Approaches to Solving Infinite Q-Learning Oscillations in a Grid-world

Ethan Joachim Eldridge

May 1, 2012

Abstract

This paper presents a brief overview of Temporal Difference learning, specifically the subset known as Q-Learning. Q-Learning is experimentally tested on a simple grid-world environment on a path following experiment. Initial results with an ϵ-greedy policy result in poor perform on complex environments. During the course of experimentation a problem involving the greedy mechanism of the policy cause circular state-action pairings between adjacent spaces of the grid-world. New methods are hypothesized to solve the dilemma of infinite oscillatory actions, and the resuts show a drastic improvement in performance.

1 Introduction

In machine learning reinforcement learning is used when a problem has ill-defined mappings from a given input to perfectly correct output. Techniques like supervised learning cannot be used in tasks where trial and error is the greatest teacher. However, because of the action to state representations of reinforcement learning, trial and error fits perfectly with the methods commonly used.

An *agent* is defined as something that can sense the state of its current environment as well as take actions that change its current state to a new one. In addition to states and actions, the agent always has a goal to attain. However, once this goal is attained the challenge is to award the agent's actions accordingly. This is commonly known as the *credit assignment problem* as coined in [Sut88a]. Commonly used methods for prediction such as Markov chains and backpropagation networks tend to have large problems with credit assignment.

In the case of Markov chains: if a state B leads to termination and has a value of $^3/_4$ and a state A leads to B 100% of the time; then a reasonable choice for the value of A is also $^3/_4$. However, a batch updating Monte Carlo method will assign no credit to A because the movement A$\rightarrow$B gives no return, it is only the B state that leads to termination. A Temporal Difference (TD) method will find that since A leads to B and B leads to termination, A must also have a value of $^3/_4$ because it led to B every time[1].

Backpropagation networks learn to train themselves by pairing an output with a set of inputs. The weights and values of neurons is normally thought of as a black box. However, if one thinks about the update rule, the value of an output (correct or not) is propagated backwards equally throughout the network, while the value of the output is perfect[2], the value is propagated to *all* the neurons, and not just the ones causing the error. TD methods, because of their incremental nature are able to assign value to actions and states that actually caused any errors.

$$V(s_t) \leftarrow V(s_t) + \alpha \left(r_t + \gamma V(s_{t+1}) - V(s_t) \right) \quad (1)$$

Temporal Difference learning uses an incremental updating rule that follows the general rule of Equation 1.[SB98] Where the value of a state s_t is estimated by $V(s_t)$ given the observed reward r_{t+1} and the estimate of the next state $V(s_{t+1})$. This means that the algorithm can learn by taking a single step and updating its previous estimates. By using

[1] Example numbers taken from [SB98]

[2] Outputs in supervised learning are always the correct answer to the given input because backpropagation is a supervised learning technique.

a greedy approach to which actions to take, the algorithm can build up a value estimate of each of the possible states it can be in. The parameters α and γ change the behavior of the algorithm. α is a stepwise parameter that varies from one step to the next. The i^{th} reward for the action a is defined to be $^1/_i$. γ is a discounting parameter that affects how much we trust our next prediction. Setting $\alpha = 1$ causes all timesteps to be considered equal and $\alpha = 0$ causes no learning at all, while $\gamma = 1$ causes a greedy approach to estimation.

There are generally two types of policies in TD-learning: on or off. An on-policy is updated from the results of actual actions and behaviors, while off-policies have the capability to use hypotheticals to test possible actions. Stated plainly, an online policy learns from experience while an offline policy learns from hypotheticals. That being said, an off-line policy does not sit around and do nothing, rather it's estimates are updated using hypotheticals from step to step and then an action is taken based on the most recent estimates.

Temporal difference is based on the idea of using the future predictions as an estimate of the actual optimal value. Despite the oddity of using an ongoing estimate of the optimal value as a replacement for the value itself, TD-Learning has been proven to converge given the correct parameter settings in [Sut88b].

2 Q-Learning

Q-Learning is an off-policy TD method developed by Chris Watkins in [WD92]. However, instead of a state value $V(s_t)$, there is instead a state-action pair $Q(s, a)$ estimate table that the algorithm updates. The update rule is similar to Equation 1, and is shown in Equation 2. Each state-action pair is updated by its old value plus an estimate of the next reward given that the best move is taken. Specifically, $\max_a(Q(s_{t+1}, a)$ is the maximum reward possible by moving to the next state. The typical policy to follow is $\epsilon - Greedy$, that is, with some probability ϵ a random action is taken, otherwise the most desirable action is taken (greed).

$$Q(s_t, a_t) \leftarrow Q(s_t, a_t) + \ldots \alpha \left(r_t + \gamma \max_a (Q(s_{t+1}, a)) - Q(s_t, a_t) \right) \quad (2)$$

The general algorithm is be described succinctly by its Richard Sutton in chapter 6.5 of [SB98]. The algorithm is shown below:

```
Initialize Q(s,a) arbitrarily
Repeat (for each episode)
Initialize s
Repeat
Choose a from s
Take action a, observe r, s'
Update Q by equation 2
s ← s'
until s is terminal
```

This is an incremental learning process, and by choosing the maximum-valued action we are using a greedy policy. Q-Learning has been shown to converge closely to the optimal value given that it follows a 'soft' policy[MR07]. A soft policy replaces the $\max_a(Q(s_{t+1}, a))$ in Equation 2 with a random action with probability $1 - \epsilon$. This enables the agent to both explore and exploit what it's learned so far.

3 The Experimental Environment

I tested my Q-learning agent in a simple gridworld. My grid world is a simplified version of the one used by Cay Horstmann's in the AP Computer Science Exams of 2008[Hor08]. Unlike his, my version only allows for movement in the four cardinal directions. Also, the gridworld is constructed so that there are 4 possible states for each gridspace: empty, invalid, food, and goal. Each of these states dictates an amount of reward that the agent will receive upon entering that space. A simple example map is shown in Figure 1. In addition to the 3 states listed above, a space can also be invalid which will cause the episode of the learning session to restart.

Each world is saved in a 256-Color bitmap file. Using specific RGB values, one can build a map of any

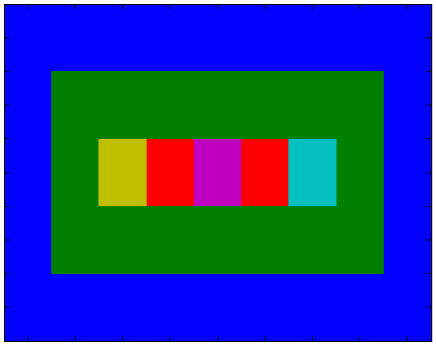


Figure 1: An example of a simple gridmap – The blue is an invalid space, yellow is the starting location, red is food, and cyan is the goal space. The agent is represented by the purple square.

State	RGB Value	Matlab Value
Start	(34,177,76)	113
Wall(Invalid)	(237,28,36)	79
Food	(63,72,204)	210
Empty	(255,255,255)	255

Table 1: The pairings between the states of each grid space and their Matlab Code, as well as RGB value for map creation.

size using a simple graphics editor. The RGB values for each of the 4 possible grid states are listed in Table 1 below. Using Matlab's imread function, we can load the image in and convert it to a proper environment for the agent. The RGB values are converted by Matlab to a different value which is also detailed in the table.

In the initial experiment I used the simple map shown in Figure 1, the difficulty was then scaled to a version which I estimate at a moderate difficulty, shown in Figure 2a. The last map I tried (shown in Figure 2b) was one similar to crossing a bridge with cliffs on the sides. The difficulty of each of these grid worlds is also tied to how long the paths are. A longer path will allow for more accumulation of total reward, but also can be more difficult to explore because of the greedy method employed by the agent.

The parameters to Q-learning set for the experiment are as follows: $\epsilon = 10\%$(greedy), $\alpha = .7$, $\gamma = .1$, with 100 episodes per run. On the first map (Figure 1) the average runtime over twelve runs was 0.5986 seconds. The moderate difficulty was run 12 times with an average of 1.8627 seconds per episode. And the last map's average runtime was 3.5001 seconds. The grid world parameters for rewards are shown in Table 2

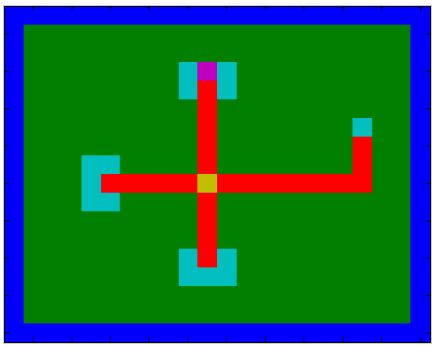
State	Reward
Wall	-30
Empty	-1
Goal	10
Food	2
Invalid	-100

Table 2: Rewards for each square of the gridworld

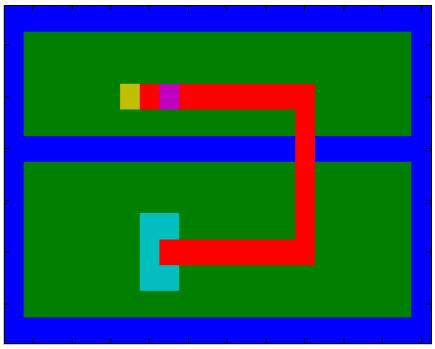
4 Initial Results

Using the parameters described above, the initial Q-Learning of the grid world is shown by average goal and failure counts in Table 3. The number of possible state-action pairs is also displayed in an attempt to give an intuition of the difficulty of the problem, as well as the standard deviation of the number of goals over 10 runs. The number of possible states are dependent on the number of actions as well as the possible number of grid positions. For example, the simple test map is a 5×9 grid and there are a total of 4 actions. Therefore, the possible number of Q-Pairings is $5 \cdot 9 \cdot 4 = 180$. The path defined by the map itself also comes into play when observing difficulty. A longer path is more difficult to learn because it requires exploring the entire path while still making use of the nature of the ϵ-greedy policy to go far enough down the path to explore. A typical issue that seemed to occur is the oscillation between two food spaces side by side when the next down the path was not explored, this is discussed more in the next section.

The first map was easily solved by the algorithm. This was probably due to the small path between the start state and the goal, as well as the simplicity of the path itself. With only 3 food patches between the starting area and the goal, it was usually difficult for the algorithm to not realize that right actions were the most valuable. Because of the stochastic nature of the algorithm ($\epsilon = .1$) even once the proper path was found, there was still a minute chance of reaching an invalid state because of the small size of the map



(a) **The Second Trial Map** – estimated to be easy to reach a goal, more difficult to reach the maximum rewarding state to the right.



(b) **The Bridge Experiment** – A much more difficult map, there is only way to cross a bridge to the goal. The total path length is also much longer.

Figure 2: The other two trial maps used in experimentation

(The chance to hit an invalid square from most food square was about one percent if the two actions was a result of the random movements).

The second map was considered harder than the first due to the search space size. However, one can see that the bottom path is only a single food-space larger than the first map and only the rightmost path is nonlinear. This rightmost path is the optimal path for reward, as it involves picking up the most food. But due to the distance necessary to travel to the goal point was often left unexplored because easier solutions were found first.

The third map, although having a smaller search space then the second, was considerably harder due to the strict path necessary to get to the goal. The "bridge" portion being the only valid space to move to the bottom section where the goal lay was a significant obstacle in the agent's way. Also, the oscillatory motions seen in longer paths due to the ϵ-greedy policy inhibits exploration significantly enough that in many cases the bottom section of the map was never even explored.

Obviously something must be done to inhibition this effect. A few possible choices are to cause the agent to "pick up" food that it moves over, effectively removing any duplicate rewards the agent might gain by moving to that area again. However, to implement this would undermine the ability to learn more than one action that can lead to any state in a single run. Another method could be to implement the policy *softmax*[SB98] which assigns weights to each of the actions and then randomly selects one, taking the rank of each action into account. By doing this, we cause the worst action (such as running into an invalid zone) to become unlikely to be selected, but give it enough of a chance that exploration is not hurt.

The difficulty of each map is certainly related between the number of possible goals, invalids, food spaces as well as the length of paths. The algorithm itself seems to fail on longer paths due to its greedy nature and a possible solution is presented in the next section.

Map	Goals	Failures	Search Size	Std. Dev.
1	863.5	136.5	180	15.8272
2	817	183	1584	45.6898
3	6.3	993.7	1056	6.9450

Table 3: The Initial results of Q-Learning on each map and their search sizes. Results averaged over 10 runs, with standard deviation of goals displayed.

5 Oscillatory Movements and Hypothetical Solutions

To fully understand the issue of the oscillations, let us consider the simple map's initial path with some example values. Initially, all estimated values are zero. In the first step, the algorithm happens to move rightward, since the space is a food space, the algorithm receives reinforcement which ends up equating a value of .25 with moving rightward from the initial space. Once upon this space, the algorithm once again searches in its four cardinal directions, the algorithm happens to notice that to its left is a valuable space, and all the other are zero. Thus the algorithm moves back to the space it was previously and then continues this pattern until the small chance ϵ breaks it out towards the next food space.

0	0	0	Initial Values
.25 $\rightarrow$	0	0	One step
.25 $\rightarrow$	$\leftarrow$ .25	0	Second Step
.25 $\rightarrow$	$\leftarrow$ .50	0	Third Step
.50 $\rightarrow$	$\leftarrow$ .50	0	Fourth Step
...			etc.

This is very clearly a problem arising from the equal value of adjacent spaces. A possible mechanism to fix this would be to start all state-action pairs in a random state. This would certainly allow for a solution to the above problem, but the time until convergence could be much much longer due to requiring to learn the proper values for all possible states being visited, *and to correct any errors in the random numbers*. Depending on the range of the random numbers, this correction of error could take a long period of time.

A more sensible approach would be to use the softmax policy mentioned in the previous section. This would enable exploration of more likely candidates for solutions, however given that the higher rank is likely to be chosen, the same oscillatory problems may occur. Despite the oscillations, the small ϵ movements may have enough effect on the distributions of the weights for each candidate action that the oscillations could be undone. However, this method is still relying on random chance itself in order to break free, and therefore is not the best strategy.

Since the random variable seems to be able to stop these oscillations, perhaps increasing the probability of random actions would be effective. However, if ϵ is chosen too large, then we effectively have random search, and might as well throw Q-Learning's entire algorithm out the window. Instead, a more effective method would be a *variable* ϵ, defined to change in time in relation to the number of episodes. Starting this number out high, and lowering it asymptotically to the current $\epsilon = .1$. This would allow for a larger exploration initially in order to learn the state-action states better, while leveling off as the algorithm learned which actions take it to the goal state. If oscillations still occurred later on after ϵ leveled off, the small amount of random chance would still be able to kick it out of its loop.

Another approach, also mentioned above briefly, would be to cause the rewarding states of the grid to be "picked up" by the agent. This would double the overhead of the algorithm in order to keep track of which state-action pairs it had already seen in a single episode, however, by removing the continuous reinforcement of adjacent spaces can exert on each other would remove oscillations entirely from the space.

6 Implemented Approaches to Oscillatory Movements

To attempt to dampen and reduce the effects of oscillatory movements during Q-Learning the following methods were implemented:

6.1 Variable ϵ

Defined by equation 3, the current $\epsilon_t(e_{cur})$ is based on which episode e the training cycle is on. The chance of a random move we want to approach is

Map	Goals	Failed Trials	Std Dev
1	892.6	12	313.6347
2	895.6	30	310.8191
3	0	All	NA

Table 4: Results of Variable ϵ on each map with 1000 episodes. If a trial had to be forced to quit due to being stuck in oscillation this is indicated as a failed trial.

labeled ϵ_0 and the number of episodes we would like it to take to reach that value is denoted e_{max}.

$$\epsilon_t(e_{cur}) = \frac{(\epsilon_0 - 1)}{e_{max}} * e + 1 \quad (3)$$

This procedure enables maximum exploration during the beginning phrases and a more intelligent exploitation approach during the later part of the agents life. As e approaches e_{max} we hoped to see the oscillations dampened and goal achievement raised, but during many trials the agent would become fixed in an infinite oscillation in an early episode, and the program needed to be stopped. Because of this, it's clear that a variable ϵ that starts with initial random movement is not a good choice to dampen oscillations. The results on each map are stated in table 4, note that if a trial could not be completed due to the program needing to be stopped this is indicated as a failed trial.[3]

6.2 Removing Rewards for Repeated States

Since the problem of oscillations comes from the reward of repeated states it makes sense to stop the problem at its source. We can do this in a few different ways. We can prohibit rewards for repeated states entirely with Remove Repeated States(RRS), remove repeated state rewards only for food rewards (RRFR), or we can prohibit any reward for any repeated opposing actions (ROA). All of these methods increase the number of goals in all maps, as well as dampen oscillations significantly. Table 5 shows the average goal return over 1000 episodes for 10 runs with parameters identical to those in §3. The ROA method showed the most significant increase,

[3]Only completed trials were used for calculation of averages and standard deviations.

Map	Goals	Failures	Std Dev
RRS			
1	870.1	129.9	25.8390
2	940.1	59.9	21.3096
3	134.7	856.3	51.9659
RRFR			
1	791	209	67.6609
2	783	217	119.6931
3	35.7	964.3	55.1544
ROA			
1	875.1	124.9	14.0906
2	941.8	58.2	23.7431
3	221.9	778.1	43.1237

Table 5: The results of the new methods RRS,RRFR, and ROA on each map given 1000 episodes. Resulted averaged over 10 runs. (All parameters the same as original algorithm runs.)

and outperformed the RRS and RRFR algorithms as well as the original Q-Learning algorithm.

RRFR attempts to solve the oscillation problems by causing the agent to "pick up" food during an episode. This removes the reward for visiting a food space twice and helps to discourage oscillatory patterns in movement. This strategy allows for negative reinforcement to occur for being off the food path, which can result in oscillations not on the path. Despite back and forth movements still being possible, they were seen far less in trials, and this first come, first served reward policy definitely improved the number of goals recorded on all maps.

Expanding on RRFR, RRS functions on the same see-first principle, but applied to all rewards not just food. The first reward it sees for a state-action is the only reward it ever considers during that episode. On the second visit to any space, *from any action* results in no reward or penalty. This effectively reduces the algorithm to reinforcing only its first choices. This effectively counters the oscillations in movement because the reward for moving back and forth is no longer there; without this reinforcing reward for oscillations the level of oscillations decrease dramatically. RSS performed much better than the original algorithm on all maps, and better than RRFR as well.

The last method, ROA performed very well on all maps, and considerably better than the original al-

Map	Goals	Failures	Std Dev	Most Goals
1	8918.3	1081.7	33.0052	8952
2	9557.4	442.6	77.7163	9647
3	2434.8	7565.2	274.55	2865

Table 6: Extended ROA results, 10 runs with 10,000 episodes per run. Averages and maxima are shown

gorithm. The most goals ever recorded during the 10 trials was 357 – a drastic difference from the 2 goals attained by the original algorithm and more than twice that of RRS. An opposing action is seen as the move that causes a change to the state to be undone. A left move is opposed by a right, and an upward by a downward. By realizing that oscillatory movements are nothing but a switch between choosing left then right then left, and restricting the reward to no longer reward such moves, we result in an oscillatory resistant algorithm. Stated in a different way, the actions to move left and right are considered the same in the visitation matrix that each episode keeps track of. This decreases the overhead that one would need to keep track of all visitations from all directions, and also dampens oscillations. Because ROA performed the best, it was ran again with 10000 episodes for each map. These results are displayed in Table 6 and show that it seems to be a trend that roughly one fifth of the total episodes result in success on the third map.

7 Conclusions

The issue of oscillation in the grid world arises from the nature of the estimation function. Because the greedy policy insures that almost all of the time the most valuable state is selected for the next action, a single wrong move can result in an increasing oscillatory estimate of two adjacent states rewards. To protect the agent from making repeated mistakes during difficult navigations multiple strategies were proposed. Out of the strategies proposed, the Repeated Opposing Actions algorithm (ROA) resulted in the best results. This algorithm resulted in a consistent significant increase in number of successful episodes compared to the initial results. The other two methods involving removing rewards for repeated actions also showed a performance increase over the initial results as well. The proposed variable ϵ approach did not work very well and actually resulted in a number of failed runs due to infinite oscillations.

8 Future Work

For future work, one could change equation 3 to not start at completely random, but a more modest upper bound on the exploration. Also, initial hybrids between variable ϵ and ROA are showing results less impressive than standard ROA, but this could be due to the stochastic nature of variable ϵ interfering with the learning process in general. Also, increasing the complexity of the actions to include the ordinal directions could result in more interesting movement patterns and grid-world design and mazes to be observed.

References

[Hor08] Cay Horstmann. Gridworld case study. horstmann.com, Accessed April 14, 2012, 2008.

[MR07] Francisco S. Melo and M. Isabel Ribeiro. Q-learning with linear function approximation. In *Proceedings of the 20th annual conference on Learning theory*, COLT'07, pages 308–322, Berlin, Heidelberg, 2007. Springer-Verlag.

[SB98] Richard S. Sutton and Andrew G. Barto. *Reinforcement Learning I: Introduction*. 1998.

[Sut88a] Richard S. Sutton. Learning to Predict by the Methods of Temporal Differences. *Machine Learning*, 3:9–44, 1988.

[Sut88b] Richard S. Sutton. Learning to predict by the methods of temporal differences. In *MACHINE LEARNING*, pages 9–44. Kluwer Academic Publishers, 1988.

[WD92] C.J.C.H. Watkins and P. Dayan. Q-learning. *Machine learning*, 8(3):279–292, 1992.