# Resource Management with Deep Reinforcement Learning

Studeny X[1,*] and Student Y[1,†]

[1]*Department of Mathematics and Informatics, St. Kliment Ohridski University of Sofia, 5 James Bourchier Blvd, 1164, Sofia, Bulgaria*

(Dated: January 23, 2021)

**Abstract** Solving resource management problems is essential in most computer systems and networks nowadays. Most of these problems are solved by using specifically designed heuristics based on the workload and environment of the system. We use Deep RL to create a system which learns to manage its resources from experience. Our results show that the system can outperform some of the standard (heuristic) methods, can adapt to different environment workload and be optimized for a specific system objective.

## I. INTRODUCTION

Every adequate computer system should be able to handle its resources optimally. This is mostly accomplished nowadays using heuristic algorithms, which take into consideration the load of the system and the specific environment. In most cases, if we want to improve the heuristic methods, we have to take into account the workload and the specifics of the particular environment, making them difficult to implement and maintain. Our idea is to create a system which learns to manage resources on its own from experience in such a way that it optimizes the overall resource utilization. This is a challenging endeavour, as these systems are usually complex and difficult to model accurately. The execution of a task depends on server metrics, system load, synchronization and much more. For this topic the only prerequisite knowledge required outside RL is basic understanding of job scheduling and resource allocation in computer systems.

In reinforcement learning the agents learn to make better decisions from their previous experience by interacting with the environment. The agent receives a reward based on how well it is doing on the task, so that it can improve its future actions. If we can create a reward which can guide our agent towards good solutions for our system objective, then this makes RL the right tool for building such a system. Furthermore, these systems normally make repetitive decisions thus making the resource management an appropriate RL task. Also an enormous amount of data is being easily generated and by continuous training, the RL agent can be optimized for a specific workload (e.g. small jobs).

We used synthetic dataset for our experiments. Based on our results, the RL system has better performance than some of the most well-known heuristic methods used today (SJF and Packer). We present the results in more detail in III. The advantage which we think is most significant of such a system is that it can adapt to different underlying systems without any prior knowledge of their behaviour, which makes the system more generic and widely applicable. All the code used for writing this paper is available in [1].

*Electronic address: x@uni-sofia.bg
†Electronic address: y@uni-sofia.bg

## II. METHODS

In this section we place the problem within RL by defining the state, action, reward spaces, together with the environment, the algorithms and neural network architectures used in the implementation.

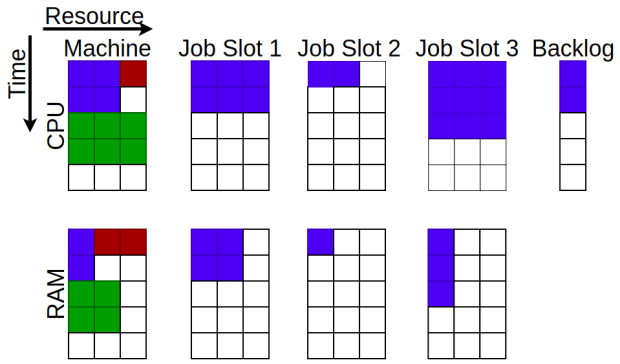


FIG. 1: An example of the state with two resources and three pending job slots.

**State space.** Each job which arrives to be scheduled for execution is represented as a pair of resource vector $r_j = (r_{j,1}, r_{j,2}, ..., r_{j,d})$, where $r_{j,i}$ is the resource requirement for the *i-th* resource type (e.g. CPU, GPU, memory) and $T_j$ which is the duration of the job. The state space consists of multiple two-dimensional grids, as shown on 1. The different colors in the cluster grids represent different jobs, which have been scheduled. On 1 the job placed under job slot 1 has a resource vector $r = (3, 2)$ and a duration of 2. Because this state is used as input to a neural network, we want it to have a fixed size. That's why we maintain a state with only the first $M$ jobs which have arrived for scheduling. The jobs after the first $M$ are put in the backlog, which just stores them in a queue and whenever a slot becomes empty from the

queue, a job is transferred from the backlog to that particular slot. The whole state of the system consists of the machine grid, the grids of the different jobs in the work queue and the grid of the backlog. Representing the state space in such a way also makes the action space linear in the size of the work queue.

**Action space.** The agent will pick actions based on a policy, which is the probability of picking an action in a given state. As we mentioned we keep a state with a fixed size $M$, but because the agent may want to schedule any subset of these $M$ jobs and for this problem there can be over $2^{100}$ such pairs thus making it impossible to store them all in memory. The approach which we have adapted is to have an action space with size $M + 1$, where action $= i$ means that the agent wants to schedule the job at that particular index in the working queue and the last action at index $M + 1$ is the void action. In order to allow the agent to schedule any subset of the $M$ jobs currently in the queue, on each time step time gets frozen, until the agent picks the void action or an invalid action. Invalid action is when the agent tries to schedule a job at index $i$, but the working queue does not contain a job at that index.

**Reward space.** The reward signal which we used is created so that we can make the agent optimize the system objective, which in our case is the average job slowdown. The average slowdown is calculated by the formula $\frac{1}{|J|}\sum_{j\in J}\frac{C_j}{T_j}$, where $J$ is the set of jobs currently in the system, $C_j$ is the finish time of the job i.e. the time between entering the system and finishing execution. This is divided by $T_j$ which is the length of the job. By adding this normalization we prevent the system from biasing towards large jobs. To optimize this objective our reward is $\sum_{j\in J}\frac{-1}{T_j}$. When the discount factor is set to 1, the cumulative reward is the same as the negative of the sum of the job slowdowns for some time step $t$. It is therefore possible by changing the reward to make the system optimize for different objective e.g. completion time or maximum resource utilization.

**Environment.** The environment is represented as a cluster with $d$ resource types and each type is a grid with height $20t$ and width $1r$ where each square on the grid represents a workload and duration of 1 for that particular resource type. For example on 1 there are 3 jobs currently running in the cluster presented with different colors for clarity, 3 jobs waiting to get scheduled in the working queue and 2 jobs in the backlog. The green job, which is currently running in the machine has a resource vector $r = (3, 2)$ and a duration of $T = 2$, which means that it will finish for 2 time steps and will require 3 units of CPU and 2 units of RAM in order to complete successfully. This environment has some simplifications from a real cluster, like currently it does not support preemption [2] or job fragmentation [3]. Even though, this simplifies the task from the real one, it still captures the essential characteristics of job scheduling.

**Algorithms.** We use deep neural network with one hidden layer to represent the policy $\pi_\theta(s, a)$. RL with

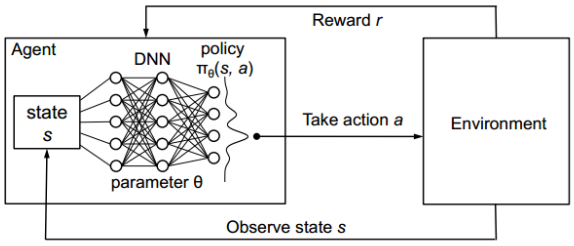


FIG. 2: Reinforcement learning with policy represented as Deep Neural Network

policy as Deep Neural Network is shown on 2. We have used RL algorithms that perform gradient-descent on the policy parameters. More specifically the algorithm which we used and which is also used in the original paper [4] is REINFORCE [5]. In general these algorithms are known as policy gradient methods. The idea of policy gradient is to estimate the gradient by observing the trajectories $\tau$ which we get by following the policy and then update the parameters $\theta$ of the policy using gradient descent. As we know, in RL the overall goal is to maximize the objective $J$ which is the expected cumulative discounted reward:

$$J = \mathrm{E}_{\tau\sim P_\pi}\left[G(\tau)|S_{t=0} = s_0\right], \quad G(\tau) = \sum_{t=1}^{T} r(s_t, a_t)$$

The gradient of the objective with respect to $\theta$ is:

$$\nabla_\theta J(\theta) = \nabla_\theta \mathrm{E}_{\tau\sim P_\pi}\left[\sum_{t=1}^{T} r(s_t, a_t)|S_{t=0} = s_0\right]$$

However, we don't know the transition probabilities $p(s'|s, a)$ and because of that we have to estimate the gradient from samples. The initial state distribution and the transition probabilities are independent of $\theta$, therefore using the definition of $P_{\pi_\theta}$ we see that:

$$\nabla_\theta P_{\pi_\theta}(\tau) = \nabla_\theta \pi_\theta(\tau), \text{ where } \pi_\theta(\tau) = \prod_{t=1}^{T} \pi(a_t|s_t)$$

Now, using MC we can estimate the gradient from a sample of multiple trajectories.

$$\nabla_\theta J(\theta) = \mathrm{E}_{\tau\sim P_\pi}\left[\nabla_\theta \log P_{\pi_\theta}(\tau) G(\tau)\right]$$

After that we update the policy parameters via gradient descent using the following rule:

$$\theta \leftarrow \theta + \alpha \sum_{j=1}^{T} \nabla_\theta \log \pi_\theta(\tau_j) G(\tau_j)$$

However, this has a problem which occurs due to high variance of the gradient estimate. That's why we have

used a variant of the REINFORCE [5] algorithm, which reduces the variance by subtracting a baseline value from each return $G(\tau_j)$. Our update rule has the form:

$$\theta \leftarrow \theta + \alpha \sum_{j=1}^{N} \sum_{t=1}^{T} \nabla_\theta \log \pi_\theta(a_t^j | s_t^j) \left[ \sum_{t'=t}^{T} r(a_{t'}^j | s_{t'}^j)) - b \right]$$

The overall effect is that by doing this we reinforce actions which lead to better returns in general.

To further improve performance of policy gradient algorithm we examined the Natural policy gradient approach described in [6]. Ordinary gradient descent methods typically search for the smallest parameter change $\Delta\theta$ which causes the biggest negative change of the loss function and they generally perform very well on supervised learning tasks. However, RL introduces a further problem - the non-stationarity of the space, so the search is now oriented to finding the biggest $\Delta\theta$ which induces the smallest change in the policy, but in the right direction. We need to establish a distance measure between the distributions of the current policy and the updated one. When dealing with Euclidean parametric space the distance measure is:

$$\|\theta - \theta'\|^2 = \sqrt{(\theta - \theta')^2}$$

where $\theta' = \theta + \Delta\theta$ is the new policy. However, normally the parametric space of a RL task has a Riemannian metric structure, introduced in [7], so another metric should be applied to measure distance in Riemannian sense. One common measurement of the statistical distance between two distributions $p$ and $q$ is the Kullback-Leibler (KL) divergence ([8]):

$$D_{KL}(p\|q) = \mathrm{E}_{x\sim p}\left[\log \frac{p(x)}{q(x)}\right] = \int p(x) \log \frac{p(x)}{q(x)}$$

The KL divergence has the property of not being symmetrical ($D_{KL}(p\|q) \neq D_{KL}(q\|p)$), so most commonly the symmetric KL divergence is used, also called Jensen-Shannon ($JS$) divergence:

$$D_{JS}(p\|q) = \frac{D_{KL}(p\|q) + D_{KL}(q\|p)}{2}$$

In cases when symmetric KL divergence is used to measure the distance between two distributions, the corresponding Riemannian metric is the Fisher information matrix (FIM), defined as the Hessian matrix of the KL divergence around $\theta$:

$$F(\theta) = \nabla^2 D_{JS}(\pi_\theta(a_t, s_t)\|\pi_{\theta'}(a_t, s_t))\big|_{\theta=0}$$

This equation requires that second order derivatives are calculated, they are complex and heavy to obtain, especially when $\theta$ parameters are too many. For this reason we use another formula making use of the log probabilities of the distribution:

$$F(\theta) = \mathrm{E}_{\pi_\theta}\left[\nabla_\theta \log \pi_\theta \left(\nabla_\theta \log \pi_\theta\right)^T\right]$$

The Fisher information matrix is useful as it allows to locally approximate the KL divergence between two distributions (when $\Delta\theta$ is small). Using a second order Taylor expansion we get:

$$D_{JS}(p\|q) \approx \Delta\theta^T F(\theta) \Delta\theta$$

The KL divergence is then locally quadratic, which means that the update when performing gradient descent optimization will be linear. With natural gradient descent we move along the statistical manifold defined by $p$ updating the gradient of $J(\theta)$ using the local curvature of the KL-divergence surface, meaning that we move in the direction of $\tilde{\nabla}_\theta J(\theta)$ - the natural gradient of $J(\theta)$:

$$\tilde{\nabla}_\theta J(\theta) = F(\theta)^{-1} \nabla_\theta J(\theta)$$

In this sense Natural gradient descent moves in the direction:

$$\Delta\theta = -\alpha \tilde{\nabla}_\theta J(\theta)$$

where $\alpha$ is the step size. Finally, the update rule becomes:

$$\theta' = \theta - \alpha \tilde{\nabla}_\theta J(\theta)$$

We estimated $F(\theta)$ using the empirical Monte Carlo methods described in [9].

## III. RESULTS

We built setup similar to the one described in [4]. We have a fixed job arrival rate which is used to regulate the load of the cluster. The duration of each job is chosen uniformly between $1t$ and $3t$ with 80% probability for these so called small jobs and the rest are also chosen uniformly between $10t$ and $15t$ - big jobs. The cluster has 2 resources, each with capacity $1r$ and each job has a dominant resource, which is randomly picked. The resource demands of each job are between $0.25r$ and $0.5r$ for the dominant resource and between $0.05r$ and $0.1r$ for the other resource. The deep neural network has one hidden layer, which has 20 neurons and the cluster has a fixed $20t$ duration for scheduling incoming jobs. The work queue from which the agent chooses jobs for scheduling has a fixed size of 10, but the agent can also observe all the other jobs in the backlog, which also has s fixed size of 60. In each training iteration we run 20 MC simulations for each job set and after that the policy parameters get updated using ADAM [10] with a learning rate of 0.001. The size of each job set is also fixed and based on the job rate the overall cluster load can be regulated and estimated. The calculation of the cluster load is not specified in the original paper [4], so we have come up with the following formula for estimating it: $L = Q + M$, where $Q$ is the load of the working queue and $M$ is the machine load. $Q = \frac{\text{jobs in the queue}}{\text{queue size}}$ and $M = \frac{1}{K}\sum_{i=1}^{K} R_i$, where $R_i = \frac{\text{unavailable slots}}{\text{all slots}}$ is the load of the $i\text{-}th$ resource type

in the cluster. In the results which we show we have used $r = 50$ and $t = 3$. The objective which we measure is the average job slowdown for each episode. On 3 we have shown the average slowdown and the mean reward with its variance achieved at each iteration during training. The results shown are from training for 1000 episodes and cluster load of around $70 - 80\%$.

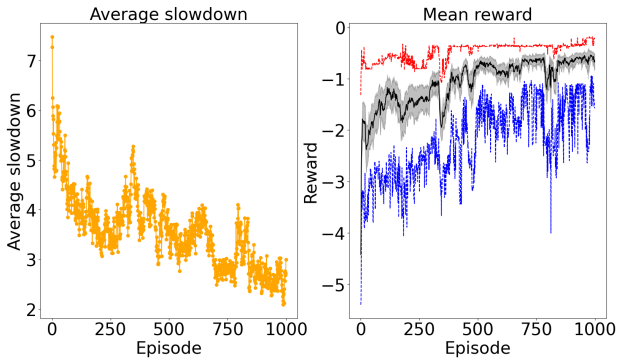


FIG. 3: Average slowdown and the mean final reward with its variance for 1000 episodes.

We have compared the Deep RL algorithm with the two most well-known job scheduling algorithms today - SJF and Packer. The SJF algorithm allocates jobs in a increasing order of their duration and the Packer algorithm allocates jobs by picking the one with the highest dot product of job resource vector and machine resource vector as described in [11]. We have compared them based on the system objective for which the training was done - the average job slowdown and cluster load approximately 70-80%. The comparison results are plotted on 4.

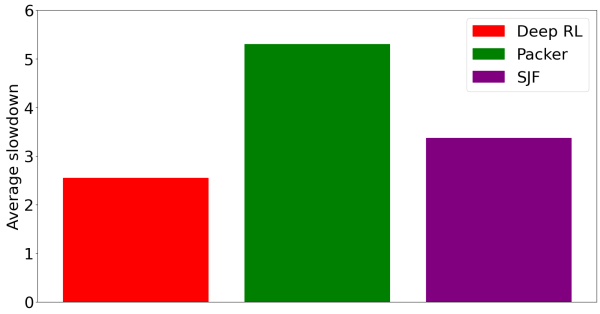


FIG. 4: Average slowdown of the Deep RL, Packer and SJF algorithms for cluster load approximately 70-80%

As we can see SJF performs better than Packer because it allocates the smallest jobs first. The Deep RL algorithm performs slightly better than the SJF algorithm, because it is able to learn that it is optimal to keep some resources free in the cluster in order to be able to schedule future arriving small jobs as fast as possible thus reducing the average slowdown. Optimization for other objectives is also possible by crafting the appropriate reward. For example to optimize for average completion time we can use as reward at each timestep $-|J|$, which is the negative number of unfinished jobs currently in the system or for average resource utilization we can use the computed system load, because higher load = better resource utilization. The major advantage of such an algorithm over the other heuristic algorithms is that it can adapt to different underlying systems, without any prior knowledge of their behaviour and optimize for different objectives. These facts make the system more generic and widely applicable. We also made an attempt to implement Natural policy gradient algorithm, described in [6], but unfortunately a working solution could not be achieved. Reasons why the policy network could not converge are not at full extent clear, but several anomalies were found, namely the Fisher information matrix was too large to populate memory, so tests had to be conducted with the compact environment state which is not so representative as the complete state. What is more, the training of the network slowed down by a factor of 4 per iteration. This happened because of the exhaustive Monte Carlo approximation of $F(\theta)$. In an attempt to enforce faster convergence we also tried some of the methods described in [12]. In that approach the step size is assigned:

$$\alpha = \sqrt{\frac{2\epsilon}{\nabla_\theta J(\theta)^T F(\theta) \nabla_\theta J(\theta)}}$$

where $\epsilon$ is a small arbitrary value. The term under the root appeared to become negative despite our numerous attempts to find a solution. One speculation on why this happened is that $F(\theta)$ is an approximation of the real Fisher matrix and does not have the actual values that have to be there.

## IV. CONCLUSION AND OUTLOOK

In this final project paper we have reproduced some of the results from [4] and we show that it is appropriate to use Deep RL methods to manage job scheduling systems. Our RL agent can directly learn from experience and has better performance than the heuristic methods for solving these type of problems, so it definitely can be used as their replacement. The model has some limitations as also stated in [4], for example it does not model the dependencies between different jobs (shared memory, CPU cache etc.), does not include job fragmentation and the job scheduling is non-preemptive, which simplifies the overall problem. However, even with these simplifications the model captures the essentials of a job scheduling cluster. Finally, we think that these limitations open some new direction for research and once overcome the model will become more realistic and can be deployed to a physical cluster for real testing.

[1] "Resource management with deep reinforcement learning code repository," https://github.com/angelbeshirov/resource-management (2021).

[2] D. R. Belgaum, S. Soomro, Z. Alansari, S. Musa, M. Alam, and M. M. Su'ud, *Load Balancing with preemptive and non-preemptive task scheduling in Cloud Computing* (2019).

[3] S. Mitra, S. S. Mondal, N. Sheoran, N. Dhake, R. Nehra, and R. Simha, *DeepPlace: Learning to Place Applications in Multi-Tenant Clusters* (2019).

[4] H. Mao, M. Alizadeh, I. Menache, and S. Kandula, *Resource management with deep reinforcement learning* (2016).

[5] R. S. Sutton, D. A. McAllester, S. P. Singh, Y. Mansour, and et al., *Policy gradient methods for reinforcement learning with function approximation* (1999).

[6] S.-i. Amari, Neural Computation **10** (2000).

[7] S.-i. Amari, *Differential-Geometrical Methods in Statistics*, Vol. 10 (2000).

[8] D. Pollard, *Asymptopia: An Exposition of Statistical Asymptotic Theory* (???, 1999) unpublished book manuscript.

[9] J. Peters and S. Schaal, Neural networks : the official journal of the International Neural Network Society **21** **4**, 682 (2008).

[10] D. P. Kingma and J. Ba, *Adam: A Method for Stochastic Optimization* (2017).

[11] R. Grandl, G. Ananthanarayanan, S. Kandula, S. Rao, and A. Akella, *Multi-resource packing for cluster schedulers* (2014).

[12] J. Schulman, S. Levine, P. Moritz, M. I. Jordan, and P. Abbeel, (2017), arXiv:1502.05477 [cs.LG] .