RESEARCHING THE ABILITY OF REINFORCEMENT LEARNING AGENTS TO ADAPT TO CHANGES IN THEIR TRAINING ENVIRONMENT

Tomos Ody (ODY14527411)
Department of Computer Science, University of Lincoln

***Abstract*—The quick brown fox jumped over the lazy dog Lorem ipsum dolor sit amet, consectetuer adipiscing elit. Ut purus elit, vestibulum ut, placerat ac, adipiscing vitae, felis. Curabitur dictum gravida mauris. Nam arcu libero, nonummy eget, consectetuer id, vulputate a, magna. Donec vehicula augue eu neque. Pellentesque habitant morbi tristique senectus et netus et malesuada fames ac turpis egestas. Mauris ut leo. Cras viverra metus rhoncus sem. Nulla et lectus vestibulum urna fringilla ultrices. Phasellus eu tellus sit amet tortor gravida placerat. Integer sapien est, iaculis in, pretium quis, viverra ac, nunc. Praesent eget sem vel leo ultrices bibendum. Aenean faucibus. Morbi dolor nulla, malesuada eu, pulvinar at, mollis ac, nulla. Curabitur auctor semper nulla. Donec varius orci eget risus. Duis nibh mi, congue eu, accumsan eleifend, sagittis quis, diam. Duis eget orci sit amet orci dignissim rutrum.**

I. Introduction

A. Rationale

Much of the Reinforcement Learning **(RL)** literature is focused around specific best case applications to specific problem domains. This is an improvement over past trends described in Kaelbling's survey of the subject from 1996 Kaelbling, Littman, and Moore 1996 where research was being done on poorly defined problem domains. More recently RL research has become more focused into complex and specific problem domains. However there is little research comparing the various RL methods to one another in a comprehensive manner. Often with brief discussions of the various methods without evidence validating the chosen RL methodology. More often than not the choice of method comes down to examination of the problem domain to be explored.

The focus of this project is in comparing the performance of the more popular Q-Learning **(QL)** control policy (Watkins and Dayan 1992) against the lesser known State-Action-Reward-State-Action **(SARSA)** control policy (Rummery and Niranjan 1994) in as comprehensive a manner as possible. With the aim of understanding how applicable these methods are to various problem domains and situations. In theory SARSA should always outperform QL since it always evaluates it's chosen action (Sutton and A G. Barto 2018). And is the more computationally expensive method as a result.

Q-Learning and SARSA are very closely related temporal difference **(TD)** problems and so many optimizations of one are applicable to the other. This makes them simple to compare against one another as well as understand the differences in reinforcement learning environments. However QL overshadows SARSA in the literature quite dramatically, to the point SARSA is considerably harder to research than QL in the academic literature, however I have been unable to find an obvious reason why.

B. Aims and Objectives

The aim of this project is to understand the differences in performance of QL and SARSA and by extension offline and online control policies for TD problems. With focus being drawn to how the environments used for training affect the performance of the various control policies.

Further research will be conducted to understand the scope of TD problem domains as well as explore potential optimisations of QL and SARSA. This will allow the analysis to be more complete and present additional opportunities for comparison between the control policies.

The reason for the prevalence of QL over SARSA in the academic literature should be explored. Experimental data will be considered alongside the literature to inform the evaluation of these control policies for TD problems.

C. Hypothesis

D. Background

1) Reinforcement Learning: RL is distinct from most Artificial Intelligence **(AI)** disciplines in that instead of relying on a large pool of ground truth examples for the learning to be done upon. Instead RL relies on the design of the learning environment to teach an AI how to perform a task. In most cases this is done through reward functions with the RL agent inferring from these rewards how the task should be performed (Kaelbling, Littman, and Moore 1996).

This however relies on the designed learning environment being simple enough for an agent to learn from it's interactions with the environment alone. This leads to the main challenge in RL, exploration vs exploitation. This describes the balance between the agent exploring the environment to update all actions with their corresponding rewards and exploiting the already learned actions, taking the optimal action. This is important as it not only decides how the agent learns the environment but also its ability to interact with the environment effectively.

Since if the agent always takes random actions to explore the environment fully it will never learn the optimal strategy. Likewise if the agent only follows the optimal strategy it will not learn better strategies if they exist. This can drastically affect the time it takes to train the agent since being too close to one extreme or the other causes the agent to either never learn the optimal strategy or learn an incorrect optimal strategy by never encountering the optimal strategy (Sutton and A G. Barto 2018).

2) Q-Learning: Q-Learning was initially proposed by Watkins in their Phd thesis Watkins 1989 building on MDPs to solve more complex problems without the difficulty of building an equally complex model to solve it. Q-Learning was formalised 3 years later in Watkins and Dayan 1992.

"It provides agents with the capability of learning to act optimally in Markovian domains by experiencing the consequences of actions, without requiring them to build maps of the domains." (ibid., p. 55)

Q-Learning simplifies the environment into a Q-table with the dimensions

$$[number\ of\ states,\ number\ of\ actions]$$

Each point in the table corresponds to a Q-value for an action at the given state. This table allows for the agent to learn from the experience of each state and action. By following the maximum Q-value the optimal path should be followed given that all Q-values are representative of the environment. However, the discrete nature of the Q-table means that QL is limited to discrete problems or the environment must be discretized leading to a loss in state and action accuracy (Gaskett, Wettergreen, and Zelinsky 1999).

3) SARSA:

4) Temporal Difference Learning:

E. Report Structure

Contents

I Introduction 1
- I-A Rationale 1
- I-B Aims and Objectives 1
- I-C Hypothesis 1
- I-D Background 1
  - I-D1 Reinforcement Learning 1
  - I-D2 Q-Learning 2
  - I-D3 SARSA 2
  - I-D4 Temporal Difference Learning 2
- I-E Report Structure 2

II Literature Review 2
- II-A Academic basis 2
- II-B Scope of reinforcement learning 3
- II-C Reinforcement learning techniques 3

III Methodology 4

IV Design and Development 4

V Experiments and Evaluation 4

VI Discussions and Reflective Analysis 4

VII Conclusion 4

References 4

Appendix 5

Acknowledgements 6

II. Literature Review

A. Academic basis

The standard RL model consists of an agent interacting with an environment via perception and action. At each step of the training process the agent receives some state information. Based on this information the agent will then take an action based on it's observations of the environment. Depending on the action taken a scalar reinforcement signal is sent to the agent as a reflection of success. The agent aims to maximise the long-run sum of the reinforcement signal. This process leads to the agent choosing the rewarded actions leading to the development of the desired behaviour in the agent. As outlined in Kaelbling, Littman, and Moore 1996, a survey of RL field from 1996. This basic outline is corroborated by Busoniu, Babuska, and De Schutter 2008, a survey of multiagent RL from 2008 and Kober, Bagnell, and Peters 2013, a survey of RL in robotics from 2013. This basic conception of RL models is consistent across all literature and appears in papers up to the current day.

This learning method does not fit neatly into the two main machine learning paradigms of supervised and unsupervised learning. Instead of being directly trained by correct examples or inferring patterns from the data, a RL agent learns from it's own interactions with the environment quantified by a scalar reward described in the book Reinforcement Learning: An Introduction Sutton and A G. Barto 2018, cited in many RL works. This agent driven learning style leads to a trade-off between exploration and exploitation (Sutton and A G. Barto 2018; Kaelbling, Littman, and Moore 1996; Busoniu, Babuska, and De Schutter 2008; Kober, Bagnell, and Peters 2013). Due to the need for the agent to explore the environment as well as perform actions to maximise the positive reward (exploitation). The trade-off stems from the incompatibility of these two behaviours which are both required for an effective agent. Since the agent must explore the environment to understand the positive and negative options possible yet avoid negative actions to succeed (Kaelbling, Littman, and Moore 1996). For most RL problems Markov Decision Processes (MDP) are applied as the methodology for modelling decisions to calculate action rewards accounting for the exploration-exploitation trade-off (Kober, Bagnell, and Peters 2013). MDPs work by considering rewards temporally, meaning that a series of actions can be attributed to the delayed reward function (Kaelbling, Littman, and Moore 1996; Busoniu, Babuska, and De Schutter 2008; Kober, Bagnell, and Peters 2013). This is in contrast to more simple greedy strategies which reward the agent directly for each individual action. However these methods can fall victim to unlucky reward sampling early in the training process (Kaelbling, Littman, and Moore 1996). A useful framing problem for understanding the balancing of exploration and exploitation is the K-Armed Bandit problem. This is used in most survey papers (Kaelbling, Littman, and Moore 1996; Busoniu, Babuska, and De Schutter 2008; Kober, Bagnell, and Peters 2013) to explore the performance of RL methods in the scope of exploration vs exploitation. The more recent papers Busoniu, Babuska, and De Schutter 2008; Kober, Bagnell, and Peters 2013 analysing these methods to balance exploration and exploitation tend towards the more dynamic methods like MDPs as opposed to the more simplistic greedy strategies mentioned in Kaelbling, Littman, and Moore 1996. The other simplistic methods suggested by ibid. are not mentioned in the more recent surveys of RL Busoniu, Babuska, and De Schutter 2008; Kober, Bagnell, and Peters 2013, only in Sutton and A G. Barto 2018 are these methods mentioned as introductory methods to RL. The literature seems to be in agreement on many of the details of RL methods, however many methods mentioned in the older Kaelbling, Littman, and Moore 1996 seem to be less widely used in the surveys written in recent years (Busoniu, Babuska, and De Schutter 2008; Kober, Bagnell, and Peters 2013).

Most of the methods discussed in RL are very old, for example MDPs were originally proposed by Bellman 1958 in 1958 which is cited in each of the RL surveys Kaelbling, Littman, and Moore 1996; Busoniu, Babuska, and De Schutter 2008; Kober, Bagnell, and Peters 2013 as well as the seminal Sutton and A G. Barto 2018. The vast majority of improvements made to RL methods are minor improvements to MDPs.

B. *Scope of reinforcement learning*

Reinforcement learning is a very adaptable machine learning approach due to the wide range of approaches and methodologies that can be used to form a solution. However Kaelbling, Littman, and Moore 1996 concludes that RL performs well on small problems but poorly on larger problems due to the efficiency of generalising the problems. This point is reaffirmed by Wirth et al. 2017, adding that an efficient generalisation of a complex problem requires expert domain knowledge. This suggests that whilst versatile, RL is only generally suited to simple problems.

One of the primary areas of RL research is robotics since the domain of controlling a robot fits neatly into a RL problem as discussed in Kober, Bagnell, and Peters 2013, whereas other machine learning disciplines are very difficult to utilise for robotic control systems. This view of RL in robotics is shared by Smart and Pack Kaelbling 2002. Many other hurdles are introduced by using reinforcement learning in a real context such as observation inaccuracies and the expense of hardware Kober, Bagnell, and Peters 2013, however this does not pertain to this project. Robotic RL agents instead present a strong base of real agent-environment interactions (ibid.).

Up till this point single-agent systems have been discussed, however Multi-Agent Reinforcement Learning (MARL) is another large body of work with multiple agents training in the same environment. This domain is similar to single-agent RL but multiple agents open up a new set of challenges and advantages. As described in Busoniu, Babuska, and De Schutter 2008 the main challenge faced is an exacerbation of the exploration-exploitation trade-off due to inter-agent interactions adding a new level of complexity to the issue. Multiple agents can also be turned into an advantage in MARL by allowing agents to share their experience with other agents. Additionally more advanced reward functions can encourage collaborative actions between agents. One clear advantage over other RL domains is that MARL can benefit from parallel computing increasing the computational efficiency.

Most RL environments could be described as games, unsurprisingly the most active domain of RL is playing video games. Video games present a host of environments easily adaptable to RL applications, however the low complexity tolerance of RL still takes effect. This leads to either very simplistic games being used to train complete agents to perform well such as in Bellemare, Veness, and Bowling 2012 or more high level applications like Amato and Shani 2010 where a RL agent controls the strategies of an artificial intelligence agent designed to play the game. These applications both use MDPs with Q-Leaning (Watkins and Dayan 1992) to consider the higher level temporal planning required for playing video games.

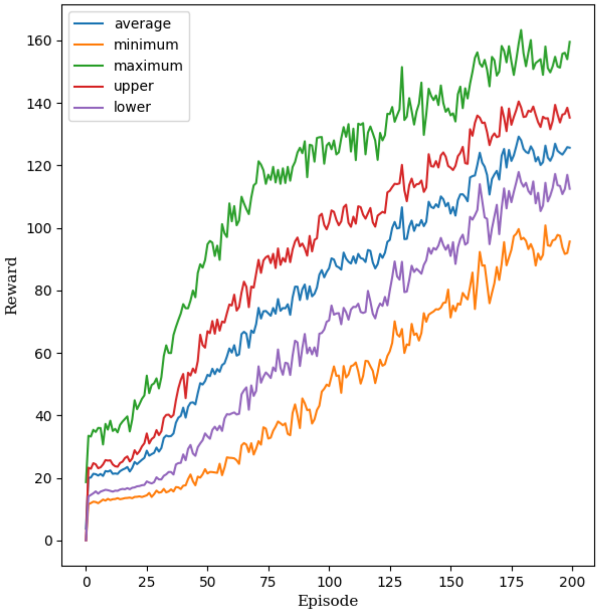
A criticism is raised in Shoham, Powers, and Grenager 2003, a critical survey of MARL using MDPs, as to the definition of many RL solutions in that they are often performed from an ill defined domain basis in how learning best occurs. It outlines a four stage method of properly defining a RL problem, describing how too many RL solutions work off previous bases without exploring their applicability to the problem being addressed. This demonstrated by the contrast between Amato and Shani 2010 where little academically supported domain knowledge is presented versus Ng et al. 2006 where the domain knowledge is comprehensively explored before designing the RL solution. This suggests that certain areas of RL problems are ill guided.

C. *Reinforcement learning techniques*

Whilst not encompassing all of RL, MDPs represent the vast majority of modern machine learning solutions. This is due to the simplicity with which a complex problem can be represented, taking into account time and previous actions as outlined in Andrew G. Barto and Mahadevan 2003. As a result of this focus there is a wealth of different MDP strategies in the literature to draw from. An MDP is a framework for modelling actions with controllable yet undetermined outcomes to be optimised Bellman 1958. Utilising an MDP a complex problem can be abstracted down to a comparably computationally simple problem, this however requires much domain specific knowledge to be properly exercised Shoham, Powers, and Grenager 2003.

MDPs are a mathematical framework to quantify and rationalise decisions made in a Stochastic Game (SG) Bellman 1958. SGs are step by step games which quantify an environment probabilistically Shapley 1953, this hugely simplifies reward and action computation in RL problems Busoniu, Babuska, and De Schutter 2008.

Fig. 1. oho it's a huge graph



In both Bellemare, Veness, and Bowling 2012; Amato and Shani 2010 Q-Learning is used, it is an approach to MDPs which allows a RL agent to explore an environment to experience the various reward states associated with each action. The Q-Learning task will use these immediate rewards to plan temporally how to maximise the reward overall as described in Watkins and Dayan 1992.

Much of the research pertaining to RL problems seems much too condensed, with single authors appearing in multiple papers such as Barto in Andrew G. Barto and Mahadevan 2003 as well as the book Sutton and A G. Barto 2018. Kaelbling also appears in both Kaelbling, Littman, and Moore 1996; Smart and Pack Kaelbling 2002. Other concentrations like this appear all across the RL literature which makes me sceptical of the academic basis of many of these methodologies.

III. Methodology

IV. Design and Development

V. Experiments and Evaluation

VI. Discussions and Reflective Analysis

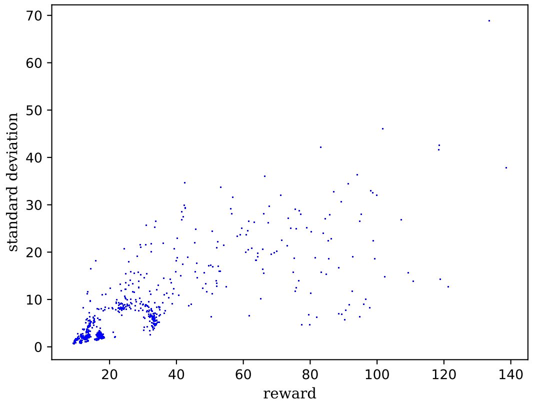
VII. Conclusion

References

Amato, Christopher and Guy Shani (2010). “High-level Reinforcement Learning in Strategy Games”. In: *Proceedings of the 9th International Conference on Autonomous Agents and Multiagent Systems*. Vol. 1. Toronto, Canada, pp. 75–82. ISBN: 978-0-9826571-1-9.

Barto, Andrew G. and Sridhar Mahadevan (2003). In: *Discrete Event Dynamic Systems* 13.4, pp. 341–379. DOI: 10.1023/a:1025696116075.

Fig. 2. this is much more manageable



Bellemare, Marc G., Joel Veness, and Michael H. Bowling (2012). “Investigating Contingency Awareness Using Atari 2600 Games”. In: *AAAI*.

Bellman, Richard (1958). “Dynamic programming and stochastic control processes”. In: *Information and Control* 1.3, pp. 228–239. DOI: 10.1016/s0019-9958(58)80003-0.

Busoniu, L., R. Babuska, and B. De Schutter (2008). “A Comprehensive Survey of Multiagent Reinforcement Learning”. In: *IEEE Transactions on Systems, Man, and Cybernetics, Part C (Applications and Reviews)* 38.2, pp. 156–172. DOI: 10.1109/tsmcc.2007.913919.

Gaskett, Chris, David Wettergreen, and Alexander Zelinsky (1999). “Q-Learning in Continuous State and Action Spaces”. In: *Proceedings of the 12th Australian Joint Conference on Artificial Intelligence: Advanced Topics in Artificial Intelligence*. AI ’99. London, UK, UK: Springer-Verlag, pp. 417–428. ISBN: 3-540-66822-5. URL: http://dl.acm.org/citation.cfm?id=646077.678231.

Kaelbling, L. P., M. L. Littman, and A. W. Moore (1996). “Reinforcement Learning: A Survey”. In: *Journal of Artificial Intelligence Research* 4, pp. 237–285. DOI: 10.1613/jair.301.

Kober, Jens, J. Andrew Bagnell, and Jan Peters (2013). “Reinforcement learning in robotics: A survey”. In: *The International Journal of Robotics Research* 32.11, pp. 1238–1274. DOI: 10.1177/0278364913495721.

Ng, Andrew Y. et al. (2006). “Autonomous Inverted Helicopter Flight via Reinforcement Learning”. In: *Experimental Robotics IX*. Ed. by Marcelo H. Ang and Oussama Khatib. Berlin, Heidelberg: Springer Berlin Heidelberg, pp. 363–372. DOI: 10.1007/11552246_35.

Rummery, Gavin Adrian and M. Niranjan (1994). “On-line Q-learning using connectionist systems”. In:

Shapley, L. S. (1953). “Stochastic Games”. In: *Proceedings of the National Academy of Sciences* 39.10, pp. 1095–1100. DOI: 10.1073/pnas.39.10.1095.

Shoham, Yoav, Rob Powers, and Trond Grenager (June 2003). “Multi-Agent Reinforcement Learning: a critical survey”. In:

Smart, W.D. and L. Pack Kaelbling (2002). “Effective reinforcement learning for mobile robots”. In: *Proceedings 2002 IEEE International Conference on Robotics and Automation*. DOI: 10.1109/robot.2002.1014237.

Sutton, R S. and A G. Barto (2018). *Reinforcement Learning: An Introduction*. Second edition. MIT Press.

Watkins, Christopher (1989). “Learning from Delayed Rewards”. PhD thesis. Cambridge, UK: King’s College. URL: https://www.cs.rhul.ac.uk/home/chrisw/thesis.html.

Watkins, Christopher and Peter Dayan (1992). “Q-learning”. In: *Machine Learning* 8.3, pp. 279–292. DOI: 10.1007/BF00992698.

Wirth, Christian et al. (2017). “A Survey of Preference-Based Reinforcement Learning Methods”. In: *Journal of Maching Learning Research* 18, 136:1–136:46.

Appendix

```
import gym
import math
import numpy as np

# Q-learning class to train and test q table for given environment
class SinKew:
    def __init__(self, init, pol, env, cOS, cAS, dis, maxS, nTst, log, ver,
            rTst, rTrn):

        # Set poliy bools for control of Q-learning
        if pol == 'q_lrn':
            self.polQ = True
            self.polS = False
        elif pol == 'sarsa':
            self.polQ = False
            self.polS = True
        else: print('Not a valid control policy')

        # Set log flag for training
        if log:
            self.log = True
        else:
            self.log = False

        # Set constant flags and values for the Q-learning object
        self.initialisation = init
        self.environment = env
        self.cont_os = cOS
        self.cont_as = cAS
        self.dis = dis
        self.maxSteps = maxS
        self.nTests = nTst
        self.logFlag = log
        self.verboseFlag = ver
        self.renderTest = rTst
        self.renderTrain = rTrn

    # Initialize environment and Q-table
    def init_env(self, resolution):

        # Create numpy array to store rewards for use in statistical tracking
        self.timestep_reward_res = np.zeros(resolution)
        self.resolution = resolution
        self.res = 0

        # Initialize environment
        self.env = gym.make(self.environment).env
        self.env.reset()

        # If observation space is continuous do calculations to create
        #   corresponding bins for use with Q table
        if self.cont_os:
            self.os_high = self.env.observation_space.high
            self.os_low = self.env.observation_space.low

            # Set bounds for infinite observation spaces in 'CartPole-v1'
            if self.environment == 'CartPole-v1':
                self.os_high[1], self.os_high[3] = 5, 5
                self.os_low[1], self.os_low[3] = -5, -5

            # Discretize the observation space
            self.discrete_os_size = [self.dis] * len(self.os_high)
            self.discrete_os_win_size = (self.os_high\
                  - self.os_low) / self.discrete_os_size
        # Use number of observations if no discretization is required
        else: self.discrete_os_size = [self.env.observation_space.n]

        # The same for action space
        if self.cont_as:
            self.dis_centre = self.dis / 2

            self.as_high = self.env.action_space.high
            self.as_low = self.env.action_space.low

            self.discrete_as_size = [self.dis] * len(self.as_high)
            self.discrete_as_win_size = (self.as_high\
                  - self.as_low) / self.discrete_as_size
            self.action_n = self.dis
        else:
            self.discrete_as_size = [self.env.action_space.n]
            self.action_n = self.env.action_space.n

        # Initialise q-table with supplied type
        if self.initialisation == 'uniform':
            self.Q = np.random.uniform(low = 0, high = 2, size=(
                self.discrete_os_size + self.discrete_as_size))
        elif self.initialisation == 'zeros':
            self.Q = np.zeros((self.discrete_os_size +
                self.discrete_as_size))
        elif self.initialisation == 'ones':
            self.Q = np.ones((self.discrete_os_size +
                self.discrete_as_size))
        else: print('initialisation method not valid')

        return

    # Get the discrete state from the state supplied by the environment
    def get_discrete_state(self, state):

        discrete_state = ((state - self.os_low) /\
                self.discrete_os_win_size) - 0.5

        return tuple(discrete_state.astype(np.int))

    # Get the continuous action from the discrete action supplied by e-greedy
    def get_continuous_action(self, discrete_action):

        continuous_action = (discrete_action - self.dis_centre) *\
                self.discrete_as_win_size

        return continuous_action

    # e-Greedy algorithm for action selection from the q table by state with
    #   flag to force greedy method for testing. Takes input for decaying
    #   epsilon value. Gets the continuous action if needed
    def e_greedy(self, epsilon, s, greedy=False):

        if greedy or np.random.rand() > epsilon: d_a = np.argmax(self.Q[s])
        else: d_a = np.random.randint(0, self.action_n)

        if self.cont_as: a = self.get_continuous_action(d_a)
        else: a = d_a

        return a, d_a

    # Perform training on the Q table for the given environment, called once per
    #   episode taking variables to control the training process
    def lrn(self, epsilon, episode, penalty, exponent, length, alpha, gamma):

        # Set vars used for checks in training
        steps = 0
        maxS = False
        done = False
        render = False

        # Reset environment for new episode and get initial discretized state
        if self.cont_os: d_s = self.get_discrete_state(self.env.reset())
        else:
            s = self.env.reset()
            d_s = s

        # Create numpy arrays and set flag for log mode
        if self.log:
            modeL = True
            history_o = np.zeros((length, len(d_s)))
            history_a = np.zeros(length)
        else:
            modeL = False

        # Report episode and epsilon and set the episode to be rendered
        #    if the resolution is reached
        if episode % self.resolution == 0 and episode != 0:
            if self.verboseFlag: print(episode, epsilon)
            if self.renderTrain: render = True

        # Create values for recording rewards and task completion
        total_reward = 0

        # Get initial action using e-Greedy method for SARSA policy
        if self.polS: a, d_a = self.e_greedy(epsilon, d_s)

        # Loop the task until task is completed or max steps are reached
        while not done:
            if render: self.env.render()

            # Get initial action using e-Greedy method for Q-Lrn policy
            if self.polQ: a, d_a = self.e_greedy(epsilon, d_s)

            # Get next state from the chosen action and record reward
            s_, reward, done, info = self.env.step(a)
            total_reward += reward

            # Discretise state if observation space is continuous
            if self.cont_os: d_s_ = self.get_discrete_state(s_)
            else: d_s_ = s_

            # If max steps have been exceeded set episode to complete
            if maxS: done = True

            # If the task is not completed update Q by max future Q-values
            if not done:
                if self.polQ:
                    max_future_q = np.max(self.Q[d_s_])

                    # Update Q-value with Bellman Equation
                    self.Q[d_s + (d_a, )] = self.Q[d_s + (d_a,)]\
                            + alpha * (reward + gamma *\
                            max_future_q - self.Q[d_s + (d_a,)])

                # Select next action based on next discretized state using
                #    e-Greedy method for SARSA policy
                if self.polS:
                    a_, d_a_ = self.e_greedy(epsilon, d_s)

                    # Update Q-value with Bellman Equation
                    self.Q[d_s + (d_a, )] = self.Q[d_s + (d_a,)]\
                            + alpha * (reward + gamma *\
                            self.Q[d_s_ + (d_a_,)] - self.Q[d_s + (d_a,)])

            # If task is completed set Q-value to zero so no penalty is applied
            if done:
                # If max steps have been reached do not apply penalty
                if maxS:
                    pass
                # Apply normal penalty to the current q value(q_lrn)
                elif self.polQ: self.Q[d_s + (d_a, )] = penalty
                elif self.polS:
                    # Update Q-value with Bellman Equation with next SA value
                    #    as 0 when the next state is terminal
                    self.Q[d_s + (d_a, )] = self.Q[d_s + (d_a,)]\
                            + alpha * (reward + gamma *\
                            penalty - self.Q[d_s + (d_a,)])
                # If log penalties are used apply penalty respective to the
                #    exponent of the relative position 1 to 10
                elif modeL and steps > length and epsilon == 0:
                    for i in range(length):
                        self.Q[tuple(history_o[i].astype(np.int)) +\
                                (int(history_a[i]), )]\
                                += penalty * math.exp(exponent) ** i

                # Iterate the resolution counter and record rewards
                if self.res == self.resolution: self.res = 0
                else:
                    self.timestep_reward_res[self.res] = total_reward
                    self.res += 1

                # Print resolution results if verbose flag is set
                if self.verboseFlag and episode % self.resolution == 0\
                        and episode != 0:
                    print(np.average(self.timestep_reward_res),
                            np.min(self.timestep_reward_res),
                            np.max(self.timestep_reward_res))

                # Close the render of the episode if rendered
                if render: self.env.close()

            # Record states and actions into rolling numpy array for applying
            #    log panalties
            if modeL and epsilon == 0:
                history_o = np.roll(history_o, 1)
                history_a = np.roll(history_a, 1)
                history_o[0] = d_s
                history_a[0] = d_a

            # Set next state to current state (Q-Learning) control policy
            if self.polQ: d_s = d_s_

            # Set next state and action to current state and action (SARSA)
            if self.polS: d_s, d_a, a = d_s_, d_a_, a_

            # If max steps are reached complete episode and set max step flag
            if steps == self.maxSteps: maxS = True
            steps += 1

        return

    # Test function to test the Q-table
    def test_qtable(self):
        # Create array to store total rewards and steps for each test
        rewards = np.zeros(self.nTests)

        # Iterate through each test
        for test in range(self.nTests):
            # Reset the environment and get the initial state
            d_s = self.get_discrete_state(self.env.reset())

            # Set step and reward counters to zero and done flag
            steps = 0
            total_reward = 0
            done = False

            # Set greedy flag sets greedy method to be used by e-Greedy
            epsilon = 0
            greedy = True

            # Loop until test conditions are met iterating the steps counter
            while not done:
                if self.renderTest: self.env.render()
                steps += 1

                # Get action by e-greedy method
                a, d_a = self.e_greedy(epsilon, d_s, greedy)

                # Get state by applying the action to the environment and
                #    add reward
                s, reward, done, info = self.env.step(a)
                total_reward += reward
                d_s = self.get_discrete_state(s)

                if steps == self.maxSteps: done = True

            # Record total rewards and steps
            rewards[test] = total_reward

        if self.renderTest: self.env.close()

        # Get averages of the steps and rewards and failure percentage for tests
        avg_rwd = np.average(rewards)
        std_rwd = np.std(rewards)

        return avg_rwd, std_rwd
```

[linewidth=5.4]

Acknowledgements