Q-Learning and Collection Agents

Tom O'Neill Leland Aldridge Harry Glaser

CSC242, Dept. of Computer Science, University of Rochester

{toneill, hglaser, la002k}@mail.rochester.edu

Abstract

Reinforcement learning strategies allow for the creation of agents that can adapt to unknown, complex environments. We attempted to create an agent that would learn to explore an environment and collect the trash within it. While the agent successfully explored and collected trash many times, the training simulator inevitably crashed as the Q-learning algorithm eventually failed to maintain the Q-value table within computable bounds.

Introduction

In complex environments, it is almost impossible to account and program for every possible scenario than an agent may face. Reinforcement learning is one solution to this problem because it allows for the creation of robust and adaptive agents (Russell and Norvig[(2)]).

Our goal was to develop an agent that when placed in an unknown environment, the agent would explore the environment, collect any trash found, and finally return to its starting location once all of the trash had been collected. We chose to develop a learning agent such that we could train it in many simulated environments thus allowing it to learn the optimal behaviors of a trash collection agent. Once the agent was trained, it was expected to be able execute the learned behaviors in any relatively similar environment and perform well.

According to Russell and Norvig[(2)], two popular reinforcement strategies are active and passive learning. In both cases agents make decisions based on expected utilities, though they differ in the method in which the utility of states is determined. We briefly explain the difference between these strategies to motivate the decision to use active learning in our environment.

Passive Learning

With passive learning, agents require a model of the environment. This model tells the agent what moves are legal and what the results of actions will be. This is particularly useful because it allows the agent to look ahead and make better choices about what actions should be taken. However, passive

learning agents have a fixed policy and this limits their ability to adapt to or operate in unknown environments.

Active Learning

Unlike passive learning agents, active learning agents do not have a fixed policy. Active learning agents must learn a complete model of the environment. This means that the agent must determine what actions are possible at any given state since it is building a model of its environment and does not yet have an optimal policy. This allows active learning agents to learn how to effectively operate in environments that are initially unknown. However, the lack of a fixed policy slows the rate at which the agent learns the optimal behaviors in its environment.

Motivation for Active Learning

We chose to explore the behavior of a collection agent in an unknown environment. The agent was expected to learn how to optimally move around the environment while collecting trash, and ultimately make its way back to the starting point once all of the trash was collected.

Since the shape of the environment, trash distribution, and starting location of the agent are all unknown to the agent, we chose to use an active learning reinforcement technique called Q-learning.

The Q-Learning Algorithm

Since Q-learning is an active reinforcement technique, it generates and improves the agent's policy on the fly. The Q-learning algorithm works by estimating the values of state-action pairs. The purpose of Q-learning is to generate the Q-table, Q(s,a), which uses state-action pairs to index a Q-value, or expected utility of that pair. The Q-value is defined as the expected discounted future payoff of taking action a in state s, assuming the agent continues to follow the optimal policy (Russell and Norvig[2]).

Q-learning generates the Q-table by performing as many actions in the environment as possible. This initial Q-table generation is usually done offline in a simulator to allow for many trials to be completed quickly. The update rule for setting values in the table is as follows in Eqation-1:

$$Q[s,a] \leftarrow Q[s,a] + \alpha(N_{sa}[s,a])(r + \gamma \max_{a'} Q[s',a'] - Q[s,a])$$

Eqation-1

In Equation 1, α is the learning factor and γ is the discount factor. These values are positive decimals less than 1 and are set through experimentation to affect the rate at which the agent attempts to learn the environment. $N_{sa}[s,a]$ is a table of frequencies indexed by the same state-action pair, the value of the table is equivalent to the number of times the agent has attempted that state-action pair. The variables *s* and *a* represent the current state and action of the agent, respectively. Finally, r is the reward from performing s' and *a',* the previous state and action, respectively. (Russell and Norvig[2])

The initial entries of $N_{sa}[s,a]$ are 0 for all pairs, and the initial values for $Q[s,a]$ are arbitrary. The values of $Q[s,a]$ can be arbitrary because the effect of this update rule after many trials is to essentially average neighboring expected utilities of state-action pairs to a smooth and near-optimal dynamically generated policy.

Simulator Design

We developed a simulator to train the agent offline in many different environments. The simulator generated randomized environments, maintained the structures necessary for Q-learning, and provided appropriate feedback to the agent such that it could learn.

The Agent and Environment

In every case, the agent explored and collected trash in an *N* by *N* obstacle-free grid world. The agent always started at position (0, 0) and could move north, south, east, and west, and pickup trash. The agent did not know where it was in the grid world, nor could it tell if it was at a boundary of the world. Additionally, the agent did not have any sight – it could not tell if trash was nearby beyond receiving a reward for successfully picking up trash (assuming it existed in the agent's grid square). Actions always executed successfully with the expected outcome, with the exception that moving into a wall at a boundary square did not move the agent, and picking up trash in a clean square did not trigger a reward.

Trash was distributed randomly throughout the grid world each trial with a 20% chance of any cell having trash in it. The results in this paper were generated using a 4 by 4 grid, though both the percent of trash and size of the grid are arbitrary.

Rewards

The agent received rewards of 20 points for collecting trash and 50 points for returning home after all trash had been collected. For each movement or empty pickup attempt, the agent was penalized 1 point. These values were chosen arbitrarily and varied during experimentation in an attempt to produce optimal behavior.

Simulator Implementation

This section describes some of the important implementation details of the simulator. The complete implementation of the simulator, written in python, can be found in the in the appendix of this paper.

In this simulator, a state is defined as a 3-tuple of:

(currentPosition, trashAtLocation, numTrashCollected)

- *currentPosition* is a 2-tuple of integers representing of the agent's grid location (x, y)
- *trashAtLocation* is a Boolean, True if there is/was trash at the agent's current position and False otherwise
- *numTrashCollected* is a integer representing the number of trash items collected thus far

We used this style of state description so that we could account for each important dimension governing the agent's behavior. The agent needs to learn the value of each grid square as it varies with the number of each trash items collected and whether or not the agent has already found trash at that location.

Main Loop

The entire simulator was driven by a short loop that evaluated the previous action and then chooses the best available action, performs the action, and then repeats.

```
1   # main loop
2   firststate = ((0,0),grid[0][0], 0)
3   action = Qlearning(Q_INIT, firststate)
4   while True:
5       actions += 1
6       reward = takeAction(action)
7       action = Qlearning(reward, (curPos, grid[curPos[0]][curPos[1]], picked))
8
```

Main Loop of TrashMan.py

Lines 2 and 3 initialize the starting position and value of the first state such that lines 6 and 7 can repeatedly generate the explore and collect policy of the agent. Line 5 acts as a measure of performance for each trial of the agent, fewer actions are expected on each trial as the agent's policy gets closer to the optimal policy for this environment.

Taking Actions

Taking an action must update the agent's state, but it cannot update the environment until after the Q-learning function has had a chance to analyze the rewards associated with that action in the environment. Additionally, the only

chance to calculate rewards in the simulator is when the agent takes actions. Thus the combined method for taking actions and calculating rewards:

```
1   # take the specified action: returns the reward for that action
2   def takeAction(action):
3       global curPos, trashes, grid, picked, actions
4       x, y = curPos
5       reward = PENALTY_ALIVE
6       if action == ACTION_PICKUP:
7           if grid[curPos[0]][curPos[1]]:
8               reward = REWARD_PICKUP
9       elif action == ACTION_NORTH:
10          if y < GRID_LENGTH - 1:
11              y += 1
12      elif action == ACTION_EAST:
13          if x < GRID_LENGTH - 1:
14              x += 1
15      elif  action == ACTION_SOUTH:
16          if y > 0:
17              y -= 1
18      elif action == ACTION_WEST:
19          if x > 0:
20              x -= 1
21
22      curPos = (x, y)
23      if curPos == (0, 0) and trashes == 0 and reward != REWARD_PICKUP:
24          reward = REWARD_SUCCESS
25
26      return reward
```

Action Taking and Reward Calculating Function

Line 5 shows the default reward for any action is the penalty for being alive. If the agent successfully collects trash (lines 6-8) or is at the starting location after collecting all of the disbursed trash (lines 23-24), then the agent is rewarded instead of penalized. Finally, on each of the agent's movement actions the position of the agent is checked and the action is ignored if it would move the agent outside the environment.

Learning

By far the most complicated function in the simulator is the function that implements the Q-learning aspect of the agent's behavior. The function must first compute the value of taking that action based on the received reward and then calculate the next best action to take. Finally the environment must be updated to reflect the results of the agent's actions before the next action can be taken.

```
1   # Q-learning function
2   def Qlearning(reward, state):
3       global prevState, prevAction, prevReward, trashes, grid, actions, picked
4       if prevState != None:
5           if not Q.has_key((prevState, prevAction)):
6               Q[(prevState, prevAction)] = Q_INIT
7
8           if not Nsa.has_key((prevState, prevAction)):
9               Nsa[(prevState, prevAction)] = 0
10
11          # update visited states
12          Nsa[(prevState, prevAction)] += 1
13
14          # Q-learning equation
15          Q[(prevState, prevAction)] += ALPHA * Nsa[(prevState, prevAction)] * (reward +
    GAMMA * getMaxQ(state) - Q[(prevState, prevAction)])
16          #debug:print prevState, prevAction, Q[(prevState, prevAction)]
17
```

Agent's Q-Learning Function (Part 1 of 2)

First, the function checks to see if that state-action pair has ever been tried before and if it hasn't, the state-action pair is initialized with an arbitrary value in the Q-table (lines 5-6). Likewise, if the frequency table does not have a value for the state-action pair, the value is initialized at 0 (lines 8-9). Once the two important tables are set, the frequency of the state-action pair is incremented (line 12) before our implementation of Equation 1 is executed.

Line 15 of this function is probably the most important line in the simulator – it's where the learning takes place! All of it should look familiar from Equation

1, with the exception of *getMaxQ(state)* which merely returns the highest possible Q-value for an action in the specified state.

```
17
18        # updates
19        pos = state[0]
20        if reward == REWARD_PICKUP:
21            grid[curPos[0]][curPos[1]] = False
22            trashes -= 1
23            picked += 1
24
25        if reward == REWARD_SUCCESS:
26            prevState = None
27            prevAction = None
28            prevReward = None
29            print "GRID CLEAN! actions:", actions
30            actions = 0
31            grid = makeGrid()
32            picked = 0
33        else:
34            prevState = state
35            prevAction = argmaxexplo(state)
36            prevReward = reward
37
38        return prevAction
```

Agent's Q-Learning Function (Part 2 of 2)

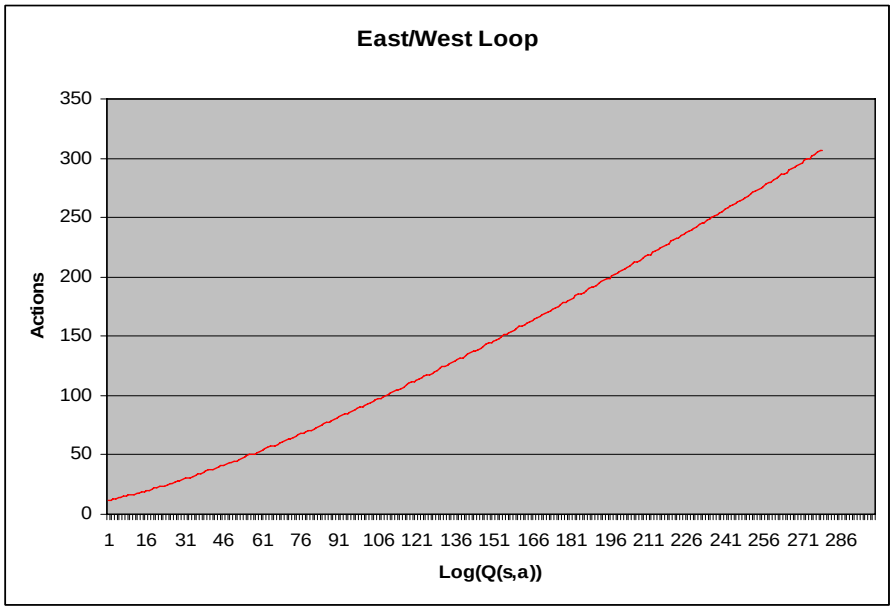
After the completion of line 15, the environment can be updated to reflect the result of the actions. If the agent picked up trash (lines 20-23), then the trash must be removed from that location and the relevant accounting variables updated. If the agent completed its task, then the world must be reset with a new trash grid and the agent's action history must be cleared (lines 25-32). Otherwise, the function sets up the state and action variables so that the function can be called again after the action it returns is executed.

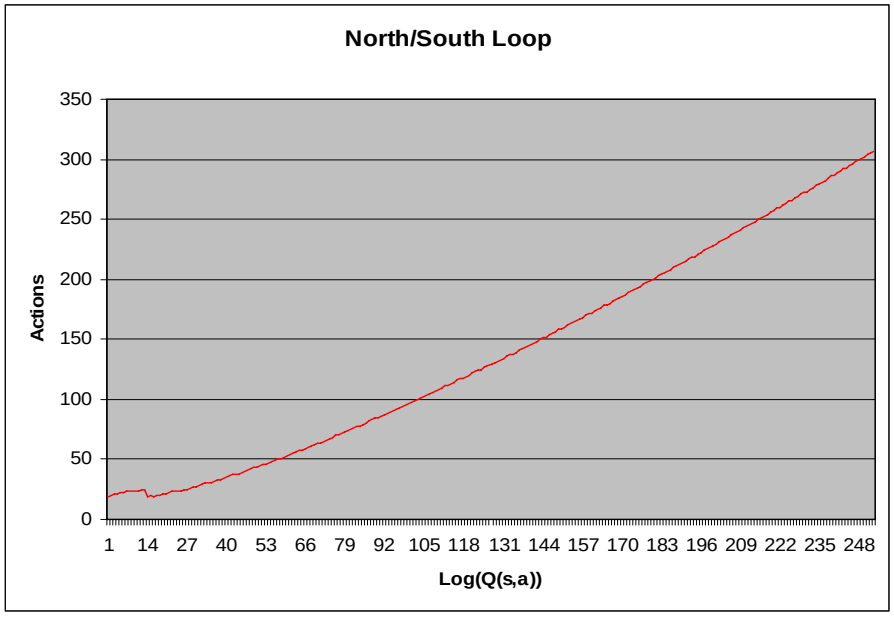
Results

After running the simulator for many trials with many different values for the constants (α, γ, N, reward scale values, exploration tendency, etc) we could

not get the Q-value table to converge. In every configuration, the simulator would oscillate between two state-action pairs and repeatedly increase each pair's Q-value. Since each member of the oscillation loop was the other member's previous state-action pair, the rate of growth for both Q-values was exponential.

The following two graphs show the exponential growth of the Q-values. The X-axis is in a $\log_{10}$ scale. As shown, both oscillating cases grew their Q-values to over 10^{250} in less than 350 action steps before crashing the simulator.





Due to diverging behavior of the Q-values in the Q-table, we were unable to construct an agent that learned a useable collection strategy and hence cannot report on the results of a meaningful policy.

Discussion and Future Work

It's important to note that the simulator's Q-table did not immediately diverge. The agent was able to successfully complete its task many times before the seemingly inevitable divergence. It's unclear whether or nor the agent successfully learned even part of an optimal policy before the Q-table diverged.

The divergence of the Q-value table was not entirely unexpected. Gordon[(1)] and Weiring[(3)] have separately discussed the nature of Q-learning and the inability to guarantee its convergence in many circumstances. Gordon specifically addresses the problem of oscillations leading to divergence and

Weiring discusses how off-policy reinforcement learning methods often cause the system to diverge.

One possible reason for the divergence in our system is that the environment is too dynamic for Q-learning's standard update style. The standard update can result in very fast swings of Q-values, and thus quickly break the averaging nature of Q-learning.

We believe that the Q-learning algorithm for active reinforcement learning in this environment may not have been the correct algorithm to use, specifically because of the diverging results we encountered.

Other active learning algorithms with stronger convergence properties and averaging updating methods instead of standard updating methods may fair better in this environment. Additionally, it's very possible to describe this environment with different state-action pairs that could be better suited for learning then those currently used by our simulator.

Distribution of Work

- Tom O'Neill wrote this paper
- Leland Aldridge was the mathematical brains behind understanding and implementing Q-learning
- Harry Glaser coded most of the simulator and read parables about learning during debugging sessions

Notes to TAs

CB suggested we deviate from the project spec and pursue Q-learning applied to an agent in a garbage collection environment - given the amount of passive learning code already supplied made the "hook it up to quagents and run it" significantly less of a learning project than implementing something like Q-learning. Thanks!

References

1. Gordon, *Reinforcement learning with function approximation converges to a region. Advances in Neural Information Processing Systems*. The MIT Press, 2001. http://citeseer.ist.psu.edu/gordon01reinforcement.html
2. Russell and Norvig, *Artificial Intelligence A Modern Approach, Second Edition.* Prentice Hall, 2003.
3. Weiring, *Convergence and Divergence in Standard and Averaging Reinforcement Learning.* Intelligent Systems Group, Institute of Information and Computer Sciences, Utrecht University. http://www.cs.uu.nl/people/marco/GROUP/ARTICLES/ecml_rl_convergence.pdf 4/21/2006.

Appendix: TrashMan.py Source Code

```
1   # Q-learning agent for "trash world"
2   #
3   # by Leland Aldridge, Harry Glaser, Tom O'Neill
4   # for CSC 242 Project 4: Learning
5   # 4/21/2006
6
7   import sys, random
8
9   # useful utilities
10  def debug(s):
11      if DEBUG:
12          print s
13
14  def enum(*args):
15      g = globals()
16      i = 0
17      for arg in args:
18          g[arg] = i
19          i += 1
20      return i
21
22
23  # GLOBALS ###################################################################
24
25  # Macros
26  NUM_ACTIONS =  enum('ACTION_PICKUP',
27                      'ACTION_NORTH',
28                      'ACTION_EAST',
29                      'ACTION_SOUTH',
30                      'ACTION_WEST')
31  DEBUG = False
32  GRID_LENGTH = 4
33  THRESH = .2
34  TRASH_CHAR = '1'
35  CLEAR_CHAR = '-'
36  Q_INIT = 0
37
38  # Learning tables
39  Q = {}    # indexed by (a, s) utility of taking action a in state s
40  Nsa = {} # indexed by (a, s) number of times action a has been taken when in state s
41  trash = {}
42
43  # Learning parameters
44  ALPHA = 0.5 # learning factor
45  GAMMA = 0.5 # discount factor
46  N = 5       # number of times to try an action to get a good feel for it
47
48  # globals
49  trashes = 0      # num trashes left
50  picked = 0       # num trashes collected
51  curPos = (0, 0) # current position
52  grid = []        # the world
53  actions = 0      # num actions taken so far
54
```

```
55    # remembered information
56    prevState = None
57    prevAction = None
58    prevReward = None
59
60    # Utilities
61    REWARD_SUCCESS = 50
62    REWARD_PICKUP = 20
63    PENALTY_ALIVE = -1
64
65
66    # read grid from file
67    def readGrid(filename = 'quagent.itemgrid'):
68        line_num = 0
69        itemGrid = []
70        for i in range(GRID_LENGTH):
71            itemGrid += [[]]
72            for j in range(GRID_LENGTH):
73                itemGrid[i] += [False]
74
75        infile = open(filename)
76        i = -1
77        for line in infile:
78            i += 1
79            j = -1
80            if i >= GRID_LENGTH:
81                break
82            for c in line:
83                j += 1
84                if j >= GRID_LENGTH:
85                    break
86                if c == TRASH_CHAR:
87                    itemGrid[i][j] = True
88                elif c != CLEAR_CHAR:
89                    sys.stderr.write('Error: invalid token, line ' + str(i) + ', col ' + str
      (j) + ': ' + c + '\n')
90        return itemGrid
91
92    # construct random grid
93    def makeGrid():
94        global trashes
95        itemGrid = []
96        for i in range(GRID_LENGTH):
97            itemGrid += [[]]
98            for j in range(GRID_LENGTH):
99                if random.random() < THRESH:
100                   itemGrid[i] += [True]
101                   trashes+=1
102               else:
103                   itemGrid[i] += [False]
104
105       return itemGrid
106
107   # print grid to stdout
```

```
108     def printGrid(itemGrid):
109         for i in itemGrid:
110             for j in i:
111                 sys.stdout.write(str(j) + ' ')
112             sys.stdout.write('\n')
113 
114 
115     # take the specified action: returns the reward for that action
116     def takeAction(action):
117         global curPos, trashes, grid, picked, actions
118         x, y = curPos
119         reward = PENALTY_ALIVE
120         if action == ACTION_PICKUP:
121             if grid[curPos[0]][curPos[1]]:
122                 reward = REWARD_PICKUP
123         elif action == ACTION_NORTH:
124             if y < GRID_LENGTH - 1:
125                 y += 1
126         elif action == ACTION_EAST:
127             if x < GRID_LENGTH - 1:
128                 x += 1
129         elif  action == ACTION_SOUTH:
130             if y > 0:
131                 y -= 1
132         elif action == ACTION_WEST:
133             if x > 0:
134                 x -= 1
135 
136         curPos = (x, y)
137         if curPos == (0, 0) and trashes == 0 and reward != REWARD_PICKUP:
138             reward = REWARD_SUCCESS
139 
140         return reward
141 
142 
143     # returns the highest possible Q-value for an action in the specified state
144     def getMaxQ(state):
145         action = None
146         max = 0
147 
148         for curAction in range(NUM_ACTIONS):
149             key = (state, curAction)
150             if not Q.has_key(key):
151                 Q[key] = Q_INIT
152             if action == None:
153                 action = curAction
154                 max = Q[key]
155             else:
156                 n = Q[key]
157                 if n >= max:
158                     action = curAction
159                     max = n
160         return max
161 
```

```
162     # returns the current reward for exploring further
163     def exploration(utility, frequency):
164         if frequency < N:
165             return REWARD_SUCCESS
166         else:
167             return utility
168 
169     # determines the best action to take in the current state
170     def argmaxexplo(state):
171         bestaction = -1
172         oldmaxexplo = 0
173 
174         for action in range(NUM_ACTIONS):
175             key = (state, action)
176             if not Q.has_key(key):
177                 Q[key] = Q_INIT
178             if not Nsa.has_key(key):
179                 Nsa[key] = 0
180 
181             e = exploration(Q[key], Nsa[key])
182 
183             if bestaction == -1:
184                oldmaxexplo = e
185                bestaction = action
186 
187             if e > oldmaxexplo:
188                 oldmaxexplo = e
189                 bestaction = action
190         return bestaction
191 
192     # Q-learning function
193     def Qlearning(reward, state):
194         global prevState, prevAction, prevReward, trashes, grid, actions, picked
195         if prevState != None:
196             if not Q.has_key((prevState, prevAction)):
197                 Q[(prevState, prevAction)] = Q_INIT
198 
199             if not Nsa.has_key((prevState, prevAction)):
200                 Nsa[(prevState, prevAction)] = 0
201 
202             # update visited states
203             Nsa[(prevState, prevAction)] += 1
204 
205             # Q-learning equation
206             Q[(prevState, prevAction)] += ALPHA * Nsa[(prevState, prevAction)] * (reward +
    GAMMA * getMaxQ(state) - Q[(prevState, prevAction)])
207             #debug:print prevState, prevAction, Q[(prevState, prevAction)]
208 
209         # updates
210         pos = state[0]
211         if reward == REWARD_PICKUP:
212             grid[curPos[0]][curPos[1]] = False
213             trashes -= 1
214             picked += 1
```

```
215         elif reward == REWARD_SUCCESS:
216             prevState = None
217             prevAction = None
218             prevReward = None
219             print "GRID CLEAN! actions:", actions
220             actions = 0
221             grid = makeGrid()
222             picked = 0
223         else:
224             prevState = state
225             prevAction = argmaxexplo(state)
226             prevReward = reward
227 
228         return prevAction
229 
230     # init and main loop
231     def run():
232         global trashes, curPos, grid, picked, actions, ALPHA, GAMMA, N
233 
234         # init
235         grid = makeGrid()
236         printGrid(grid)
237 
238         ALPHA = float(sys.argv[1])
239         GAMMA = float(sys.argv[2])
240         N = int(sys.argv[3])
241 
242         print "ALPHA: ", ALPHA, "\nGAMMA: ", GAMMA, "\nN: ", N, "\nGRID_LENGTH: ",
    GRID_LENGTH
243 
244         # main loop
245         firststate = ((0,0),grid[0][0], 0)
246         action = Qlearning(Q_INIT, firststate)
247         while True:
248             actions += 1
249             reward = takeAction(action)
250             action = Qlearning(reward, (curPos, grid[curPos[0]][curPos[1]], picked))
251 
252 
253     run()
254 
255 
```