SLA-based Spark Job Scheduling in Cloud with Deep Reinforcement Learning

Muhammed Tawfiqul Islam[1], Shanika Karunasekera[1], Rajkumar Buyya[1]

Abstract

Big data frameworks such as Spark and Hadoop are widely adopted to run analytics jobs in both research and industry. Cloud offers affordable compute resources which are easier to manage. Hence, many organizations are shifting towards a cloud deployment of their big data computing clusters. However, job scheduling is a complex problem in the presence of various Service Level Agreement (SLA) objectives such as monetary cost reduction, and job performance improvement. Most of the existing research does not address multiple objectives together and fail to capture the inherent cluster and workload characteristics. In this paper, we formulate the job scheduling problem of a cloud-deployed Spark cluster and propose a novel Reinforcement Learning (RL) model to accommodate the SLA objectives. We develop the RL cluster environment and implement two Deep Reinforce Learning (DRL) based schedulers in TF-Agents framework. The proposed DRL-based scheduling agents work at a fine-grained level to place the executors of jobs while leveraging the pricing model of cloud VM instances. In addition, the DRL-based agents can also learn the inherent characteristics of different types of jobs to find a proper placement to reduce both the total cluster VM usage cost and the average job duration.

Keywords: Spark, Cloud VM,Cluster-scheduling, SLA, Big Data, Tensorflow Cost-minimization, Performance Improvement, Deep Reinforcement Learning

[1]Cloud Computing and Distributed Systems (CLOUDS) Laboratory
School of Computing and Information Systems
The University of Melbourne, Australia.
Email: {tawfiqul.islam,karus,rbuyya}@unimelb.edu.au

1. Introduction

Big data processing frameworks such as Hadoop[1], Spark[2], Storm[2] became extremely popular due to their use in the data analytics domain in many significant areas such as science, business, and research. These frameworks can be deployed in both on-premise physical resources or on the cloud. However, cloud service providers (CSPs) offer flexible, scalable, and affordable computing resources on a pay-as-you-go model. Furthermore, cloud resources are easy to manage and deploy than physical resources. Thus, many organizations are moving towards the deployment of big data analytics clusters on the cloud to avoid the hassle of managing physical resources. Service Level Agreement (SLA) is an agreed service terms between consumers and service providers, which includes various Quality of Service (QoS) requirements of the users. In the job scheduling problem of a big data computing cluster, the most important objective is the performance improvement of the jobs. However, when the cluster is deployed on the cloud, job scheduling becomes more complicated in the presence of other crucial SLA objectives such as the monetary cost reduction.

In this work, we focus on the SLA-based job scheduling problem for a cloud-deployed Apache Spark cluster. We have chosen Apache Spark as it is one of the most prominent frameworks for big data processing. Spark stores intermediate results in memory to speed up processing. Moreover, it is more scalable than other platforms and suitable for running a variety of complex analytics jobs. Spark programs can be implemented in many high-level programming languages, and it also supports different data sources such as HDFS[3], Hbase[4], Cassandra[5], Amazon S3[3]. The data abstraction of Spark is called Resilient Distributed Dataset (RDD)[6], which by design is fault-tolerant.

When a Spark cluster is deployed, it can be used to run one or more jobs. Generally, when a job is submitted for execution, the framework scheduler is responsible for allocating chunks of resources (e.g., CPU, memory), which are

[2] `https://storm.apache.org/`

[3] `https://aws.amazon.com/s3/`

called executors. A job can run one or more tasks in parallel with these executors. The default Spark scheduler can create the executors of a job in a distributed fashion in the worker nodes. This approach allows balanced use of the cluster and results in performance improvements to the compute-intensive workloads as interference between co-located executors are avoided. Also, the executors of the jobs can be packed in fewer nodes. Although packed placement puts more stress on the worker nodes, it can improve the performance of the network-intensive jobs as communication between the executors from the same job becomes intra-node. However, depending on the target objective, users need to determine the type of executor placement that should be used by the scheduler. This often requires inherent knowledge of both the resources and the workload characteristics. Besides, addressing additional SLA objectives such as cost and performance is not possible by the framework scheduler. There are lots of existing works[7, 8, 9, 10, 11, 12, 13, 14, 15, 16, 17, 18, 19] that focused on various SLA objectives. However, these works do not consider the implication of executor creations along with the target objectives. Most of the works also assume the cluster setup to be homogeneous. However, this is not the case for a cloud-deployed cluster, where different sizes of the VM instances can be used to deploy the cluster to leverage the pricing model to reduce the monetary cost. Finally, heuristic-based and performance model-based solutions often focus on a specific objective, and can not be generalized to adapt to a wide range of objectives while considering the inherent characteristics of the workloads.

Recently, Deep Reinforcement Learning (DRL) based approaches are used to solve complex real-world problems[20], where a DRL agent does not have any prior knowledge of the environment. Instead, it interacts with the real environment, explores different situations, and gathers rewards based on its actions. These experiences are used by the agent to build a policy which maximizes the overall reward. The reward is nothing but a model of the desired objectives. In this paper, we propose DRL-based job scheduling agents for Apache Spark, which addresses the challenges mentioned above. We formulate the scheduling problem and propose an RL model for the job scheduling problem. We also

formulate the reward in a way that it can reflect the target SLA objectives such as monetary cost and average job duration reductions. We implement a Q-Learning-based agent (Deep Q-Learning or DQN), and a policy gradient-based agent (REINFORCE), which automatically learn to schedule jobs efficiently while considering different SLA objectives and the inherent features of the workloads. We also develop a scheduling environment for the cloud-deployed Spark cluster to train the DRL agents. Both the scheduling environment and the agents are developed on top of TensorFlow (TF) Agents. The scheduling agents can interact with the environment to learn about the basics of scheduling, such as satisfying resource capacity and demand constraints for the jobs. Besides, we also train the agents to minimize both the monetary cost of VM usage and the average job duration. The environment states and reward signals drive the learning for an agent. When the agent interacts with the scheduling environment, it gets a reward depending on the chosen action. In our proposed RL model for the scheduling problem, an action is a selection of a worker node (VM) for the creation of an executor of a specific job.

In summary, the **contributions** of this work are as follows:

- We provide an RL model of the Spark job scheduling problem in cloud computing environments. We also formulate the rewards to train DRL-based agents to satisfy resource constraints, optimize cost-efficiency, and reduce average job duration of a cluster.
- We develop a prototype of the RL model in a python environment and plug it to the TF-Agents framework.
- We implement two DeepRL-based agents, DQN and REINFORCE, and train them as scheduling agents in the TF-agent framework.
- We conduct extensive experiments with real-world workload traces to evaluate the performance of the DRL-based scheduling agents and compare them with the baseline schedulers.

The rest of the paper is organized as follows. In section 2, we discuss the

existing works related to this paper. In section 3, we formulate the scheduling problem. In section 4, we present the proposed RL model. In section 5, we describe the proposed DRL-based scheduling agents. In section 6, we exhibit the implemented RL environment. In section 7, we provide the experimental setup, baseline algorithms, and the performance evaluation of the DRL-based agents. In section 7.7, we discuss different strategies learned by the DRL agents and their limitations. Section 8 concludes the paper and highlights future work.

2. Related Work

2.1. Framework Schedulers

Apache Spark uses the (First in First out) FIFO scheduler by default, which places the executors of a job in a distributed manner (spreads out) to reduce overheads on single worker nodes (or VMs if cloud deployment is considered). Although this strategy can improve the performance of compute-intensive workloads, due to the increasing network shuffle operations, network-intensive workloads can suffer from performance overheads. Spark can also consolidate the core usage to minimize the total nodes used in the cluster. However, it does not consider the cost of VMs and the runtime of jobs. Therefore, costly VMs might be used for a longer period, incurring a higher VM cost. Fair[4] and DRF[21] based schedulers improve the fairness among multiple jobs in a cluster. However, these schedulers do not improve SLA-objectives such as cost-efficiency in a cloud-deployed cluster.

2.2. Performance model and Heuristic-based Schedulers

There are a few works which tried to improve different aspects of scheduling for Spark-based jobs. Most of these approaches build performance models based

[4] `https://hadoop.apache.org/docs/stable/hadoop-yarn/hadoop-yarn-site/FairScheduler.html`

on different workload and resource characteristics. Then the performance models are used for resource demand prediction, or to design sophisticated heuristics to achieve one or more objectives.

Sparrow[7] is a decentralized scheduler which uses a random sampling-based approach to improve the performance of the default Spark scheduling. Quasar[8] is a cluster manager that minimizes resource utilization of a cluster while satisfying user-supplied application performance targets. It uses collaborative filtering to find the impacts of different resources on an application's performance. Then this information is used for efficient resource allocation and scheduling. Morpheus[9] estimates job performance from historical traces, then performs a packed placement of containers to minimize cluster resource usage cost. Moreover, Morpheus can also re-provision failed jobs dynamically to increase overall cluster performance. Justice[10] uses deadline constraints of each job with historical job execution traces for admission control and resource allocation. It also automatically adapts to workload variations to provide sufficient resources to each job so that the deadline is met. OptEx[11] models the performance of Spark jobs from the profiling information. Then the performance model is used to compose a cost-efficient cluster while deploying each job only with the minimal set of VMs required to satisfy its deadline. Furthermore, it is assumed that each job has the same executor size, which is the total resource capacity of a VM. Maroulis et al.[12] utilize the DVFS technique to tune the CPU frequencies for the incoming workloads to decrease energy consumption. Li et al.[13] also provided an energy-efficient scheduler where the algorithm assumes that each job has an equal executor size, which is equivalent to the total resource capacity of a VM.

The problems with the performance model and heuristic-based approaches are: (1) they only work on specific SLA objectives and can not be generalized to multiple objectives (2) the performance models depend heavily on the past data, which sometimes can be obsolete due to various changes in the cluster environment (3) it is difficult to tune or modify heuristic-based approaches to incorporate workload and cluster changes. Therefore, recently many researchers

are focusing on RL-based approaches to tackle the scheduling problem in a more efficient and scalable manner.

2.3. DRL-based Schedulers

The application of Deep Reinforcement Learning (DRL) for job scheduling is relatively new. There are a few works which tried to address different SLA objectives of scheduling cloud-based applications.

Liu et al.[22] developed a hierarchical framework for cloud resource allocation while reducing energy consumption and latency degradation. The global tier uses Q-learning for VM resource allocation. In contrast, the local tier uses an LSTM-based workload predictor and a model-free RL based power manager for local servers. Wei et al.[23] proposed a QoS-aware job scheduling algorithm for applications in a cloud deployment. They used DQN with target network and experience replay to improve the stability of the algorithm. The main objective was to improve the average job response time while maximizing VM resource utilization. DeepRM[14] used REINFORCE, a policy gradient DeepRL algorithm for multi-resource packing in cluster scheduling. The main objective was to minimize the average job slowdowns. However, as all the cluster resources are considered as a big chunk of CPU and memory in the state space, the cluster is assumed to be homogeneous. Decima[15] also uses a policy gradient agent and has a similar objective as DeepRM. Here, both the agent and the environment was designed to tackle the DAG scheduling problems within each job in Spark, while considering interdependent tasks. Li et al.[24] considered an Actor Critic-based algorithm to deal with the processing of unbounded streams of continuous data with high scalability in Apache Storm. The scheduling problem was to assign workloads to particular worker nodes, while the objective was to reduce the average end-to-end tuple processing time. This work also assumes the cluster setup to be homogeneous and does not consider cost-efficiency. DSS[17] is an automated big-data task scheduling approach in cloud computing environments, which combines DRL and LSTM to automatically predict the VMs to which each incoming big data job should be scheduled to improve

Table 1: Comparison of the related works

Features	Related Work										Our Work
	[22]	[23]	[14]	[15]	[24]	[17]	[25]	[18]	[26]	[27]	
Performance Improvement	✓	✓	✓	✓	✓	✓	✓	✓	✓	✓	✓
Cost-efficiency	✗	✓	✗	✗	✗	✓	✗	✓	✓	✓	✓
Multi-Objective	✓	✓	✗	✗	✗	✓	✗	✓	✓	✓	✓
Multiple VM Instance types	✗	✗	✗	✗	✗	✗	✗	✗	✓	✗	✓
Fine-grained Executor	✗	✗	✗	✗	✗	✗	✗	✗	✗	✗	✓

the performance of big data analytics while reducing the resource execution cost. Harmony[25] is a deep learning-driven ML cluster scheduler that places training jobs in a way that minimizes interference and maximizes average job completion time. It uses an Actor Critic-based algorithm and job-aware action space exploration with experience replay. Besides, it has a reward prediction model, which is trained using historical samples and used for producing reward for unseen placement. Cheng et al.[18] used a DQN-based algorithm for Spark job scheduling in Cloud. The main objective of this work is to optimize the bandwidth resource cost, along with node and link energy consumption minimization. Spear[26] works to minimize the makespan of complex DAG-based jobs while considering both task dependencies and heterogeneous resource demands at the same time. Spear utilizes Monte Carlo Tree Search (MCTS) in task scheduling and trains a DRL model to guide the expansion and roll-out steps in MCTS. Wu et al.[27] proposed an optimal task allocation scheme with a virtual network mapping algorithm based on deep CNN and value-function based Q-learning. Here, tasks are allocated onto proper physical nodes with the objective being the long-term revenue maximization while satisfying the task requirements. Thamsen et al.[19] used a Gradient Bandit method to improve the resource utilization and job throughput of Spark and Flink jobs, where the RL model learns the co-location goodness of different types of jobs on shared resources.

In summary, most of these existing approaches focus mainly on performance improvement. Furthermore, these works also assume that each job/task will be assigned to one VM or worker-node only. Moreover, many works also assume the cluster nodes to be homogeneous, which may not be the case when the cluster is deployed on the cloud. Thus, these works do not consider a fine-grained level of executor placement in Spark job scheduling. In contrast, our scheduling agents can place executors from the same job in different VMs (when needed, to optimize for a specific policy), and guarantees to launch all of the executors of a job on the required resources. In addition, our agents can handle different sizes of executors of jobs, and different VM instance sizes with a pricing model. Furthermore, our agent can be trained to optimize a single objective such as cost-efficiency or performance improvement. In addition, our agent can also be trained to balance between multiple objectives. Lastly, the proposed scheduling agents can learn the inherent characteristics of the jobs to find the proper placement strategy to improve the target objectives, without any prior information on the jobs or the cluster. A summary of the comparison between our work and other related works is shown in Table 1.

3. Problem Formulation

We consider a Spark cluster set up using cloud Virtual Machines (VM) as the worker nodes. Generally, Cloud Service Providers (CSPs) offer different instance types for VMs where each type varies on resource capacity. For our problem, we assume that any type or a mix of different types of VM instances can be used to deploy the cluster.

In the deployed cluster, one or more jobs can be submitted by the users; and the users specify the resource demands for their submitted jobs. The job specification contains the total number of executors required and the size of all these executors in-terms of CPU and memory. A job can be of different types and can be submitted at any time. Therefore, job arrival times are stochastic, and the job scheduler in the cluster has no prior knowledge about the arrival

Table 2: Definition of Symbols

Symbol	Definition
N	The total number of VMs in the cluster
M	The total number of jobs in the cluster
E	The current total number of executors running in the cluster
ψ	The index set of all the jobs, $\psi = 1, 2, ..., N$
δ	The index set of all the VMs, $\delta = 1, 2, ..., M$
ω	The index set of all the current executors, $\omega = 1, 2, ..., E$
vm^i_{cpu}	CPU capacity of a VM, $i \in \delta$
vm^i_{mem}	Memory capacity of a VM, $i \in \delta$
vm^i_{price}	Unit price of a VM, $i \in \delta$
vm^i_T	The total time a VM was used, $i \in \delta$
e^k_{cpu}	CPU demand of an executor, $k \in \omega$
e^k_{mem}	Memory demand of an executor, $k \in \omega$
job^j_T	The completion time of a job, $j \in \psi$

of jobs. The scheduler processes each job on a FCFS (First Come First Serve) basis, which is the usual way of handling jobs in a big data cluster. However, the scheduler has to choose the VMs where the executors of the current job should be created. The target of the scheduler is to reduce the overall monetary cost of the whole cluster for all the jobs. In addition, it has an additional target of reducing job completion times. The notation of symbols for the problem formulation can be found in table 2.

Suppose N is the total number of VMs that were used to deploy a Spark cluster. These VMs can be of any instance type/size (which means the resource capacities may vary in-terms of CPU and memory). M is the total number of jobs that need to be scheduled during the whole scheduling process. When a job is submitted to the cluster, the scheduler has to create the executors in one or more VMs and has to follow the resource capacity constraints of the VMs and the resource demand constraints of the current job. The users submit the resource demand of an executor in two dimensions – CPU cores and memory. Therefore, each executor of a job can be treated as a multi-dimensional box

that needs to be placed to a particular VM (bin) in the scheduling process. Therefore, the CPU and memory resource demand and capacity constraints can be defined as follows:

$$\sum_{k \in \omega} (e^k_{cpu} \times x_{ki}) \leq vm^i_{cpu} \qquad \forall i \in \delta \tag{1}$$

$$\sum_{k \in \omega} (e^k_{mem} \times x_{ki}) \leq vm^i_{mem} \qquad \forall i \in \delta \tag{2}$$

where x_{ki} is a binary decision variable which is set to 1 if the executor k is placed in the VM i; otherwise it is set to 0.

When an executor for a job is created, resources from only 1 VM should be used and the scheduler should not allocate a mix of resources from multiple VMs to one executor. This constraint can be defined as follows:

$$\sum_{i \in \delta} x_{ki} = 1 \qquad \forall k \in \omega \tag{3}$$

After the end of the scheduling process, the cost incurred by the scheduler for running the jobs can be defined as follows:

$$Cost_{total} = \sum_{i \in \delta} (vm^i_{price} \times vm^i_T) \tag{4}$$

Additionally, we can define the average job completion times for all the jobs as follows:

$$Avg_T = (\sum_{j \in \psi} job^j_T)/M \tag{5}$$

As we want to minimize both the cost of using the cluster and the average job completion time for the jobs, the optimization problem is to minimize the following:

$$\beta \times Cost + (1 - \beta) \times Avg_T \tag{6}$$

where $\beta \in [0, 1]$. Here, β is a system parameter which can be set by the user to specify the optimization priority for the scheduler. Note that, equation 6 can be generalized to address additional objectives if required.

The above optimization problem is a mixed-integer linear programming (MILP) [28] and non-convex [29], generally known as the NP-hard problem [30]. To solve this problem optimally, an optimal scheduler needs to know the job completion times before making any scheduling decisions. This makes the scheduler design extremely difficult as it requires the collection of job profiles and the modeling of job performance which depends on various system parameters. Furthermore, if the number of jobs, executors and the total cluster size increase, solving the problem optimally may not be feasible. Although, heuristics-based algorithms are highly scalable to solve the problem, they do not generalize over multiple objectives and also do not capture the inherent characteristics of both the cluster and the workload to improve the target goal.

4. Reinforcement Learning (RL) Model

Reinforcement learning (RL) is a general framework where an agent can be trained to complete a task through interacting with an environment. Generally, in RL, the learning algorithm is called the agent, whereas the problem to be solved can be represented as the environment. The agent can continuously interact with the environment and vice versa. During each time step, the agent can take an action on the environment based on its policy ($\pi(a_t|s_t)$). Thus, the action (a_t) of an agent depends on the current state (s_t) of the environment. After taking the action, the agent receives a reward (r_{t+1}) and the next state (s_{t+1}) from the environment. The main objective of the agent is to improve the policy so that it can maximize the sum of rewards.

In this paper, the learning agent is a job scheduler which tries to schedule jobs in a Spark cluster while satisfying resource demand constraints of the jobs, and the resource capacity constraints of the VMs. The reward it gets from the environment is directly associated with the key scheduling objectives such as cost-efficiency, and the reduction of average job duration. Therefore, by maximizing the reward, the agent learns the policy which can optimize the target objectives. Fig. 1 shows the proposed RL framework of our job schedul-

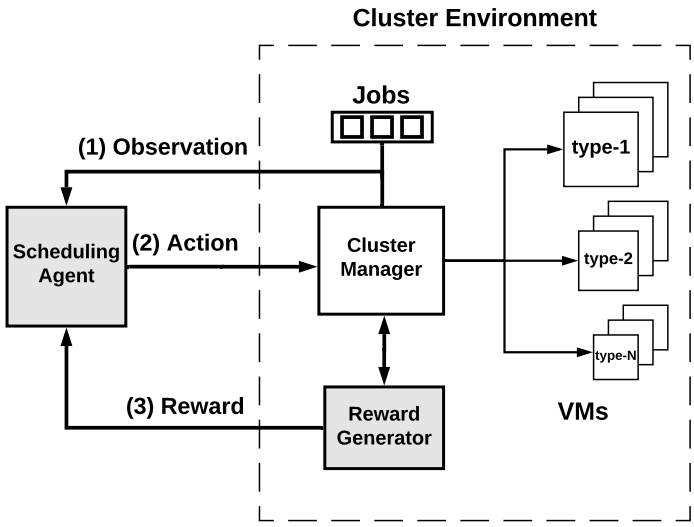
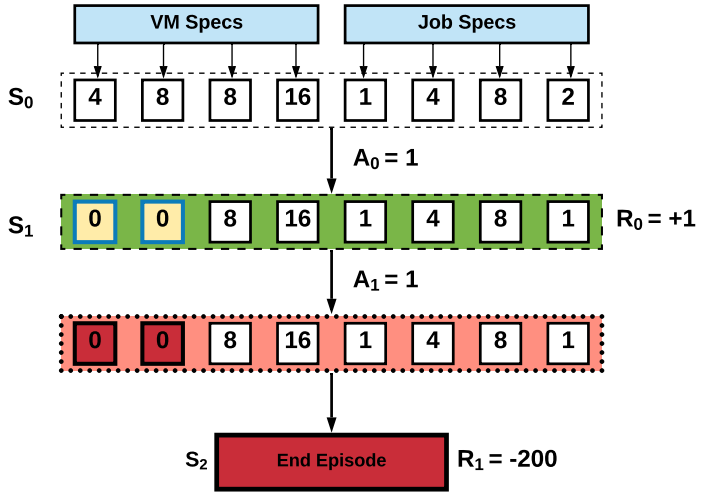


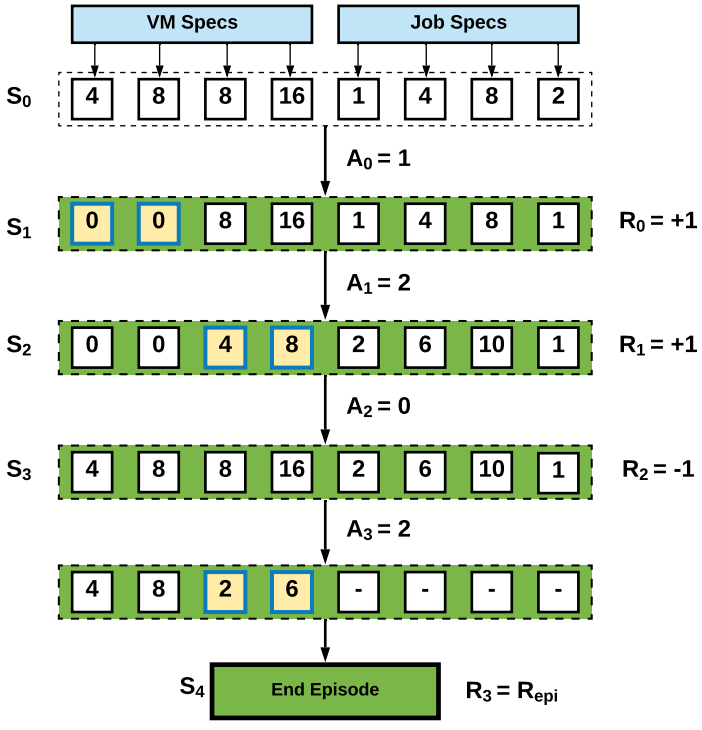
Figure 1: The proposed RL model for the job scheduling problem, where a scheduling agent is interacting with the cluster environment.

ing problem. We treat all the components as part of the cluster environment (highlighted with the big dashed rectangle), except the scheduler. The cluster manager monitors the state of the cluster. It also controls the worker nodes (or cloud VMs) to place executor(s) for any job. In each time-step, the scheduling agent gets an observation from the environment, which includes both the current job's resource requirement, and also the current resource availability of the cluster (exposed by the cluster monitor metrics from the cluster manager). An action is the selection of a specific VM to create a job's executor. When the agent takes an action, it is carried out in the cluster by the cluster manager. After that, the reward generator calculates a reward by evaluating the action on the basis of the predefined target objectives. Note that, in RL environment, the reward given to agent is always external to the agent. However, the RL algorithms can have their internal reward (or parameter) calculations which they continuously update to find a better policy.

We assume the time-steps in our model to be discrete, and event driven.



(a) Failed Episode



(b) Successful Episode

Figure 2: Example scenarios for state transitions in the proposed environment.

Therefore, the state-space moves from one time-step to the next only after an agent takes an action. The key components of the RL model are specified as follows:

Agent: The agent acts as the scheduler which is responsible for scheduling jobs in the cluster. At each time step, it observes the system state and takes an action. Based on the action, it receives a reward and the next observable state from the environment.

Episode: An episode is the time interval from when the agent sees the first job and the cluster state to when it finishes scheduling all the jobs. In addition, an episode can be terminated early if the agent chooses a a certain bad action.

State Space: In our scheduling problem, the state is the observation an agent gets after taking each action. The scheduling environment is the deployed cluster where the agent can observe the cluster state after taking each action. However, only at the start of the scheduling process or an episode, the agent receives the initial state without taking any action. The cluster states have the following parameters: CPU and memory resource availability of all the VMs in the cluster, the unit price of using each VM in the cluster, and the current job specification which needs to be scheduled. Actions are the decisions or placements the agent (scheduler) makes to allocate resources for each of the executors of a job. Resource allocation for each executor is considered as one action, and after each action, the environment returns the next state to the agent. The current state of the environment can be represented using a 1-dimensional vector, where the first part of the vector is the VM specifications: $[vm^1_{cpu}, vm^1_{mem}, ...vm^N_{cpu}, vm^N_{mem}]$, and the second part of the vector is the current job's specification: $[jobID, e_{cpu}, e_{mem}, j_E]$. Here, $vm^1_{cpu}, ...vm^N_{cpu}$ represents the current CPU availability of all the N VMs of the cluster, whereas $vm^1_{mem}, ...vm^N_{mem}$ represents the current memory availability of all the N VMs of the cluster. $jobID$ represents the current jobID, e_{cpu} and e_{mem} represents the CPU and memory demand of one executor of the current job, respectively. As all the executors for one job have the same resource demand, the only other required information is the total number of executors that has to be created for

that job, which is represented by j_E. Therefore, the state-space grows larger only with the increase of the size of the cluster (total number of VMs), and does not depend on the total number of jobs. Each job's specification is only sent to the agent as part of the state after its arrival if all the previous jobs are already scheduled. After each successive action, the cluster resource capacity will be updated due to the executor placement. Therefore, the next state will reflect the updated cluster state after the most recent action. Until all the executors of the current job is placed, the agent will keep receiving the job specification of the current job, with the only change being the j_E parameter which will be reduced by 1 after each executor placement. When it becomes 0 (the job is scheduled successfully), only then the next job specification will be presented along with the updated cluster parameters as the next state.

Action Space: The action is the selection of the VM where one executor for the current job will be created. If the cluster does not have sufficient resources to place one or all the executors of the current job, the agent can also do nothing and wait for previously scheduled jobs to be finished. In addition, to optimize a certain objective (e.g, cost, time), the agent may decide not to schedule a job right after it arrives. Therefore, if there are N number of VMs in the cluster, there are $N + 1$ number of possible discrete actions. Here, we define Action 0 to specify that the agent will be waiting and no executor will be created, where action 1 to N specifies the index of the VM chosen to create an executor for the current job.

Reward: The agent receives a reward whenever it takes an action. There can be either a positive or a negative reward for each action. Positive reward motivates the agent to take a good action and also to optimize the overall reward for the whole episode. In contrast, negative rewards generally train the agent to avoid bad actions. In RL, when we want to maximize the overall reward at the end of each episode, an agent has to consider both the immediate and the discounted future reward to take an action. The overall goal is to maximize the cumulative reward over an episode, so sometimes taking an action which incurs an immediate negative reward might be a good step towards bigger positive

rewards in future steps. Now, we define both the immediate reward and episodic reward for an agent.

Suppose, in the worst case, all the VMs are turned on for the whole duration of the episode, where each job took its maximum time to complete because of bad placement decisions. Thus it gives us the maximum cost that can be incurred by a scheduler in an episode as:

$$Cost_{max} = \sum_{j \in \psi} job^{j}_{Tmax} \times \sum_{i \in \delta} vm^{i}_{price} \tag{7}$$

Therefore, if we find the episodic VM usage $Cost_{total}$ incurred by an agent (as shown in Eqn. 4), the normalized episodic cost can be defined as:

$$Cost_{normalized} = \frac{Cost_{total}}{Cost_{max}} \tag{8}$$

Depending on the priority of the cost objective, the β parameter can be used to find the episodic cost as:

$$Cost_{epi} = \beta \times (1 - Cost_{normalized}) \tag{9}$$

In an ideal case where all the jobs' executors are placed according to the job type (e.g., distributed placement of executors for CPU-bound jobs, compact placement of network-bound jobs), we can get the minimum average job completion time for an episode as follows:

$$Avg_{Tmin} = (\sum_{j \in \psi} job^{j}_{Tmin})/M \tag{10}$$

Similarly, if all the jobs' executors are not placed according to the job characteristics, we can get the maximum average job completion time for an episode as follows:

$$Avg_{Tmax} = (\sum_{j \in \psi} job^{j}_{Tmax})/M \tag{11}$$

Therefore, if we find the episodic average job completion time for an agent (as shown in Eqn. 5), the normalized episodic average job completion time can be defined as:

$$Avg_{Tnormalized} = \frac{Avg_T - Avg_{Tmin}}{Avg_{Tmax} - Avg_{Tmin}} \tag{12}$$

Depending on the priority of the average job duration objective, the β parameter can be used to find the episodic average job duration as:

$$Avg_{Tepi} = (1 - \beta) \times (1 - Avg_{Tnormalized}) \tag{13}$$

Let R_{fixed} is a fixed episodic reward which will be scaled up or down based on how the agent performs in each episode to maximize the objective function. Thus the final episodic reward R_{epi} can be defined as:

$$R_{epi} = R_{fixed} \times (Cost_{epi} + Avg_{Tepi}) \tag{14}$$

Note that, $\beta \in [0,1]$. In addition, both $Cost_{epi}$ and $Avg_{Tepi} \in [0,1]$. Therefore, the sum of $Cost_{epi}$ and Avg_{Tepi} can be at most 1, which will lead to a reward of exactly R_{fixed}. For example, if β is chosen to be 0, it means an agent will be trained to reduce average job duration only. In the best case scenario, if an agent can achieve Avg_T to be equal to Avg_{Tmin}, the value of Avg_{Tepi} will be 1 and the value of $Cost_{epi}$ will be 0. Therefore, the value R_{epi} will be equal to R_{fixed} which indicates the agent has learned the most time-optimized policy.

Example workout of the state-action-reward space: We show an example workout of the state, action and reward of the proposed RL model in Fig. 2. In this example scheduling scenarios, the cluster is composed with 2 VMs with specifications: $VM_1 \rightarrow \{cpu = 4, mem = 8\}$, and $VM_2 \rightarrow \{cpu = 8, mem = 16\}$. In addition, two jobs arrive one after another with specifications: $job_1 \rightarrow \{jobID = 1, e_{cpu} = 4, e_{mem} = 8, j_E = 2\}$, and $job_2 \rightarrow \{jobID = 2, e_{cpu} = 6, e_{mem} = 10, j_E = 1\}$. Now, Fig. 2a shows a scenario where the agent has chosen VM_1 twice to place both of the executors of job_1. After the first placement, the agent received a positive reward $R_0 = 1$ as it was a valid

placement. As VM_1 did not have any space left to accommodate anything, the second action taken by the agent was invalid, thus the environment did not execute that action. Instead, the episode was terminated and the agent was given a high negative reward (-200). In the second scenario shown in Fig. 2b, the agent successfully placed all the executors for job_1. Then as there was not sufficient resources to place the executor of the job_2, the agent has chosen to wait (Action 0) for resources to be freed. After job_1 is finished, resources are freed. Then the agent successfully placed the executor for job_2, ends the episode and gets the episodic reward.

5. DeepRL Agents for Job Scheduling

To solve the job scheduling problem in the proposed RL environment, we use two DRL-based algorithms. The first one is Deep Q-Learning (DQN), which is a Q-Learning based approach. The other one is a policy gradient algorithm which is called REINFORCE. Both of the algorithms work with any RL environment with discrete state and action spaces.

5.1. DQN Agent

5.1.1. Q-Learning:

Q-Learning works by finding the *Quality* of a state-action value, which is called Q-function. Q-function of a policy π, $Q^{\pi}(s, a)$ measures the expected sum of rewards acquired from state s by taking action a first and then using policy π at each step after that. The optimal Q-function $Q^{*}(s, a)$ is defined as the maximum return that can be received by an optimal policy. The optimal Q-function can be defined as follows by the Bellman optimality equation:

$$Q^{*}(s, a) = \mathbb{E}\left[r + \gamma \max_{a'} Q^{*}(s', a')\right] \tag{15}$$

Here, γ is the discount factor which determines the priority of a future reward. For example, a high value of γ helps the learning agent to achieve more future rewards, while a low value in γ motivates to focus only on the immediate

reward. For the optimal policy, the total sum of rewards can be received by following the policy until the end of a successful episode. The expectation is measured over the distribution of immediate rewards of r and the possible next states s'.

In Q-Learning, the Bellman optimality equation is used as an iterative update $Q_{i+1}(s,a) \leftarrow \mathbb{E}\left[r + \gamma \max_{a'} Q_i(s',a')\right]$, and it is proved that it converges to the optimal function Q^*, i.e. $Q_i \rightarrow Q^*$ as $i \rightarrow \infty$ [31].

5.1.2. Deep Q-Learning (DQN):

Q-learning can be solved as dynamic programming (DP) problem, where we can represent the Q-function as a 2-dimensional matrix containing values for each combination of s and a. However, in high-dimensional spaces (the total number of state and action pairs are huge), the tabular Q-learning solution is infeasible. Therefore, a neural is generally trained with parameters θ, to approximate the Q-values, i.e., $Q(s,a;\theta) \approx Q^*(s,a)$. Here, the following loss at each step i needs to be minimized:

$$L_i(\theta_i) = \mathbb{E}_{s,a,r,s' \sim \rho(.)}\left[(y_i - Q(s,a;\theta_i))^2\right] \tag{16}$$

where $y_i = r + \gamma \max_{a'} Q(s',a';\theta_{i-1})$.

Here, ρ is the distribution over transitions $\{s,a,r,s'\}$ sampled from the environment. y_i is called the Temporal Difference (TD) target, and $y_i - Q$ is called the TD error.

Note that the target y_i is a changing target. In supervised learning, we have a fixed target. Therefore, we can train a neural net to keep moving towards the target at each step by reducing the loss. However, in RL, as we keep learning about the environment gradually, the target y_i is always improving, and it seems like a moving target to the network, thus making it unstable. A target network has fixed network parameters as it is a sample from the previous iterations. Thus, the network parameters from the target network are used to update the current network for stable training.

Furthermore, we want our input data to be independent and identically distributed (i.i.d.). However, within the same trajectory (or episode), the iterations are correlated. While in a training iteration, we update model parameters to move $Q(s, a)$ closer to the ground truth. These updates will influence other estimations and will destabilize the network. Therefore, a circular **replay-buffer** can be used to hold the previous transitions (state, action, reward samples) from the environment. Therefore, a mini-batch of samples from the replay buffer is used to train the deep neural network so that the data will be more independent and similar to i.i.d.

DQN is an off-policy algorithm that it uses a different policy while collecting data from the environment. The reason is if the ongoing improved policy is used all the time, the algorithm may diverge to a sub-optimal policy due to the insufficient coverage of the state-action space. Therefore, an ϵ-greedy policy is used that selects the greedy action with probability $1 - \epsilon$ and a random action with probability ϵ so that it can observe any unexplored states, which ensures that the algorithm does not get stuck in local maxima. The DQN [32] algorithm we use with replay buffer and target network is summarized in Algorithm 1.

ALGORITHM 1: DQN Algorithm

foreach *iteration* 1 ... N **do**
    Collect some samples from the environment by using the collect policy (ϵ-greedy), and store the samples in the replay buffer
    Sample a batch of data from the replay buffer
    Update the agent's network parameter θ (Using Eqn. 16)
end

5.2. REINFORCE Agent

DQN optimizes for the state-action values, and by doing so, it indirectly optimizes for the policy. However, the policy gradient methods operate on modelling and optimizing the policy directly. The policy is usually modelled with a parameterized function with respect to θ, written as π_θ. Accordingly,

$\pi(a_t|s_t)$ is the probability of choosing the action a_t given a state s_t at time step t. The amount of the reward an agent can get depends on this policy.

In a conventional policy gradient algorithm, a batch of samples is collected in each iteration, then the update shown in Eqn. 17 is applied to the policy using the collected samples.

$$_\theta\mathbb{E}_{\pi_\theta}\left[\sum_{t=0}^{T}\gamma^t r_t\right] = \mathbb{E}_{\pi_\theta}\left[\sum_{t=0}^{T}{}_\theta log\pi_\theta(a_t|s_t)R_t\right] \tag{17}$$

Here, γ is the discount factor, whereas s_t, a_t, and r_t are used to represent the state, action, and reward at time t, respectively. T is the length of any single episode. R_t is the discounted cumulative return, which can be computed as shown in Eqn. 18.

$$R_t = \sum_{t'=t}^{T}\gamma^{t'-t}r_t \tag{18}$$

Here, t' starts from the current time step t, which means that if the current action is taken, we will get an immediate reward of r_t, which also influences on how much reward we can accumulate up to the end of the episode.

The expected return is shown in Eqn. 17 uses the maximum log-likelihood, which measures the likelihood of an observed data. In RL context, it means how likely we can expect the current trajectory under the current policy. When the likelihood is multiplied with the reward, the likelihood of a policy is increased if it generates a positive reward. On the other hand, the likelihood of the policy is decreased if it gives a less or a negative reward. In summary, the model tries to keep the policy which worked better and tends to throw away policies which did not work well. However, as the formula is shown as an expectation, it cannot be used directly. Therefore, a sampling-based estimator is used instead, which is shown in Eqn. 19.

$$_\theta J(\theta) \approx \frac{1}{N}\sum_{i=1}^{N}\left(\sum_{t=1}^{T}{}_\theta log\pi_\theta(a_{i,t}|s_{i,t})\right)\left(\sum_{t=1}^{T}r(s_{i,t},a_{i,t})\right) \tag{19}$$

Here, $_\theta J(\theta)$ is the policy gradient of the target objective J, parameterized with θ. We also assume that in each iteration, N trajectories are sampled (

ALGORITHM 2: REINFORCE Algorithm

1 **foreach** *iteration* 1 ... N **do**
2 Sample τ_i from $\pi_\theta(a_t|s_t)$ by following the current policy in the environment
3 Find the policy gradient ${}_\theta J(\theta)$ (Using Eqn. 19)
4 $\theta \leftarrow \theta + \alpha_\theta J(\theta)$
5 **end**

$\tau_1, ..., \tau_N$), where each trajectory τ_i is a list of states, actions, and rewards: $\tau_i = s_t^i, a_t^i, r_t^i$ for time-steps $t = 0$ to $t = T_i$. In this work, we use the **REINFORCE**[33] algorithm, as shown in Algorithm 2. This algorithm works by utilizing Monte Carlo roll-outs (learning by computing the reward after executing a whole episode). After the collection step (line 2), the algorithm updates the underlying network using the updated policy gradient with a learning parameter α (line 4). Note that, while sampling a trajectory, the ϵ-greedy policy is used.

6. RL Environment Design and Implementation

We have developed a simulation environment in *Python* to represent a cloud-deployed Spark cluster. The environment holds the state of the cluster, and an agent can interact with it by observing states, and taking any action. Whenever an action is taken, the immediate reward can be observed, but the episodic reward can only be observed after the completion of an episode. The episodic reward may be positive or negative depending on whether an episode was completed successfully or terminated early following a bad action taken by the agent. The features of our developed environment are summarized as follows:

1. The environment exposes the state (comprised of the latest cluster resource statistics and the next job), to the agent in each time step.
2. After an action is taken by an agent, the environment can detect valid/invalid placements and assign positive/negative rewards accordingly.

Table 3: The action-event-reward mapping of the proposed RL environment.

No.	Action	Event	Reward
1	0	Previously placed 1 or more executors of the current job, but now waiting to place any remaining executor (s)	-200
2	0	No placement	-1
3	1 ... N	Proper placement of one executor of the current job	+1
4	1 ... N	Improper placement of an executor: resource capacity or resource demand constraints violation	-200
5	1 ... N	All jobs are scheduled, a successful completion of an episode	R_{epi} (Eqn. 14)

3. Based on the agent's performance in an episode, the environment can award the episodic reward (the environment acts as a reward generator). Therefore, for a simulated cluster with a workload trace, the environment can derive the cost and time values which are required to find the episodic reward.
4. The environment can also vary the job durations from the goodness of an agent's executor placement.

As mentioned before, instead of representing fixed intervals of real-time; the time-steps refer to arbitrary progressive stages of decision-making and acting. We have incorporated TF-agents API calls to return the transition or termination signals after each time step. Fig. 3 shows the workflow of the environment during the agent training process. The red and green circles indicate the events which trigger negative and positive rewards, respectively, from the environment. A summary of the *'action leading to the event and reward'* is summarized in Table 3. In this table, the serial No. of each reward corresponds to the red/green circle shown in Fig. 3. The implemented environment can be used with TF-agents to train one or more DRL agents. Specifically, the agents can be trained to achieve one or more target objectives such as cost-efficiency, performance improvement. As discussed before, we have designed the reward signals to achieve both cost-efficiency and average job duration reduction. The implemented environment can be extended to change or incorporate one or more rewards/objectives. We call the implemented environment **RM_DeepRL**, which

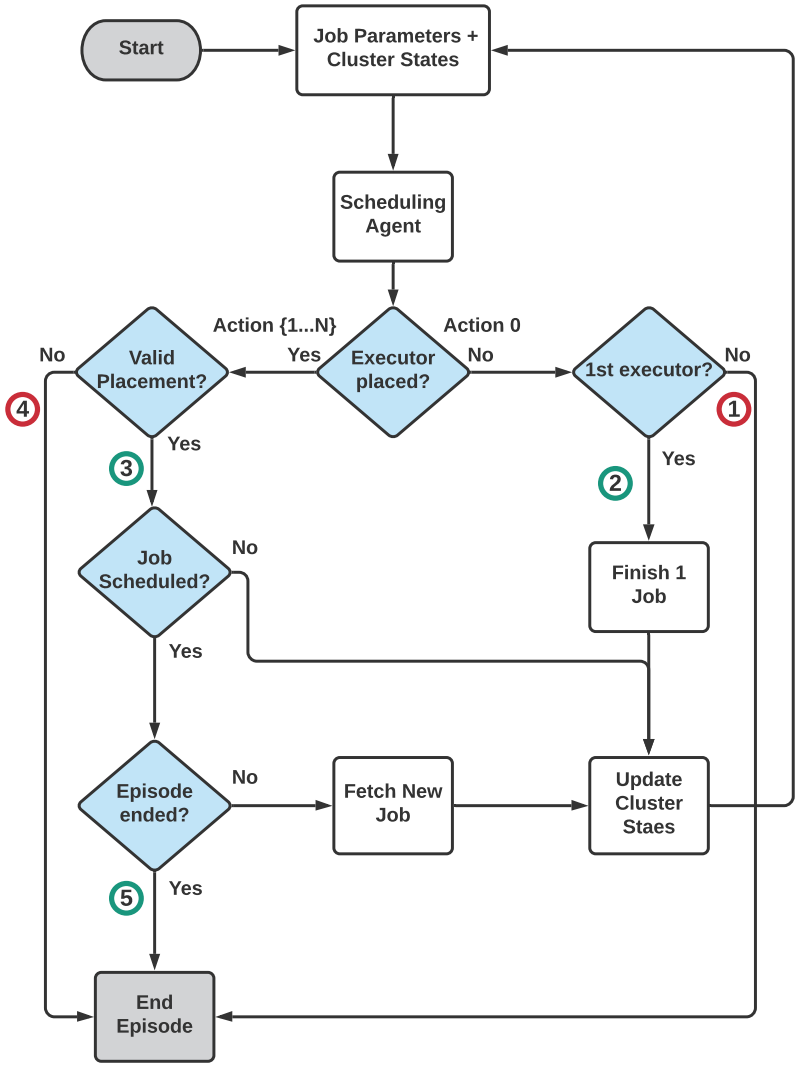


Figure 3: The workflow of the proposed environment in response to different agent actions. The red and green circles indicate the events which trigger negative and positive rewards, respectively, from the environment.

Table 4: Cluster Resource Details

Instance Type	CPU Cores	Memory (GB)	Quantity	Price
m1.large	4	16	4	$0.24/h
m1.xlarge	8	32	4	$0.48/h
m2.xlarge	12	48	4	$0.72/h

is an open-source RL-based cluster scheduling environment[5] with TensorFlow-Agents as the backend.

7. Performance Evaluation

In this section, we first discuss the experimental settings which include the cluster resource details, workload generation, and baseline schedulers. Then, we present the evaluation and comparison of the DRL agents with the baseline scheduling algorithms.

7.1. Experimental Settings

Cluster Resources: We have chosen different VM instance types with various pricing models so that we can train and evaluate an agent to optimize cost while the cluster is deployed on public cloud. The cluster resource details are summarized in Table 4. Note that, the pricing model of the VM instances is similar to the AWS EC2 instance pricing (in Australia).

Workload: We have used the BigDataBench[34] benchmark suite and took 3 different applications from it as jobs in the cluster which are: WordCount (CPU-intensive), PageRank (Network or IO intensive) and Sort (memory-intensive). We have randomly set job requirements within a range of 1-6 (for CPU cores), 1-10 (for memory in GB), and 1-8 (for total executors) and then profiled each job in the real Spark cluster 10 times to find the average job duration. Note that, the real cluster also has the same cluster resources as mentioned in Table 4.

[5] `https://github.com/tawfiqul-islam/RM_DeepRL`

Table 5: Hyper-parameters for DRL-agents and the environment parameters.

Parameter	Value	Parameter	Value
R_{fixed}	10000	Optimization Priority (β)	[0.0,0.25,0.50,0.75,1.00]
Batch Size	64	No. of Fully Connected Layers for Q-Network	200
No. of Evaluation Episodes	10	Policy Evaluation Interval	1000
Epsilon (ϵ)	0.001	Training Iteration	10000 (Normal) 20000 (Burst)
Learning Rate (α)	0.001	Optimizer	AdamOptimizer
Discount Factor (γ)	0.9	Job duration increase for a bad placement	30%
Collect Steps per Iteration (DQN) Collect Episodes per Iteration (RE)	10 10	Avg_{Tmin}, Avg_{Tmax}	Profiled from real runs of the corresponding job
Replay Buffer Size	10000	Avg_T, $Cost_{max}$	Dynamically calculated by the environment depending on the cluster and workload specs

The job arrival times: Job arrival rates of 24 hours is extracted from the Facebook Hadoop Workload Trace[6] to be used as the job arrival times in the simulation. We have chosen job arrival patterns: normal (50 jobs arriving in a 1-hour time period), and burst (100 jobs arriving in only 10 minutes).

Baseline Schedulers: We have used 4 different baselines to compare with the DRL-based algorithms. These are:

1. **Round Robin (RR):** The default approach of the Spark Scheduler to distributively place the executors in VMs.
2. **Round Robin Consolidate (RRC):** Another round-robin approach of the Spark scheduler to minimize the total number of VMs used. Note that it works by packing executors on the already running VMs to avoid launching unused VMs.
3. **First Fit (FF):** We develop this baseline to place as many executors as possible to the first available VM to reduce cost.
4. **Integer Linear Programming (ILP):** This algorithm uses a Mixed ILP solver to find optimal placement of all the executos of the current

[6] `https://github.com/SWIMProjectUCB/SWIM/wiki/Workloads-repository`

job. During each decision making step, the whole optimization problem is dynamically generated by using the current cluster state and the job specification. In addtion, to improve the performance we have used job profile information to include the estimated job completion time within the model so that the problem can be solved optimally.

Note that, all the baseline schedulers, and our proposed scheduling agents make dynamic decisions from the current view of the cluster, and do not have a global view of the whole problem. In addition, the schedulers have no knowledge about the job characteristics which determine the placement goodness for a particular type of job.

TensorFlow Cluster details: We have used 4 VMs (each with 16 CPU cores and 64GB of memory) from the Nectar Research Cloud[7] to train the DRL agents. The TensorFlow version 2.0, and TF-Agent version 0.5.0 were installed along with python 3.7 in each of the VMs.

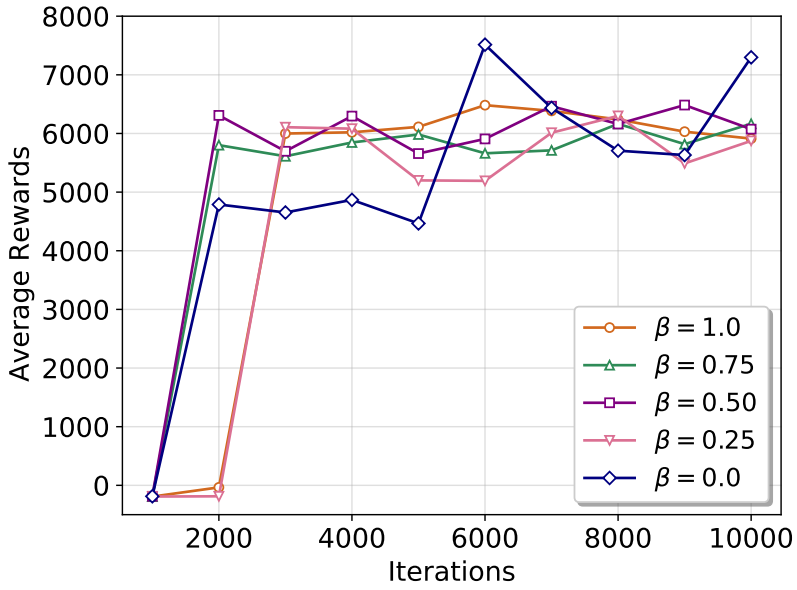
Hyperparameters: Hyper-parameters settings for both DQN and REINFORCE agents, along with other environment parameters are listed in Table 5.

7.2. Convergence of the DRL Agents

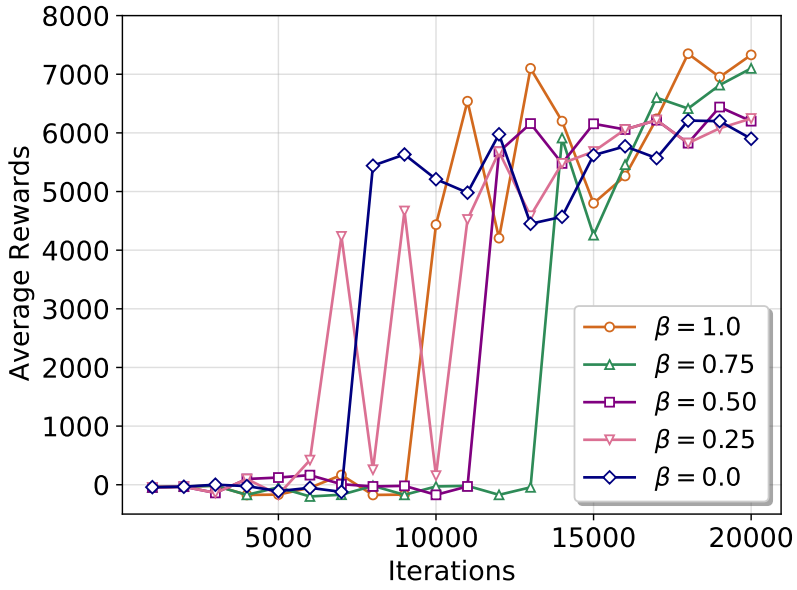
Fig. 4 and Fig. 5 represent the convergence of the DQN and REINFORCE algorithms, respectively. We have trained the DRL-agents with varying β parameter values to showcase the effects of single or multiple reward maximization. The evaluation of the algorithms is done after every 1000 iterations, where we calculate the average rewards from the 10 test runs of the trained policy. For the normal job arrival pattern, we have trained the agents for 10000 iterations, and for the burst job arrival pattern, we have trained the agents for 20000 iterations.

A higher value of β indicates that the agent is rewarded more for optimizing VM usage cost. In contrast, a lower value of β indicates the agent is optimized more for the reduction of average job duration. We have varied the values of β

[7] `www.nectar.org.au`

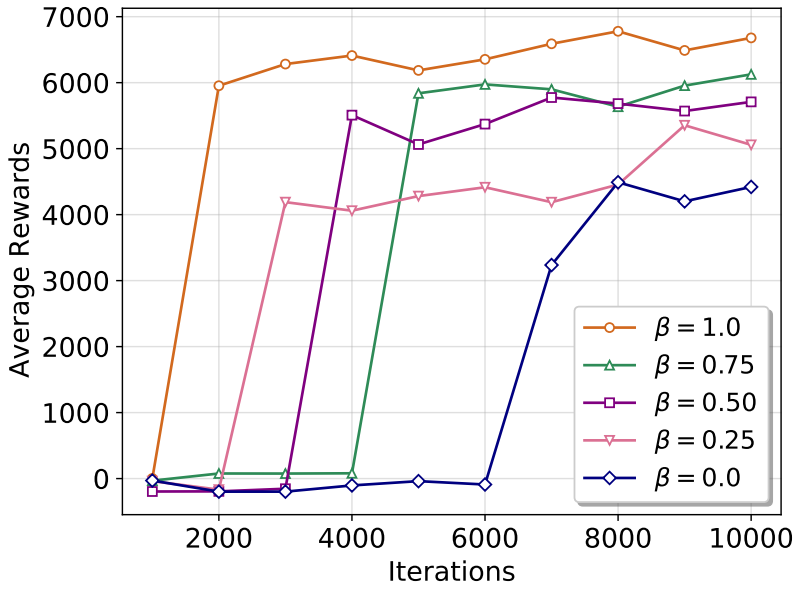


(a) Normal Job Arrival

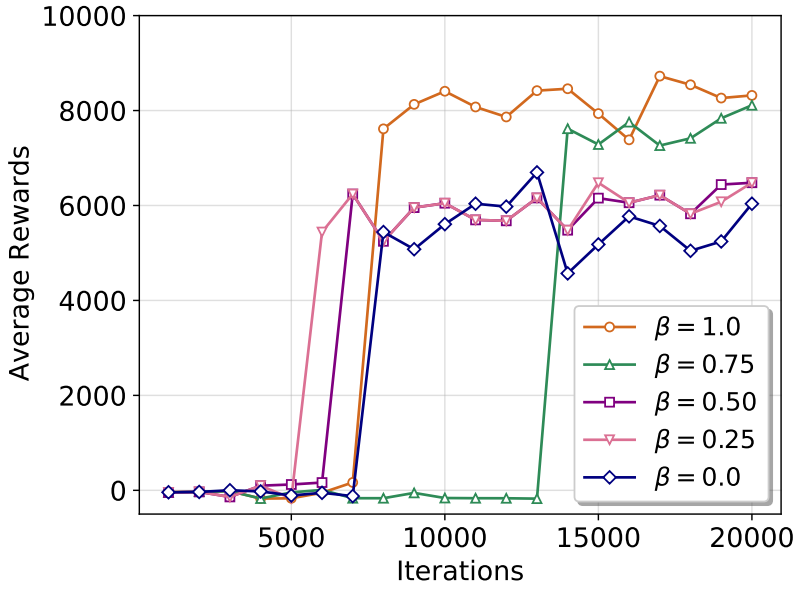


(b) Burst Job Arrival

Figure 4: Convergence of the DQN algorithm.



(a) Normal Job Arrival



(b) Burst Job Arrival

Figure 5: Convergence of the REINFORCE algorithm.

from 0 to 1, where value 1 indicates that the agent is optimized for cost only. Thus the reward for optimizing average job duration is ignored in the episodic reward. In contrast, a value of 0 of β indicates that the agent is optimized for reducing average job duration only. Any value of β excluding 0 and 1 indicates a mix-mode of operation, where an agent tries to optimize both rewards with different priorities (for values 0.25 and 0.75) or with the same priority (for the value of 0.50). Note that, the episodic rewards can vary and are calculated based on the cluster resource state, job specifications and arrival rates. Additionally, the final episodic reward varies between different optimization targets, so various training settings result in distinctive maximal rewards for an episode.

Fig. 4a and 4b represent average rewards accumulated by the DQN agent in training for the normal and burst job arrival patterns, respectively. Similarly, Fig. 5a and 5b represent average reward accumulation in training iterations by the REINFORCE agent. Note that, the average rewards are made up with both fixed rewards received for each successive executor placement and the final episodic reward, and is not the same as the actual VM usage cost or average job duration values. However, accumulating higher total reward implies the agent has learned a better policy which can optimize the actual objectives. Initially, both agents receive negative rewards and gradually start receiving more rewards after exploring the state-space over multiple iterations. Due to the randomness induced by the ϵ-greedy, sometimes the rewards drop for both algorithms. However, the training of REINFORCE agent is more stable than the DQN agent. Both agents required more time to converge with the workload with burst job arrival pattern as there are more jobs, and the agents have to learn to wait (action 0) when the cluster does not have sufficient resources to accommodate the resource requirements of a burst of jobs.

7.3. Learning Resource Constraints

It can be also observed from both Fig. 4 and Fig. 5 that the training environment works properly to train the agent to avoid bad actions such as violating resource capacity and demand constraints with the use of huge negative

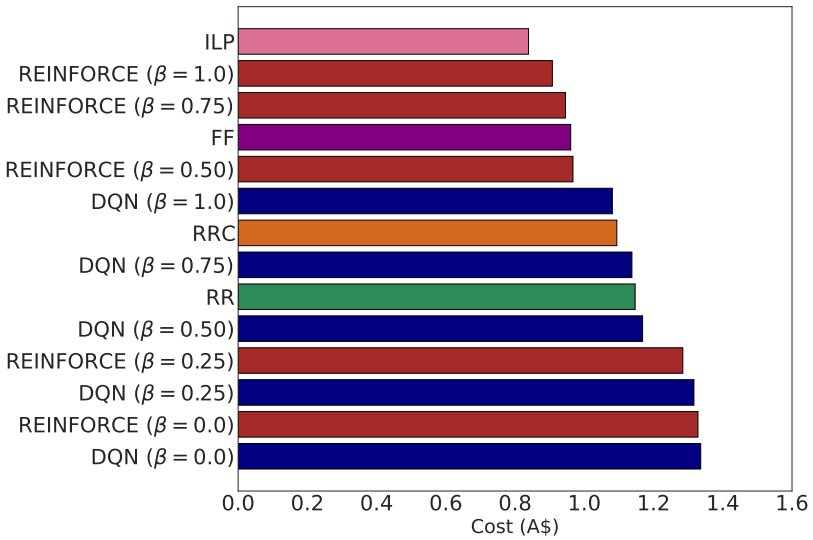
rewards. Therefore, at the start of the training process, both the algorithms incur huge negative rewards. However, after taking some good actions (executor placements while satisfying the constraints), the environment awards small immediate rewards, which motivates the agents to eventually complete the episode by scheduling all the jobs successfully. After the agents learn to schedule properly without violating the resource constraints, it can start learning to optimize the target objectives as it can observe different episodic reward depending on all the actions taken over a whole episode.

7.4. Evaluation of Cost-efficiency

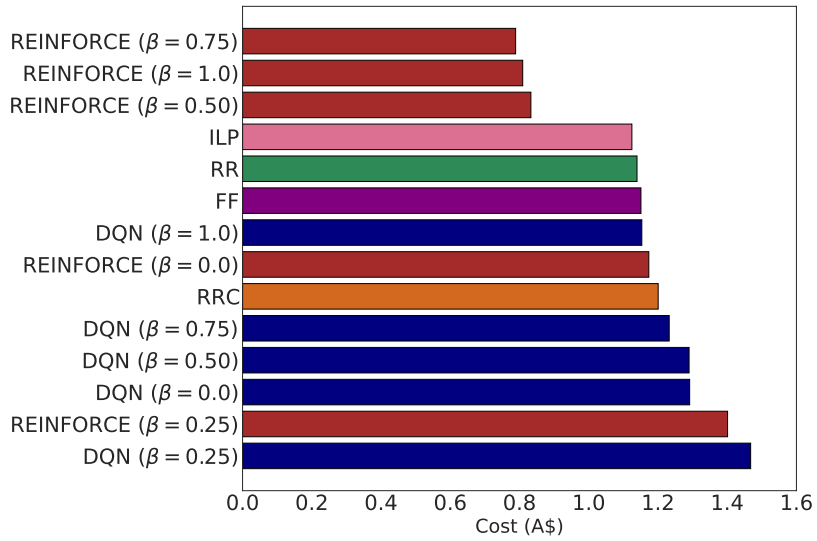
We evaluate the proposed DRL-based agents and the baseline scheduling algorithms regarding VM usage cost over a whole scheduling episode. In particular, we calculate the total usage time of each VM in the cluster and find the total cost of using the cluster. Fig. 6a exhibits the comparison of the scheduling algorithm while minimizing the VM usage cost with a normal job arrival pattern. As the job arrivals are sparse, the algorithms significant job duration increase due to improper placements. Therefore, tight packing on fewer VMs results in lower VM usage cost. The ILP algorithm outperforms all the other algorithms and incurs the lowest VM usage cost (0.84$), as it utilizes the job completion time estimates to find the cost-optimal placements of executors. Both REINFORCE (β=1.0, cost-optimized), and REINFORCE (β=0.75) performs closely to the ILP algorithm, and incur 0.91$ and 0.95$, respectively. Therefore, these agents have only a slight increase in cost from the ILP algorithm, which is 8% and 12%, respectively.

7.5. Evaluation of Average Job Duration

For the burst job arrival pattern, often there are not enough cluster resources to schedule all the jobs, so the scheduling algorithms have to wait until resources are freed up by the already running jobs. In addition, as the job arrival is dense, there can be a lot of job duration increase due to the bad placements for a particular type of job. For example, if a network-bound job is scheduled across



(a) Normal Job Arrival



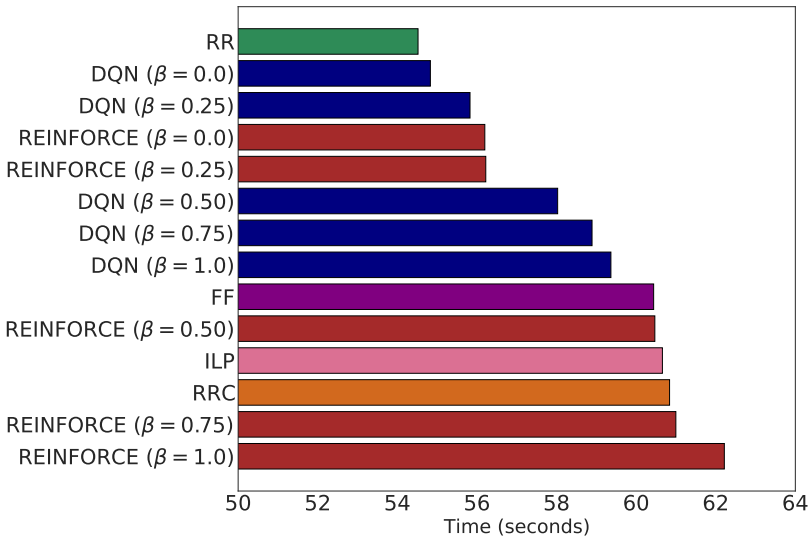
(b) Burst Job Arrival

Figure 6: Comparison of the total VM usage cost incurred by different scheduling algorithms in a scheduling episode.

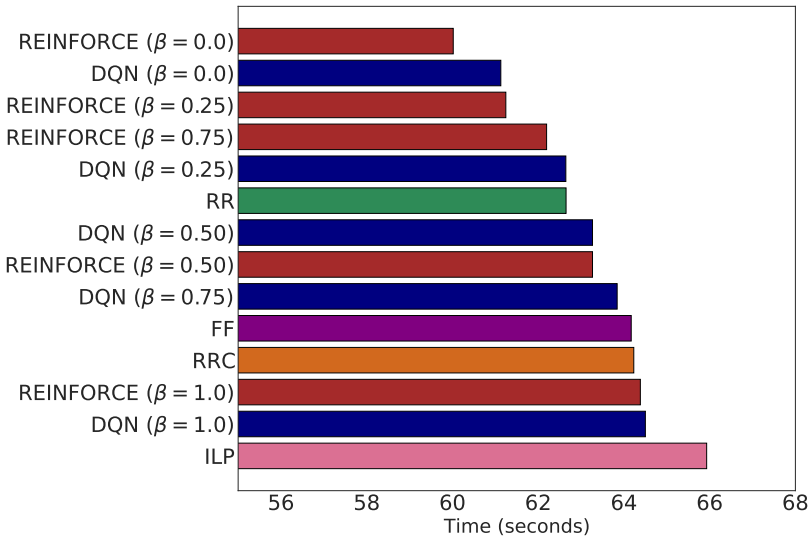
multiple VMs, the job run-time will increase, which might lead to an increasing VM usage cost. Although the ILP algorithm utilizes job completion time estimates, it still suffers from job duration increase if the job placement is not matched with the job characteristics, which is reflected in Fig. 6b. The REINFORCE agents achieve a significant cost-benefit, where three REINFORCE agents (β values of 0.75, 1.00 and 0.50) incur only 0.78\$, 0.81\$, and 0.83\$, respectively. In comparison, the best baseline ILP incurs 1.12\$, which is 30% more than the best REINFORCE agent ($\beta = 0.75$). Although surprising, the REINFORCE agent with $\beta = 0.75$ optimizes VM costs better than the $\beta = 1.0$ version, because for a burst job arrival, taking both objectives into consideration trains a better policy which in the long-run can optimize cost more effectively. The RR algorithm does not perform well because it cares only about distributing the executors, which results in higher VM usage cost. Although both FF and RRC algorithms try to minimize VM usages, for network-bound jobs, restricting to use only a few VMs can instead increase the cost due to the job duration increase. The DQN agents show a mediocre performance while minimizing VM usage cost, as the trained policy is not as good as the REINFORCE to learn the underlying job characteristics to minimize VM usage time.

We calculate the average job duration for all the jobs scheduled in an episode to compare the performance of the scheduling algorithms. Fig. 7a shows the comparison between the scheduling algorithms while reducing the average job duration. For the normal job arrival pattern, the RR algorithm performs the best as it cares only about distributing the jobs among multiple VMs. As there are more memory-bound and CPU-bound jobs combined than the network-bound jobs, the RR algorithm does not acquire significant job duration penalties due to distributed placement of network-bound jobs. RR algorithm is closely followed by the time-optimized versions of the DQN (β=0.0 and β=0.25) and the REINFORCE (β=0.0 and β=0.25) agents, respectively. The DQN (β=0.0) only increases the average job duration by 1%, whereas the REINFORCE (β=0.25) increases the average job duration by 4% when compared with the RR algorithm.

For the burst job arrival pattern, jobs often have to wait before more re-

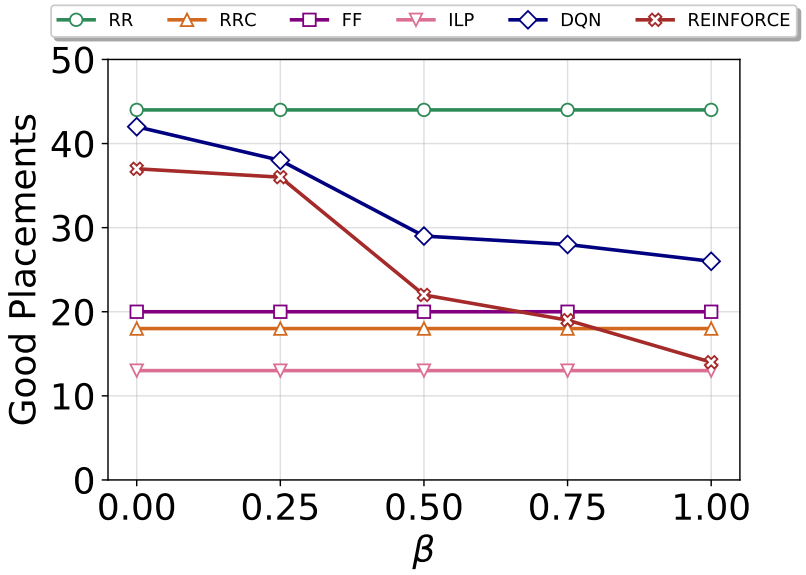


(a) Normal Job Arrival

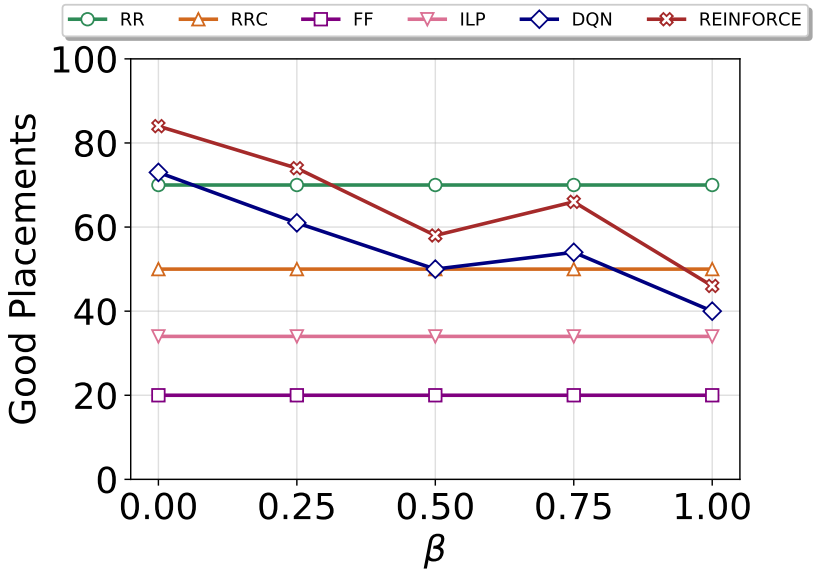


(b) Burst Job Arrival

Figure 7: Comparison between the scheduling algorithms regarding the average job duration in a scheduling episode.



(a) Normal Job Arrival

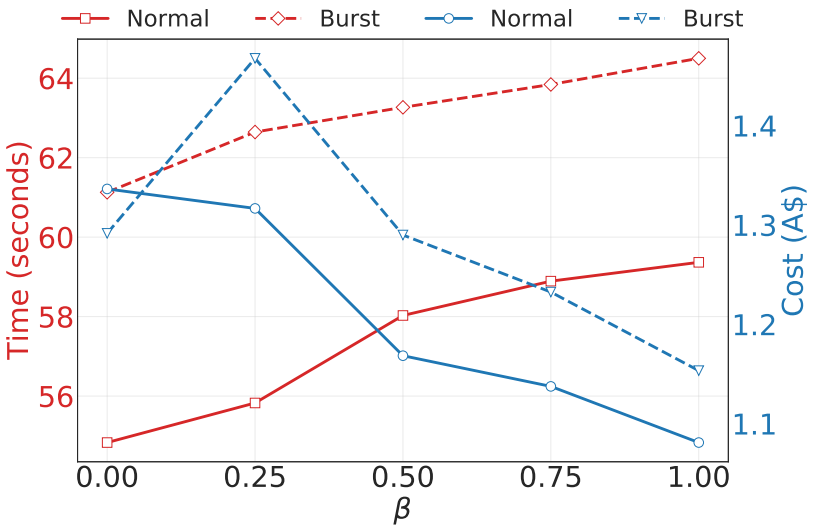


(b) Burst Job Arrival

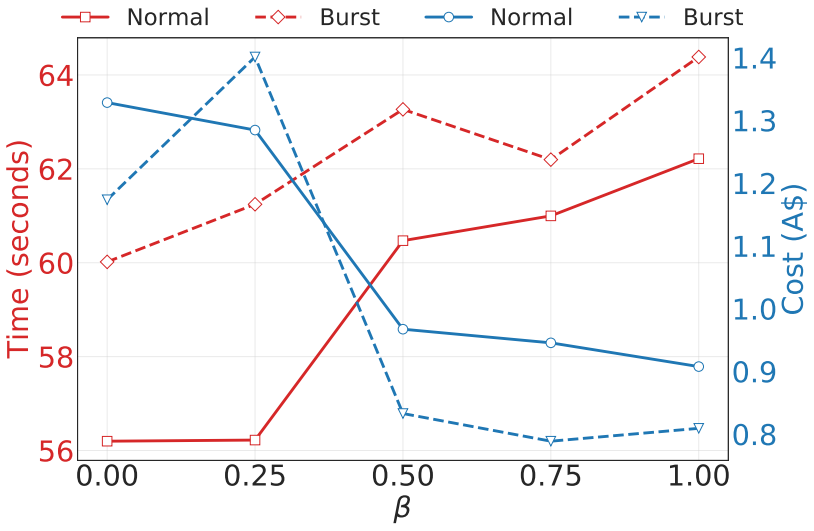
Figure 8: Comparison of good placement decisions made by each scheduling algorithm in a scheduling episode.

sources are available. In addition, if the job placement is not matched with the job characteristics, the completion time of the jobs will increase, which results in a higher average job duration. In this scenario, our proposed REINFORCE and DQN agents can capture the underlying relationship between job duration and job placement goodness and incorporates this information as a strategy to reduce job duration in the trained policy. As shown in Fig. 7b, REINFORCE (β=0.0) outperforms the best among the baseline algorithms RR and reduces the average job duration by 2%.

The underlying job characteristics reflect the ideal placement for a particular type of job. In addition, it impacts both the cost-minimization and job duration reduction objectives. Therefore, we also measured the number of good placements by each algorithm. Fig. 8a and 8b represents the good placement decisions made by all the algorithms in normal and burst job arrival patterns, respectively. As the baseline scheduling algorithms operate on a fixed objective and can not capture workload characteristics, the number of good placement decisions are fixed for each of the baseline algorithms. Therefore, the β parameter does not affect these algorithms, so the results from these algorithms appear as horizontal lines. However, both DQN and REINFORCE agents can be tuned to be cost-optimized, time-optimized or a mix of both. There is a decreasing trend in the number of good placements seen for both agents while the β parameter is increased (while moving towards cost-optimized from time-optimized version). The performance of DQN and REINFORCE discussed for average job duration reduction can be explained from these graphs. It can be observed that the DQN agent makes more good placements than the REINFORCE agent for the normal job arrival pattern, which results in lower average job duration for the DQN agents. In contrast, the REINFORCE agent makes more good placement decisions for the burst job arrival pattern, thus reducing the average job duration better than the DQN agent.



(a) DQN Agent



(b) REINFORCE Agent

Figure 9: The effects of the β parameter while using a multi-objective episodic reward in the RL environment. β=0.0 means time optimized only. β=1.0 means cost optimized only. Rest of the values represent a mix mode where both rewards have shared priority.

7.6. Evaluation of Multiple Reward Maximization

Fig. 9a and 9b exhibits the effects of the β parameter while optimizing multiple rewards. The solid lines represent the normal job arrival pattern, whereas the dashed lines represent the burst job arrival pattern. In addition, if a line is in blue colour, it represents the effect on time, whereas a red line reflects the effect on cost. It can be observed that both DQN and REINFORCE agents show stable results while maximizing one or more objectives. With the increase of β value, the agents are trained more towards optimizing the cost instead of the time reduction. While multiple rewards need to be optimized, the β parameter can be tuned to train the agents to learn a balanced policy which prioritizes both objectives. For example, in Fig. 9a, the solid blue and red lines at β=0.50 represents a DQN agent which provides a balanced outcome while optimizing both cost and time.

7.7. Learned Strategies

Here, we summarize the different strategies learned by the agents:

1. The DRL agents learn the VM capacity and job demand constraints through the negative reward from the environment when taking bad actions such as constraint violation or partial executor placements.
2. The DRL agents learn to optimize cost by packing executors in fewer VMs. However, depending on the job characteristics, they also learn to spread out executors to avoid job duration increase, which in turn results in better cost and time rewards (as showcased in the placement goodness evaluation graphs).
3. The agents can learn to handle both normal or burst job arrival patterns. The DRL agents decide to wait by choosing action 0 continuously when no cluster resources are available to run any more jobs. Although the agents incur a negative immediate reward (-1) while waiting to schedule, they choose it to avoid invalid or partial executor placement, which will lead to high negative rewards.

4. The agents can also learn a stable policy which balances multiple rewards (as shown from the β parameter tuning graphs).

8. Conclusions and Future Work

Job scheduling for big data applications in the cloud environment is a challenging problem due to the many inherent VM and workload characteristics. Traditional framework schedulers, LP-based optimization, and heuristic-based approaches mainly focus on a particular objective and can not be generalized to optimize multiple objectives while capturing or learning the underlying resource or workload characteristics. In this paper, we have introduced an RL model for the problem of Spark job scheduling in the cloud environment. We have developed a prototype RL environment in TF-agents which can be utilized to train DRL-based agents to optimize one or multiple objectives. In addition, we have used our prototype RL environment to train two DRL-based agents, namely DQN and REINFORCE. We have designed sophisticated reward signals which help the DRL agents to learn resource constraints, job performance variability, and cluster VM usage cost. The agents can learn to optimize the target objectives without any prior information about the jobs or the cluster, but only from observing the immediate and episodic rewards while interacting with the cluster environment. We have shown that our proposed agents can outperform the baseline algorithms while optimizing both cost and time objectives, and also showcase a balanced performance while optimizing both targets. We have also discussed some key strategies discovered by the DRL agents for effective reward maximization.

Currently, we have not included the co-location goodness of different jobs which may affect the job duration further. In addition, we have not included the implications of turning the VMs on or off. However, we plan to incorporate these features with a more sophisticated reward design in the future. In addition, we want to explore how the agents behave in the actual environment with variable cluster dynamics.

References

[1] V. K. Vavilapalli, A. C. Murthy, C. Douglas, S. Agarwal, M. Konar, R. Evans, T. Graves, J. Lowe, H. Shah, S. Seth, et al., Apache hadoop yarn: Yet another resource negotiator, in: Proceedings of the 4th ACM Annual Symposium on Cloud Computing, 2013.

[2] M. Zaharia, R. S. Xin, P. Wendell, T. Das, M. Armbrust, A. Dave, X. Meng, J. Rosen, S. Venkataraman, M. J. Franklin, A. Ghodsi, J. Gonzalez, S. Shenker, I. Stoica, Apache spark: A unified engine for big data processing, Communications of the ACM 59 (11) (2016) 56–65.

[3] K. Shvachko, H. Kuang, S. Radia, R. Chansler, The hadoop distributed file system, in: Proceedings of the 26th IEEE Symposium on Mass Storage Systems and Technologies (MSST), 2010.

[4] L. George, HBase: the definitive guide: random access to your planet-size data, O'Reilly Media, Inc., 2011.

[5] A. Lakshman, P. Malik, Cassandra: a decentralized structured storage system, ACM SIGOPS Operating Systems Review 44 (2) (2010).

[6] M. Zaharia, M. Chowdhury, T. Das, A. Dave, J. Ma, M. McCauley, M. J. Franklin, S. Shenker, I. Stoica, Resilient distributed datasets: A fault-tolerant abstraction for in-memory cluster computing, in: Proceedings of the 9th USENIX Conference on Networked Systems Design and Implementation (NSDI), 2012.

[7] K. Ousterhout, P. Wendell, M. Zaharia, I. Stoica, Sparrow, in: Proceedings of the Twenty-Fourth ACM Symposium on Operating Systems Principles - SOSP '13, ACM Press, New York, New York, USA, 2013, pp. 69–84.

[8] C. Delimitrou, C. Kozyrakis, Quasar: Resource-efficient and qos-aware cluster management, in: Proceedings of the 19th International Conference on Architectural support for programming languages and operating systems (ASPLOS), 2014.

[9] S. A. Jyothi, C. Curino, I. Menache, S. M. Narayanamurthy, A. Tumanov, J. Yaniv, R. Mavlyutov, I. n. Goiri, S. Krishnan, J. Kulkarni, S. Rao, Morpheus: Towards automated slos for enterprise clusters, in: Proceedings of the 12th USENIX Conference on Operating Systems Design and Implementation (OSDI), 2016.

[10] S. Dimopoulos, C. Krintz, R. Wolski, Justice: A deadline-aware, fair-share resource allocator for implementing multi-analytics, in: Proceedings of the IEEE International Conference on Cluster Computing (CLUSTER), 2017.

[11] S. Sidhanta, W. Golab, S. Mukhopadhyay, Optex: A deadline-aware cost optimization model for spark, in: Proceedings of the 16th IEEE/ACM International Symposium on Cluster, Cloud and Grid Computing (CCGrid), 2016.

[12] S. Maroulis, N. Zacheilas, V. Kalogeraki, A Framework for Efficient Energy Scheduling of Spark Workloads, in Proceedings of the International Conference on Distributed Computing Systems (ICDCS) (2017).

[13] H. Li, H. Wang, S. Fang, Y. Zou, W. Tian, An energy-aware scheduling algorithm for big data applications in Spark, Cluster Computing Journal (2019).

[14] H. Mao, M. Alizadeh, I. Menache, S. Kandula, Resource Management with Deep Reinforcement Learning, in: Proceedings of the 15th ACM Workshop on Hot Topics in Networks - HotNets '16, ACM Press, New York, New York, USA, 2016, pp. 50–56.

[15] H. Mao, M. Schwarzkopf, S. B. Venkatakrishnan, Z. Meng, M. Alizadeh, Learning scheduling algorithms for data processing clusters, SIGCOMM 2019 - Proceedings of the 2019 Conference of the ACM Special Interest Group on Data Communication (2019) 270–288.

[16] M. Assuncao, A. Costanzo, R. Buyya, A cost-benefit analysis of using cloud

computing to extend the capacity of clusters, Cluster Computing 13 (2010) 335–347.

[17] G. Rjoub, J. Bentahar, O. Abdel Wahab, A. Bataineh, Deep smart scheduling: A deep learning approach for automated big data scheduling over the cloud, Proceedings - 2019 International Conference on Future Internet of Things and Cloud, FiCloud 2019 (2019) 189–196.

[18] Y. Cheng, G. Xu, A Novel Task Provisioning Approach Fusing Reinforcement Learning for Big Data, IEEE Access 7 (2019) 143699–143709.

[19] L. Thamsen, J. Beilharz, V. T. Tran, S. Nedelkoski, O. Kao, Mary, Hugo, and Hugo*: Learning to schedule distributed data-parallel processing jobs on shared clusters, Concurrency Computation (March) (2020) 1–12.

[20] D. Silver, J. Schrittwieser, K. Simonyan, I. Antonoglou, A. Huang, A. Guez, T. Hubert, L. Baker, M. Lai, A. Bolton, Y. Chen, T. Lillicrap, F. Hui, L. Sifre, G. Driessche, T. Graepel, D. Hassabis, Mastering the game of go without human knowledge, Nature 550 (2017) 354–359.

[21] A. Ghodsi, M. Zaharia, B. Hindman, A. Konwinski, S. Shenker, I. Stoica, Dominant resource fairness: Fair allocation of multiple resource types, in: Proceedings of the 8th USENIX Conference on Networked Systems Design and Implementation (NSDI), 2011.

[22] N. Liu, Z. Li, J. Xu, Z. Xu, S. Lin, Q. Qiu, J. Tang, Y. Wang, A Hierarchical Framework of Cloud Resource Allocation and Power Management Using Deep Reinforcement Learning, in: 2017 IEEE 37th International Conference on Distributed Computing Systems (ICDCS), IEEE, 2017, pp. 372–382.

[23] Y. Wei, L. Pan, S. Liu, L. Wu, X. Meng, DRL-Scheduling: An Intelligent QoS-Aware Job Scheduling Framework for Applications in Clouds, IEEE Access 6 (2018) 55112–55125.

[24] T. Li, Z. Xu, J. Tang, Y. Wang, Model-free control for distributed stream data processing using deep reinforcement learning, Proceedings of the VLDB Endowment 11 (6) (2018) 705–718.

[25] Y. Bao, Y. Peng, C. Wu, Deep Learning-based Job Placement in Distributed Machine Learning Clusters, Proceedings - IEEE INFOCOM 2019-April (2019) 505–513.

[26] Z. Hu, J. Tu, B. Li, Spear: Optimized dependency-aware task scheduling with deep reinforcement learning, Proceedings - International Conference on Distributed Computing Systems 2019-July (2019) 2037–2046.

[27] C. Wu, G. Xu, Y. Ding, J. Zhao, Explore Deep Neural Network and Reinforcement Learning to Large-scale Tasks Processing in Big Data, International Journal of Pattern Recognition and Artificial Intelligence 33 (13) (2019) 1–29.

[28] A. M. Maia, Y. Ghamri-Doudane, D. Vieira, M. F. de Castro, Optimized placement of scalable iot services in edge computing, in: 2019 IFIP/IEEE Symposium on Integrated Network and Service Management (IM), 2019, pp. 189–197.

[29] S. Burer, A. N. Letchford, Non-convex mixed-integer nonlinear programming: A survey, Surveys in Operations Research and Management Science 17 (2) (2012) 97 – 106.

[30] X. Wang, J. Wang, X. Wang, X. Chen, Energy and delay tradeoff for application offloading in mobile cloud computing, IEEE Systems Journal 11 (2) (2017) 858–867.

[31] V. Mnih, K. Kavukcuoglu, D. Silver, A. Graves, I. Antonoglou, D. Wierstra, M. Riedmiller, Playing atari with deep reinforcement learning (2013). arXiv:1312.5602.

[32] 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, D. Hassabis, Human-level control through deep reinforcement learning, Nature 518 (7540) (2015) 529–533.

[33] R. J. Williams, Simple statistical gradient-following algorithms for connectionist reinforcement learning, Machine Learning 8 (3-4) (1992) 229–256.

[34] L. Wang, J. Zhan, C. Luo, Y. Zhu, Q. Yang, Y. He, W. Gao, Z. Jia, Y. Shi, S. Zhang, et al., Bigdatabench: A big data benchmark suite from internet services, in: Proceedings of the 20th IEEE International Symposium on High Performance Computer Architecture (HPCA), 2014.