Temporal-Difference Learning Solution to Gridworld Using Sarsa Algorithm

COSC 4368

Alexander Hebert

Introduction

Reinforcement learning (RL) involves an agent learning through experience, trial and error, and success and failure by interacting with its environment. Success results in positive reinforcement and failure results in negative reinforcement. Reinforcement is quantified with a reward function, *R(s)*, which affects the values of states and actions. States and actions are described mathematically by Markov Decision Processes (MDP). In order to find optimal solutions, there must be a trade-off between exploration and exploitation. The nature of the trade-off depends on whether the environment is stationary or non-stationary. Three common approaches for implementing RL are dynamic programming, Monte Carlo methods, and temporal-difference (TD) learning. [1] The first and second methods will be summarized briefly. TD will be explained more thoroughly and followed by a gridworld problem using the Sarsa algorithm.

RL is formulated using policies Π(s), utility functions U(s), reward function R(s), and sometimes a model of the environment. Utility functions as in [2] are also called state-value functions V(s) and action-value functions Q(s,a) as in [1]. Policies define the relation between possible actions in each state and which action is selected. In passive RL, the policy is fixed as opposed to active RL where the policy changes with experience in the environment. Utility functions specify value of states or state-action pairs relative to long term success in reaching a goal. A reward function specifies whether states or state-action pairs are beneficial (positive), harmful (negative), or neutral; goal states are terminal states with maximum or high reward. Models approximate an environment to some extent and are used in planning actions [1].

Utility functions are governed by Bellman equations, which follow.
For a fixed policy [2]:

$$U^{\Pi}(s) = R(s) + \gamma \sum_{s'} P(s'|s,\mathrm{a})\ U^{\Pi}(s')$$

where *a* is an action; *s'* is the state after *s*; γ is the discount rate with value $0 \leq \gamma \leq 1$; and $P(s'|s,\mathrm{a})$ is the transition model probability. As γ tends to zero, an agent focuses on short-term rewards, and as γ tends to one, an agent focuses on long-term rewards [1].

For an optimal policy [2]:

$$U\ (s) = R(s) + \gamma \max_a \sum_{s\prime} P(s'|s, \mathrm{a})\ U(s')$$

with $U\ (s)$ being the optimal utility function (* denotes optimal).

Solving the Bellman equations directly is sometimes possible but rarely practical or efficient; it is analogous to an exhaustive search (brute force). The main reasons include lack of accurate model of the environment, computation time and memory, and satisfaction of the Markov property [1]. Therefore approximate solution methods for the Bellman equations are used in practice.

One method is dynamic programming (DP), which applies a group of algorithms to find optimal solutions. The main algorithms in DP are policy iteration and value iteration. DP requires a complete model of the environment as a MDP. DP's disadvantages are necessity of a model and computational expense in time and memory. [1] For problems with large state spaces, DP is not practical because it must computes passes through the entire space (e.g. game of backgammon has $\approx 10^{50}$ states [2]). Approximate DP methods such as asynchronous DP in [1] or approximate adaptive DP in [2] can improve computational efficiency by several orders of magnitude [2].

Another approach is Monte Carlo methods (MCM). MCM learn through simulated experience by "sampling sequences of states, actions, and rewards from interaction with an environment" ("averaging sample returns") [1]. MCM requires a model with sample transitions but not complete probability distributions as in DP. MCM differs from TD learning in that policies and utilities are updated only after entire episodes/trials instead of step-by-step increments in TD. [1] MCM can achieve optimal performance, but they do not learn in real-time like TD learning.

The third approach is temporal-difference learning. TD learning relies on the differences in utilities between the current and next states or state-action pairs, hence, the name temporal-difference. The TD equation follows:

$$U^{\Pi}(s) \leftarrow U^{\Pi}(s) + \alpha[R(s) + \gamma U^{\Pi}(s') \quad U^{\Pi}(s)]$$

where α is the learning rate with value $0 \leq \alpha \leq 1$. A constant value of α will cause the algorithm to fail to converge because it does not satisfy two conditions from stochastic approximation theory [1], which follow:

$$\sum_{k=1}^{\infty} \alpha_k(a) = \infty \quad and \quad \sum_{k=1}^{\infty} \alpha_k^2(a) < \infty$$

The learning rate can satisfy those conditions if it decays adequately. A simple case that will converge is

$$\alpha_k(a) = \frac{1}{k}$$

where k is the number of past rewards. If a table of number of visits to each state and action is kept (i.e. $N(s, a)$), those values can be used to decay α more selectively. The next equation shows such a function for α.

$$\alpha(s, a) = \frac{1}{\beta + N(s, a)}$$

where β is a constant (e.g. β = 10). For stationary problems, convergence of α is preferred since the algorithm will be stable and ideally, optimal. For non-stationary problems (more common), an agent may learn optimal behavior initially but fail to adapt as conditions change in the environment. As α decays to zero, learning slows and effectively stops. A constant α (e.g. α = 0.1) is better suited for non-stationary environments where the rewards and hazards are changing with time. [1]

The two main types of TD learning algorithms are Q-learning and Sarsa. Q-learning is an off-policy algorithm, and Sarsa is on-policy. Both algorithms are forms of active RL, do not require any model of the environment, and use the action-value function Q(s,a) as a utility function. In Q-learning, Q(s,a) approximates the optimal action-value function Q^*(s,a) [1]. The Q-learning update equation is

$$Q(s, a) \leftarrow Q(s, a) + \alpha \left[R(s) + \gamma \max_{a'} Q(s', a') \quad Q(s, a) \right]$$

The Sarsa name comes from the quintuple (*s, a, r', s', a'*). The Sarsa update equation is

$$Q(s,a) \leftarrow Q(s,a) + \alpha[R(s) + \gamma Q(s',a') \quad Q(s,a)]$$

The Sarsa algorithm will converge to $Q^*(s,a)$ if "all state-action pairs are visited an infinite number of times" [1]. Both algorithms will achieve an optimal policy if they are Greedy in the Limit of Infinite Exploration (GLIE) [1,2].

Exploration is necessary to discover better solutions, which ideally lead to optimal ones. Exploration can be purely random, based on experience in an environment, or a combination. A simple method is ε-greedy where the best state or state-action pair is always selected (greedy); except with small probability ε, another choice is selected randomly with uniform distribution. The probability ε can be constant (e.g. ε = 0.1) or a function of N(s,a) so that ε decreases with the number of visits and approaches zero (i.e. GLIE). Another method is the exploration function *f(u,n)*. That function must be increasing in *u* and decreasing in *n*, which achieves the trade-off. The exploration function is a more intelligent approach because it encourages an agent to visit unexplored regions rather than just random choices.

Results and Discussion

TD learning is applied to a simple gridworld problem using the Sarsa algorithm. The 8x10 gridworld comes from Example 7.4 in [1]. Attempts were made to implement eligibility traces for the Sarsa(λ) algorithm. However, the results were erratic with lower performance than the standard algorithm. The gridworld has 80 valid states with four possible actions per state (up, down, left, right). If an action is selected that moves the agent off a boundary, then there is a reward of -1. The reward is zero unless the agent reaches the goal state with a reward of 10, which terminates an episode/trial. The goal state (red X) is always located at (7, 9) where the numbers correspond to (row #, column #) like matrix indices. The starting location (green circle) is (1,1) for the first experiment and a random location (except the goal state) for the second experiment. The basic gridworld is shown in Figure 1; disregard the minus signs on all vertical axes. All programs were coded in Matlab.

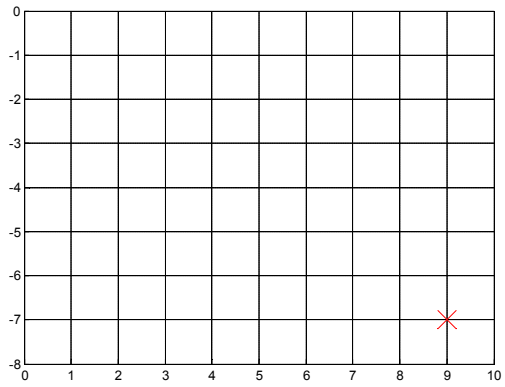

Figure 1. Blank 8x10 gridworld with goal state at (7,9).

The Sarsa algorithm is implemented using the Sarsa update equation with $\gamma = 1$. The learning rate α and probability ε are both functions of N(s):

$$\alpha(s) = \frac{1}{10 + N(s)} \quad and \quad \varepsilon(s) = \frac{1}{10 + N(s)}$$

For simplicity, an ε-greedy policy is used to achieve exploration, but it converges to a greedy policy in the limit as the probability decreases.

Experiment 1:

The starting location (green circle) is fixed at (1,1). Following the ε-greedy policy, the Sarsa algorithm learns the path to the goal state at (7,9). An example of an initial episode is shown in Figure 2.

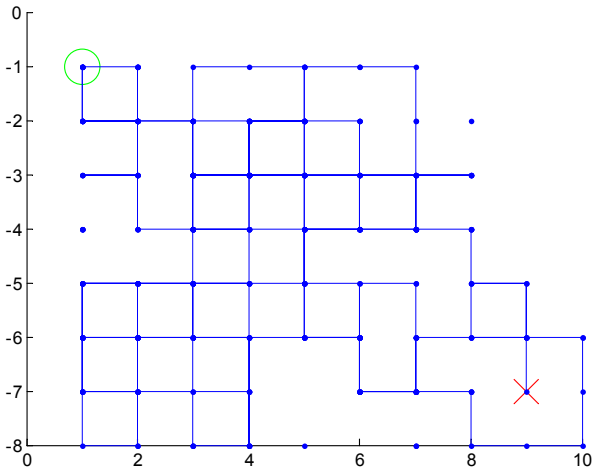

Figure 2. Initial episode for fixed start at (1,1).

The initial episode can vary greatly in the number of steps to reach the goal. In Figure 1, there are hundreds of steps before finding the goal. After running the program for 150 episodes, the Sarsa algorithm has learned the optimal or near optimal path as shown in Figure 3. The agent reaches the goal after only 14 steps.

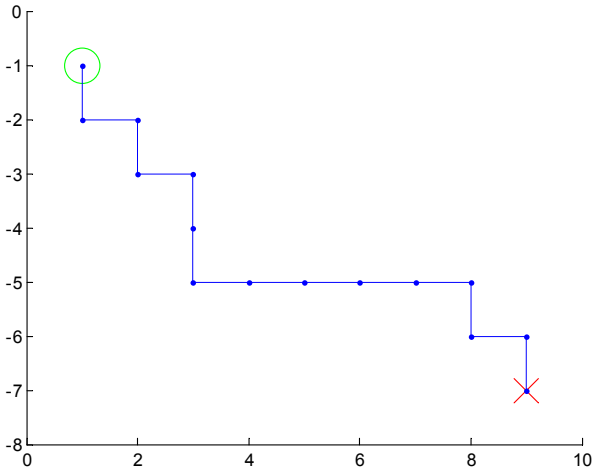

Figure 3. Episode #150 for fixed start at (1,1).

It is of interest to know how many episodes were needed to achieve near optimal performance. Therefore, the number of steps to reach the goal state is plotted versus the episode number in Figure 4. Sarsa achieved near optimal performance after less than 10 episodes. Unfortunately, the number of steps continues to oscillate instead of converging.

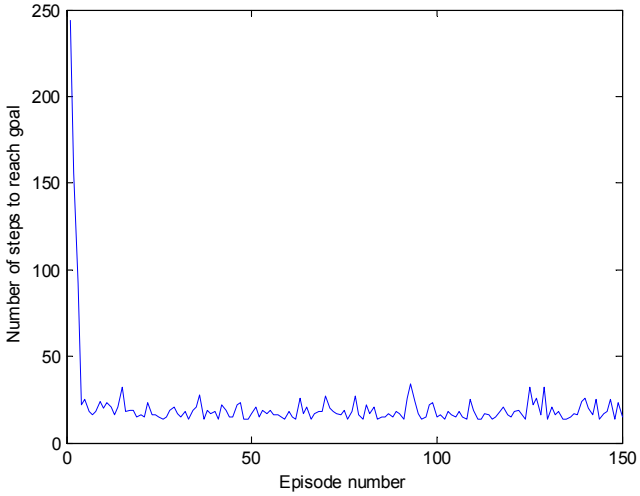

Figure 4. Number of steps to reach goal vs. episode number for fixed start at (1,1).

The Q-values for the action-value function Q(s,a) are plotted as a colored contour in Figure 5. The Q-values increase from blue towards the goal region in red. The near optimal path from the starting location at (1,1) is evident in the shape of the contours.

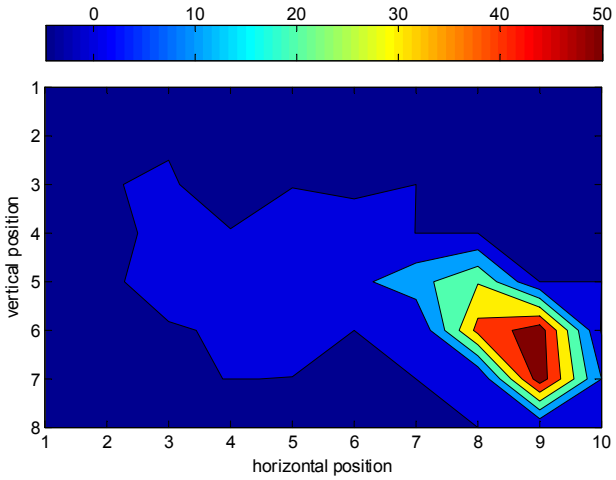

Figure 5. Colored contour plot of Q-values for Q(s,a) with fixed starting location at (1,1).

Experiment 2:

The starting location is randomly generated, but it cannot be the goal state. Following the ε-greedy policy, the Sarsa algorithm learns the path to the goal state at (7,9). An example of an initial episode is shown in Figure 6, which is very similar to the fixed start in Figure 2.

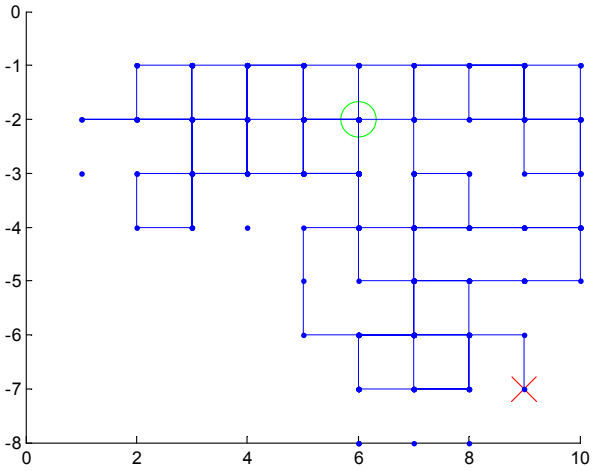

Figure 6. Initial episode for random starting location.

Again, the number of steps to reach the goal state is plotted versus the episode number in Figure 7. Sarsa achieved near optimal performance after about 100 episodes. As for the fixed start, the

number of steps continues to oscillate instead of converging. Also, the maximum number of steps is limited to 500 when the program will stop without reaching the goal.

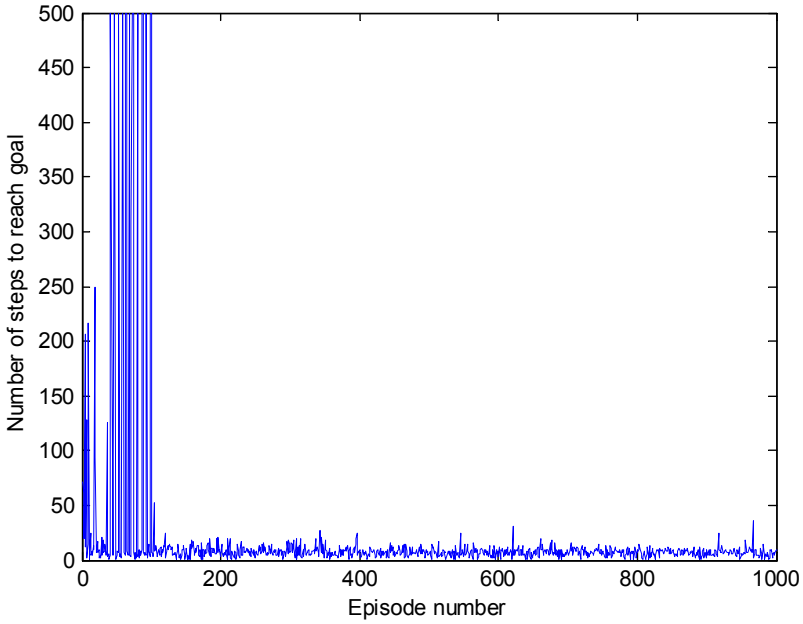

Figure 7. Number of steps to reach goal vs. episode number for random starting location.

The Q-values for the action-value function Q(s,a) are plotted as a colored contour in Figure 8. The Q-values increase from blue towards the goal region in red. Starting locations are randomly generated so the contour shapes are more general than for the fixed start.

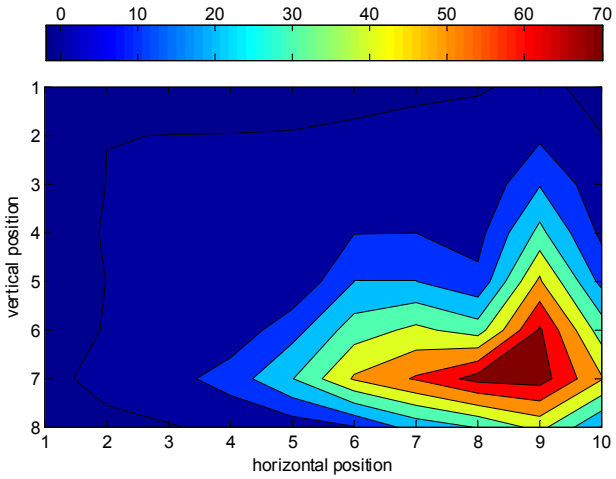

Figure 8. Colored contour plot of Q-values for Q(s,a) with random starting location.

Conclusion

Reinforcement learning was briefly surveyed, and a Matlab program implementing a TD learning algorithm was developed. The main aspects of RL were discussed, including Bellman equations, trade-off between exploration and exploitation, and three solution methods. Temporal-difference learning is a simple (model free) but effective form of RL. The Sarsa algorithm was applied to a gridworld problem and achieved near optimal results. Eligibility traces were attempted for Sarsa(λ) but results were erratic and did not approach optimal performance, which is likely due to incorrect implementation. Performance for standard Sarsa in the fixed start and random starting location experiments was similar. In both experiments, number of steps to the goal state approached optimal but continued to oscillate moderately. An ε-greedy policy was used with N(s) in order to converge to greedy in the limit so there is probably an implementation flaw in the Matlab program, causing oscillations. The learning rate was also a function of N(s) since without it, the Q-values increased without bound. The gridworld problem is stationary so constant α and ε will not yield optimal results. If there are time-varying hazards and goal states (non-stationary environment), an algorithm with constant α and ε will likely outperform a GLIE counterpart. Two interesting quotes I read while researching RL are "optimism in the face of uncertainty" [3] in regard to exploration and "failure is the shortest path to success" [5] in regard to learning from mistakes.

References

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

[2] S. Russel and P. Norvig, "Chp. 22: Reinforcement Learning," *Artificial Intelligence: A Modern Approach*, 3rd ed., Pearson, 2009.

[3] P. Abbeel, "Lectures 10 and 11: Reinforcement Learning," UC Berkeley CS188, YouTube (CS188Spring2013), 2013 [https://www.youtube.com/watch?v=ifma8G7LegE].

[4] J. McClelland, "Chp. 9 Temporal Difference Learning," Stanford, 2013 [http://www.stanford.edu/group/pdplab/pdphandbook/handbookch10.html].

[5] J.S.R. Jang, C.T. Sun, and E. Mizutani, "Chp 10: Learning From Reinforcement," *Neuro-Fuzzy and Soft Computing*, Prentice-Hall, 1997.