Experimenting Q-Learning with Function Approximation

Syed Mohammed Umar Farooq

Department of Electrical Engineering, IIT Madras

Abstract

This report explores the application of Q-learning on a specific state and reward model using both tabular and function approximation approaches. We begin by implementing vanilla Q-learning using a constant learning rate and the Robbins-Monro algorithm. We then extend the approach to function approximation, using polynomial functions to represent the Q-values, and apply both constant and Robbins-Monro learning strategies to study their impact on learning performance.

1 Introduction

Q-learning is a widely used reinforcement learning algorithm that aims to learn the optimal action-value function, $Q(s, a)$, through interactions with an environment. In this project, we consider a specific reward model and state-transition setup and investigate the effect of different learning configurations on performance.

We begin with a baseline implementation of vanilla Q-learning, first using a constant learning rate, and then employing a Robbins-Monro learning rate schedule to allow for convergence over time. After establishing baseline behaviors, we explore Q-learning with function approximation. Specifically, we approximate the Q-function using polynomial features, allowing generalization across states. Both constant and Robbins-Monro learning rates are again tested in this setting.

Our goal is to understand the convergence behavior and learning efficiency of these methods under a controlled environment.

2 Markov Decision Process (MDP)

A Markov Decision Process (MDP) is a formal framework for modeling decision-making in situations where outcomes are partly random and partly under the control of an agent. It provides the foundation for most reinforcement learning (RL) problems.

MDPs are widely used in areas such as robotics, game theory, and economics—any domain involving sequential decisions under uncertainty. They enable agents to interact with an environment, learn from experience, and make decisions that maximize cumulative rewards.

Core Components of an MDP

An MDP consists of the following components:

- **Environment:** The external system with which the agent interacts. It defines the dynamics of the state transitions and rewards.
- **States ($\mathcal{S}$):** A set of all possible configurations in which the environment can exist.
- **Actions ($\mathcal{A}$):** The set of all possible actions the agent can take while in a given state.
- **Transition Model (P):** A probability distribution $P(s' \mid s, a)$ that describes the likelihood of reaching state s' from state s after taking action a.
- **Reward Function (R):** A mapping $R(s, a)$ that provides immediate feedback (positive or negative) to the agent for performing action a in state s.
- **Policy (π):** A strategy or mapping from states to actions, $\pi : \mathcal{S} \to \mathcal{A}$. It defines the agent's behavior.
- **Discount Factor (γ):** A scalar $\gamma \in [0, 1)$ that determines the importance of future rewards relative to immediate ones.

At each time step, the agent observes the current state, selects an action based on its policy, receives a reward, and transitions to a new state as defined by the environment. The aim is to

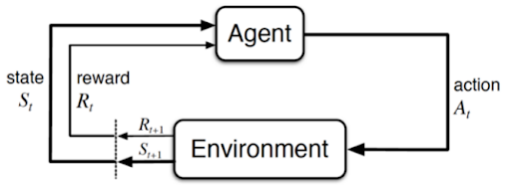

Figure 1: A typical interaction in a Markov Decision Process (MDP).

learn a policy that maximizes the expected cumulative reward over time.

This formulation lays the groundwork for learning algorithms like Q-learning, which estimates the optimal action-value function for guiding decisions.

3 Value Functions and Bellman Equations

In reinforcement learning, value functions estimate the expected return when following a policy. These functions help an agent evaluate the goodness of states and actions in a Markov Decision Process (MDP).

State-Value Function ($V^\pi(s)$)

The state-value function under a policy π is defined as the expected return starting from state s:

$$V^\pi(s) = E_\pi \left[\sum_{t=0}^{\infty} \gamma^t R_{t+1} \mid S_0 = s \right]$$

Action-Value Function ($Q^\pi(s,a)$)

The action-value function represents the expected return starting from state s, taking action a, and thereafter following policy π:

$$Q^\pi(s,a) = E_\pi \left[\sum_{t=0}^{\infty} \gamma^t R_{t+1} \mid S_0 = s, A_0 = a \right]$$

Bellman Expectation Equations

The Bellman expectation equations express value functions recursively in terms of immediate reward and the value of successor states:

For the state-value function:

$$V^\pi(s) = E_\pi \left[R_{t+1} + \gamma V^\pi(S_{t+1}) \mid S_t = s \right]$$

For the action-value function:

$$Q^*(s,a) = E\big[R_{t+1} + \gamma \max_{a'} Q^*(S_{t+1}, a') \,\big|\, S_t = s,\ A_t = a \big]$$

Bellman Optimality Equations

The optimal value functions are defined in terms of the maximum expected return obtainable from each state or state-action pair:

For the optimal state-value function:

$$V^*(s) = \max_a E \left[R_{t+1} + \gamma V^*(S_{t+1}) \mid S_t = s, A_t = a \right]$$

For the optimal action-value function:

$$Q^*(s,a) = E\left[R_{t+1} + \gamma \max_{a'} Q^*(S_{t+1}, a') \mid S_t = s, A_t = a \right]$$

The optimal policy π^* can then be derived as:

$$\pi^*(s) = \arg\max_a Q^*(s,a)$$

4 Dynamic Programming

Dynamic Programming (DP) refers to a class of algorithms that solve complex problems by breaking them down into simpler subproblems and solving them recursively. In the context of Markov Decision Processes (MDPs), DP is used to compute the optimal value functions and policies when a complete model of the environment (i.e., transition probabilities and reward function) is known.

Policy Evaluation

The goal of policy evaluation is to compute the value function $V^\pi(s)$ for a given policy π. This is done iteratively using the Bellman expectation equation:

$$V_{k+1}(s) = \sum_a \pi(a|s) \sum_{s',r} p(s', r|s, a) \left[r + \gamma V_k(s') \right]$$

Policy Improvement

Once the value function for a policy is known, we can improve the policy by acting greedily with respect to the current value function:

$$\pi'(s) = \arg\max_a \sum_{s',r} p(s', r|s, a) \left[r + \gamma V^\pi(s') \right]$$

Policy Iteration

Policy Iteration is an algorithm that alternates between policy evaluation and policy improvement until convergence to the optimal policy π^*.

Value Iteration

Value Iteration combines policy evaluation and improvement into a single update:

$$V_{k+1}(s) = \max_a \sum_{s',r} p(s', r|s, a) \left[r + \gamma V_k(s')\right]$$

This process is repeated until the value function converges to the optimal value function $V^*(s)$, from which the optimal policy can be derived.

5 Q-Learning

Q-Learning is a model-free, off-policy reinforcement learning algorithm that learns the optimal action-value function $Q^*(s, a)$ through interaction with the environment.

Objective

The Bellman optimality equation for $Q^*(s, a)$ is:

$$Q^*(s,a) = E\big[R_{t+1} + \gamma \max_{a'} Q^*(S_{t+1}, a') \,\big|\, S_t = s,\ A_t = a\big]$$

Since the model is unknown, Q-learning approximates this using the update rule [1]:

$$Q(s,a) \leftarrow Q(s,a) + \alpha\big[R + \gamma \max_{a'} Q(s',a') - Q(s,a)\big]$$

where:

- α is the learning rate,
- γ is the discount factor,
- R is the received reward,
- s' is the next state.

Exploration vs. Exploitation: ε-Greedy Policy

At each step, the agent:

- Chooses a random action with probability ε (exploration),
- Chooses $a = \arg\max_{a'} Q(s, a')$ with probability $1 - \varepsilon$ (exploitation).

Q-Learning Algorithm

1. Initialize $Q(s, a)$ arbitrarily.
2. For each episode:
   - Start in initial state s.
   - Repeat:
     (a) Choose action a using ε-greedy policy.
     (b) Observe reward R and next state s'.
     (c) Update:
     $$Q(s,a) \leftarrow Q(s,a) + \alpha\big[R + \gamma \max_{a'} Q(s',a') - Q(s,a)\big]$$
     (d) Set $s \leftarrow s'$.
     until convergence.

6 Function Approximation

- In tabular Q-learning, we maintain a separate value for every state-action pair. This becomes infeasible when the state or action space is large or continuous.
- **Function approximation** helps in generalizing across similar states or actions by learning a parameterized function to estimate the value function or Q-function.
- Instead of a table, we use a function $\hat{Q}(s, a; \boldsymbol{w})$ where $\boldsymbol{w}$ is a vector of parameters (e.g., weights in a neural network).
- According to [1] and [2], the update rule of $\boldsymbol{w}$:
  $$\boldsymbol{w} \leftarrow \boldsymbol{w} + \alpha\big(R + \gamma \max_{a'} \hat{Q}(s', a'; \boldsymbol{w}) - \hat{Q}(s, a; \boldsymbol{w})\big) \nabla_{\boldsymbol{w}} \hat{Q}(s, a; \boldsymbol{w})$$
- This approach allows learning in problems with huge or continuous state-action spaces using methods like:
  - Linear function approximation
  - Non-linear approximators (e.g., neural networks)
  - Tile coding, radial basis functions (RBF), etc.

Linear Function Approximation

In linear function approximation, the Q-value is represented as a linear combination of features:

$$\hat{Q}(s, a; \boldsymbol{w}) = \boldsymbol{w}^\top \boldsymbol{\phi}(s, a)$$

where $\boldsymbol{\phi}(s, a)$ is the feature vector representing the state-action pair and $\boldsymbol{w}$ is the weight vector.

The update rule for $\boldsymbol{w}$ is:

$$\boldsymbol{w} \leftarrow \boldsymbol{w} + \alpha\delta\nabla_{\boldsymbol{w}}\hat{Q}(s,a;\boldsymbol{w}) = \boldsymbol{w} + \alpha\delta\boldsymbol{\phi}(s,a)$$

where the temporal-difference error δ is

$$\delta = R + \gamma \max_{a'} \hat{Q}(s',a';\boldsymbol{w}) - \hat{Q}(s,a;\boldsymbol{w})$$

This allows the algorithm to generalize Q-values across similar states and actions, facilitating learning in larger or continuous spaces.

7 Experiments

In this section, we detail the empirical experiments conducted to evaluate the performance of Q-learning under stochastic transitions. All experiments were implemented in `MATLAB`.

Environment Setup

The environment is modeled with:

State Space: $\mathcal{S} = \{0, 1, \ldots, p-1\}$, where p is a natural number.

Action Space: $\mathcal{A} = \{0, 1, \ldots, p-1\}$.

Noise Space: A uniformly distributed space $\mathcal{N} = \{0, 1, \ldots, p-1\}$ with $P(n = i) = \frac{1}{p}$ for all i.(We can use any probability distribution)

The transition dynamics are governed by:

$$s_{t+1} = (s_t + a_t + n_t) \bmod p$$

where n_t is sampled from the noise space. This introduces stochasticity into an otherwise deterministic system.

All experiments were conducted using $\gamma = 0.9$ & $p = 100$, meaning that each of the state, action, and noise spaces contained 100 elements.

Reward Model

We define the reward function as:

$$R(s_t, a_t) = s_t \cdot a_t$$

This encourages the agent to select higher-value actions in higher-value states, enabling exploration of the full state-action landscape.

Vanilla Q-Learning

The standard Q-learning algorithm updates the Q-values using the rule:

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

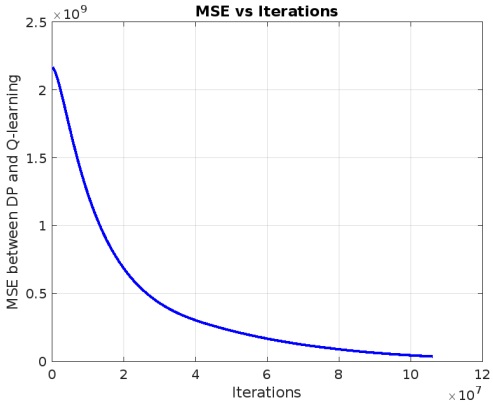

Figure 2: MSE vs Iterations graph for Q-learning with constant learning rate

Case 1: Constant Learning Rate ($\alpha = 0.001$) In our setup:

- $\alpha = 0.001$ was used as a constant learning rate.
- ϵ-greedy exploration was applied with $\epsilon = 0.5$.
- **Convergence:** Achieved after 10^8 iterations.
- **Time Taken:** 2137 seconds.
- **Accuracy:** Many Q-values were significantly different from the true values obtained via Dynamic Programming (DP).
- MSE between optimal values(calculated from DP) and Q-learning values at convergence is around 3.6×10^7. So the RMS error is around 6000.
- From Figure 2, we can infer that convergence is very slow, even after 10^8 iterations, the MSE is very high.

Q-Learning with Robbins-Monro Algorithm

Conditions for the Robbins-Monro algorithm [3]:

$$\sum_{t=1}^{\infty} \alpha_t = \infty, \quad \sum_{t=1}^{\infty} \alpha_t^2 < \infty,$$

We also evaluated a variant using a diminishing learning rate schedule inspired by the Robbins-Monro conditions. For each state-action pair (s,a), the learning rate was updated as:

$$\alpha(s,a) = \frac{\alpha_0}{1 + N(s,a)}$$

where $N(s,a)$ counts how often a state-action pair has been visited.

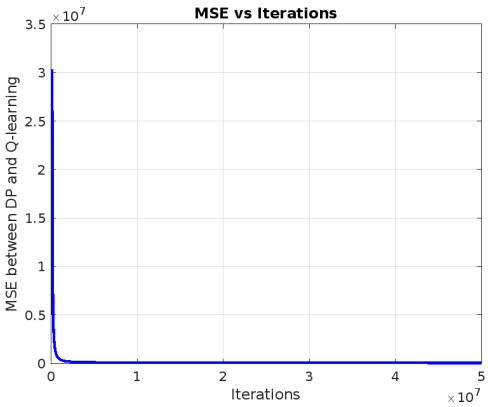

Figure 3: MSE vs Iterations graph for Q-learning with Robbins-Monro learning rate

This choice is motivated by the need to balance initial exploration with eventual convergence [1]. At the beginning of learning, higher learning rates help the agent adjust quickly to new information. As more samples are collected for a given (s, a) pair, the learning rate decreases, reducing the effect of noise and stabilizing the Q-value updates.

This learning rate satisfies the Robbins-Monro conditions.

The update rule becomes:

$$Q(s,a) \leftarrow Q(s,a) + \frac{\alpha_0}{1+N(s,a)} \left[R + \gamma \max_{a'} Q(s',a') - Q(s,a) \right]$$

This method stabilizes convergence by reducing the step size as more data is gathered, especially beneficial in noisy environments.

In our setup:

- $\alpha_0 = 1$.
- **Convergence:** Achieved after 5×10^7 iterations.
- **Time Taken:** 996 seconds.
- **Accuracy:** Most Q-values are near to those computed by DP.
- MSE between optimal values(calculated from DP) and Q-learning values at convergence is around 39611. So the RMS error is around 199.
- From Figure 3, we can infer that convergence is very fast.

Implementation Notes

The agent starts from a random initial state and explores the environment using ϵ-greedy policy. Transitions are sampled according to the uniform noise distribution. The algorithm halts when the maximum change in Q-values falls below a threshold (10^{-6}).

Results

From our experiments, the Q-learning variant using the Robbins-Monro learning rate schedule demonstrated faster convergence and required less computation time compared to the standard Q-learning algorithm with a constant learning rate. Additionally, the diminishing step size led to more stable updates in the presence of noise, resulting in Q-values that were closer to those obtained via Dynamic Programming. With increasing number of iterations the MSE can be reduced.

Q-Learning with Linear Function Approximation

To scale Q-learning to large MDPs efficiently, we implemented a linear function approximation approach:

$$\hat{Q}(s,a;w) = \phi(s,a)^T w$$

where:
$\phi(s,a)$ is the feature vector representing (s,a),
w is the weight vector of learnable parameters.

Feature Vector Construction

Before computing features, both the state and action values were normalized to lie in the interval $[0,1]$:

$$\tilde{s} = \frac{s}{p-1}, \quad \tilde{a} = \frac{a}{p-1}$$

We then constructed a feature vector using a 10th-degree polynomial(we can use polynomial of any degree except 1) basis over the normalized state-action pair $(\tilde{s}, \tilde{a})$:

$$\phi(s,a) = [1,\ \tilde{s},\ \tilde{a},\ \tilde{s}^2,\ \tilde{s}\tilde{a},\ \tilde{a}^2,\ \ldots,\ \tilde{s}^{10},\ \tilde{s}^9\tilde{a},\ \ldots,\ \tilde{a}^{10}]^T$$

This resulted in a total of 66 features, covering all monomials of total degree up to 10 in two variables.

For polynomial of degree k, the number of features are $(k+1)(k+2)/2$.

The reason for not using a polynomial of degree 1 as the feature representation stems from the structure of the action-value function as defined by the Bellman Expectation Equation:

$$Q^*(s,a) = E\left[R_{t+1} + \gamma \max_{a'} Q^*(S_{t+1}, a') \,\middle|\, S_t = s,\ A_t = a \right]$$

Table 1: Runtime, Convergence and RMS error Comparison

Algorithm	Runtime (s)	Convergence at	RMS Error
Vanilla Q-Learning	2137	1×10^8	6000
Q-Learning + Robbins-Monro	996	Not Converged	199
Function Approximation (Constant Learning rate)	1115	5×10^7	6000
Function Approximation (Robbins-Monro)	570	3×10^7	1673

In our setting, the reward is defined as $R_{t+1} = s_t.a_t$, which is deterministic and depends directly on the current state and action. Substituting this into the equation yields:

$$Q^*(s,a) = s_t.a_t + E[\gamma \max_{a'} Q^*(S_{t+1}, a') \mid S_t = s,\ A_t = a]$$

A degree-1 polynomial feature vector, $\phi(s,a) = [1,\ \tilde{s},\ \tilde{a}]$, includes only linear terms and omits the crucial interaction term $s \cdot a$. Since $Q^*(s,a)$ depends explicitly on this term, degree-1 features are insufficient to capture its structure. This motivates the use of higher-order or custom features that include $s \cdot a$.

Learning Algorithm and Update Rule

The Q-learning update was performed using stochastic gradient descent on the Bellman residual:

$$\boldsymbol{w}_{t+1} = \boldsymbol{w}_t + \alpha_t \cdot \delta_t \cdot \phi(s_t, a_t)$$

where:
$\delta_t = r_t + \gamma \max_{a'} \hat{Q}(s_{t+1}, a'; \theta_t) - \hat{Q}(s_t, a_t; \boldsymbol{w}_t)$ is the TD error,
α_t is the learning rate at time t.
We tested two learning rate schedules:
- **Constant Learning Rate:** $\alpha_t = 0.001$
- **Robbins-Monro Schedule:** $\alpha_t = \frac{1}{N(s_t,a_t)}$
The Robbins-Monro condition requires:

$$\sum_{t=1}^{\infty} \alpha_t = \infty, \quad \sum_{t=1}^{\infty} \alpha_t^2 < \infty$$

Initialization and Exploration Strategy

- Weight vector θ was initialized randomly with small values: $\theta \sim \mathcal{N}(0, 0.1^2)$ - We used ϵ-greedy exploration with $\epsilon = 0.6$. - Discount factor was set to $\gamma = 0.90$

Implementation Notes

The agent starts from a random initial state and explores the environment using ϵ-greedy policy. Transitions are sampled according to the uniform noise distribution. The algorithm halts when the maximum change in weight vector w falls below a threshold (10^{-6}).

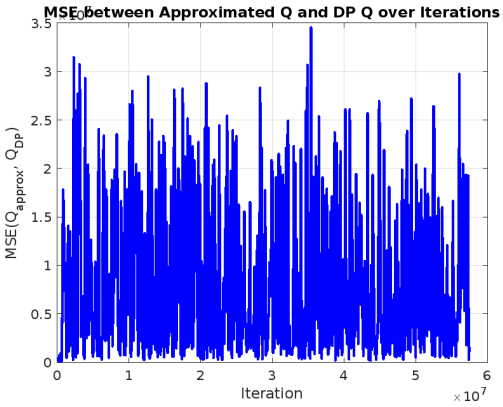

Figure 4: MSE(10^7) vs Iterations graph for Function Approximation with constant learning rate

Results with Function Approximation

We evaluated the performance of Q-learning using 10-degree polynomial function approximation (66 features per (s,a) pair).

Case 1: Constant Learning Rate ($\alpha = 0.001$)

- **Convergence:** From thresold limit, it will run around upto 5×10^7 iterations, but observing figure 4, due to the spikes in the plot, it is not converged.
- **Time Taken:** 1115 seconds.
- **Accuracy:** Learned Q-values were not close to those from DP.
- MSE between optimal values(calculated from DP) and Q-function approximation values upto 5×10^7 iterations is around 3.5×10^7, the RMS error will be around 6000.

Case 2: Robbins-Monro Learning Rate
According to [2], a condition that implies the convergence of this approximation method with probability 1 is:

$$\|\phi(s,a)\|_2 \leq 1 \quad \forall (s,a)$$

Since our initial feature vector $\phi(s,a)$ is a 10th-degree polynomial basis, we normalize it

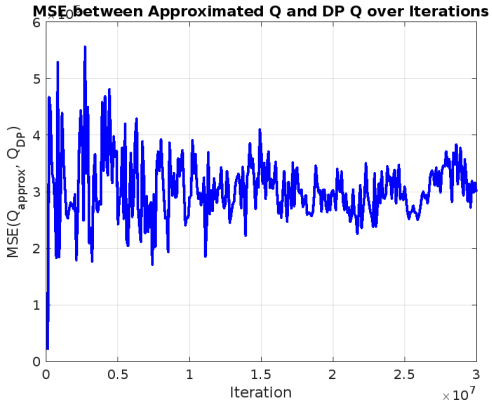

Figure 5: MSE(10^6) vs Iterations graph for Q-learning with Robbins-Monro learning rate

to satisfy this condition by defining:

$$\phi(s,a) \leftarrow \frac{\phi(s,a)}{\|\phi(s,a)\|_2} \quad \forall (s,a)$$

- **Convergence:** Achieved after 3×10^7 iterations.
- **Time Taken:** 570 seconds.
- **Accuracy:** Many Q-values were closer to DP results than with constant learning rate.
- MSE between optimal values(calculated from DP) and Q-function approximation is around 2.8×10^6, the RMS error is around 1673.
- From Figure 5, we can infer that convergence is very fast.

Observation: The Robbins-Monro adaptive learning rate scheme not only improved convergence stability but also accelerated training overall. Initially, the higher step sizes enabled the algorithm to explore and update Q-values more aggressively. As the number of visits to each state-action pair increased, the step sizes naturally decreased, leading to more stable and smoother convergence by reducing the variance introduced by noise. In contrast, function approximation with a constant learning rate lacks such adaptivity, and convergence is not guaranteed.

With Learning Rate $\alpha_t = \frac{1}{t}$

This learning rate satisfies the Robbins-Monro conditions:

$$\sum_{t=1}^{\infty} \alpha_t = \infty, \quad \sum_{t=1}^{\infty} \alpha_t^2 < \infty.$$

- Experiments show that using $\alpha_t = \frac{1}{t}$ leads to slow convergence and requires a large number of iterations for both Q-learning and function approximation.
- Since the learning rate decreases over time, state-action pairs (s,a) that are visited infrequently may not receive sufficient updates, hindering proper value learning.

Discussion

Function approximation significantly reduces memory usage by replacing the 100×100 Q-table with just 66 weights, enabling generalization across unseen state-action pairs and scaling to larger problems.

Key observations:

- The 10^{th}-degree polynomial features effectively approximated the true Q-values.
- Robbins-Monro learning takes more time to converge on increasing p or on increasing k, but it gives stable convergence than constant rates.
- Learning rates must be chosen carefully. For instance, while the Robbins-Monro condition (e.g., $\alpha_t = 1/t$) guarantees convergence, such learning rates can lead to slower convergence in practice.
- Q-leaning using Robbins-Monro algorithm can achieve faster convergence.

Conclusion

- Linear function approximation using 10-degree polynomial features offers a compact and effective means to generalize across large state-action spaces.
- The Robbins-Monro algorithm demonstrated significantly faster and more accurate convergence than the fixed learning rate method, highlighting the benefits of adaptive learning rates in stochastic environments and large-scale Markov Decision Processes.
- Function approximation remains a powerful approach for scaling reinforcement learning to large domains and is even faster than vanilla Q-learning.

References

[1] D. Katselis, "Stochastic Approximation and Robbins-Monro Algorithm," *ECE 586: Vector Space Methods*, University of Illinois

at Urbana-Champaign. [Online]. Available: https://katselis.web.engr.illinois.edu/ECE586/Lecture10.pdf

[2] A. Rajeswaran, S. Ghotra, B. Ravindran, and S. Bhatnagar, "Finite-time analysis of the stochastic approximation variant of TD learning," *IEEE Transactions on Automatic Control*, vol. 60, no. 4, pp. 1010–1015, 2015. [Online]. Available: https://ieeexplore.ieee.org/stamp/stamp.jsp?tp=&arnumber=7068926

[3] C. J. C. H. Watkins and P. Dayan, "Q-learning," in *Advances in Neural Information Processing Systems (NeurIPS)*, vol. 3, 1992. [Online]. Available: https://proceedings.neurips.cc/paper_files/paper/2001/file/6f2688a5fce7d48c8d19762b88c32c3b-Paper.pdf