Assignment

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.
e 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.

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.

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

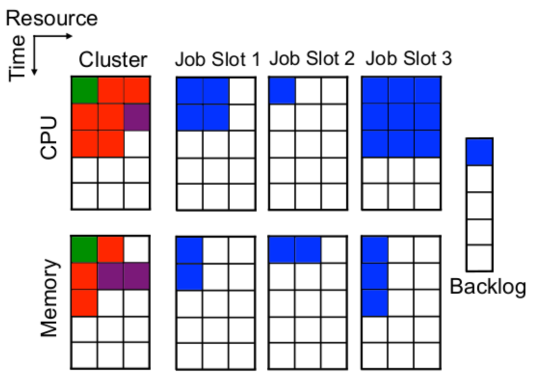


Figure 2: An example of a state representation, with two resources and three pending job slots.

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]

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 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.

P

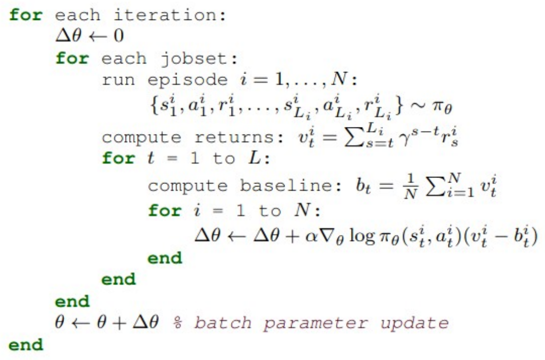
3 Training algorithm

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 before. 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 before.

Recall that REINFORCE estimates the policy

A drawback of this 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 *timestep* t across all episodes[4] with the same jobset



```
for each iteration:
    $\Delta\theta \leftarrow 0$
    for each jobset:
        run episode $i = 1, \ldots, N$:
            $\{s_1^i, a_1^i, r_1^i, \ldots, s_{L_i}^i, a_{L_i}^i, r_{L_i}^i\} \sim \pi_\theta$
        compute returns: $v_t^i = \sum_{s=t}^{L_i} \gamma^{s-t} r_s^i$
        for $t$ = 1 to $L$:
            compute baseline: $b_t = \frac{1}{N} \sum_{i=1}^{N} v_t^i$
            for $i$ = 1 to $N$:
                $\Delta\theta \leftarrow \Delta\theta + \alpha \nabla_\theta \log \pi_\theta(s_t^i, a_t^i)(v_t^i - b_t^i)$
            end
        end
    end
    $\theta \leftarrow \theta + \Delta\theta$ % batch parameter update
end
```

Figure 3: Pseudo-code for training algorithm.

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 $-|J|$ (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.

5 Requirements

1). Upgrade the RL training prat into meta-learning part to achieve fast adaptation, compare it with unchanged RL method to show the performance; change the reward function to try different objective, providing figures;

2).Upgrade the converged result, achieving more percentage of tasks which meet the objective, providing figures;

3). Compare the training result of meta-learning with existed methods in codes(Tetris, random), providing figures.

[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); nevertheless, this approach yielded good results in practice.

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