An Investigation into Structured Environments

Özlem Çakmak
Technische Universität München
Munich, Germany
Email: c.oezlem@mytum.de

Dominik Meyer
Technische Universität München
Munich, Germany
Email: meyerd@mytum.de

***Abstract*—Even simple Reinforcement Learning problems can have large state spaces. Dealing with those is challenging and can lead to high computational costs. A way to reduce large state spaces is exploiting symmetries in the state set. In this paper we focus on temporal-difference learning – especially on the Sarsa algorithm – and study the occurrence of symmetries in Reinforcement Learning problems by using the example of the board game tic-tac-toe. We study different methods to detect and exploit those symmetries to obtain a more efficient learning process.**

I. Introduction

Reinforcement Learning (RL) is a machine learning method which is particularly suitable for learning problems without a complete knowledge of the environment. The basic elements of RL are the learning agent and an uncertain environment. The agent learns through experience from interaction with the environment to achieve a given goal by receiving positive or negative rewards from this environment depending on the performed actions.

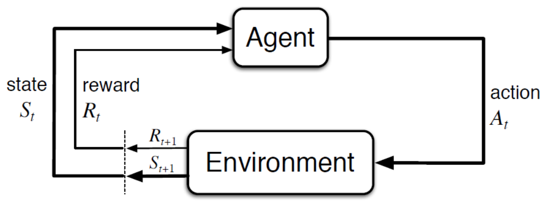

Fig. 1. Interaction between agent and environment [1]. Starting from a state S_t with the corresponding reward R_t, the agent takes an action A_t. In return, the environment responds with the successor state S_{t+1} and reward R_{t+1}.

The learning environment is considered a Markov Decision Process (MDP, [1]) which is defined by the quintuple $(\mathcal{S}, \mathcal{A}, \mathcal{P}, \mathcal{R}, \gamma)$. It consists of a set of states $\mathcal{S}$, a set of actions $\mathcal{A}$, the transition probability $\mathcal{P}$ for transitioning from the current state s to the successor state s' by taking an action $a \in \mathcal{A}$. The reward r along this transition is provided by the reward function $\mathcal{R}$. The last element $\gamma \in [0, 1]$ is a discount factor which discounts future rewards. If the state and action set are both finite, it is called a finite MDP. S^+ is the set of all states plus the terminal states which end the current interaction between agent and environment. These segments of agent-environment interactions are called episodes. After reaching a terminal state the agent is reset to a starting state s_0 and a new episode begins. RL tasks consisting of subsequent episodes are called episodic tasks. In continuous RL tasks the agent environment interaction goes on continuously without reaching a terminal state.

Besides the reward function there are three other main subelements of RL: a policy, a value function and an optional model of the environment [1]. An optimal policy maps to each state s an optimal action a and such controls the behavior of the learning agent. While rewards can be seen as pleasure and pain and indicate the immediate outcome of an action, the value function assigns to each state a value indicating which behavior is good in the long run. The value v_π of a state s is defined by the return G_t – which is the sum of discounted future rewards – that can be expected starting from this specific state s and following the policy π

$$\begin{aligned} v_\pi(s) &:= \mathbb{E}_\pi[G_t | S_t = s] \\ &= \mathbb{E}_\pi\left[\sum_{k=0}^{\infty} \gamma^k R_{t+k+1} \middle| S_t = s\right]. \end{aligned} \tag{1}$$

A model of the environment is not necessarily required but if available it can include information like predictions of the successor state or reward.

There are three algorithm types for solving a RL problem: dynamic programming (DP), Monte Carlo (MC) methods and temporal-difference (TD) learning. This paper focuses on TD methods.

II. Temporal-Difference Learning

TD methods have some major advantages over DP and MC methods: In comparison to DP methods there is no need for a model, rewards or transition probabilities [1]. Thus TD methods can deal with uncertain environments and are further able to adapt to changes in the environment. The advantage over MC methods is that TD methods are fully incremental, meaning that every step contributes to the learning process and there is no need to wait until an episode has ended. Therefore, TD methods can also be applied to continuing tasks. Also TD methods bootstrap – i.e. new estimates are based on previous estimates – and they are proven to converge under certain constraints.

The update rule for the simplest TD algorithm, TD(0), is denoted as

$$V(S_t) \leftarrow V(S_t) + \alpha[R_{t+1} + \gamma V(S_{t+1}) - V(S_t)]. \tag{2}$$

In this method, the state values are stored in a table. One episode of TD(0) looks as follows: First, all state values are initialized with an arbitrary default value[1] and the first state s of an episode is determined. Then an action a is selected according to the policy π. After performing that action the reward r and the successor state s' can be observed. With this information the value of the current state s is updated according to the TD(0) update rule and s' is set as the new current state s. This is repeated until a terminal state is reached which ends the current episode.

Sarsa algorithm: The Sarsa algorithm uses an action-value function $q_\pi(s,a)$ instead of the state-value function (1) and indicates how valuable it is to take an action a in a state s. So it maps a value to an state-action pair instead of just mapping a value to a state. The advantage of Sarsa over other TD algorithms is that transition probabilities are not necessary. Also the consideration of state-action pairs instead of just states is more convenient in terms of symmetry reduction which will be discussed in the next chapter.

1) Sample-average method: The sample-average method estimates the true value of an action a by averaging over all received rewards $R_1, R_2, \dots, R_{K_a}$ after taking that action K_a times. The estimated value $Q_t(a)$ of action a at time step t is then

$$Q_t(a) = \frac{R_1 + R_2 + \cdots + R_{K_a}}{K_a}. \tag{3}$$

A default value is defined for $K_a = 0$, e.g. zero and for $K_a \to \infty$ the estimated action-value $Q_t(a)$ converges to the true action-value $q_*(a)$.

2) Incremental updates: One drawback of the sample-average method are the increasing memory and computational requirements as t increases. To derive an incremental update formula we consider Q_k as the action-value estimate of an action a for its kth reward R_k which is the average of its first $k-1$ rewards. The average of all k rewards is then

$$\begin{aligned} Q_{k+1} &= \frac{1}{k}\sum_{i=1}^{k} R_i \\ &= \frac{1}{k}\left(R_k + \sum_{i=1}^{k-1} R_i\right) \\ &= \frac{1}{k}\left(R_k + (k-1)Q_k\right) \\ &= Q_k + \frac{1}{k}\left[R_k - Q_k\right]. \end{aligned} \tag{4}$$

The update rule (4) in its general form

$$\begin{aligned} &NewEstimate \leftarrow OldEstimate \\ &\qquad + StepSize[Target - OldEstimate] \end{aligned} \tag{5}$$

is often used in RL algorithms. By moving toward the target the estimation error $[Target - OldEstimate]$ is reduced. The step-size parameter $SteStepSize$ can either vary with time or be set to a constant value in order to adapt to changes in the environment which is necessary in non-stationary problems.

The corresponding computation of (1) for action-values is denoted as

$$\begin{aligned} q_\pi(s,a) :&= \mathbb{E}_\pi[G_t | S_t = s, A_t = a] \\ &= \mathbb{E}_\pi\left[\sum_{k=0}^{\infty} \gamma^k R_{t+k+1} \middle| S_t = s, A_t = a\right] \\ &= \mathbb{E}_\pi\left[R_{t+1} + \gamma \sum_{k=0}^{\infty} \gamma^k R_{t+k+2} \middle| S_t = s, A_t = a\right] \\ &= \mathbb{E}_\pi\left[R_{t+1} + \gamma q_\pi(S_{t+1}, A_{t+1}) | S_t = s, A_t = a\right] \end{aligned} \tag{6}$$

The Sarsa algorithm uses (6) as a target. As a bootstrapping method it substitutes the true action-value q_π – i.e. the mean reward that is received after taking that action – by its current estimate. From this and (4) follows the update rule for the Sarsa algorithm

$$\begin{aligned} Q(S_t, A_t) \leftarrow & Q(S_t, A_t) \\ & + \alpha[R_{t+1} + \gamma Q(S_{t+1}, A_{t+1}) - Q(S_t, A_t)]. \end{aligned} \tag{7}$$

III. Symmetries in RL

One of the problems in RL tasks are large state spaces, since the storage and time complexity grows with the number of states. As a result of this problem either the learning process takes too long or the states can not be visited often enough to obtain a reliable approximation of the state or state-action values.

An approach to solve this problem is to take account of symmetries in RL algorithms. By doing that the primary MDP can be mapped to a reduced MDP with a smaller state and action set [2].

Symmetries in RL problems can appear in different forms which is shown in the following figure:

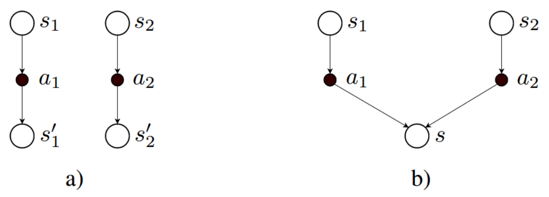

Fig. 2. a) State symmetry b) State-action symmetry

The big circles in the graphs depict the states, the dots the actions.

The first graph shows the case of state symmetry. s_1 and s_2 are symmetric if the transition probability $p(s'_1, r_1 | s_1, a_1)$ for the transition from s_1 to s'_1 by taking the action a_1 and receiving the reward r_1 is the same as $p(s'_2, r_2 | s_2, a_2)$.

The second graph shows the case of symmetric state-action pairs: Here the states s_1 and s_2 which are not symmetric

[1]optimistic initial values

according to the first case can lead to an equivalent successor state s'.

A concrete example for symmetries in RL is provided by board games. In this case symmetries occur as reflection symmetries or rotations of the board, the interchangeability of the players or as different state-action pairs [3]. The following figure shows how the symmetries depicted in Fig. 2 occur in tic-tac-toe.

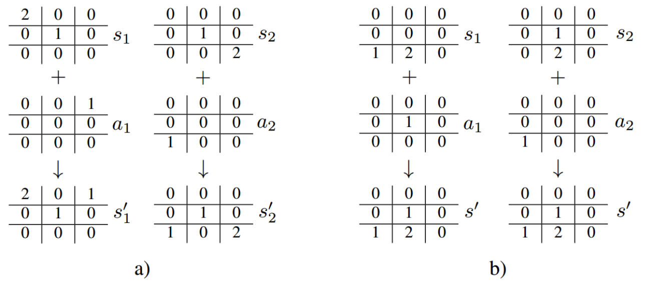

Fig. 3. a) State symmetry b) State-action symmetry

In Fig. 3 a) s_2, a_2 and s'_2 can be obtained by rotating s_1, a_1 and s'_1, respectively. If the initial states and actions in Fig. 3 b) are compared separately, one state or action can not be obtained by rotating or reflecting the other. However, the state-action pairs $(s_1|a_1)$ and $(s_2|a_2)$ lead to a single successor state s' and thus are symmetric in terms of state-action symmetry. So a greater number of symmetries can be exploited which is why we chose the Sarsa algorithm. One possibility to exploit symmetries in RL algorithms is the set-states approach.

IV. Set-States

The set-states approach defines a new, smaller state set C with the states c which represent the different symmetries occurring in S. For this purpose each state s of S shall be assigned to a single state c of C. The challenge herein is to perform the classification of the states automatically, hence even small problems have large state sets and therefore, the manual classification would get very tedious.

If a model is available one could detect equivalent states by comparing their transition probabilities within a tolerance of ϵ. For the usual case without an existing model the transition probabilities could be approximated from several runs of the algorithm. We investigate another approach which is to execution of the Sarsa algorithm until the Q-values converge and assign state-action pairs with same Q-values within a certain tolerance level to the representative states c.

V. Experimental Results

To test the set-states approach, we chose the familiar two-player board game tic-tac-toe. The player that first manages to place three of his respective marks in a horizontal, vertical or diagonal line wins.

This game is a deterministic problem, so performing an action in a specific state always leads to the same successor state. Therefore, all transition probabilities equal to one and are not distinctive for the different states. For this reason we implemented the Sarsa algorithm and used Q-Values for mapping the states s to their representative states c.

The Sarsa player was trained as follows: For faster learning we used Sarsa(λ). The parameter λ indicates the use of eligibility traces. They track the lastly used state-action pairs, so that not only the last state-action pair is updated when a reward is obtained, but all state-action pairs that led to this reward. The most recent step gets the full update, the previous step gets a percentage of λ^2 of the full update and so on.

A simple random player (player 2) was chosen as an opponent which randomly places its mark on a free spot following the discrete uniform distribution.

Preceding tests showed that an optimistic initial value of 1 leads to faster learning and also that 100 000 episodes are sufficient for this problem.

The set of states exists of all states of the game board which can appear to the Sarsa player (player 1). In these states the number of marks by player 2 is either equal or one more than the number of marks by player 1.

The set of states exists of all game boards in which it is the turn of the Sarsa player (player 1). I.e. in each state the number of marks by player 2 is either equal or one more than the number of marks by player 1. The action set consists of up to 9 actions depending on the number of free spots of a particular state. The Q-values are stored in a table (Q-table). For a simple way to index the Q-values, 0 is used as a mark for free spots and 1 and 2 for player 1 and 2, respectively. By vectorizing the game board a number with base 3 is obtained which easily can be converted to a decimal number and used to index the states.

The following table contains all parameters for the test setup:

Parameter	Training Value	Testing Value
Number of tasks	10	10
Number of episodes	100 000	1 000
Initial Q-values	1	1
Reward	± 1	± 1
ϵ	0.1	0
α	0.5	0
λ	0.7	0
γ	1	0

TABLE I
Parameters for the training and testing phases.

Training: In the first run the Sarsa player is trained with the parameters given above. After that, we have the mean Q-values of ten tasks. This mean values are used to detect the symmetric states. For this we compare the Q-values of one state s_k to the Q-values of all other states s_i. If the relative error

$$e = \frac{Q(s_k) - Q(s_i)}{Q(s_k)}$$

of each action is smaller than one, the mean squared error is calculated and saved. Now a threshold can be set for the MSE to consider state s_k as the set-state of s_i. The result is a lookup table for every state. If there is an entry for a given state, the Q-values of the representative state is used and updated instead. It follows a second run with the same parameters besides that the lookup table is used before every update step.

Testing: In both training runs the Q-tables are stored after one third, two thirds and the total number of episodes. Letting the Sarsa player run again with these Q-tables and excluding the learning process during these runs, the following results are obtained.

Q-table after t episodes	**Score after 1st run** (Won - Lost - Draw)	**Score after 2nd run** (Won - Lost - Draw)
	EXPERIMENT 1	
3334	945 - 3 - 52	948 - 3 - 49
6668	930 - 4 - 66	929 - 9 - 62
10 000	938 - 2 - 60	931 - 9 - 60
	EXPERIMENT 2	
3334	940 - 3 - 57	941 - 3 - 56
6668	944 - 4 - 52	936 - 8 - 56
10 000	950 - 6 - 44	952 - 4 - 44

TABLE II
RESULTS OF THE EXPERIMENTS

In the training phase of experiment 1, α and ϵ are constant. For experiment 2, α decreases with the number of visits k of $(s|a)$ and ϵ with the number of episodes t, i.e. $\alpha = 10/(k + 19)$, $\epsilon = 4\,000/(t + 39\,999)$. Table II shows that the difference between the results with and without the exploitation of symmetries is not significant. This can also be seen in Fig. 4.

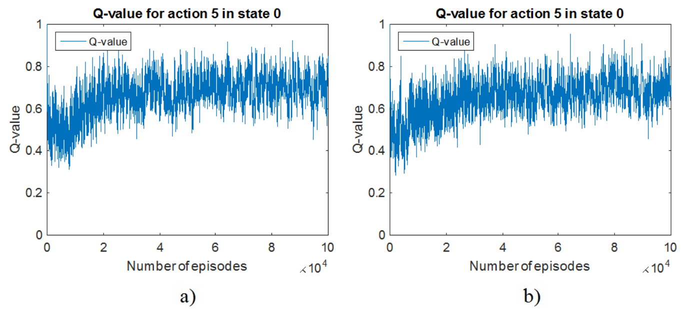

Fig. 4. Courses of $Q(s = 0, a = 5)$ after the training phases of experiment 1. The case without symmetry exploitation is depicted in a) and with symmetry exploitation in b)

However, the overall number of wins in experiment 2 is higher than in experiment 1. But this is highly dependent on the decreasing parameters chosen for α and ϵ and the threshold set for the relative error during the detection of symmetric states. This threshold determines which states are assigned to a set-state and thus the size of the reduced set of states. A higher threshold leads to a greater reduction of the reduced set of states but also to incorrect assignments of states to set-states. The incorrectly assigned states cause a wrong update of the set-state and this again leads to wrong optimal actions for correctly assigned states. Eventually, the results are worse than without the consideration of symmetries. In Table III a small threshold of 0.08 was used. Here, the majority of the assignments to set-states was correct but many states remained unassigned.

	Experiment 1	Experiment 2
size of original set of states	4520	4520
size of reduced set of states	3417	3416
actual number of set-states	337	337
correctly classified set-states	316	309

TABLE III
RESULTS OF THE EXPERIMENTS

VI. CONCLUSION

The experiments showed that the exploitation of symmetries is strongly dependent on the correct detection of symmetric states. Also that using the mean squared error of Q-values as a similarity measure is not suited for this task. Therefore, further experiments with other similarity measures are necessary. Also the correct adjustment of the decreasing parameters of α and ϵ have a huge impact on the results and should be taken into account.

REFERENCES

[1] R. S. Sutton and A. G. Barto, *Reinforcement Learning: An Introduction*, 1st ed. Cambridge, MA, USA: MIT Press, 1998.
[2] R. Fitch et al., *Structural abstraction experiments in reinforcement learning*, AI 2005: Advances in Artificial Intelligence. Springer, 2005.
[3] S. Schiffel, *Symmetry Detection in General Game Playing*, Association for the Advancement of Artificial Intelligence. 2010.