Task Scheduling for Control System Based on Deep Reinforcement Learning

Yuhao Liu[a], Yuqing Ni[a,*], Chang Dong[b], Jun Chen[a] and Fei Liu[a]

[a]Key Laboratory of Advanced Control for Light Industry Processes (Ministry of Education), Jiangnan University, Wuxi, 214122, China
[b]Durham University Business School, Durham University, Durham, UK

ARTICLE INFO

Keywords:
Task scheduling
Control system
Cloud server
LQR
BPP
Reinforcement learning

ABSTRACT

We investigate the control system's computational task scheduling problem within limited time and with limited CPU cores in the cloud server. We employ a neural network model to estimate the runtime consumption of linear quadratic regulators (LQR) under varying numbers of CPU cores. Building upon this, we model the task scheduling problem as a two-dimensional bin packing problem (2D BPP) and formulate the BPP as a Markov Decision Process (MDP). By studying the characteristics of the MDP, we simplify the action space, design an efficient reward function, and propose a Double DQN-based algorithm with a simplified action space. Experimental results demonstrate that the proposed approach has improved training efficiency and learning performance compared to other packing algorithms, effectively addressing the challenges of task scheduling in the context of the control system.

1. Introduction

Control science and technology is pivotal in advancing humanity's technological landscape, offering a theoretical foundation and enabling automation across various industrial sectors. Proportional-integral-derivative (PID) control is commonly used in industrial automation for regulating factory parameters, while fuzzy control excels in managing complex nonlinear systems. The aerospace industry leverages optimal control for aircraft performance optimization [1, 2], and the automotive sector employs PID to fine-tune engine power, with fuzzy control enhancing Anti-lock Braking System (ABS) efficiency [3]. In power systems, distributed model predictive control ensures economic dispatch, while power electronics control manages energy conversion [4].

Implementing these control systems or methods in specific scenarios necessitates computing resources to aid in obtaining controllers. These computations involve basic mathematical operations or complex matrix operations to determine controller parameters. However, the computational time varies depending on the model of CPUs or the quantity of CPU cores. As the computational complexity of control system solvers continues to rise, local processors' processing power becomes insufficient for solving intricate problems. Consequently, the shift from local processors to cloud computing has become inevitable.

Cloud computing [5] stands as an internet-based computing model that enables individuals and businesses to access computing resources through remote servers. It offers network and computing resources as services, known as cloud services, which users can rent to fulfill their requirements. Infrastructure as a Service [6] stands out as a popular service solution within cloud services. Leading providers like AWS, Azure, Alibaba Cloud, and Huawei Cloud typically offer virtual machines (VMs) to meet users' computing needs, encompassing CPU and memory. With the development of cloud services, technology companies have begun to deploy dedicated clouds for specific tasks. For example, Alibaba has deployed a large-scale ML-as-a-Service cloud [7] specifically for solving machine learning related tasks. When control tasks appear in large numbers, technology companies may need to deploy dedicated service clouds to solve control related tasks.

In the realm of cloud computing, providers employ a multi-NUMA (Non-Uniform Memory Access) [8] architecture to deliver high-capacity VMs with significant memory and CPU resources. Each server encompasses multiple NUMAs, and each NUMA houses several CPU cores and memory space. The number of CPU cores per server remains limited. When handling varying VM creation requests, such as CPU demands ranging from 2 cores to 64 cores, the service provider creates VMs with the corresponding CPU cores to manage user tasks [9, 10]. VMs with different core capacities require distinct durations to process computational tasks. Upon receiving VM requests, the cloud service provider schedules VM creation based on the server's capacity and allocates user tasks at different intervals. In the current cloud environment, to ensure the user's resource request experience, once the user submits a task request, it is necessary to make a timely task scheduling decision to determine the time period for which the task should run and feedback this information to the user. The server then creates a VM to fulfill the user's task based on the task scheduling decision. These characteristics require the system to perform online task scheduling and to make task scheduling decisions as soon as the task request arrives, without the need for prior knowledge. During this process, we can create a resource-time table to pre-plan users' task requests for better utilization of server resources. For example, the

*Corresponding author
yuhaoliu@stu.jiangnan.edu.cn (Y. Liu); yuqingni@jiangnan.edu.cn (Y. Ni); chang.dong@durham.ac.uk (C. Dong); junchen@jiangnan.edu.cn (J. Chen); feiliu@jiangnan.edu.cn (F. Liu)

cloud server places the task sequences in such a resource-time table based on the user's task requests in a specific period (e.g., 24 hours) with limited CPU cores. Once the table is planned, the cloud scheduler follows this table to direct the server to allocate resources for each task in the task sequence in different periods. On this basis, determining how to schedule the resource-timetable to achieve efficient utilization of server resources becomes an open problem.

This scheduling problem encapsulates two constraints: time and CPU cores, driven by the finite working hours and limited CPU cores in the server. The objective is to efficiently utilize server resources within working hours, framed as a two-dimensional bin packing problem (2D BPP) [11, 12, 13]. This 2D BPP translates time on one axis and CPU cores on the other, providing an optimal solution to the task scheduling problem [14, 15].

Addressing the 2D BPP has seen extensive prior work. Traditional heuristic algorithms, including First-Fit [16], Next-Fit [17], and Best-Fit [18] methods, have played a significant role in balancing space utilization and efficiency [19]. However, they often fall short of providing an optimal solution. Additionally, meta-heuristic algorithms like genetic algorithms, simulated annealing, and ant colony algorithms have also found some application in solving the 2D BPP [20, 21]. Previous researchers have integrated deep learning into solutions for combinatorial optimization problems [22, 23]. Some relevant studies have focused on the application of deep learning models to solve the Traveling Salesman Problem and scheduling problems [24]. Furthermore, research has emphasized the combination of traditional methods with deep reinforcement learning (DRL) [25, 26] to fully leverage the advantages of both. In traditional reinforcement learning [27], the Q-learning algorithm [28, 29] creates a table to store the collected interaction data, allowing for the evaluation of the value of each state-action pair. Based on these values, it selects the optimal policy. With the development of deep learning, Deep Q-Network (DQN) [30, 31, 32] replaces the state-action value table with parameterized functions and utilizes each sampled interaction data to update and improve the deep neural network. This significantly enhances learning efficiency. Using these methods, research has explored using images as inputs and employing convolutional neural networks as the Q-network to solve the 2D BPP [33]. The Actor-Critic method [34] in reinforcement learning has also proven effective in tackling the 2D BPP [35].

In this work, we investigate the scheduling of computational tasks in control systems under constraints of limited time and a finite number of CPU cores. Our exploration commences by focusing on the infinite-time linear quadratic optimal control problem within linear systems. Addressing this optimal control issue involves designing a linear quadratic regulator (LQR) to ensure system stability and meet performance criteria. The coefficients in the LQR bear a direct association with the CPU runtime required for solving this control problem. To comprehensively understand this relationship, we systematically measure the time needed to obtain the LQR for different coefficient matrices across various CPU core configurations, generating a dataset. Acknowledging the impossibility of accounting for all conceivable coefficient matrix scenarios, we leverage a neural network to estimate runtime under different matrices and CPU core numbers. The collected data serves as training input for the neural network, enabling it to generalize and predict runtimes for obtaining LQRs under diverse conditions.

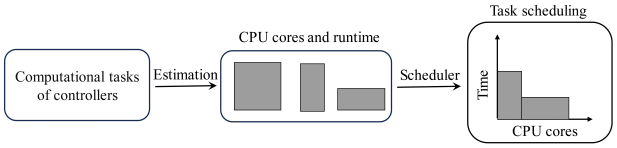


Figure 1: Illustration of the complete process of task scheduling.

Armed with information on runtime variations under different CPU core numbers for control problems, we proceed to create VMs with varying CPU core numbers, adhering to the constraint of a limited CPU core quantity. Subsequently, we strategically plan the schedule of these VMs within the server's resource-time table, factoring in the time each VM requires to solve its designated task. A visual representation of this process is elucidated in Fig. 1. Addressing the scheduling conundrum involves solving a 2D BPP. For this purpose, we deploy a model-free Double DQN, a method seamlessly integrating deep neural networks with Q-learning. This amalgamation furnishes an efficient algorithm for devising optimal scheduling strategies. Expanding on this approach, we streamline the action space of the Markov Decision Process (MDP) and formulate more judicious and effective reward functions. This refinement enhances the learning speed and resource utilization efficiency of the scheduling design based on Double DQN. Compared to some other reinforcement learning algorithms, the simplified action space approach proposed in this paper shortens the training episode of the model and can train efficient models in shorter episodes. The results of this method also outperform traditional heuristic algorithms.

The main contributions of this work are summarized as follows:

1) We investigate the resource requirements of computational tasks in LQR with the background of utilizing cloud server resources for computational tasks in control systems. We propose a packing algorithm based on Double DQN to address the task scheduling problem within limited time and with limited CPU cores in the server.

2) In the MDP modeling of our proposed packing algorithm based on Double DQN, we represent the state of the bins using a tuple composed of four elements, establish an action space that continuously shrinks as the packing progresses, and design an efficient reward function. Compared to other packing algorithms, our approach

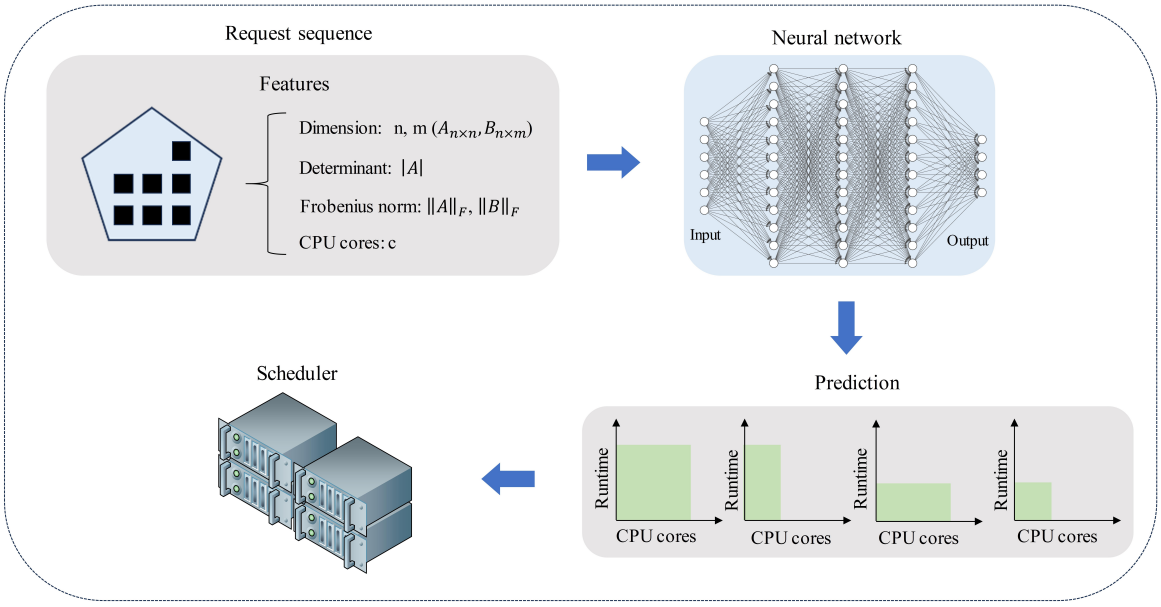


Figure 2: Illustration of a neural network predicting and classifying the required CPU cores and runtime for LQR.

addresses the challenges of large action spaces and slow training speeds and demonstrates superior performance.

2. Neural Network-Based CPU Runtime Estimation for LQR

This paper aims to address the control system's computational task scheduling problem within limited time and with limited CPU cores in the sever. The first step in solving this problem involves measuring the CPU runtime under varying numbers of CPU cores for specific control problems. By combining the obtained data, we employ deep learning methods to predict the runtime needed for a certain category of control problems under varying numbers of CPU cores.

In optimal control, LQR is an important control method. For example, in the field of aircraft control, LQR is often used as a control method to make the aircraft perform maneuvers according to a desired trajectory with good stability [36]. In the field of the Internet of Things (IoT), LQR can be used as temperature controllers in smart buildings to regulate the temperature in the building [37]. So in this paper, we illustrate the infinite-time linear quadratic optimal control problem in linear systems. This optimal control problem is addressed by designing an LQR to meet system stability and performance criteria. The LQR controller solution in the optimal control problem is used as a representative of the control problem.

For a given continuous-time linear time-invariant system, its basic expression can be described using the state-space model,

$$\dot{x}(t) = Ax(t) + Bu(t), \tag{1}$$

where $x(t) \in \mathbb{R}^n$ is the system state vector, $u(t) \in \mathbb{R}^m$ is the control input vector, $t \in [0, \infty)$, and A and B are coefficient matrices of corresponding dimensions. This system is fully controllable. In the design of the optimal controller to satisfy system stability and specific performance criteria, the optimal control strategy is derived by calculating the quadratic cost function,

$$J = \int_0^\infty (x(t)^T F x(t) + u(t)^T R u(t))dt, \tag{2}$$

where R is positive definite and F is positive semi-definite. By designing a linear feedback controller to minimize the cost function J,

$$u(t) = -Kx(t), \tag{3}$$

where $u(t)$ is the control input and K is the state feedback matrix, a key component of the LQR used to compute the control input. Through the computation of the Algebraic riccati equation (ARE),

$$A^T P + PA - PBR^{-1}B^T P + F = 0, \tag{4}$$

we can compute matrix P. To make the algebraic Riccati equation admit a unique positive definite solution $P > 0$, $(A, F^{\frac{1}{2}})$ is assumed to be observable [38, 39, 40]. Once P is computed, K can be obtained as follows.

$$K = R^{-1}B^T P. \tag{5}$$

After calculating the state feedback matrix K, the input of the controller can be obtained using Eq. (3). Therefore, obtaining the LQR is essentially equivalent to the problem of computing K.

2.1. Computational Resource Measurement

As mentioned earlier, the essence of obtaining LQR lies in computing K through Eq. (4) and Eq. (5). This paper relies

on experimental testing to determine the runtime required for calculating K on a computer under varying numbers of CPU cores. For the solution to Eq. (4) and Eq. (5), we conduct tests by using state and input matrices of varying dimensions. For time assessments, we utilize the built-in timing tool, i.e., the time module in Python. After conducting multiple tests and averaging the results, we obtain the required time for different dimensions of state matrices and input matrices under varying numbers of CPU cores in obtaining the LQR.

2.2. Neural Network Modeling

From the approach mentioned above of obtaining the runtime under varying numbers of CPU cores required for different dimensions of inputs, it is evident that the dimensions of the state and input matrices have an impact on the time and memory resources required for obtaining the LQR. However, in practical problem-solving, it's not feasible to consider all possible dimensions. Therefore, we introduce a neural network model. To map the features of the data to the required number of CPU cores and the corresponding computing time, we use a neural network consisting of four fully connected layers. The input layer contains six nodes, which receive the six features of the data respectively. The output layer contains four nodes, each of which is a category that consumes different amounts of resources. During our measurement of the running time used in the data, we combine formula analysis and use different combinations of features for the measurement, and eventually decide to use six elements as inputs to the neural network. The first five of them are features of matrix A and matrix B, and the sixth feature is the computing power of the CPU. Since the resource pool has two constraints, i.e., time and CPU, we select four different combinations of time and CPU as network outputs. These include the matrix dimensions, n and m, the determinant of matrix A, the Frobenius norm of matrices A and B, and the CPU cores c. Using these features as network inputs, the corresponding time required and CPU cores are generated as outputs of the network. Based on the number of classifications, the CPU cores and time required for the LQR are divided into different ranges, with different labels representing various time and CPU cores. The network structure is shown in Fig. 2 and it consists of fully connected layers.

In this structure, the neural network is designed to learn the mapping between the dimensions of input coefficient matrices and the associated CPU cores and time requirements. This allows us to efficiently estimate the runtime needed for different dimensions of state and input coefficient matrices under varying numbers of CPU cores when obtaining the LQR.

3. DRL-Based Task Scheduling

As discussed in the previous section, we obtained the CPU runtime required for LQRs with different coefficient matrices under varying numbers of CPU cores. In this section, building upon the findings from the previous section, we address the task scheduling problem by scheduling the resource-timetable within limited time and with limited CPU cores to achieve efficient utilization of the server.

In this paper, we assume that the cloud service provider's server contains W CPU cores, and the server operates for H hours each day. This creates a resource-timetable with a length of H on the vertical axis and W on the horizontal axis. Users can choose from different types of VMs created by the server with varying numbers of CPU cores. The total number of CPU cores for all concurrently created VMs must not exceed the total server core count W. Various users may opt for different VM types and convey their respective computational tasks to the cloud service provider, forming a sequence of tasks. Within the time interval from 0 to H, the server can receive computing tasks from the task sequence. Each computing task requires different processing times under the specified VM type. For instance, the same computing task takes less time to be completed when users choose VM types with more cores compared to those with fewer cores. Building upon this, we assume that a task sequence is obtained from the neural network described in the previous section. For each task in the sequence, we determine the required number of CPU cores, denoted as w ($0 < w < W$), and the corresponding processing time, denoted as h ($0 < h < H$). And we assume this task sequence follows uniform distribution and the tasks in the task sequence must be arranged in order.

Next, the resource-timetable can be viewed as a finite two-dimensional bin. This bin is used for scheduling specific time slots for executing computing tasks, with CPU cores on the horizontal axis and time on the vertical axis. When the server receives a task, we can create a VM of the appropriate duration based on the CPU cores and task's time requirements and schedule it on this bin. Each created VM occupies a certain number of CPU cores and a specific length of time on the bin. In this process, the task sequence can be seen as a sequence of rectangles. Thus, the task scheduling process is formulated as a 2D BPP, and our goal is to maximize the packing density of a sequence of rectangles in a finite two-dimensional bin.

In this problem, each item in the sequence of rectangular objects cannot be rotated, as the length and width represent different meanings. Each rectangular item is placed sequentially, and the length h_i and width w_i of the i-th rectangular item I_i satisfy $0 < h_i < H$ and $0 < w_i < W$. In this scenario, the objective is to maximize resource utilization, called packing efficiency (PE), before the sequence concludes.

$$\mathrm{PE} = \frac{\text{used area in the bin}}{\text{total area in the bin}}. \tag{6}$$

3.1. Markov Decision Process

BPP can be described as a Markov Decision Process (MDP). An MDP is a discrete-time stochastic control process. At each time step i, the stochastic process is in a certain state s_i, and the agent can choose an available action a_i in the state s_i. The stochastic process will randomly transition to a new state s_{i+1} after the next time step $i + 1$, providing the agent with corresponding feedback $\mathcal{R}_{a_i}(s_i, s_{i+1})$. The

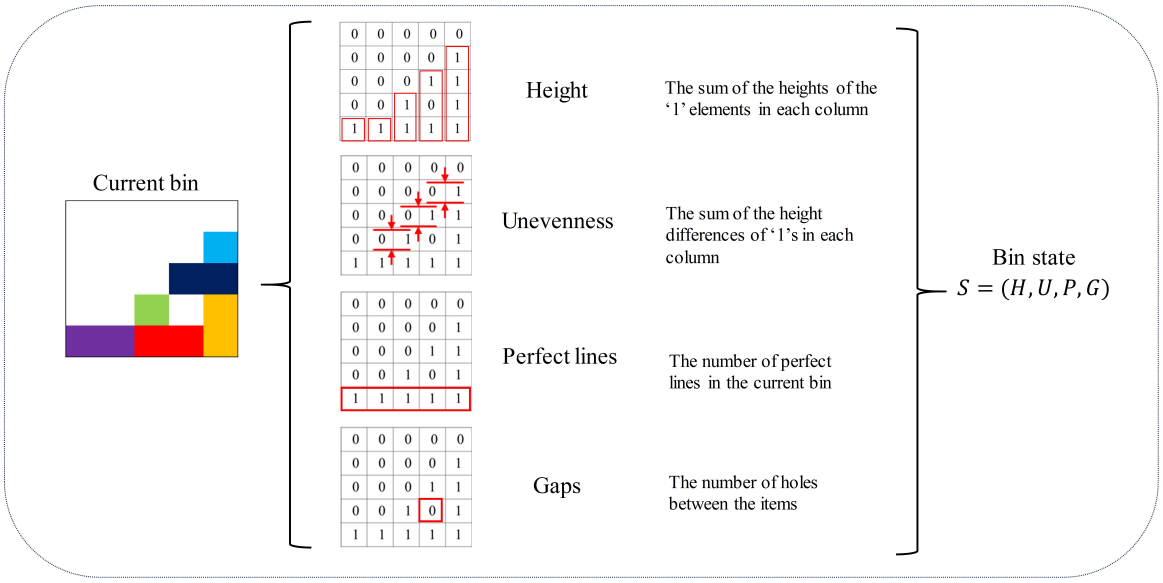


Figure 3: Illustration of extracting four features from the current bin. The four features are used to represent the state of the bin.

chosen action influences the probability of the stochastic process entering the new state s_{i+1}. Specifically, it is given by the state transition function $\mathcal{P}_{a_i}(s_i, s_{i+1})$. Therefore, the next state s_{i+1} depends on the current state s_i and the agent's action a_i. However, given s_i and a_i, it is conditionally independent of all previous states and actions. From this, it can be inferred that an MDP is characterized by four main elements, represented by a quadruple $\mathcal{M} = (\mathcal{S}, \mathcal{A}, \mathcal{P}, \mathcal{R})$, where $\mathcal{S}$ represents the finite set of states that the system can occupy, $\mathcal{A}$ is the set of available actions, $\mathcal{P}$ is the state transition matrix and $\mathcal{R}$ is the reward function.

The state transition matrix $\mathcal{P}$ defines the transition probabilities between all state pairs. The MDP in this paper can be considered model-free. Since this paper does not leverage $\mathcal{P}$ to optimize decisions during the agent training process, we do not introduce $\mathcal{P}$ extensively here. And next, we define $\mathcal{S}$, $\mathcal{A}$, and $\mathcal{R}$ individually. All the notations in section 3 are in Tab. 1.

State

The square container is represented by a two-dimensional array of $H \times W$, where unoccupied space is represented by "0" and occupied space is represented by "1". The items to be placed are also represented by a two-dimensional array of $l_i \times w_i$ $(0 < l_i < H, 0 < w_i < W)$, where all elements are "1". At time step i, the system's state is determined by the current arrangement in the square container. Inspired by Teris, we design a representation of the current state of the bin, which includes an evaluation of the number of filled rows, the gaps created when filled, and the overall bumpiness. Such a representation is intended to allow the agent to evaluate the current action by the change of the bin state after executing the action. These four features are used to represent the current state of the bin, namely: perfect lines, gaps, unevenness, and height. "Perfect lines" indicate the number of rows that are fully occupied (all elements are "1") in the current state. "Gaps" checks whether there are gaps between elements in each column, representing the number of holes between the items. "Unevenness" represents the degree of unevenness in the placement process, that is, the sum of the height differences of "1"s in each column. "Height" represents the sum of the heights of the "1" elements in each column in the current state.

Fig. 3 represents a two-dimensional array of $W = 5, H = 5$ at time step i, where some parts are filled with the element "1" while the rest is filled with the element "0". To visually explain the meanings of each feature, different colors are used to represent different items. The white color represents the unoccupied area, while other colors represent the placed items. In Fig. 3, the purple square and the red square, along with the yellow square, form a perfect line at the bottom of the container, with a quantity of 1. There exists a hole in the column containing the black square and the red square. The height of the column formed by the blue square, the black square, and the yellow square is 4. The height difference with the column to its left is 1. Following this method, at time step i, perfect lines = 1, gap = 1, unevenness $= 1 + 1 + 1 + 0 = 3$, height $= 4 + 3 + 2 + 1 + 1 = 11$. Hence, the state of this bin can be represented as $s_i = (11, 3, 1, 1)$.

The action space and its simplification

In each state, the agent can execute a total of $H \times W$ actions, where each action corresponds to the coordinates of each element in the $H \times W$ matrix. Each coordinate corresponds to the upper-left unit square of the item to be placed. Consequently, not every action is valid, as some actions might result in stacking on occupied cells or placing objects beyond the boundaries of the bin, rendering such operations infeasible. Thus, to expedite training and enhance resource utilization, the action space is pruned, and only

Table 1
Notations in Section 3

Notation	Explanation
s_i	The state at time step i
a_i	The action to be performed at time step i
r_i	The reward after executing a_i
I_i	The item to be placed at time step i
h_i	Length of I_i
w_i	Width of I_i
c_i	The cluster formed at time step i
$C^s_{c_i}$	The size of c_i
$C^c_{c_i}$	The compactness of c_i
A_{c_i}	The size of the smallest rectangle that can enclose c_i
L_{S_i}	The number of perfect lines in state S_i
L_{new}	The difference between $L_{S_{i+1}}$ and L_{S_i}
K_w	A constant multiple of the bin width W
τ	The soft update ratio
γ	Discount factor, $\gamma \in (0, 1)$

valid actions are retained based on the current panel state. When there is no valid action in the action space for a certain state, the state is the terminal state. The process of simplification for action space is shown in Algorithm 1.

Algorithm 1 Simplification for action space.

```
Input action space 𝒜 , item I_i at time step i;
for action in 𝒜 do
    Place I_i in the bin;
    if I_i overlaps with other object and exceeds the
    boundaries of the bin then
        Remove I_i from the bin;
        Delete the current action from action space 𝒜;
    end if
end for
```

Reward function

During the bin packing process, to get higher space utilization, we prefer to place the subsequent arriving items near the previously placed items, preferably without leaving gaps. When there is a gap in a row in the bin and there happens to be an item that can fill the gap, the priority of filling the gap is higher than placing it near a previously placed item. Starting from such a bin packing experience, the following reward function is set.

At time step i, for a given state s_i within its action space, each action yields a different subsequent state s_{i+1}. For these actions that may lead to different outcomes, the following reward function $\mathcal{R}(s_i, a_i)$ is defined:

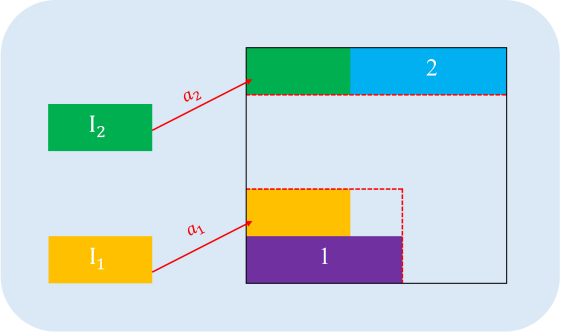


Figure 4: Illustration of the reward for calculating the cluster size and new perfect lines after performing an action.

$$\mathcal{R}(s_i, a_i) = \begin{cases} -2, & \text{the current state has no valid actions,} \\ C^s_{c_i} \times C^c_{c_i} + L_{new} \times K_w, & \text{when a new perfect line is formed,} \\ C^s_{c_i} \times C^c_{c_i}, & \text{otherwise.} \end{cases} \tag{7}$$

The reward function introduces three concepts: cluster size ($C^s_{c_i}$), compactness ($C^c_{c_i}$), and new perfect lines (L_{new}). K_w is a constant multiple of the bin width. These represent the size of the clusters, the degree of compactness of the clusters, and the number of newly formed perfect lines, respectively. According to this reward function, for each placement, the reward value of the corresponding action is related to the size of the formed cluster and the number of newly created perfect lines. In other words, the tighter the placement and the more perfect lines generated, the more reward will be obtained. Following this approach for placement can simultaneously improve overall utilization.

"The size of a cluster ($C^s_{c_i}$)" is explained as follows: If an element is connected to another element either horizontally or vertically, they belong to the same cluster. The value of $C^s_{c_i}$ is related to the size of the cluster formed after the execution of the action; the larger the cluster, the larger the value of $C^s_{c_i}$.

"Compactness ($C^c_{c_i}$)" is explained as follows: At time i, action a_i is taken, resulting in a new cluster c_i with a size of $C^s_{c_i}$. We introduce a new variable, A_{c_i}, which represents the size of the smallest rectangle that can enclose cluster c_i. Compactness is then defined as

$$C^c_{c_i} = \frac{C^s_{c_i}}{A_{c_i}}, \tag{8}$$

where $C^c_{c_i} \in (0, 1]$. For example, as shown in Fig. 4, if action a_1 is taken for item I_1, leading to cluster c_1 represented by

"1" with a cluster size of $C^s_{c_1} = 5$, and the smallest enclosing rectangle has a size of $A_{c_1} = 6$, then the compactness is $C^c_{c_1} = 5 / 6$. Thus, the reward for action a_1 is $r_1 = C^s_{c_1} \times C^c_{c_1} = 5 \times 5 / 6 = 4.17$.

"New perfect lines (L_{new})" is explained as follows: It refers to the number of new perfect lines generated by the current action, which is calculated by subtracting the current state's perfect lines from the perfect lines in the next state. As shown in Fig. 4, suppose that taking action a_2 for item I_2 in a new cluster c_2, with a cluster size of $C^s_{c_2} = 5$. The minimum enclosing rectangle size for this cluster, denoted as A_{c_2}, is 5. The compactness, $C^c_{c_2}$, is calculated as 5 / 5 = 1. Simultaneously, a new perfect line is formed, with a quantity of 1. The quantity of perfect lines before executing the action, denoted as L_{s_2}, is 0. After executing the action, the quantity of new perfect lines in the new state, denoted as L_{s_3}, is 1. Therefore, $L_{new} = L_{s_3}$ - L_{s_2}= 1 - 0 = 1. If we set the value of K_w to 5, the reward for action a_2 is $r_2 = C^s_{c_2} \times C^c_{c_2} + L_{new} \times K_w = 5 \times 1 + 1 \times 5 = 10$.

3.2. Deep Q-Network

In the learning procedure, the agent executes an action according to the observed current state s_i and policy π at each time step i and the policy π is evaluated based on the state-action value function $Q(s_i, a_i)$,

$$\begin{aligned} Q_\pi(s_i, a_i) &= \mathbb{E}\big[G_i = \sum_{k=0}^{\infty} \gamma^k r_{i+k+1} \mid (s_i, a_i)\big] \\ &= \mathbb{E}\big[r_{i+1} + \gamma Q_\pi(s_{i+1}, a_{i+1}) \mid (s_i, a_i)\big], \end{aligned} \tag{9}$$

where G_i is the cumulative return for each time step i and r_{i+k+1} denotes the reward obtained at step $i + k + 1$. The variable γ is the discount factor. And the variable γ close to 1 focuses more on long-term cumulative rewards, while γ close to 0 focuses more on short-term rewards. By maximizing the state-action value function overall policies, we can obtain the optimal state-action value function $Q^*(s_i, a_i)$:

$$Q^*(s_i, a_i) = \max_\pi Q_\pi(s_i, a_i). \tag{10}$$

The update of the Q function needs to be iteratively calculated to approximate the optimal $Q^*(s_i, a_i)$, and the scheme is called Q-learning. The deep Q-learning algorithm (DQN) employs a deep neural network to play the role of Q function, which can be denoted as the Q-network $Q_\theta(s_i, a_i)$ with θ being the parameters of the neural network. The neural network takes the next state s_{i+1} obtained by performing action a_i under the state s_i as input and generates the value of the action as output. The neural network shown in Fig. 5 consists of three FCs. In the training of the network, θ is updated by

$$\begin{aligned} \theta^* = \arg\min_\theta \mathbb{E}_{(s_i, a_i, r_i, s_{i+1}) \sim D}\Big[\| r_i \\ + \gamma \max_{a_{i+1}} Q_\theta(s_{i+1}, a_{i+1}) - Q_\theta(s_i, a_i) \|^2 \Big], \end{aligned} \tag{11}$$

where D is a buffer that stores the transitions (s_i, a_i, r_i, s_{i+1}) during the interaction between the agent and the environment and $r_i + \gamma \max_{a_{i+1}} Q_\theta(s_{i+1}, a_{i+1}) - Q_\theta(s_i, a_i)$ is called temporal difference error. The role of the replay buffer is mainly to make the samples satisfy the independence assumption and to increase the sample utilization. The data obtained from interaction sampling in MDP does not satisfy the independence assumption by itself because the state at this moment is related to the state at the previous moment. Non-independently and identically distributed data has a significant impact on training the neural network, causing the neural network to be fitted to the most recently trained data. Using empirical playback can break the correlation between the samples and allow them to satisfy the independence assumption.

Algorithm 2 Double DQN-Based BPP Algorithm.

Initialize replay buffer D to capacity N;
Initialize action-value function Q with random weights θ;
Initialize target action–value function $\hat{Q}$ with weight $\theta^- = \theta$;
For episode = 1,..., M **do**
  Initialize state s_1;
  For i = 1,..., T **do**
    According to the item I_i, simplify action space $\mathcal{A}$;
    With probability ϵ select a random action a_i from action space $\mathcal{A}$;
    Otherwise select $a_i = \arg\max_{a_i} Q_\theta(s_i, a_i)$;
    Execute action a_i and observe reward r_i and next state s_{i+1};
    Store transition (s_i, a_i, r_i, s_{i+1}) in D;
    Sample random minibatch of transitions (s_i, a_i, r_i, s_{i+1}) from D;

$$y_i = \begin{cases} r_i, & \text{if } s_{i+1} \text{ is terminal} \\ r_i + \gamma \max_{a_{i+1}} Q_{\theta^-}(s_{i+1}, a_{i+1}), & \text{otherwise} \end{cases}$$

    Perform a gradient descent step on $(y_i - Q_\theta(s_i, a_i))^2$ with respect to the network parameters θ;
    Update θ^- following: $\theta^- = (1 - \tau)\theta^- + \tau\theta$;
  End for
End for

Since the temporal difference error contains the output of the neural network, the target is constantly changing while updating the parameters of the network, which can very easily cause instability in the training of the network. To address this problem, a replica of $Q_\theta(s_i, a_i)$ is introduced which is called the target Q network and denoted as $Q_{\theta^-}(s_i, a_i)$. Eq. (11) can be modified to be

$$\begin{aligned} \theta^* = \arg\min_\theta \mathbb{E}_{(s_i, a_i, r_i, s_{i+1}) \sim D}\Big[\| r_i \\ + \gamma \max_{a_{i+1}} Q_{\theta^-}(s_{i+1}, a_{i+1}) - Q_\theta(s_i, a_i) \|^2 \Big]. \end{aligned} \tag{12}$$

During the process of updating θ, θ^- is copied from θ every fixed numbers steps. This method, known as Double DQN,

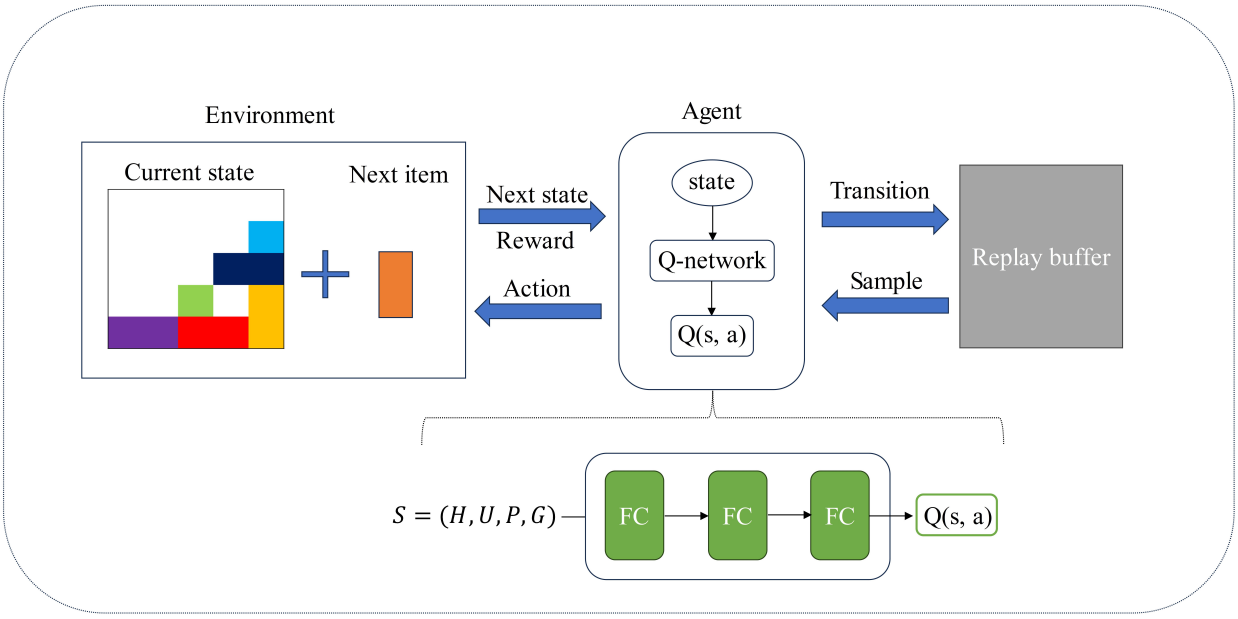


Figure 5: Illustration of the training process for agent.

is used to train the network more consistently and improve the state-action value overestimation problem [30].

At the beginning of the i-th time step, the Q-network generates output as the action-value function for the state-action pair (s_i, a_i). Considering the exploration-exploitation dilemma [27], the agent uses the policy of ϵ-greedy ($\epsilon \in (0, 1)$) to take action.

$$a_i = \begin{cases} \arg\max_{a_i} Q(s_i, a_i), & \text{with the probability of } 1-\epsilon, \\ \text{random } a_i \text{ in } \mathcal{A}, & \text{otherwise.} \end{cases}$$

The ϵ-greedy policy is to choose the action of the optimal Q^* with probability $1-\epsilon$ and to choose a random action for all other cases. The agent executes the action, i.e., placing the incoming item into the bin, and thereby receives a reward r_i from the environment. Meanwhile, the target network computes the Q-value for this state-action pair. And the network updates parameters by Eq. (12). Compared with the original Double DQN algorithm, the update form of θ adopts a soft update technique:

$$\theta^- = (1-\tau)\theta^- + \tau\theta, \tag{13}$$

where τ denotes the soft update ratio. This technique is helpful to reduce the fluctuation of the output curve during network training. The whole process of Double DQN-Based BPP is shown in Fig. 5 and the algorithm is shown in Algorithm 2.

4. Experiments And Results

The application of scheduling algorithms is in the actual service cloud or CloudSim and OpenStack, two cloud computing infrastructure and service frameworks. However, the algorithm proposed in this paper is different from the heuristic algorithm and requires parameter tuning and a certain amount of training time. Therefore, this section uses Python to simulate the scheduling process and train and verify the model. In this section, we first obtain the resource quantity prediction of the control task and then perform resource scheduling. And to verify the effectiveness and efficiency of the proposed model in this paper, this section presents experimental results for both resource measurement and neural network modeling, as well as experiments and results for the evolution of task scheduling into the BPP. The neural network achieves excellent performance in the measurement and classification of CPU cores and runtime. In the BPP experiments, we provide a detailed description of the data, procedures, and parameters of the training settings. We then compare the effectiveness of the model proposed in this paper with other packing algorithms and present the packing utilization results using bins of different sizes. Our model outperforms these algorithms.

4.1. Results of Neural Network Classification

We choose to test the CPU runtime required by the LQR through experiments. As indicated by Eq. (4) and Eq. (5), obtaining the LQR requires obtaining the P matrix and the K matrix, where the input matrix consists of four matrices: A, B, F, and R. We randomly generate A and B matrices of different dimensions while ensuring controllability and observability, where A is an $n \times n$ square matrix, and B is an $n \times m$ matrix. The matrices F and R are diagonal matrices with dimensions corresponding to matrix A and elements set to 1. In the experimental testing, we utilize the API, time module, in Python to test the running time of obtaining the LQR controllers. Due to the Global Interpreter Lock (GIL) in Python, only one thread can be in an execution state at

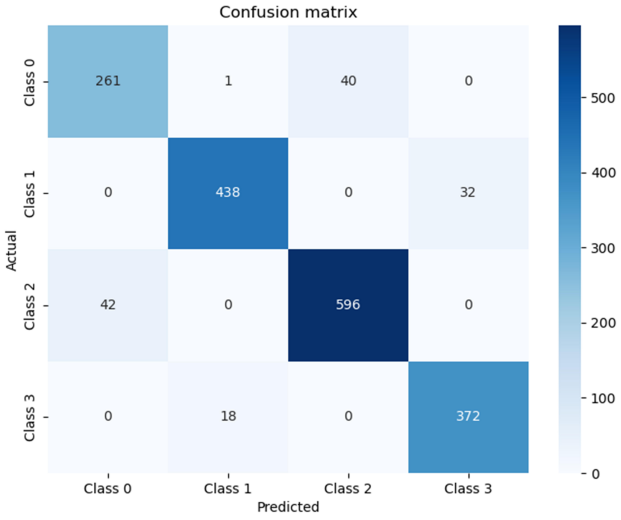


Figure 6: The confusion matrix for the neural network classification, class 0, 1, 2, 3 represent categories (u_l, t_l), (u_s, t_l), (u_l, t_s) and (u_s, t_s) respectively.

any given time. Therefore, when running the program on a single core, we limit the core's performance to simulate differences in performance with varying numbers of cores on cloud servers. We conduct multiple tests and take the average values to minimize testing errors. The experiments are conducted on a computer with a 12th Gen Intel(R) Core(TM) i5-12500 processor.

Based on the above approach, the obtained dataset ranges from 1 to 6 for n and from 1 to 10 for m, resulting in 60 possible combinations for matrices A and B. Each combination corresponds to 100 sets of random matrix data, resulting in a total of 6000 sets of data. These data are run on a a single CPU core with frequencies of 1.93GHz and 2.2GHz respectively and their runtime is measured, resulting in 12,000 sets of data. For these 12,000 sets of data, we plan to categorize them into four groups based on the runtime in the dataset and the different performances of the single core: (u_l, t_l), (u_s, t_l), (u_l, t_s) and (u_s, t_s). u_s and u_l represent small and large performances of CPU core, and t_s and t_l represent small and large time requirements. To address this classification problem, we introduce the neural network structure shown in Fig. 2, with 6 input nodes, 3 hidden layers each consisting of 32 nodes, and 4 output nodes representing the four categories. The model employs the Adam optimizer with a learning rate of $1e^{-3}$, and the final classification accuracy is 93%, the confusion matrix for the classification is shown in Fig. 6.

4.2. BPP Experiments and Results

After estimating the CPU cores and runtime for different dimensions of LQR using a neural network and classifying them based on their resource needs, the scheduling of tasks can be initiated. As described in Section 3, the resource

Table 2
Hyperparameters during model training with 5×5 bin size

learning rate	$2e^{-3}$
discount factor	0.99
batch size	512
replay buffer capacity N	$5e4$
optimizer	Adam
activation function	Relu
soft update ratio τ	$5e^{-3}$
initial epsilon	1.0
final epsilon	$1e^{-3}$
episode	$2e4$
K_w	0.5

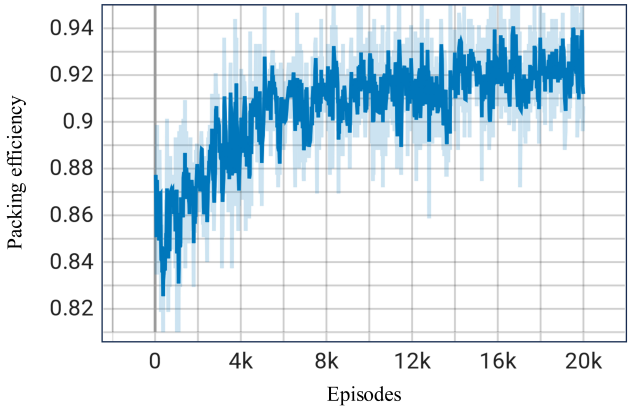
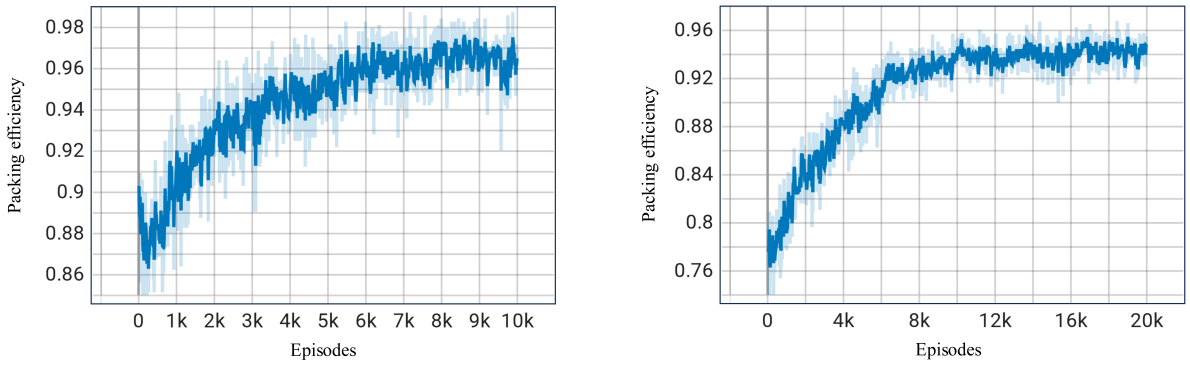


Figure 7: Packing efficiency (y) vs. Iteration (x) plot during training phase with rectangular items inside a squared bin (5 × 5).

scheduling involved in this paper can be depicted as a 2D BPP, which we resolve using the Double DQN from deep reinforcement learning. The implementation of this method is carried out in PyTorch.

This model utilizes the Adam optimizer for optimization. The Q-network structure follows the configuration illustrated in Fig. 5, with 64 nodes in each hidden layer. The specific parameters for the model training can be found in Tab. 2.

We chose a square bin with a size of $W \times W$ to place the items of different sizes. To do this, we first generate a certain number of items of size $w \times h$ ($1 \le w \le 2, 1 \le h \le 2$). Their shapes are rectangles and squares. The ratio of the number of each category is the same as the ratio of their sizes, and the total size needs to exceed the capacity of the bin. The order of items in this sequence is random, and then these items are sent to the algorithm one by one and the sequence L will be placed in the $W \times W$ bin. During the iterations of



(a) 8×8 bin (b) 10×10 bin

Figure 8: Packing efficiency (y) vs. Episode (x) plot during training phase with rectangular items inside a square bin (8×8 and 10×10).

training, send the items in L in order and allocate appropriate rewards after each action. The Q-network is updated based on the reward value after each action is executed, eventually resulting in an approximately optimal packing strategy.

For the baselines choosing, we first select two heuristic algorithms: ShelfNextFit and Skyline as baselines. We also select a reinforcement learning algorithm: Deep-Pack for comparison with the proposed method. The baselines are summarized as follows:

- ShelfNextFit [14]: Place the new item on the last open shelf.
- Skyline [41]: Select the empty space with the smallest y-axis and the smallest x-axis to place the new item.
- Deep-Pack [33]: Use a deep convolutional network to fit the action value function in DQN, and place the items according to the action value.

Fig. 7 shows the packing efficiency of the 5×5 bin during the training phase. Due to the simplification of the action space by retaining only valid actions, the bin utilization rate is relatively high in the early stages of training. As the iterations progress, the packing efficiency becomes progressively higher until it levels off. In Tab. 3, we present the packing results for the 5×5 bin using both the proposed method and baselines, respectively. Compared with heuristic algorithms, the performance of the algorithm proposed in this paper surpasses them. Compared with Deep-Pack, the superiority of the proposed algorithm is mainly reflected in the convergence speed of the algorithm.

Furthermore, this paper also extends the bin sizes, which introduces an issue of increased action space. In Deep-Pack, the number of actions is determined by the size of the bin, which is $W \times W$. With the expansion of bin sizes, the action space would proportionally increase, leading to a significant increase in the matrix that the convolution has to handle. This would inevitably result in a huge increase in training time. However, in the proposed method, data preprocessing is conducted in advance, with the characteristics of the bins serving as the network input. Only valid actions are included in the action space, greatly reducing the training time while still achieving favorable results, as shown in Fig. 8. Because Deep-Pack requires massive time to train in the expanded size bin packing, we only compare the proposed method with the heuristic algorithm, as shown in Tab. 4.

Table 3
Utilization of proposed method and baselines with 5×5 bin size

BinPack method	Bin size	Train episodes	Utilization
Proposed Method	5×5	**20,000**	**92.32%**
Deep-Pack	5×5	300,000	91.00%
ShelfNextFit	5×5	——	76%
Skyline	5×5	——	80%

Table 4
Utilization of proposed method and baselines with different bin size

Bin size	BinPack method	Utilization
5×5	Proposed Method	**92.32%**
	ShelfNextFit	76%
	Skyline	80%
8×8	Proposed Method	**96.36%**
	ShelfNextFit	70.31%
	Skyline	84.38%
10×10	Proposed Method	**94.25%**
	ShelfNextFit	76%
	Skyline	91%

Table 5
Ablation study with different bin size

Bin size	Reward function	Action space	Utilization
5×5	Proposed	Simplified	**92.32%**
	Proposed	Total	47.15%
	+1	Simplified	90.05%
	+1	Total	42%
8×8	Proposed	Simplified	**96.36%**
	Proposed	Total	15.82%
	+1	Simplified	90.93%
	+1	Total	15.68%
10×10	Proposed	Simplified	**94.25%**
	Proposed	Total	14%
	+1	Simplified	93.08%
	+1	Total	10.5%

4.3. Ablation Study

In addition to the training and evaluation of the proposed method, ablation studies are conducted using modified models generated by partially masking modules/parts. In the first study, the reward function was modified based on the proposed method by changing it to the simplest reward setting, where each time an item is successfully placed, the environment will return a reward with the value of 1. Such a reward treated all placement actions equally, challenging the learning process. As indicated in Tab. 5, the final experimental outcomes were also inferior to the method proposed in this paper. In the second study, the method of simplifying the action space was not employed during the packing process. Since the action space was not simplified, there was a significant probability that the agent would choose actions that had been selected before, leading to negative rewards and the termination of the current episode. This had a substantial impact on training, requiring more time to achieve satisfactory results compared to the method of simplifying the action space. The larger the action space, the more challenging the training process becomes. The experimental results showed a noticeable difference between the two approaches under the same training episodes. In the third study, a simple reward function and the total action space were employed, and the final results are presented in Tab. 5. In these studies, the training episodes corresponding to the same bin size are identical and the results in Tab. 5 indicate that the proposed reward function and the simplified action space method improved the efficiency of the algorithm, enabling the agent to achieve better results in fewer training episodes.

5. Conclusion

This paper investigates the control system's computational task scheduling problem within limited time and limited CPU cores in the cloud server, taking the problem of obtaining LQR optimal controllers in the control system as an example. A neural network modeling approach is employed to predict runtime for obtaining LQR with different coefficient matrices under varying numbers of CPU cores. After framing the task scheduling problem as a 2D BPP, we propose a BPP algorithm based on Double DQN with a simplified action space and validate its performance through experiments with bins of varying sizes. The experimental results demonstrate that the Double DQN algorithm outperforms baselines, effectively addressing task scheduling problems in the context of the control system.

In future work, we plan to further explore this method on a larger set of tasks and a wider variety of control methods, such as computational resource prediction and scheduling for common methods such as fuzzy control, adaptive control, and robust control. By studying the performance and behavior of this method in different control tasks, we hope to demonstrate its wide applicability and scalability.

References

[1] J. Z. Ben-Asher, Optimal Control Theory with Aerospace Applications, American Institute of Aeronautics and Astronautics, 2010.

[2] J. Zhu, E. Trélat, M. Cerf, Geometric Optimal Control and Applications to Aerospace, Pacific Journal of Mathematics for Industry vol. 9, no. 1 (2017) 1–41.

[3] P. Girovskỳ, J. Žilková, J. Kaňuch, Optimization of Vehicle Braking Distance Using a Fuzzy Controller, Energies 13 (2020) 3022.

[4] M. A. Velasquez, J. Barreiro-Gomez, N. Quijano, A. I. Cadena, M. Shahidehpour, Distributed Model Predictive Control for Economic Dispatch of Power Systems with High Penetration of Renewable Energy Resources, International Journal of Electrical Power & Energy Systems 113 (2019) 607–617.

[5] L. Qian, Z. Luo, Y. Du, L. Guo, Cloud Computing: An Overview, in: Proc. Cloud Computing: First International Conference., Springer, 2009, pp. 626–631.

[6] D. Rani, R. K. Ranjan, A Comparative Study of SaaS, PaaS and IaaS in Cloud Computing, International Journal of Advanced Research in Computer Science and Software Engineering 4 (2014).

[7] Q. Weng, W. Xiao, Y. Yu, MLaaS in the Wild: Workload Analysis and Scheduling in Large-Scale Heterogeneous GPU Clusters, in: 19th {USENIX} Symposium on Networked Systems Design and Implementation ({NSDI} 22), 2022.

[8] C. Lameter, Numa (Non-Uniform Memory Access): An Overview: NUMA Becomes More Common Because Memory Controllers Get Close To Execution Units on Microprocessors., Queue 11 (2013) 40–51.

[9] D. Saxena, A. K. Singh, A Proactive Autoscaling and Energy-efficient VM Allocation Framework Using Online Multi-resource Neural Network for Cloud Data Center, Neurocomputing 426 (2021) 248–264.

[10] J. Sheng, Y. Hu, W. Zhou, et al., Learning to Schedule Multi-NUMA Virtual Machines via Reinforcement Learning, Pattern Recognition 121 (2022) 108254.

[11] V. V. Vazirani, Approximation Algorithms, volume 1, Springer, 2001.

[12] A. Lodi, S. Martello, D. Vigo, Recent Advances on Two-dimensional Bin Packing Problems, Discrete Applied Mathematics 123 (2002) 379–396.

[13] A. Wolke, B. Tsend-Ayush, C. Pfeiffer, M. Bichler, More than Bin Packing: Dynamic Resource Allocation Strategies in Cloud Data Centers, Information Systems 52 (2015) 83–95.
[14] Z. Zhu, J. Sui, L. Yang, Bin-packing Algorithms for Periodic Task Scheduling, International Journal of Pattern Recognition and Artificial Intelligence 25 (2011) 1147–1160.
[15] G. L. Stavrinides, H. D. Karatza, Scheduling Multiple Task Graphs in Heterogeneous Distributed Real-time Systems by Exploiting Schedule Holes with Bin Packing Techniques, Simulation Modelling Practice and Theory 19 (2011) 540–552.
[16] G. Dósa, J. Sgall, First Fit bin packing: A Tight Analysis, in: Proc. 30th International Symposium on Theoretical Aspects of Computer Science (STACS 2013), Schloss Dagstuhl-Leibniz-Zentrum fuer Informatik, 2013.
[17] E. G. Coffman, Jr, M. R. Garey, D. S. Johnson, R. E. Tarjan, Performance Bounds for Level-Oriented Two-Dimensional Packing Algorithms, SIAM Journal on Computing 9 (1980) 808–826.
[18] G. Dósa, J. Sgall, Optimal Analysis of Best Fit Bin Packing, in: Proc. International Colloquium on Automata, Languages, and Programming, Springer, 2014, pp. 429–441.
[19] B. S. Baker, J. S. Schwarz, Shelf Algorithms for Two-Dimensional Packing Problems, SIAM Journal on Computing 12 (1983) 508–525.
[20] E. Falkenauer, A Hybrid Grouping Genetic Algorithm for Bin Packing, Journal of Heuristics 2 (1996) 5–30.
[21] J. Levine, F. Ducatelle, Ant Colony Optimization and Local Search for Bin Packing and Cutting Stock Problems, Journal of the Operational Research Society 55 (2004) 705–716.
[22] J. Schmidhuber, Deep Learning in Neural Networks: An Overview, Neural Networks 61 (2015) 85–117.
[23] Y. Bengio, A. Lodi, A. Prouvost, Machine Learning for Combinatorial Optimization: A Methodological Tour D'horizon, European Journal of Operational Research 290 (2021) 405–421.
[24] M. Saadatmand-Tarzjan, On Computational Complexity of The Constructive-optimizer Neural Network for The Traveling Salesman Problem, Neurocomputing 321 (2018) 82–91.
[25] D. Wang, N. Gao, D. Liu, J. Li, F. L. Lewis, Recent Progress in Reinforcement Learning and Adaptive Dynamic Programming for Advanced Control Applications, IEEE/CAA Journal of Automatica Sinica 11 (2024) 18–36.
[26] B. Kiumarsi, K. G. Vamvoudakis, H. Modares, F. L. Lewis, Optimal and Autonomous Control Using Reinforcement Learning: A Survey, IEEE Transactions on Neural Networks and Learning Systems 29 (2018) 2042–2062.
[27] R. S. Sutton, A. G. Barto, Reinforcement Learning: An Introduction, MIT press, 2018.
[28] C. J. Watkins, P. Dayan, Q-learning, Machine Learning 8 (1992) 279–292.
[29] B. Jang, M. Kim, G. Harerimana, J. W. Kim, Q-learning Algorithms: A Comprehensive Classification and Applications, IEEE Access 7 (2019) 133653–133667.
[30] H. Van Hasselt, A. Guez, D. Silver, Deep Reinforcement Learning with Double Q-Learning, in: Proc. The AAAI Conference on Artificial Intelligence, volume 30, 2016.
[31] V. Mnih, K. Kavukcuoglu, D. Silver, et al., Human-level Control Through Deep Reinforcement Learning, Nature 518 (2015) 529–533.
[32] V. Mnih, K. Kavukcuoglu, D. Silver, et al., Playing Atari with Deep Reinforcement Learning, arXiv preprint arXiv:1312.5602 (2013).
[33] O. Kundu, S. Dutta, S. Kumar, Deep-pack: A Vision-Based 2D Online Bin Packing Algorithm with Deep Reinforcement Learning, in: Proc. 2019 28th IEEE International Conference on Robot and Human Interactive Communication (RO-MAN), IEEE, 2019, pp. 1–7.
[34] T. Haarnoja, A. Zhou, K. Hartikainen, et al., Soft Actor-Critic Algorithms and Applications, arXiv preprint arXiv:1812.05905 (2018).
[35] L. Zhang, D. Li, S. Jia, H. Shao, Brain-Inspired Experience Reinforcement Model for Bin Packing in Varying Environments, IEEE Transactions on Neural Networks and Learning Systems 33 (2022) 2168–2180.
[36] A. Ashraf, W. Mei, L. Gaoyuan, Z. Anjum, M. M. Kamal, Design Linear Feedback and LQR Controller for Lateral Flight Dynamics of F-16 Aircraft, in: Proc. 2018 International Conference on Control, Automation and Information Sciences (ICCAIS), IEEE, 2018, pp. 367–371.
[37] M. Barbiero, A. Rossi, L. Schenato, LQR Temperature Control in Smart Building via Real-time Weather Forecasting, in: Proc. 2021 29th Mediterranean Conference on Control and Automation (MED), IEEE, 2021, pp. 27–32.
[38] C. Peng, W. Zhang, Pareto Optimality in Infinite Horizon Mean-Field Stochastic Cooperative Linear–Quadratic Difference Games, IEEE Transactions on Automatic Control 68 (2023) 4113–4126.
[39] C. Peng, W. Zhang, Multiobjective Dynamic Optimization of Cooperative Difference Games in Infinite Horizon, IEEE Transactions on Systems, Man, and Cybernetics: Systems 51 (2021) 6669–6680.
[40] C. Peng, W. Zhang, L. Ma, Infinite horizon multiobjective optimal control of stochastic cooperative linear-quadratic dynamic difference games, Journal of the Franklin Institute 358 (2021) 8288–8307.
[41] L. Wei, D. Zhang, Q. Chen, A Least Wasted First Heuristic Algorithm for The Rectangular Packing Problem, Computers & Operations Research 36 (2009) 1608–1614.

Acknowledgment

This research was funded by the National Natural Science Foundation of China under Grant 62303196, the Natural Science Foundation of Jiangsu Province of China under Grant BK20231036, the Basic Research Funds of Wuxi Taihu Light Project under Grant K20221005, and the 111 Project under Grant B23008.

Yuhao Liu received the B.Eng. degree from the School of Internet of Things Engineering, Jiangnan University, Wuxi, China, in 2022. He is currently working toward the M.Eng. degree with the Key Laboratory of Advanced Process Control for Light Industry (Ministry of Education), School of Internet of Things Engineering, Jiangnan University, Wuxi, China. His research interests include cloud resource scheduling and deep reinforcement learning.

Yuqing Ni is an associate professor at the School of Internet of Things Engineering, Jiangnan University, Wuxi, China. She received the B.Eng. degree from the College of Control Science and Engineering, Zhejiang University, Hangzhou, China, and the Ph.D. degree in Electronic and Computer Engineering from the Hong Kong University of Science and Technology, Hong Kong, in 2016, and 2020, respectively. From April 2019 to June 2019, she was a visiting student in the School of Electrical Engineering and Computer Science, KTH Royal Institute of Technology, Stockholm, Sweden. Prior to her current position, she was a senior engineer at Huawei from 2020 to 2021. Her research interests include security and privacy in cyber–physical system, networked state estimation, and deep reinforcement learning.

Chang Dong is an assistant professor in Operations Management at Durham University Business School. Before joining Durham University, he obtained a Ph.D. degree in Department of Industrial Engineering and Decision Analytics at Hong Kong University of Science and Technology in 2019, and then worked as a postdoctoral fellow in Chinese University of Hong Kong from 2020 to 2022. His research interests broadly include topics in supply chain management, marketing-operations interface, and innovations and new business strategies.

Jun Chen received the B.Sc. degree in automation from the Wuxi Institute of Light Industry, Wuxi, China, in 2003, and the M.Sc. and Ph.D. degrees in control theory and control engineering from Jiangnan University, Wuxi, China, in 2005 and 2009, respectively. Currently, she is an associate professor with the Institute of Automation, Jiangnan University. Her research interests include fuzzy control theory, advanced control theory and their applications.

Fei Liu received the B.Sc. degree in electrical technology and the M.Sc. degree in industrial automation from the Wuxi Institute of Light Industry, Wuxi, China, in 1987 and 1990, respectively, and the Ph.D. degree in control science and control engineering from Zhejiang University, Zhejiang, China, in 2002. He is currently a professor with the Institute of Automation, Jiangnan University, WuXi, China. His main research interests include robust control, process control system, Markov jumping system, anti-disturbance control, and their applications.

Durham University
Durham Research Online

Citation on deposit:

Liu, Y., Ni, Y., Dong, C., Chen, J., & Liu, F. (2024). Task scheduling for control system based on deep reinforcement learning. Neurocomputing, 610, 128609.
https://doi.org/10.1016/j.neucom.2024.128609

For final citation and metadata, visit Durham Research Online URL:
https://durham-repository.worktribe.com/output/3084273

Copyright Statement:

This accepted manuscript is licensed under the Creative Commons Attribution 4.0 licence. https://creativecommons.org/licenses/by/4.0/