APPLYING REINFORCEMENT LEARNING TO A 4×4 TIC TAC TOE GAME

Shengjun(Daniel) Zhang*
*Cyber-Physical Energy Systems Laboratory, Department of Electrical Engineering,
University of North Texas, Denton, TX 76207 USA

***Abstract*—Reinforcement learning is essential for applications where there is no single correct way to solve a problem. In this project, I show that reinforcement learning is very effective at learning how to play the game *Tic-Tac-Toe*, despite the high-dimensional state. [1] The agent is not given information about what the blocks or grids look like - it must learn these representations and directly use the reward and *Q*-values to develop an optimal strategy. The *Q*-agent uses basic *Q*-Learning algorithm, and shows that it is able to achieve super-human performance.**

***Index Terms*—Reinforcement Learning, Tic Tac Toe, Markov Decision Process(MDP), *Q*-Learning.**

I. INTRODUCTION

THE idea that we learn by interacting with our environment is probably the first to occur to us when we think about the nature of learning. When an infant plays, waves its arms, or looks about, it has no explicit teacher, but it does have a direct sensorimotor connection to its environment. Exercising this connection produces a wealth of information about cause and effect, about the consequences of actions, and about what to do in order to achieve goals. Throughout our lives, such interactions are undoubtedly a major source of knowledge about our environment and ourselves. Whether we are learning to drive a car or to hold a conversation, we are acutely aware of how our environment responds to what we do, and we seek to influence what happens through our behavior. Learning from interaction is a foundational idea underlying nearly all theories of learning and intelligence. [2]

Based on the idea above, people explored an approach called *reinforcement learning* to solve this kind of problems, which is much more focused on goal-directed learning from interaction than are other approaches to machine learning. [2]

A. Markov Decision Process

In reinforcement learning, the decision maker(agent) aims to achieve the optimal behavior in the presence of uncertainty by interacting with the environment, which is usually modeled as a Markov Decision Process(MDP). [3]
Markov Decision process provides a formalism for reasoning about planning and acting in the face of uncertainty. There are many possible ways of defining MDP, and many of these definitions are equivalent up to small transformations of the problem. One definition is that an MDP M is a tuple$(S, D, A, P_{sa}, \gamma, R)$ consisting of [4]

Corresponding author: S. Zhang (email: ShengjunZhang@my.unt.edu).

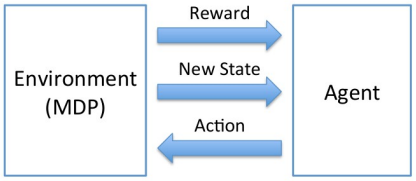


Fig. 1: Typical RL problem. An RL agent interacts with its environment and, upon observing the consequences of its actions, can learn to alter its own behaviour in response to rewards received.

- S: A set of possible **states** of the world.
- D: An **initial state distribution**(a probability distribution over S).
- A: A set of possible **actions** from which we may choose on each time step.
- P_{sa}: The **state transition distributions**. For each state $s \in S$ and action $a \in A$, this gives the distribution over to which state we will randomly transition if we take action a in state s.
- γ: A number in $[0, 1]$ called the **discount factor**.
- R: $S \mapsto \mathbb{R}$: The reward function, bounded by R_{max}. ($|R(s)| \leq R_{max}$ for all s).

B. Goal of Reinforcement learning

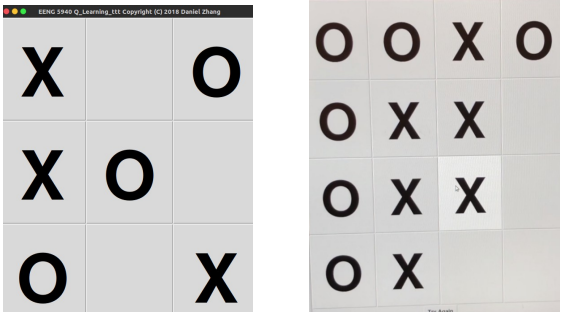
Events in an MDP proceed as follows. We begin in an initial state s_0 drawn from the distribution D. At each time step t, we then have to pick an action a_t, as a result of which our state transitions to some s_{t+1} drawn from the probability transition distribution $P_{s_t a_t}$. By repeatedly picking actions, we traverse some sequence of states $s_0, s_1, ...$ Our total payoff is then the sum of discounted rewards along this sequence of states

$$R(s_0) + \gamma R(s_1) + \gamma^2 R(s_2) + \cdots. \quad (1)$$

In reinforcement learning, our goal is to find a way of choosing actions $a_0, a_1, \cdots$ over time, so as to maximize the expected value of the rewards given in Equation(1).

C. Challenges in RL

It is instructive to emphasise some challenges faced in RL:



(a) 3×3 Tic Tac Toe (b) 4×4 Tic Tac Toe

Fig. 2: 3×3 Tic Tac Toe **VS** 4×4 Tic Tac Toe

- The optiaml policy must be inferred by trial-and-error interaction with the environment. The only learning signal the agent receives is the reward.
- The observations of the agent depend on its actions and can contain strong temporal corrections.
- Agents must deal with long-range time dependencies: Often the consequences of an action only materialise after many transitions of the environment. This is known as the (temporal) *credit assignment problem*.

D. Tic Tac Toe

Consider the familiar childs game of tic-tac-toe. Two players take turns playing on a three-by-three board. One player plays Xs and the other Os until one player wins by placing three marks in a row, horizontally, vertically, or diagonally, as the X player has in the game shown to the right. If the board fills up with neither player getting three in a row, the game is a draw. Because a skilled player can play so as never to lose, let us assume that we are playing against an imperfect player, one whose play is sometimes incorrect and allows us to win. For the moment, in fact, let us consider draws and losses to be equally bad for us. How might we construct a player that will find the imperfections in its opponents play and learn to maximize its chances of winning?

Although this is a simple problem, it cannot readily be solved in a satisfactory way through classical techniques. For example, the classical ***minimax*** solution from game theory is not correct here because it assumes a particular way of playing by the opponent.

In this report, I will discuss using reinforcement learning algorithm solves a 4×4 Tic Tac Toe game instead of a 3×3 one, which is shown in Fig.2.

II. Related Work

The related work in this area is primarily by Dr. Lisa Meeden, who is a Full Professor in the Computer Science Department at Swarthmore College. [5] They used *Q*-learning, a temporal difference algorithm, to try to learn the optimal playing strategy. *Q*-learning creates a table that maps states and actions to expected rewards. The goal of temporal difference learning is to make the learner's current prediction for the reward that will be received by taking the action in the current state more closely match the next prediction at the next time step. However, further improvements involve reducing the size of the *Q*-table for tic-tac-toe by using the symmetry of the board, using a neural network to approximate the *Q*-table instead of explicity representing the *Q*-values table.

III. Reinforcement Learning

So far, we have introduced the key formalism used in RL, the MDP, and briefly noted some challenges in RL. In the following, we will distinguish between different classes of RL algorithms. There are two main approaches to solving RL problems: methods based on *value functions* and methods based on *policy search*. There is also a hybrid, *actor-critic* approach, which employs both value functions and policy search. I will now explain these approaches very briefly.

A. Value Functions

Value function methods are based on estimating the value (expected return) of being in a given state. The *state-value function* $V^\pi(s)$ is the expected return when starting in state s and following π henceforth:

$$V^\pi(s) = \mathbb{E}[R|s,\pi] \tag{2}$$

The optimal policy, π^*, has a corresponding state-value function $V^*(s)$, and vice-versa, the optimal state-value function can be defined as:

$$V^*(s) = \max_\pi V^\pi(s), \forall s \in S \tag{3}$$

If we had $V^*(s)$ available, the optimal policy could be retrieved by choosing among all actions available at s_t and picking the action a that maximizes $\mathbb{E}_{s_{t+1}\sim\tau(s_{t+1}|s_t,a)}[V^*(s_{t+1})]$.

In the RL setting, the transition dynamics τ are unavailable. Therefore, we construct another function, the *state-action value* or *quality function* $Q^\pi(s,a)$, which is similar to V^π, except that the initial action a is provided, and π is only followed from the succeeding state onwards:

$$Q^\pi(s,a) = \mathbb{E}[R|s,a,\pi]. \tag{4}$$

The best policy, given $Q^\pi(s,a)$, can be found by choosing a greedily at every state: $\operatorname{argmax}_a Q^\pi(s,a)$. Under this policy, we can also define $V^\pi(s)$ by maximizing $Q^\pi(s,a) : V^\pi(s) = \max_a Q^\pi(s,a)$.

Dynamic Programming: To actually learn Q^π , we exploit the Markov property and define the function as a Bellman equation [6], which has the following recursive form:

$$Q^\pi(s_t,a_t) = \mathbb{E}_{s_{t+1}}[r_{t+1} + \gamma Q^\pi(s_{t+1}, \pi(s_{t+1}))]. \tag{5}$$

This means that Q^π can be improved by *bootstrapping*, i.e., we can use the current values of our estimate of Q^π to improve our estimate. This is the foundation of *Q*-learning [7] and the state-action-reward-state-action (SARSA) algorithm [8]:

$$Q^\pi(s_t,a_t) \leftarrow Q^\pi(s_t,a_t) + \alpha\delta, \tag{6}$$

where α is the learning rate and $\delta = Y - Q^\pi(s_t,a_t)$ the temporal difference(TD) error; here, Y is a target as in a standard regression problem. SARSA, an *on-policy* learning algorithm,

is used to improve the estimate of Q^π by using transitions generated by the behavioural policy (the policy derived from Q^π), which results in setting $Y = r_t + \gamma Q^\pi(s_{t+1}, a_{t+1})$. *Q*-learning is *off-policy*, as Q^π is instead updated by transitions that were not necessarily generated by the derived policy. Instead, *Q*-learning uses $Y = r_t + \gamma \max_a Q^\pi(s_{t+1}, a)$, which directly approximates Q^*.

To find Q^* from an arbitrary Q^π , we use *generalised policy iteration*, where policy iteration consists of *policy evaluation* and *policy improvement*. Policy evaluation improves the estimate of the value function, which can be achieved by minimixing TD errors from trajectories experienced by following the policy. As the estimate improves, the policy can naturally be improved by choosing actions greedily based on the updated value function. Instead of performing these steps separately to convergence (as in policy iteration), generalised policy iteration allows for interleaving the steps, such that progress can be made more rapidly. [6]

B. Policy Search

Policy search methods do not need to maintain a value function model, but directly search for an optimal policy π^* . Typically, a parameterised policy π_θ is chosen, whose parameters are updated to maximise the expected return $\mathbb{E}[R|\theta]$ using either gradient-based or gradient-free optimization. Neural networks that encode policies have been successfully trained using both gradient-free and gradient-based methods. Gradient-free optimization can effectively cover low-dimensional parameter spaces, but despite some successes in applying them to large networks, gradient-based training remains the method of choice for most DRL algorithms, being more sample-efficient when policies possess a large number of parameters [6].

When constructing the policy directly, it is common to output parameters for a probability distribution; for continuous actions, this could be the mean and standard deviations of Gaussian distributions, whilst for discrete actions this could be the individual probabilities of a multinomial distribution. The result is a stochastic policy from which we can directly sample actions. With gradient-free methods, finding better policies requires a heuristic search across a predefined class of models. Methods such as evolution strategies essentially perform hill-climbing in a subspace of policies whilst more complex methods, such as compressed network search, impose additional inductive biases [9]. Perhaps the greatest advantage of gradient-free policy search is that they can also optimize non-differentiable policies.

Based on the content we learned in class, I just introduced some basic background about *Policy Search*.

IV. Q-Learning

First of all, the general *Q*-Learning algorithm is given as in the right algorithm table:

where $\mathcal{O}(n|\mathcal{A}|)$ and $\mathcal{O}(n)$ denote the complexity. The δ calculated on line 6 of Algorithm 1 highlights the difference between the better estimate of the Q function based on a single interaction, $Q^+(s,a)$, and the current estimate, $Q(s,a)$.

Algorithm 1: *Q*-Learning

Input: MDP $\diagdown\{\mathcal{P}, \mathcal{R}\}, \alpha, \epsilon$
Output: π
1: $\theta \leftarrow$ Initialize arbitrarily
2: $\langle s, a\rangle \leftarrow \langle s_0, \pi^\epsilon(s^0)\rangle$
3: **while** *time left* **do**
4: Take action a and receive reward r and next state s'
5: $Q^+(s,a) \leftarrow r + \gamma \max_{a'} Q(s', a')$, $\{\mathcal{O}(n|\mathcal{A}|)\}$
6: $\delta \leftarrow Q^+(s,a) - Q(s,a)$
7: $\theta \leftarrow \theta + \alpha\delta\phi(s,a)$, $\{\mathcal{O}(n)\}$
8: $\langle s, a\rangle \leftarrow \langle s', \pi^\epsilon(s')\rangle$, $\{\mathcal{O}(n|\mathcal{A}|)\}$
9: **end while**
10: **return** π greedy w.r.t Q

This quantity is called the *temporal difference*(TD) error in the literature [2]. The feature function ϕ maps each state-action pair to a vector of feature values; θ is the weight vector specifying the contribution of each feature across all state-action pairs.

However, in our case, this general algorithm can be simplied, i.e. $\phi = 1$, because we don't need to consider the weight between state and action pairs at all, and eventurally our θ will converge to the real Q-value, which simplified our implementation. Overall, the algorithm that I used for this game is given as below:

Algorithm 2: *Q*-Learning for Tic-Tac-Toe

Input: MDP $\diagdown\{\mathcal{P}, \mathcal{R}\}, \alpha, \epsilon$
Output: π
1: $\theta \leftarrow$ Initialize arbitrarily
2: $\langle s, a\rangle \leftarrow \langle s_0, \pi^\epsilon(s^0)\rangle$
3: **while** *time left* **do**
4: Take action a and receive reward r and next state s'
5: $Q^+(s,a) \leftarrow r + \gamma \max_{a'} Q(s', a')$
6: $\delta \leftarrow Q^+(s,a) - Q(s,a)$
7: $\theta \leftarrow \theta + \alpha\delta$
8: $\langle s, a\rangle \leftarrow \langle s', \pi^\epsilon(s')\rangle$
9: **end while**
10: **return** π greedy w.r.t Q

V. Implementation

A. Q-Learning

1) Reward Function: I considered X to be the maximizer and O to be the minimizer. Therefore, a win for X will result in an external reward of **+1**, while a win for O will result in an external reward of **-1**. Any other state of the game will result in an external reward of **0** (including tied games). I assumed that either player may go first.

2) Pseudocode for Learning Session: details are in the table of Algorithm 3 [5]. For this 4×4 case, based on the high dimensional *Q*-table, the complexity of *Q*-learning algorithm and the computer capacity, after evaluation all the factors, I choose the parameter of *maxGames* is 50,000,000(i.e. learning iteration is 50,000,000).

Algorithm 3: Pseudocode for Learning Session

```
function gameLearning (startState,maxGames)
state = startState
games = 0
while games < maxGames do
    stateKey = makeKey(state, player)
    if stateKey not in table then
        addKey(stateKey)
    end
    action = chooseAction(stateKey, table)
    nextState = execute(action)
    nextKey = makeKey(nextState, opponent(player))
    reward = reinforcement(nextState)
    if nextKey not in table then
        addKey(nextKey)
    end
    updateQvalues(stateKey, action, nextKey, reward)
    if game over then
        reset
        state = startState
        game += 1
    else
        state = nextState
        switchPlayers()
    end
end
```

3) Pseudocode for Updating Session: details are shown as below [5]:

Algorithm 4: Pseudocode for Updating Session

```
function updateQValues (stateKey, action, nextKey,
reward)
if game over then
    expected = reward
else
    if player is X then
        expected = reward + (discount *
        lowestQvalue(nextKey))
    else
        expected = reward + (discount *
        highestQvalue(nextKey))
    end
end
change = learningRate * (expected -
table[stateKey][action])
table[stateKey][action] += change
```

B. ϵ- Greedy Policy

In practice, one can ensure that every (reachable) state is visited *infinitely often* by using an ϵ-greedy policy.

$$\text{with probability } \epsilon : \text{choose an action at random}$$
$$\text{with probability } 1-\epsilon : a = \arg\max_a Q(s,a)$$

Practically, I choose $\epsilon = 0.15$ to train the Q-agent because my training iteration is chosen as 50,000,000, which is very close to $3^{16} = 43,046,721$. If I was able to train the Q-agent 10 more times than 3^{16}, I would like choose $\epsilon = 0.1$ as normal.

C. Data Structure

It is very natural for me to think about using the data structure called *HashMap*, which is a *Map* based collection class that is used for storing Key and value pairs. The reason that Python suits for this project is the built-in ***dictionary*** in Python is exactly using the same mechanism as HashMap. I can easily represent the Q-values table as a dictionary keyed on states, where the state is a string representing a combination of the current player and the current board. The value associated with each state should be another dictionary keyed on actions. There are nine possible locations to play on a tic-tac-toe board which will be numbered 0-15. The value associated with each action is the current prediction of the expected value of taking that action in the current state.

D. Interface

I used the module called ***tkinter***, which is maybe the easist one to implement camparing to other modules like ***GTK***.

VI. Results

A video can be found at the following link: https://youtu.be/vwLyn7iPNuA. The console shows the Q-value, and the Q-agent chooses its action based on all the Q-value table. After 50,000,000 training iterations, I am not able to beat the Q-agent as a human being. As the video shows, the Q-agent knows how to defend me and the rules to win the game. In the video, I created 3 scenarios to test whether the Q-agent knows how to win the game: concatenating in a horizontal line, a vertical one and a diagonal one, respectively.

VII. Conclusion

We were able to successfully play the game Tic-Tac-Toe by learning straight from the score, achieving super-human results. However, training was not consistent in that more training did not necessarily correlate with improving Q-value. Future work could attempt to use neural network to find the feature function ϕ (shown in Algorithm 1) to approximate Q-value table. Overall, our results show that reinforcement learning is a step in the right direction and has a lot of potential for further applications.

Acknowledgment

The author would like to thank Dr. X. Zhong, Dr. T. Yang, Lisha Yao, Wen Du, Veena Chidurala and Randy Burrow from the Information Technology Service Department for help.

REFERENCES

[1] K. Chen, "Deep reinforcement learning for flappy bird."

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

[3] K. Zhang, Z. Yang, H. Liu, T. Zhang, and T. Başar, "Fully decentralized multi-agent reinforcement learning with networked agents," *arXiv preprint arXiv:1802.08757*, 2018.

[4] A. Y. Ng, "Shaping and policy search in reinforcement learning," Ph.D. dissertation, University of California, Berkeley, 2003.

[5] L. Meeden, "Cs63 lab 6: Learning to play tic-tac-toe," 2011.

[6] K. Arulkumaran, M. P. Deisenroth, M. Brundage, and A. A. Bharath, "A brief survey of deep reinforcement learning," *arXiv preprint arXiv:1708.05866*, 2017.

[7] C. J. Watkins and P. Dayan, "Q-learning," *Machine learning*, vol. 8, no. 3-4, pp. 279–292, 1992.

[8] G. A. Rummery and M. Niranjan, *On-line Q-learning using connectionist systems*. University of Cambridge, Department of Engineering, 1994, vol. 37.

[9] J. Koutník, G. Cuccu, J. Schmidhuber, and F. Gomez, "Evolving large-scale neural networks for vision-based reinforcement learning," in *Proceedings of the 15th annual conference on Genetic and evolutionary computation*. ACM, 2013, pp. 1061–1068.

Shengjun(Daniel) Zhang received the B.S. degree in *automation of honors program* from China Agricultural University, Beijing, China, in 2014, and the M.S. degree in electrical engineering from New York University, New York, NY, in 2017.

Since 2018, he has been at the Cyber-Physical Energy Systems Lab as a Ph.D student, University of North Texas, Denton, TX, USA. His research interests are in control theory, machine learning, deep reinforcement learning, distributed optimization, and connected autonomous vehicles. Personal Website.