Topic 2: Deep Learning for Cloud Computing

Team Member Information
Xuan Zuo xuanzuo@usc.edu
Yingzhu Liu yingzhul@usc.edu
Shahid Mohammed shaikbep@usc.edu

Problem Description

With the rapid development of the cloud computing industry, resource and task management have become key factors. For cloud service providers (CSPs), they care about how to minimize costs while ensuring quality of service (QoS). In this project, we will implement a system using deep reinforcement learning(DRL). This system cloud minimizes costs by resource provisioning and task scheduling. We refer to relevant literature[13] and system implementation methods, we use google cluster[1] usage traces to build our user workload model, and use a deep Q network to find the best RP/TS strategy. The most challenging part is dealing with the large amount of user requests, maintain the dynamic task queue(s), and build the two-stage RP-TS processor.

Project Timeline

In phase III, we merge the DQN part and the environment part, train the model, and test on small-scale data and compare with baselines, get the conclusions. In this project, we use a dynamic task queue(s) for user requests, and improved the environment by parallel processing. We also modify the abstract, introduction, Model Description, training, test result and get the conclusion to complete the report.

We completed the corresponding parts according to the proposal. In phase I, we have learned the theory of reinforcement learning and DQN, and had some understanding of the RP/TS system which we will implement. We downloaded our user data from google cluster and read the data introduction carefully. We used Python and C ++ to decompress and preprocess the data, modeled the user workload, transformed the DAG to a task queue, which is the input of our environment and DQN. We also finished the abstract and introduction parts of final report.

In phase II, we built the environment of two-stage RP-TS processor and the deep Q network model, implemented one baseline (Round-Robin Method) and tested small scale data on the baseline. We added the model description, experiment results and discussion of our final report.

Deep Learning for Cloud Computing

Xuan Zuo
University of Southern California
Los Angeles, CA, USA
xuanzuo@usc.edu

Yingzhu Liu
University of Southern California
Los Angeles, CA, USA
yingzhul@use.edu

Shahid Mohammed
University of Southern California
Los Angeles, CA, USA
shaikbep@usc.edu

***Abstract*—With the development of the cloud computing industry, the scale of cloud service providers (CSPs) has also increased. However, due to the low utilization of the server, a large amount of electric and computing power is wasted. For CSPs, it is necessary to reduce costs while maintaining Quality-of-Service (QoS). In this project, we built a system to reduce energy costs by using resource supply (RP) and task scheduling (TS) decisions. We use Google cluster data[1] and consider the dependencies between different tasks and parallel processing to build user workload model. The environment model simulates the server architecture, energy consumption and dynamic electricity price to obtain the energy cost and rejection rate of the CSPs under current strategy. Through the usage of two-stage RP-TS processor based on deep Q network (DQN), with target network and experience replay, it automatically generates the optimal long-term decisions. To evaluate system performance, we used the Round-Robin method(RR) as our baseline, and compared the running time, energy consumption, and rejection rate as measurement. Base on tests, our system has increased energy consumption by an average of 220% and reduce reject rate by 50% compared to baseline.**

Keywords—deep Q learning, deep reinforcement learning, task scheduling, cloud resource management.

I. INTRODUCTION

In the past decade, the cloud computing industry has grown rapidly. For cost and security reasons, many enterprises choose to lease servers from Cloud Service Providers (CSPs) instead of establishing and maintaining their own local servers[2, 3]. However, for CSPs, not only the construction costs are high, but also the maintenance and operation costs (electricity costs) are also high. With the rise of large "super large-scale" data centers, it is estimated that by 2020, the power consumption of data centers will be about 73 billion kWh[4, 5]. One of the reasons for this high-power consumption is the low server utilization. The data shows[6], for cloud computing servers, the best energy utilization rate is 70-80%, but in fact due to the uncertainty of user request, it is often below 50%. So, we need more effective ways to simulate user requested and make decisions.

In this project, we focus on the provision of resources and task schedule, which is also the core issue of machine learning. Among many machine learning algorithms, Reinforcement Learning (RL) is the most suitable for decision making[7]. Studies in recent years have shown that Deep Neural Networks (DNNs)[8] and DL performs really well on decision control problems in complex high-dimensional systems, such as gaming[9], Go[10], and cooling datacenters[11]. In our project, energy costs are affected by server architecture, server utilization, power consumption, and real-time electricity prices, and we will minimize long-term costs by adjusting task schedule and resource provision. Therefore, according to previous work[8, 10-11], we consider using the RL algorithm for this problem. In this problem, CSPs need a system that can automatically generate optimal policies under different user requests and available resources, and this is the advantage of artificial agents. With RL, models can automatically learn from changing environments and feedback, and minimize cost. To achieve this goal, we adopted a combination of RL and deep learning—DRL. We refer to Google DeepMind's algorithm[15], use Deep Q-Networks (DQN) to store previous decision-making experience, and replay these experiences to obtain better learning results.

In the previous paper, Li[20] proposed a model of a cloud server platform, which simulates server architecture and energy consumption. Mao[12] proved the advantages of RL in dealing with resource provision and task schedule problems: the repetitiveness of decision-making generates a large amount of training data for the RL algorithm; RL models the complex decision-making process as a deep neural network similar to the model used for game agents. In addition, Cheng[13] proposed an innovative two-stage RP-TS processor based on DRL, which It is optimized in terms of decision-making methods and speed. And Tong[14] proposed the performance of different DQL algorithms on task scheduling problem, and define the cloud computing environment, task model, and scheduling structure. Inspired by those, we implement the two-stage RP-TS processor and pay more attention on task independence and task schedule details, we update the task status in real time and maintain a dynamic ready task queue to reduce running time and rejection rate.

In this project, the system is mainly composed of two parts: user workload model and two-stage RP-TS processor. The user workload model sorts and managers user requirements into task queue(s); then, the two-stage RP-TS processor simulates the resource information of the server, and the agent(DQN) part generates decisions. In order to improve the efficiency of the system, we dynamically update the ready task queue and implement the framework that same task can running parallel on different Virtual Machines(VMs). Our goal is to minimize long-

term energy costs through training while ensuring that the rejection rate is maintained at a low level. we will first implement this system, and then test our system's performance in complex environments through diverse measurement, and refer to other research results for optimization[14].

The main contribution of our work is as follows:

- We implemented a Deep Q-Learning training algorithm on agents that can minimizes long-term total costs while ensuring QoS through RP / TS.
- We established an environment framework, used multiple languages and libraries, simulated the server architecture, resources , and infrastructures of CSPs, and provide real-time resources usage status and task to agent, calculate costs and rejection rates based on the dynamic electricity prices.
- We built a user workload model with dependency mode[16], handling directed acyclic graph(DAG) tasks into ready task queue(s), considered the dependence and sequence relationship between each task, and dynamically updated.

II. FRAMEWORK OF DRL-BASED RP-TS SYSTEM

The system is composed of user workload model, environment model and agent(DQN). We consider two types of resources (CPU and memory), and user requests are converted into ready task queues through the user workload model to reach the environment. The environment passes real-time status and individual tasks to the agent, and the agent allocates tasks based on long-term optimal goals. The user workload model is used to process a large number of user requests and generate ready task queues based on dependency and admission control. The environment model includes power consumption model, dynamic price model, rejection rate and recourse release functions, which are used to provide real-time resource usage and task information to agents. Deep Q network is the system's decision-making agent, which generates long-term optimal decisions through experience replay and fixed q-target.

A. User Workload Model

In this system, we use "dependency mode scheduling"[16], which means we consider the dependencies between each task. Each job consists of one or more tasks, and the tasks may be independent or dependent on each other. Therefore, the user workload can be viewed as composed of multiple Directed Acyclic Graphs(DAG). In this case, we can build user models more precisely and improve the usability of the system[17-19]. The user workload model processing large amount of user requests as input.

1) Job Characteristics

One user workload model consists of many jobs and a job contains multiple tasks. There exists dependencies between multiple tasks. For example, if task A depends on the results and data of task B, then task A can only be executed after task B is completed. To record relationships between tasks and follow the dependency rule, we use a DAG to represent a job. In a DAG, a vertex represents a task, and an edge represents the relationship between two tasks.

2) Task Characteristics

For each task, users provide information like required $CPU(D_{CPU})$, required $RAM(D_{RAM})$, start time T_{start}, estimated executable time L, user specified hard deadline T_{ddl} and dependencies between tasks within one job. Based on Admission Control Policy, assume with infinite resources, if a task cannot be completed by the head deadline, it should be rejected immediately. Also, according to SLA, $T_{start} + L \leq T_{ddl}$ should be satisfied. Besides, the CSP supports VMs with the required amount of $CPU(R_{CPU})$and memory $RAM(R_{RAM})$, and to execute one task on one VM successfully, $D_{CPU} \leq R_{CPU}$ and $D_{RAM} \leq R_{RAM}$ must be satisfied.

3) Parallel tasks processing

To improve the efficiency of the environment, we assigned one task to different servers instead of to only one server. For each task, we first divided it into a random number n (range from 1 to 5) of sub-tasks and each one has a random percentage CPU and RAM of the total R_{CPU} and R_{RAM} of this task. And we assume the first n-1 subtasks can run in data parallelism[31] and the last subtask merge all results and data from the first n-1 subtasks, so the first n-1 sub-tasks are independent and the last one depends on them. Then we treat all sub-tasks as separate tasks and deal with them as what we did before. Besides, if one sub-task is rejected, the others which came from the same task need to be rejected too. In this way, each sub-task has a smaller required CPU and RAM, so each VM can be allocated more efficiently. Also, if one task needs to run for a long time, dividing it into several sub-tasks and running them in parallel can save a lot of time.

B. Environment Model

We expect that our system will minimize long-term energy consumption. To achieve this goal, we have built an environmental model[20] to simulate the real-world cloud server platform structure and calculate energy consumption and costs. We used real user request data[1] and process it by user workload model. After that, we build an environment model focus on real-time resources usage information provide and total energy cost calculate.

The Figure 1 is the flow chart of the environment. The task queue is generated by the user workload model and as input to the environment model. It is dynamic and the length of tasks is unknown[14], so we use DRL and deep Q network(DQN) as an agent to automatically generate the best decision.

For task data of different scales, we simulated the establishment of different numbers of servers and server farms: for small-scale problems(less than 5,000 tasks), we set 100~300 servers and 10 server farms; for large-scale problems(more than 50,000 tasks), we set 500~5000 servers and (servers' number / 50) server farms[13]. For each server, we assume that there are 1 unit of CPU resources and 1 unit of RAM resources. We connect some number of servers together as a server farm. In reality, the number and structure of server farms are determined by the cloud service providers(CSPs). In each server, we assume there are 5 virtual machines, and each task can runs in one of those virtual machines.

During the operation of the system, we pass an individual task to the two-stage RP-TS processor every time, and calculate the rewards according to the agent's decision. The two-stage RP-TS processor is based on DQN and makes a two-step judgment based on the current situation: first determine the server farm to which the task is assigned, and then select the server. When a task runs on a virtual machine, the corresponding resources will be occupied (currently available resources = previously available resources - resources required by the task). During the whole process, we update the resource usage information and calculate total energy cost based on it.

1) Power Consumption Model

The energy consumption of the server is not linear, the optimal energy utilization rate is about 70%[21], and then significantly increased. In order to minimize the consumption of cloud computing servers, we hope to stabilize the server-side utilization rate around the optimal value. Therefore, we have established a model of energy consumption and estimated the current energy efficiency (power) through the current use status of the server.

We use the following formula[22] to calculate the utilization rate of server m, where C_{CPU} is the total resource amount (unit resource), $\sum_{v=1}^{V} R_{CPU}$ is the amount of resources being occupied, where R_{CPU} is the amount of resources required for each task, and v represents the tasks that currently running on this server.

$$U_m = \frac{\sum_{v=1}^{V} R_{CPU}}{C_{CPU}} \quad (1)$$

The power of the server consists of two parts: static power Pwr_S and dynamic power Pwr_D. Among them, the static power is fixed when the server is running, while the dynamic power changes with the utilization rate.

$$Pwr_S = \begin{cases} 0 & U_m = 0 \\ const. & U_m > 0 \end{cases} \quad (2)$$

$$Pwr_D = \begin{cases} \alpha * U_m & U_m < \widetilde{U_m} \\ \alpha * \widetilde{U_m} + (U_m - 0.7)^2 \beta & U_m \geq \widetilde{U_m} \end{cases} \quad (3)$$

U_m is the utilization rate of server m, $\alpha = 0.5, \beta = 10$, $\widetilde{U_m}$ is the optimal utilization which is 0.7[22]. And the total power of server m Pwr_m is the sum of static power Pwr_S and dynamic power Pwr_D.

2) Dynamic Price Model

Dynamic electricity prices will vary depending on time and electricity consumption. In reality, it also changes with the region and the power supply company; in our project, threshold is 1.5 kwh, and $T_{lowprice}$ is from 12 pm to 6 pm[23]. And lower price p_l is 5.91 cents/kwh, higher price p_h is 8.27 cents/kwh[24].

The total energy cost is the sum of $P[\,t, Pwr\,]$ during the whole running period.

$$P[\,t, Pwr\,] = \begin{cases} p_l * Pwr & Pwr < threshold \text{ and } t \in T_{lowprice} \\ p_h * Pwr & Pwr \geq threshold \text{ and } t \notin T_{lowprice} \end{cases} \quad (4)$$

3) Reject Rate

In addition to energy cost, the reject rate is also one of our measurements for evaluating system performance. For each task, we have a deadline, if we cannot complete the task with the current resources before the deadline, the task will be rejected. Since the task's required for the virtual machine is mainly on the CPU and RAM, we only consider these two type of resources when we choose the rejected tasks. When a task is overdue, we will reject it immediately, otherwise we will continue to track the CPU and RAM availability of each virtual machine and accept it when possible. We will allocate a task only if the current CPU and memory resources are sufficient for the task.

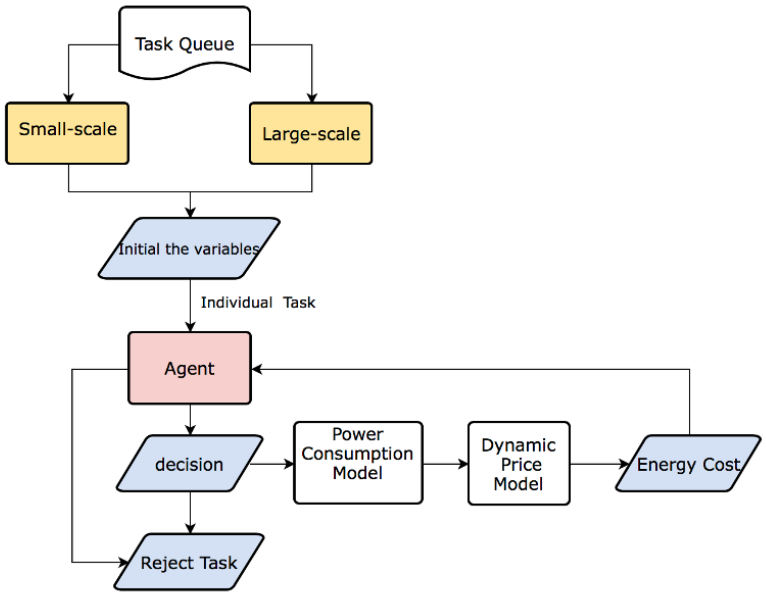

Fig. 1. Flow chart of environment. Task queue is from user workload model which is explain in detail in Section III.

4) Recourse Release

The running time of each task takes up CPU and RAM is not infinite, in reality, it will automatically be released after the end of the process. In order to simulate this process, for each task, we randomly set a running time for it. During that period, the corresponding resource of virtual machine will be occupied by that task. In other words, when agent assigns a task to a server's virtual machine, we can get an ending time which represent the time of task is expected to complete. When the occupied time is over, we change the task's state from "ready" to "finished", and release the corresponding CPU and RAM resources. Through this method, we can dynamically simulate the changes of resources in the server system and help the agent to make the decision.

C. Agent(Deep Q Network)

Figure 2 below shows the workflow of the Two-Stage RP-TS Processor[12,13]. We use two DQNs as agents: in the first stage, task will be assigned to server farms, and in the second stage, task will be allocation to specific servers and virtual machines. In the environment, we track the resource status information of the current cloud server platform, record and update the basic information of the task (resource requirements, running time, dependencies, status) through the user workload model. The agent makes decisions based on the current task and status information, and learns through the reward function.

State Space: In deep reinforcement learning, state space defines the range of possible states that the agent may perceive [9]. In the two-stage decision-making process, the DQN of each stage will obtain the current status information and task information as input from the environment. For stage1's agent, it will get the current resources status information (CPU and memory) of each server farm and individual task information (required resources, task status) , so the state space is the range composed of the above two kinds of information. For stage2, agent will obtain the resources status information of all servers and virtual machines in server farm f and task information (required resources, task status, and deadline). The agents need to make decisions (allocation task) inside the corresponding action space through the information obtained from the environment, and they also learning from the rewards that are from the environment. The environment updates the current cloud server platform resource status and task status according to the agents' decision, and calculates the value of corresponding reward functions.

Through finite steps of exploration and exploitation, the agent will be trained as the decision maker that could generate the long-term optimal solution.

Action Space: Just like state space, action space defines the actions that are available to agents[9]. In stage1, DQN needs to select a server farm from all server farms for the current task. In other words, the agent in stage 1 will make an action in the space which has the same dimension with the number of server farms, so the action space is all server farms, $A_{stage1} = \{serverFarm_1, \dots, serverFarm_n\}$. After the stage1's agent makes the decision about the server farm f, the stage2's agent will select a specific servers and virtual machines in the server farm f, so the action space will be the all servers that set in the server farm f, we can represented the action space of stage 2 DQN as: $A_{stage2} = \{Server^1{}_f, \dots, Server^n{}_f\}$.

Reward Functions: In order to enable our two-stage RP-TS processor to automatically generate long-term optimal solutions under the real-time input through deep Q learning, we need to feedback each step of the decision through the reward function. The following are the reward functions of our two stages decision[13].

$$R_1 = P[t_n, Pwr_F(t_{n-1}) - Pwr_F(t_n)] \quad (5)$$

$$R_2 = P[t_n, Pwr_m(t_{n-1}) - Pwr_m(t_n)] \quad (6)$$

t_n is the current time, and $Pwr_F(t)$ represents the total power of servers in the farm on time t, and $Pwr_m(t)$ represents the power of individual server on time t. When a decision is made, we focus on the power usage change during this process, and we use the energy cost of the power change as the rewards for the agent. The reason of we use two reward functions separately is that when the agent makes the judgment, it first selected the number of server farm, and then the final server is selected in the stage two. And P function is the dynamic electric price model, we use it to calculate the energy cost.

Q-Learning: In order to get the best decision under changing conditions, we use Q-learning for decision learning[25]. It

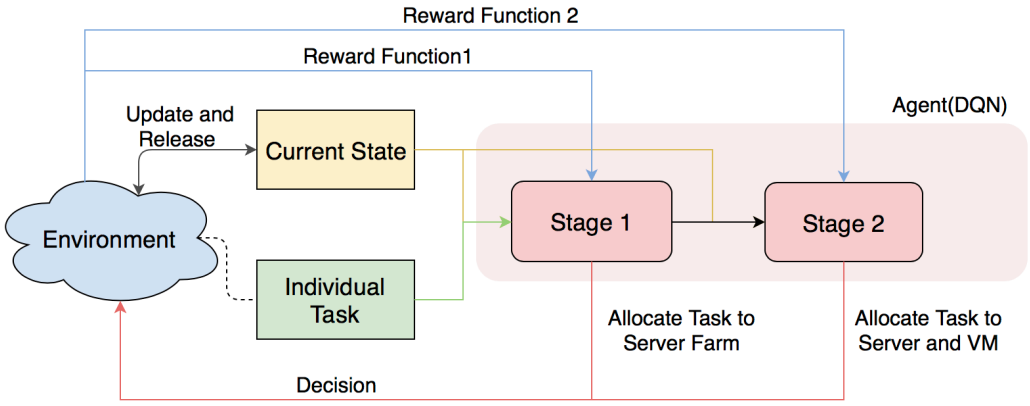

Fig. 2. Two-stage processor based on reinforcement learning.

learns the action-value function Q(s, a): how to take a good action at a particular state. We sample an action from the current state and find out the reward R and the new state. From the memory table, we determine the next action a' to take which has the maximum Q(s', a')[26, 27].

Algorithm 1 Algorithm of Q-Learning

Initial state s, start with Q_0 for all s,a

for all iteration until convergence **do**
  Action a, get next state $s\prime$
  if $s\prime$ is terminal **then**
    $target = R(s, a, s\prime)$
    Sample new initial state $s\prime$
  else
    $target = R(s, a, s\prime) + \gamma \max_{a\prime} Q_k(s\prime, a\prime)$
  end if
  $Q_{k+1}(s, a) \leftarrow (1 - \alpha)Q_k(s, a) + \alpha[target]$
  $s \leftarrow s\prime$
end for

Fig. 3. Algorithm of Q-Learning.

However, if the combinations of states and actions are too large, the memory requirement for Q will be too high and takes very longer time for computation. To overcome that we design a model which directly calculates Q(s, a), this network itself is a deep network. In DQN, we generalize the approximation of the Q-value function rather than remembering the solutions[26].

1) Experience Replay:

This is the most important part of DQN. Basically, in Q-learning we learn from state, action, reward, and next state, make the decision and then discard them. However, there may be many rare experiences which could be very important for our model. In experience replay we store our learnings in a memory buffer and we try to recall those learnings and learn from them[28].

2) Fixed Q-Target:

The main idea of introducing fixed Q targets is that both labels and predicted values are functions of the same weights. We create two deep networks, ω^- and ω. We use the first one to retrieve Q values while the second one includes all updates in the training and for say 10000 we synchronize ω^- and ω. The idea here is to fix the Q-value targets temporarily so we don't chase a moving target[28]. DQN uses Huber loss to improve the performance.

3) Architecture:

It has two important components in its deep neural network and agent. The neural network maps the state space to action space which is basically doing the job of Q table. The agent is like the brain which initiates the DQN with all the parameters which vary depending on type of environment. The important functions of agent are steps—which store the current state, action. The action from the DQN also a random action based on epsilon to explore the new actions, learn—updates the parameters based on batches of experiences. There are more advanced varieties in DQN like double DQN, prioritized experience replays and dueling DQN[27].

III. IMPLEMENTATION OF DRL-BASED RP-TS SYSTEM

In this part, we will introduce the algorithm and implementation of DRL-based RP-TS system. We used Python and C ++ to model the user workload model and environment framework, and implemented our artificial agent—Deep Q-Network(DQN).

A. Data preprocessing

In this part, we downloaded the data from "Google clusterdata-2011-2/tast_events"[1]. Since the data is separated in multiple files, we read data from each file, merge them into one dataframe, drop some data with useless attributes like "missinginfo", "machineID", "eventtype", "username", "schedulingclass" and "different-machine-constraint" and write them to file("input.txt"). Because the size of the dataset is very large, we only slice the part we need through preprocessing, it will minimize the size of the dataset and reduce the running time of following steps.

This part was written in file "preprocess.py" in Python language, and use library/packages "csv", "gzip", "pandas" and "numpy".

B. Build user workload model

Use In this part, we implemented the model using Object-Oriented programming.

We have built 2 classes "Task" and "Job". In the "Task" class, the constructor has 7 input parameters. We get and store information about a task by the constructor. In the "Job" class, we store all tasks in a "map" data structure with the library "map" and "vector"—the key of map is Job Id, and the value of the map is the vector of tasks so that we can put the tasks with the same Job Id together. Also, we override the comparator of vector sorting method for sorting tasks purposes with the library "algorithm". We sort the tasks with the same Job Id individually by task index. Then, we realized a method to write the task information into an output file with the library "iterator".

In the main function, we first defined the input, output files ("input.txt" and "output.txt") with the library "fstream" and a Job object. Then we get the input information line by line and one line represents a Task object. After getting setting each Task object, we add it into the Job object. When we get all the information from the input file and build the Job object, we sorted the Job object in terms of Job Id and task index. At last, we wrote them into the output file.

This part was written in "userWorkload.cpp" in C++ language, with the library "fstream", "map", "vector", "algorithm" and "iterator".

C. Improved user workload model with inter-job task dependencies

In this part, we have improved the user workload model. Because the inter-job dependencies were not provided in the

data set[1] and they exist in the real world, we randomly generated dependencies between the inter-job tasks and built the task queue based on directed acyclic graphs (DAGs).

We did this part in python language and the input of this part is the output of the userWorkload.cpp. First, we defined two classes: Task and DAG. The instance variables and the meaning of them in two classes are at Figure 4, we also set the deadline and occupy time for each task.

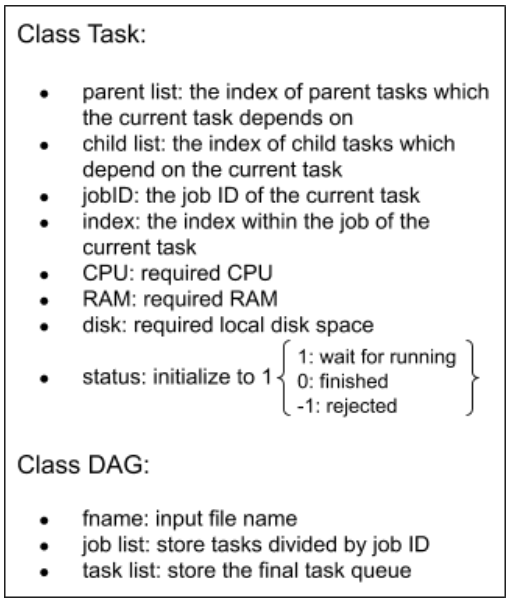

Fig. 4. The attributes of class Task and DAG.

Then, we implemented DAGs in three main steps. The Figure 5 is shown as follows. In step 1, read the input file with the given "fname" line by line. Each line includes job ID, index within the job, CPU, RAM and local disk space information for one task. Create one Task object with the information and store it to the job list.

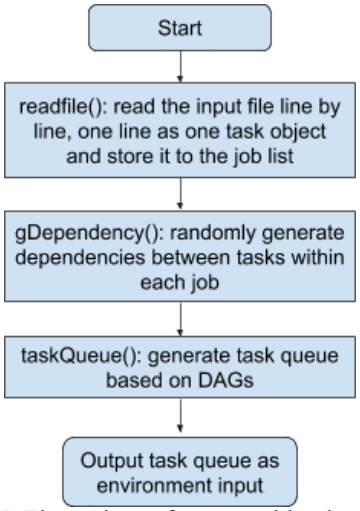

Fig. 5. Flow Chart of user workload model.

In step 2, randomly generate dependencies between tasks within each job. The flow chart(Figure 6, 7) of this step is shown as below which shows how to generate dependencies between tasks in one job and in the program I did this flow for each job. Besides, assign each task a random parent task with probability 0.5 (this probability can be changed) by setting the random range from (-len(job)) to (len(job) - 1).

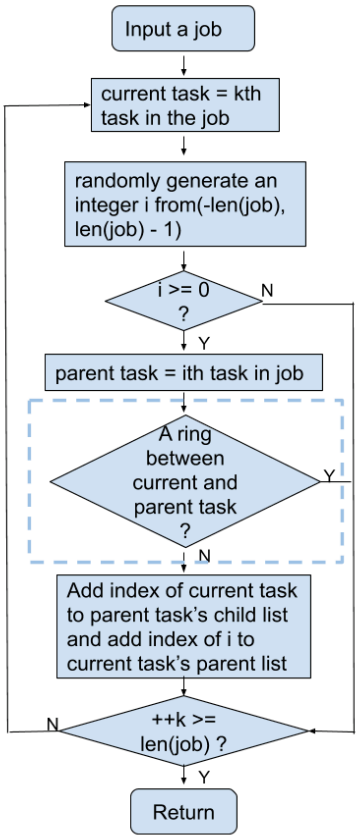

Fig. 6. Flow chart of step 2: randomly generate dependencies between tasks within each job.

The detailed explanation of the blue dotted frame is as follows:

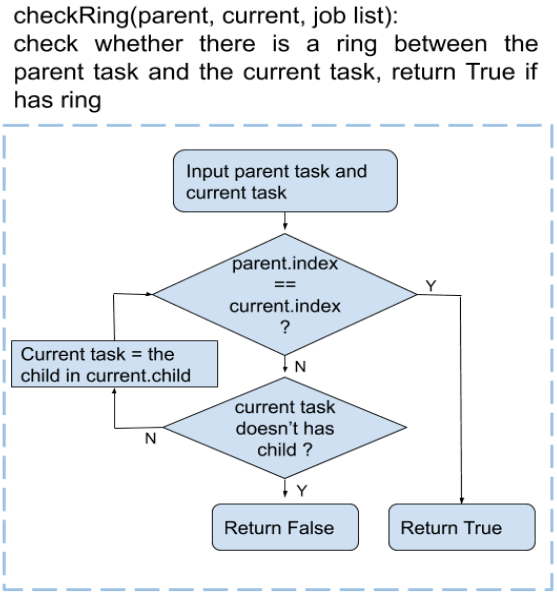

Fig. 7. Check ring inside task dependencies.

In step 3, generate the task queue based on directed acyclic graphs (DAGs). The flowchart(Figure 8, 9) for this part shows how to generate the task queue in one job and in the program I did this flow for each job. For the task in one job, add the tasks without parent tasks limitation to the task list first. Then change the status of these tasks to finished(0) and update the status of their child tasks which depend on them. And repeat the last step until all the tasks in this job add to the task queue. Additionally, in the following training part, once one task is rejected, the updateStatus() function needs to be called for rejecting its child tasks. And if the status of one task is finished(0) or rejected(-1), this task does not need to be trained.

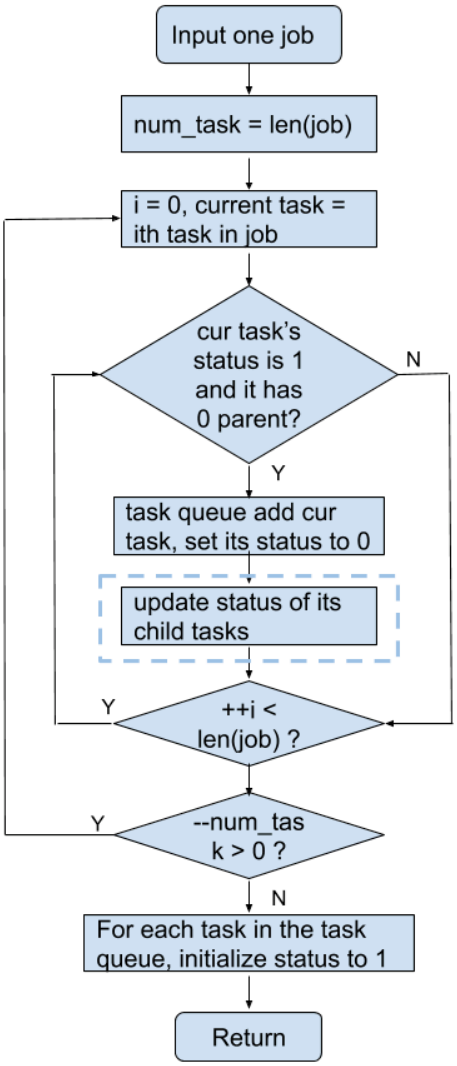

Fig. 8. Flow chart of generate the task queue based on directed acyclic graphs (DAGs).

The detailed explanation of the blue dotted frame is as follows:

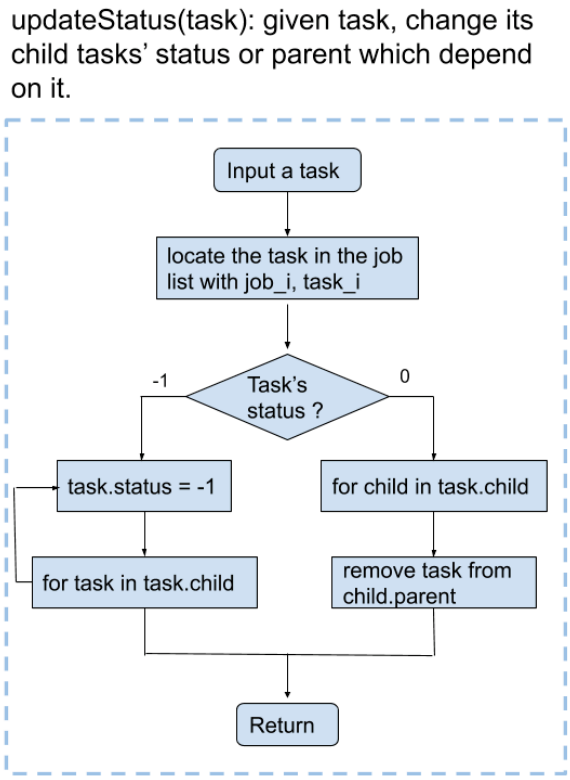

Fig. 9 Flow chart of task status update.

D. *Built environment*

For the framework to simulate the server architecture and resource availability of CSPs, we implement an environment model, which define by a class. The main role of the environment is to pass the individual task of ready task queues that obtained from the user workload model to the agent, and provide the real-time resource status information to the agent at the same time(Figure 1).

In the initialized part, we set the number of virtual machine, server, and server farm based on the scale of user request's(the request scale and server number are inputs that we decide). We use n dimension lists to store the resources usage information of each server farm and each virtual machine. The task is a list of object Task, which is come from the user workload model by class DAG. For status of each VMs, we also define a n dimension list to store the all tasks that currently running on each VMs.

In a real cloud server platform, the servers are connected to each other to form multiple server farms. We simulate that structure through combine some number of servers together. For calculate the energy cost and rewards, we use two lists to count the current power usage for each server and each farm.

For power consumption model, we have a method getPwr(), it follow the equation 1~3 to calculate the normalized total power usage. The rewardFcn1() and rewardFcn2() are the reward functions of stage1 and stage2, which correspond to equations 5 and 6, respectively. To reduce the computing time, we focus on the changes of the server that tasks are assigned before and after resource allocation, and update the

corresponding server's power usage. We also have an elecPrice() for dynamic price model(equation 4). All parameters and equations are based on the related paper[13, 21-24].

In the real world, the completed task will automatically terminated and release resources. In the environment model, we use a function releaseByTime() to simulate this process. After the agent decide the server farm, server, and virtual machine, we will call releaseByTime() to release the resources of the completed task. In the releaseByTime(), we check the task list in the virtual machine, when the estimated completion time of a task is earlier than the current time, it means that task has been completed, so we change the task status to completed and release the corresponding resource. After this, we check the feasibility of the current task by checkRej().

For every task, we will check whether it can be completed under the current resource(CPU and memory) conditions before the deadline, so in checkRej() we focus on the estimated completion time of the task. If the deadline is exceeded, we will reject the task and update the status of tasks, otherwise it will wait until the remaining resources are sufficient to complete the task or the deadline is exceeded.

In trainDQN(), which is the interface with our agent(DQN). As Figure 2 shows, there are two DQN agents that will allocation task to server farm and server during two-stage decision-making process. First we will initialize the DQN-stage 1 and DQN-stage2, then use the ready task queue we got from user workload model as input. When the task queue is not empty, we will pass the individual task and the current resources status information to the DQN, and let the DQN to make the decision. The DQN-stage 1 will assign task to a server farm f, and DQN-stage 2 will choose a server s in the server farm f, after that, we randomly choose a virtual machine on that server s. After we got the decision, we check the feasibility of the task, if it is feasible, we allocate the corresponding resources, set the estimated completion time of the task is current time + running time. Else we change the task status to -1 and add the task to reject task list. And we will also call the corresponding reward function to provide the rewards for DQN. Throughout the process, we will dynamically update task status and maintenance ready task queue, so our agents will loop until all tasks are terminated (completed or rejected). After that, it will output the total energy cost and reject rate as output.

For update the corresponding resources, we have a function UpdateServerFarmState(). It will update the resource usage information of server farms and virtual machines after a task is allocation.

This part was written in file "env_DQN.py" in Python language, we use "time" and "random" libraries to get current time and generate random number.

Algorithm 2 Algorithm for Two-Stage RP-TS Processor
Input: ready tasks
Output: energy cost, reject rate, and run-time
Initialize environment, DQN-stage1, and DQN-stage2
Generate ready task queue
while task queue is not empty **do**
  for each task in ready task queue **do**
    Run DQN-stage1 and assign task to server farm
    Run DQN-stage2 and assign task to server
    Assign task to VM
    Release resources for completed tasks
    if task is not reject **then**
      Allocation resources for the task
      DQN-stage1 learns from reward 1
      DQN-stage2 learns from reward 2
    end if
    Update task status
  end for
  Update task queue
end while
return total energy cost and reject rate

Fig. 10. Algorithm of Two-Stage RP-TS Processor

E. Built DQN

The DQN model was built using the package PyTorch[29], we have used Linear as a layer for both the input and output Q-values. The features of the model optimizer is Adam, for loss is mean square error, the device is CPU if the system has GPU, it will use it. The range of epsilon is [1.0, 0.01], gamma was chosen as 0.99, learning rate=0.0001.

The DQN has been tested with a dummy environment but with real-time input data which was extracted from the Google Cluster and modified in phase I. However the action space, rewards were generated randomly. The final part to interact with the real environment will be done in phase III.

The method extractData does the extraction of data from the "output.txt" (generated in phase I). The class LinearDeepQNetwork builds a neural network of the above-described features which takes a state as input and returns Q-values of actions. The class Agent initializes the neural network and chooses the action with the highest Q-value. In the method learn() the state, action, reward, next_state will be fed to two networks evaluation and the target network. Q_target, loss, optimizer, and decrement_epsilon are updated here at every step. The methods getReward, getNextState are used to randomly generate reward and next state.

This part was written in file "DQN_skeleton.py" in Python language.

F. Baseline

For the baseline, we have already implemented the Round-Robin method[30] which is in the class environment "RR" function. In this method, we arranged the task server by server and VM by VM. The flowchart(Figure 11) is for processing one batch of tasks in a task queue who have no parent tasks or whose

parent tasks were all finished. After dealing with this task queue, generate a new batch of tasks who are not rejected and whose parent tasks are all finished. And do these until all tasks have been processed.

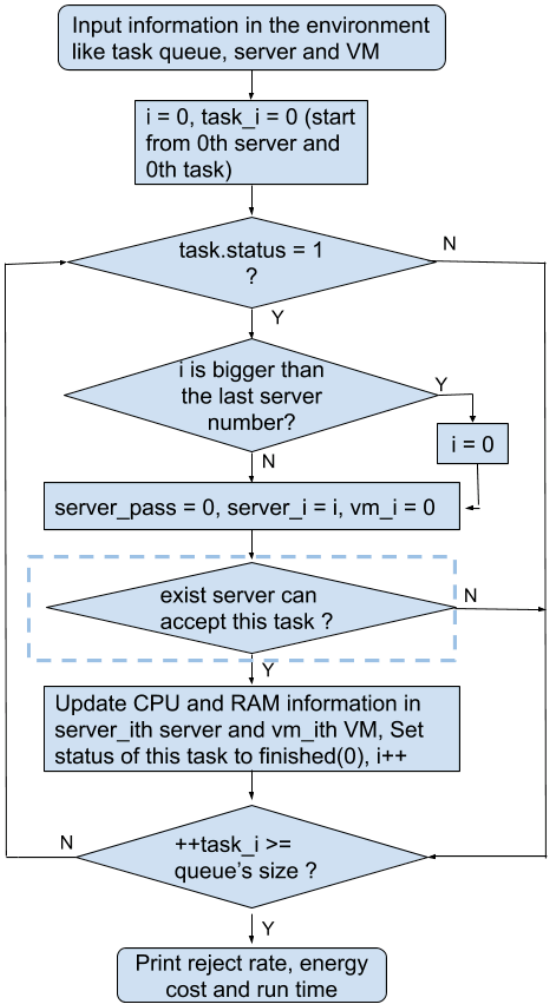

Fig. 11. Flow chart of baseline: Round-Robin method.

From server_ith server to (server_i-1)th server in a cycle, search from its 0th VM to the last VM to find which VM can accept this task. Before checking whether this VM can hold the task, release tasks which have been finished in this VM first. If there exists a VM which can take this task, assign current system time as the start time to this task and check SLA, if time condition follows SLA, return VM index and server_i and. Otherwise, reject this task immediately, record the reject number and move to the next task. Once a task was assigned to a VM or rejected, change the status of it and pop it from the task queue.

IV. Experiment Results

We test our model with 1000 to 5000 user requests and 100 to 300 servers that are clustered into 10 server farms on Round-Robin Method[30] and our agent(DQN). We use the parameters and data sets described above and get the following results:

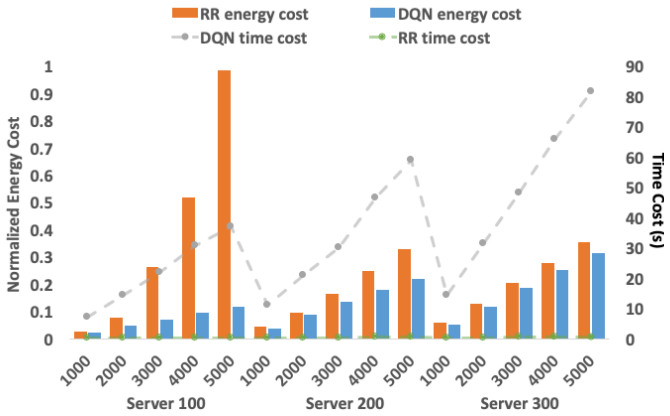

Fig. 12. Normalized energy cost and time cost of baseline and agent(DQN).

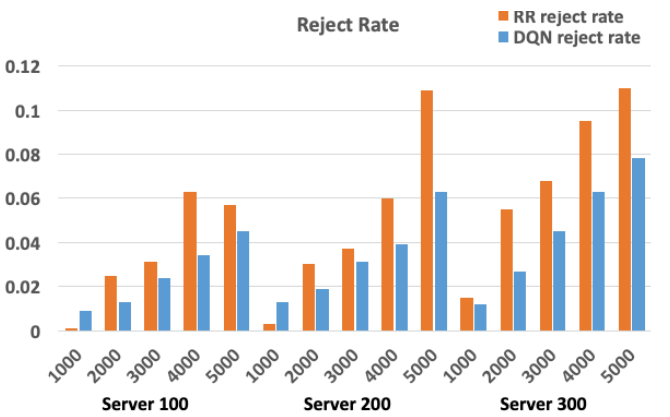

Fig. 13. Reject rate of baseline and agent(DQN).

We also test RR method on original environment and improved environment on same measurements. The result showing in Figure 14, 15.

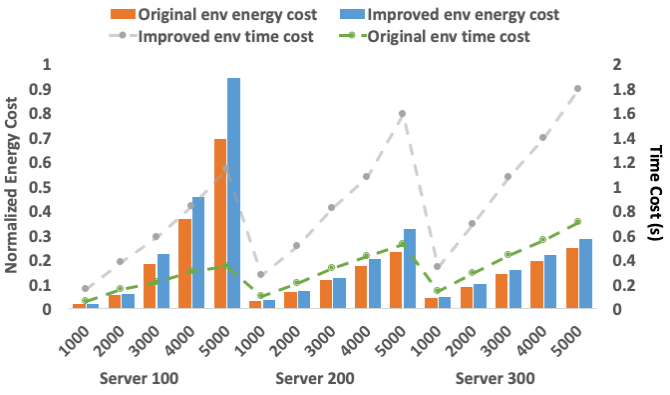

Fig. 14. Normalized energy cost and time cost of environment and improved environment.

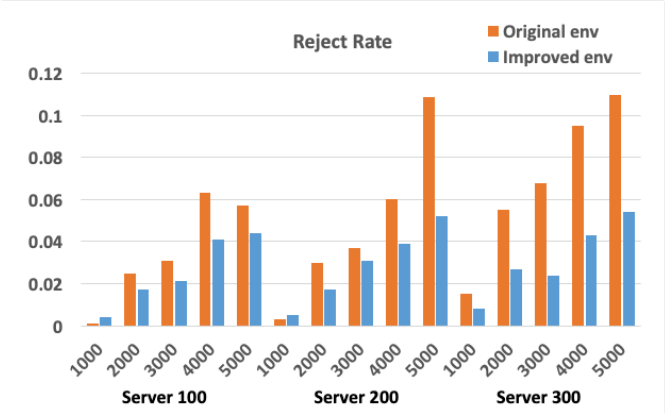

Fig. 15. Reject rate of environment and improved environment.

V. DISCUSSION

A. Comparison Between Baseline and DQN Agent

The Figure12 and Figure 13 are the graphs for the results of DQN agent and round-robin. The above results are for the 100 - 300 servers in 10 farms with 1000 - 5000 tasks.

Time cost analysis: DQN takes significantly more time than RR when the numbers of tasks and servers increase. This is because it has to select an experience batch from its experience buffer and has to calculate the q values for all of them and select the best one, which will cost long time. On contrary to this, RR just allocate tasks in order.

Energy cost Analysis: We are more concerned for the energy cost, interestingly energy cost for the DQN is much lower than RR. This is because DQN agent learns from the reward functions and keeps selecting the server farm and server that maximum reward which in our case is the one with low power consumption. The energy cost for the DQN can be higher in the beginning but it will reduce very low as it gets trained. For different server numbers, the energy cost saved in DQN are different. When the server number is 100, there is up to 900% energy cost saving, while when the server number is 300, there is only up to 120% saving. And the average energy cost saving is about 220%. The energy cost changes slowly with different server numbers, so the stability of DQN is much better than RR.

Reject rate: we can see that the reject rate for RR is almost twice as DQN agent. The reject rate in RR is higher and increases more rapidly than DQN. This is because when there are more tasks fed to DQN, it learns from the experiences and gets better and better. RR allocate all tasks in order, thus when the number of tasks increase, more tasks will be rejected due to running out of time or resources.

B. Comparison Between Original Environment and Improved Environment

In the original environment, one task only can run on one VM and in the improved environment, one task can be assigned to different servers. In Figure 14 and Figure 15, "Original_env" represents that tasks run in the original environment and "Improved_env" means tasks assigned in the improved environment.

From Figure 14, the data of energy cost and time cost for tasks in the original environment is lower than that for tasks in the improved environment. However, in Figure 15, the reject rates in the improved environment are roughly half of those in the original environment. The reason may be that when we divided one task into several subtasks, we just treat sub-tasks as one task with much smaller required CPU and RAM. Then one VM can hold more tasks with a higher utilization rate and more energy is consumed. And at the same time, instead of arranging one task once, the model should assign its subtasks for several times which depends on how many subtasks it has. Therefore, the time cost must be higher in the improved environment than in the original environment.

Besides, when the server number is 300, the energy cost in the improved environment is very close to that in the original environment, while the reject rate in the improved environment is twice as that in the original environment, so in this condition, the improved environment has a better performance.

REFERENCES

[1] Google cluster data. [Online]. Available: https://github.com/google/cluster-data

[2] J. W. Rittinghouse and J. F. Ransome, Cloud computing: implementation, management, and security. CRC press, 2016.

[3] B. Hayes, "Cloud Computing," Communications of the ACM, 2008.

[4] A. Shehabi et al., "United States Data Center Energy Usage Report", 2016. Available: 10.2172/1372902.

[5] S. Srikantaiah et al., "Energy aware consolidation for cloud computing," Cluster Computing, 12(1):1–15, 2009.

[6] C. Kozyrakis, "Resource efficient computing for warehouse-scale datacenters," in Design, Automation & Test in Europe Conference & Exhibition (DATE), 2013, pp. 1351–1356, IEEE, 2013.

[7] V. François-Lavet, P. Henderson, R. Islam, M. Bellemare and J. Pineau, "An Introduction to Deep Reinforcement Learning", Foundations and Trends® in Machine Learning, vol. 11, no. 3-4, pp. 219-354, 2018. Available: 10.1561/2200000071.

[8] M. Hagan, M. Beale and H. Demuth, Neural network design. Boston: PWS Pub., 1999.

[9] V. Mnih et al., "Human-level control through deep reinforcement learning", Nature, vol. 518, no. 7540, pp. 529-533, 2015. Available: 10.1038/nature14236.

[10] D. Silver et al., "Mastering the game of go with deep neural networks and tree search," Nature, vol. 529, no. 7587, pp. 484–489, 2016.

[11] J. Gao and R. Evans. Deepmind ai reduces google data centre coolingbill by 40%. https://deepmind.com/blog/deepmind-ai-reduces-google-data-centre-cooling-bill-40/.

[12] 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. Available: 10.1145/3005745.3005750.

[13] M. Cheng, J. Li and S. Nazarian, "DRL-cloud: Deep reinforcement learning-based resource provisioning and task scheduling for cloud service providers", 2018 23rd Asia and South Pacific Design Automation Conference (ASP-DAC), 2018. Available: 10.1109/aspdac.2018.8297294.

[14] Z. Tong, H. Chen, X. Deng, K. Li and K. Li, "A scheduling scheme in the cloud computing environment using deep Q-learning", Information Sciences, vol. 512, pp. 1170-1191, 2020. Available: 10.1016/j.ins.2019.10.035 [Accessed 28 February 2020].

[15] D. Silver, "Deep Reinforcement Learning", Deepmind, 2016. [Online]. Available: https://deepmind.com/blog/article/deep-reinforcement-learning.

[16] S. Xavier and S. J. Lovesum, "A survey of various workflow scheduling algorithms in cloud environment," International Journal of Scientific and Research Publications, 3(2), 2013.

[17] M. Isard et al., "Dryad: distributed data-parallel programs from sequential building blocks," in ACM SIGOPS operating systems review, vol. 41, no. 3. ACM, 2007, pp. 59–72.

[18] D. Warneke and O. Kao, "Nephele: efficient parallel data processing in the cloud," in Proceedings of the 2nd workshop on many-task computing on grids and supercomputers. ACM, 2009, p. 8.

[19] R. Chen et al., "On the efficiency and programmability of large graph processing in the cloud", Technical Report MSR-TR-2010-44, Microsoft Research, 2010.

[20] J. Li et al., "Negotiation-based resource provisioning and task scheduling algorithm for cloud systems," in Quality Electronic Design (ISQED), 2016 17th International Symposium on. IEEE, 2016, pp. 338–343.

[21] M. Cheng, J. Li, P. Bogdan and S. Nazarian, "H2O-Cloud: A Resource and Quality of Service-Aware Task Scheduling Framework for Warehouse-Scale Data Centers", IEEE Transactions on Computer-Aided Design of Integrated Circuits and Systems, 2019. Available: 10.1109/tcad.2019.2930575.

[22] Y. Gao et al., "An energy and deadline aware resource provisioning, scheduling and optimization framework for cloud systems," in Hardware/Software Codesign and System Synthesis. IEEE, 2013, pp. 1–10.

[23] J. Abushnaf, A. Rassau, and W. Go´rnisiewicz, "Impact of dynamic energy pricing schemes on a novel multi-user home energy management system," Electric power systems research, vol. 125, pp. 124–132, 2015.

[24] A.-H. Mohsenian-Rad and A. Leon-Garcia, "Optimal residential load control with price prediction in real-time electricity pricing environments," IEEE transactions on Smart Grid, vol. 1, no. 2, pp. 120–133, 2010.

[25] S. J. Bradtke and M. O. Duff, "Reinforcement learning methods for continuous-time markov decision problems," in Advances in neural information processing systems, 1995, pp. 393–400.

[26] J. Hui, "RL — DQN Deep Q-network," Medium, Mar. 26, 2019. https://medium.com/@jonathan_hui/rl-dqn-deep-q-network-e207751f7ae4 (accessed Apr. 10, 2020).

[27] "Lec3-DQN.pdf," Google Docs. https://drive.google.com/file/d/0BxXI_RttTZAhVUhpbDhiSUFFNjg/view?usp=drive_open&usp=embed_facebook (accessed Apr. 10, 2020).

[28] U. Singh, "Deep Q-Network with Pytorch," Medium, Mar. 18, 2019. https://medium.com/@unnatsingh/deep-q-network-with-pytorch-d1ca6f40bfda (accessed Apr. 10, 2020).

[29] A. Paszke, "Reinforcement Learning (DQN) Tutorial — PyTorch Tutorials 1.5.0 documentation", Pytorch.org, 2020. [Online]. Available: https://pytorch.org/tutorials/intermediate/reinforcement_q_learning.html. [Accessed: 04- May- 2020].

[30] "Round-robin scheduling", En.wikipedia.org. [Online]. Available:https://en.wikipedia.org/wiki/Round-robin_scheduling. [Accessed: 29- Mar- 2020]. *30

[31] "Data parallelism", En.wikipedia.org, 2020. [Online]. Available: https://en.wikipedia.org/wiki/Data_parallelism. [Accessed: 27- Apr- 2020].