DeepJS: Job Scheduling Based on Deep Reinforcement Learning in Cloud Data Center

Fengcun Li
State Key Laboratory of Networking and Switching Technology, Beijing University of Posts and Telecommunications
No.10 Xi Tu Cheng Road
Beijing, China
lfc@bupt.edu.cn

Bo Hu
State Key Laboratory of Networking and Switching Technology, Beijing University of Posts and Telecommunications
No.10 Xi Tu Cheng Road
Beijing, China
hubo@bupt.edu.cn

ABSTRACT
Job scheduling is a key building block of a cloud data center. Hand-crafted heuristics cannot automatically adapt to the change of the environment and optimize for specific workloads. We present the DeepJS, a job scheduling algorithm based on deep reinforcement learning under the framework of the bin packing problem. DeepJS can automatically obtain a fitness calculation method which will minimize the makespan (maximize the throughput) of a set of jobs directly from experience. Through a trace-driven simulation, we demonstrate the convergence and generalization of DeepJS and the essence of DeepJS learning. The results prove that DeepJS outperforms the heuristic-based job scheduling algorithms.

CCS Concepts
• **Computer systems organization → Cloud computing**

Keywords
Job Scheduling; Deep Reinforcement Learning; Bin Packing Problem; Cloud Data Center

1. INTRODUCTION
Job scheduling is a key building block of a cloud data center. Efficient utilization of computing cluster matters for enterprises. A good scheduling policy packs work tightly to reduce fragmentation and increase the throughput. Resource management is a difficult online decision-making problem where appropriate solutions rely on understanding workloads and environments.

In order to solve the problem of resource management, a lot of sophisticated heuristics have been designed, such as fair scheduling[2][3], first-fit[14], simple packing strategies[9]. These heuristics prioritize generality, ease of understanding, and straightforward implementation over achieving the ideal performance on a specific workload. In order to achieve good performance, these heuristics must be carefully tested and adjusted in practice. This process usually has to be repeated if certain aspects of the problem, such as the characteristics of the workload or the optimization goals of resource management change.

As reinforcement learning agent can learn decision-making policy directly from experience and adapt to the dynamic environment, reinforcement learning is well-suited to resource management systems. In the public cloud which provides IaaS service, reinforcement learning has been applied in virtual machine auto-configuration[5][6] and making virtual machine online migration decisions[7][8]. There are some attempts to design job scheduling algorithms in the cloud data center based on reinforcement learning[4][11]. DeepRM[11] represents the cluster resource state and job resource requirements as images and uses a reinforcement learning agent to make job scheduling decisions. Instead of treating the cluster as a single collection of resources and ignoring machine fragmentation effects, the reinforcement learning agent proposed in [4] works in a multi-resource multi-machine environment in which resources in different machines are represented by different channels in the neural network input layer. Since the number of neural network input channels must be fixed, the number of machines is also assumed to be static. In practice, machines become offline now and then due to hardware or software faults in large scale cluster.

In this paper, we present the DeepJS[1], a job scheduling algorithm based on deep reinforcement learning. Instead of directly using deep reinforcement learning model for job scheduling[4][11], DeepJS is embedded in the framework of the bin packing problem. In solving a bin packing problem, fitness is calculated which describe how a machine and a task is matched in order to maximize the throughput[9]. An optimal calculation method of fitness is the key to optimally solve the bin packing problem. DeepJS will automatically obtain a fitness calculation method which will minimize the makespan of a set of jobs through deep reinforcement learning. In the experiments, we replay a workload from production to verify the convergence and generalization of the algorithm and explores the nature of the agent learning. The simulation results show that the proposed algorithm reduces the makespan.

The rest of this paper is organized as follows. Section 2 motivates the presentation of DeepJS. Section 3 introduces the system model of DeepJS. Section 4 provides experimental results based on traces from an Alibaba cluster. We conclude in Section 5.

2. MOTIVATION
The job scheduling problem is analogous to multi-dimensional bin packing when considering multiple resource types. Given the size of the ball and box in the R_d space, where d is the number of resource types, multi-dimensional bin packing assigns the balls to the fewest number of bins. Doing so maximizes the number of simultaneously scheduled tasks, thus minimizing makespan. This is an APX-Hard problem.

Due to the high computational complexity, solving the bin packing problem generally requires the use of heuristic algorithms, such as the first-fit and best-fit algorithms, which are fast but often do not have an optimal solution.

Tetris[9] is presented to solve the online bin packing problem. Tetris projects task requirements t_r and machine available resource m_r into Euclidean space and picks the <

[1] The source code of DeepJS can be find here. https://github.com/RobertLexis/CloudSimPy/

$task, machine >$ pair with the highest dot product value. And only the tasks whose requirements are satisfiable are considered. Using dot product to calculate the fitness prefers large tasks and tasks that require resources in proportions similar to what is available. Dot product may seem to be reasonable but not necessarily be optimal. In industry, some even simpler fitness calculation methods are applied[12].

These handcrafted calculation methods are general for the workload. Although some parameters can be adjusted to adapt to workloads with different characteristics, the adjustment of the parameters relies on the operator's experience. Due to the complexity of the relationship between parameters and optimization goals, the process is cumbersome and the improvement cannot be guaranteed.

With reinforcement learning, we only need to design a suitable reward function that reflects the goal of performance optimization. The reinforcement learning model can automatically learn the fitness calculation method for a specific workload.

3. SYSTEM MODEL

3.1 Reinforcement Learning

The model of reinforcement learning is shown in Figure 1. At each time step t, the agent observes the state s_t and makes the action a_t. The state of the environment is transferred from s_t to s_{t+1}, and the agent is given a reward r_t. The state transition of the environment and the rewards obtained have Markov property, that is, the probability of state transition and the rewards obtained depend only on the state of the environment s_t and the action a_t, despite the state and action before the time t. Through interaction with the environment, the agent can observe these quantities during the training process. The goal of the agent is to maximize the expectation of cumulative reward $\mathbb{E}[\sum_{t=1}^{T} r_t]$.

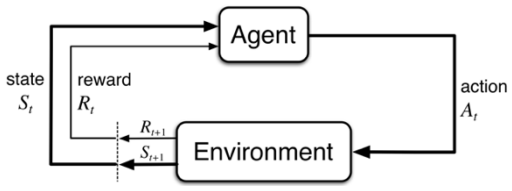

Figure 1 Reinforcement Learning Paradigm

In reinforcement learning, the policy based on which the agent makes decisions is defined as the probability distribution $\pi(a \mid s)$ in the action space under the condition of the environment state s. In many practical problems, there will be many $\{state, action\}$ pairs, so it is not feasible to store the probability distribution in a table, instead it common to use an approximator. An approximator is a function which has a certain number of parameters θ that can be adjusted, so the policy can be represented as $\pi_\theta(a \mid s)$. There are many forms of approximators to choose from. Recently, Deep Neural Network has been successfully applied as an approximator in many large-scale reinforcement learning problems[13].

We adopt a fully connected neural network as the brain of DeepJS and use policy gradient to train its learnable parameters. Let $\mathcal{J}(\theta)$ be the expectation of cumulative reward and τ be a trajectory that the agent experienced in one episode. We have the following formula:

$$\mathcal{J}(\theta) = \mathbb{E}_{\tau \sim p_\theta(\tau)}[r(\tau)]$$

in which:

$$\begin{aligned} p_\theta(\tau) &= p(s_1, a_1, \cdots, s_T, a_T) \\ &= p(s_1)\pi_\theta(a_1 \mid s_1)\prod_{t=2}^{T} p(s_t \mid s_{t-1}, a_{t-1})\pi_\theta(a_t \mid s_t) \end{aligned}$$

$$r(\tau) = r(s_1, a_1, \cdots, s_T, a_T) = \sum_{t=1}^{T} r_t = \sum_{t=1}^{T} r(s_t, a_t)$$

In order to perform gradient descent, we calculate the gradient of $\mathcal{J}(\theta)$:

$$\begin{aligned} \nabla_\theta \mathcal{J}(\theta) &= \nabla_\theta \int p_\theta(\tau) r(\tau) \mathrm{d}\tau \\ &= \int p_\theta(\tau) \nabla_\theta \log p_\theta(\tau) r(\tau) \mathrm{d}\tau \end{aligned}$$

In practice, the expectation can be approximated by the average on the N trajectories using Monte Carlo Method:

$$\begin{aligned} \nabla_\theta \mathcal{J}(\theta) &\approx \frac{1}{N}\sum_{i=1}^{N} \nabla_\theta \log p_\theta(\tau_i) r(\tau_i) \\ &= \frac{1}{N}\sum_{i=1}^{N} (\sum_{t=1}^{T} \nabla_\theta \log \pi_\theta(a_{i,t} \mid s_{i,t}))(\sum_{t=1}^{T} r(s_{i,t}, a_{i,t})) \end{aligned}$$

As the gradient usually has high variance, we can introduce the causality, the discount factor, and baseline:

$$\nabla_\theta \mathcal{J}(\theta) \approx \frac{1}{N}\sum_{i=1}^{N}\sum_{t=1}^{T} \nabla_\theta \log \pi_\theta(a_{i,t} \mid s_{i,t})(\sum_{t'=t}^{T} \gamma^{t'-t} r(s_{i,t'}, a_{i,t'}) - b_t)$$

There are many different kinds of baseline, such as state-dependent baseline $V_\pi(s_t)$, etc. This paper uses a simple average as the baseline. For the specific calculation method, see section 3.2.5.

3.2 Algorithm Design

We consider d resource types such as CPU, memory, storage, bandwidth, and so on. The job arrives online and enters the job queue waiting for scheduling. Just like in industry, each job has one or more tasks; each task has multiple instances; all instances within a task execute the same binary with same resource requirement but different input. An instance is the smallest scheduling unit. The scheduler selects one or more tasks at a schedule time step and dispatches their instances to the m machines in the computing cluster.

A profile of task i is given by its resource requirement vector $r_i = (r_{i,1}, \cdots, r_{i,d})$ and its duration T_i. Once be scheduled onto a machine the task instance from the task i will occupy the resources r_i for T_i time steps until the task instance is completed. The available resource vector for machine j in the computing cluster is $a_j = (a_{j,1}, \cdots, a_{j,d})$, and a machine j can accommodate task i's instance only if the following conditions are met:

$$a_{j,k} \geq r_{i,k} \qquad \forall k \in [1, d]$$

We refer to machine j fit to task i, and (j, i) as fit machine-task pair. The agent will choose the machine-task pair which has the highest fitness from fit machine-task pair list as a scheduling decision.

3.2.1 State Space

We use the variable-length fit machine-task pair list to represent the environment state. Before each scheduling, according to the available resources of each machine in the computing cluster and

the resource requirements of each task, the agent will get fit machine-task pair list. As the scheduling progresses, the length of the fit machine-task pair list may change.

3.2.2 Action Space

At a certain time, there are N pending tasks and M machines in the cluster. The agent may schedule any subset of the tasks to any subset of the machines. The size of the action space is $(M + 1)^N$ (1 means that the agent decides to not schedule this task to any machine, and leave it pending), such a large action space will make the learning of the agent extremely difficult. By allowing the agent to perform multiple scheduling actions in sequence on each scheduling time step, the action space can be linear to the length of fit machine-task pair list, that is, the size of the action space is $M \times N$.

At each scheduling time step, the time does not elapse, and the agent continuously makes scheduling decisions until the fit machine-task pair list is empty, then performs a $None$ action (only for indicating the end of a scheduling time step, which is convenient for giving agent reward signal). After that the agent sleeps; the agent is reactivated in the next scheduling time step, try to make scheduling decisions. The scheduling process is as shown in Algorithm 1.

Algorithm 1 Agent Scheduling Process

```
while True do
    pair_list ← get_pair_list()
    while pair_list ≠ ∅ do
        pair ← select_best_pair(pair_list)
        machine ← pair.machine
        task ← pair.task
        schedule task to machine
        pair_list ← get_pair_list()
    end while
    sleep for one timeout
end while
```

Algorithm 1 Agent Scheduling Process

3.2.3 Rewards

In order for the agent to learn effectively and obtain an effective scheduling policy that optimizes a specific performance metric, we need to design a reasonable reward signal to guide the learning process of the agent. Different optimization goals require different reward signals. In order to minimize the makespan, -1 can be given as reward at each time step until all the jobs are completed. The agent will try to maximize the expectation of the cumulative reward, thereby achieve the optimization goal of minimizing makespan.

It should be noted that the agent will get only one reward signal in one time step, and the reward is given when the agent takes the $None$ action (ie, the fit machine-task pair list is empty, and this scheduling time step ends).

3.2.4 Agent Design

We use a fully connected neural network as the brain of the agent. As we can see from section 3.2.1, the environment state is represented by a variable-length fit machine-task pair list, but the input to the fully connected neural network must be fixed-dimensional.

We let each item (a fit machine-task pair, is fixed in dimension) from variable-length fit machine-task pair list pass through the neural network separately to calculate the fitness between the machine and task. After obtaining the fitness for all fit machine-task pairs in the list, the agent chooses the fit machine-task pair with the highest fitness as the scheduling decision. In actual implementation, each fit machine-task pair does not need to pass through the neural network separately, but all fit machine-task pairs can pass through the neural network as a batch so that all fitness can be quickly obtained through one neural network forward propagation. Common deep learning frameworks support variable length in the batch dimension. The design of the agent is shown in Figure 2.

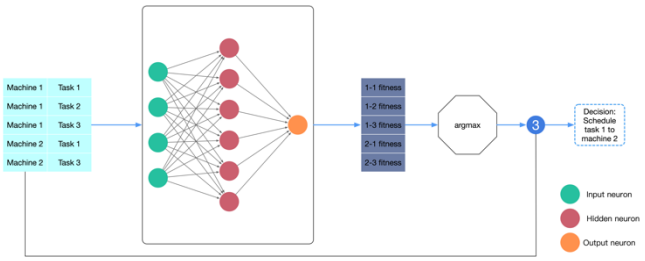

Figure 2 Agent Design

3.2.5 Training Algorithm

We train the neural network in an episodic manner. Algorithm 2 gives the training process of the neural network. In each episode, a certain number of jobs arrive in online mode, and the agent makes scheduling decisions. When all jobs are finished, the episode ends. In an episode, the environment state perceived by the agent s_i, the action a_i, the reward r_i form a trajectory $[s_1, a_1, r_1, \cdots, s_L, a_L, r_L]$, where L is the number of decisions made by the agent. In order to train a generalized scheduling policy, we generate a lot of job arrival sequence and train model on each sequence for several iterations. And in each iteration, we perform N simulations to get N trajectories.

Algorithm 2 Training Algorithm

$\alpha \leftarrow 0.001$
$job\_sequence \leftarrow generate\_job\_sequence()$
for each iteration **do**
  for $i \leftarrow 1, N$ **do**
    $\tau_i \leftarrow [s_1^i, a_1^i, r_1^i, \cdots, s_{L_i}^i, a_{L_i}^i, r_{L_i}^i]$
    for $t \leftarrow 1, L_i$ **do**
      $q_t^i \leftarrow \sum_{t'=t}^{L_i} r_{t'}^i$
    end for
  end for
  $L \leftarrow max(L_1, \cdots, L_N)$
  for $t \leftarrow 1, L$ **do**
    $b_t \leftarrow \frac{1}{N}\sum_{i=1}^{N} q_t^i$
  end for
  for $i \leftarrow 1, N$ **do**
    $\Delta\theta \leftarrow 0$
    for $t \leftarrow 1, L_i$ **do**
      $advantage_t^i \leftarrow q_t^i - b_t$
      $\Delta\theta \leftarrow \Delta\theta + \nabla_\theta log\ \pi_\theta(a_t^i | s_t^i)\ advantage_t^i$
    end for
    $\theta \leftarrow \theta + \alpha\Delta\theta$
  end for
end for

Algorithm 2 Training Algorithm

We use N trajectories from one iteration to calculate the baseline. The baseline value at a decision point is the average of the cumulative rewards at the same index in these N different trajectories (a similar approach was taken in [11]). The lengths of the different trajectories in the same iteration may also be different. We pad zeros at the end of the cumulative rewards sequence of the shorter trajectory.

4. Evaluation

We evaluate DeepJS via trace-driven simulation using the log data Alibaba Cluster Data V2017 [1] from Alibaba's production computing cluster.

4.1 Setup

The log data Alibaba Cluster Data V2017 is replayed to train the agent. In this dataset, 5,216 jobs (31,756 tasks, 2,551,075 task instances) arrived in the online mode [10]. In order to compare the performance of the algorithms, we made the resources of the cluster tenser. In our simulation, the cluster has 5 machines. Each machine has 64 CPU cores and 1 unit memory.

The input dimension of the neural network is 6, which corresponds to the available CPU and memory of a machine, the CPU and memory requirements of a task, the duration of that task, and the number of task instances that have not been scheduled yet of that task.

We use the neural network shown in Table 1.

Table 1 Trace-driven Simulation Neural Network

layer(type)	output shape	parameter #	activation function
Input layer	(None, 6)	0	None
Hidden layer 1	(None, 3)	$6 \times 3 + 3$	tanh
Hidden layer 2	(None, 9)	$3 \times 9 + 9$	tanh
Hidden layer 3	(None, 6)	$9 \times 6 + 6$	tanh
Output layer 4	(None, 1)	$6 \times 1 + 1$	None

4.2 Convergence and Generalization

We divide these 5216 jobs into chunks which each has 10 consecutive jobs along the time axis as shown in Figure 3. DeepJS will be trained on each chunk from left to right, and only 120 trajectories are generated for each chunk. These 120 trajectories are from 10 iteration and 12 trajectories per iteration as described in Algorithm 2. Before training on each chunk, the makespan of the scheduling solution on that chunk given by DeepJS will be recorded. This ensures that each recorded makespan is on the job chunk that the DeepJS has never seen. We trained DeepJS in this sliding manner on the first 100 job chunks. It is worth noting that each job chunk contains a different number of tasks, and the number of task instances per task is also inconsistent. That is, the workload changes over time.

Figure 3 Job Chunks

We define the makespan difference as the makespan of other algorithms minus the makespan of DeepJS, which corresponds to the reduction in makespan.

The makespan difference between different algorithms is shown in Figure 4. As DeepJS sliding to the right along the time axis, the number of job chunks seen by it is increasing, and the number of training iterations is also accumulated. The scheduling solution given by DeepJS is becoming stably superior to other algorithms. After the dashed line in Figure 4, except the job chunks in the dashed circle, DeepJS's solutions on other job chunks are superior to other algorithms. The convergence and generalization of DeepJS are proved. Note the job chunks on which all scheduling algorithms have equal makespan, i.e. the makespan is a certain constant. This is because the makespan is roughly decided by the arrival time of the last few jobs in that job chunk. So that the makespan has no room for optimization.

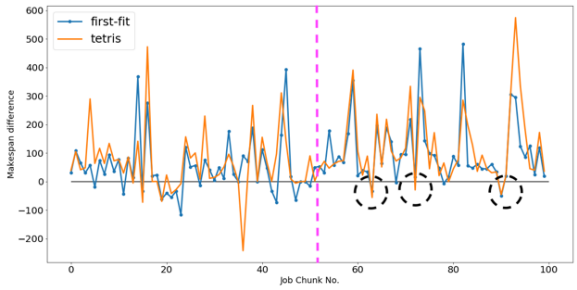

Figure 4 Makespan Difference

4.3 The Essence of Learning

The key consideration when designing DeepJS is to enable DeepJS to obtain a fitness calculation method via reinforcement learning. It is this consideration that makes the decision process of DeepJS more transparent and interpretable.

If the design of DeepJS is effective, it is obvious that when trained under different conditions, the fitness values calculated via different DeepJS agents on the same fit machine-task pair should be relevant. That is, a DeepJS agent thinks that a machine and a task is a good match, so another DeepJS agent should think so. Therefore, two DeepJS agents in two clusters with a different number of machines are trained using the sliding manner. The cluster where DeepJS agent A is located has 5 machines 64-core CPUs and 1 unit memory, the cluster where DeepJS agent B is located has 10 machines with 64-core CPUs and 1 unit of memory. The two agents are trained separately. After the training is completed, we select some fit machine-task pairs in a trajectory generated by DeepJS agent A arbitrarily, and each agent calculates the fitness for each fit machine-task pair. The correlation between fitness calculated by the two agents is shown in Figure 5(a). The correlation coefficient is 0.74 and p value is 0, which means that fitness calculated by the two agents are relevant.

Such a relationship also exists in DeepJS agents C and D trained on different job chunks. DeepJS C is trained on job chunk 5 whereas D on job chunk 6. As shown in Figure 5(b), the correlation coefficient is 0.55 and p value is 0.

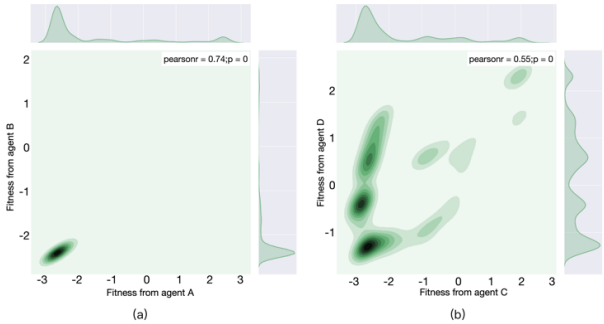

Figure 5 Correlation between fitness

As we can see, DeepJS has grasped the essence of job scheduling problems.

4.4 Performance Improvement

In order to verify DeepJS can effectively reduce makespan, DeepJS is trained on the first 200 job chunks using the sliding

manner, tested on the last 320 job chunks. The makespan reduction in shown in Figure 6. DeepJS performed well on the test set as a whole, but in the second half of the test set, although the makespan was reduced the performance fluctuated. It was reasonable as DeepJS goes deeper into the future without tuning its parameters gradually and the variation of the workload characteristics accumulates. Fortunately, in the actual deployment, the sliding training manner can be used to train the agent online and prevent the parameters of DeepJS from becoming stale.

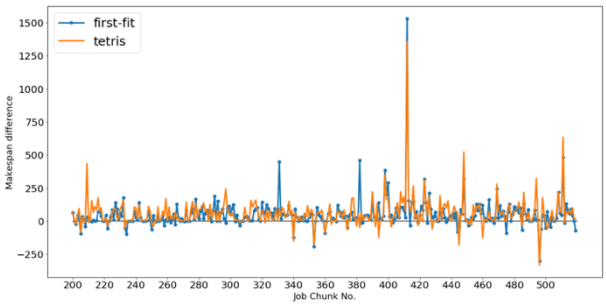

Figure 6 Makespan Difference on Test Set

Table 2 Makespan Difference Statistics

Algorithm	< 0	$= 0$	> 0
First-fit	45	36	239
Tetris	44	37	239

Statistics of the reduction of the makespan on the test set relative to other algorithms are shown in Figure 7. Figure 7(a) shows the mean and standard deviation of the makespan difference. Figure 7(b) is a box plot showing the maximum, minimum, median, and upper and lower quartiles of the makespan difference. From Table 2, We can see that DeepJS can reduce the makespan in the average sense, reduces makespan on 75% of the job chunks, and slightly increases the completion time on only 12% of the job chunks.

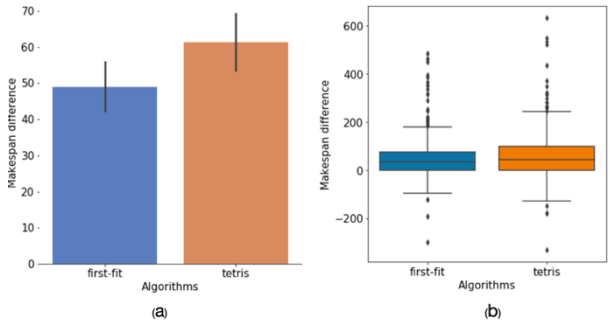

Figure 7 Makespan Difference Statistics

5. CONCLUSIONS

We have introduced DeepJS, a job scheduling algorithm based on deep reinforcement learning. DeepJS can automatically obtain a fitness calculation method which will minimize the makespan directly from experience. We have evaluated DeepJS through a trace-driven simulation, and shown that it outperforms the heuristic-based job scheduling algorithms.

6. ACKNOWLEDGMENTS

This work was supported in part by the National Science and Technology Major Projects for the New Generation of Broadband Wireless Communication Networks under Grant 2017ZX03001014, the National Natural Science Foundation of China for Distinguished Young Scholars under Grant 61425012.

7. REFERENCES

[1] Alibaba. (n.d.). Alibaba/clusterdata. Retrieved from https://github.com/alibaba/clusterdata/tree/master/cluster-trace-v2017

[2] Hadoop: Fair Scheduler. (n.d.). Retrieved April 12, 2019, from https://hadoop.apache.org/docs/current/hadoop-yarn/hadoop-yarn-site/FairScheduler.html

[3] Ghodsi, A., Zaharia, M., Hindman, B., Konwinski, A., Shenker, S., and Stoica, I. 2011. Dominant Resource Fairness: Fair Allocation of Multiple Resource Types. In *Nsdi* (Vol. 11, No. 2011, pp. 24-24).

[4] Chen, W., Xu, Y., and Wu, X. 2017. Deep reinforcement learning for multi-resource multi-machine job scheduling. arXiv preprint arXiv:1711.07440.

[5] Rao, J., Bu, X., Xu, C. Z., Wang, L., and Yin, G. 2009, June. VCONF: a reinforcement learning approach to virtual machines auto-configuration. In *Proceedings of the 6th international conference on Autonomic computing* (pp. 137-146). ACM.

[6] Yazdanov, L., and Fetzer, C. 2013, June. Vscaler: Autonomic virtual machine scaling. In *2013 IEEE Sixth International Conference on Cloud Computing* (pp. 212-219). IEEE.

[7] Basu, D., Wang, X., Hong, Y., Chen, H., and Bressan, S. 2019. Learn-as-you-go with megh: Efficient live migration of virtual machines. *IEEE Transactions on Parallel and Distributed Systems*.

[8] Duggan, M., Duggan, J., Howley, E., and Barrett, E. 2017. A reinforcement learning approach for the scheduling of live migration from under utilised hosts. *Memetic Computing*, 9(4), 283-293.

[9] Grandl, R., Ananthanarayanan, G., Kandula, S., Rao, S., and Akella, A. 2015. Multi-resource packing for cluster schedulers. *ACM SIGCOMM Computer Communication Review*, 44(4), 455-466.

[10] Lu, C., Ye, K., Xu, G., Xu, C. Z., and Bai, T. 2017. Imbalance in the cloud: An analysis on alibaba cluster trace. In *2017 IEEE International Conference on Big Data (Big Data)* (pp. 2884-2892). IEEE.

[11] Mao, H., Alizadeh, M., Menache, I., and Kandula, S. 2016. Resource management with deep reinforcement learning. In *Proceedings of the 15th ACM Workshop on Hot Topics in Networks* (pp. 50-56). ACM.

[12] Netflix. (n.d.). Netflix/Fenzo. Retrieved from https://github.com/Netflix/Fenzo/wiki

[13] Silver, D., Huang, A., Maddison, C. J., Guez, A., Sifre, L., Van Den Driessche, G., ... and Dieleman, S. 2016. Mastering the game of Go with deep neural networks and tree search. *nature*, 529(7587), 484.

[14] Bin packing problem. (2019, February 21). Retrieved from https://en.wikipedia.org/wiki/Bin_packing_problem