Resource Management with Deep Reinforcement Learning

Hongzi Mao*, Mohammad Alizadeh*, Ishai Menache†, Srikanth Kandula†
Massachusetts Institute of Technology*, Microsoft Research†
{hongzi, alizadeh}@mit.edu, {ishai, srikanth}@microsoft.com

Abstract– Resource management problems in systems and networking often manifest as difficult online decision making tasks where appropriate solutions depend on understanding the workload and environment. Inspired by recent advances in deep reinforcement learning for AI problems, we consider building systems that learn to manage resources directly from experience. We present DeepRM, an example solution that translates the problem of packing tasks with multiple resource demands into a learning problem. Our initial results show that DeepRM performs comparably to state-of-the-art heuristics, adapts to different conditions, converges quickly, and learns strategies that are sensible in hindsight.

1. INTRODUCTION

Resource management problems are ubiquitous in computer systems and networks. Examples include job scheduling in compute clusters [17], bitrate adaptation in video streaming [23, 39], relay selection in Internet telephony [40], virtual machine placement in cloud computing [20, 6], congestion control [38, 37, 13], and so on. The majority of these problems are solved today using meticulously designed heuristics. Perusing recent research in the field, the typical design flow is: (1) come up with clever heuristics for a simplified model of the problem; and (2) painstakingly test and tune the heuristics for good performance in practice. This process often has to be repeated if some aspect of the problem such as the workload or the metric of interest changes.

We take a step back to understand some reasons for why real world resource management problems are challenging:

1. The underlying systems are complex and often impossible to model accurately. For instance, in cluster scheduling, the running time of a task varies with data locality, server characteristics, interactions with other tasks, and interference on shared resources such as CPU caches, network bandwidth, etc [12, 17].
2. Practical instantiations have to make online decisions with noisy inputs and work well under diverse conditions. A video streaming client, for instance, has to choose the bitrate for future video chunks based on noisy forecasts of available bandwidth [39], and operate well for different codecs, screen sizes and available bandwidths (e.g., DSL vs. T1).
3. Some performance metrics of interest, such as tail performance [11], are notoriously hard to optimize in a principled manner.

In this paper, we ask if machine learning can provide a viable alternative to human-generated heuristics for resource management. In other words: *Can systems learn to manage resources on their own?*

This may sound like we are proposing to build Skynet [1], but the recent success of applying maching learning to other challenging decision-making domains [29, 33, 3] suggests that the idea may not be too far-fetched. In particular, *Reinforcement Learning (RL)* (§2) has become an active area in machine learning research [30, 28, 32, 29, 33]. RL deals with agents that learn to make better decisions *directly from experience* interacting with the environment. The agent starts knowing nothing about the task at hand and learns by *reinforcement* — a reward that it receives based on how well it is doing on the task. RL has a long history [34], but it has recently been combined with *Deep Learning* techniques to great effect in applications such as playing video games [30], Computer Go [33], cooling datacenters [15], etc.

Revisiting the above challenges, we believe RL approaches are especially well-suited to resource management systems. First, decisions made by these systems are often highly repetitive, thus generating an abundance of training data for RL algorithms (e.g., cluster scheduling decisions and the resulting performance). Second, RL can model complex systems and decision-making policies as deep neural networks analogous to the models used for game-playing agents [33, 30]. Different "raw" and noisy signals[1] can be incorporated as in-

HotNets-XV, November 09-10, 2016, Atlanta, GA, USA

 ISBN 978-1-4503-4661-0/16/11…$15.00
DOI: http://dx.doi.org/10.1145/3005745.3005750

[1] It is usually easy to identify signals that are relevant to a decision-making task (but not how to filter/combine them to make decisions).

put to these neural networks, and the resultant strategy can be used in an online stochastic environment. Third, it is possible to train for objectives that are hard-to-optimize directly because they lack precise models if there exist reward signals that correlate with the objective. Finally, by continuing to learn, an RL agent can optimize for a specific workload (e.g., small jobs, low load, periodicity) and be graceful under varying conditions.

As a first step towards understanding the potential of RL for resource management, we design (§3) and evaluate (§4) DeepRM, a simple multi-resource cluster scheduler. DeepRM operates in an online setting where jobs arrive dynamically and cannot be preempted once scheduled. DeepRM learns to optimize various objectives such as minimizing average job slowdown or completion time. We describe the model in §3.1 and how we pose the scheduling task as an RL problem in §3.2. To learn, DeepRM employs a standard *policy gradient* reinforcement learning algorithm [35] described in §3.3.

We conduct simulated experiments with DeepRM on a synthetic dataset. Our preliminary results show that across a wide range of loads, DeepRM performs comparably or better than standard heuristics such as Shortest-Job-First (SJF) and a packing scheme inspired by Tetris [17]. It learns strategies such as favoring short jobs over long jobs and keeping some resources free to service future arriving short jobs *directly from experience*. In particular, DeepRM does not require any prior knowledge of the system's behavior to learn these strategies. Moreover, DeepRM can support a variety of objectives just by using different reinforcement rewards.

Looking ahead, deploying an RL-based resource manager in real systems has to confront additional challenges. To name a few, simple heuristics are often easier to explain, understand, and verify compared to an RL-based scheme. Heuristics are also easier to adopt incrementally. Nevertheless, given the scale and complexity of many of the resource management problems that we face today, we are enticed by the possibility to improve by using reinforcement learning.

2. BACKGROUND

We briefly review Reinforcement Learning (RL) techniques that we build on in this paper; we refer readers to [34] for a detailed survey and rigorous derivations.

Reinforcement Learning. Consider the general setting shown in Figure 1 where an *agent* interacts with an *environment*. At each time step t, the agent observes some state s_t, and is asked to choose an action a_t. Following the action, the state of the environment transitions to s_{t+1} and the agent receives reward r_t. The state transitions and rewards are stochastic and are assumed to have the Markov property; i.e. the state transition probabilities and rewards depend only on the state of the environment s_t and the action taken by the agent a_t.

It is important to note that the agent can only control its actions, it has no apriori knowledge of which state the environment would transition to or what the reward may be. By interacting with the environment, during training, the agent can observe these quantities. The goal of learning is to maximize the expected cumulative discounted reward: $\mathbb{E}\left[\sum_{t=0}^{\infty} \gamma^t r_t\right]$, where $\gamma \in (0, 1]$ is a factor discounting future rewards.

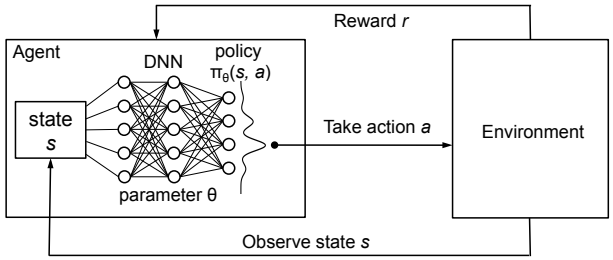


Figure 1: Reinforcement Learning with policy represented via DNN.

Policy. The agent picks actions based on a *policy*, defined as a probability distribution over actions $\pi : \pi(s, a) \rightarrow [0, 1]$; $\pi(s, a)$ is the probability that action a is taken in state s. In most problems of practical interest, there are many possible {state, action} pairs; up to 2^{100} for the problem we consider in this paper (see §3). Hence, it is impossible to store the policy in tabular form and it is common to use *function approximators* [7, 27]. A function approximator has a manageable number of adjustable parameters, θ; we refer to these as the *policy parameters* and represent the policy as $\pi_\theta(s, a)$. The justification for approximating the policy is that the agent should take similar actions for "close-by" states.

Many forms of function approximators can be used to represent the policy. For instance, linear combinations of features of the state/action space (i.e., $\pi_\theta(s, a) = \theta^T \phi(s, a)$) are a popular choice. Deep Neural Networks (DNNs) [18] have recently been used successfully as function approximators to solve large-scale RL tasks [30, 33]. An advantage of DNNs is that they do not need hand-crafted features. Inspired by these successes, we use a neural network to represent the policy in our design; the details are in §3.

Policy gradient methods. We focus on a class of RL algorithms that learn by performing *gradient-descent* on the policy parameters. Recall that the objective is to maximize the expected cumulative discounted reward; the gradient of this objective given by [34]:

$$\nabla_\theta \mathbb{E}_{\pi_\theta}\left[\sum_{t=0}^{\infty} \gamma^t r_t\right] = \mathbb{E}_{\pi_\theta}\left[\nabla_\theta \log \pi_\theta(s, a) Q^{\pi_\theta}(s, a)\right]. \quad (1)$$

Here, $Q^{\pi_\theta}(s, a)$ is the expected cumulative discounted reward from (deterministically) choosing action a in state s, and subsequently following policy π_θ. The key idea in policy gradient methods is to estimate the gradient by observing the trajectories of executions that are obtained by following the policy. In the simple *Monte Carlo Method* [19], the agent samples multiple trajectories and uses the empirically computed cumulative discounted reward, v_t, as an unbiased estimate of $Q^{\pi_\theta}(s_t, a_t)$. It then updates the policy parameters

via gradient descent:

$$\theta \leftarrow \theta + \alpha \sum_t \nabla_\theta \log \pi_\theta(s_t, a_t) v_t, \qquad (2)$$

where α is the step size. This equation results in the well-known REINFORCE algorithm [35], and can be intuitively understood as follows. The direction $\nabla_\theta \log \pi_\theta(s_t, a_t)$ gives how to change the policy parameters in order to increase $\pi_\theta(s_t, a_t)$ (the probability of action a_t at state s_t). Equation 2 takes a step in this direction; the size of the step depends on how large is the return v_t. The net effect is to reinforce actions that empirically lead to better returns. In our design, we use a slight variant [32] that reduces the variance of the gradient estimates by subtracting a baseline value from each return v_t. More details follow.

3. DESIGN

In this section, we present our design for online multi-resource cluster scheduling with RL. We formulate the problem (§3.1) and describe how to represent it as an RL task (§3.2). We then outline our RL-based solution (§3.3) building on the machinery described in the previous section.

3.1 Model

We consider a cluster with d resource types (e.g., CPU, memory, I/O). Jobs arrive to the cluster in an online fashion in discrete timesteps. The scheduler chooses one or more of the waiting jobs to schedule at each timestep. Similar to prior work [17], we assume that the resource demand of each job is known upon arrival; more specifically, the resource *profile* of each job j is given by the vector $\mathbf{r}_j = (r_{j,1}, \ldots, r_{j,d})$ of resources requirements, and T_j – the *duration* of the job. For simplicity, we assume no preemption and a fixed allocation profile (i.e., no malleability), in the sense that $\mathbf{r}_j$ must be allocated continuously from the time that the job starts execution until completion. Further, we treat the cluster as a single collection of resources, ignoring machine fragmentation effects. While these aspects are important for a practical job scheduler, this simpler model captures the essential elements of multi-resource scheduling and provides a non-trivial setting to study the effectiveness of RL methods in this domain. We discuss how the model can be made more realistic in §5.

Objective. We use the *average job slowdown* as the primary system objective. Formally, for each job j, the slowdown is given by $S_j = C_j/T_j$, where C_j is the completion time of the job (i.e., the time between arrival and completion of execution) and T_j is the (ideal) duration of the job; note that $S_j \geq 1$. Normalizing the completion time by the job's duration prevents biasing the solution towards large jobs, which can occur for objectives such as mean completion time.

3.2 RL formulation

State space. We represent the state of the system — the current allocation of cluster resources and the resource profiles of jobs waiting to be scheduled — as distinct *images* (see Figure 2 for illustration). The cluster images (one for each resource; two leftmost images in the figure) show the allocation of each resource to jobs which have been scheduled for service, starting from the current timestep and looking ahead T timesteps into the future. The different colors within these images represent different jobs; for example, the red job in Figure 2 is scheduled to use two units of CPU, and one unit of memory for the next three timesteps. The job slot images represent the resource requirements of awaiting jobs. For example, in Figure 2, the job in Slot 1 has a duration of two timesteps, in which it requires two units of CPU and one unit of memory.[2]

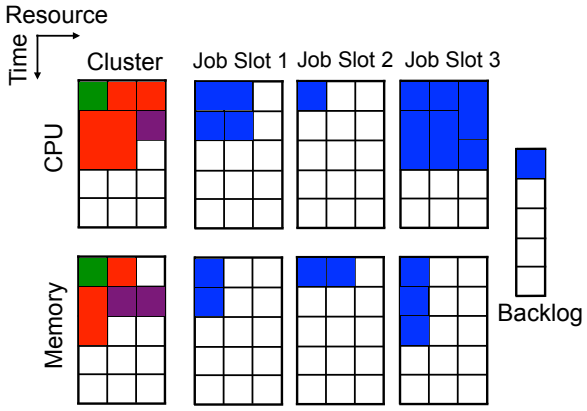


Figure 2: An example of a state representation, with two resources and three pending job slots.

Ideally, we would have as many job slot images in the state as there are jobs waiting for service. However, it is desirable to have a fixed state representation so that it can be applied as input to a neural network. Hence, we maintain images for only the first M jobs to arrive (which have not yet been scheduled). The information about any jobs beyond the first M is summarized in the *backlog* component of the state, which simply counts the number of such jobs. Intuitively, it is sufficient to restrict attention to the earlier-arriving jobs because plausible policies are likely to prefer jobs that have been waiting longer. This approach also has the added advantage of constraining the action space (see below) which makes the learning process more efficient.

Action space. At each point in time, the scheduler may want to admit any subset of the M jobs. But this would require a large action space of size 2^M which could make learning very challenging. We keep the action space small using a trick: we allow the agent to execute more than one action in each timestep. The action space is given by $\{\emptyset, 1, \ldots, M\}$, where $a = i$ means "schedule the job at the i-th slot"; and $a = \emptyset$ is a "void" action that indicates that the agent does not wish to schedule further jobs in the current timestep. At each timestep, time is frozen until the scheduler either chooses the

[2]We tried more succinct representations of the state (e.g., a job's resource profile requires only $d + 1$ numbers). However, they did not perform as well in our experiments for reasons that we do not fully understand yet.

void action, or an invalid action (e.g., attempting to schedule a job that does not "fit" such as the job at Slot 3 in Figure 2). With each valid decision, one job is scheduled in the first possible timestep in the cluster (i.e., the first timestep in which the job's resource requirements can be fully satisfied till completion). The agent then observes a state transition: the scheduled job is moved to the appropriate position in the cluster image. Once the agent picks $a = \emptyset$ or an invalid action, time actually proceeds: the cluster images shift up by one timestep and any newly arriving jobs are revealed to the agent. By decoupling the agent's decision sequence from real time, the agent can schedule multiple jobs at the same timestep while keeping the action space linear in M.

Rewards. We craft the reward signal to guide the agent towards good solutions for our objective: minimizing average slowdown. Specifically, we set the reward at each timestep to $\sum_{j \in \mathcal{J}} \frac{-1}{T_j}$, where $\mathcal{J}$ is the set of jobs currently in the system (either scheduled or waiting for service).[3] The agent does not receive any reward for intermediate decisions during a timestep (see above). Observe that setting the discount factor $\gamma = 1$, the cumulative reward over time coincides with (negative) the sum of job slowdowns, hence maximizing the cumulative reward mimics minimizing the average slowdown.

3.3 Training algorithm

We represent the policy as a neural network (called policy network) which takes as input the collection of images described above, and outputs a probability distribution over all possible actions. We train the policy network in an *episodic* setting. In each episode, a fixed number of jobs arrive and are scheduled based on the policy, as described in §3.2. The episode terminates when *all* jobs finish executing.

To train a policy that generalizes, we consider multiple examples of job arrival sequences during training, henceforth called *jobsets*. In each training iteration, we simulate N episodes for each jobset to explore the probabilistic space of possible actions using the current policy, and use the resulting data to improve the policy for all jobsets. Specifically, we record the state, action, and reward information for all timesteps of each episode, and use these values to compute the (discounted) cumulative reward, v_t, at each timestep t of each episode. We then train the neural network using a variant of the REINFORCE algorithm described in §2.

Recall that REINFORCE estimates the policy gradient using Equation (2). A drawback of this equation is that the gradient estimates can have high variance. To reduce the variance, it is common to subtract a *baseline* value from the returns, v_t. The baseline can be calculated in different ways. The simple approach that we adopt is to use the average of the return values, v_t, where the average is taken at the same

[3]Note that the above reward function considers all the jobs in the system; not just the first M. From a theoretical standpoint, this makes our learning formulation more challenging as the reward depends on information not available as part of the state representation (resulting in a Partially Observed Markov Decision Process [31]); nevertheless, this approach yielded good results in practice.

```
for each iteration:
    Δθ ← 0
    for each jobset:
        run episode i = 1,...,N:
            {s_1^i, a_1^i, r_1^i, ..., s_{L_i}^i, a_{L_i}^i, r_{L_i}^i} ~ π_θ
        compute returns: v_t^i = Σ_{s=t}^{L_i} γ^{s-t} r_s^i
        for t = 1 to L:
            compute baseline: b_t = (1/N) Σ_{i=1}^{N} v_t^i
            for i = 1 to N:
                Δθ ← Δθ + α ∇_θ log π_θ(s_t^i, a_t^i)(v_t^i − b_t^i)
            end
        end
    end
    θ ← θ + Δθ  % batch parameter update
end
```

Figure 3: Pseudo-code for training algorithm.

timestep t across all episodes[4] with the same jobset (a similar approach has been used in [32]). Figure 3 shows the pseudo-code for the training algorithm.

3.4 Optimizing for other objectives

The RL formulation can be adapted to realize other objectives. For example, to minimize average completion time, we can use $-|\mathcal{J}|$ (negative the number of unfinished jobs in the system) for the reward at each timestep. To maximize resource utilization, we could reward the agent for the sum of the resource utilizations at each timestep. The makespan for a set of jobs can also be minimized by penalizing the agent one unit for each timestep for which unfinished jobs exist.

4. EVALUATION

We conduct a preliminary evaluation of DeepRM to answer the following questions.

- When scheduling jobs that use multiple resources, how does DeepRM compare with state-of-the-art mechanisms?
- Can DeepRM support different optimization objectives?
- Where do the gains come from?
- How long does DeepRM take to train?

4.1 Methodology

Workload. We mimic the setup described in §3.1. Specifically, jobs arrive online according to a Bernoulli process. The average job arrival rate is chosen such that the average load varies between 10% to 190% of cluster capacity. We assume two resources, i.e., with capacity $\{1r, 1r\}$. Job durations and resource demands are chosen as follows: 80% of the jobs have duration uniformly chosen between $1t$ and $3t$; the remaining are chosen uniformly from $10t$ to $15t$. Each job has a dominant resource which is picked independently at random. The demand for the dominant resource is chosen uniformly between $0.25r$ and $0.5r$ and the demand of the other resource is chosen uniformly between $0.05r$ and $0.1r$.

DeepRM. We built the DeepRM prototype described in §3 using a neural network with a fully connected hidden layer

[4]Some episodes terminate earlier, thus we zero-pad them to make every episode equal-length L.

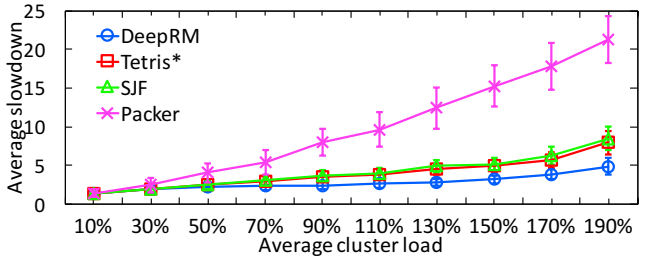


Figure 4: Job slowdown at different levels of load.

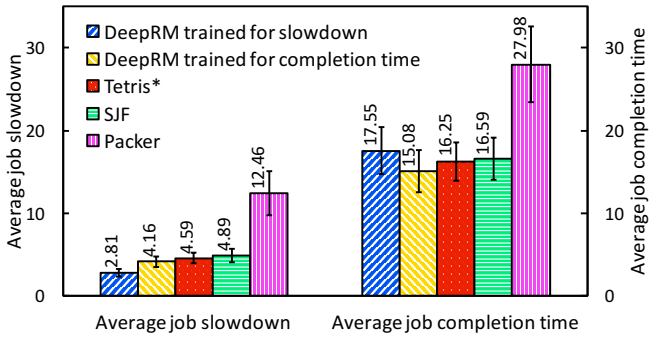


Figure 5: Performance for different objectives.

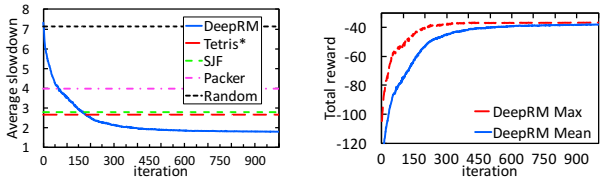
with 20 neurons, and a total of $89,451$ parameters. The "images" used by the DeepRM agent are $20t$ long and each experiment lasts for $50t$. Recall that the agent allocates from a subset of M jobs (we use $M = 10$) but can also observe the number of other jobs ("backlog" which we set to 60 jobs). We use 100 different jobsets during training. In each training iteration, per jobset we run $N = 20$ Monte Carlo simulations in parallel. We update the policy network parameters using the *rmsprop* [21] algorithm with a learning rate of 0.001. Unless otherwise specified, the results below are from training DeepRM for 1000 training iterations.

Comparables. We compare DeepRM against a *Shortest Job First (SJF)* agent which allocates jobs in increasing order of their duration; a *Packer* agent which allocates jobs in increasing order of alignment between job demands and resource availability (same as the packing heuristic in [17]); and *Tetris⋆*, an agent based on Tetris [17] which balances preferences for short jobs and resource packing in a combined score. These agents are all work-conserving, and allocate as many jobs as can be accommodated with the available resources (in the preference order).

Metrics. We measure the various schemes on a few different aspects: the average job slowdown (§3.1) and the average job completion time. To observe the convergence behavior of DeepRM, we also measure the total reward achieved at each iteration during training.

4.2 Comparing scheduling efficiency

Figure 4 plots the average job slowdown for DeepRM versus other schemes at different load levels. Each datapoint is an average over 100 new experiments with jobsets not used during training. As expected, we see that (1) the average slowdown increases with cluster load, (2) SJF performs better than Packer because it allocates the smaller jobs first, and (3) Tetris⋆ outperforms both heuristics by combining their advantages. The figure also shows that DeepRM is comparable to and often better than all heuristics. As we will see shortly, DeepRM beats Tetris⋆ at higher loads because it automatically learns to keep some resources free so that small jobs arriving in the near future can be scheduled quickly; with Tetris⋆, small jobs may sometimes have to wait until a (large) existing job finishes because Tetris⋆ enforces work-conservation, which may not always be the best strategy when jobs cannot be preempted. Remarkably, DeepRM is able to learn such strategies directly from experience — without any prior knowledge of what strategies (e.g., favoring shorter jobs) are suitable.



(a) average slowdown (b) total reward

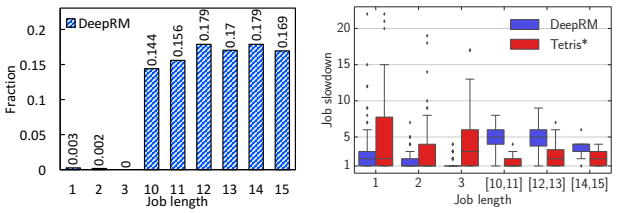
Figure 6: Learning curve showing the training improves the total reward as well as the slowdown performance.

Other objectives. Figure 5 shows the behavior for two objectives (average job slowdown and average job completion time) when the cluster is highly loaded (load=130%). Recall that DeepRM uses a different reward function for each objective ($-|\mathcal{J}|$ to optimize average job completion time, and $\sum_{j\in\mathcal{J}} \frac{-1}{T_j}$ for average job slowdown; see §3.4). As before we see that Tetris⋆ outperforms the other heuristics. However, DeepRM is the best performing scheme on each objective when trained specifically to optimize for that objective with the appropriate reward function. Thus DeepRM is customizable for different objectives.

4.3 Understanding convergence and gains

Convergence behavior. To understand the convergence behavior of DeepRM, we look deeper into one specific datapoint of Figure 4: training DeepRM to optimize the average job slowdown at a load of 70%. Figure 6(a) plots the average job slowdown achieved by the policy learnt by the end of each iteration. To compare, the figure also plots the values for the other schemes. As expected, we see that DeepRM improves with iteration count; the starting policy at iteration 0 is no better than a random allocation but after 200 iterations DeepRM is better than Tetris⋆.

Would running more iterations further improve the average job slowdown? Figure 6(b) plots the maximum reward across all of the Monte Carlo runs at each iteration and the average reward. As expected, both values increase with iterations as DeepRM's policy improves. Further, higher rewards, which is what the learning algorithm explicitly optimizes for, correlate with improvements in average job slowdown (Fig. 6(a)). Finally, recall that the policy is a probability distribution over possible actions. Hence, a large gap



(a) length of withheld jobs (b) slowdown vs. job length

Figure 7: DeepRM withholds large jobs to make room for small jobs and reduce overall average slowdown.

between the maximum and average rewards implies the need for further learning; if some action path (sampled from the policy output by the neural network) is much better than the average action path, there is room to increase reward by adjusting the policy. Conversely, when this gap narrows, the model has converged; as we see near the 1000th iteration.

We have thus far measured number of iterations but how long does one iteration take to compute? Our multi-threaded implementation took 80 seconds per iteration on a 24-core CPU server. Offloading to a GPU server may speedup the process further [10, 2]. Finally, note that the policy need only be trained once; retraining is needed only when the environment (cluster or job workload) changes notably.

Where are the gains from? To understand why DeepRM performs better, we examine the case with load 110%. Recall that, unlike the other schemes, DeepRM is not work-conserving, i.e., it can hold a job even if there are enough resources available to allocate it. We found that in 23.4% of timesteps in this scenario, DeepRM was not work-conserving. Whenever DeepRM withholds a job, we record its length, and plot the distribution of the length of such jobs in Figure 7(a). We see that DeepRM almost always withholds only large jobs. The effect is to make room for yet-to-arrive small jobs. This is evident in Figure 7(b): the slowdown for small jobs is significantly smaller with DeepRM than Tetris*, with the tradeoff being higher slowdown for large jobs. Since in this workload (§4.1), there are 4× more small jobs than large jobs, the optimal strategy to minimize average job slowdown (Figure 4) in a heavy load is to keep some resources free so that newly arriving small jobs can be immediately allocated. DeepRM learns such a strategy automatically.

5. DISCUSSION

We next elaborate on current limitations of our solution, which motivate several challenging research directions.

Machine boundaries and locality. Our problem formulation assumes a single "large resource pool"; this abstracts away machine boundaries and potential resource fragmentation. This formulation is more practical than might appear since many cluster schedulers make independent scheduling decisions per machine (e.g., upon a hearbeat from the machine as in YARN [36]). By having the agent allocate only one machine's worth of resources, the same formulation can be employed in such scenarios. We also currently do not take into account *data locality* considerations [41, 22]. This can can be done by giving a higher reward to data-local allocations, or perhaps, enabling the agent to learn the true value of data locality with appropriate observation signals.

Job models. We did not model inter-task dependencies that are common in data-parallel jobs. For example, jobs may consist of multiple stages, each with many tasks and different resource requirements [17]. Further, the resource profile of a job may not be known in advance (e.g., for non-recurring jobs [4]), and the scheduler might have get an accurate view only as the job runs [14]. The RL paradigm can in principle deal with such situations of partial observability by casting the decision problem as a POMDP [9]. We intend to investigate alternative job models and robustness to uncertainty in resource profiles in future work.

Bounded time horizon. Notice that DeepRM uses a small time horizon whereas the underlying optimization problem has an infinite time horizon. The bounded horizon is needed for computing the baseline (see §3.3). We hope to overcome this issue by replacing the time-dependent baseline with a *value network* [34] that estimates the average return value.

6. RELATED WORK

RL has been used for a variety of learning tasks, ranging from robotics [25, 24] to industrial manufacturing [26] and computer game playing [34]. Of specific relevance to our work is Zhang and Dietterich's paper [42] on allocating human resources to tasks before and after NASA shuttle missions. Our job scheduling setup has similarities (e.g., multiple jobs and resources), but differs crucially in being an online problem, whereas the NASA task is offline (all input is known in advance). Some early work uses RL for decentralized packet routing in a switch [8], but the problem sizes were small and neural network machinery was not needed. Recently, learning has been applied to designing congestion control protocols using a large number of offline [37] or online [13] experiments. RL could provide a useful framework for learning such congestion control algorithms as well.

Motivated by the popularity of data-parallel frameworks, cluster scheduling has been studied widely recently. Several scheduler designs address specific issues or objectives, such as fairness [41, 16], locality [22, 41], packing [17] and straggling tasks [5]. We are not aware of any work that applies reinforcement learning for data-parallel cluster scheduling.

7. CONCLUSION

This paper shows that it is feasible to apply state-of-the-art Deep RL techniques to large-scale systems. Our early experiments show that the RL agent is comparable and sometimes better than ad-hoc heuristics for a multi-resource cluster scheduling problem. Learning resource management strategies *directly from experience*, if we can make it work in a practical context, could offer a real alternative to current heuristic based approaches.

Acknowledgments. We thank the anonymous HotNets reviewers whose feedback helped us improve the paper, and Jiaming Luo for fruitful discussions. This work was funded in part by NSF grants CNS-1617702 and CNS-1563826.

8. REFERENCES

[1] Terminator, http://www.imdb.com/title/tt0088247/.

[2] M. Abadi, A. Agarwal, P. Barham, E. Brevdo, Z. Chen, C. Citro, G. S. Corrado, A. Davis, J. Dean, M. Devin, et al. Tensorflow: Large-scale machine learning on heterogeneous systems, 2015. *Software available from tensorflow. org*, 2015.

[3] P. Abbeel, A. Coates, M. Quigley, and A. Y. Ng. An application of reinforcement learning to aerobatic helicopter flight. *Advances in neural information processing systems*, page 1, 2007.

[4] S. Agarwal, S. Kandula, N. Bruno, M.-C. Wu, I. Stoica, and J. Zhou. Reoptimizing data parallel computing. In *NSDI*, pages 281–294, San Jose, CA, 2012. USENIX.

[5] G. Ananthanarayanan, S. Kandula, A. G. Greenberg, I. Stoica, Y. Lu, B. Saha, and E. Harris. Reining in the outliers in map-reduce clusters using mantri. In *OSDI*, number 1, page 24, 2010.

[6] M. Armbrust, A. Fox, R. Griffith, A. D. Joseph, R. Katz, A. Konwinski, G. Lee, D. Patterson, A. Rabkin, I. Stoica, et al. A view of cloud computing. *Communications of the ACM*, (4), 2010.

[7] D. P. Bertsekas and J. N. Tsitsiklis. Neuro-dynamic programming: an overview. In *Decision and Control,*. IEEE, 1995.

[8] J. A. Boyan and M. L. Littman. Packet routing in dynamically changing networks: A reinforcement learning approach. *Advances in neural information processing systems*, 1994.

[9] A. R. Cassandra and L. P. Kaelbling. Learning policies for partially observable environments: Scaling up. In *Machine Learning Proceedings 1995*, page 362. Morgan Kaufmann, 2016.

[10] T. Chilimbi, Y. Suzue, J. Apacible, and K. Kalyanaraman. Project adam: Building an efficient and scalable deep learning training system. In *OSDI*, pages 571–582, Broomfield, CO, Oct. 2014. USENIX Association.

[11] J. Dean and L. A. Barroso. The tail at scale. *Communications of the ACM*, pages 74–80, 2013.

[12] C. Delimitrou and C. Kozyrakis. Quasar: Resource-efficient and qos-aware cluster management. ASPLOS '14, pages 127–144, New York, NY, USA, 2014. ACM.

[13] M. Dong, Q. Li, D. Zarchy, P. B. Godfrey, and M. Schapira. Pcc: Re-architecting congestion control for consistent high performance. In *NSDI*, pages 395–408, Oakland, CA, May 2015. USENIX Association.

[14] A. D. Ferguson, P. Bodik, S. Kandula, E. Boutin, and R. Fonseca. Jockey: guaranteed job latency in data parallel clusters. In *Proceedings of the 7th ACM european conference on Computer Systems*. ACM, 2012.

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

[16] A. Ghodsi, M. Zaharia, B. Hindman, A. Konwinski, S. Shenker, and I. Stoica. Dominant resource fairness: Fair allocation of multiple resource types. NSDI'11, pages 323–336, Berkeley, CA, USA, 2011. USENIX Association.

[17] R. Grandl, G. Ananthanarayanan, S. Kandula, S. Rao, and A. Akella. Multi-resource packing for cluster schedulers. SIGCOMM '14, pages 455–466, New York, NY, USA, 2014. ACM.

[18] M. T. Hagan, H. B. Demuth, M. H. Beale, and O. De Jesús. *Neural network design*. PWS publishing company Boston, 1996.

[19] W. K. Hastings. Monte carlo sampling methods using markov chains and their applications. *Biometrika*, (1), 1970.

[20] B. Heller, S. Seetharaman, P. Mahadevan, Y. Yiakoumis, P. Sharma, S. Banerjee, and N. McKeown. Elastictree: Saving energy in data center networks. NSDI'10, Berkeley, CA, USA, 2010. USENIX Association.

[21] G. Hinton. Overview of mini-batch gradient descent. *Neural Networks for Machine Learning*.

[22] M. Isard, V. Prabhakaran, J. Currey, U. Wieder, K. Talwar, and A. Goldberg. Quincy: fair scheduling for distributed computing clusters. In *ACM SIGOPS*, 2009.

[23] J. Junchen, D. Rajdeep, A. Ganesh, C. Philip, P. Venkata, S. Vyas, D. Esbjorn, G. Marcin, K. Dalibor, V. Renat, and Z. Hui. A control-theoretic approach for dynamic adaptive video streaming over http. SIGCOMM '15, New York, NY, USA, 2015. ACM.

[24] L. P. Kaelbling, M. L. Littman, and A. W. Moore. Reinforcement learning: A survey. *Journal of artificial intelligence research*, 1996.

[25] J. Kober, J. A. Bagnell, and J. Peters. Reinforcement learning in robotics: A survey. *The International Journal of Robotics Research*, 2013.

[26] S. Mahadevan and G. Theocharous. Optimizing production manufacturing using reinforcement learning. In *FLAIRS Conference*, 1998.

[27] I. Menache, S. Mannor, and N. Shimkin. Basis function adaptation in temporal difference reinforcement learning. *Annals of Operations Research*, (1), 2005.

[28] V. Mnih, A. P. Badia, M. Mirza, A. Graves, T. P. Lillicrap, T. Harley, D. Silver, and K. Kavukcuoglu. Asynchronous methods for deep reinforcement learning. *CoRR*, 2016.

[29] V. Mnih, K. Kavukcuoglu, D. Silver, A. Graves, I. Antonoglou, D. Wierstra, and M. A. Riedmiller. Playing atari with deep reinforcement learning. *CoRR*, 2013.

[30] V. Mnih, K. Kavukcuoglu, D. Silver, A. A. Rusu, J. Veness, M. G. Bellemare, A. Graves, M. Riedmiller, A. K. Fidjeland, G. Ostrovski, S. Petersen, C. Beattie, A. Sadik, I. Antonoglou, H. King, D. Kumaran, D. Wierstra, S. Legg, D. H. I. Antonoglou, D. Wierstra, and M. A. Riedmiller. Human-level control through deep reinforcement learning. *Nature*, 2015.

[31] G. E. Monahan. State of the art - a survey of partially observable markov decision processes: theory, models, and algorithms. *Management Science*, (1), 1982.

[32] J. Schulman, S. Levine, P. Moritz, M. I. Jordan, and P. Abbeel. Trust region policy optimization. *CoRR, abs/1502.05477*, 2015.

[33] D. Silver, A. Huang, C. J. Maddison, A. Guez, L. Sifre, G. van den Driessche, J. Schrittwieser, I. Antonoglou, V. Panneershevlvam, M. Lanctot, S. Dieleman, D. Grewe, J. Nham, N. Kalchbrenner, I. Sutskever, T. Lillicrap, M. Leach, K. Kavukcuoglu, T. Graepel, and D. Hassabis. Mastering the game of go with deep neural networks and tree search. *Nature*, 2016.

[34] R. S. Sutton and A. G. Barto. *Reinforcement Learning: An Introduction*. MIT Press, 1998.

[35] R. S. Sutton, D. A. McAllester, S. P. Singh, Y. Mansour, et al. Policy gradient methods for reinforcement learning with function approximation. In *NIPS*, 1999.

[36] V. K. Vavilapalli, A. C. Murthy, C. Douglas, S. Agarwal, M. Konar, R. Evans, T. Graves, J. Lowe, H. Shah, S. Seth, B. Saha, C. Curino, O. O'Malley, S. Radia, B. Reed, and E. Baldeschwieler. Apache hadoop yarn: Yet another resource negotiator. SOCC '13, pages 5:1–5:16, New York, NY, USA, 2013. ACM.

[37] K. Winstein and H. Balakrishnan. TCP Ex Machina: Computer-generated Congestion Control. In *SIGCOMM*, 2013.

[38] K. Winstein, A. Sivaraman, and H. Balakrishnan. Stochastic forecasts achieve high throughput and low delay over cellular networks. In *NSDI*, pages 459–471, Lombard, IL, 2013. USENIX.

[39] S. Yi, Y. Xiaoqi, J. Junchen, S. Vyas, L. Fuyuan, W. Nanshu, L. Tao, and B. Sinopoli. Cs2p: Improving video bitrate selection and adaptation with data-driven throughput prediction. SIGCOMM, New York, NY, USA, 2016. ACM.

[40] X. Yin, A. Jindal, V. Sekar, and B. Sinopoli. Via: Improving internet telephony call quality using predictive relay selection. In *SIGCOMM*, SIGCOMM '16, 2016.

[41] M. Zaharia, D. Borthakur, J. Sen Sarma, K. Elmeleegy, S. Shenker, and I. Stoica. Delay scheduling: a simple technique for achieving locality and fairness in cluster scheduling. In *EuroSys*, 2010.

[42] W. Zhang and T. G. Dietterich. A reinforcement learning approach to job-shop scheduling. In *IJCAI*. Citeseer, 1995.