allocate**' and the job slot will be free, a new job is pulled from the backlog and assigned to the same job slot from which the job was just scheduled on to the machine. At the end of each step function, we return back the observation space (§3.2), the reward received (§3.4) by the agent by virtue of its action taken, and some extra information which gives us the job slowdown and completion time of each job, if the job was scheduled successfully. The episode execution is terminated when there are no jobs in the job slot, no jobs in the backlog and no jobs running on the machine. Finally, at the end of the termination, we reset the machine, the job slot, and the job backlog.

4.2 Training

For training of our agents, we make use of the Stable Baselines framework. Stable baselines framework is a set of improved implementations of Reinforcement Learning algorithms based in OpenAI baselines [3]. Details of the stable baselines policies which we use are given below.

- **A2C** (Actor 2 Critic) - A2C is a part of the on-policy family and it is a policy gradient algorithm. The key principle is to divide the model in two: one for computing an action based on a state and another one for producing the Q values of the action. The "Actor" is responsible to update the policy distribution from the feedback provided by the critic. Whereas the Critic is responsible for estimating the value function and both Critic and Actor functions are parameterized with neural networks [12].
- **DQN** (Deep Q Network) - It was developed by enhancing a reinforcement learning algorithm called Q-Learning with deep neural networks and combining a technique called experience replay to work for complex, high-dimensional environments, like Atari game, or in the field of robotics [7].
- **TRPO** (Trust Region Policy Optimization) - TRPO is an iterative approach that is similar to policy gradient methods which can be applied for optimizing policies like neural networks. It consists of a trust region that makes sure that the local approximations of the function are correct and it also ensures that the policy is not deviating from the initial point. This deviation is measured by KL-divergence. TRPO with hyperparameter tuning can provide considerable improvements in the results [9].
- **PPO2** (Proximal Policy Optimization) - It is the implementation by OpenAI which is dedicated to GPU, it uses vectorized environments for multiprocessing. It uses a combination of having multiple workers (A2C) and a trust region to improve the actor similar to TRPO. PPO2 algorithm uses clipping to avoid bigger updated which ensures that the new policy does not deviate from the old policy [10].
- **ACKTR** (Actor Critic using Kronecker-factored Trust Region) - ACKTR is used to learn control policies for Atari agents for discrete action spaces and for simulated robots for continuous action spaces. Pixels are given as inputs for both techniques. ACKTR is a combination of three unique techniques: First is trust region optimization for more consistent improvement. Second, a distributed Kronecker factorization to improve sample efficiency and scalability and lastly actor-critic methods [8].

We train our agent for 1000 training iterations with $\boldsymbol{N}$ episodes per iteration. We use an episodic setting with the policy determining the jobs to be scheduled. The agents are trained based on the specified objective, i.e., either for minimizing the average job slowdown or for minimizing the average job completion time.

The training of the stable baseline agents is by default done on the default hyperparameters like learning rate, discount factor and so on, which has been defined in the libraries. The agent trained using these default parameters is capable of giving a good performance but in some case when you have a custom environment, action space and observation space, the agent might not perform well. To help agent achieve the required performance there is a technique called hyperparameter optimization. This technique is implemented by the service called the raytune framework [5]. In the process of training, hyperparameter optimization was implemented using raytune for phase 1 as well as for phase 2. The raytune framework requires the input of some key hyperparameters with an upper and a lower bound. Raytune performs various trails which consists of the unique value set of hyperparameters. Raytune runs parallel threads in the system and trains agents with this unique set of hyperparameters. Furthermore, after the completion of all the defined trails the finest performing agent with the unique set of hyperparameters is reported by the raytune framework.

4.3 Work Load

The network consists of two resources, viz., CPU and memory. The capacity of each resource is set to a default value of 10 units via the parameter res_slot in **parameters.py**, and the jobs that get generated in the network are such that 80% of the total jobs are small and 20% of the total fraction are large. The lengths of the smaller jobs vary between 1 and 3 timesteps and the larger jobs are of lengths varying between 10 and 15 timesteps. Each job randomly chooses between CPU and memory as a resource requirement to utilize the resource slots. For that, it can either choose one of the resources (CPU, memory) as a dominant resource whose value varies between 0.25 * (res_slot) and 0.5 * (res_slot) and the other resource as a secondary resource whose size varies between 0.05 * (res_slot) and 0.1 * (res_slot).

In the first phase of the project, the cluster capacity of the resource cluster was calculated by setting the value of the parameter time_horizon to 20 and the maximum size of each resource (CPU and memory), i.e., res_slot parameter to 10 in the **parameters.py** file. The total cluster capacity is the product of the above 3 parameters which indicates the maximum units of jobs a resource cluster can handle at a time. The load of the generated jobs was calculated by summing up the products of job length, CPU, and memory requests of each job. For the load variation experiment, we calculated the average job slowdowns at different cluster loads. The job scheduling experiment was repeated for different job sets consisting of a varying number of jobs through simu_len and arrival rates via the new_job_rate parameters respectively in such a way that the sum total load of the generated jobs varied between 10% and 190% of the total cluster capacity. Also, for each cluster load, the average job slowdowns were recorded and plotted in the graph.

In order to come up with an optimized solution for the load variation experiment, we did some research about the cluster load and made appropriate improvements to the cluster load calculation as the previous logic was dependent on the length of the generated jobs. In our second phase, we used the formula as stated in [6], and developed an exact method for calculating the cluster load independent of the simu_len. The updated formula for calculating the load is given by Equation 4.1.

$$\mathbf{Cluster\,load} = \frac{\sum_{\mathbf{j=1}}^{\mathbf{n}} (\mathbf{j}[\mathbf{CPU\,demand}] + \mathbf{j}[\mathbf{memory\,demand}]) * \mathbf{j}[\mathbf{job\,length}]}{\mathbf{Resource\,slot} * \mathbf{n}} \tag{4.1}$$

where, j = Each job in a job sequence,
n = number of jobs in a job sequence,
j[CPU demand] = CPU requirement of a job(in units),
j[memory demand] = memory requirement of a job(in units),
j[job length] = Length of the job(in timesteps),

Resource slot = Maximum capacity of each resource cluster(in units).

By using Equation 4.1, we can calculate the load on each cluster resource, and by passing different job rates we can achieve different loads. So, in our load variation experiment, we vary different job rates such as 0.1, 0.2, 0.3, 0.4, 0.5..., 1 to get the different cluster loads from 10% to 190%.

5 Experimental Evaluation

In this chapter, we will discuss the evaluation results of the trained Discrete and Multi-Binary action space agents. The evaluation criteria will be focused on how the trained agents performed with respect to heuristic based approaches like Shortest-Job-First (SJF), Packer, and random agent. Also, we will investigate the main reason owing to which the trained agents show considerable performance gain.

5.1 Discrete

This section is divided into two sub sections providing analysis of the trained agents with discrete action space during the phase 1 and phase 2 of the project respectively.

5.1.1 Phase 1

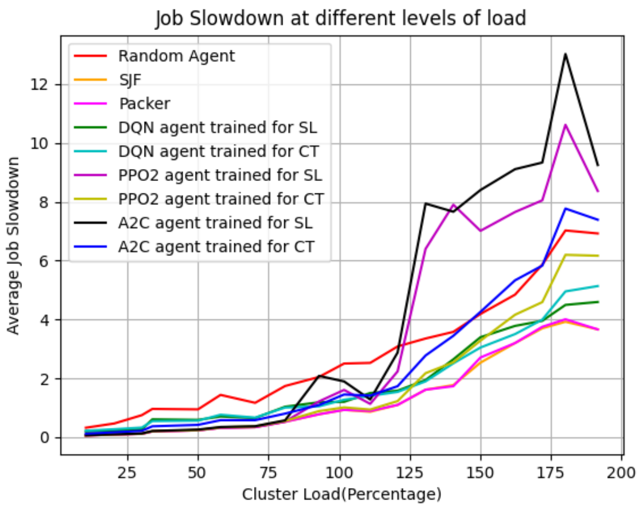


Figure 5.1: Average Job Slowdown at different loads (Discrete Environment Phase 1).

During the first phase of implementation, we predominantly trained three stable baselines agents namely DQN, PPO2, and A2C (§4.2) which support discrete actions. All the agents were trained for both the objectives, i.e., minimizing the average job slowdown and minimizing the average job completion time. The training of the agents was done using the stacked observation space (§3.2) for 1000 iterations. Figure 5.1 shows the performance of agents subjected to varying cluster loads. In the initial stage when the cluster load is low and there are fewer jobs to schedule, the trained agents and the heuristic based agents perform almost the same. Ideally, at a higher cluster load, the trained DeepRM agent's show better performance with respect to that of SJF. However, with the stacked representation of observation space, the agents were not able to learn a lot. In Figure 5.1, it is visible at higher cluster load higher than 100%, SJF was still performing significantly better than that of its closest competing rival DQN agent. The

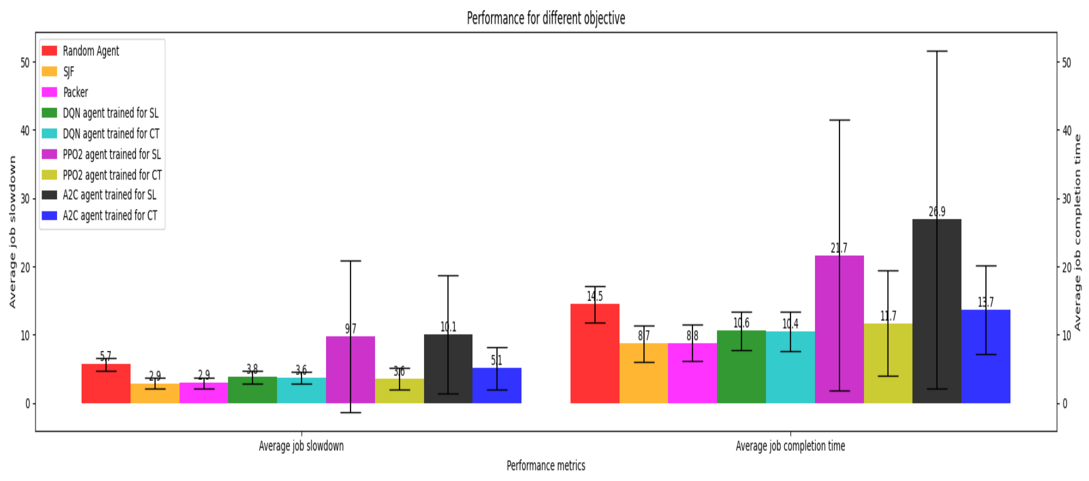


Figure 5.2: Performance at cluster load 130% (Discrete Environment Phase 1).

performance comparison graph for all the agents trained on the discrete environment at cluster load 130% is plotted in Figure 5.2. On the left side, we plot the results when the objective was to minimize the average job slowdown. Amongst all the trained agents, the DQN agent trained for objective job slowdown and objective job completion time performs better than A2C and PPO2. However, amongst all the agents, we can see that SJF which favors the jobs of a small duration performs better in comparison to that of the trained agents. SJF could achieve a slowdown of 2.9, whereas the corresponding counterpart DQN could achieve a maximum slowdown of 3.8. On the other hand, also in the context of minimizing average job completion time, the trained agents did not function well. The best performing trained agent DQN with a completion time of around 10, still underperformed considerably in comparison to SJF and Packer with a completion time of around 8.

5.1.2 Phase 2

Since the performance of the trained agents (§5.1.1) was not on par with the heuristic based approaches, in the second phase of implementation the entire observation space was modified. The enhanced observation space resembled images like representation comprising of machine status, detailed job requirements, and job backlog (§3.2). The workload calculation was also changed as mentioned in §4.3. In addition to stable baselines agents DQN, A2C, and PPO2 we also trained TRPO agent, which provides support for a discrete action space environment. The training of the agents was done for 1000 iterations as specified in §4.2. Figure 5.3 shows the performance of the retrained agents at cluster load 130% and with modified observation space. For the objective of minimizing the average job slowdown, A2C and TRPO agents trained for

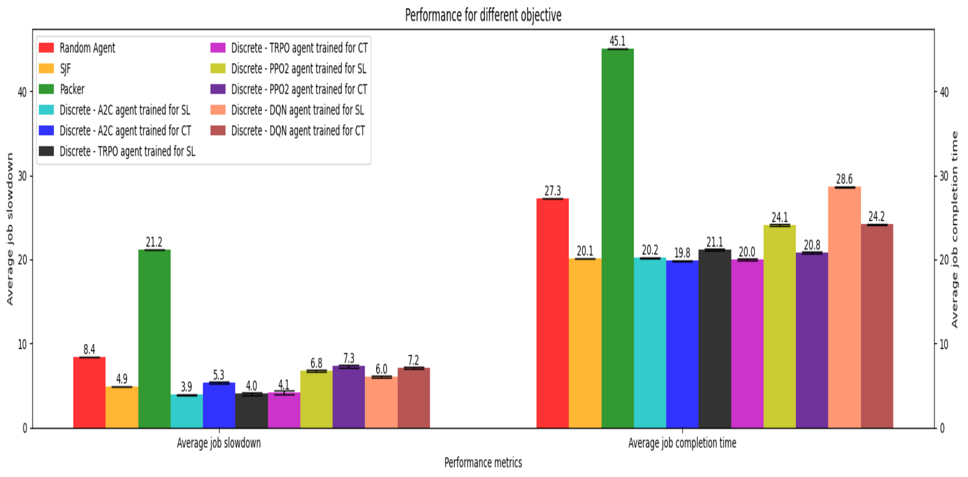


Figure 5.3: Performance at cluster load 130% (Discrete Environment Phase 2).

minimizing the job slowdown outperformed other trained agents like DQN and PPO2. Also, with respect to heuristic based agents like SJF and Packer, the trained A2C and TRPO agents showed superior performance. A2C and TRPO agents were able to achieve a slowdown of 3.9 and 4.0 respectively, which was considerably better than that of SJF with a slowdown of 4.9. For the objective of minimizing the average job completion time, even though A2C and TRPO trained for minimizing the job completion time performed better than all the other counterparts, the difference between the trained (A2C and TRPO) agents and its closest heuristic based competitor SJF was very marginal. The completion time for the A2C agent was 19.7, while that of TRPO and SJF was 20.1 and 20.0 respectively. To further boost the performance gain, we also used a custom policy in which instead of using default stable baselines agent parameters, we specified the number of neural network layers and the number of neurons in the neural network. The hyperparameters of the neural network were also tuned using raytune. The results however were almost the same.

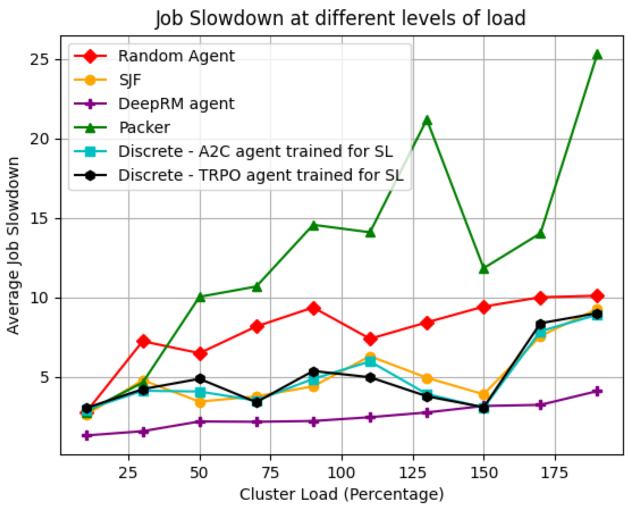


Figure 5.4: Average Job Slowdown at different loads (Discrete Environment Phase 2).

Subject to varying load, the trained agent performed better at higher load. Figure 5.4 shows the analysis of the Random agent, SJF, Packer, and the best performing A2C and TRPO agents (Figure 5.3). At low cluster load, there is not much difference between the trained A2C agent, the trained TRPO agent, and SJF. However, the performance gain of A2C and TRPO is clearly visible at cluster load higher than 110%. Although the agents were trained at cluster load 130%, they showed better performance at different loads. Training agents at specific load may in the future also yield significant performance improvement over SJF. When compared with the results in [6], i.e., the DeepRM agent, the A2C, and TRPO agents trained at cluster load 130% almost performed similarly to that of the DeepRM agent. A considerable difference between the DeepRM and the trained A2C and TRPO agents is clearly visible at other loads because the agents (A2C and TRPO) were trained only at 130% cluster load, while the DeepRM agent was trained at all the cluster loads. Training A2C and TRPO at all the cluster loads might give comparable performance to that of DeepRM at varying cluster loads. For simplicity we analyse the best performing A2C agent. Figures 5.5 and 5.6 show the slowdown curve and the reward curve for A2C agent. In Figure 5.6, the A2C agent converges after almost 100 iterations but we can observe a lot of variance in the training. Also, in Figure 5.5, it can be observed that the A2C agent's slowdown curve values were less than that of SJF.

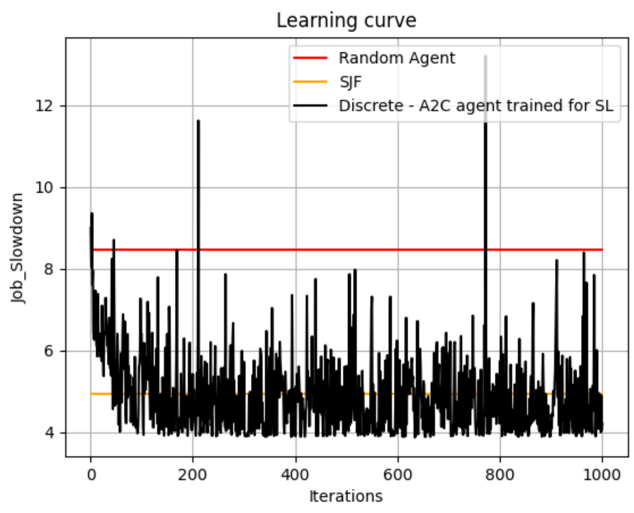


Figure 5.5: Learning Curve for Slowdown for A2C agent (Discrete Environment Phase 2).

The performance gain of the A2C agent in Figure 5.3 is by virtue of the fact that the trained agent learns to hold the larger jobs even if sufficient resources are available for scheduling the jobs. To analyze the A2C agent's supremacy over SJF we look at the distribution of input jobs and the way the jobs are scheduled by the agent. Figure 5.7 represents the fraction of job lengths used for inference. Here, a larger number of small length jobs are generated in comparison to bigger length jobs. In Figure 5.8, it is evident that the trained A2C agent holds more jobs of length greater than 10 benefiting the future arriving smaller jobs by reserving certain space for them. The consequence of laying aside free cluster resources is reduced overall job slowdown.

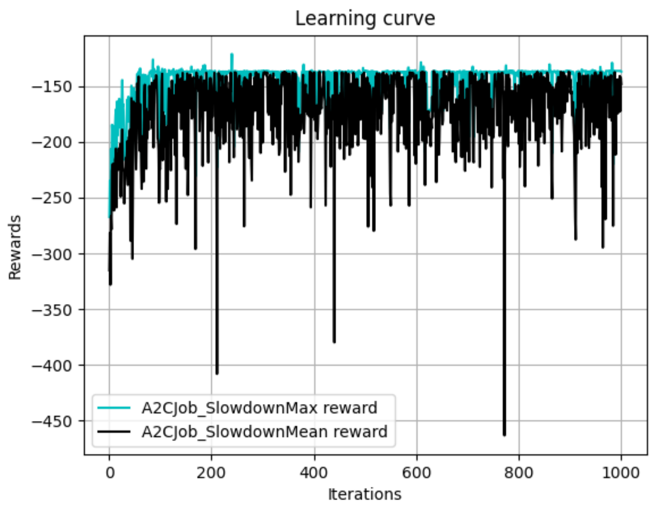


Figure 5.6: Reward Curve for Slowdown for A2C agent (Discrete Environment Phase 2).

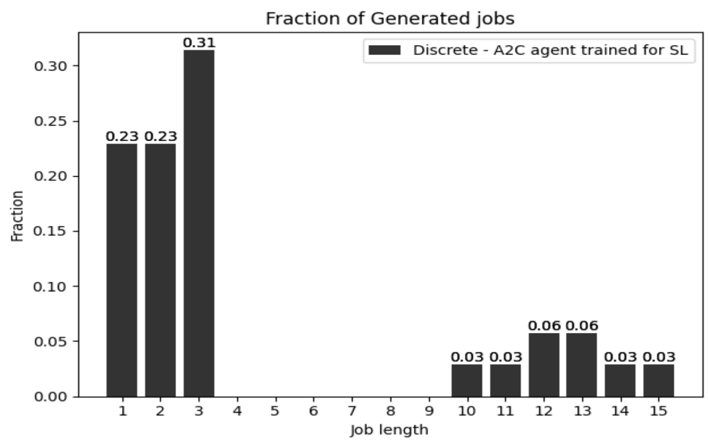


Figure 5.7: Fraction of jobs generated for A2C agent (Discrete Environment Phase 2).

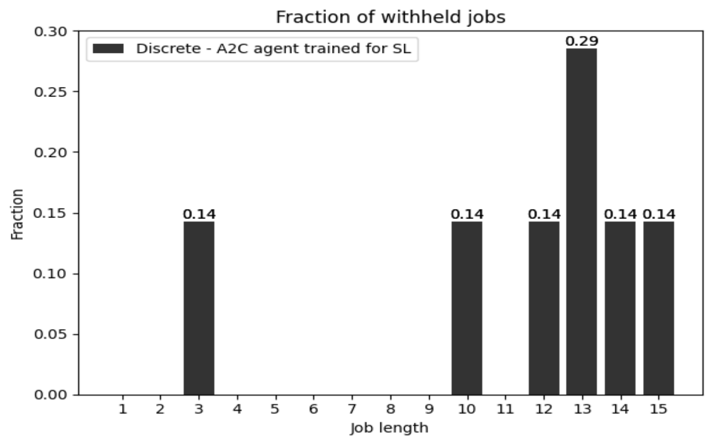


Figure 5.8: Length of withheld jobs for A2C agent (Discrete Environment Phase 2).

5.2 Multi-Binary

This section is divided into two sub sections to show the experimental results achieved with the multi-binary action space during the phase 1 and phase 2 of the project respectively.

5.2.1 Phase 1

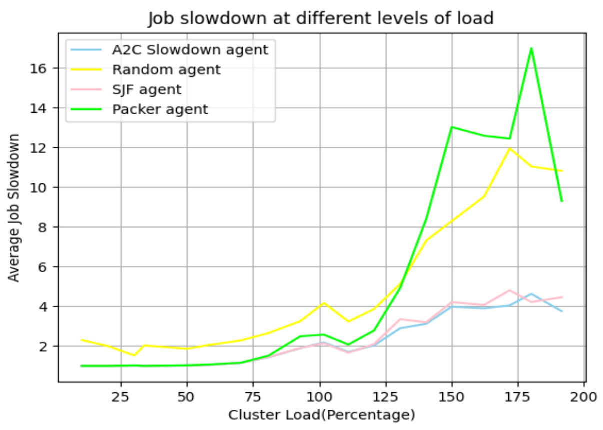


Figure 5.9: Average Job slowdown at different loads (Multi-Binary Environment Phase 1).

We conducted an experiment to test the behavior of all agents at different cluster loads and we were able to draw the results as shown in Figure 5.9. Figure 5.9 represents the average job slowdown achieved by the trained agent in comparison with the heuristic based approaches such as SJF (Shortest Job First) which schedules the job with the shortest length first, Packer which schedules the job with higher resource requirements, and the random agent which schedules the jobs randomly under different loads. From Figure 5.9, initially, when the load was very low almost all agents performed equally well. But with increased cluster load, trained A2C agent remained more promising compared with the heuristic based approaches as it performed better even at higher loads.

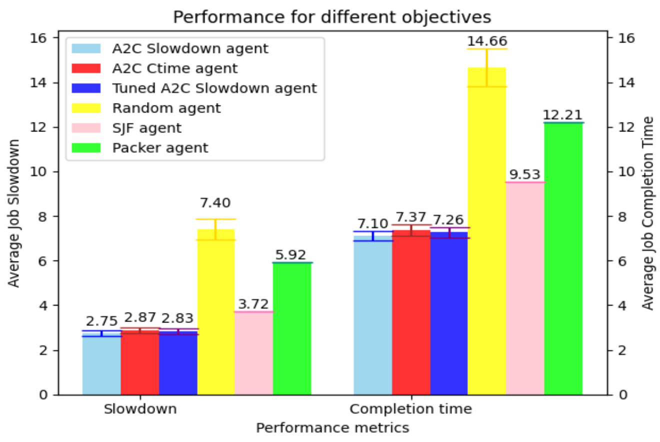


Figure 5.10: Performance at cluster load 130% (Multi-Binary Environment Phase 1).

For evaluating the performance of trained and tuned agents, we used two performance metrics, and they are minimizing the average job slowdown and minimizing the average job completion time. At first, we trained A2C for two different metrics, slowdown and completion time, and saved the models for each objective and later we compared the performance of each agent. Figure 5.10 shows the corresponding behavior of the trained A2C agent and tuned A2C agent which is tuned with varying hyperparameters at a higher cluster load of 130% against the heuristic based approaches. From this, we can observe that the A2C agent trained and tuned for slowdown performed equally well by outperforming the other agents in the experiment. A2C agent trained for completion time still performed better than the Packer and SJF.

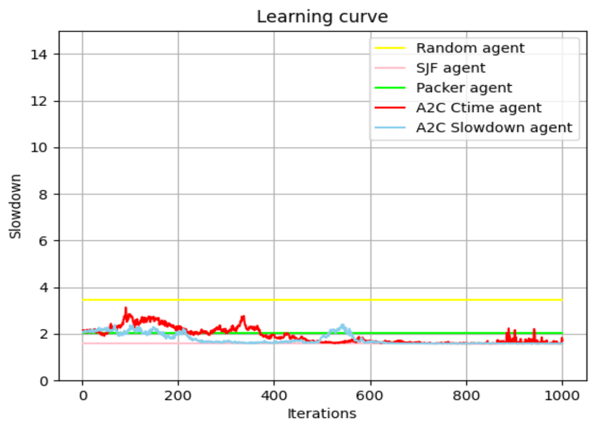


Figure 5.11: Learning curve for Slowdown for A2C agent (Multi-Binary Environment Phase 1).

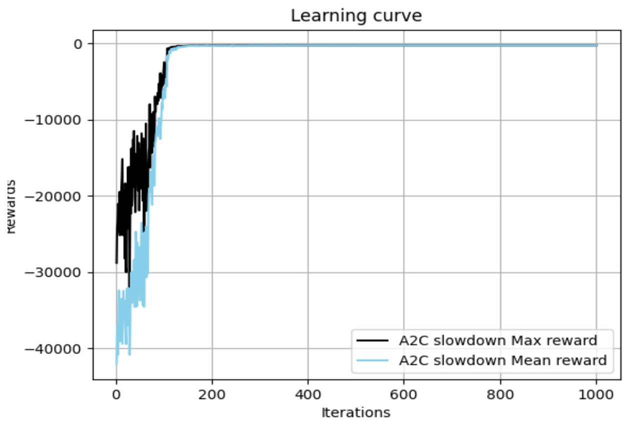


Figure 5.12: Reward Curve for Slowdown for A2C agent (Multi-Binary Environment Phase 1).

To study the learning curve, we trained the A2C agent for 1000 iterations at a cluster load of 70%, and we compared the results with SJF, Packer, and Random agent. If we observe in Figure 5.11 the average job slowdown achieved by the A2C agent over the iterations through the policy learned in comparison to that of the other agents, the trained A2C agent did not perform so well in the earlier iterations. With increasing iterations, the agent started learning and took better actions to improve the overall performance. So after 200 iterations, it even performed better than Packer and SJF. The same thing holds good even for Figure 5.12 which represents the maximum and mean reward achieved by the A2C agent over 1000 iterations in which the overall reward improved with increasing iterations.

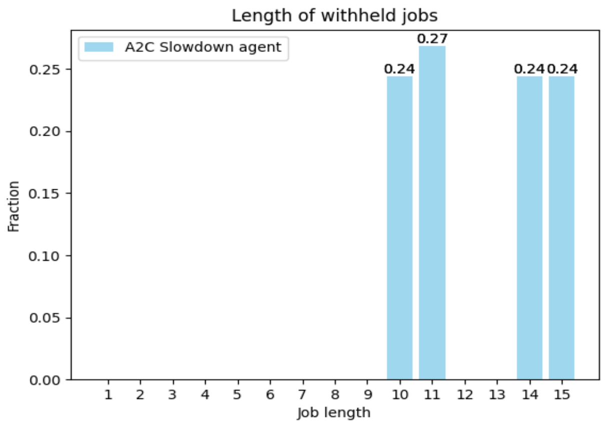


Figure 5.13: Length of withheld jobs for A2C agent (Multi-Binary Environment Phase 1).

To understand why the A2C agent performs better than SJF and Packer, we conducted another experiment to check if its behavior is the same for all jobs meaning for both small jobs and big jobs at a cluster load of 110%. Figure 5.13 shows that most of the jobs which get withheld are actually larger jobs. During our experiment, we also tested for Packer, but in the case of Packer, it may not be so and it may even withhold more smaller length jobs. The reason the A2C agent does so is because it is not work conserving. The agent here will learn to hold the larger jobs even if there are enough resources available so that it can favor the smaller jobs when they arrive.

5.2.2 Phase 2

We repeated the same experiments which we did in the first phase but with extra additions this time. The observation space was completely modified as discussed in §3.2 and the cluster load calculation was reformulated as explained in §4.3. When the cluster load variation experiment was repeated for different values of cluster loads by varying the new_job_rate of generated jobs, more smoother curves were obtained during phase 2 because of code enhancements and reformulation. The slowdowns of all the agents increased directly with the increasing loads as shown in Figure 5.14. The agents trained through stable baselines like A2C and PPO2 performed better when compared with the standard heuristics like SJF and Packer especially at the higher values of cluster loads and yielded very promising results. For inferring the correctness of the Multi-Binary environment results of A2C and PPO2 trained using reinforcement learning, we compared the results obtained with those in [6], and after plotting the results in a graph as shown in Figure 5.14, we can observe that there is a very minor difference between the results in [6] and our Multi-Binary environment especially at the higher values of cluster loads. The reason for these minor mismatches may be due to the differences in the action space and the model. The python libraries used for training would also affect the end results. Even though the results of the multi-binary action space are slightly different from the original results, we can observe that the curves are still in agreement with the results in [6].

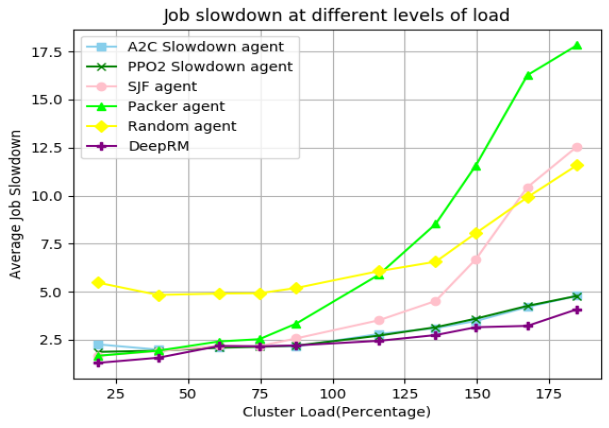


Figure 5.14: Average Job slowdown at different loads (Multi-Binary Environment Phase 2).

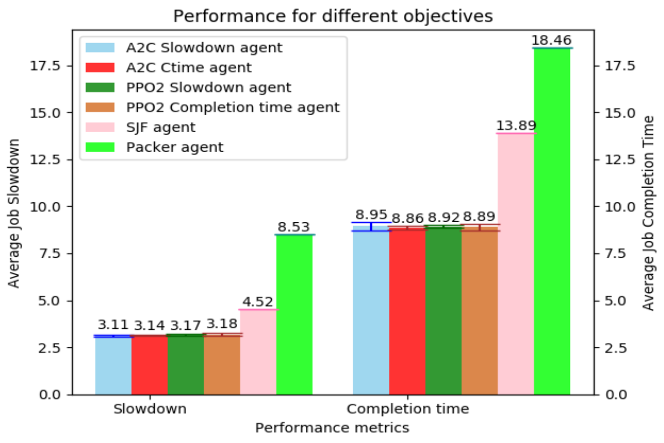
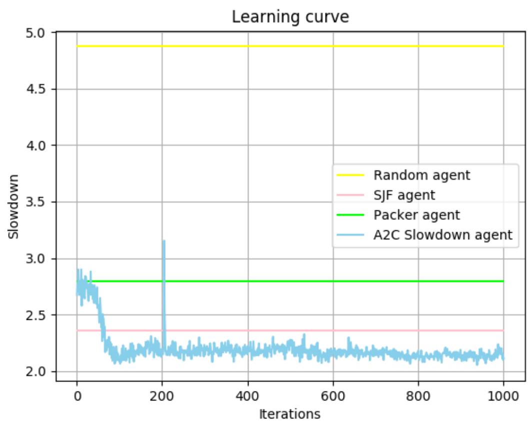


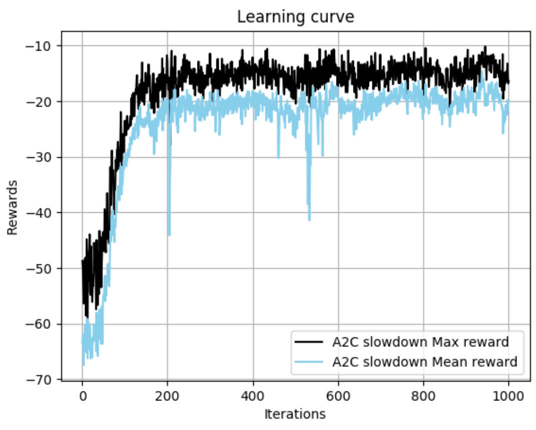
Figure 5.15: Performance at cluster load 130% (Multi-Binary Environment Phase 2).

The experiment in [6] was repeated for different agents for a higher value of cluster load of 130%, and the collected results were plotted in a box plot as shown in Figure 5.15. We can observe from Figure 5.15 that the trained agents performed really well by yielding lower values of slowdowns and completion times when compared with other scheduling techniques such as SJF, Packer, and even Random agent. The A2C and PPO2 agents trained for slowdown yielded slightly better slowdown results when compared with the same agents (A2C, PPO2) that were trained for completion time objective and vice versa.

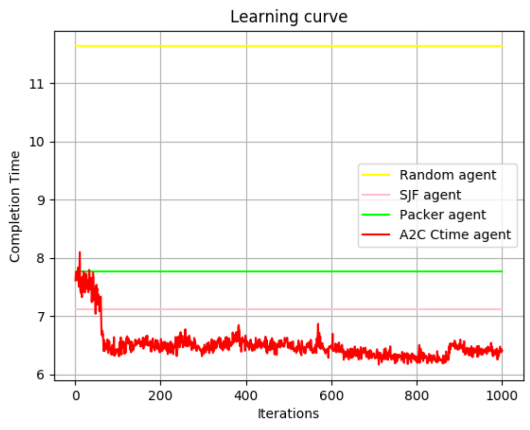
Each agent (A2C and PPO2) was trained for 2 different objectives, the first objective was to optimize the slowdown value through training and the reward or penalty was calculated in such a way that it improved its slowdown with every timestep it got trained. Similarly, the agents were again trained objective completion time, and the reward calculation was slightly different here for improving the completion time with respect to training. Figure 5.16a and 5.16c shows the training curves for the agents trained at the load of 70% for slowdown and completion times respectively, and by looking at the curves we can make an observation that the performance increases with increased training time and achieves a good convergence after being trained for certain number of timesteps. When the objective is to minimize the average



(a) average slowdown.

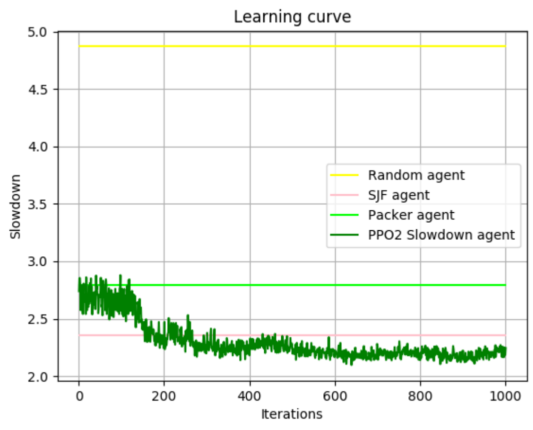


(b) total reward.

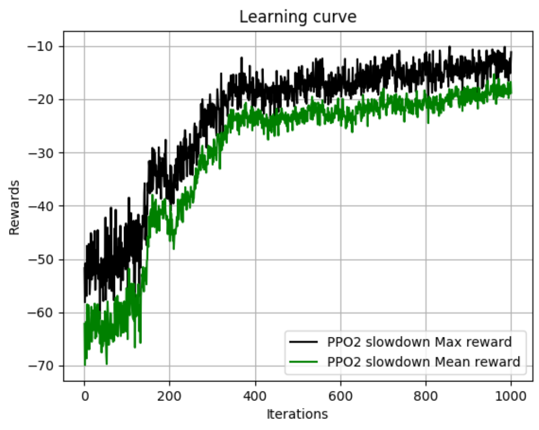


(c) average completion time.

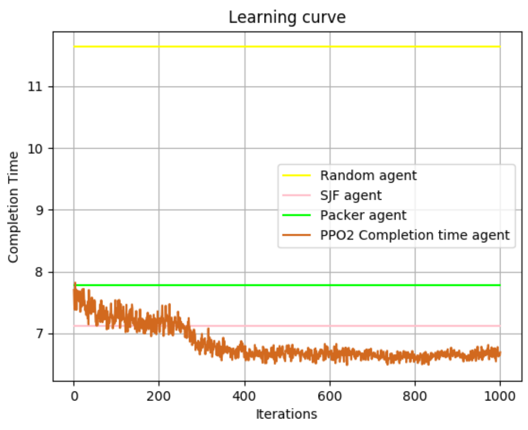
Figure 5.16: Learning curve showing the slowdown, reward and completion time achieved over iterations for A2C agent (Multi-Binary Environment Phase 2).



(a) average slowdown.



(b) total reward.

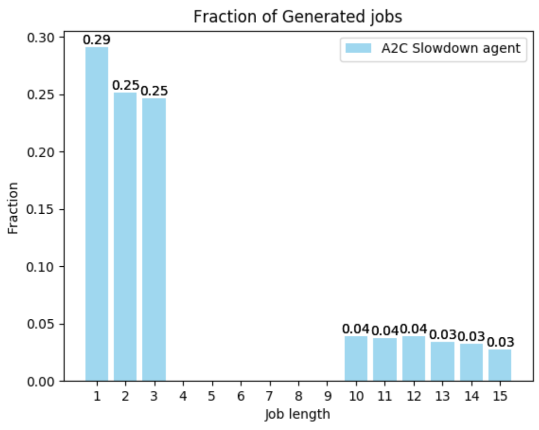


(c) average completion time.

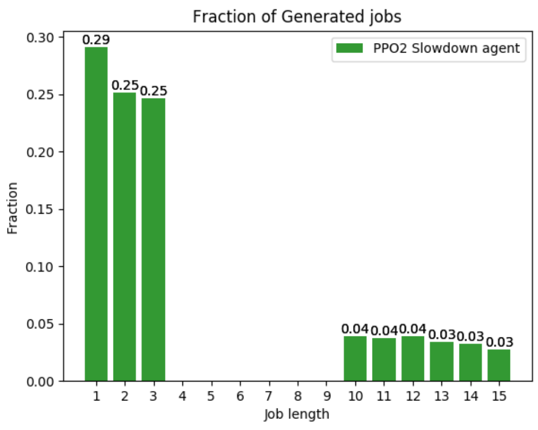
Figure 5.17: Learning curve showing the slowdown, reward and completion time achieved over iterations for PPO2 agent (Multi-Binary Environment Phase 2).

job slowdown, we can observe that the slowdown value of the agent decreases with increasing reward and that can be observed by the results of Figure 5.16a and 5.16b respectively. Similarly, when the objective is to minimize the average job completion time, the completion time value decreases with increasing training time as shown in Figure 5.16c. Also by looking at the Figures 5.17a, 5.17b, and 5.17c which shows the training curves for PPO2 agent at the load of 70% we can draw a similar inference.

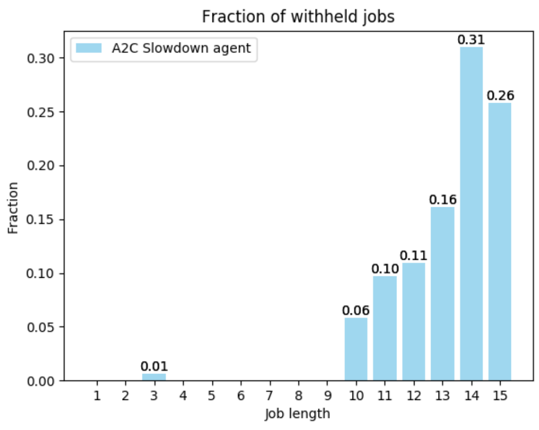
Another experiment was conducted by adjusting the new job rate in such a way that the cluster load was 110%. The lengths of the generated jobs were such that 80% of the generated jobs were smaller jobs meaning their lengths varied from 1 to 3 timesteps and only a small fraction, i.e., 20% of the generated jobs were larger jobs meaning their lengths ranged from 10 to 15 timesteps. Figure 5.18a and 5.18b show the fraction of the generated jobs in the order of their job lengths in timesteps for both A2C and PPO2 agents. Figure 5.18c and 5.18d shows the fraction of the jobs that are being withheld. When the generated jobs were scheduled using RL agents, a large fraction of the jobs whose lengths were 10 or above were being withheld and only a negligible fraction of smaller jobs were being withheld even though the number of smaller



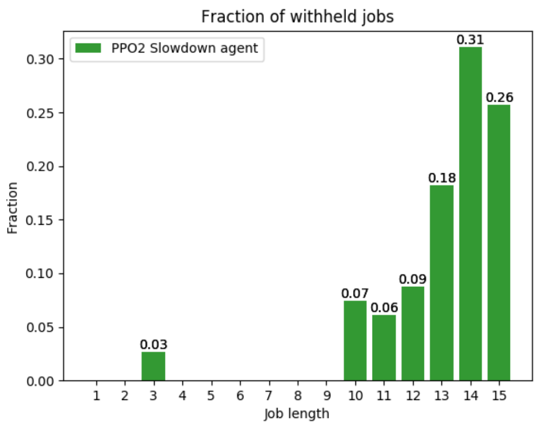
(a) Fraction of jobs generated for A2C.



(b) Fraction of jobs generated for PPO2.



(c) Withheld jobs for A2C.



(d) Withheld jobs for PPO2.

Figure 5.18: Fraction of Generated jobs and length of withheld jobs at 110% for both A2C and PPO2 (Multi-Binary Environment Phase 2).

jobs were 4 times higher as compared to the larger jobs. This was because RL agents like A2C and PPO2 learned to keep some CPU and memory resources reserved so that they can favor the smaller jobs by assigning them immediately once they arrive by holding the larger jobs. This eventually resulted in improving the overall mean slowdown of the job sets.

6 Challenges

To begin with, we started by creating an environment that takes discrete actions as specified in [6]. The observation space used and the way in which the workload was calculated was also slightly primitive at the beginning. Since the trained agents performed poorly at the beginning, we thought of using another approach for scheduling a subset of jobs i.e., using multi-binary action space for training the agents. We observed that in the case of multi-binary actions, the variance in the training was less, and also the performance of the trained agents was better in comparison to that of the discrete action space environment. However, the results in the first phase were still not matching to that in [6].

Immediately after finishing the first phase of the project, we were required to have an efficient formula for computing the cluster load. For that, initially, we generated the inter arrival times between each job by using an exponential distribution function [1], and along with job lengths and job sizes, the generated inter arrival times were used in the environment for the job scheduling experiment. For each job in the environment, we incremented the current timer with the inter arrival time irrespective of the '**MoveOn**' action. This consequently resulted in the scheduler waiting for multiple timesteps before scheduling or getting a new job even when there was enough resource available. So after that, we had to manually find a suitable number of jobs for which the cluster load was 190%, and by trial and error, we got the desired result with the value of the parameters simu_len = 200 and new_job_rate = 1. Now we were left to try out different new_job_rate values manually in order to vary the load from 10% to 190%. After implementing this we found out that, inter arrival time technique is not suitable for the current problem as in our environment a new job enters the waiting queues or backlog only upon the '**MoveOn**', and if we are incrementing the timer with inter arrival time just to favor the new jobs, the completion time for the existing jobs which are waiting in job slot and backlog will also increase. Because of these environmental issues and our time constraints we had to stop researching and further experimenting on this technique. However, this challenge can be considered for future work as this technique is an alternative to the current approach and with our current knowledge, we believe decent results can still be achieved through this technique if appropriate changes are made in the current environment.

Owing to the fact that the performance of the agents was not good, another challenge was to modify the observation space. Instead of stacked observation space (§3.2), we changed our observation space to image like representation given in [6]. Since the new observation space was a bit larger in size, the training time for the agents also increased.

7 Conclusion

Machine learning techniques have corroborated to be highly efficient for providing a solution to a variety of traditional complex problems and in the current continuance, these strategies are being incorporated in various domains for solving network problems. The Reinforcement Learning technique was deployed in this research paper for resolving multi-resource job scheduling problems at different cluster loads by training the stable baseline agents and saving the best performing models. The performance of the trained models was estimated by using two metrics, namely the average job completion time and the average job slowdown.

From the experimental results of Discrete and Multi-Binary environments (chapter 5), we can make an observation that the variations or the value of standard deviation was a little higher in Phase 1 results and its value was significantly lowered in Phase 2 by training the agent with multiple job sets. When we make a comparison between the Discrete and Multi-Binary environments, we can observe that the curves obtained for the Multi-Binary environment are in good agreement and they neither face the issues of overfitting or underfitting. The ability of the Multi-Binary environment to predict a subset of actions instead of a single discrete action in each iteration increases the chances of successful job allocation and thus guarantees accurate outcomes in every run. When the results of an agent trained through RL technique was compared with the traditional job allocation techniques like Shortest Job First, Packer, etc., the performance of the RL agent was in comparison with the SJF and the Packer at lower values of workload and as the workload increased in the network, RL agent performed better as it had learned to handle different job sets at varying cluster loads. With these results, we can conclude that Machine learning RL techniques can be incorporated in real-world network scenarios and these techniques have ample scope in the future as they can be used as alternatives to traditional heuristic based approaches.

8 Future Work

Our experimental results mainly focus on using multi-layer perceptron with fully connected layers for making scheduling decisions. Research work in [11] has shown that using convolutional networks as a replacement to the multi-layer perceptron, can give better overall job slowdown. Convolutional neural networks are used in [11], but it uses the traditional policy gradient algorithm. However, we can improve it further by considering the more advanced algorithms like A2C, but with the addition of a convolutional neural network policy. In our work, we use synthetic data set as an input. However, the study in [4] shows that using real data traces, reinforcement learning models can perform better with good convergence during the training process. During our literature reading, we have gone through some papers where we collected some information about improving the slowdown in which it is mentioned that the alteration to state representation can also improve the slowdown. The main idea was that instead of scheduling the CPU and memory for each job at a time depending on the resource availability, scheduling the CPU of all jobs first and then scheduling the jobs with memory as a requirement helps in achieving a better slowdown.

Furthermore, the input jobs we consider consists of the length of the job, its CPU resource requirement, and its memory requirement. By modifying the reward function we can make the agents to learn a deadline aware policy in which the agents are penalized for missing the job deadline. The environment can also be extended to multi-machine multi-resource [2] scenario in which instead of one machine we have multiple machines available for job scheduling. Also, a fault tolerance mechanism can be introduced so that the jobs whose executions terminate abruptly are scheduled again.

Bibliography

[1] Exponential. `https://numpy.org/doc/stable/reference/random/generated/numpy.random.exponential.html`.

[2] W. Chen, Y. Xu, and X. Wu. Deep reinforcement learning for multi-resource multi-machine job scheduling. *CoRR*, abs/1711.07440, 2017. URL `http://arxiv.org/abs/1711.07440`.

[3] Intel. Stable baselines docs. `https://stable-baselines.readthedocs.io/en/master/`, 2018.

[4] S. Liang, Z. Yang, F. Jin, and Y. Chen. Data centers job scheduling with deep reinforcement learning. In H. W. Lauw, R. C.-W. Wong, A. Ntoulas, E.-P. Lim, S.-K. Ng, and S. J. Pan, editors, *Advances in Knowledge Discovery and Data Mining*, pages 906–917, Cham, 2020. Springer International Publishing. ISBN 978-3-030-47436-2. URL `https://doi.org/10.1007/978-3-030-47436-2_68`.

[5] R. Liaw, E. Liang, R. Nishihara, P. Moritz, J. E. Gonzalez, and I. Stoica. Tune: A research platform for distributed model selection and training. *arXiv preprint arXiv:1807.05118*, 2018.

[6] H. Mao, M. Alizadeh, I. Menache, and S. Kandula. Resource management with deep reinforcement learning. In *Proceedings of the 15th ACM Workshop on Hot Topics in Networks*, HotNets '16, page 50–56, New York, NY, USA, 2016. Association for Computing Machinery. ISBN 9781450346610. doi: 10.1145/3005745.3005750. URL `https://doi.org/10.1145/3005745.3005750`.

[7] V. Mnih, K. Kavukcuoglu, D. Silver, A. A. Rusu, J. Veness, M. G. Bellemare, A. Graves, M. Riedmiller, A. K. Fidjeland, G. Ostrovski, S. Petersen, C. Beattie, A. Sadik, I. Antonoglou, H. King, D. Kumaran, D. Wierstra, S. Legg, and D. Hassabis. Human-level control through deep reinforcement learning. *Nature*, 518(7540):529–533, Feb. 2015. ISSN 00280836. URL `http://dx.doi.org/10.1038/nature14236`.

[8] C. W. M. L. R. Schulman. Openai baselines: Acktr a2c. `https://openai.com/blog/baselines-acktr-a2c/`, 2017.

[9] J. Schulman, S. Levine, P. Moritz, M. I. Jordan, and P. Abbeel. Trust region policy optimization. *CoRR*, abs/1502.05477, 2015. URL `http://arxiv.org/abs/1502.05477`.

[10] J. Schulman, F. Wolski, P. Dhariwal, A. Radford, and O. Klimov. Proximal policy optimization algorithms. *CoRR*, abs/1707.06347, 2017. URL `http://arxiv.org/abs/1707.06347`.

[11] Y. Ye, X. Ren, J. Wang, L. Xu, W. Guo, W. Huang, and W. Tian. A new approach for resource scheduling with deep reinforcement learning, 06 2018.

[12] C. yoon. Understanding actor critic methods. https://towardsdatascience.com/understanding-actor-critic-methods-931b97b6df3f, 2019.