Deep Q-learning (DQN) for Multi-agent Reinforcement Learning

Seyed Mohammad Asghari

1 Background

We begin with reviewing single-agent and multi-agent reinforcement learning.

1.1 Single-Agent Reinforcement Learning

In a single-agent, fully-observable, RL setting, there is an agent interacting with an environment through taking actions. At each time t, the agent observes the current state $S_t \in \mathcal{S}$ of the environment, chooses an action $U_t \in \mathcal{U}$ according to a stochastic policy π, and receives a reward signal $R_t = r(S_t, U_t)$. Then, the environment transits to a new state $S_{t+1} \in \mathcal{S}$ according to the transition probability function $\mathbb{P}(S_{t+1}|S_t, U_t)$. The objective is to find a policy π for the agent maximizing the expectation of discounted return $G_t = \sum_{k=t}^{\infty} \gamma^{k-t} R_k$ where R_k is the reward received at time k and $\gamma \in [0, 1)$ is a discount factor. The action-value function Q of a policy π is $Q(s, u) = \mathbb{E}^{\pi}[G_t|S_t = s, U_t = u]$. The optimal action value function $Q^*(s, u) = \max_{\pi} Q^{\pi}(s, u)$ obeys the Bellman optimality equation $Q^*(s, u) = r(s, u) + \gamma \sum_{s'} \mathbb{P}(s'|s, u) \max_{u'} Q^*(s', u')$. Given Q^*, the optimal policy π^* for each state $s \in \mathcal{S}$ can be recovered by selecting the action with the highest Q-function: $\pi^*(s) = \text{argmax}_u Q^*(s, u)$. This property has led to various number of learning algorithms that directly estimate Q-function such as Q-Learning Watkins (1989).

1.1.1 Q-learning

In Q-Learning, an agent begins with an arbitrary estimate (Q_0) of Q^* and iteratively improves its estimate as follows: at each time t, then agent takes an action u_t for the state s_t according to a policy, receives a reward $r_t = r(s_t, u_t)$, and observes a new state s_{t+1}. Based on the experience $< s_t, u_t, r_t, s_{t+1} >$, the agent updates the Q-function according to

$$Q_{t+1}(s_t, u_t) = Q_t(s_t, u_t) + \alpha_t[r_t + \gamma \max_{u_{t+1}} Q(s_{t+1}, u_{t+1}) - Q(s_t, u_t)], \tag{1}$$

where $\alpha_t \in [0, 1)$ is learning rate or step size which determines to what extent newly acquired information overrides old information. In (1), the term $r_t + \gamma \max_{u_{t+1}} Q(s_{t+1}, u_{t+1})$ is refereed to as *target* and the term $Q(s_t, u_t)$ is refereed to as *estimation*. A common way of deriving a new policy during the iterations is to follow ϵ-greedy strategy. Based on this strategy, at each state s_t, with probability ϵ a random action is selected and with probability $1 - \epsilon$, the action maximizing Q_t is selected. This kind of strategies are used to introduce a form of *exploration* which is an important concept in RL: by randomly selecting actions that are sub-optimal according to its current estimates, the agent can discover and correct its estimates when appropriate.

1.1.2 Deep Q-Learning (DQN)

Deep Q-Learning (DQN) Mnih et al. (2015) is a variation of the classic Q-Learning algorithm with 3 main contributions:

1. Instead of tabular Q function used in Q-learning, DQN uses multi-layer neural networks to approximate the Q function. In other words, in DQN the action-value function $Q(\cdot, \cdot)$ is approximated by $Q(\cdot, \cdot; \theta_t)$ where θ_t represents the parameters of the neural network.
2. Instead of using one Q function for calculating both target and estimation (see (1)), DQN uses two separate neural networks, a *target* neural network $\bar{Q}(\cdot, \cdot; \bar{\theta}_t)$ and an *online* neural network $Q(\cdot, \cdot; \theta)$. More specifically, for any experience $< s_t, u_t, r_t, s_{t+1} >$, DQN uses the target network to calculate the target $r_t + \gamma \max_{u_{t+1}} \bar{Q}(s_{t+1}, u_{t+1}; \bar{\theta}_t)$ and uses the online network to calculate the estimation $Q(s_t, u_t; \theta_t)$. The target and estimation values are then used to compute the following loss
$$[r_t + \gamma \max_{u_{t+1}} \bar{Q}(s_{t+1}, u_{t+1}; \bar{\theta}_t) - Q(s_t, u_t; \theta_t)]^2. \tag{2}$$
3. Instead of using the last obtained experience to update the Q function, DQN stores the experiences in a replay memory Lin (1992) and uses mini-batches of experiences taken uniformly at random to update the parameters of the online network by using stochastic gradient descent to minimize the loss in (2). The target network is not optimized directly and only periodically updated to the online network. In this way training can proceed in a more stable manner. The use of experience replay memory can alleviate two issues: i) Neural Networks are prone to "forgetting" things they have learnt; and ii) There exists a large amount of correlation between consecutive experiences. By keeping the previous experiences in a replay memory and taking samples randomly from this memory, DQN prevents the network from learning based on correlated experiences and also allows it to re-learn from the past experiences.

1.1.3 Extensions to DQN

There have been several extensions to the original DQN proposed by Mnih et al. (2015). Here, we picked a set of extensions that address distinct concerns associated with the original DQN.

Double DQN (DDQN): In the DQN described above, the target network $\bar{Q}(\cdot, \cdot; \bar{\theta}_t)$ is used both to select the best action for the next state and to calculate the target to evaluate the selected action (see 2). This makes it more likely to select overestimated values, resulting in overoptimistic value estimates. In order to prevent this issue, the authors of Double DQN in Van Hasselt et al. (2016) proposed a simple trick to decompose the action selection from the action evaluation. More specifically, in the double DQN, the online network is used to select an action and the target network is used to generate the target value for that action. Therefore, the loss in (2) can be rewritten as follow for the double DQN,
$$\big[r_t + \gamma \bar{Q}\big(s_{t+1}, \operatorname*{argmax}_{u_{t+1}} Q(s_{t+1}, u_{t+1}; \theta_t); \bar{\theta}_t\big) - Q(s_t, u_t; \theta_t)\big]^2. \tag{3}$$

It has been shown that this change can substantially reduce the overestimations that were present in DQN and hence, it improves the performance.

Prioritized experience replay: In the DQN, the samples are taken uniformly at random from the experience replay memory. However, by treating all samples the same, we are ignoring a simple intuition from the real world, that is, we can potentially learn more from the experiences for which the outcomes differ more from our expectations. To leverage this fact, prioritized experience replay (PER) Schaul et al. (2015) samples experiences with probability p_t proportional to the absolute difference between the target and estimation values for this experience which is known as *TD error*. That is,
$$p_t \propto \big(|r_t + \gamma \max_{u_{t+1}} \bar{Q}(s_{t+1}, u_{t+1}; \bar{\theta}_t) - Q(s_t, u_t; \theta_t)| + \epsilon\big)^{\omega}, \tag{4}$$

where ϵ is a small positive constant ensuring that no experience has zero probability of being sampled. Further, the exponent ω determines how much prioritization is used, with $\omega = 0$ corresponding to the uniform case. For the Atari benchmark suite Bellemare et al. (2013), the use of prioritized experience replay has led to both faster learning and to better final policy quality across most games as compared to uniform experience replay.

Dueling networks: In the DQN network architecture, although there exist two networks, which are online network and target network, for each of these networks, a single neural network is used to

estimate the action-value function Q that measures the value of choosing a particular action when in a particular state. The key insight behind the dueling network architecture Wang et al. (2015) is that for many states, it is unnecessary to estimate the value of each action choice. To this end, in this network architecture, two separate functions are estimated, a value function V that measures how good it is to be in a particular state and an advantage function A that measures the relative importance of choosing a particular action when in a particular state. Then, these two functions are aggregated to estimate the action-value function Q. For example, an aggregating module for the online network (similarly for the target network) proposed in Wang et al. (2015) is as follows,

$$Q(s_t, u_t; \theta_t, \alpha_t, \beta_t) = V(s_t; \theta_t, \beta_t) + \Big(A(s_t, u_t; \theta_t, \alpha_t) - \frac{1}{|\mathcal{U}|} \sum_{u' \in \mathcal{U}} A(s_t, u'_t; \theta_t, \alpha_t)\Big), \quad (5)$$

where θ_t represents the parameters of shared layers (e.g., the convolutional layers if they exist) between the estimators for the value function V and the advantage function A, while α_t and β_t are the parameters of separate layers for the value function V and the advantage function A, respectively.

1.2 Multi-Agent Reinforcement Learning

In a fully observable, cooperative multi-agent problem, there exist n agents where the set of agents is denoted by $\mathcal{A} = \{1, 2, \ldots, n\}$. In this problem, at each time t, each agent $a \in \mathcal{A}$ observes the current state $S_t \in \mathcal{S}$ of the environment and chooses an action $U_t^a \in \mathcal{U}^a$ according to a stochastic policy π^a. The actions of the agents form a joint action $\boldsymbol{U}_t$. As a result of this joint action, all agents receive a reward signal $R_t = r(S_t, \boldsymbol{U}_t)$ and the environment transits to a new state $S_{t+1} \in \mathcal{S}$ according to the transition probability function $\mathbb{P}(S_{t+1}|S_t, \boldsymbol{U}_t)$. The objective is to find a set of policies $\pi^a, a \in \mathcal{A}$, for all the agents maximizing the expectation of discounted return $G_t = \sum_{k=t}^{\infty} \gamma^{k-t} R_k$ where R_k is the reward received at time k and $\gamma \in [0, 1)$ is a discount factor.

One natural approach for such a fully observable, cooperative multi-agent RL problem is to consider a "meta-agent" who chooses the joint action $\boldsymbol{U}_t$ according to $\boldsymbol{\pi}$ which is a vector including policies $\pi^a, a \in \mathcal{A}$, that is, $\boldsymbol{\pi} = (\pi^1, \ldots, \pi^n)$. This meta-agent learns Q-function $Q(s, \boldsymbol{u}) = \mathbb{E}^{\boldsymbol{\pi}}[G_t|S_t = s, \boldsymbol{U}_t = \boldsymbol{u}]$ that conditions on the state and joint action of all agents. Then, all the aforementioned algorithms for a single-agent RL problem can be applied to this problem. However, the meta-agent needs to choose the joint action $\boldsymbol{U}_t$ in $\prod_{a \in \mathcal{A}} \mathcal{U}^a$ where this space grows exponentially in the number of agents. Therefore, this issue makes this approach not scalable and hence, impractical for problem with large number of agents.

Another approach which can be applied to multi-agent RL problems (not necessarily to fully observable, cooperative problems) is independent Q-learning (IQL) Tan (1993). In this approach, which is the simplest and most popular approach for multi-agent RL problems, each agent $a \in \mathcal{A}$ ignores the presence of other agents in the environment and learns its own Q-function $Q^a(s, u^a) = \mathbb{E}^{\pi^a}[G_t|S_t = s, U_t^a = u^a]$ that conditions on the state and its own action. Compared to the first approach, IQL is appealing because it avoids the scalability issue discussed above. As Q-learning can be extended to DQN for deep RL problems, IQL can be extended to independent DQN by having each agent performs DQN to train its own neural network for approximating its own Q-function.

Despite the simplicity of IQL, it suffers from one key issue: the environment appears non-stationary from the view point of any agents as it contains the other agents which are independently updating their policies as learning progresses. The non-stationarity of the environment from the view point of each agent rules out any convergence guarantees that exist for Q-learning. In spite of this issue, IQL has strong empirical results Matignon et al. (2012), however, such results do not involve deep learning.

There is one key challenge for applying IQL to deep RL problems: while the existence of the experience replay memory helps to stabilize the training of a deep neural network and also improves sample efficiency as it can repeatedly re-learn from past experiences, its combination with IQL seems problematic. Due to the non-stationarity of the environment from the perspective of each agent introduced by IQL, the experience replay memory may not longer reflect the current dynamics of the environment the agent should learn. Therefore, reusing obsolete experiences can continuously confuse the neural network and make the training unstable.

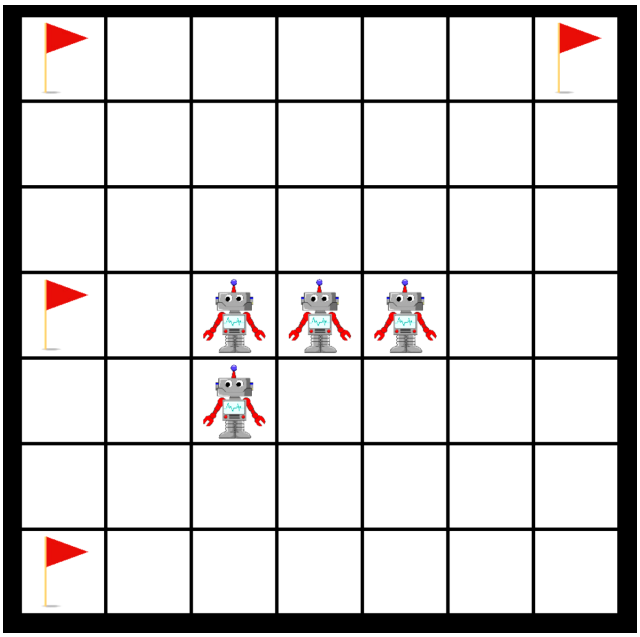


Figure 1: Agents-landmarks environment with $n = 4$ agents

2 Experiments

2.1 Environments

To perform our experiments, we have designed our own environments.

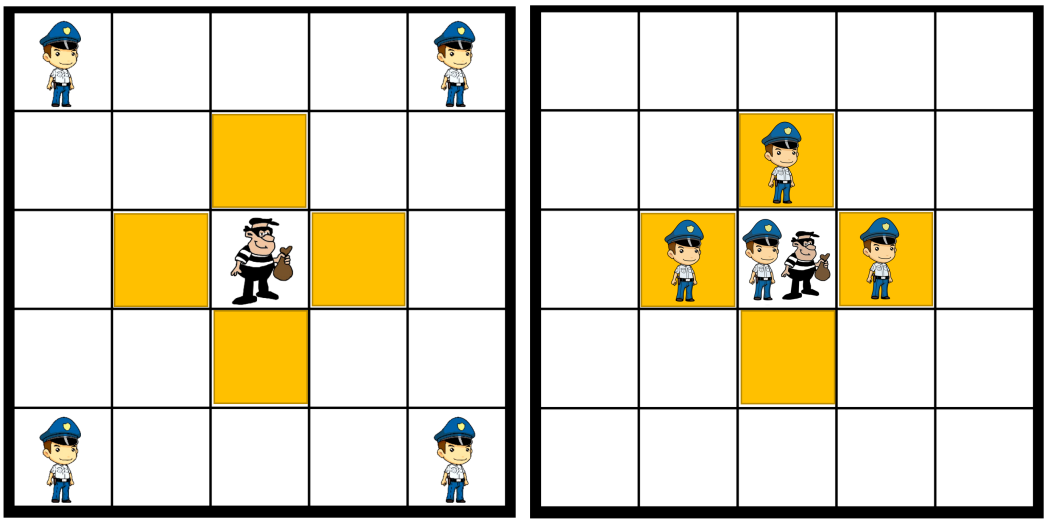
2.1.1 Agents-landmarks

In this environment[1] which is shown in Figure 1, there exist n agents that must cooperate through actions to reach a set of n landmarks in a two dimensional discrete $k \times k$ grid environment. We assume that the grid is bounded with walls, that is, if moving in a particular direction causes the agent to bump into a wall, the movement fails, and the agent stays in the same cell. The state of the environment includes the locations of all the agents, $d_t^1, d_t^2, \ldots, d_t^n$, and the locations of all the landmarks, $g^1, g^2, \ldots, g^n$. This environment runs in episodes and for every episode, the initial locations of the agents, $d_0^1, d_0^2, \ldots, d_0^n$, and the locations of all the landmarks, $g^1, g^2, \ldots, g^n$, are uniformly sampled such that no two locations are the same. Each episode consists of a number of time steps. At each time step, the agents simultaneously execute one of the five possible actions: move-up, move-down, move-left, move-right, or stand-still. If more than one agent moves to a cell, the movement fails and the agents stay in the their own cells.

Covering all the landmarks by all the agents at the same time is considered as the terminal step of the episode. We consider four reward functions:

1. Sparse team reward: At each time step, all the agents receive a penalty of -1 for delay in covering all the landmarks altogether or a zero penalty if all the landmarks have been covered.
2. Partial team reward: At each time step, all the agents receive a penalty of -1 multiplied by the number of uncovered landmarks.
3. Full informative team reward: At each time step, all the agents receive a penalty of $-1 \times \sum_{l=1}^{n} \min_{a \in \mathcal{A}} ||d_t^a - g^l||_1$ where $\min_{a \in \mathcal{A}} ||d_t^a - g^l||_1$ is the minimum distance from landmark l to all the agents at time step t.

[1] A similar environment, but with continuous state and action spaces, was first introduced in Mordatch and Abbeel (2017); Lowe et al. (2017).



(a) (b)

Figure 2: Predators-prey environment with $n = 4$ predators

2.1.2 Predators-prey

The pursuit problem is a popular multi-agent environment in which some agents, called predators, have to capture one or several preys in a gridworld Benda et al. (1986). In our environment, n predators must cooperate with each other to capture one prey in a two dimensional discrete $k \times k$ grid environment. We assume that the grid is bounded with walls, that is, if moving in a particular direction causes a predator or pray to bump into a wall, the movement fails, and the agent stays in the same cell. The state of the environment includes the locations of all the predators, $d_t^1, d_t^2, \ldots, d_t^n$, and the location of the prey, g_t. This environment runs in episodes and for every episode, the initial locations of the predators, $d_0^1, d_0^2, \ldots, d_0^n$, and the initial location of the prey, g_0, are uniformly sampled such that no two locations are the same. Each episode consists of a number of time steps. At each time step, the predators simultaneously execute one of the five possible actions: move-up, move-down, move-left, move-right, or stand-still. If more than one predator moves to a cell, the movement fails and the predators stay in their own cells. For the prey's movement model, we consider three following settings:

1. Still prey: In this setting, the location of the prey will be fixed over time. This setting corresponds to the case where the environment (from the view point of a meta-agent who chooses actions for all predators) is deterministic.

2. Random prey: In this setting, the prey chooses one of the five possible actions of moving up, down, left, right, or standing in the same cell with uniform probability. This setting corresponds to the case where the environment (from the view point of a meta-agent who chooses actions for all predators) is stochastic but independent of the actions of the predators.

3. Random escaping prey: In this setting, the prey first finds the set of "good" neighbor cells and then chooses one of these cells uniformly at random to move in to. For each cell, the set of adjacent cells are the ones located at the top, bottom, left, and right of this cell, which are not a wall. In Figure 2a, the set of adjacent cells for the prey are denoted in orange. Then, the set of good neighbor cells include the current cell itself and any of the adjacent cells which is not occupied by a predator. Similar to the random prey setting, the environment in this setting is also stochastic but it depends on the actions of the predators.

If the prey has m adjacent cells, it is captured when $m - 1$ of these cells are occupied by the predators and the one predator moves to the location of the prey while the other predators remain, for support, on their current positions. Figure 2b shows an example of situation where the prey is

captured. Capturing the prey is considered as the terminal step of the episode. We consider two reward functions:

1. Sparse team rewards: At each time step, all the predators receive a penalty of -1 for delay in capturing the prey or a zero penalty if they successfully capture the prey.
2. Full informative team reward: At each time step, all the agents receive a penalty of $-1 \times \sum_{a=1}^{n} ||d_t^a - g_t||_1$ where $||d_t^a - g_t||_1$ is the distance from predator a to the prey at time step t.
3. Joint team and individual rewards: At each time step, each predator a receives two rewards, the team reward of $-1 \times \sum_{a=1}^{n} ||d_t^a - g_t||_1$ and the individual reward of $||d_t^a - g_t||_1$.

2.2 Evaluation methodology

We trained the agents in both environments described above using the Independent Deep Q-learning (IDQN). For each environment, we considered the maximum number of time steps per episode to be 500. More specifically, each agent is equipped with two neural networks, online and target, to approximate its own state-action Q-function and a memory to store the experiences obtained through interacting with the environment. The agents are trained with Uniform Experience Replay (UER) memory and Prioritized Experience Replay (PER) memory. In addition to IDQN, we considered Independent double Q-learning (IDDQN) and dueling IDDQN.

2.3 Model architecture

We use the DQN architecture described by Mnih et al. (2015) with a few modifications. We do not have the set of convolutional layers since the input to the neural network is not an image. The input to the neural network is state representation and there is a separate output unit for each possible action. We considered a 3 layer neural network where the first two layers are fully-connected, each consists of 512 rectifier units. The output layer is a fully-connected linear layer with single output for each possible action. For the dueling DQN, following the architecture of Wang et al. (2015), we consider two sequences (or streams) of fully-connected layers where for the first one, the third layer has single output for each possible action while for the second one, the layer has one single output. The outputs of these two streams are then combined using Equation 5.

For the PER memory, we use proportional prioritization of Schaul et al. (2015). To sample a mini-batch of size k, the range of 0 to sum of all samples priorities is divided equally into k ranges a and then, a value is uniformly sampled from each range.

2.4 Hyper-parameter tuning

The combinatorial space of hyper-parameters is too large for an exhaustive search. We did not perform a systematic grid search owing to the high computational cost. Instead, we have only performed an informal search. The values of all hyper-parameters are provided in Table 1.

We linearly anneal ϵ from 1.0 to 0.01 over the of exploration phase (100000 time steps) and fixed it at 0.01 thereafter. For the first environment, we train each agent's network for a total of 50000 episodes while for the second environment, each network is trained for 20000 episodes. We use a replay memory of 10000 most recent time steps.

We use the RMSProp algorithm with a learning rate of 0.00005. Further, we use a discount factor of 0.99 and perform a learning update every 4 time steps, using mini-batches of 32 transitions.

For replay prioritization, we use priority exponent ω of 0.5, and linearly increase the importance sampling (IS) exponent from 0.4 to 1 over the course of exploration phase (100000 time steps).

References

Marc G Bellemare, Yavar Naddaf, Joel Veness, and Michael Bowling. 2013. The arcade learning environment: An evaluation platform for general agents. *Journal of Artificial Intelligence Research* 47 (2013), 253–279.

Table 1: List of hyper-parameters and their values

Parameter	Value
RMSProp learning rate	0.00005
Adam learning rate	0.0000625
Initial ϵ for exploration	1.00
Final ϵ for exploration	0.01
Number of exploration steps	100000
Steps between target network updates	1000
Replay memory size	10000
Minibatch size	32
Steps between online network updates	4
Discount factor	0.95
Prioritization type	proportional
Prioritization exponent ω	0.5
Prioritization importance sampling β	$0.4 \rightarrow 1.00$

M. Benda, V. Jagannathan, and R. Dodhiawala. 1986. *On Optimal Cooperation of Knowledge Sources - An Empirical Investigation*. Technical Report BCS–G2010–28. Boeing Advanced Technology Center, Boeing Computing Services, Seattle, WA, USA. `http://www.cs.utexas.edu/~shivaram`

Long-Ji Lin. 1992. Self-improving reactive agents based on reinforcement learning, planning and teaching. *Machine learning* 8, 3-4 (1992), 293–321.

Ryan Lowe, Yi Wu, Aviv Tamar, Jean Harb, OpenAI Pieter Abbeel, and Igor Mordatch. 2017. Multi-agent actor-critic for mixed cooperative-competitive environments. In *Advances in Neural Information Processing Systems*. 6379–6390.

Laetitia Matignon, Guillaume J Laurent, and Nadine Le Fort-Piat. 2012. Independent reinforcement learners in cooperative markov games: a survey regarding coordination problems. *The Knowledge Engineering Review* 27, 1 (2012), 1–31.

Volodymyr Mnih, Koray Kavukcuoglu, David Silver, Andrei A Rusu, Joel Veness, Marc G Bellemare, Alex Graves, Martin Riedmiller, Andreas K Fidjeland, Georg Ostrovski, et al. 2015. Human-level control through deep reinforcement learning. *Nature* 518, 7540 (2015), 529.

Igor Mordatch and Pieter Abbeel. 2017. Emergence of grounded compositional language in multi-agent populations. *arXiv preprint arXiv:1703.04908* (2017).

Tom Schaul, John Quan, Ioannis Antonoglou, and David Silver. 2015. Prioritized experience replay. *arXiv preprint arXiv:1511.05952* (2015).

Ming Tan. 1993. Multi-agent reinforcement learning: Independent vs. cooperative agents. In *Proceedings of the tenth international conference on machine learning*. 330–337.

Hado Van Hasselt, Arthur Guez, and David Silver. 2016. Deep Reinforcement Learning with Double Q-Learning.. In *AAAI*, Vol. 2. Phoenix, AZ, 5.

Ziyu Wang, Tom Schaul, Matteo Hessel, Hado Van Hasselt, Marc Lanctot, and Nando De Freitas. 2015. Dueling network architectures for deep reinforcement learning. *arXiv preprint arXiv:1511.06581* (2015).

Christopher John Cornish Hellaby Watkins. 1989. *Learning from delayed rewards*. Ph.D. Dissertation. King's College, Cambridge.