Review of Q-Learning Approaches to Sequential Decision Making

Jimmy Y. Lin

1 Sequential Decision Making

1.1 Introduction

Sequential decision making gained recognition through its uses within economics during the mid-20th century. This often involves actions which are made in stages. The total reward after the full time horizon is typically considered unknown at each stage of the decision, sometimes referred to as decision epochs. Each time we take an action, we interact with the environment, which returns us a new state.

The use of sequential decision making has expanded over the years outside beyond economic operations, for example the breakthrough development of AlphaGo (Silver et al., 2016), which uses a Monte Carlo Tree Search approach (Coulom, 2006) to sequential decision making, this was the first AI to defeat a grandmaster Go player.

1.2 Model

The most common and simple way of modelling sequential decision making is with what is known as a **Markov decision process** (MDP) (Puterman, 2014).

Markov decision processes can be described by the tuple (S, A, P, R). In this case, S denotes the state space, such that $s \in S$ denotes each state, A denotes the action space, $P(s \to s', a)$ denotes the transition probability from s to s' given action $a \in A$, and $R(s, a)$ denotes the reward associated with received after transition from s given action $a \in A$.

We can denote a policy function $\pi : S \to A$, a mapping from a state to an action. Our objective within sequential decision making is to find the optimal policy $\pi^*(s)$ made at state $s \in S$, which maximises the expected cumulative reward made within our time horizon T,

$$\pi^*(s) = \arg\max_{a \in A} \mathbb{E}\left[\sum_{t=0}^{T} \gamma^t R(s, a)\right]. \tag{1.1}$$

Here, we also introduce a discount term γ to account for present value of future rewards. For example, in economics, future rewards are worth less due to inflation and lost opportunity costs, hence $\gamma < 1$. If we value future rewards and current rewards equally, denote $\gamma = 1$.

2 Dynamic Programming Methods

2.1 Bellman's Equation

The optimal policy can be solved by using the **Bellman's equation** (Bellman, 1954). Bellman approached this problem by giving a value function for each policy, at each decision epoch such that,

$$V_\pi(s) = \mathbb{E}_{s' \sim P(s'|s,a)}\left[R(s, \pi(s)) + \gamma V_\pi(s')\right], \quad \forall s, s' \in S,\ t = 0, ..., T. \tag{2.1}$$

This equation can be solved backwards recursively for each time, state and action combination, using the boundary condition $v_{T+1}(s_{T+1}) = 0$, where no value can be gained after the final period in the time horizon. The optimal policy is therefore given by,

$$V_*(s) = \max_{a \in A} \mathbb{E}_{s' \sim P(s'|s,a)}\left[R(s, a) + \gamma V_*(s')\right], \quad \forall s, s' \in S,\ t = 0, ..., T. \tag{2.2}$$

Although Bellman's equation provides an exact solution to the problem, there are two major practical challenges with this method. The **curse of dimensionality** (Bellman, 1957), describes the issues of exponentially increasing computational costs and storage requirement of exact solutions. The **curse of modelling**

(Bertsekas, 2007), describes the necessity for accurate model for an optimal solution, whereas in reality simulations may provide a more realistic model. The two practical issues combined motivates us to find simulation algorithms which can approximate an optimal solution.

2.2 Tabular Q-Learning

One way to address these challenges is by using a technique known as Q-Learning (Murphy, 2024). Within Q-Learning, we have a function $Q(s,a)$,

$$Q_\pi(s,a) = \mathbb{E}_\pi \left[\sum_{t=0}^{\infty} \gamma^t R(s_t, a_t) \mid s_0 = s, a_0 = a \right]. \tag{2.3}$$

This describes the value of taking a specific action within a certain state. The optimal action-value function Q_* can be given by,

$$Q_*(s,a) = \mathbb{E}_{s' \sim P(s'|s,a)} \left[R(s,a) + \gamma \max_{a'} Q_*(s', a') \right]. \tag{2.4}$$

Hence the optimal policy is derived by using the optimal action-value function as,

$$\pi_*(s) = \arg\max_a Q_*(s,a). \tag{2.5}$$

The state action pair values are updated through a combination of exploration and exploitation through multiple simulations of the system. **Exploration** is necessary to learn the value of as many actions within state as possible. **Exploitation** is necessary to take the best action based on what we know as the best value function. The simplest method to balance this is what is known as ϵ-greedy, where with a set $\epsilon \in [0,1]$, we explore with probability ϵ, and we exploit with probability $1-\epsilon$.

Through each run of the simulation, termed an episode, we can update the Q-function. The idea is to use the residuals, which can be defined as the difference between the realised present value of an action taken, and the current estimated Q-function value. This is then multiplied by our learning rate which we denote as α. The Q function update is given by,

$$Q(s,a) \leftarrow Q(s,a) + \alpha \left[R(s,a) + \gamma \max_{a'} Q(s', a') - Q(s,a) \right]. \tag{2.6}$$

2.3 Double Q-Learning

Despite tabular Q-learning proven to convergence to the optimal policy with the correct setup conditions (Watkins and Dayan, 1992), there are certain issues faced by tabular Q-learning, one of which is known as positive bias (Sutton and Barto, 2018). This typically occurs when random noise makes an option seem more appealing, thus correction to the re-evaluation of the value becomes slow.

One of the proposed solution to this is Double Q-Learning (Van Hasselt, 2010). How double Q-learning works is that it has instead two tables of Q values, $Q_1(s,a)$ and $Q_2(s,a)$. Every update of Q instead takes the residual of the realised reward and present value of future rewards compared to the other Q table rather than its own. This can attempt to avoid noise from one-off high reward period. The update rule can be given as follows,

$$Q_1(s,a) \leftarrow Q_1(s,a) + \alpha \left[R(s,a) + \gamma Q_2 \left(s', \arg\max_{a'} Q_1(s', a') \right) - Q_1(s,a) \right], \tag{2.7}$$

$$Q_2(s,a) \leftarrow Q_2(s,a) + \alpha \left[R(s,a) + \gamma Q_1 \left(s', \arg\max_{a'} Q_2(s', a') \right) - Q_2(s,a) \right]. \tag{2.8}$$

Similar to tabular Q-learning, we can explore and exploit using ϵ-greedy. The choice of which Q function to update can be done randomly, with 50-50 chance that one or the other function gets updated. Double Q-Learning should be approximately the same speed as standard tabular Q-Learning. However, due to using two tables as opposed to one, it requires double the memory.

2.4 Q-Learning with Linear Function Approximation

The alternative to tabular Q-learning is find some parametric approximation to the Q-values, which we denote as $Q(s,a;\boldsymbol{w})$. We aim to minimise the loss function $\mathcal{L}(\boldsymbol{w})$, which can be defined as the mean squared error,

$$\mathcal{L}(\boldsymbol{w}) = \frac{1}{2}\mathbb{E}_{\boldsymbol{w}}\left[\left(\left(R(s,a) + \gamma \max_{a'} Q(s',a';\boldsymbol{w})\right) - Q(s,a;\boldsymbol{w})\right)^2\right]. \tag{2.9}$$

We can optimise the parameter weight value for $\boldsymbol{w}$ by using an update formulation based on stochastic gradient descent,

$$\boldsymbol{w} \leftarrow \boldsymbol{w} + \alpha \nabla_{\boldsymbol{w}} \mathcal{L}(\boldsymbol{w}). \tag{2.10}$$

The simplest case of function approximation can be a **linear** function approximation. In this case, we approximate the Q-function as $Q(s,a;\boldsymbol{w}) = \boldsymbol{w}^\intercal \boldsymbol{\phi}(s,a)) = \sum_{i=1}^{d} w_i \phi_i(s,a)$. In this case, $\phi(s,a)$ is a feature vector given state-action pair (s,a). $\boldsymbol{\phi}(s,a) = [\phi_1(s,a), ..., \phi_d(s,a)]$ where d is the number of features. The simple way to set this, and one which we implement have only features based on the dimension of the states, so $\boldsymbol{\phi}(s,a) = [s_1, ..., s_{|S|}]$. Alternatively, you can implement custom features tailored towards specific problems. Common feature selections includes (Sutton and Barto, 2018):

- **Polynomial features**: Combination of state variables as polynomials. An example with two dimensional states,
$$\boldsymbol{\phi}(s,a) = [s_1, s_2, s_1 s_2, s_1^2, s_2^2, s_1 s_2^2, s_1^2 s_2, s_1^2 s_2^2].$$
- **Fourier basis**: Models using sinusodal function, known to work well in a range of dynamic programming problems therefore useful when uncertain on choices. An example with two dimensional states,
$$\boldsymbol{\phi}(s,a) = [1, \cos(\pi s_1), \cos(\pi s_2), \cos(\pi(s_1 + s_2))].$$

Once we have chosen the desired feature vector, we substitute the linear approximation into the loss function $\mathcal{L}(\boldsymbol{w})$, we can derive the update rule as follows,

$$\boldsymbol{w} \leftarrow \boldsymbol{w} + \alpha \left[R(s,a) + \gamma \max_{a'} Q(s',a';\boldsymbol{w}) - Q(s,a;\boldsymbol{w})\right] \phi(s,a). \tag{2.11}$$

The advantage of linear function approximation approach compared to tabular is its decrease in memory requirement as it no longer stores value for each state-action pair combination but solely the values for the weights. We can also combine this with ideas from double Q-learning to further improve the learning rate (Van Hasselt et al., 2016). The disadvantages, however, are that it is not as exact as the tabular method, and also it is part of what is known as the **deadly triad** (Sutton and Barto, 2018) meaning there is possibility of instability and divergence.

3 Empirical Experiment

This section implements the methods in sections 2 to a simplified airline revenue management problem. First, we will explain the sample problem, then we follow by explaining how we chose to set up the Q-learning methods, as well as additional policy comparison to act as baselines. Finally, after running 5 instances with 5,000 episodes for each method, we will compare the results and discuss the differences in methods and what contributes to a better or worse method. The algorithms used were sourced from the *simple_rl* package in Python (Abel, 2019).

3.1 Sample Airline Management Problem

Suppose we are the manager of an airline company operating only two routes, one from Frankfurt to London, and another from London to New York. The first flight has a capacity of 40, and the second has a capacity of 20. The company takes booking requests up to $T = 100$ discrete time periods before the flight departures. The company sells tickets for a single leg, £200 for the Frankfurt to London flight (product 1), and £600 for the London to New York flight (product 2). The company also sells a ticket for both flights at a discounted price of £700 (product 3). Suppose that at each time period, there can only be one demand with probabilities $\lambda_1 = 0.2$, $\lambda_2 = 0.15$ and $\lambda_3 = 0.1$, where subscripts each product number, and the probability of no product requested $\lambda_0 = 0.55$. As a manager, with each booking request, we can either choose to accept the request, where we would receive the reward at the cost of reduced inventory, or we can reject the request, where we maintain inventory quantity but gain no reward. Now, we will define the (S, A, P, R) tuple in the MDP model.

- S: We define the state space to be of 6-dimensions, $(c_1, c_2, t, d_1, d_2, d_3)$. These are the key features identified to affect decisions. Here, c_1 is remaining capacity for leg 1, c_2 is remaining capacity for leg 2, t is time remaining until flight departures. The variables d_1, d_2 and d_3 denotes which flight is being requested at that time period, these are binary variables taking 1 if requested, and 0 if not and satisfies the inequality $d_1 + d_2 + d_3 \leq 1$.
- A: The available actions are to accept or reject, thus $a \in \{\text{accept}, \text{reject}\}$. However, note that the accept option is only available if the requested legs has available capacity. For example, if $d_1 = 1$, to accept, we require $c_1 > 1$. Likewise if $d_2 = 1$, we require $c_2 > 1$. When $d_3 = 1$, we require $c_1, c_2 > 1$. If $d_1 + d_2 + d_3 = 0$, when there is no demand at a particular time period, there is no actions to take. In this case, the transition and reward will be described in the next sections.
- P: In this model, we can write the transition function $s' = f(s, a)$ as follows,

$$s' = \begin{cases} (c_1, c_2, t-1, d_1', d_2', d_3'), & \text{if } a = \text{reject or } d_1 + d_2 + d_3 = 0, \\ (c_1 - 1, c_2, t-1, d_1', d_2', d_3'), & \text{if } a = \text{accept and } d_1 = 1, \\ (c_1, c_2 - 1, t-1, d_1', d_2', d_3'), & \text{if } a = \text{accept and } d_2 = 1, \\ (c_1 - 1, c_2 - 1, t-1, d_1', d_2', d_3'), & \text{if } a = \text{accept and } d_3 = 1, \\ \text{Terminate}, & \text{if } c_1 = c_2 = 0 \text{ or } t = 0. \end{cases}$$

  The new demand state is random based off arrival probabilities described, which is randomly generated, independent of action taken.
- R: The reward R can also be written as a piecewise function solely based off action and state. We do not need knowledge of s',

$$R(s, a) = \begin{cases} 0, & \text{if } a = \text{reject or } d_1 + d_2 + d_3 = 0, \\ 200, & \text{if } a = \text{accept and } d_1 = 1, \\ 600, & \text{if } a = \text{accept and } d_2 = 1, \\ 700, & \text{if } a = \text{accept and } d_3 = 1. \end{cases}$$

  No reward is returned upon termination of an episode. If absolutely necessary because of a coding structure, you would simply return 0.

Despite small action space, this is actually a relatively high-dimensional problem with 695,688 possible state-action pairs, some more likely to occur than others. Therefore we may be able to expect the linear function approximation to converge quicker good solution as opposed to the tabular methods.

3.2 Q-Learning Tuning Parameters

The main tuning parameters in Q-learning is the learning rate α, but as we are also implementing ϵ-greedy for exploration-exploitation, ϵ is also a tuning parameter. Watkins and Dayan (1992) states that for Q-learning to converge to the optimal solution, we must have that $\sum_{n=0}^{\infty} \alpha_n \to \infty$ and the choice of ϵ must provide sufficient exploration so that each state-action pair can be visited "infinitely" many times.

Our choice of learning rate parameter α and ϵ involves a decay function based on function from Darken and Moody (1990), adjusted for reinforcement learning (Abel, 2019),

$$\alpha(t) = \frac{\alpha_0}{1 + \frac{\text{step number}}{1000} \cdot \frac{\text{episode number}}{2000}} \text{ and } \epsilon(t) = \frac{\epsilon_0}{1 + \frac{\text{step number}}{1000} \cdot \frac{\text{episode number}}{2000}}. \tag{3.1}$$

For the initial parameters, we have set all $\epsilon_0 = 0.2$. We set Q-learning and double Q-learning initial learning rate to $\alpha_0 = 0.1$, but in linear function approximation, we set this to $\alpha = 0.01$ due to more common updates and less parameters to update. This function satisfies conditions to convergence $\sum_{n=0}^{\infty} \alpha_n \to \infty$ (Darken and Moody, 1990).

3.3 Results

First, we present a sample instance of 5000 episodes in Figure 1. We see that tabular and double Q-learning performs similarly, since the base algorithm is very similar. However, we will see later that there is slight improvement within double Q-learning as opposed to tabular. Linear Q function approximation however seems to converge far quicker, this is likely due to number of unvisited state-action pair being large for the other algorithms, leading to many randomised actions, whereas in linear function approximation estimates this based off other state-action pair characteristics. The issue with linear function approximation is its variance in the earlier episodes, this is likely since it is still calibrating its weights $\boldsymbol{w}$.

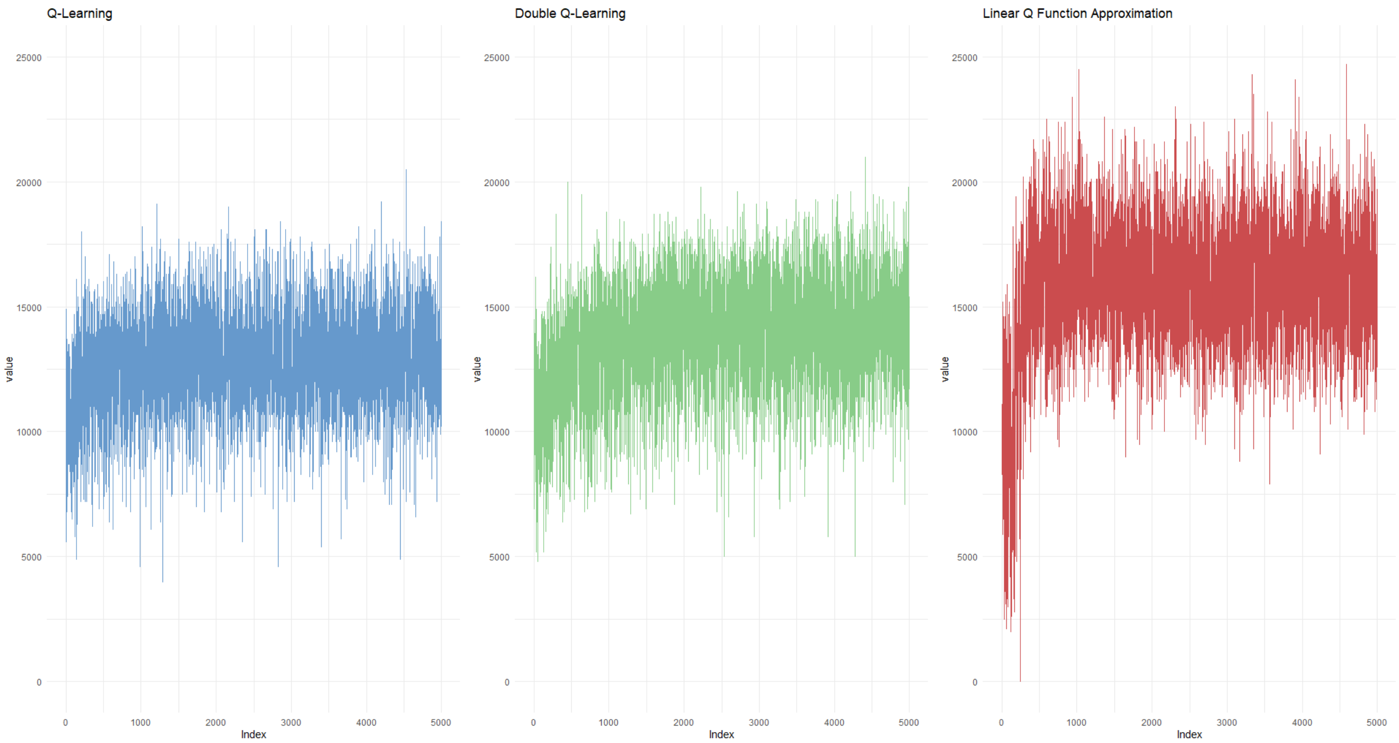

Figure 1: Episodic Reward from Sample Instance.

In Figure 2 and Figure 3, we have smoothed episodic reward (smoothed with B-spline for visualisation purposes) and mean cumulative episodic reward. We see double Q-learning was able to outperform in convergence speed compared to tabular Q-learning. Only linear function approximation was able to outperform accepter heuristic but due to poor starting performance, it was not able to outperform on cumulative rewards. This is due to the other algorithms requiring more time to converge to the optimal solution. All algorithms were able to outperform the random agent.

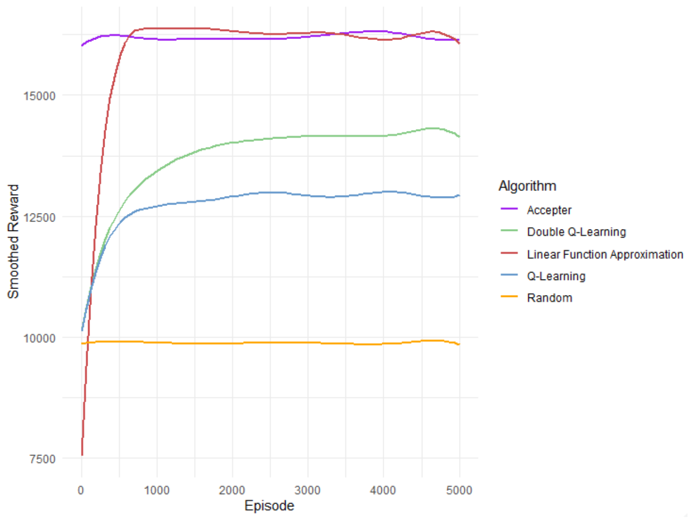

Figure 2: Smoothed Episodic Reward.

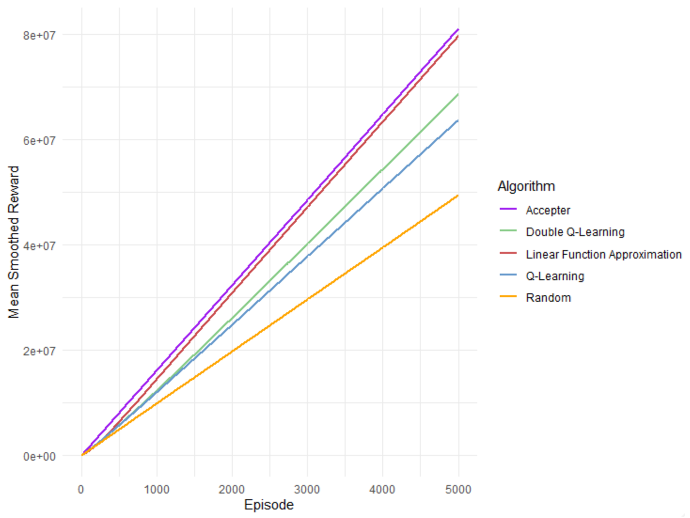

Figure 3: Mean Cumulative Episodic Reward.

4 Further Research

An interesting extension would be to compare algorithms in a different MDP problem with different characteristics, for example simpler small number of state action pair, large action pair, or a combination of these, in order to see how well our algorithms scale. Furthermore, we could also extend our current airline revenue management model by implementing our method to continuous time, but with discrete decision events. This may be simulated using a Poisson arrival process and implementing a continuous time ADP method.

In terms of methodology, there are many areas to explore for further extentions. We can look at alternative non-linear function approximations, such as neural networks (Mnih et al., 2015), or gradient boosting algorithms (Abel et al., 2016). Alternatively, we can also move away from value-based models and look at other methods such as policy-based algorithms, and how they compare to the value based models. It may be expected that these may perform better in environments with continuous action space.

We can also alter the way we decide the trade-offs between explore and exploit. Our method of ϵ-greedy is solely a simple heuristic to approach this. However, there are many different methods we are able to use for exploration-exploitation tradeoffs (Murphy, 2024), for example another heuristic commonly used is the Boltzmann exploration which assigns probability for action taken proportional to Q-values using the softmax function.

References

Abel, D. (2019). simple_rl: Reproducible Reinforcement Learning in Python. In *ICLR Workshop on Reproducibility in Machine Learning*.

Abel, D., Agarwal, A., Diaz, F., Krishnamurthy, A., and Schapire, R. E. (2016). Exploratory gradient boosting for reinforcement learning in complex domains. *arXiv preprint arXiv:1603.04119*.

Bellman, R. (1954). The theory of dynamic programming. *Bulletin of the American Mathematical Society*, 60(6):503–515.

Bellman, R. (1957). *Dynamic Programming.* Princeton University Press, Princeton, NJ.

Bertsekas, D. P. (2007). *Dynamic Programming and Optimal Control, Vol. II.* Athena Scientific, Belmont, MA, 3rd edition.

Coulom, R. (2006). Efficient selectivity and backup operators in Monte-Carlo tree search. In *International conference on computers and games*, pages 72–83. Springer.

Darken, C. and Moody, J. (1990). Note on learning rate schedules for stochastic optimization. *Advances in neural information processing systems*, 3.

Mnih, V., Kavukcuoglu, K., Silver, D., Rusu, A. A., Veness, J., Bellemare, M. G., Graves, A., Riedmiller, M., Fidjeland, A. K., Ostrovski, G., Petersen, S., Beattie, C., Sadik, A., Antonoglou, I., King, H., Kumaran, D., Wierstra, D., Legg, S., and Hassabis, D. (2015). Human-level control through deep reinforcement learning. *Nature*, 518:529–533.

Murphy, K. (2024). Reinforcement learning: An overview. *arXiv preprint arXiv:2412.05265*.

Puterman, M. L. (2014). *Markov decision processes: discrete stochastic dynamic programming.* John Wiley & Sons.

Silver, D., Huang, A., Maddison, C. J., Guez, A., Sifre, L., Van Den Driessche, G., Schrittwieser, J., Antonoglou, I., Panneershelvam, V., Lanctot, M., et al. (2016). Mastering the game of Go with deep neural networks and tree search. *nature*, 529(7587):484–489.

Sutton, R. S. and Barto, A. G. (2018). *Reinforcement Learning: An Introduction.* MIT Press, 2nd edition.

Van Hasselt, H. (2010). Double Q-learning. *Advances in neural information processing systems*, 23.

Van Hasselt, H., Guez, A., and Silver, D. (2016). Deep Reinforcement Learning with Double Q-learning. In *Proceedings of the AAAI conference on artificial intelligence*, volume 30.

Watkins, C. J. C. H. and Dayan, P. (1992). Q-learning. *Machine Learning*, 8(3-4):279–292.