A Distriubuted Implementation for Reinforcement Learning

Yi-Chun Chen[1] **and Yu-Sheng Chen**[1]

[1]ICME, Stanford University

Abstract. *In this CME323 project, we implement a distributed algorithm for model-free reinforcement learning. Each processor independently makes decision, interacts with the environment, and update one Q-function that is shared by all processors. After updating, processors again make decisions according to this shared Q-function. The distributed structure can be considered multi-agent learning, since each processor is an independent learner but sharing some information with others. Empirical test on one benchmark problem shows that this distributed implementation indeed accelerates the convergence to optimal policy.*

1. Reinforcement Learning

In this section, we review the basic concepts of reinforcement learning and the model-free method that will be paralleled in later section.

1.1. Introduction

Reinforcement learning [1] is an area of machine learning, concerned with how agents ought to take actions in an environment as to maximize cumulative reward. The environment is usually formulated as Markov decision process. However, the details of this Markov decision process, such as state transition model and reward model, are not given to agents in advance. Therefore, agents must estimate the environment from experience and make decisions based on their estimated models of environment.

There are two approaches to conquer the reinforcement learning problems. One is model-based method [2]. Agent estimates transition and reward models directly from experience. For instance, in maximum likelihood model-based method, agent estimates transition model by counting $N(s, a, s')$, which is the number of events that the environment transits from state s to state s' by acting action a. Once agent estimates the model, the corresponding optimal policy can be obtained by value-iteration or policy iteration.

The other approach is model-free method. In contrast to model-based methods, model-free reinforcement learning does not require building explicit representations of the transition and reward models. Avoiding explicit representations is attractive, especially when the problem is high dimensional. Model-free methods estimate the Q-function directly, which is a function of state-action pairs (s, a) and estimates the maximal cumulative reward after acting action a on state s. Q-learning [3], SARSA [1], and eligibility traces [1] are well-known model-free methods.

1.2. Q-learning

Q-learning [3] is one of the most popular reinforcement learning algorithm. The idea is to apply incremental estimation of the Bellman equation of Markov decision process

$$Q(s, a) = R(s, a) + \sum_{s'} T(s' \mid s, a)\text{max}_{a'} Q(s', a). \tag{1}$$

Instead of using T and R, the transition and reward models, incremental update rule only uses current state s, action a, next state s', and reward r to estimate Q-function:

$$Q(s,a) \leftarrow Q(s,a) + \alpha \left[r + \gamma \text{max}_{a'} Q(s',a') - Q(s,a) \right], \tag{2}$$

where $\alpha \in [0,1]$ is the learning rate and $\gamma \in [0,1]$ is the discount factor.

Once $Q(s,a)$ is updated, the optimal action is simply $a^* = \text{argmax}_{a'} Q(s,a')$. However, keeping choosing the optimal action from estimated $Q(s,a)$ might lead to over-exploiting. In reinforcement learning, it is important to balance exploitation and exploration. For example, if we continuously explore the environment, we will obtain more comprehensive understanding of the model but be unable to cumulate rewards; if we continuously make the decisions we believe is best without ever trying new strategy, as always choosing $a^* = \text{argmax}_{a'} Q(s,a')$, then we may miss out on improving our strategy and accumulating more reward. To balance exploitation and exploration, in this project, ϵ-greedy strategy is used. With probability $1-\epsilon$, the best action $a^* = \text{argmax}_{a'} Q(s,a')$ is chosen; with probability ϵ, we randomly and uniformly choose other actions.

Agorithm 1 and Figure 1 summarize the pseudocode and structure of Q-learning.

Algorithm 1 Single-processor Q-learning

function Q-LEARNING
- $t \leftarrow 0$
- $s_0 \leftarrow$ initial state
- Initialize Q
- **loop**
  - Choose action a_t based on Q and some exploration strategy (ϵ-greedy)
  - Observe new state s_{t+1} and reward r_t
  - $Q(s_t,a_t) \leftarrow Q(s_t,a_t) + \alpha \left[r_t + \gamma \text{max}_{a'} Q(s_{t+1},a') - Q(s_t,a_t) \right]$
  - $t \leftarrow t+1$

2. Distributed Implementation of Q-Learning

In this section, we propose a distributed version of Q-learning, and further analyze the algorithm structure and runtime.

2.1. Structure and Algorithm

Assume we have P processors. We can treat each processors as an independent agent. On the other hand, all agents can get access to a shared Q-function and make decisions accordingly. After receiving the feedback from the environment, all agents update the shared Q-function. The structure is shown in Figure 2. Purple circles represent processors. Each one independently interacts with the environment, as the blow arrows indicate. In addition, their corresponding current states can be different. Q-function help all agents to make decisions and require their updates, as shown by brown arrows. If there is only one purple circle, then the structure in Figure 2 coincides the structure in Figure 1. Note that communication between processors and Q functions are all-to-one for updating and one-to-all structure for decision making.

Algorithm 2 summarizes the pseudocode for distributed Q-learning. Each processor p records its own current state $s_{p,t}$, chooses action $a_{p,t}$, and receives next state $s_{p,t+1}$

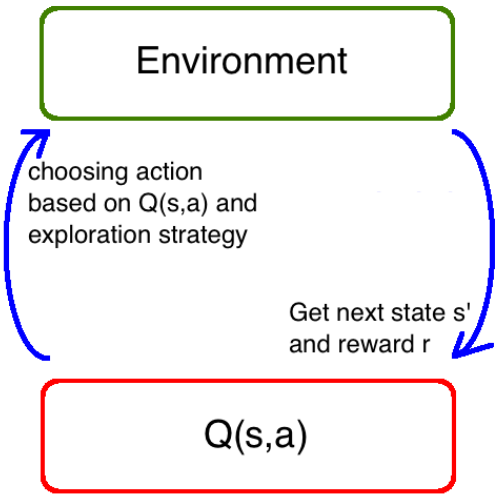

Figure 1. Single-processor Q-learning

and reward $r_{p,t}$ from the environment. Note that, for each processor, there is no need to wait other processors while making decisions or updating Q-function. Once it receives the feedback from the environment, it updates immediately. This no-waiting update rule will not cause harm to the convergence, since the expected Q-functions for all agents are the same which is the true optimal Q-function. The stopping criteria for loop can be set as reaching maximum number of iterations or converging within predetermined norm difference of Q-functions. In our implementation, we choose the former as the stopping criteria.

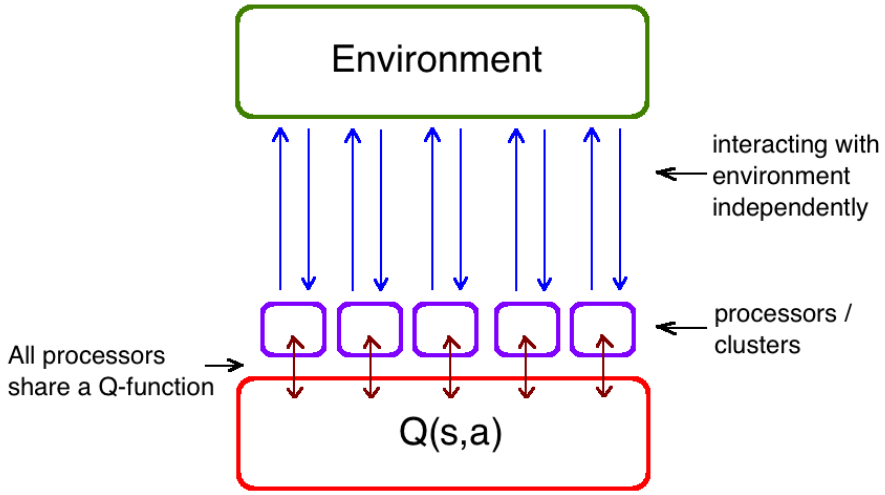

Figure 2. Multi-processor Q-learning

2.2. Runtime Analysis

Apparently, by using P processors for Q-learning, we are able to obtain P times number of samples than one processor case during any time period. More samples means con-

Algorithm 2 Multi-processor Q-learning

function DISTRIBUTED Q-LEARNING
  Initialize Q
  for each processor (agent) p parallelly **do**
    $t_p \leftarrow 0$
    $s_{p,0} \leftarrow$ initial state
    loop
      Choose action a_{p,t_p} based on Q and some exploration strategy (ϵ-greedy)
      Observe new state s_{p,t_p+1} and reward r_{p,t_p}
      $Q(s_{p,t_p}, a_{p,t_p}) \leftarrow Q(s_{p,t_p}, a_{p,t_p}) + \alpha \left[r_{p,t_p} + \gamma \text{max}_{a'} Q(s_{p,t_p+1}, a') - Q(s_{p,t_p}, a_{p,t_p}) \right]$
      $t_p \leftarrow t_p + 1$

Table 1. Work, depth, and shuffle size on each processor

	Work	Depth
Make Decision	$O(\|A\|)$	$O(\log(\|A\|))$
Update Q	$O(\|A\|)$	$O(\log(\|A\|))$
Shuffle Size	$O(\|A\|)$	

verging to optimal policy faster. The speed-up is more apparent at the beginning period of the reinforcement learning, as shown in the experiment section.

Table 1 further analyzes the work and depth for each processor. When making decision, $a^* = \text{argmax}_{a'} Q(s, a')$ can be obtained by work $O(|A|)$ and depth $O(\log(|A|))$, where $|A|$ is the number of actions. When updating Q-function, calculating $\text{max}_{a'} Q(s_{p,t_p+1}, a')$ also requires work $O(|A|)$ and depth $O(\log(|A|))$. The shuffle size between each processor and Q-function is $O(|A|)$, since each processor p only needs the array $[Q(s_{p,t}, 1), Q(s_{p,t}, 2), Q(s_{p,t}, 3), \ldots, Q(s_{p,t}, |A|)]$ to make decision and $[Q(s_{p,t+1}, 1), Q(s_{p,t+1}, 2), Q(s_{p,t+1}, 3), \ldots, Q(s_{p,t+1}, |A|)]$ to update.

3. Experiment

In this section, we show the implementation of the proposed distributed Q-learning with programming language Scala on platform Spark [4], and also test it on one classic reinforcement learning problem, the 4×3 maze [5].

3.1. Spark Implementation

The distributed algorithm is implemented in Scala as shown in Figure 3. After launching the Spark Context, the initial state of each agent is stored distributedly in the cluster with default partition of Spark. Each agent is modeled as a Mapper process which executes function $mapper$ and outputs current state, action, local Q value, and its own next state. The function $mapper$ implementation is shown in Figure 4. After agents finish their local processes, P Q-value outcomes are calculated among the cluster. To complete the update of Q table, all agents concurrently write to the Q table stored in the driver machine. This method only requires $O(1)$ depth since there's no dependency between agents. And only one of the agents sharing the same entry of the Q table can update the Q table stored in

the driver machine. Another method of Q value integration is first average all Q values corresponding to the same table entry, then update the final Q table using the α-update rule. This method requires $O(\log(P))$ depth at most, since there are at most $O(P)$ agents sharing the same Q table entry. Finally, each agent updates its own state obtained from the environment, which completes this iteration.

```
def Q_learning_4x3_new(s:Int, p:Int, t_max:Int, mode:String) : Array[Array[Double]] = {

    // s is the initial state
    // t_max is the max number of iteration
    // gamma is the discount factor

    var t = 0

    val pS = Array.fill(p)(s) // same initial state

    while (t < t_max)
    {
        val distPS = sc.parallelize(pS) // partition by default
        val res = distPS.map(mapper).cache().collect()

        // update Q
        updateQ(res, mode)
        for (i <- 0 to (p-1))
        {
            pS(i) = res(i)._4
        }

        t = t + 1
        if (t % 100 == 0) println(s"$t")
    }

    return Q
}
```

Figure 3. The Q-learning main flow in Scala

```
def mapper(s:Int) : Tuple4[Int, Int, Double, Int] = {

    // control
    var a:Int = eps_greedy(Q(s), eps_explore)
    // sample
    var pair_sp_r = sample_in_machine(T,R,s,a)
    // update Q table
    var qval = Q(s)(a) + alpha * (pair_sp_r._2 + gamma * Q(pair_sp_r._1).max - Q(s)(a))
    return (s, a, qval, pair_sp_r._1)
}
```

Figure 4. The agent process executed in parallel in Scala

3.2. 4×3 Maze

The 4×3 maze is depicted in Figure 5. The environment have 11 states, each corresponding to a grid position, and 4 actions, each corresponding to move in a direction. There are two states leading to reward, as shown in Figure 5, one for reward 1 and the other for reward -1. When robot reaches $+1$ grid or -1 grid, the game restarts.

Figure 6 shows the relation between runtime and cumulative reward with different number of processors. Cumulative reward is defined as $\sum_{t=0}^{\infty} \gamma^t r_t$. In addition, $\gamma = 0.95$ is given by the experiment setup. The learning rate α is 0.1 and the exploration rate ϵ

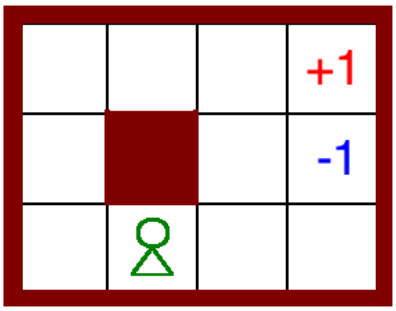

Figure 5. The 4×3 Maze

is 0.1. Figure 6 demonstrates that if we have more processors, then we can converge to optimal policy faster. With 10 processors, we obtain optimal policy within 0.5 second. With 3 processors, optimal policy is obtained within 1.7 second. With only 1 processor, optimal policy can not be obtained within 3 seconds. Furthermore, with more processors, samples can cover the state space more uniformly and the resulting Q-function would be less biased. For example, the $p = 1$ curve in Figure 6 is strongly zigzagging, since the learned policy is unstable due to sampling bias. While $p = 3$ and $p = 10$ curves are more smooth.

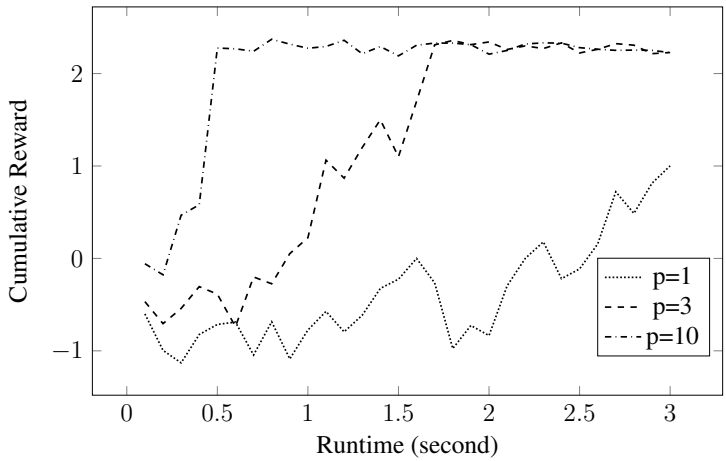

Figure 6. The relation between runtime and cumulative reward with respect to different number of processors

4. Conclusion

In this project, we demonstrate a distributed version of Q-learning, and analyze runtime and algorithm structure. We further implement the algorithm with language Scala on platform Spark. The Empirical demonstration on a benchmark problem shows that the distributed implementation can accelerate the convergence to optimal policy.

References

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

[2] P. R. Kumar, “A Survey of Some Results in Stochastic Adaptive Control”, *SIAM Journal on Control and Optimization*, vol. 23, no. 3, pp. 329-380, 1995.

[3] C. J. Watkins and P. Dayan, “ Q-learning”. Machine learning, vol. 8, pp. 279-292, 1992.

[4] Apache Spark, `http://spark.apache.org/`

[5] S. Russell and N. Peter, *Artificial Intelligence: A Modern Approach*, 3rd ed. Upper Saddle River, NJ: Pearson, 2010.