Why is RL a good fit?

- Resource management problems are ubiquitous in computer systems and networks. They often manifest as difficult online decision making tasks where appropriate solutions depend on understanding the workload and environment.
- Traditionally, the typical design flow is:
  - come up with clever heuristic for a simplified model of the problem
  - painstakingly test and tune the heuristics for good performance in practice.
- Can systems learn to manage resources on their own?

In Reinforcement Learning, an agent interacts with an environment. The agent observes some state, and takes an action based on its policy $\pi_\theta$. Through the interactions, the environment evolves its states and feedbacks the agent reward signals. The goal is to maximize total discounted award $\sum_{t=0}^{\infty} y^t r_t$.

The agent learns to tune its policy parameter $\theta$ to achieve higher expected total reward, through its experience in state action function $Q$: $Q(s, a) = \mathbb{E}_\pi \left[ \sum_{t=0}^{\infty} y^t r_t \mid s_0 = s, a_0 = a \right]$.

In practice, the training of parameter $\theta$ follows policy gradient, and the above $Q$ can be obtained by samples $v$:

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

Why is RL a good fit?

- Computer systems generate a large amount of data for training
- A natural framework for easy-to-identify signals and observations
- Optimize the policy directly from experience
- Train for objectives that are hard-to-optimize analytically
- Adapt towards different workloads in varying conditions

**Components of Reinforcement Learning**

- Agent
- Environment
- State $s_t$ and action $a_t$
- Reward $r_t$ at state $s_t$ and action $a_t$

**State:**
- Cluster: resource pool of multiple types with a provisioning time
- Jobs: blocks of resource demand and duration in time

**Action:** Select which new job to put into the cluster, assuming no preemption and fixed allocation profile

**Dynamics:** New job(s) arrive along the time, while allocated jobs blocks move up simulating jobs being processed in the cluster

**Objective:** average job slowdown, given by $\text{completion time} / \text{job duration}$

**Reward:** $-1 / \text{job duration}$ penalty for all jobs in the system

**Algorithm:**

```python
for each iteration:
    $\Delta \theta = 0$
    for each jobset:
        run episode $i = 1, \ldots, N$:
            $\{s_i, a_i, r_i, \ldots, s_{i,N}, a_{i,N}, r_{i,N} \} \sim \pi_\theta$
            compute returns: $v_i = \sum_{t=0}^{N} y^t r_i$
            for $i = 1$ to $L$:
                compute baseline: $b_i = \frac{1}{L} \sum_{t=0}^{L} v_i$
                for $i = 1$ to $N$:
                    $\Delta \theta \leftarrow \Delta \theta + \alpha \nabla_\theta \log \pi_\theta(s_i, a_i)(v_i - b_i)$
    end
end
$\theta \leftarrow \theta + \Delta \theta$ % batch parameter update
```

Train the policy neural network using REINFORCE algorithm with an episodic setting:

- Sample batches of episodes, where a set of jobs arrive and get scheduled, and we evaluate the cumulative reward following each decision.
- Update neural network parameters based on the policy gradient for the batch.

Intuition: the algorithm compare the outcome from each decision and tune the policy to perform more likely on the decisions that lead to better return.

**EVALUATION**

- In a multi-resource bi-model (many small jobs mixed with sporadic big jobs) distributed workload, DeepRM outperforms existing schemes in all workloads.

- By designing different reward signal, DeepRM can tune towards different objectives. E.g., -1 penalty corresponds to minimizing job completion time.

- Learning curve and training procedure of DeepRM

- Where are the gains from: being non-work conservative, holding big jobs to leave room for small jobs, resulting in better slowdown for small jobs. DeepRM learns this strategy.