Pacman Capture the Flag with Deep Reinforcement Learning

GROUP 1

Alex Hermansson Gianluigi Silvestri

1992-11-18 1994-01-06

grobgeld@kth.se giasil@kth.se

Abstract

In this assignment, we develop a team of intelligent agents for the adversarial game Pacman Capture the Flag. The agents learn to play the game from visible information through rewards, resembling the way animals learn from dopamine signals. This method, called Reinforcement Learning, has had previous success in other game domains, but not in the Multi-Agent scenario. We present a feasible solution approach that is shown to steadily improve by playing games on its own against a baseline team.

1 Introduction

In the study of Artificial Intelligence (AI), a common testbed is that of computer games. It is a controlled environment, complex enough to provide many of the essential problems that arises in the study of intelligence and in cognition. In this assignment, we are asked to develop an AI that is able to play the game "Pacman Capture the Flag" from UC Berkeley [1]. To assess the quality of the solution, a small tournament is played against other teams. The approach that we decided to use to build our AI is based on Reinforcement Learning [2]. Rather than studying ad-hoc solutions to solve the game, we try to train our agents and let them learn to play with the "trial and error" mechanism, aiming to achieve human-level performances like in [3].

1.1 Contribution

In the following report, we propose a possible approach to use Deep Reinforcement Learning in an Adversarial Multi-Agent setting, in particular in the game of Pacman Capture the Flag.

1.2 Outline

In section 2 we present an overview of the Pacman Capture the Flag game, together with the principles of Reinforcement Learning, DQN algorithm and its recent improvements. In section 3 the method adopted to use the DQN on the Pacman Capture the Flag environment are explained. Finally, we report the obtained results in section 4 and discuss possible improvements in section 5.

2 Related Work

2.1 Pacman Capture the Flag

Pacman Capture the Flag is a game inspired by the popular arcade game Pacman [4]. In the original version, Pacman is controlled by the player, and the goal is to navigate through a maze eating food while escaping from the enemies, that are represented as colored ghosts. In the "Capture the Flag" variant, there are two teams competing against each other. The maze is divided in two sides, and each team has a defensive and an offensive side. The teams are composed by the same number of agents (usually two). When an agent is on the defensive side it is a ghost, while when it is on the offensive side it is a Pacman. An example of environment can be seen in figure 1.

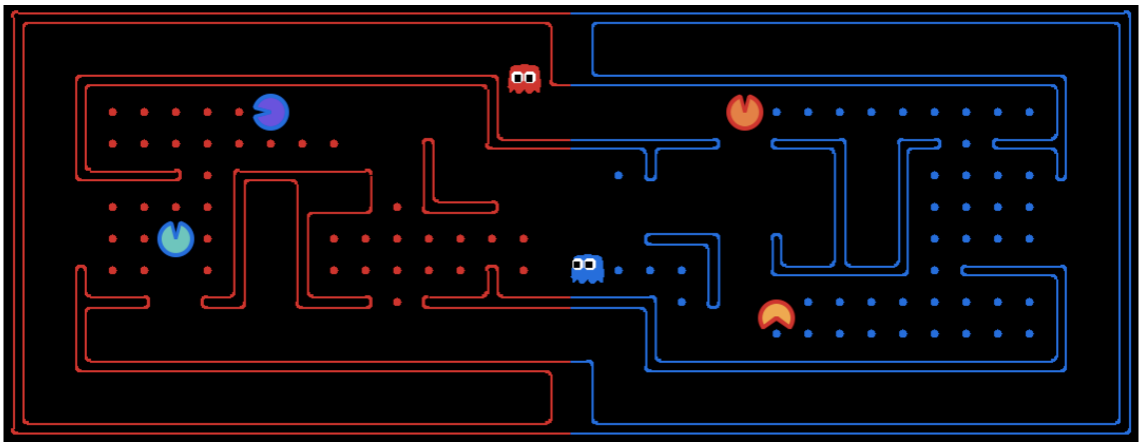

Figure 1: A Pacman Capture the Flag game environment.

To score points in a game, a team has to eat food on the offensive side and return it to the defensive side. The game ends when no more than two units of food are left on the offensive side, or when the limit of allowed moves is reached (300 moves per agent). In the latter case, the team that returned more food wins. More detailed rules can be found in the contest website [1].

2.2 Reinforcement Learning

The nature of learning can often be thought of as interaction with an environment. All animals and humans do this regularly in order to achieve some sort of internal or external goals. Reinforcement Learning (RL) is the computational approach to learn by interacting with an environment and is a subfield of machine learning.

The problem of RL can be formalized as a Markov decision process (MDP) and the "learning" part is concerned with learning a mapping from states to actions to maximize a scalar reward signal. This mapping is called policy and is denoted π. A finite Markov decision process is a classical formalization of sequential decision making, where actions a affect both immediate and future states s, and rewards r. All of these can be seen as random variables $s \in \mathcal{S}$, $a \in \mathcal{A}$, and $r \in \mathcal{R}$, and the interaction that the agent has with the environment is visualized in figure 2.

In a setting where an intelligent agent interacts with an environment and learns a policy, the problem of exploration vs exploitation arises. One simple idea is to choose the action that is predicted to give the most cumulative future reward with some probability (exploitation), and the rest of the time try out random actions (exploration). This way of acting is called ϵ-greedy policy, and means that the agent performs random exploration with probability ϵ, and the rest of the time exploits the knowledge already acquired.

The idea of formalizing the goal as a reward signal is one of the most noticeable characteristics of Reinforcement Learning. This way of guiding learning to achieve goals can be summarized in the reward hypothesis [2]:

That all of what we mean by goals and purposes can be well thought of as the maximization of the expected value of the cumulative sum of a received scalar signal (called reward).

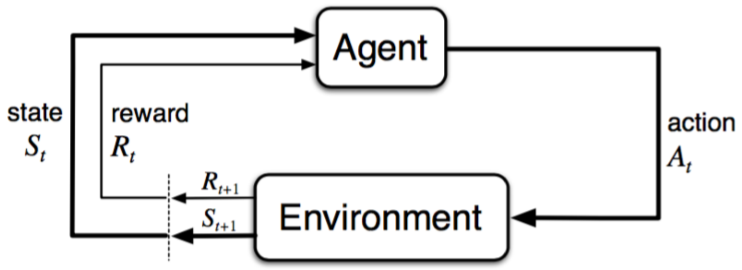

Figure 2: Agent-Environment interaction in an MDP, figure taken from [2].

2.2.1 Q-Learning

A common approach to RL problems is to evaluate states or action-state pairs in terms of how much future rewards can be expected when following a policy π. The expected return from time t, denoted G_t, is defined as the discounted sum of future rewards

$$G_t = R_{t+1} + \gamma R_{t+2} + \gamma^2 R_{t+3} + \cdots = \sum_{k=0}^{\infty} \gamma^k R_{t+k+1} \tag{1}$$

The parameter $\gamma \in [0, 1)$ is called a discount factor and accounts for that present rewards are more valuable since they are more certain than future ones. Also, it is more convenient mathematically since the sum in equation (1) is always finite if the reward sequence is bounded.

Intuitively, the value of a state $s \in \mathcal{S}$ under a policy π is the expected return when starting in s and following π. For discrete MDPs, the state-value, denoted $v_\pi(s)$, is formally defined by

$$v_\pi(s) = \mathbb{E}_\pi \left[G_t | S_t = s \right], \quad \forall s \in \mathcal{S} \tag{2}$$

In a similar way, the value of an action-state pair, also called action-value function, is the expected return for being in a state s, taking an action a and thereafter following a policy π. It is denoted $q_\pi(s, a)$ and is defined by

$$q_\pi(s, a) = \mathbb{E}_\pi \left[G_t | S_t = s, A_t = a \right], \quad \forall s \in \mathcal{S}, \forall a \in \mathcal{A} \tag{3}$$

According to the Bellman equation [2], the action-value for the policy π can also be computed recursively with

$$q_\pi(s, a) = \mathbb{E}_\pi \left[R_{t+1} + \gamma q_\pi(S_{t+1}, A_{t+1}) | S_t = s, A_t = a \right] \tag{4}$$

This formulation is essential to all of Reinforcement Learning, and in particular for Q-learning. In Q-learning, the goal is to try and update an estimate, denoted Q, of the optimal action-value function, denoted q_* and defined by

$$q_*(s, a) = \max_\pi q_\pi(s, a) \tag{5}$$

To update the estimates of q_*, we use the Q-learning algorithm proposed in [5] defined by

$$Q(S_t, A_t) \leftarrow Q(S_t, A_t) + \alpha \left[R_{t+1} + \gamma \max_a Q(S_{t+1}, a) - Q(S_t, A_t) \right] \tag{6}$$

Under certain mild assumptions, the Q-learning algorithm has been shown to converge to q_* with probability 1 [6]. In more detail, the algorithm updates estimates of q_* for each state and each action. To do so, first the Q-values for all the state-action pairs are initialized together with an initial state. Then, an action is chosen according to the policy derived from the current Q-values (usually the ϵ-greedy policy), a reward is obtained and the successor state is observed. This procedure is repeated until a terminal state is reached, determining the end of a so called episode. Afterwards, a new episode is started and the whole process is repeated until convergence of the Q-values or until a certain number of episodes is reached.

One problem with standard Q-learning is that values for all states and actions have to be stored. If the state space is large, this is infeasible for obvious reasons, for example consider the case of chess where, as a rough estimate, the number of different board positions is in the order of 10^{40}.

2.2.2 DQN

A solution to the space complexity is to introduce a function approximation $f : \mathcal{S} \times \mathcal{A} \rightarrow \mathbb{R}$ that maps a state-action pair to a Q-value, instead of storing all Q-values explicitly. Q-learning has been shown to still converge if a linear function approximator is used, but for non-linear function approximators there is a possibility that it diverges. One of the first successful applications of non-linear function approximation with a deep neural network together with Q-learning was done by Google DeepMind [3]. The artificial agent was termed DQN or a deep Q-network and consisted of a deep convolutional neural network that approximated the action-values in the Q-learning algorithm. In order to overcome the problem of instability of learning that has been seen previously, two concepts were introduced: experience replay and separate networks for action-values Q_t and their target values Q_t^{target} defined from equation (6) by

$$Q_t^{target} = R_{t+1} + \max_a Q(S_{t+1}, a) \tag{7}$$

One reason for instability in learning a nonlinear function approximation is that the assumption of independent and identically distributed data does not hold when samples are collected sequentially in the RL setting. Instead [3] proposes to store experience tuples $e = (s, a, r, s')$ in a replay memory $D_t = \{e_1, \dots, e_t\}$ and sample experiences uniformly from this memory to update the network with these instead of doing online learning. This breaks the correlation between the samples, leading to more stable learning. Another advantage of experience replay is that it can reduce the amount of interaction the agent has to do with the environment due to the fact that experiences can be reused for learning.

The loss for a batch of sampled experiences is the squared distance between Q_t and Q_t^{target} for that batch

$$L_t(\theta) = \mathbb{E}_{(s,a,r,s')\sim U(D)} \left[(Q_t^{target}(\theta^-) - Q_t(\theta))^2 \right] \tag{8}$$

where θ^- are the parameters for the target network. These are updated every C steps to the current network parameters θ of the online network, and otherwise held fixed. This is to reduce the correlation between the predicted Q-values and the target Q-values, again in accordance with [3].

Equation (8) is used as the loss function for the network such that the parameters, denoted θ, of the network can be updated with stochastic gradient descent, where the gradients are computed with backpropagation.

2.2.3 Prioritized Experience Replay

One limitation that can be seen with the experience replay that was used in the DQN agent is that experiences are sampled uniformly. Intuitively, some experiences are more valuable than others, and these should be used more than experiences that do not hold much information. A way to overcome this issue was introduced in [7]. The experience j can be given a priority score p_j based on the temporal-difference (TD) error, such that the priority is measured by

$$p_j = \left| R_j + \gamma \max_a Q(S_j, a; \theta^-) - Q(S_{j-1}, A_{j-1}; \theta) \right| + \delta \tag{9}$$

Where δ is a small positive constant so that also experience with zero TD error has a small probability to be sampled. Instead of sampling uniformly, the probability is based on the priority such that the probability of sampling experience j is

$$P(j) = \frac{p_j^\alpha}{\sum_k p_k^\alpha} \tag{10}$$

Results from [7] shows that Prioritized Experience Replay gives a huge performance gain compared to using the uniformly sampled experiences.

2.2.4 Double DQN

Both the Q-learning and the DQN algorithms have been shown to overestimate action-values. It has also been shown that such overestimations are quite common and that they tend to harm performance. The Double Q-learning, first introduced in the tabular case and then extended to DQN (termed Double DQN), deals with this issue and reduces the overestimations with the effect of better performance [8]. The reason for the overestimations is the max operator for the target Q-values in equation (7). The max operator uses the same values both to select and evaluate an action, this leads to a higher likelihood of selecting overestimated values thus leading to optimistic values.

In Double DQN, the way to reduce these overestimations of the Q-values is to have separate networks to select actions and evaluating action-values [8]. The natural choice is to use the online network and the target network for these two tasks. Thus the target for Double DQN is instead

$$Q_t^{target} = R_{t+1} + \gamma Q(S_{t+1}, \underset{a}{\operatorname{argmax}}\, Q(S_{t+1}, a; \theta); \theta^{-}) \tag{11}$$

where the action is chosen with the online network, but the action-value is evaluated with the target network.

3 Our method

The model used to train our agents is mainly based on the DQN described in section 2.2.2, extended with Double DQN (section 2.2.4) and Prioritized Experience Replay (section 2.2.3). The main differences are in the representation of the input state and in the way we assign rewards. Also the structure of the Deep Neural Network used is slightly different, and the hyperparameters will be listed in Section 4. For the implementation of the Prioritized Experience Replay, we used a sum tree as suggested in [7] to make the sampling efficient.

3.1 Input state

The input state used in [3] is a preprocessed tensor of dimension 84x84x4, where each 84x84 layer is a frame of the game, and four different frames are stacked together. The frames are not consecutive, but usually taken one every four frames. In the Pacman Capture the Flag environment, we don't have direct access to the screen, but all the information needed can be retrieved through the provided code. Thus, we used an approach similar to the one described in [9]. In the established tournament rules, the teams are composed by two agents, and the games were played using random maps. Since these maps have fixed size 34x18, our input state was composed by seven 34x18 matrices stacked on the top of each other, each matrix representing a different feature:

- Walls: each cell is 1 if there is a wall, 0 otherwise;
- Food: each cell is 1 if it contains food that we are defending, -1 if it contains food to eat, 0 otherwise;

- Capsules: each cell is 1 if it contains a capsule that we are defending, -1 if it contains a capsule to eat, 0 otherwise;
- Our Agent: The matrix is all zeros apart from the position of our agent, that is -1 when it is a ghost and 1 when it is a Pacman.
- Team Mate: The matrix is all zeros apart from the position of our team mate, that is -1 when it is a ghost and 1 when it is a Pacman.
- Opponents: The matrix is all zeros apart from the position of our opponents, that is -1 when it is a ghost and 1 when it is a Pacman. This is applicable only when the opponents can be seen from one of our agents.
- Scared Agents: This matrix is all zeros, with exception when one of the four agents is scared. In that case, the cell containing the scared agent will have value 1. If the scared ghost is on the opponent team and none of our agent can see it, then nothing will change.

3.2 Rewards

The reward in the Atari games is assigned based on the score of the game. In our case, using the score as reward is not very useful. In fact, in the beginning of the training the agents would move randomly, and since the only way to score points is to eat food and return it, the possibilities of this happening are very low. To overcome this issue, we introduced additional rewards in order to let the agent learn the desired behavior:

- The initial reward at each turn is $R = -0.01$. This helps to stimulate the agent to take actions that give positive reward.
- $R = R - 0.5$ if the action taken is *STOP*. In this way, the agent will stop only if there is a good reason.
- $R = R + D$ if the agent is not carrying food. D is the difference between the current and the previous state in distance to the closest food to eat. In this way, the agents gets a +1 reward when it gets closer to the food, and a negative reward if it is eaten by an opponent. The condition of not carrying any food helps when the agent is about to eat food. In fact, if the second closest unit of food is far, the agent will get a negative reward for eating, and as a consequence would learn to stop in front of the closest food. When the agent is eaten, it will get a negative reward, since the distance to the closest food increases.
- $R = R + 10$ if the agent eats food.
- $R = R + (10 * Food\_returned)$ when the agent returns food. In other words, the reward is 10 multiplied by the units of food returned in the current turn.
- $R = (10 * DS)$ if there are not more than two units of food to eat left and the agent is a Pacman. In this case, DS is the difference between the current and the previous state in distance to the start position. Since the game would end in a win if the agent returns the currently carried food, this reward is useful to teach the agent to go back to its side of the maze.

All the rewards listed above help the agent to learn an offensive behavior. In order to introduce defensive behavior, the following rewards were added:

- A positive reward to go closer and a negative reward to go farther from an opponent if the agent is a ghost and the opponent is a Pacman;
- A negative reward to go closer and a positive reward to go farther from an opponent if the agent is a scared ghost and the opponent is a Pacman;
- A negative reward to go closer and a positive reward to go farther from an opponent if the agent is a Pacman and the opponent is a ghost.

These updates can be used only if the opponent is seen by at least one of the agents in our team, and the reward is multiplied by 10 and divided by the squared distance between the agent and the opponent, in order to give more importance to situations where the agent is very close to the opponent. A last reward is $R = R + 20$, and is given if the agent eats an opponent.

During the games, the DQN may select actions that are not allowed. To deal with these cases, we didn't introduce any negative reward, but rather select a random action among the allowed ones. However, the agent should learn to avoid actions that are not allowed and instead choose the action that gives the highest reward.

4 Experimental Results

In order to participate in the tournament, the agents needed to be able to play on random maps, with no distinction between being in the blue or in the red team. When trained on a single map, and always in the same team, the agent was not able to generalize, thus we decided to train two different agents: a "simple version" trained with the same map and the same team to test the performances of the learning algorithm, and a "tournament" version always trained on different maps. Due to technical issues, we were not able to train agents in different teams with the same network, so we trained a blue and a red tournament agent. In both versions, the training was performed in a way inspired by [10], where two agents in the same team use a shared memory and a shared network. To balance exploration and exploitation, one agent had exploration rate $\epsilon = 0.3$ and the other $\epsilon = 0.1$. During training, the opponent team was always the basic team provided with the Pacman Capture the Flag environment.

4.1 Simple Agent

The Deep Neural Network used to train this agent was composed by two convolutional layers with ReLu activation and one hidden layer with ReLu activation, while linear activation for the output layer. For the first layer we used sixteen filters 3x3 with stride=1, while for the second layer we used 32 filters 3x3 with stride=1. As hidden layer we used 256 hidden units, and the learning rate was set to $\frac{0.00025}{4}$. The network architecture is the same used in [9]. This network is relatively fast to train, and we could see the results of the training without the complication of different mazes. To verify the effective improvement of the agents behavior, we trained them for 7000 games, and saved the total amount of food eaten and food returned by the team in each game, together with the final score of the game. The figures 3, 4, and 5 show the results of the training, and the plotted values are the average of 10 consecutive games.

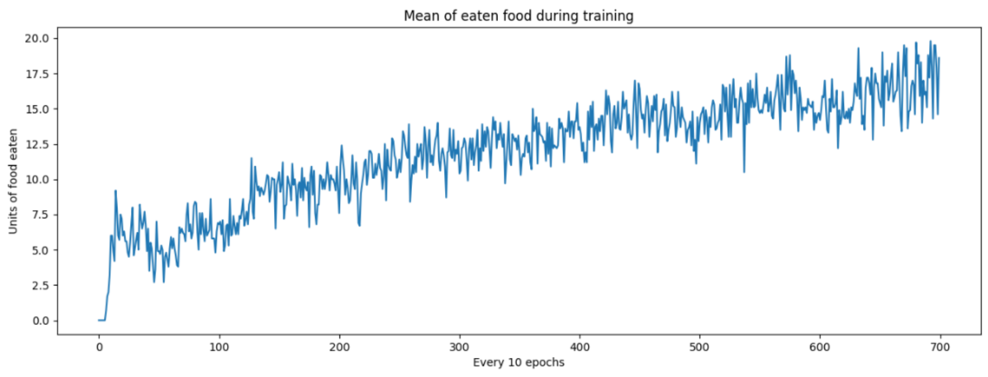

Figure 3: Food eaten by our team averaged for every ten games.

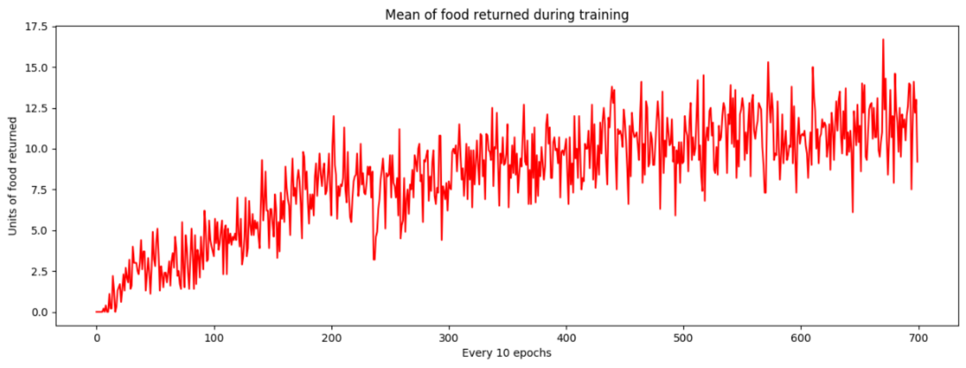

Figure 4: Food returned by our team averaged for every ten games.

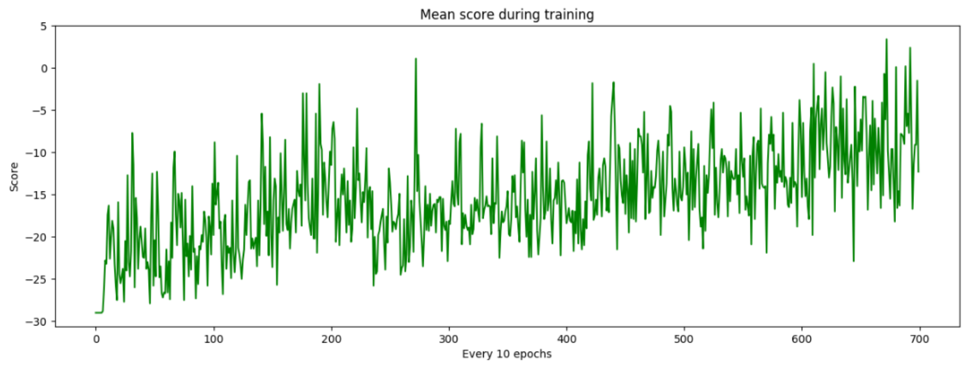

Figure 5: Final score of the game averaged for every ten games (the score is calculated as our food returned minus opponent's food returned).

4.2 Tournament Agent

The network used was the same as for the Simple Agent, but with 32 filters in the first convolutional layer, 64 in the second and 512 hidden units. After training the agent for about 3000 games, we participated to the tournament without scoring any point. The best performances were achieved by group using AI based on MinMax strategies or Behavior Trees.

5 Summary and Conclusions

In this assignment we have studied and applied the popular DQN algorithm to train agents to play the game Pacman Capture the Flag. Our learning algorithm showed promising results when always used on the same map. However, when trained on several different maps for 3000 games, the agents didn't show an "intelligent behavior", and could barely reach the offensive side of the maze. To achieve better results there are several strategies to be tested. The first option would be increasing the amount games played. Since we can see that for the simple agent in section 4.1 the performances still improve after 7000 games, we believe that for the more complicated tournament agents many more games are required. Other factors to investigate are the input state representation and he hyperparameters of the Deep Neural Network. A last improvement can most likely be obtained by modifying the reward function. In fact, the agents learn the offensive behavior well (figure 3 and 4), but as can be presumed from the low scores (figure 5) they lack a good defensive strategy. In addition, we have not introduced any reward to emphasize the win or loss of a game, since we have not found a reasonable way to introduce information about the time left in the input state. A final remark is about the nature of the problem: Pacman Capture the Flag is a Multi-Agent game, and the outcome of the game doesn't only depend in the behavior of a single agent. Thus, assigning rewards for action taken by other agents (for example when an opponent returns food) would lead our agent to misunderstand the effect of its actions. Additional research can be done in the field of Multi-Agent Reinforcement Learning to find a suitable way to deal with this problem.

References

[1] UC Berkeley. Pacman Capture the Flag. URL `ai.berkeley.edu/contest`.

[2] Richard S Sutton and Andrew G Barto. *Reinforcement learning: An introduction*, volume 1. MIT press Cambridge, 1998.

[3] Volodymyr Mnih, Koray Kavukcuoglu, David Silver, Andrei A Rusu, Joel Veness, Marc G Bellemare, Alex Graves, Martin Riedmiller, Andreas K Fidjeland, and Georg Ostrovski. Human-level control through deep reinforcement learning. *Nature*, 518(7540):529, 2015. ISSN 1476-4687.

[4] Wikipedia. Pac-Man. URL `https://en.wikipedia.org/wiki/Pac-Man`.

[5] Christopher John Cornish Hellaby Watkins. Learning from delayed rewards, 1989.

[6] Christopher J C H Watkins and Peter Dayan. Q-learning. *Machine learning*, 8 (3-4):279–292, 1992. ISSN 0885-6125.

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

[8] Hado Van Hasselt, Arthur Guez, and David Silver. Deep Reinforcement Learning with Double Q-Learning. In *AAAI*, volume 16, pages 2094–2100, 2016.

[9] Tycho van der Ouderaa. Deep Reinforcement Learning in Pac-man. 2016.

[10] Dan Horgan, John Quan, David Budden, Gabriel Barth-Maron, Matteo Hessel, Hado van Hasselt, and David Silver. Distributed Prioritized Experience Replay. *arXiv preprint arXiv:1803.00933*, 2018.