Dejavu: Reinforcement Learning-based Cloud Scheduling with Demonstration and Competition

***Abstract*—As Cloud's adoption surges across industries, the limitations of its default scheduler, particularly on large scales or for jobs outside of its initial design scope, have become increasingly prominent. With the expansion of cloud usage, the industry is facing increased demands for integrating diverse cloud architectures. However, the default schedulers in various cloud services were primarily engineered with a focus on predictable tasks that exhibit minimal variance. Despite this need, clear and adaptable strategies to navigate these complex scenarios remain elusive, largely due to inherent design challenges.**

To address these issues, this paper presents Dejavu. This novel approach combines reinforcement learning with neural networks to learn and resolve scheduling problems more effectively. To tackle the extended training time associated with reinforcement learning, we've applied pretraining using Demonstrations from existing heuristics, thereby improving training efficiency in our cloud scheduling solution. This process prepares the neural network for subsequent reinforcement learning. A robust reward function is devised to push Dejavu to compete with, and eventually outperform, the exploited heuristics and other existing algorithms. The experimental results demonstrate the efficacy of Dejavu, showing remarkable improvements in key metrics. Specifically, compared to the default scheduler, it boosts resource utilization by 6% and shortens scheduling time by 3% during the scheduling period. Thus, it represents a significant leap forward in cloud scheduling, offering improved efficiency and versatility.

***Index Terms*—Container-based cloud, Scheduling, Reinforcement learning, Offline RL**

I. Introduction

In the previous decade, there has been a significant surge in the integration of cloud services among mid-sized and large enterprises, marking one of the swiftest expansions of cost in IT departments. As documented in a comprehensive study by Gartner, end-user spending on public cloud services globally reached an impressive $597.3 billion in 2023, demonstrating a dramatic escalation from $260 billion in 2020. This remarkable growth trajectory in the cloud industry serves as an incentive for enterprises to transition their systems to the cloud for potential cost efficiency.

However, this move often incurs a variety of additional expenditures, primarily due to the maintenance of multiple cluster infrastructures. Large-scale corporations, including Samsung or Google, which run diverse workloads, face challenges associated with the independent management of multiple clusters for different purposes. The need for managing separate clusters, which may create scenarios of surplus demand in one cluster while others remain underutilized, leads to unnecessary costs. These organizations are aware of the potential benefits of cluster integration, yet most fail to achieve this primarily due to scheduling issues.

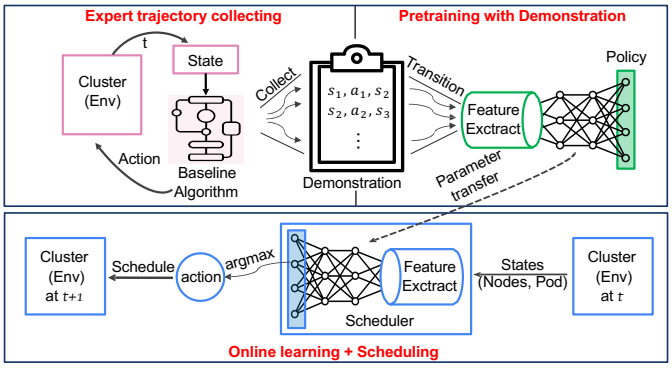


Fig. 1: The architecture of the proposed scheduling algorithm

Traditional heuristics applied on clusters are optimized for upcoming jobs, suggesting that these systems lack the flexibility to handle diverse workloads, potentially leading to physical or financial damage. Therefore, the efficient deployment of expensive compute clusters is a critical concern for enterprises. Even small improvements in resource utilization, such as reducing idle capacity and minimizing turnaround times, could yield substantial cost savings at scale. In this context, cluster schedulers emerge as vital tools for achieving such savings.

The challenge of optimal resource scheduling is typically classified as a combinatorial optimization problem, a category that includes numerous problems that are either NP-hard or NP-complete. Traditional approaches to these problems often involve approximation methods or heuristic strategies. Over time, empirical research has proposed various heuristics, such as first-fit, best-fit, and shortest-job-first (SJF), among others. However, the efficacy of these heuristic strategies is contingent on several factors, including the statistical patterns of resource demands and constraints on multiple resources. If the underlying scenario or the weight of importance shifts, a heuristic algorithm optimized for previous tasks may underperform.

Yet, the practical implementation of this ideal is fraught with difficulties, primarily due to the large-scale and intricate nature of resource management such as the non-stationary arrival and departure of jobs in cloud environments. Consequently, developing a dynamic, adaptable resource scheduler remains a formidable challenge in the field.

In response to the outlined challenges, this paper proposes Dejavu as illustrated in Fig. 1, tailored to accommodate various task scheduling and resource allocation scenarios in the container-based cloud environment. Our contributions through Dejavu are as follows:

- By pretraining the neural network, we significantly reduce

the training time required for scheduling. This expedited process not only enhances efficiency but also contributes to superior scheduling performance.
- In the process of creating a demonstration, rather than indiscriminately recording every timestep of each episode, we develop a mechanism to measure the similarity between experiences. This allows us to prevent consecutive learning from analogous experiences, thereby avoiding repetitiveness and redundancy.
- Dejavu employs a competitive reward function. This function, which is calculated based on the comparison with the kube-scheduler and existing heuristic algorithms, enables our scheduler to be finely tuned to outperform the baselines. This ensures its adaptability and efficacy in a variety of scheduling scenarios.
- Adding to the contributions, we also develop a simulator that faithfully virtualizes the characteristics of clouds. This advancement has significantly simplified the design of the reward function and the assessment of algorithm performance, further enhancing the overall efficiency of the system.

Through these innovations, Dejavu seeks to address the existing issues in cloud scheduling, paving the way for more efficient and versatile cloud computing.

The rest of the current paper is organized as follows: In Section II, we provide an overview of key concepts and knowledge related to our research. Our proposed methodology is described in Section III. including the problem formulation, building framework, and algorithm design. Experimental evaluation and result analysis are carried out in Section IV. We survey the literature in Section V. with the following discussion in Section VI. Finally, conclusions and some prospects are given in Section VII.

II. Preliminaries

A. Reinforcement learning

Reinforcement learning (RL) has garnered significant attention due to its ability to learn from interactions with an environment without relying on pre-existing labeled datasets. It enables an agent to learn optimal actions through trial and error, making it suitable for domains where explicit instructions or expert knowledge may be limited or unavailable. It models a problem as a Markov Decision Process (MDP), in which an agent decides an action based on observed state changes and receives rewards. Based on it, the objective of this agent is to maximize cumulative reward over time (i.e., Expected return). Through exploration and exploitation, the agent interacts with the environment to learn the optimal policy.

However, it often suffers from the dilemma that exploration can harm the system as it makes wrong decisions. Therefore, many RL-based systems utilize a model that was trained offline but in some applications, exploration during runtime is inevitable since it is hard to guarantee the stationary from the previous environment.

B. Speed up reinforcement learning with demonstration

The ultimate goal of RL is to train the model to predict values that closely align with the outputs of the defined reward function. One of the primary challenges of RL is the extensive time required for training as it requires tons of samples. The model must undergo numerous iterations, making mistakes and experiencing failures, before it can achieve a satisfactory level of performance.

For example, by training the network with decisions made by an expert or relatively more optimized algorithm, convergence time can be reduced. This approach, known as offline RL, allows the model to learn from the experience of other heuristics, thereby accelerating the convergence of the training process. However, this approach suffers from a compounding error issue, which stems from slight alterations in its decision-making process, resulting in a cascading effect that could jeopardize the entire sequence of decisions. Alternatively, an offline reinforcement learning approach could be utilized, which essentially shares similarities with its online counterpart. Nevertheless, this method also grapples with problems related to distribution shifts caused by the discrepancies between collected demonstrations and real-world experiences. Although the problem persists, it is an active area of research with many potential solutions being explored.

Within the realm of offline RL strategies, we opted for the Deep Q-Learning from Demonstrations (DQfD) approach in [17]. DQfD was proposed by the research team at Google DeepMind as a means to address the inherent issues of offline RL, by simultaneously employing temporal difference (TD) error and supervised loss.

III. Proposed Methodology

This section offers a comprehensive introduction to the proposed model. Initially, we present the modeling of nodes and pods to accurately describe the scheduling environment. Subsequently, we construct the Dejavu scheduler by incorporating models of the container-based cloud environment, scheduling agents, possible actions, and reward functions.

A. Environment

We consider a cluster composed of $\boldsymbol{n}$ number of nodes in it. Each of these nodes is equipped with the capability to deploy individual pods within themselves, thereby facilitating efficient computational tasks and enabling flexible scheduling. In this paper, we assume that if any resource in a node becomes fully consumed by an incoming pod, the scheduling process is considered unsuccessful. In other words, if the node is unable to accommodate the additional resource requirement of a new pod, it signifies a failure of the scheduling decision.

In addition, we take into account $\boldsymbol{d}$ types of resources (e.g., CPU, memory). Each of these resource types plays a crucial role in enabling the smooth functioning of the nodes and, in turn, the cluster. The variation in resource types further adds complexity to our scheduling and resource allocation problems, emphasizing the necessity for an efficient and optimal scheduling algorithm.

In every time step, jobs that meet the designated arrival time are submitted to the cluster in an online manner. This encapsulates the real-world unpredictability and randomness of job arrivals, reflecting the dynamic nature of actual computational workloads. Each job is defined not just by its computational requirements, but also by its individual resource quota, effectively setting a limit on the resources it consumes.

Moreover, each job is characterized by a predefined but unknown duration to both cluster and scheduler, which adds an additional layer of complexity to the scheduling problem. Once a job is completed, it is promptly terminated and removed from its host node.

B. Design of a Markov Decision Process (MDP) model

State definition. First of all, we define the state in our RL-base system. The state space can be represented through a combination of the individual nodes' states within a cluster and the status of the pending pod.

$$State_t = [Cluster_t, Pod_t^{-1}] \tag{1}$$

$$Cluster_t = [Node_t^1, Node_t^2, ..., Node_t^n] \tag{2}$$

Starting with the cluster state, we consider a cluster composed of ***n*** nodes. The nodes' state at timestep ***t*** can be denoted as $Cluster_t$ which is the aggregation of each node's state. Then, we denote the state of i^{th} node at a given time ***t*** as $Node_t^i$.

$$Node_t^i = [CPU_t^i, Memory_t^i] \tag{3}$$

The individual node's state $Node_t^i$ encapsulates the available resources within itself from its total resource pool. For our experimentation, we utilized 2 key resources (CPU and memory) which can be collected by default.

$$Pod_t^{-1} = [CPU_t^{pod^{-1}}, Memory_t^{pod^{-1}}] \tag{4}$$

Also, we consider the earliest queued pod awaiting deployment within the Kubernetes cluster. We represent the resource quota of the pending pod at time ***t*** as Pod_t^{-1}. Here, the Pod_t^{-1} captures the resources needs of a pod.

$$State_t = [Cluster_t, Pod_t^{-1}] = [CPU_t^1, Memory_t^1, ..., CPU_t^n, Memory_t^n, CPU_t^{pod^{-1}}, Memory_t^{pod^{-1}}] \tag{5}$$

When aggregating all those individual node states and pods states at time ***t*** as $State_t$, we combined and represented them as a one-dimensional vector, the length of which corresponds to the number of nodes multiplied by the number of resources plus the number of resources for the pending pod resources.

Action definition. The scheduler has been engineered to make a single decision at each point in time for the pod that is queued first, or earliest in the queue. The primary purpose of this scheduling approach is to determine an optimal node for the deployment of this pod.

$$Action_t \in \{0, node_i | i = 1, 2, ..., n\} \tag{6}$$

Accordingly, the action as referred to in this context, is the decision taken by the scheduler to select the specific node where the pod will be deployed. Action **0** signifies inaction, which implies that no pod should be scheduled on any node.

Algorithm 1 Base Reward Algorithm

Require: S'_t ▷ Cluster's state in next step
Require: A_t ▷ Action taken at time t

procedure R(S'_t, A_t)

$Cpu'_t, Mem'_t \leftarrow S'_t$
$[Cpu_t^1, Cpu_t^2, ..., Cpu_t^n] \leftarrow Cpu'_t$
$[Mem_t^1, Mem_t^2, ..., Mem_t^n] \leftarrow Mem'_t$

$is\_failed \leftarrow$ If Schedule succeed
$is\_idle \leftarrow$ If did nothing while it was available

$Avg_{Cpu} = \frac{\sum_{i=1}^{n} Cpu_t^i}{n}$ ▷ n = # of nodes
$Avg_{Mem} = \frac{\sum_{i=1}^{n} Mem_t^i}{n}$

$Std_{Cpu} = \frac{\sum_{i=1}^{n} (Cpu_t^i - Avg_{Cpu})^2}{n}$
$Std_{Mem} = \frac{\sum_{i=1}^{n} (Cpu_t^i - Avg_{Mem})^2}{n}$

$RBD1 = \sqrt{((Std_{cpu})^2 + (Std_{mem})^2)}$

$RBD2 = abs(Cpu_t^{A_t} - Mem_t^{A_t})$

$$PWD = \begin{cases} -1, & \text{if } is\_failed, \\ -1, & \text{if } is\_idle, \\ 0, & \text{otherwise.} \end{cases}$$

$Reward \leftarrow \alpha \cdot RBD1_c + \beta \cdot RBD2_c + \gamma \cdot PWD$

return $Reward$
end procedure

Reward function. We designed the reward signal to guide the reinforcement learning agent towards optimal solutions for our primary objective: the minimization of scheduling and turnaround time, which has the potential to significantly enhance the scheduling performance. Specifically, we formulate equations to encapsulate the factors we deem critical to our model.

At first, we incorporate a penalty for incorrect decisions made by the scheduler. In a real-world scenario, the scheduler should avoid erroneous decisions such as attempting to deploy when there are no pending pods or scheduling on a node that is already at capacity or would exceed its capacity if a pending pod was scheduled. Although such issues can be mitigated through extra measures such as node filtering in the actual domain, we integrate this aspect into our reward mechanism to streamline the scheduler's architecture, thereby potentially reducing scheduling time.

Secondly, we consider the degree of resource balance across nodes. If schedulers consistently deploy to the same node or do so more frequently, it can result in the node being saturated and potentially experiencing a slowdown in performance due to the accumulation of pods, while other nodes remain relatively idle. To circumvent this, we introduce a factor to estimate the degree of imbalance in resource distribution across nodes. This factor is represented in the following Algorithm 1. This equation returns the standard deviation of each resource, taking into account the squared mean of them for normalization.

Lastly, we take into account the balance of resources within the scheduled node. The optimal schedule should strive to balance resource distribution within the scheduled node as

well. Given that the scheduling problem is NP-hard and decisions must be made in a greedy manner based on current information, it is imperative to balance resources as much as possible to accommodate future incoming pods. If this is not considered, the cluster may become unavailable while there are still many idle resources in the other nodes. By calculating the difference in the resource utilization ratio of the scheduled node, we can evaluate the quality of the decision.

Finally, we define the reward function as follows, in which α, β, and γ are the empirical values set according to the cluster or objectives to scale the factors.

Competitive reward. Building upon the designed reward factors presented earlier in Algorithm 1, we introduce the competitive reward function, an innovative and potentially more optimizing reward function, as described in Algorithm 2. This new reward function operates on the principle of competition with the baseline scheduler. Our objective is not only to outperform the existing heuristic algorithms but also to navigate the nuances of task scheduling, which might not be easily assessed based purely on absolute performance.

Unlike the previously discussed reward function, the competitive approach measures the agent's decisions against those made by the baseline scheduler under identical conditions. To implement this, we duplicate the environment's state and apply it to the baseline algorithm to retrieve its decision. When evaluating each action, we calculate the individual factors $RBD1$ and $RBD2$ for both the agent and the baseline scheduler. After that, we compute the difference.

The final step involves the addition of PWD to the cumulative differences. The inclusion of PWD serves as a preventative measure against incorrect decisions, ensuring a balance between competitive optimization and decision accuracy. Through this advanced reward mechanism, our model strives for superior performance while retaining an awareness of the decision-making trends of its competitors.

Algorithm 2 Competitive Reward Algorithm

Require: $f_{rbd1}, f_{rbd2}, f_{pwd}$ ▷ Reward factor functions
procedure R($env, \pi_{rl}, \pi_{base}$) ▷ π is policy

$S_t \leftarrow env.get\_state()$
$env2 \leftarrow env.copy()$

$a_{rl} \leftarrow \pi_{rl}(S_t)$ ▷ Retrieve actions
$a_{base} \leftarrow \pi_{base}(S_t)$

$S_{rl}^{t+1} \leftarrow env.step(a_{rl})$ ▷ Take one step
$S_{base}^{t+1} \leftarrow env2.step(a_{base})$

$RBD1_c \leftarrow abs(f_{rbd1}(S_{rl}^{t+1}) - f_{rbd1}(S_{base}^{t+1}))$
$RBD2_c \leftarrow abs(f_{rbd2}(S_{rl}^{t+1}) - f_{rbd2}(S_{base}^{t+1}))$

$PWD \leftarrow f_{pwd}(env)$ ▷ Check if RL does wrong

$Reward \leftarrow RBD1_c + RBD2_c + PWD$

return $Reward$
end procedure

C. Neural network design

We present the design of DQfD network, which is based on the Deep Q-Network (DQN) architecture. The DQN consists of a Q-network, which functions as a policy network for decision-making, and its duplicate, the Q-target-network. Both of these networks are comprised of fully connected layers. The network takes the aforementioned state as an input and outputs the predicted rewards for all possible actions. Then, the agent takes the action with the greatest estimated reward value and gets the feedback in the form of the next state and reward. Upon executing an action and receiving the corresponding reward, our model is capable of calculating the loss and subsequently performing an optimization step. This process enables the model to refine its predictions and improve its performance over time.

The Deep Q-learning from Demonstrations (DQfD) operates over the DQN network, integrating temporal difference (TD) updates with the supervised identification of the demonstrator's actions. The use of TD error and supervised loss optimizes the network for additional training. The supervised loss component allows the algorithm to mimic the demonstrator's actions, while the TD loss provides a framework for it to understand a consistent value function, enabling ongoing learning through reinforcement learning (RL).

D. Training algorithm

1) Pretraining with demonstration

Given that RL is designed to learn from a running environment, there is a risk of causing damage to the system or the environment during the training process if it affects the system while exploring. This may not be a significant issue in simulated environments such as Atari games, but it becomes a critical concern in real-world systems like cloud clusters. Alternatively, in our circumstances, the agent needs to learn within a live setting where its actions have tangible repercussions. This necessitates that the agent demonstrate strong online performance from the beginning of the training.

To expedite the learning process and establish a solid baseline, we propose leveraging the knowledge of existing scheduling algorithms. For instance, the kube-scheduler used in Kubernetes employs a simple greedy algorithm that scores nodes based on resource availability, and it performs well under typical conditions.

By utilizing the decisions made by the baseline algorithms for preliminary offline training, we can accelerate the RL model's learning process and quickly reach a level of performance that is comparable to existing solutions. This approach allows us to harness the existing algorithm's decisions, which provide the RL agent with appropriate search directions for optimal solutions, thereby enabling faster learning.

In order to gather demonstrations from the baseline algorithms, we operated a simulated cloud environment and allowed the baseline algorithm-based schedulers to interact with it. We collected sets of $[state_t, action_t]$ until the completion of each episode, which is defined as the point at which all pods in the scenario have been scheduled. Upon collecting all the data, we obtained a substantial volume of baseline demonstrations. In a practical application, demonstrations can

also be conveniently gathered from a running cluster. The decision trajectory (τ) collected from each episode can be denoted as follows:

$$\tau \in \{s_1, a_1, s_2, a_2, ..., s_{-1}, a_{-1}\} \tag{7}$$

where s_t represents the state and a_t is the action taken by the baseline at time t. Upon gathering the $[s_t, a_t]$ trajectory, we extract the state-action-next_action set from it to create a new dataset specifically for pretraining. This process allows us to structure the data in a way that is most beneficial for our reinforcement learning model. The demonstration dataset (D), thus formed, can be represented as follows:

$$D = \{(s_1, a_1, s_2), (s_2, a_2, s_3), ..., (s_{-1}, a_{-1}, s_{last}), \} \tag{8}$$

This method of pretraining with demonstration not only saves time but also ensures that our RL-based scheduler can be safely and effectively deployed in real-world environments.

In the initial pretraining phase, the agent selects mini-batches from the demonstration data to refine the network, employing four types of losses: 1-step double Q-learning loss, n-step double Q-learning loss, supervised large margin classification loss, and L2 regularization loss on the network's weights and biases. The supervised loss is employed for categorizing the demonstrator's actions, and the Q-learning loss ensures the network aligns with the Bellman equation and serves as a foundation for TD learning.

The importance of supervised loss in pretraining cannot be understated. Given the demonstration data inherently covers a narrow segment of the state space without encompassing all possible actions, there are many state-action combinations for which there is no grounding data to assign realistic values. If we were to simply pretrain the network with Q-learning updates directed towards the maximum value of the subsequent state, the network would gravitate towards the largest of these unanchored variables, spreading these values across the Q function. We, therefore, incorporate a large margin classification loss to prevent this, as in [18].

$$J_E(Q) = max_{a \in A}[Q(s, a) + l(a_E, a)] - Q(s, a_E) \tag{9}$$

By incorporating n-step returns (where n equals 10), we manage to extend the values from the expert's trajectory to all preceding states, thereby enhancing the pretraining process. The n-step return, represented by the follwoing equation:

$$r_t + \gamma r_{t+1} + ... + \gamma^{n-1} r_{t+n-1} + max_a \gamma^n Q(s_{t+n}, a) \tag{10}$$

which is computed using a forward-looking view, akin to the method used in A3C. [19].

An L2 regularization loss is further applied to the network's weights and biases, assisting in preventing overfitting on the somewhat limited demonstration dataset. The total loss employed to refine the network is a cumulative function of these four losses, represented by the following equation:

$$J(Q) = J_{DQ}(Q) + \lambda_1 J_n(Q) + \lambda_2 J_E(Q) + \lambda_3 J_{L2}(Q) \tag{11}$$

Algorithm 3 Selective demonstration collecting algorithm

Require: D_{prev}, D_{cand} ▷ Lastly queued Demo & candidate

procedure SELECT_DEMO($S_{prev}, S_{cand}, thres$)

  $S_{prev}, A_{prev} \leftarrow D_{prev}$ ▷ State and Action

  $S_{cand}, A_{cand} \leftarrow D_{cand}$

  $similarity = \frac{S_{prev} \cdot S_{cand}}{(||S_{prev}|| * ||S_{cand}||}$

  if $|similarity - 1| > thres$ **then**

    return $True$

  else

    if $A_{prev} != A_{cand}$ **then**

      return $True$

    else

      return $False$

    end if

  end if

end procedure ▷ Add demo if it returns True

2) Selective Demonstration collecting for better learning

We aim to prevent the inherent issue of compounding error and overfitting in offline learning. Our focus is to make the process more efficient by curating the demonstration data, and minimizing the redundancy and similarity between the collected trajectories. For this, we leverage cosine similarity in Algorithm 3, to sample demonstrations selectively. The trajectory generated by the baseline schedulers was treated as vectors. These vectors encapsulate an array of data, corresponding to each episode of the scheduling process. We calculate the cosine similarity between each incoming trajectory with its preceding one recorded. If the similarity is within a certain criterion (5% in our experiment), indicating a high degree of overlap, we refrained from recording the new trajectory.

On top of it, there is a caveat in this approach. We consider the case where two similar trajectories could result in different scheduling decisions. This implies that the two trajectories could potentially offer diverse learning experiences. By implementing this methodology, we aim to sample a more diverse and representative set of demonstrations for training the RL model. This strategy facilitated a more balanced learning process, reducing the impact of inherent drawbacks of the imitation learning process.

3) Reinforcement Learning

Reinforcement learning models are expected to handle a variety of states and consistently make optimal decisions. However, given that our baseline scheduler (the teacher) already performs optimally in general, the model has limited opportunities to experience failure or drastic situations. For instance, our pre-trained model may not know how to act in situations where it has made incorrect choices over a long period, resulting in a severe imbalance of resources across all or some nodes. Therefore, it may struggle to determine the appropriate course of action.

Throughout the training process, the agent continually collects experiences over multiple episodes and maintains an experienced pool, denoted as D, with a capacity of N. As the model navigates through different scenarios, it is updated with both explored and exploited experiences, gradually relying more on the latter. The Q-network outputs Q values, and losses are calculated to perform a gradient descent step, which

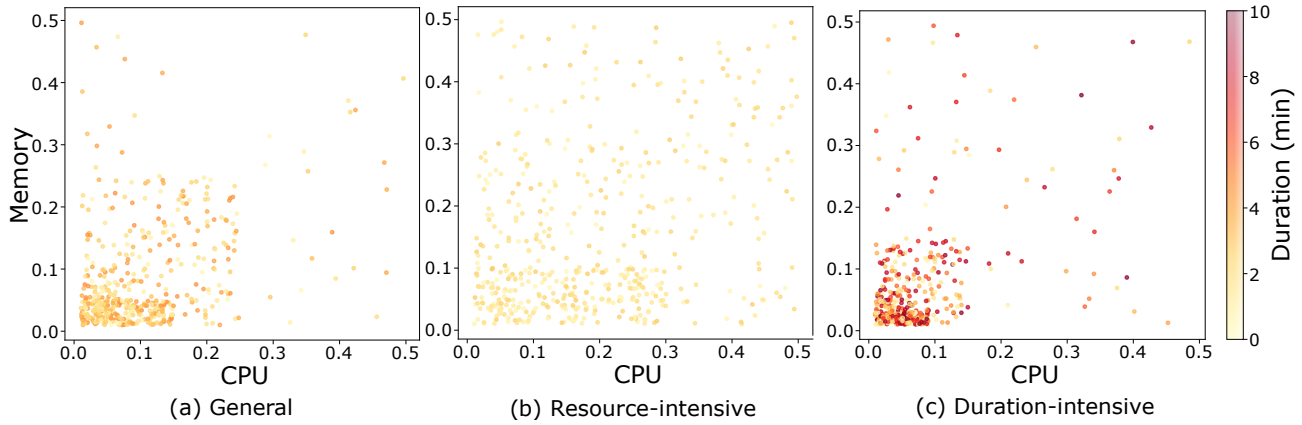


Fig. 2: Three types of workloads for experiments. Each dot in the figure represents a workload for a pod with its normalized memory and CPU requirements.

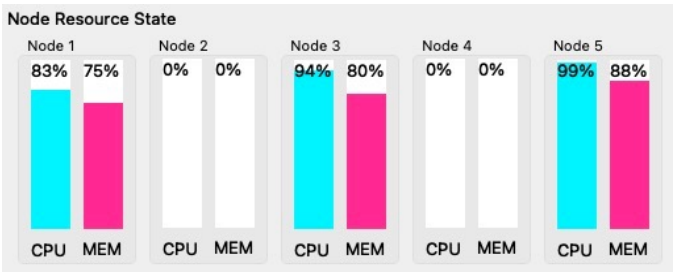


Fig. 3: Resource unbalanced across nodes

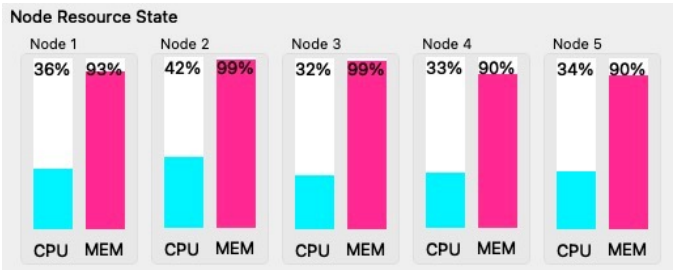


Fig. 4: Resource unbalanced within nodes

updates the network. After every K step, the Q-target-network copies the parameters of the Q-network for weight updates. This process continues until a predefined condition is met, such as a certain number of steps or episodes, or a specified level of rewards. In summary, our reinforcement learning phase builds upon the knowledge gained from pretraining with demonstrations, using a combination of exploration and exploitation to refine the model's decision-making capabilities. This approach ensures that our model is not only well-educated but also capable of adapting to new scenarios.

IV. Experimental Evaluation

We conduct training and testing of our algorithms using various configurations. This involves observing trends by modifying our final models, such as adjusting the selective demonstration similarity rate and the amount of demonstration used for training. Finally, we compare our model with representative configurations against other baseline algorithms.

A. Experimental Setup

Implementation. we developed a generalized simulated environment with a graphical user interface. This environment was not specific to Kubernetes, allowing us to test our model's performance and reward functions in a more general context. To facilitate the testing process, we implemented the OpenAI gym environment for both the real cluster and the simulated cluster. This provides a standardized framework for developing and comparing our RL models, thereby ensuring the robustness and reliability of our results.

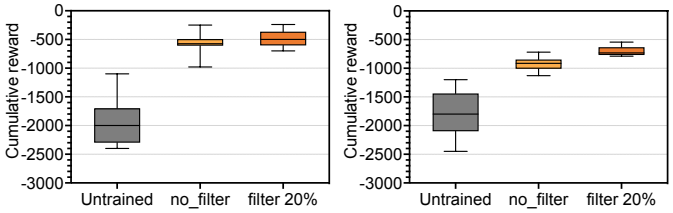
Workloads setup. To evaluate the performance of our proposed model, we utilize the Alibaba cluster trace v2018 [20], which includes data from approximately 4000 machines over a period of 8 days, amounting to 20GB traces. To diversify the challenges and thoroughly test our model, we generate different scenarios by sampling the trace data with varying objectives as shown in Fig. 2. These scenarios are as follows:

- *General workload*: Balanced mix of different tasks to represent the general situation.
- *Duration-intensive workload*: The workload is skewed toward tasks that require longer competition time, testing the model's ability to handle time-demanding tasks.
- *Resource-intensive workload*: Focus on tasks that require a high amount of resource quota, challenging the model's capacity to manage resource-intensive tasks efficiently.

B. Selective demonstration

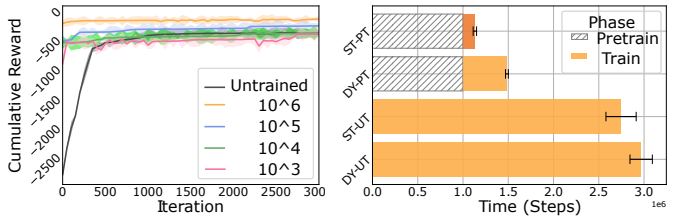
We conduct our models' initial performance assessment across various configurations of the similarity filter rates. The rate we employed was 5%. These rates indicate that if an incoming trajectory has a cosine similarity below the set threshold, the system will disregard it and not add the trajectory to demonstrations.

In Fig. 5, the initial performance results following pretraining are displayed. Out of all the tested models, the one



(a) Base rewards (b) Competitive rewards

Fig. 5: Comparison of rewards with different pretraining setup



(a) Learning curve (b) Convergence time

Fig. 6: Comparison of different training configurations

trained with 5% filtered demonstration produced superior performance. Conversely, the model trained without any filtered demonstration yielded the least impressive results among the pretrained models. However, it's worth noting that even this model outperforms those that were not pretrained at all.

The model that emerged as the most efficient in Fig. 5 - the one with 5% filtered demonstrations exhibited considerable improvements in key areas. It demonstrated a substantial 11% enhancement in initial performance compared to the model trained with unfiltered demonstration. Naturally, these pretrained models exhibit significantly superior performance when compared to the untrained DQN model.

C. Training convergence

We initially pre-train our model utilizing 10^6 demonstration instances, subsequently followed by additional training with 10^5, 10^4, and 10^3 demonstrations. Post this pretraining phase, we evaluate how the models perform when trained in the actual operational environment for 3 million timesteps. Fig. 6 illustrates the learning curve of our model (in orange), which is contrasted against varying configurations. The model trained with the maximum number of demonstrations significantly outstrips the performance of all other comparison subjects. From our observations, it is evident that our model consistently achieves the highest cumulative reward among all models. A clear correlation can be discerned between the volume of demonstrations used for training and the subsequent performance of the model. Moreover, we note that the initial gap in performance due to pretraining cannot be bridged throughout training for 3 million episodes. Upon making a rough assumption that each episode with an adequately performing scheduler takes approximately 1500 time steps at minimum, it can be concluded that an untrained model would not outperform for at least 2000 episodes. This extended duration of under-performance could lead to significant delays in a real-world computational cluster.

D. Comparison with baselines

Previous experiments have been carried out with the objective of substantiating that our final model is the most suitable option among the feasible choices for implementation. To further strengthen our confidence in the readiness of our model for practical use, we proceed to evaluate its performance from various perspectives. This also encompasses a comparison against baseline algorithms. For this comparative analysis, we employ four distinct heuristic algorithms, augmented with the Ant Colony Optimization (ACO) algorithm.

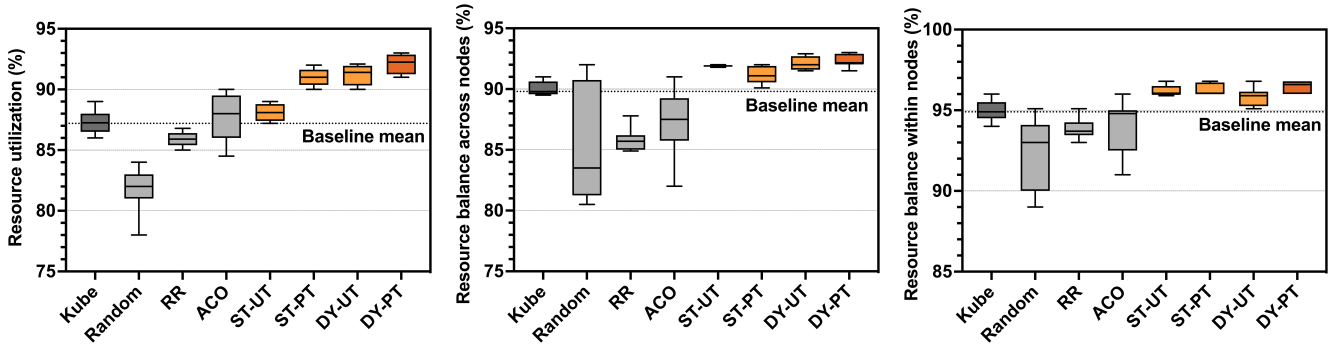
Maximizing resource utilization rate. In a scenario where a significant number of jobs are continually being queued to the cluster, maximizing the use of available resources becomes imperative. As part of our methodology, we calculate the average resource utilization rate for the entire cluster throughout each episode, continuing until the final pod is scheduled. This measurement enables us to assess the efficiency of the scheduling algorithm in managing its resources.

The resource utilization rate could be adversely affected by incorrect decisions, such as scheduling tasks to a node destined to fail due to availability issues. Furthermore, imbalanced resource allocation, as depicted in Fig. 4, could also negatively impact the rate. This is because the scheduler is left with no option but to wait until a fully engaged resource becomes idle while another resource may already be largely idle, thereby resulting in inefficient resource management.

Fig. 7 (a) presents the resultant distribution of the resource utilization rate. We extract the final 10 records from each run, which are anticipated to be optimized to the greatest extent. The model that utilizes pretraining and a competitive reward function, denoted as DY-PT, stands out with the most commendable performance among all, achieving a rate of over 92%. This figure is particularly noteworthy as it surpasses the baseline scheduler, used for pretraining, by more than 4%. In comparison to the model that operates with a base reward function in Algorithm 1, the pre-trained model exhibits a slightly superior performance, with an advantage of approximately 2%.

Balanced resource distribution. The equitable distribution of resources across and within nodes is vital for optimizing scheduling performance. This imbalance becomes particularly challenging when unexpected resource request ratios arise, such as a pod requiring 1% CPU but consuming 13% memory. Given the Alibaba trace, we tested was characterized by imbalances in general, it provides an appropriate setting for assessing the balancing capabilities of scheduling algorithms. As depicted in both Fig. 7 (b) and (c), our model significantly surpasses the performance of the baseline models. When compared to the Kube-scheduler, it demonstrates an average improvement of 2%. This result underscores the value of our pre-trained model, particularly its aptitude for resource balancing, even in the face of unpredictable job quotas.

Minimizing completion time. Contrary to the factors previously compared, completion time stands as the ultimate objective to be achieved by the scheduling algorithm. Our goal is to enhance the aforementioned factors to possibly reduce the



(a) Resource utilization rate (b) Resource balance across nodes (c) Resource balance within nodes

Fig. 7: Resource utilization rate, resource balance across nodes (RBD1) and within nodes (RBD2) with baselines.

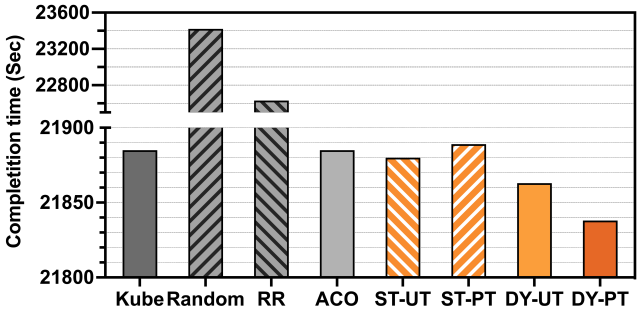


Fig. 8: Scheduling completion time

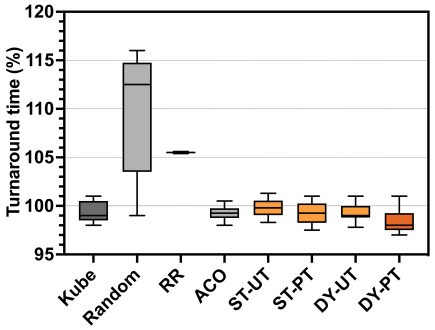


Fig. 9: Turnaround time

scheduling completion time. To facilitate comparison, we test a scenario involving the scheduling of 8,000 jobs for 80 minutes, derived from the Alibaba Cluster trace. As evidenced in Fig. 8, our model completes the assigned scenario workloads the fastest. Even the untrained model (DY-UT) surpasses the baseline models.

Minimizing turnaround time. Turnaround time, or the interval a pod must endure before its scheduled task, is crucial for efficiency. The reduction of this time is an objective of primary concern due to its significant impact on end-users experiences and decisions about migrating to different clusters. This time factor can be influenced by various elements, such as flawed decision-making and imbalanced resource scheduling, both of which can prolong the turnaround time. Fig. 9 illustrates the performance of our model in minimizing turnaround time. In comparison to the baseline model, our solution exhibits an impressive 2.4% reduction in the turnaround time. This compelling outcome underscores our model's effective functionality, despite not incorporating a reward factor that explicitly considers turnaround time. Instead, our approach is predicated on the utilization of PWD, a mechanism designed to circumvent incorrect decisions that could potentially introduce scheduling delays. The other two reward factors, $RBD1$, and $RBD2$, also ensure a comprehensive balancing of the cluster. The outcome, as depicted in our results, is a significant improvement in completion time and turnaround time, contributing to our proposed model's overall efficiency and efficacy.

V. Discussion

In this section, we delve into potential research areas for the future and various approaches that we could experiment with to develop a more efficient and resilient scheduler.

Considerations on various workloads. Our experimental results reveal that our proposed model is capable of learning universal scheduling policies that perform effectively on unseen workloads. This capability is achieved by capitalizing on pretraining and a competitive reward function, which could have potential applications across various cloud workloads. However, we must consider that workload could undergo more severe than just interarrival time shifts. Additionally, we foresee more complex and contradictory scheduling scenarios in real-world conditions, especially when integrating clouds with differing demands.

Different scheduling frameworks. In the current model, we used Kubernetes as a representative cluster for consideration. However, we have not explored more intricate scheduling cases like simultaneous or preemptive scheduling. Implementing such features in the Reinforcement Learning (RL) environments is feasible if the action space is defined appropriately. Utilizing these concepts could enable us to construct a more robust scheduler, one that proactively enhances scheduling performance. To improve the resilience of a scheduling policy against such changes, training the agent on worst-case scenarios or documented industry errors could be beneficial.

VI. Related work

Our research on RL-based cloud scheduling is situated within a broader context of cloud scheduling and ML applications. In this section, we review the relevant literature.

Non-RL based scheduling algorithms. Non-RL-based schedulers have been extensively studied. For instance, the Ant Colony algorithm has been applied to Kubernetes' resource scheduling scheme [1]. Tetris [2] proposed a multi-resource packing strategy for cluster schedulers. Also, [3] focused on specific aspects such as heterogeneity-aware cluster scheduling policies for deep learning workloads.

RL or ML-based scheduling algorithms. The application of RL and ML in cloud scheduling is promising. DeepRM [4] and RLSK [5] used deep reinforcement learning for resource management and job scheduling systems. [6] have proposed Kubernetes scheduling strategies based on LSTM and Grey model. Also, there was an approach to focus on automatic resource scaling of containers in fog clusters using reinforcement learning. [7] Other studies in [8] have proposed heuristic multi-objective task scheduling frameworks for container-based clouds. Some studies have proposed the use of deep learning for job placement in distributed machine learning clusters [9], for microservice resource allocation over scientific workflows [10], and for HPC scheduling [11].

Offline RL approaches. Focusing on the performance of the demonstrator, DAGGER [12] recurrently generates new policies by querying the expert policy outside its original state space, which has been proven to result in no regret over validation data in terms of online learning. Another common approach involves establishing a zero-sum game where the learner selects a policy and the adversary picks a reward function. [13], [14], [15]. Demonstrations have also been utilized for inverse optimal control in intricate, continuous robotic control issues [16]. However, these methods strictly adhere to imitation learning and do not facilitate learning from task rewards. DQfD [17] uses human-generated demonstrations for Atari games, implementing it during the offline RL phase with additional steps to mitigate the inherent issues of offline RL. The model can also undergo training after the pretraining phase, demonstrating solid capability from the outset.

In summary, our research builds upon these previous studies by proposing a novel RL-based cloud scheduler. Our approach aims to address some of the limitations identified in the existing literature and to contribute to the ongoing development of efficient and effective cloud scheduling strategies.

VII. Conclusion

In this paper, we proposed Dejavu, a novel approach that combines reinforcement learning with a neural network to effectively address scheduling problems. Applying pretraining using demonstrations from existing baselines, improves training efficiency and prepares the neural network for subsequent reinforcement learning. A robust reward function further pushes Dejavu to compete with, and eventually outperform, exploited heuristics and other existing algorithms. Experimental results validate the effectiveness of our model, showing significant improvements in resource utilization, scheduling time, and reduction of idle resources. Therefore, it signifies a substantial advancement in cloud scheduling, offering enhanced efficiency and versatility.

References

[1] Z. Wei-guo, M. Xi-lin, and Z. Jin-zhong, "Research on Kubernetes' Resource Scheduling Scheme," Proceedings of the 8th International Conference on Communication and Network Security (ICCNS) 2018.

[2] R. Grandl, G. Ananthanarayanan, S. Kandula,"Multi-resource packing for cluster schedulers," ACM SIGCOMM Computer Communication Review, 2014.

[3] D. Narayanan, K. Santhanam, A. Phanishayee, F. Kazhamiaka, and M. Zaharia, "Heterogeneity-Aware Cluster Scheduling Policies for Deep Learning Workloads," 14th USENIX Symposium on Operating Systems Design and Implementation(OSDI20), 2020.

[4] H. Mao, M. Alizadeh, I. Menache, and S. Kandula, "Resource Management with Deep Reinforcement Learning," Proceedings of the 15th ACM Workshop on Hot Topics in Networks - HotNets '16, 2016.

[5] J. Huang, C. Xiao, and W. Wu, "RLSK: A Job Scheduler for Federated Kubernetes Clusters based on Reinforcement Learning," IEEE International Conference on Cloud Engineering (IC2E), Apr. 2020.

[6] Yang, Ying, and Lijun Chen. "Design of kubernetes scheduling strategy based on LSTM and grey model." 2019 IEEE 14th International Conference on Intelligent Systems and Knowledge Engineering (ISKE). IEEE, 2019.

[7] Sami, Hani, et al. "Fscaler: Automatic resource scaling of containers in fog clusters using reinforcement learning." 2020 International wireless communications and mobile computing (IWCMC). IEEE, 2020.

[8] Zhu, Lilu, et al. "A heuristic multi-objective task scheduling framework for container-based clouds via actor-critic reinforcement learning." Neural Computing and Applications 35.13 (2023): 9687-9710.

[9] Bao, Yixin, Yanghua Peng, and Chuan Wu. "Deep learning-based job placement in distributed machine learning clusters." IEEE INFOCOM 2019-IEEE conference on computer communications. IEEE, 2019.

[10] Yang, Zhe, et al. "MIRAS: Model-based reinforcement learning for microservice resource allocation over scientific workflows." 2019 IEEE 39th international conference on distributed computing systems (ICDCS). IEEE, 2019.

[11] Fan, Yuping, et al. "Deep reinforcement agent for scheduling in HPC." 2021 IEEE International Parallel and Distributed Processing Symposium (IPDPS). IEEE, 2021.

[12] Ross, Stéphane, Geoffrey Gordon, and Drew Bagnell. "A reduction of imitation learning and structured prediction to no-regret online learning." Proceedings of the fourteenth international conference on artificial intelligence and statistics. JMLR Workshop and Conference Proceedings, 2011.

[13] Syed, Umar, and Robert E. Schapire. "A game-theoretic approach to apprenticeship learning." Advances in neural information processing systems 20 2007.

[14] Syed, Umar, Michael Bowling, and Robert E. Schapire. "Apprenticeship learning using linear programming." Proceedings of the 25th International conference on Machine learning. 2008.

[15] Ho, Jonathan, and Stefano Ermon. "Generative adversarial imitation learning." Advances in neural information processing systems 29 2016.

[16] Zhang, Marvin, et al. "Learning deep neural network policies with continuous memory states." 2016 IEEE international conference on robotics and automation (ICRA). IEEE, 2016.

[17] Hester, Todd, et al. "Deep q-learning from demonstrations." Proceedings of the AAAI Conference on Artificial Intelligence. Vol. 32. No. 1. 2018.

[18] Piot, Bilal, Matthieu Geist, and Olivier Pietquin. "Boosted bellman residual minimization handling expert demonstrations." Machine Learning and Knowledge Discovery in Databases: European Conference, ECML PKDD 2014, Nancy, France, September 15-19, 2014. Proceedings, Part II 14. Springer Berlin Heidelberg, 2014.

[19] Mnih, Volodymyr, et al. "Asynchronous methods for deep reinforcement learning." International conference on machine learning. PMLR, 2016.

[20] Guo, Jing, et al. "Who limits the resource efficiency of my datacenter: An analysis of alibaba datacenter traces." Proceedings of the international symposium on quality of service. 2019.