AMS 691.04 PROJECT REPORT
Cart-Pole control using Q-Learning

Rohan Bansal
115060544
rohan.bansal@stonybrook.edu

I. Abstract

This project presents a comprehensive exploration and implementation of three reinforcement learning techniques: Discrete Q-Learning, Deep Q-Learning (DQN) and Double Deep Q-Learning (DDQN). At the core of this study is the goal to analyze and compare the performance, efficiency, and accuracy of these methods in an environment.

Discrete Q-learning is the traditional algorithm that works well for a discrete state-action space. DQN, an extension of traditional Q-Learning utilizing deep neural networks, offers a potent solution for handling high-dimensional input spaces. However, DQN is known to overestimate the action values. To address this, DDQN, an evolution of DQN, introduces a dual-network architecture that decouples the selection and evaluation of the action in Q-value estimation, aiming to reduce overestimations and improve learning stability.

Our methodology involves implementing these algorithms in a simple environment consisting of a continuous observation space. The expected outcomes include a detailed comparative analysis of the three algorithms, offering insights into strengths, and limitations.

II. Introduction

Machine Learning is a vast field comprising of different techniques to make a system work in a goal-oriented way. In this vast realm, reinforcement learning (RL) occupies a unique space where agents learn by interacting with their environment and receiving feedback in the form of rewards or penalties.One of the most important RL algorithms that have emerged is Q-Learning.

Q-learning [10], a model-free, off-policy RL algorithm, promises to find the optimal action-selection policy for any given finite Markov decision process, even when the model dynamics of the environment are unknown. Over the years, its convergence properties and compatibility with function approximation methods, such as neural networks, have elevated its status in various applications, from robotics to finance, and even gaming. Various adaptations of Q-Learning have been proposed in literature. We will use the most widely adaptations - Discrete Q-Learning, Deep Q-Learning and Double Deep Q-Learning. Discrete Q-Learning is used for discrete state and action space. Deep Q-Learning (DQN) [8] proposes the use Neural Networks to approximate the Q-function and thus, is suitable for dealing with a continuous state and action space. Double Deep Q-Learning (DDQN) [5] proposes an improvement for DQN which reduces the overestimation of Q-values that occurs in DQN. We will talk more about each method in detail later.

This project aims to implement these reinforcement learning algorithms to study about their working and performance. Open AI gymnasium[3] is a toolkit designed to compare RL algorithms. We choose a control environment, CartPole, which involves balancing a pole on a moving cart. It is a simple and interesting environment with a continuous state space, hence a good environment to use Neural network based approaches on. We will talk about the environment in greater detail in later sections. The goal of the project is to implement the three Q-Learning adaptations and compare their performance by running them on the CartPole environment. We shall analyse the quantitative and theoretical differences, and further talk about how other factors like environment parameters affect the performance of each adaptation. In the end we talk about improvements and future work

III. Related Work

In recent years there have been many successes of using deep representations in reinforcement learning. Still, many of these applications use conventional architectures, such as convolutional networks, LSTMs, or auto-encoders. We study two of the most commonly used algorithms : Deep Q-Learning [8] and Double Deep Q-Learning [5]. We shall discuss how Double DQN uses two DNNs and a modified update rule to solve the overestimation problem faced by DQN. There are other works as well which do not use the conventional methods.

[9] propose a dueling network separates the estimation of the state value function and the action advantage function. Essentially, it decomposes the Q-value into two parts: one that estimates the state value (how good it is to be in a given state) and another that computes the advantage of each action (how much better it is to take a specific action compared to others). [2] present a shift from the traditional approach of estimating the expected value of rewards in reinforcement learning to considering the full distribution of possible rewards. [4] introduces noisy networks as a method to drive exploration in deep reinforcement learning. Noisy networks propose a more sophisticated method where the exploration is driven by the network's intrinsic noise, leading to more efficient and intelligent exploration strategies.

[6] presents Rainbow, which is a combination of six distinct enhancements to the Deep Q-Network (DQN) algorithm into a single unified agent. These enhancements include Double

Action Space	Discrete(2)
Observation Shape	(4,)
Observation High	[4.8 inf 0.42 inf]
Observation Low	[-4.8 -inf -0.42 -inf]
Import	`gym.make("CartPole-v1")`

Fig. 1. Cartpole environment

Q-Learning, Prioritized Replay, Dueling Networks, Multi-step Learning, Distributional RL, and Noisy Nets.

IV. Discrete Q-Learning

CartPole environment can be loaded using open AI gym APIs. Figure 1 describes the CartPole environment in brief. This environment was used by Barto and Sutton in their work[1]. A pole is attached by an un-actuated joint to a cart, which moves along a frictionless track. The pendulum is placed upright on the cart and the goal is to balance the pole by applying forces in the left and right direction on the cart.

Action Space: Discrete space with size 2. 0 if the cart has to be pushed left. 1 for pushing it right.

Observation space: The observation space has 4 variables - Cart Position, Cart Velocity, Pole Angle, Pole Angular Velocity.

Reward: Since the goal is to keep the pole upright for as long as possible, a reward of +1 for every step taken, including the termination step, is allotted.

Episode End: Episode ends if Pole Angle is greater than $\pm 12^{\circ}$ or cart position is greater than ± 2.4

Discrete Q-Learning is a greedy approach where the agent takes the best next action based on state-action function values.

A. Observation space discretization

Figure 1 describes the environment parameters. The observation space has 4 variables - Cart Position, Cart Velocity, Pole Angle, Pole Angular Velocity. It is a continuous state space, so we need discretize it to use Tabular Q-learning. The lower bounds and upper bounds of these parameters are also described in Figure 1. Cart velocity and pole velocity are unbounded and thus, we need to make them bounded to discretize the whole state space. We implement discretization by dividing the bounded space of each parameters into bins of equal size. For a new observation of that parameter, We assign the index of the bin that observation falls into. In our implementation, we use the following values

Velocity lower bound : -10
Velocity upper bound : 10
Number of bins: [20,50,20,50]

B. ϵ-greedy

ϵ-greedy is an important aspect in Q-Learning which uses the exploration-exploitation trade-off. We start by exploration of the states so that the agent learns about more states and actions which benefits the agent in the long term. We start by setting ϵ = 1 and then slowly decay it to 0.25 over the episode run. In this way, the agent is able to explore enough states and also able to exploit the learned knowledge to maximise its reward. In our implementation we use the following values

ϵ = 1
ϵ decay = 0.0002
Number of episodes = 10000

For each episode, we generate a random number between 0 and 1. If the number is less than ϵ, the agent picks a random action(Exploration). Else, it picks the action that maximises the Q-value(Exploitation).

C. Q-Table

Q-table consists of all the state-action values. For our case, the state is a tuple of size 4 with a single action. Thus, the key for our Q-Table is a tuple of size of 5. We initialise the Q-Table with random values (bounded between 0 and 1). For terminal states, Q(s,a) should be zero. For this environment, terminal states are when the cart leaves the (-2.4, 2.4) range or when pole angle is not in the range (-.2095, .2095) (or ±12°).

D. Simulation

- Initialise the environment and then loop through the training episodes

  Environment = CartPole-v1
  Learning rate(α) = 0.1
  Discount factor(γ) = 0.95
- Select action based on ϵ-greedy approach. For each episode, we generate a random number between 0 and 1. If the number is less than ϵ, the agent picks a random action. Else, it picks the action that maximises the Q-value.
- For each episode, start a while loop that ends when we reach a terminal state. Apply the action choosen in previous step to the environment. We use env.step() function to apply an action to the environment. The return values consist of the next state, reward and a bool variable that tells whether we reached a terminal state or not. The next state returned needs to be discretized as well.
- Whenever we reach a terminal state and the total reward accumulated is less than a certain threshold (say 200 in our case), we assign a high negative reward(say -300 in our case) to teach the agent that a particular action leads to negative reward.
- Update the Q-Table. The action values are updated according to equation 1

$$Q(S,A) = Q(S,A) + \alpha(R + \gamma max_a Q(S',a) - Q(S,A)) \quad (1)$$

- End the while loop if the next state is terminal state
- Repeat from step 2

Figure 2 shows the average, maximum and minimum reward gained over a window of 100 episodes. We can observe that the rewards increase over episodes which means that agent improves it's policy over time. From the graph, we can observe that exploration happens over the first 5000 episodes and then the learned policy is used in the second half. For this run, the episode is not limited to 500 iterations. We let it run beyond 500 iterations to better gauge the performance.

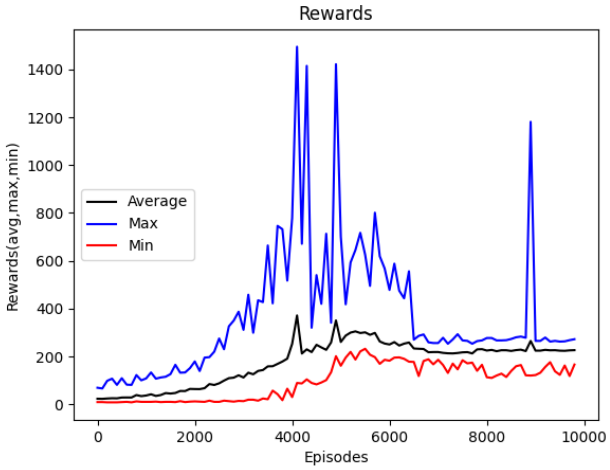

Fig. 2. Rewards (average,max,min)

E. Experimentation and Evaluation

We experiment over certain parameters to understand how they affect the learning and what values would be most optimal. We also try to reason our findings

- **Learning rate** : Learning affects how fast/quick the Q-values are updated. Figure 3 shows the trend of average rewards v/s learning rate(α). Maximum average rewards are achieved in the case of α = 0.5. The graph shows that a very small learning rate (α = 0.01) leads to a very slow learning curve. However, we see that a high learning rate(α = 1) does not lead to a steep learning curve. Infact, we can see that rewards for α = 1 are the lowest. This could be because a high learning rate may lead to the Q-values not converging quickly. Thus, an optimal learning rate is important for the agent to get to the optimal policy in minimum number of runs
- **Epsilon decay** : For implementing the ϵ-greedy approach, it is important to have an optimal value of decay rate. Too high of a decay rate would lead to less exploration and too small could lead to too many random actions. We experiment with this perimeter by running the simulation with different values. Figure 4 describes the average rewards v/s epsilon decay rate curve. We can see that a decay rate of 0.0005 is the most optimal.

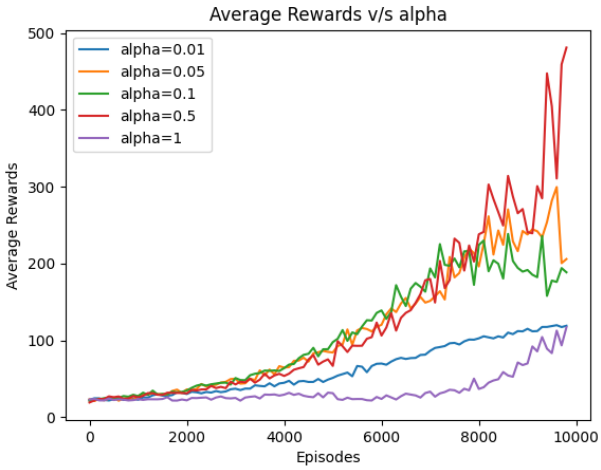

Fig. 3. Average rewards vs learning rate(α)

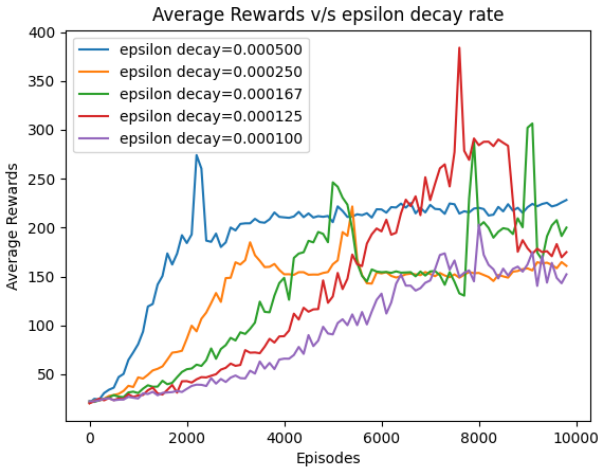

Fig. 4. Average rewards vs epsilon decay rate

V. Deep Q-Learning

In traditional Q-Learning, the Q-function is often represented in a tabular form, which becomes impractical for environments with large or continuous state spaces. Deep Q-Learning solves this by using deep neural networks to approximate the Q-function. These networks can handle complex, high-dimensional state spaces.

A. Experience Replay

Most of the mechanisms like ϵ-greedy are still applied in DQN as well. One added mechanism that DQN uses is experience replay. Instead of learning from consecutive experience samples, the agent stores these experiences in a memory buffer. It then randomly samples a batch of experiences from this buffer to learn. This approach helps in breaking the correlation between consecutive samples and stabilizes the learning process. For our experiments, we use the following values

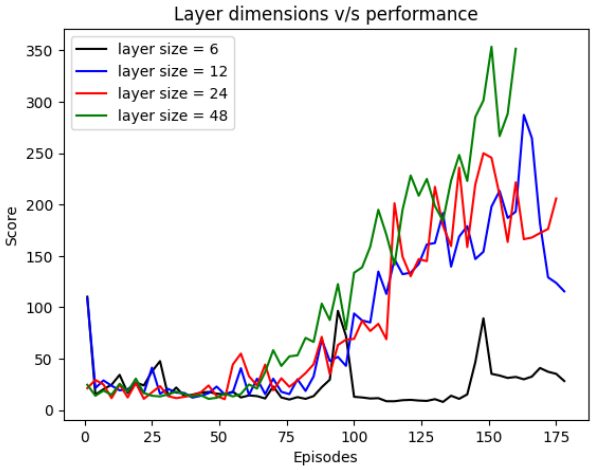

Fig. 6. Reward vs hidden layers size

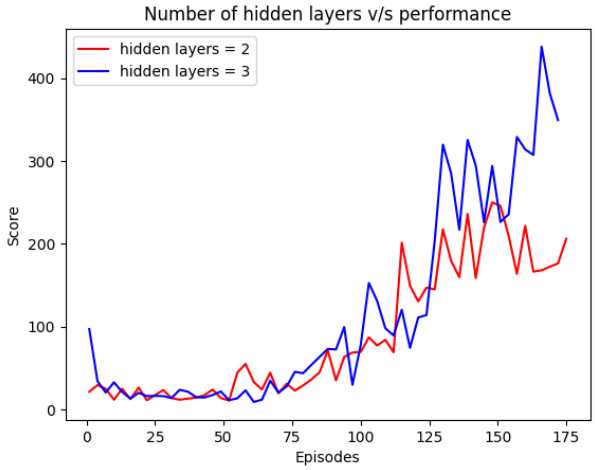

Fig. 7. Reward vs number of layers

Replay memory size = 5000
Sampled Batch size = 64

B. Learning Network

Deep Q-Learning uses deep neural networks to approximate the Q-function. We use Keras libraries [7] to implement the neural network. The input dimensions would b be the same as the observation space and the output would be the tuple of Q-value and action. Fig. 5 shows the architecture of the neural network which consists of 2 hidden layers of dimension 24 each.
Optimizer = Adam
Activation function = relu
Loss = mean square error
Learning rate = 0.001
Discount factor = 0.95
Number of episodes = 180

Layer (type)	Output Shape	Param #
dense (Dense)	(None, 24)	120
dense_1 (Dense)	(None, 24)	600
dense_2 (Dense)	(None, 2)	50

Total params: 770 (3.01 KB)
Trainable params: 770 (3.01 KB)
Non-trainable params: 0 (0.00 Byte)

Fig. 5. Learning network architecture

C. Experimentation and Evaluation

We do not repeat the same experiments conducted for discrete Q-Learning. Instead, we tune the values of hyper-parameters related to neural network and study the results. For the experiments, we keep number of episodes = 180 because training DNN is computationaly heavy and takes large amount of time without a GPU.

- **Hidden layer size** : The size of hidden layers in a neural network significantly influences its learning capabilities, affecting various aspects such as the model's capacity, generalization ability, training time, and risk of overfitting or underfitting. We run experiments by changing the dimensions of hidden layers over the values of [6,12,24,48]. Fig.6 shows the graph for Reward vs hidden layers size. We observe that for small dimensions (layer size = 6), the model does not learn enough. Small layers mean fewer neurons, and fewer neurons mean a limited ability to capture the complexity in the data, which leads to the function approximation being not accurate enough. The performance is best achieved for large hidden layer size (layer size = 48). With more parameters, the network can model more complex functions and capture intricate patterns in the data that smaller networks might miss. Using larger layers also lead improved Feature Representation and thus better function approximation
- **Number of layers :** The number of layers is an equally important parameters and affects the performance in almost a similar way as hidden layer size. We run only two experiments for this case. Fig.7 shows the graph for Reward vs number of layers. There is a stark improvement in the performance for when we increase the number of layers from 2 to 3, keeping the dimension of layers constant. The performance almost improves by a factor of 2. More layers typically mean a more complex model with a greater capacity to learn. With increased depth, a network can represent more complex functions and capture intricate patterns in data. Deeper networks also generalise better on the unseen data. This is important because it is necessary for the agent to explore as many states as possible before coming to an optimal policy. Using deeper networks can help generalising well on the states that were missed during exploration.

VI. Double Deep Q-Learning

In [5], the authors propose Double Deep Q-Learning In DQN, the same network is used to select and evaluate an action. This can lead to systematic overestimation of Q-values, as the network might preferentially select overestimated values. Double DQN separates the action selection from action evaluation. It uses two networks: one for selecting the best action (the primary network) and another for evaluating the action (the target network). So in Double DQN we have two networks - primary and target. The primary network is used for selecting actions, and the target network is used for evaluating them. The approach of Double DQN would be very similar to DQN except that there is an added Neural network. The target used by DQN is given by equation 2

$$Y_t^{DQN} = R_t + \gamma \max_a Q(S_{t+1}, a, \theta_t^-) \quad (2)$$

In equation 2, only one set of weights(θ_t^-) are used as the function approximate. This update involves taking the maximum estimated Q-value over all possible actions for the next state. This maximization step can lead to a positive bias, especially when there is noise in the Q-value estimates. Double DQN update decouples the action selection from value estimation, effectively reducing the maximization bias by having two networks (or two sets of weights) that are updated asynchronously. It is given by equation 3.

$$Y_t^{DoubleDQN} = R_t + \gamma Q(S_{t+1}, \max_a Q(S_{t+1}, a, Q_t), \theta_t^{'}) \quad (3)$$

In equation 3, two sets of weights(θ_t and $\theta_t^{'}$) are used. Notice that the selection of the action, in the argmax, is still due to the online weights θ_t. This means that, as in Q-learning, we are still estimating the value of the greedy policy according to the current values, as defined by θ_t. However, we use the second set of weights $\theta_t^{'}$ to fairly evaluate the value of this policy. This second set of weights can be updated symmetrically by switching the roles of θ_t and $\theta_t^{'}$.

A. Implementation

We make changes in the already implemented Deep Q-Learning code. Two networks are initialised instead of one with the same architecture. One network is used for action selection and the second is used to evaluate that action. The networks are updated symmetrically by switching the roles of θ_t and $\theta_t^{'}$.

B. Experimentation and Evaluation

For accurate comparison between DQN and Double DQN, we keep the hyper-parameters same in both the runs. Fig.8 shows the performance comparison of DQN against Double DQN. It can be observed that Double DQN performs better and most importantly learns much quicker than DQN. Because Double DQN does not overestimate the Q-values, it learns quicker and does not have to wait for the Q-values to get normalised.

We also tried to study the extent of overestimation that occurs in DQN. Thus, we plotted the values for DQN target

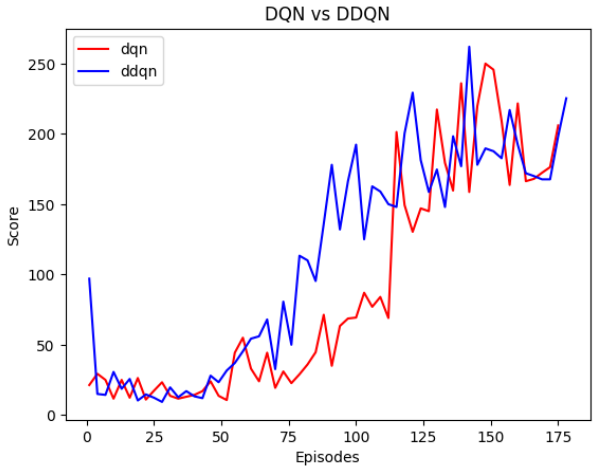

Fig. 8. DQN vs Double DQN

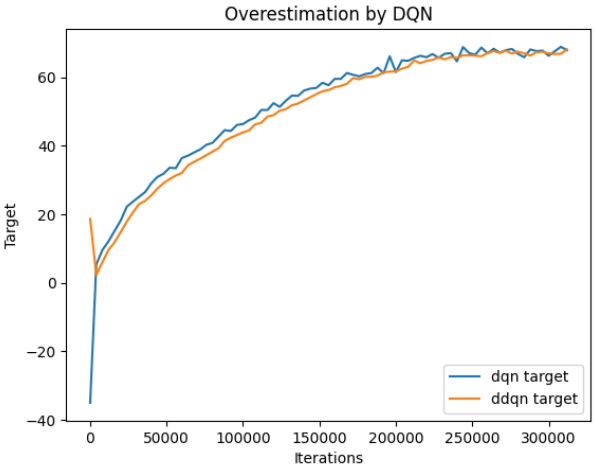

Fig. 9. DQN target vs Double DQN target

(Y_t^{DQN} in eq.2) against Double DQN target ($Y_t^{DoubleDQN}$ in eq.3). Fig.9 shows the graph for DQN target vs Double DQN target. We can observe some overestimation as the target values for DQN target are higher. The extent of overestimation might vary for different environments, depending on environment configurations. So it might be more distinct in other settings.

VII. Conclusion

In this project, we embarked on an exploratory journey into the realms of Discrete Q-Learning, Deep Q-Learning (DQN) and Double Deep Q-Learning (DDQN), three pivotal algorithms in the field of Reinforcement Learning. Our implementation and analysis of these techniques have yielded insightful results and have significantly contributed to our understanding of their practical applications and limitations.

We successfully implemented Discrete Q-Learning, a relatively simple algorithm and then extended it to implement DQN, demonstrating its capability in handling complex decision-making environments with high-dimensional inputs.

Our experiments validated the algorithm's effectiveness in learning optimal policies for an environment with a continuous state space. The implementation of DDQN marked a notable progression in our project. By addressing the overestimation bias inherent in DQN, DDQN showcased improved performance and stability.

We noticed that fine-tuning the parameters and choosing the right network architecture greatly influenced the effectiveness of both discrete Q-Learning and DQN. This highlighted the importance of a nuanced approach to configuring reinforcement learning models. The balance between exploration (trying new actions) and exploitation (using known actions) was crucial. We used strategies like epsilon-greedy and the intrinsic exploration features in DDQN to manage this balance

The computational demands, in terms of both processing power and time, were substantial. This highlighted the need for more efficient algorithms and better hardware optimization in future work.

Building upon the foundational work of this project, future endeavors could explore the integration of improvements like prioritized experience replay, dueling network architectures, or multi-step learning approaches. Applying these methods to real-world problems, such as in robotics or autonomous systems, offers exciting avenues for practical application and further research.

REFERENCES

[1] Andrew G. Barto, Richard S. Sutton, and Charles W. Anderson. "Neuronlike adaptive elements that can solve difficult learning control problems". In: *IEEE Transactions on Systems, Man, and Cybernetics* SMC-13.5 (1983), pp. 834–846. DOI: 10.1109/TSMC.1983.6313077.

[2] Marc G. Bellemare, Will Dabney, and Rémi Munos. "A Distributional Perspective on Reinforcement Learning". In: (2017). arXiv: 1707.06887 [cs.LG].

[3] Greg Brockman et al. "OpenAI Gym". In: (2016). arXiv: 1606.01540 [cs.LG].

[4] Meire Fortunato et al. "Noisy Networks for Exploration". In: (2019). arXiv: 1706.10295 [cs.LG].

[5] Hado van Hasselt, Arthur Guez, and David Silver. "Deep Reinforcement Learning with Double Q-learning". In: (2015). arXiv: 1509.06461 [cs.LG].

[6] Matteo Hessel et al. "Rainbow: Combining Improvements in Deep Reinforcement Learning". In: (2017). arXiv: 1710.02298 [cs.AI].

[7] "https://keras.io/api/models/sequential/". In: ().

[8] Volodymyr Mnih et al. "Playing Atari with Deep Reinforcement Learning". In: (2013). arXiv: 1312.5602 [cs.LG].

[9] Ziyu Wang et al. "Dueling Network Architectures for Deep Reinforcement Learning". In: (2016). arXiv: 1511.06581 [cs.LG].

[10] Christopher J. C. H. Watkins and Peter Dayan. "Q-learning". In: *Machine Learning* (1992).