Uppsala University

Reinforcement Learning 7.5 HP

Playing Pong with DQN

Author: Wenhan Zhou, Chenglong Li, Ruizhen Chen, Haote Liu

May 17, 2024

1 Intoduction

The classical algorithms in reinforcement learning are mainly table-based Q-learning algorithms. These algorithms perform well when dealing with low-dimensional problems. However, the storage and computational complexity quickly become intractable as the state and action space increase.

The Deep Q-network (DQN) algorithm is one of the first widely applied deep learning models in the field of reinforcement learning. Proposed by the research team at DeepMind in 2013 [3], it combines deep neural networks (DNN) with Q-learning, enabling learning in high-dimensional and continuous state spaces. This method was later improved in an article published in Nature in 2015 [4].

In this paper, our ultimate goal is to train DQN to play Pong, which requires preprocessing frames, stacking observations, and designing a suitable convolutional neural network. We will develop and test DQN using the Kaggle platform. We will apply our method to two games implemented in The Arcade Learning Environment (ALE) [1], which is a software platform designed for the research and development of reinforcement learning algorithms, offering a simulation environment for classic Atari 2600 games.

In the game of Pong, the agent controls one of the paddles and aims to bounce the ball past the opponent's paddle. In this game, the action space is Discrete(6), offering six possible actions: NOOP (no operation), FIRE (start the game or serve), RIGHT (move the paddle to the right), LEFT (move the paddle to the left), RIGHTFIRE (move the paddle to the right and serve), and LEFTFIRE (move the paddle to the left and serve). And we have three observation types: "RGB", "grayscale", and "ram". For "RGB", observations are color images, for "ram", they represent the state of random-access memory, and for "grayscale", they are grayscale images. If the ball passes the opponent, the agent receives a positive reward; if it misses the ball, it receives a negative reward. The goal is to maximize rewards by moving the paddle effectively.

We started with a simplified game called CartPole. CartPole is a game where the objective for the agent is to stabilize a pole standing vertically on a cart by controlling its movement either to the left or to the right. The action space in the CartPole-v1 environment is represented by a 1-dimensional ndarray, allowing for two actions: 0 for pushing the cart to the left and 1 for pushing the cart to the right. These actions dictate the direction in which a fixed force is applied to the cart. The observation space is represented by a 4-dimensional ndarray. It encompasses the following parameters: cart position, cart velocity, pole angle, and pole angular velocity. The cart position ranges from -4.8 to 4.8, while cart velocity and pole angular velocity have an infinite range from negative to positive infinity. The pole angle spans approximately -0.418 radians ($-24°$) to 0.418 radians ($24°$). The episode ends if the pole angle exceeds $\pm 12°$, the cart position exceeds ± 2.4, or truncation occurs when the episode length exceeds 500 steps. In this section, we use DQN to solve the CartPole-v1 problem and explore the impact of hyperparameters on DQN performance in the discussion. We make the code publicly available at https://github.com/MaxCHENGLONG/RL-Project-1RT747-UU.

2 Method

Q-learning is a way of storing the rewards that can be obtained by performing various actions in various states in a table, called a Q-table, in which each value is defined as $Q(s, a)$, which represents the reward obtained by executing action a in state s. The choice can be made in a greedy way, i.e., by choosing the action with the highest value.

Q-learning relies on Q-table, but one of the problems is that large state spaces typically lead to memory issues, such as the game Go. Therefore, DQN has been proposed to serve as a function approximation of the optimal action value. DQN can in principle solve problems with infinite states and limited actions; specifically, it takes the current state as input and outputs the Q value of each action.

2.1 Q-learning

Q-Learning[5] is a value-based algorithm in reinforcement learning, Q is $Q(s, a)$, at a certain moment in the state of the state s, take the action a can get the expectation of gain, the environment will be based on the agent's action to feedback the corresponding reward r, so the main idea of the algorithm is to construct the state and the action into a Q-table to store the value of the Q value, and then according to the value of the Q value to select the action that can get the maximum reward.

Q-learning is based on the Bellman optimality equation, which relates the current action value to the maximum possible value of subsequent states:

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

R_{t+1} is the reward, and γ is the discount factor, representing the present value of future rewards. Then Q-learning uses the following update rule to approximate Q^* in a mode-free manner:

$$Q(S_t, A_t) \leftarrow Q(S_t, A_t) + \alpha\left[R_{t+1} + \gamma \max_a Q(S_{t+1}, a) - Q(S_t, A_t)\right]$$

α is the learning rate, which adjusts the step size of the update, R_{t+1} is the immediate reward obtained after transitioning from state S_t to S_{t+1} by taking action a.

2.2 Deep Q-Network

DQN uses the idea of Q-learning, although the Q-value that can be obtained for which action a is selected for a given state s is computed by a deep neural network. We consider an agent interacts with an environment ε, in a sequence of actions, observations, and rewards. At time-step t the agent receives an observation $x_t \in \mathbb{R}^{84\times84\times4}$, which is four 84 by 84 images stacked on top of each other. And then the agent selects an action a_t from the set of legal game actions, $\mathcal{A} = \{1, \cdots, K\}$, and receives a reward r_t which is the change in the game score.

The goal of the agent is to interact with the emulator by selecting actions in a way that maximizes discounted returns. So we can calculate the future rewards with $G_t = \sum_{t'=t}^{T} \gamma^{t'-t} r_{t'}$, where T is the time-step at which the game terminates and γ is discount factor. Our optimal action-value function $Q^*(s, a)$ can be defined as the maximum expected return achievable by following any strategy, after seeing some sequence s and then taking some action a,

$$Q^*(s, a) = \max_\pi \mathbb{E}[G_t | S_t = s, A_t = a]$$

where π is a policy mapping sequences to actions. But this method is very difficult to calculate, so we approximate it with another function $Q(s, a; \theta) \approx Q^*(s, a)$. We use a deep

neural network with weights θ as a function approximator, and this deep neural network is called a Q-network. A Q-network can be trained by minimizing a sequence of loss function $L_i(\theta_i)$ that changes at each iteration i,

$$L_i(\theta_i) = \mathbb{E}_{s,a\sim\rho(\cdot)}\left[(y_i - Q(s,a;\theta_i))^2\right]$$

where $y_i = \mathbb{E}_{s'\sim\varepsilon}[r + \gamma \max_{a'} Q(s',a';\theta_{i-1})|s,a]$ is the target for iteration i and $\rho(s,a)$ is a probability distribution. However, training this function can be unstable and will sometimes diverge. We modify the loss function to the following form in order to minimize this condition.

$$L_i(\theta_i) = \mathbb{E}_{(s,a,r,s')\sim U(D)}\left[\text{SmoothL1Loss}\left(r + \gamma \max_{a'} Q(s',a';\theta_i^-), Q(s,a;\theta_i)\right)\right]$$

where the SmoothL1Loss loss function is defined as [2]:

$$\text{SmoothL1Loss}(x,y) = \begin{cases} 0.5(x-y)^2 & \text{if } |x-y| < 1 \\ |x-y| - 0.5 & \text{otherwise} \end{cases},$$

where $x = r + \gamma \max_{a'} Q(s',a';\theta_i^-)$ and $y = Q(s,a;\theta_i)$ are the target Q-values and the estimated Q-values respectively. To train the agent, we consider the experience replay mechanism, which works as follows. Let $e_t = (s_t, a_t, r_t, s_{t+1})$ be the experiences of the agent at each time-step t and this experiences are stored in a memory buffer $D_t = \{e_1, e_2, \cdots, e_t\}$. During learning, we apply Q-learning updates, on samples of experience $(s,a,r,s') \sim U(D)$, drawn uniformly at random from the pool of stored samples. This is the experience replay mechanism, which is intended to reduce instability during learning. θ_i are the parameters of the Q-network at iteration i and θ_i^- are the network parameters used to compute the target at iteration i. The target network parameters θ_i^- are only updated with the Q-network parameters (θ_i) every C step and are held fixed between individual updates. The loss function with respect to the weights we arrive at the following gradient,

$$\theta_i \leftarrow \theta_i + \alpha \mathbb{E}_{(s,a,r,s')\sim U(D)}\left[\begin{cases} \delta & \text{if } |\delta| < 1 \\ \text{sign}(\delta) & \text{otherwise} \end{cases} \cdot \nabla_{\theta_i} Q(s,a;\theta_i)\right]$$

Here, we use δ to denote the TD error:

$$\delta = r + \gamma \max_{a'} Q(s',a';\theta_i^-) - Q(s,a;\theta_i)$$

Based on the above, we can know that the two innovations of DQN are the **experience replay mechanism** and the use of two networks which are the Q-network and the **target network**.

- Experience Replay: Since the training samples obtained by agents interacting with the environment are not independently identically distributed, in order to solve this problem DQN introduces an experience replay mechanism. It stores the data obtained by the system exploring the environment and then randomly samples past experiences to update the parameters of the DNN. By using a buffer that replays past experiences, previous samples are mixed with current ones, which helps to reduce the correlation between the data. Moreover, experience replay also makes the samples reusable, which improves learning efficiency.

- Target Network: Nature DQN adds a new mechanism separate Target Network, when calculating y_i, instead of using the Q-network, another network (Target Network) Q' is used. The reason is that the same network is used for calculating y_i and Q estimation, which makes the value of y_i larger for samples with larger value of Q estimation. The use of another independent network makes the training oscillations less likely to diverge and more stable.

3 Experiments

3.1 CNN Architecture

The CNN used in our training consists of three convolutional layers followed by two fully connected layers. Each convolutional layer is equipped with ReLU activations to introduce non-linearities into the model. The details of the layers are as follows:

- **First Convolutional Layer**: It has 256 filters of size 8×8 with a stride of 4.
- **Second Convolutional Layer**: Applies 256 filters of size 4×4 with a stride of 2.
- **Third Convolutional Layer**: Uses 256 filters of size 3×3 with a stride of 1.
- **Fully Connected Layers**: The output from the convolutional layers is flattened and passed through a fully connected layer of 512 neurons, followed by another layer that outputs a mini-batch of Q-values for each possible action.

3.2 Hyperparameters

We utilized the following hyperparameters in the training of CartPole-v1:

- **Replay Memory Capacity**: We set the capacity of the replay buffer to 50000.
- **Episodes**: We executed a total of 1000 episodes during training.
- **Batch Size**: For each optimization step, we sampled a batch size of 32.
- **Target Update Frequency**: We updated the target network based on a fixed number of optimization steps, specifically every 100 steps.
- **Training Frequency**: Optimization steps were performed at an interval of 1.
- **Discount Factor**(γ): To balance the importance of immediate and future rewards in reinforcement learning, we set the discount factor to 0.95.
- **Learning Rate**(α):The learning rate for the optimizer was set to 1×10^{-4}.
- **Initial ϵ and Final ϵ**: 1.0 and 0.1.The probability of choosing a random action ensures that the agent explores the environment.

The training of Pong is governed by the following hyperparameters:

- **Observation Stack Size**: 4. This parameter controls the number of the most recent frames stacked together as input to the network. By using 4, we can ensure efficiency while also making sure not to miss crucial information.
- **Replay Memory Capacity**: 100 000. This is the total number of previous transitions the network can store, allowing the agent to learn from past experience and reduce correlation between training samples.
- **Batch Size**: 64. The number of experiences sample from memory during one training iteration.
- **Target Update Frequency**: 1 000 steps. This determines how often the weights of the target network are updated to those of the policy network.

- **Training Frequency**: Every 4 steps. Indicates the regularity of network updates.
- **Discount Factor**(γ): 0.99. This factor discounts the value of future rewards.
- **Learning Rate**(α): 1×10^{-4}. This sets the step size in the gradient descent update of the network weights, affecting the speed and stability of learning.
- **Initial ϵ and Final ϵ**: 1.0 and 0.01 . The probability of choosing a random action ensures that the agent explores the environment.

3.3 Training Process

This section details the training process implemented in our DQN.

1. **Initial Setup**: Our Q-Network CNN is initialized with random weights. This network is used to approximate the Q-values for different actions based on the given state inputs. We also create an empty replay memory that is set up to store experiences. This space will store memories of what happened at each step: state, action, reward, and the next state after the action is taken.
2. **Observation Processing**: Each state (scene from the game) is processed through function (`preprocess(obs)`) to transform it into suitable format for the CNN.
3. **Policy Execution (ϵ-Greedy)**: Using the `act` method, the agent selects actions based on ϵ-greedy strategy.
4. **Experience Storage**: After action being executed, the environment returns the reward and next state. The new data (current state, action, reward, next state) is stored in the replay memory.
5. **Learning by Sampling from Replay Memory**: The network periodically samples a batch of experiences from the replay memory to perform learning updates. Each sample is processed through the CNN to predict current Q-values and to compute the loss.
6. **Q-Learning Updates**: The Q-values are updated by minimizing the loss between the predicted Q-values and the computed targets.
7. **Periodic Updates to Target Network**: To stabilize the learning process, the weights of a separate target network are periodically updated with the main CNN.
8. **Repeat**: Repeat the above steps until the agent's performance reaches our goal.

4 Results

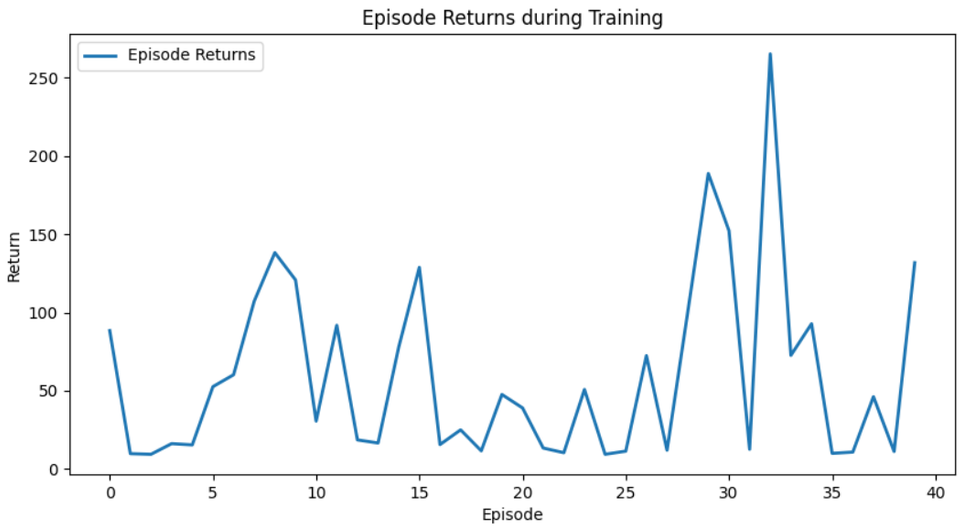

Figure 1: Average returns over 5 episodes for the CartPole-v1 environment.

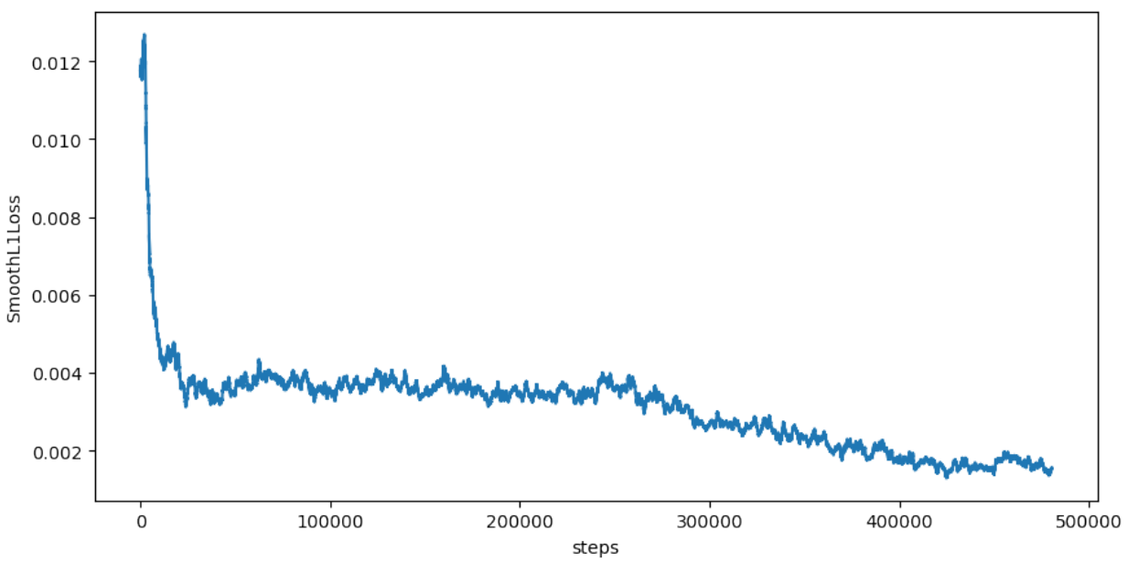

Figure 2: Loss curve when training DQN for Pong. The loss we used is the SmoothL1Loss [2] for the predicted and target Q-value. We have performed a moving average with a window size of 1000 for visualization purposes.

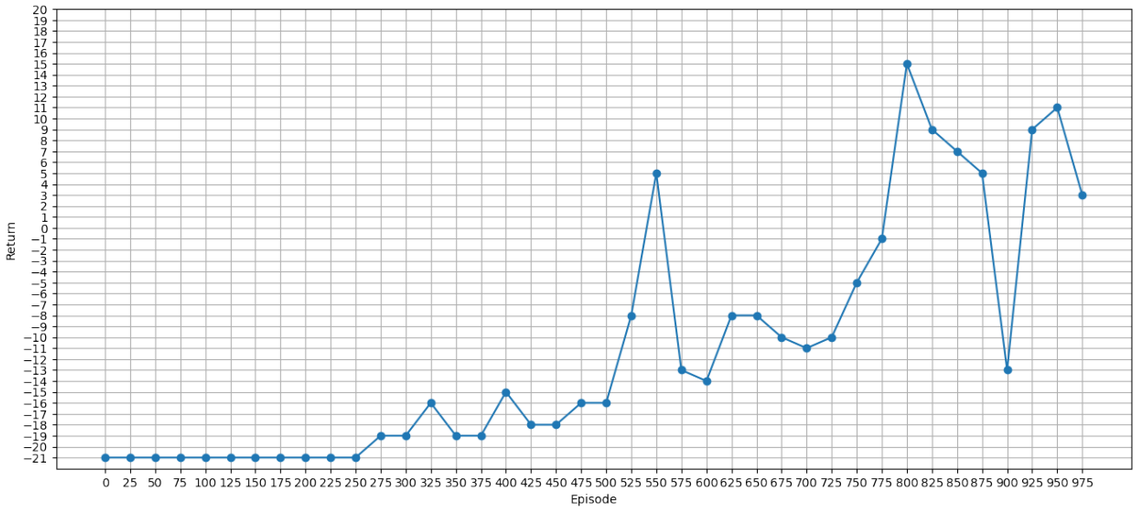

Figure 3: Return for one round of game-play when evaluated for every 25 episodes.

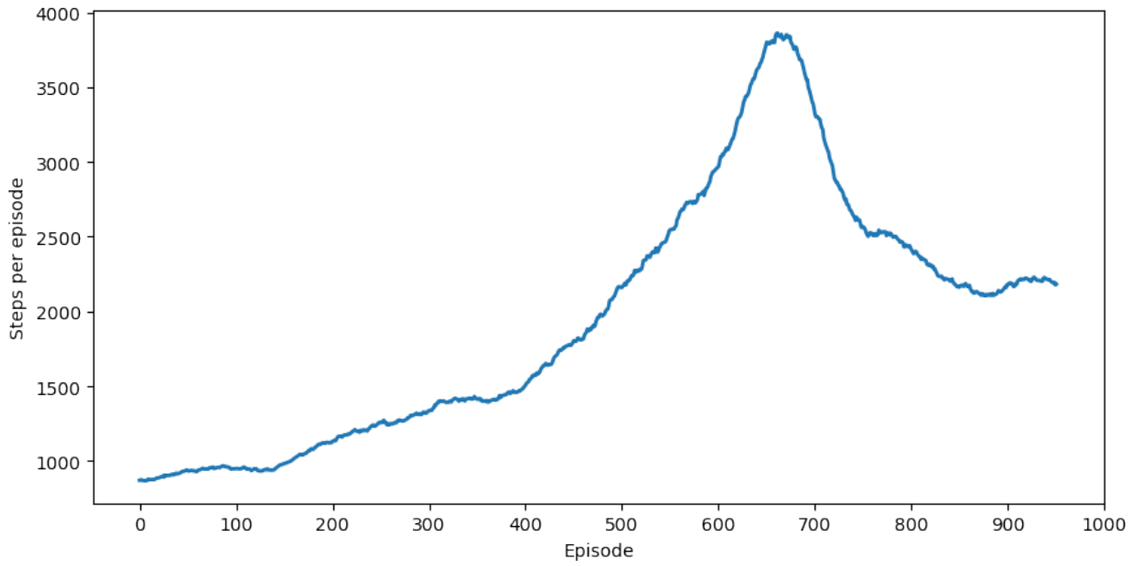

Figure 4: Number of steps per episode averaged for every 50 episodes. We can see that during the exploration phase, the games do not last very long as the DQN agent loses very fast. The lengths get longer as the agent improves and reaches its maximum when the agent approximately ties with the system opponent. Then it starts to decrease because it starts to win more.

5 Discussion

The observation stack size must at least be 2 in order to encode the position and velocity information. If it is set to some number larger than 3, then the state space may also encode reflection information. Setting it to a large value (relative to 4) will increase the memory usage without encoding significant game mechanics.

The training process goes very fast (5 seconds per episode) at the beginning as the forward passes of the DQN are mostly random. We used a linear annealing schedule for epsilon, so the time it takes for an episode to finish depends on the value of epsilon, but also how strong the agent has become. A possible option could be to implement an exponential decaying schedule to decrease the number of lengthy episodes at the end of the training.

The reply memory size is closely related to how the agent explores the observation space. This is because the weight updates are related to the samples drawn from the replay buffer. As a consequence, the agent will explore more about the observation space that is randomly sampled as suggested by the large epsilon at the beginning. Suppose we consider a smaller replay memory size, such as 10000 (about 2% of the total training process). In that case, the agent will forget to explore the states it visited at the earlier phase of training and instead explore more of the recent states. This may result in faster convergence but also increases the risk of being trapped in some local minima in the loss landscape. In practice, the differences are insignificant, both 10000 and 100000 works.

The batch size is related to the number of samples each forward pass of the DQN can handle at once, therefore, it influences the randomness of each weight update as a larger batch size indicates diverse experience samples from the observation space. It is typically good to have a larger batch size if the GPU memory is not fully utilized, this is why we increased it from 32 to 64.

Target update frequency affects the convergence and stability of the target Q-network. Small values may therefore lead to training instabilities while too large may lead to slow convergence. Note that we use a constant learning rate throughout the training because the target is non-stationary, thus, the agent must always be capable of adapting itself for dynamic targets.

The training frequency is set to 4, which is different from the typical value 1. The advantage of setting it larger than 1 is to fill up the replay memory faster, so the agent has more experience to sample from.

The discount factor is set relatively large. This is because most of the actions that the agent makes are useless, thus it is advantageous for it to be ”forgetting” and prioritize ”adaption”, especially those after the ball bounces off the pad. After that, the expected returns are relatively independent of the actions of the agent. The kind of planning that the agent needs is when the ball bounces back and it needs to position itself for a reasonable counterattack, this is where we need a large discount factor.

The choice of initial and final epsilon influences the exploration and exploitation of the agent. It is typically a good idea to have a large initial epsilon starting at $\epsilon = 1$ as it is fast to perform forward passes and the exploration phase is diverse. It is more important to choose how the epsilon decays and the final epsilon. Typically, we want the epsilon to be small when we observe good performance, but it is difficult to know in advance when the training process converges. The final epsilon is relatively insensitive to the performance, it works with $\epsilon = 0.1$, probably also with higher values. However, it is usually a good idea to set it to some non-zero value because the environment may behave unexpectedly.

References

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

[2] Ross B. Girshick. Fast R-CNN. *CoRR*, abs/1504.08083, 2015.

[3] Volodymyr Mnih, Koray Kavukcuoglu, David Silver, Alex Graves, Ioannis Antonoglou, Daan Wierstra, and Martin Riedmiller. Playing atari with deep reinforcement learning, 2013.

[4] Volodymyr Mnih, Koray Kavukcuoglu, David Silver, Andrei A. Rusu, Joel Veness, Marc G. Bellemare, Alex Graves, Martin Riedmiller, Andreas K. Fidjeland, Georg Ostrovski, Stig Petersen, Charles Beattie, Amir Sadik, Ioannis Antonoglou, Helen King, Dharshan Kumaran, Daan Wierstra, Shane Legg, and Demis Hassabis. Human-level control through deep reinforcement learning. *Nature*, 518(7540):529–533, 2 2015.

[5] Christopher J.C.H. Watkins and Peter Dayan. Technical note: Q-learning. *Machine Learning*, 8(3):279–292, 5 1992.