LU-TP 20-54
September 2020

Application of Deep Q-learning for Vision Control on Atari Environments

Jim Öhman

Department of Astronomy and Theoretical Physics, Lund University

Master thesis in collaboration with Sentian.AI
Supervised by Mattias Ohlsson (Lund University)
Co-supervised by Viktor Stagge (Sentian.AI)

LUND
UNIVERSITY

Abstract

The success of Reinforcement Learning (RL) has mostly been in artificial domains, with only some successful real-world applications. One of the reasons being that most real-world domains fail to satisfy a set of assumptions of RL theory.

In the past years, a popular way to gauge the performance of RL agents has been through a suite of Atari 2600 games. This suite has been used to benchmark the progress of building successively more intelligent agents. However, they do not capture all the challenges that make real-world tasks difficult for RL, such as having to learn and act with incomplete information.

This thesis modifies a set of Atari games to include the task of adaptive sensing for RL agents. The games are made partially observable by restricting the visible portion of the screen. The agents are then tasked to learn to control their vision while at the same time learn to play the game. This modification adds one of the extra challenges that are present in many real-world environments.

To solve these new tasks an algorithm based on a slight modification of Deep Q-learning is proposed, referred to as Myopic Deep Q-Learning (MyDQL). Furthermore, a comparison is made between two different network architectures for MyDQL, a feed-forward neural network, and a recurrent neural network.

It is shown that MyDQL can be successfully applied to the modified Atari games. Additionally, it is shown that using a recurrent neural network greatly enhances the performance of the agent on these tasks. Such an agent is able to achieve near-optimal performance on Pong, Breakout, and Space Invaders, with only 35% of the screen visible at any given time.

It is also shown that an agent with its visibility further reduced to 13% is still able to achieve impressive performance on these games.

Populärvetenskaplig beskrivning

Vi människor, och djur i allmänhet, lär oss enormt mycket från att bara interagera med våra omgivningingar. Vi samlar ständigt in information om hur våra handlingar påverkar omgivningen, och justerar vårat beteende så att vi bättre uppfyller våra mål.

En gren inom maskininlärning, som kallas förstärkningsinlärning, är ett grupp metoder som beskriver en agent som kan lära sig att uppnå mål från erfarenheter. En sådan agent interagerar med sin omgivning genom olika handlingar och får samtidigt numeriska belöningar beroende på hur omgivningen påverkas. Det slutgiltiga målet för agenten är att lära sig ett beteende som samlar in mest belöning långsiktigt.

Dessa metoder har igen blivit ett hett ämne då det kombinerats med de nyaste teknikerna inom djupinlärning. Denna kombination gjorde att förstärkningsinlärning kunde appliceras på större omgivningar och mer komplexa problem.

Majoriteten av applikationer inom förstärkningsinlärning har dock varit på artificiella omgivningar. En av anledningarna till varför är att verkliga omgivningar ofta bryter mot antagandet att omgivningen är helt observerbar för agenten. För att bättre kunna applicera förstärkningsinlärning på verkliga omgivningar krävs metoder som släpper på detta antagande.

Mycket av forskningen inom förstärkningsinlärning har de senaste åren gjorts på Atari 2600 spel. Dessa omgivningar testar en agents förmåga på en mängd olika utmaningar, och har används för att skapa agenter med bättre problemlösningsförmåga. Dessa spel har också fördelen att de är relativt enkla att implementera.

Denna masteruppsats introducerar en modifierad version av Atari 2600 spel inom vilket agenten inte har tillgång till hela spelets bildskärm utan enbart en rörlig reducerad del. Agenten måste därmed lära sig observera omgivningen på bästa möjliga sätt så att den även kan lära sig spela så bra som möjligt.

Agenter som lyckas prestera på dessa modifierade Atari 2600 spel kan möjligtvis appliceras på verkliga omgivningar som kräver samma typ av adaptiv observering. Detta skulle, till exempel, kunna vara inom robotik, där en robot använder sig av kameror för att känna av sin omgivning.

Denna masteruppsats introducerar även en metod kallad Myopic Deep Q-Learning (My-DQL), som gör det möjligt för en Atari agent att samtidigt lära sig observera och spela från sina erfarenheter.

Det visar sig att en sådan agent, som enbart har 35% av bildskärmen synlig, kan lära sig spela nästan optimalt på Atari 2600 spel som Pong, Breakout, och Space Invaders.

Contents

List of acronyms

MDP	Markov Decision Process
DQL	Deep Q-Learning
DQN	Deep Q-Network
PER	Prioritized Experience Replay
DPER	Distributed Prioritized Experience Replay
LSTM	Long Short Term Memory
MyDQL	Myopic Deep Q-Learning
MyDQN	Myopic Deep Q-Network
MyDRQN	Myopic Deep Recurrent Q-Network

List of Figures

List of Tables

1 Introduction

The theory of reinforcement learning describes a computational approach to learning from interactions. It describes an agent that interacts with an environment and receives numerical rewards. The agent is tasked to learn a high rewarding behavior from its experiences, such that it acts in a way that gives it the most amount of reward in the long run.

Reinforcement learning has recently gained in popularity due to its super-human performance in board games such as Chess, Shogi, and Go [1]. In these board games, the agent played solely against itself, and not long after it started learning it reached super-human performance.

Successful applications of reinforcement learning to artificial tasks are plenty, but successful applications to real-world tasks are few. One of the reasons is because most reinforcement learning theory needs the environment to satisfy a set of assumptions, which is often not completely satisfied in real-world tasks.

A popular way to gauge reinforcement learning agents on a wide variety of tasks is in the Arcade Learning Environment [2]. This is a suite of Atari 2600 games representing a broad range of challenges for the agent to master. This suite has been used to benchmark the progress of building successively more capable agents. This suit is popular due to its wide variety of tasks and its simple frame-work, which makes it easy to implement and analyze different agents behavior. However, they lack some of the challenges that real-world environments usually have, such as only having partial access to the state of the environment, and thus having to learn and act with incomplete information.

This thesis introduces Atari 2600 games with an added adaptive sensing task. In a normal Atari setting, the agents make decisions from observations of the whole screen, but with the adaptive sensing task, the agents are near-sighted in a two-dimensional sense. This means that only a small portion of the screen is visible at any given time. The myopic agent is able to make decisions about how to move the visible portion and is tasked to learn not only how to play the game, but also how to sense the state of the game. In this scenario, the agent is similar to a human player that has to decide where to focus its eyes on the screen of the game.

This modification of Atari environments allows for simple testing of different reinforcement learning algorithms over a wide range of tasks that also require adaptive sensing. Finding the best algorithm for these tasks could be useful in similar real-world tasks, for example in robotics, where visual inputs from cameras are used for navigation.

To solve the task of playing the Atari games while actively observing the screen, a slightly modified version of Deep Q-learning is proposed, referred to as Myopic Deep Q-learning (MyDQL). This algorithm is able to simultaneously learn a controller for the character in the game, as well as a controller for observing the game. This thesis introduces two different neural network architectures for MyDQL, a feed-forward architecture referred to as a Myopic Deep Q-Network (MyDQN), and a recurrent architecture referred to as a

Myopic Deep Recurrent Q-Network (MyDRQN). The architecture of MyDQN does not account for the partial observability of the environment, while MyDRQN is indirectly able to through its recurrent nature.

The contents of this thesis are an introduction to the theory of reinforcement learning, from the fundamentals to Deep Q-learning and its latest improvements, as well as an introduction to the Atari games that are used and the procedure that adds vision. Lastly, the MyDQL algorithm is introduced along with the MyDQN and MyDRQN architectures, as well as the performances they are able to achieve on the modified Atari games.

2 Reinforcement Learning

By looking at how animals learn, it can be seen that a large part of learning is done without an explicit teacher, but instead through interaction with the environment. A typical example is that of a new-born Gazelle that initially struggles to walk, but after about half an hour of trying, is able to run with speed.

The idea of reinforcement learning is to describe a computational approach to learning from interactions. This approach is not concerned with precisely how animals learn, but instead to learning from interactions in general. Reinforcement learning defines an agent that is able to sense the state of the environment and take actions which affects it. The agent also receives numerical rewards for the actions it takes, and is tasked to learn a policy, a map between situations and actions, such that it maximizes the amount of rewards it receives in the future. Reinforcement learning is an unsupervised approach to learning, which means that the agent is never told what actions to take, but has to learn this through trial and error.

In each moment, the agent senses the state of the environment and decides on the next action to take. The agent then performs this action and observes the environment transition to a new state and receives a reward related to that transition. This process continues and the agent gathers an increasing amount of experiences. It is from these experiences that the agent is able to incrementally adjust its behavior such that its actions become rewarding in the long run.

A depiction of the agent-environment interface can be seen in Fig. 1.

To use reinforcement learning to solve a task is to design a reward function such that the agent's goal of maximizing the reward signal leads to a behavior that solves the task. It is often much easier to design a reward function than to directly design a behavior.

Almost all reinforcement learning tasks can be formalized in terms of optimal control of a Markov Decision Process. But before its introduction, it is reasonable to talk about the Markov Chain and the Markov Reward Process, from which it is constructed.

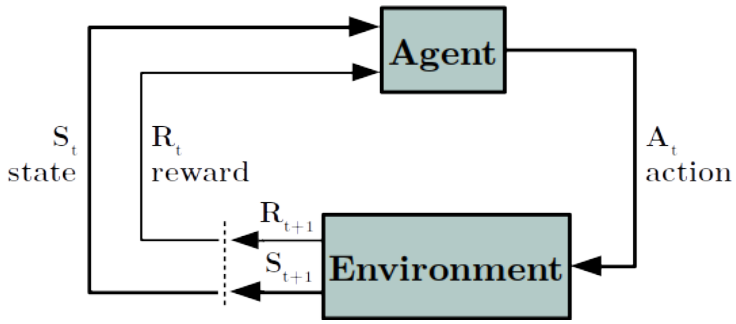


Figure 1: The agent-environment interface.

2.1 Markov Chain

A Markov chain is a stochastic time-discrete process that describes the transition dynamics of an environment between a finite set of states. An important property of a Markov chain is the Markov property, which says that future states of the environment are independent of the past given the present. This means that the probability to transition from state s to state s' is only conditional on the current state, which is alternatively stated as

$$\mathbb{P}\left[S_{t+1} = s' | S_1 = s_1, S_2 = s_2, ..., S_t = s\right] = \mathbb{P}\left[S_{t+1} = s' | S_t = s\right]. \tag{2.1}$$

Where S_t is a stochastic variable representing the state at time t.

In mathematical terms, a Markov chain is a tuple $\langle \mathcal{S}, \mathcal{P} \rangle$, where $\mathcal{S}$ is a finite set of states and $\mathcal{P}$ is the transition probability matrix $\mathcal{P}_{ss'} = \mathbb{P}\left[S_{t+1} = s' | S_t = s\right]$.

A Markov chain can be cyclic, or non-cyclic. If it is cyclic will continue indefinitely, while if it is non-cyclic at least one terminal state exists, such that the process terminates if that state is encountered.

To give an example of a cyclic Markov Chain, consider a simplified weather model depicted in Fig. 2. In this Markov chain, the environment has only three states, which are three weather conditions: sunny, cloudy, and rainy. Each day the environment transitions between these states with a probability that is only dependent on the current weather condition. If these transition probabilities are determined, possibly by observing many transitions, then predictions about future states of the environment could be made for any day. In other words, it would be possible to make weather forecasts.

To help the transition to Markov Decision Processes it is useful to imagine an observing agent at each time-step of the Markov chain. In this case, the agent is not able to impact the environment but only observe as it transition from one state to another, which is nearly the case for humans and the weather. To continue further, it is possible to give the observing agent a numerical reward when the environment transitions between states. In this case, the rewards can be seen as the observing agent's sentiment of the transitions, and the process is now known as a Markov Reward Process.

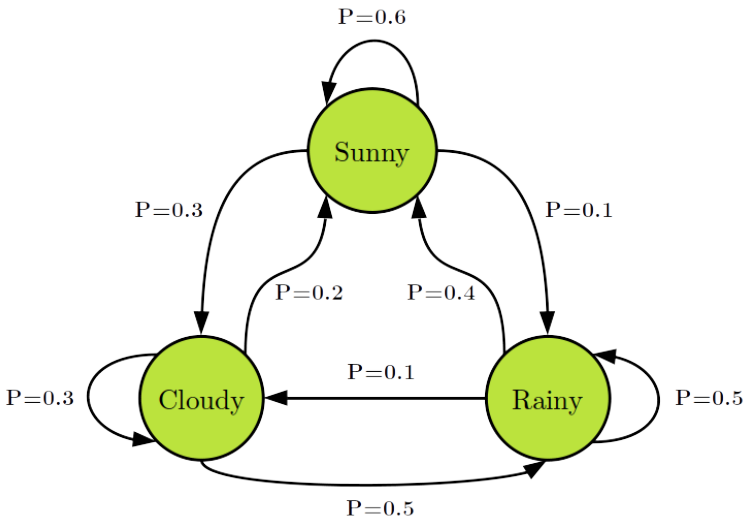


Figure 2: A Markov chain as a simple weather model.

2.1.1 Markov Reward Process

A time-discrete Markov Reward Process (MRP) is a Markov chain with numerical rewards associated with transitions. In mathematical terms an MRP is a tuple $\langle \mathcal{S}, \mathcal{P}, \mathcal{R}, \gamma \rangle$, where $\mathcal{S}$ is a finite set of states and $\mathcal{P}$ is the transition probability matrix, while $\mathcal{R}_s = \mathbb{E}[R_{t+1}|S_t = s]$ is the reward function and represents the expected next reward given the current state. The $\gamma \in [0, 1]$ is a discount factor, which is introduced for multiple reasons, and will be explained in detail later in this section and the section covering Markov Decision Processes.

Taking the point of view of the observing agent again, with the environment being in state S_t at time t. At the next time-step the agent observes the environment transition to state S_{t+1} and receives a reward R_{t+1}. This process continues indefinitely or ends at a terminal state, which results in a trajectory of experiences for the agent: $\tau(S_t) = \{S_t, R_{t+1}, S_{t+1}, R_{t+2}, \dots\}$.

It is useful to introduce the return G_t, which is defined as the total amount of discounted reward along a trajectory of experiences. In other words, the return can be written as

$$G_t = R_{t+1} + \gamma R_{t+2} + \gamma^2 R_{t+3} + \cdots = \sum_{k=0}^{\infty} \gamma^k R_{k+t+1}. \tag{2.2}$$

The discount factor $\gamma \in [0, 1]$ serves to prevent diverging returns, which can occur if the MRP is cyclic.

By using the definition of the return $G(t)$, it is possible to define the value of a state $v(s)$, as follows

$$v(s) = \mathbb{E}[G_t|S_t = s]. \tag{2.3}$$

In other words, the value of a state is the expected return from that state. It is useful to note that by this definition, the value of a terminal state is always zero.

It is possible to rewrite $v(s)$ into what is known as a Bellman equation:

$$\begin{aligned} v(s) &= \mathbb{E}[G_t|S_t = s] \\ &= \mathbb{E}[\sum_{k=0}^{\infty} \gamma^k R_{k+t+1}] \\ &= \mathbb{E}[R_{t+1} + \gamma \sum_{k=0}^{\infty} \gamma^k R_{k+t+2}|S_t = s] \\ &= \mathcal{R}_s + \gamma \sum_{s \in \mathcal{S}} \mathcal{P}_{ss'} v(s'). \end{aligned}$$

This means that the value function can be also be though of as the expected immediate reward plus the discounted expected value of the next state.

The relation of the value function to itself is at the core of temporal-difference techniques that can be used to estimate it. However, the Bellman equation for the value function of an MRP is linear, and can be represented in terms of matrices as follows

$$\begin{bmatrix} v_1 \\ v_2 \\ \vdots \\ v_n \end{bmatrix} = \begin{bmatrix} \mathcal{R}_1 \\ \mathcal{R}_2 \\ \vdots \\ \mathcal{R}_n \end{bmatrix} + \gamma \begin{bmatrix} \mathcal{P}_{11} & \mathcal{P}_{12} & \dots & \mathcal{P}_{1n} \\ \mathcal{P}_{21} & \mathcal{P}_{22} & \dots & \mathcal{P}_{2n} \\ \vdots & \vdots & \ddots & \vdots \\ \mathcal{P}_{n1} & \mathcal{P}_{n2} & \dots & \mathcal{P}_{nn} \end{bmatrix} \begin{bmatrix} v_1 \\ v_2 \\ \dots \\ v_n \end{bmatrix},$$

where the indices represents all states, and n is the total number of states. This matrix equation is more neatly written as

$$v = \mathcal{R} + \gamma \mathcal{P} v.$$

The solution of the value function can be found from the above by simple linear algebra:

$$v = (I - \gamma \mathcal{P})^{-1} \mathcal{R}.$$

If the transition probabilities and the reward function are known, it is straight forward to calculate the value function. However, many environments have a large number of states, which makes the matrix inversion computationally intractable. In many cases one is forced to use iterative methods, such as the previously mentioned temporal-difference techniques.

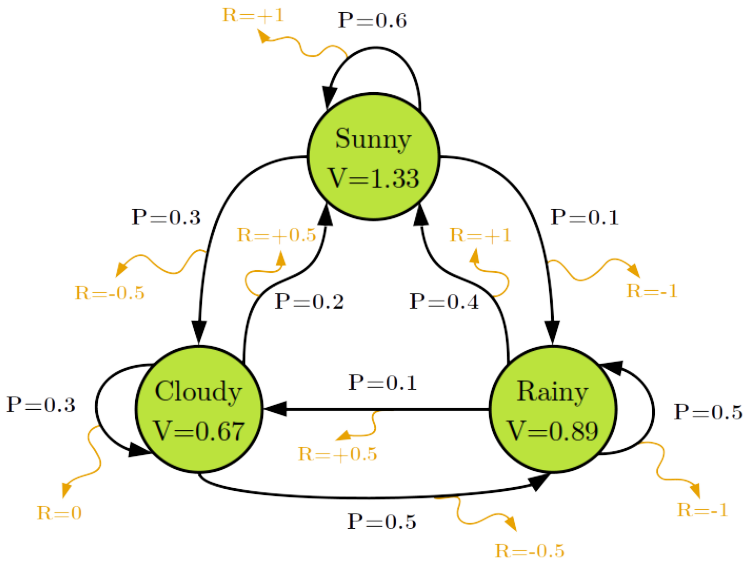


Figure 3: A Markov Reward Process for an agent observing the weather. The value of each state is represented by V and is calculated using $\gamma = 0.9$.

As an example of an MRP consider the simple weather model again, but this time with rewards between transitions that describe the sentiment of the observing agent. The rewards are constructed such that the agent likes when the weather transitions to a sunny day, and is rather passive about transitions to cloudy days, while it dislikes transitions to rainy days. Using a reward horizon of about 10 days, which means $\gamma = 0.9$, makes the state values give a measure of how good the agent is expected to feel in the following 10 days. A depiction of this MRP is given in Fig. 3.

Until this point, there has been no agency involved. In other words, the probabilities of the environment transitioning to future states have only had one condition, the present state. By allowing the agent to also take actions that can affect the transition to future states, turns the MRP into a Markov Decision Process.

2.1.2 Markov Decision Process

A Markov Decision Processes (MDP) is an MRP in which an agent is able to make decisions and take actions. These actions affect how the environments transitions between states, which means that future states are not only conditional on the current state, but also on the action taken by the agent. In mathematical terms, an MDP is a tuple $\langle \mathcal{S}, \mathcal{A}, \mathcal{P}, \mathcal{R}, \gamma \rangle$, where $\mathcal{S}$ is a finite set of states, $\mathcal{A}$ is a finite set of actions, and $\mathcal{P}$ is the transition probability matrix: $\mathcal{P}^a_{ss'} = \mathbb{P}\left[S_{t+1} = s' | S_t = s, A_t = a\right]$, and lastly $\gamma \in [0, 1]$ is a discount factor.

A typical example of an MDP is a robot that is tasked to roam around and collect cans while avoiding running out of battery. The robot's environment is in this case simply its

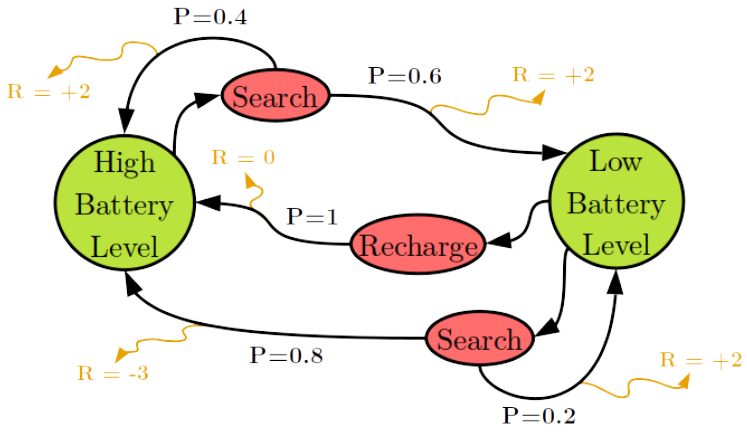


Figure 4: An MDP describing a decision making robot that is tasked to collect cans.

battery, which has two states: High Battery Level and Low Battery Level. The robot has to make decisions about either initiating a search protocol for the next time-step, where the robot continues to roam around in search of cans or to initiate a recharge protocol, where the robot returns to the charging station to charge its battery. The reward function is structured such that the robot gets rewarded for finding cans, while punished for running out of battery. If the robot initiates a search protocol at Low Battery Level and ends up at High Battery Level at the next step, it means that the robot ran out of battery and had to be rescued. A depiction of this MDP can be seen in Fig. 4.

As previously mentioned, almost all reinforcement learning tasks can be formalized in terms of optimal control of an MDP. A reinforcement learning task is considered solved when the agent has a behavior that leads to a lot of reward in the long run. But before talking about to how to find such behaviors, a couple of definitions has to be made.

As the agent is able to make decisions, it is useful to define its behavior in terms of a policy π, which is is a probability distribution over actions given states, defined as follows

$$\pi(a|s) = \mathbb{P}[A_t = a|S_t = s], \tag{2.4}$$

such that agents policy fully characterizes its behavior.

Furthermore, it is possible to redefine the transition probabilities and expected rewards based on the agents policy π:

$$\mathcal{P}^{\pi}_{ss'} = \sum_{a\in\mathcal{A}} \pi(a|s)\mathcal{P}^{a}_{ss'},$$
$$\mathcal{R}^{\pi}_{s} = \sum_{a\in\mathcal{A}} \pi(a|s)\mathcal{R}^{a}_{s}.$$

By doing this it is easy to identify the tuple $\langle \mathcal{S}, \mathcal{P}^{\pi}, \mathcal{R}^{\pi}, \gamma \rangle$ as an MRP.

The value function of an MDP is defined as the future expected return starting in a given state and following policy π:

$$v_\pi(s) = \mathbb{E}_\pi[G_t|S_t = s]. \tag{2.5}$$

Furthermore, it is useful to define an action-value function $q_\pi(s, a)$ as the expected return starting in state s and taking action a, and then following policy π. In other words,

$$q_\pi(s, a) = \mathbb{E}_\pi[G_t|S_t = s, A_t = a]. \tag{2.6}$$

The action-value function will serve to be quite useful as it gives a way of evaluating all possible actions at a given state.

Both the value function and action-value function can be represented in terms of Bellman equations:

$$v_\pi(s) = \mathcal{R}_s^\pi + \gamma \sum_{s' \in \mathcal{S}} \mathcal{P}_{ss'}^\pi v_\pi(s'),$$

and similarly for the action-value function,

$$q_\pi(s, a) = \mathcal{R}_s^a + \gamma \sum_{s' \in \mathcal{S}} \mathcal{P}_{ss'}^a \sum_{a' \in \mathcal{A}} \pi(s', a') q_\pi(s', a').$$

For an MDP it is possible to precisely define an optimal policy, as the value functions can serve as a partial ordering over the policies. This means that π is defined to be better than $\pi^{'}$ if the corresponding value function v_π is greater than, or equal to $v_{\pi'}$, in every state. In other words,

$$\pi \geq \pi' \text{ if } v_\pi(s) \geq v_{\pi'}(s),\ \forall s \in \mathcal{S} \tag{2.7}$$

The optimal policy π_* is thus defined as the policy for which $\pi_* \geq \pi,\ \forall \pi$.

Furthermore, an optimal value function v_* is defined as the value function that is greater than, or equal to, any other value function, in every state. In other words,

$$v_*(s) = \max_\pi v_\pi(s). \tag{2.8}$$

Similarly, the optimal action-value function q_* is defined as

$$q_*(s, a) = \max_\pi q_\pi(s, a). \tag{2.9}$$

It can be shown that for any Markov Decision Process

- There exists an optimal policy π_* that satisfies $\pi_* \geq \pi,\ \forall \pi$.
- The value function of an optimal policy is an optimal value function, $v_{\pi_*}(s) = v_*(s)$.
- The action-value function of an optimal policy is an optimal action-value function, $q_{\pi_*}(s,a) = q_*(s,a)$.

If an optimal action-value function is known, it would be easy to extract a fully deterministic optimal policy, as follows

$$\pi_*(a|s) = \begin{cases} 1 & \text{if } a = \arg\max_{a \in \mathcal{A}} q_*(s,a), \\ 0 & \text{otherwise.} \end{cases} \tag{2.10}$$

Furthermore, there are Bellman equations related to the optimal value and action-value functions, known as Bellman optimality equations

$$v_*(s) = \max_a \mathcal{R}_s^a + \gamma \sum_{s' \in S} \mathcal{P}_{ss'}^a v_*(s'),$$
$$q_*(s,a) = \mathcal{R}_s^a + \gamma \sum_{s' \in S} \mathcal{P}_{ss'}^a \max_{a'} q_*(s',a').$$

These equations are typically the most important in finding the optimal policy. However, they are non-linear and has in general no closed-form solutions. Fortunately, there are iterative methods that can solve these equations, such as: Value iteration, Policy iteration, SARSA, and Q-learning [3]. Only Q-learning is relevant for this project and is introduced in the next section.

2.2 Q-learning

A breakthrough in reinforcement learning came with the discovery of the Q-learning algorithm [4]. This is an iterative method that solves the Bellman optimality equation for the action-value function.

As previously mentioned, there are other iterative methods to solve the Bellman optimality equations, but Q-learning has many advantages over the others. Value iteration and Policy iteration requires many sweeps over all states, which makes them impractical for large environments. Both SARSA and Q-learning use the experiences of trajectories directly and are therefore independent of the size of the state-space. Furthermore, SARSA and Q-learning are quite similar, but the difference is that Q-learning can update along trajectories

that are not generated by the agent's policy, as it is directly approximating the optimal action-value function. Being able to gather experiences with a different policy than the agent's policy can be quite beneficial, as will be shown later.

The Q-learning algorithm updates an initially arbitrary action-value function Q by experiences generated from some policy acting in the environment, not necessarily the agent's policy. Starting in state S_t an action A_t is selected from the experience gathering policy and the environment transitions to state S_{t+1}, and the agent receives a reward R_{t+1}. This sequence of events can be represented as an experience $e_t = (S_t, A_t, R_{t+1}, S_{t+1})$. By taking experiences like these, the Q-learning algorithm can update the previous action-value estimate $Q(S_t, A_t)$ in the following manner

$$Q(S_t, A_t) \leftarrow Q(S_t, A_t) + \alpha \left[R_{t+1} + \gamma \max_a Q(S_{t+1}, a) - Q(S_t, A_t) \right], \tag{2.11}$$

where α is a step-size parameter controlling the amount of change at each iteration. If the update rule converges, the action-value estimate satisfies the Bellman optimality equation and must therefore be an optimal action-value function. The exact convergence guarantees are discussed below.

2.2.1 Convergence Guarantees

The only requirement to guarantee convergence of the Q-learning algorithm is that every state-action pair is continually visited and that the step-size α is appropriately small. If those requirements are satisfied Q-learning converges with probability 1 to q_* after an infinite amount of updates.

The convergence guarantees are comforting but are rarely useful in practice. Having to visit every state-action pair an infinite amount of times is usually not needed for getting a decent approximation of the optimal policy. This is possibly because state-action pairs that are far from the trajectories of an optimal policy are less important to estimate correctly, as an optimal agent is unlikely to visit those anyway.

2.2.2 Exploration-Exploitation Trade-off

There is a problem with learning from experiences that is fundamental to reinforcement learning, and that is the problem of exploration-exploitation trade-off. As the agent is only learning from experiences, the policy gathering experiences has to be exploratory enough such that the agent can learn the differences between each state-action pair, while it also has to exploit what it has learned to keep improving. If the policy is fully exploratory, i.e. completely random, then it might not be possible to reach states which require the agent to avoid many terminal states on the way. On the other hand, if the policy is fully greedy, which means that it only selects actions it currently believes are the most valuable, it might never learn that other actions are better in the long run. This means that the policy

gathering experiences need to balance between exploration and exploitation throughout the learning process.

A fully exploiting policy is to only take actions that maximizes the current Q-value estimate, i.e. actions defined by

$$a = \arg\max_{a \in \mathcal{A}} Q(s, a). \tag{2.12}$$

A fully exploratory policy would be one from which all actions are chosen uniformly at random.

Since Q-learning directly approximates the optimal action-value function, the Q-value target is independent of the experience gathering policy. This allows for experiences to be gathered by almost any policy, as long as each state-action pair has a non-zero probability of being selected. This means that the experience gathering policy can be designed rather freely, such that the degree of exploration can be controlled during training. One such policy is called an ϵ-greedy policy and is a simple mixture between a fully exploiting and a fully exploratory policy. The ϵ-greedy policy selects an action uniformly at random with probability ϵ and is otherwise greedy according to 2.12. The amount of exploration is, therefore, controlled by the parameter ϵ, which becomes an additional hyperparameter of the algorithm. This is a simple method but tends to work a lot better than being fully greedy.

But as can be imagined, it could be beneficial to have different amounts of exploration throughout the learning process. There are methods of dynamically adjusting ϵ and tends to work better than having one fixed exploration. However, it is also possible to have many policies gathering experiences, each with their own fixed exploration rate. This is the method that will be used in this thesis but is better introduced fully later on.

2.2.3 Representing the Q-value Function

There are different ways of representing the Q-value function in practice, one way is to use look-up tables, or arrays, to store the current action-value estimates. In such a table, the rows typically correspond to states and columns to actions. An example of a look-up table for a Q-value function can be seen in Table 1.

Unfortunately, using look-up tables is only feasible for environments in which the state-space is not too big. A lot of environments have large state-spaces, such as Atari games, and some even have enormous ones. As an example, the board game Backgammon has about 10^{20} different states. Fortunately, there now exists ways to deal with this problem, which recently reignited the popularity of reinforcement learning. For Q-learning it is the introduction of an artificial neural network, defining the method of Deep Q-Learning.

Table 1: An example of a look-up table for a Q-value function at iteration t, with a total of n actions and m states.

$\mathcal{S}/\mathcal{A}$	a_1	a_2	...	a_n
s_1	$Q_t(s_1, a_1) = 0.57$	$Q_t(s_1, a_2) = 1.56$	...	$Q_t(s_1, a_n) = 1.32$
s_2	$Q_t(s_2, a_1) = 0.22$	$Q_t(s_2, a_2) = 0.36$	...	$Q_t(s_2, a_n) = 1.66$
$\vdots$	$\vdots$	$\vdots$	$\ddots$	$\vdots$
s_m	$Q_t(s_m, a_1) = 2.15$	$Q_t(s_m, a_2) = 1.76$	...	$Q_t(s_m, a_n) = 3.27$

2.3 Deep Q-Learning

For many large state spaces, it is only feasible to represent the Q-value function by methods of function approximation. Such methods parametrize the action-value function and allow one update to affect more than one state-action pair. In Deep Q-Learning [5] an artificial neural network is used as a function approximator for the action-value function, which is also known as a Deep Q-Network (DQN). The input to a DQN is an array of numbers representing the current state of the environment, while the output is an array representing the corresponding Q-value estimates for all actions. In other words, at time-step t, the output of the DQN can be written as $Q_\theta(S_t, A_t)$, where θ represents the parameters of the network.

In Deep Q-learning the updates of the Q-values are made indirectly through stochastic gradient descent using temporal-difference (TD) errors:

$$\delta_t = R_{t+1} + \gamma \max_a Q_\theta(S_{t+1}, a) - Q_\theta(S_t, A_t). \tag{2.13}$$

This shifts the parameters of the network such that its output moves in the direction of the optimal action-value function.

Exactly how long it will take to converge, and how close the action-value function will be to the optimal one, depends now also on the architecture of the network and its hyperparameters.

As painless as Deep Q-learning may seem, the use of an artificial neural network comes with many problems. Most of these problems, along with the methods and tricks used to solve them, are introduced below.

2.3.1 Fixed Target Network

Since the updates are made on the parameters of the network, all Q-value estimates for every state are shifted at the same time. How large the shift is for each estimate depends on the similarity between their state and the one used in the update. This leads to a

problem as states that are consecutive in a trajectory are likely to be similar, but both are used to calculate one TD-error. Therefore, as the parameters of the DQN are updated, both the action-value estimate $Q_\theta(S_t, A_t)$ and its target: $R_{t+1} + \gamma \max_a Q_\theta(S_{t+1}, a)$, are respectively shifted in a similar way. This results in moving Q-value targets, which is often leads to instability.

A part of this problem can be solved rather simply and that is by introducing an additional network tasked to only calculate the Q-value targets. This target network is a fixed copy of the original network, which is from now on referred to as an online network. The parameters of the target network are only synchronized with the online network after a certain amount of updates have been made. The frequency at which one should synchronize the parameters is another hyperparameter of the algorithm. This defines the method of fixed target networks [6].

Using a fixed target network changes the TD-errors in the following manner

$$\delta_t = R_{t+1} + \gamma \max_a Q_{\theta'}(S_{t+1}, a) - Q_\theta(S_t, A_t), \tag{2.14}$$

where θ' represents the parameters of the target network.

The use of a fixed target network keeps the Q-value targets fixed for a period of time, which tends to significantly improve the learning process.

2.3.2 Double Deep Q-Network

Due to the nature of the Q-learning, and especially Deep Q-learning, a bias exists in the early stages of learning. It is possible to show that noise in the Q-value targets produces an overestimation bias, which can lead to sub-optimal policies or even poor performance. The source of the overestimation lies in always taking the largest Q-value estimate for the Q-value target.

Fortunately, there is a way to reduce this bias by utilizing the above method of a fixed target network. This works by decoupling action selection and action evaluation by using both the online network and the target network. The online network is tasked to select the action for the Q-value target, while the target network is used to evaluate it. This defined the method of Double Deep Q-Networks [7].

Using this method changes the TD-errors to:

$$\delta_t = R_{t+1} + \gamma Q_{\theta'}(S_{t+1}, \arg\max_{a \in \mathcal{A}} Q_\theta(S_{t+1}, a)) - Q_\theta(S_t, A_t). \tag{2.15}$$

This helps to reduce the bias and tends to improve the learning process.

2.3.3 Experience Replay

The experiences gathered by the agent are normally used to update the parameters in an online fashion. But this can be inefficient as consecutive experiences tend to be highly correlated. However, it is possible to reduce the correlation between consecutive updates by introducing a memory that temporarily stores experiences, called a replay memory [8]. This allows for a batch of experiences to be sampled randomly and their average TD-error is used for one update. Besides, it allows for rare experiences to be sampled multiple times, which can improve learning significantly.

The size of the replay memory can surely not be indefinitely large as one would eventually run out of computer memory. Therefore, after a size limit is reached, new experiences should replace old experiences. In any case, having a very large replay memory comes with the downside that old experiences tend to be less relevant to the current policy. Thus, the size of the replay memory is another hyperparameter that needs to be tuned.

2.3.4 Prioritized Experience Replay

Sampling uniformly from the replay memory ignores the fact that some experiences are more important for learning than others. Therefore, it would be reasonable to have a method that labels each experience with their importance, or priority, such that they can be sampled accordingly. Such a measure can be constructed from the magnitudes of the temporal differences, as is done in the method of Prioritized Experience Replay (PER) [9].

In PER experiences are not sampled uniformly anymore, but with respect to a distribution based on the priorities, which in turn are based on the temporal difference magnitudes. The priority of the i-th experience in memory can be written as $p_i = |\delta_i| + \epsilon$, where δ_i is the corresponding temporal difference, and ϵ is a small positive value that keeps the sampling probability non-zero. The sampling distribution is constructed as follows

$$P(i) = \frac{p_i^\alpha}{\sum_j p_j^\alpha}, \tag{2.16}$$

where $\alpha > 0$ is a parameter used to interpolate between a uniform and purely greedy distribution.

Sampling with a non-uniform distribution introduces an additional bias to the Q-value estimates. This bias can be mitigated by multiplying the TD-errors with importance sampling weights:

$$\omega_i = \left(\frac{1}{N} \cdot \frac{1}{P(i)} \right)^\beta, \tag{2.17}$$

where N is the size of the replay memory, and $\beta \in [0, 1]$ is a parameter controlling the bias reduction. The β parameter is introduced with the idea that accounting for the bias

is most important at the end of training. Thus, incrementally increasing β towards unity can be beneficial.

Furthermore, for stability reasons, the ω_i's are normalized by multiplying with $1/\max_j \omega_j$. This makes sure that the TD-errors are only scaled downwards.

Putting all of this together, when using PER the TD-errors will be changed as follows

$$\delta_i \rightarrow \delta_i \cdot (\omega_i / \max_j \omega_j). \tag{2.18}$$

2.3.5 Distributed Prioritized Experience Replay

Another improvement to Deep Q-learning that is central to this project is Distributed Prioritized Experience Replay (DPER) [10]. This completely decouples learning and gathering experiences by scaling Deep Q-learning using PER to multiple computing resources, which can speed up learning significantly.

In the distributed setting, an agent is split into a learner and multiple actors, that all share a prioritized replay memory. The actors and learner each have their own copy of the original network. The task of the actors is to gather experiences and calculate their initial priorities, and then store these experiences and priorities into the shared replay memory. The task of the learner is to sample experiences from the shared replay memory and update the parameters, and at the same time update the priorities of the sampled experiences.

The learner synchronizes its networks weights with the actors after a set amount of updates, which is another hyperparameter of the algorithm.

In the distributed setting it is beneficial to give each actor their own exploration rate, such that the combined set of actors can explore both narrowly and broadly at any stage of the learning process. This can be done by giving the actors their own ϵ-greedy policy, which are usually assigned such that some actors explore broadly, while most explore narrowly. Having some of the actors explore broadly keeps the learner from overfitting to certain trajectories, while many narrow explorations makes sure to gather more experiences deeper into the environment.

The DPER paper shows that using a large number of actors can greatly improve the final performance of the agent.

2.3.6 LSTMs and Distributed Prioritized Experience Replay

Using DPER with networks that contain recurrent layers, such as a Long Short-Term Memory (LSTM) [11], means that some modifications has to be made to the set-up. This thesis follows some of the methods introduced for R2D2 [12].

LSTMs have feedback connections that enables them to store information about past inputs in a hidden state, which effectively gives the agent a short-term memory.

Most of the modifications to DPER concern the hidden states of the LSTMs. These hidden states have to be stored along with the experiences, such that the learner can initialize its LSTMs correctly. Furthermore, the learner needs to unroll the LSTMs on sequences of consecutive experiences to be able to train the LSTMs properly. This means that the actors have to store experience locally and combine those to sequences of experiences. The length of the sequence is treated as an additional hyperparameter of the algorithm.

Since the learner samples sequences of experiences instead of single experiences a new definition of the priorities has to be made. A naive approach would be to make the priority of the sequence the average magnitude of the TD-errors from its experiences. However, as the the authors of R2D2 points out, averaging over long sequences tends to equalize the priorities too much. Instead, one can use a mixture of the maximum and average TD-error magnitude, in the following manner

$$P(i) = \eta \max_j |\delta_j| + (1 - \eta)|\bar{\delta}|, \tag{2.19}$$

where $P(i)$ represents the priority of the i-th sequence, and $\max_j \delta_j$ and $\bar{\delta}$ are the maximum and average of TD-errors in that sequence respectively, and $\eta \in [0, 1]$ is a parameter interpolating the priority between the two.

Lastly, the stored hidden states can become outdated with respect to the current policy, which inhibits the learning process. However, the hidden states of LSTMs tend to partially recover after some initial steps into a sequence. Thus, the beginning of a sequence can be used as a burn-in stage, that is not used in updates, such that the hidden states are able to partially recover from outdated ones. This introduces another hyperparameter to the algorithm, and that is the length of the burn-in stage.

With the above-mentioned modifications for LSTMs, the distributed setting is otherwise the same.

2.4 Environments

This section is an introduction to the unmodified environments used in this thesis. They are Atari 2600 games, which is a popular choice for reinforcement learning research. The games used in this thesis are Pong, Breakout and Space Invaders, and are implemented through the OpenAI Gym [13].

2.4.1 Pong

This environment simulates a simple game of table tennis. The agent controls a paddle on the right side of the screen moving it vertically up and down, while the left side paddle is controlled by a hard-coded opponent. A typical screen of the game can be seen in Fig. 5.

To score a point in Pong one has to shoot the ball such that the opponent is unable to return it. The one who first gets to 21 points wins.

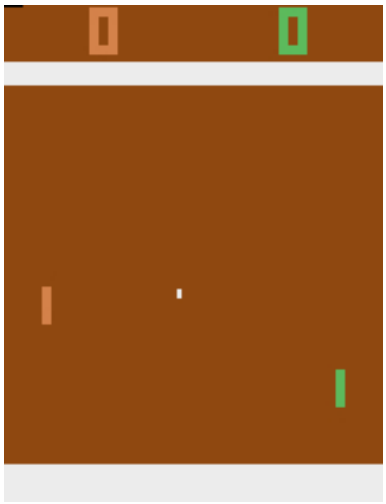


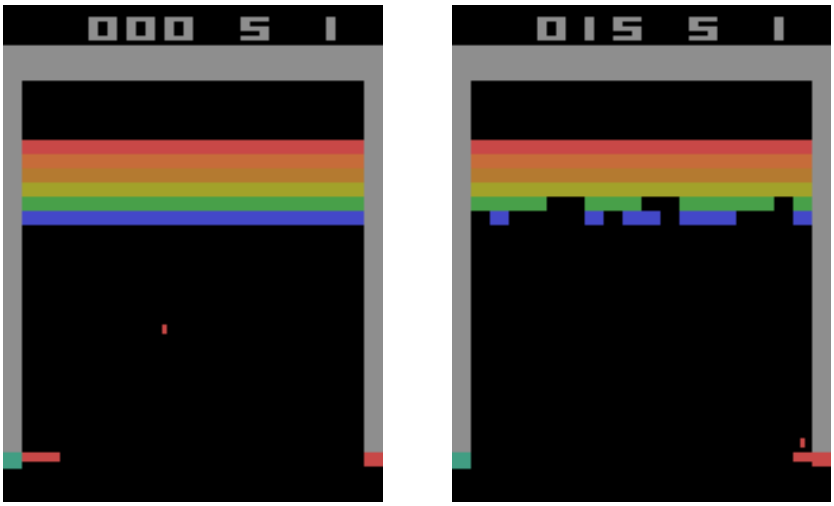
Figure 5: A typical screen of Pong.

The reward function is in accordance with the game score from the agents perspective, such that a reward of +1 is given every time the agent scores, and a reward of -1 every time the opponent scores. The agent receives a reward of 0 at any time-step in-between scoring.

This environment has three actions for movement: move up, move down, stand still.

2.4.2 Breakout

This environment is similar to Pong, but there is no opponent. The agent controls a paddle at the bottom part of the screen, and at the top part of the screen, there are six rows of bricks. The goal is to destroy them all by hitting them with the ball. As the ball hits one



$t = 27.$ $t = 512.$

Figure 6: Two typical screens of Breakout at different time-steps.

of the bricks it rebounds, and some points are awarded to the agent. Two typical screens of the game can be seen in Fig. 6. If the ball hits one of the middle rows, it speeds up, increasing the game difficulty. If the agent misses the rebound, one out of five lives are lost, and a new ball is spawned. The game is over when all lives are lost.

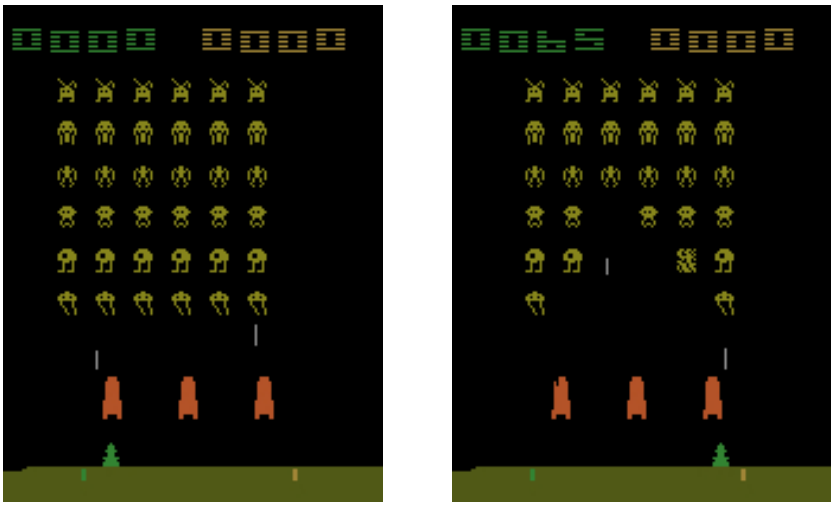
To completely beat the game the rows have to be cleared twice, as the first clearing resets all rows. A total score of 432 is given for clearing the rows once, which makes the maximum score of one episode 864.

The reward function is again in accordance with the game score from the agents perspective. A reward of +1 is given for hitting any brick on the first and second row, while the third row and fourth row gives +4, and the fifth row and sixth row gives +7. As before, a reward of 0 is given for any time-step in-between.

There are again three actions for movement: move left, move right, stand still. But, this time there is one extra action to initiate the ball after a life is lost, and to start the game.

2.4.3 Space Invaders

This environment is quite different from the other two previously mentioned. In this game the agent controls a laser cannon and is tasked to shoot down aliens before they succeed in invading. Two typical screens of the game are shown in Fig. 7. As the game progresses the aliens move horizontally and descend one step as the formation reaches the side of the screen. All aliens have laser cannons themselves, but the agent can find shelter behind bunkers that gradually gets destroyed. If the agent gets hit, one out of three lives are lost, and the laser cannon is reset to the starting position. The game is over once all three lives are lost, or if the alien formation succeeds the invasion by reaching the ground.



t = 29. t = 95.

Figure 7: Two typical screens of Space Invaders at different time-steps.

Space Invaders has three different levels. These only differ in that the alien formation starts ascending from a lower position on the screen, which increases the difficulty after each level. After all the aliens have been destroyed in one level, the agent advances to the next level. However, when the third level is cleared, it advances into itself. This means that the third level can continue indefinitely and that there is no maximum score.

The reward function is again in accordance to the awarded points. Each time an alien gets destroyed, an amount of points is given in relation to the aliens vertical position in its formation. The lowest row of aliens give +5 points and the row one step above gives +10 points, which continues to a total of +30 points for destroying an alien on the last row. Additionally, an alien mothership worth +300 points can enter the top of the screen after some time.

There are a total of six actions for this environment. Three of these actions are purely movement: move left, move right, stand still. The other three actions combine the previous actions with firing the laser cannon: move left and fire, move right and fire, stand still and fire.

2.4.4 Preprocessing and Modifications

The Atari games from the OpenAI Gym have screen sizes of width 160 pixels, height 210 pixels, and 3 color dimensions with values in the range 0 to 255.

To represent the current state of the Atari game, it is standard to stack a number of consecutive screens, such that velocities and accelerations can be detected from moving objects. It is also standard to preprocess the screens to reduce the size of the state space, by shrinking the dimensions and converting them to gray-scale.

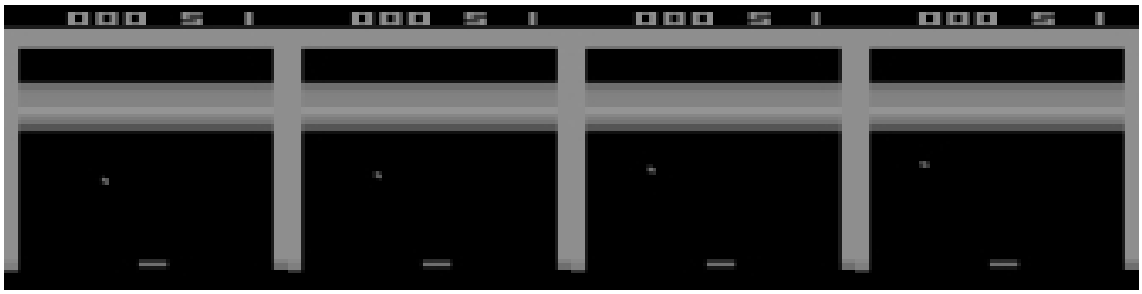


Figure 8: Four preprocessed screens representing one state of Breakout.

Furthermore, it is common to repeat the same action for several frames, such that the amount of state-action pairs are reduced. The two last consecutive screens from repeating the same action several times are usually combined into one, by taking the maximum pixel value at each position. This is done to remedy the screen flickering that exists in certain Atari games, such as Space Invaders.

An example of a fully preprocessed state of Breakout can be seen in Fig. 8.

It is standard is to clip all the rewards to either -1, 0 or 1, depending on their sign. This is done for reasons of stability and generality across all Atari games.

Furthermore, the next state after which the agent loses a life is counted as a terminal state in the Q-value targets. This helps the agent to faster learn the usefulness of a life.

Another standard modification is to automatically use the reset action after a loss of life, such that the agent can fully focus on learning how to play the game.

Lastly, as the Atari games are deterministic, a random number of no-operation actions, such as stand still, are taken at the beginning of each episode of the game. This helps to avoid overfitting and make the policies more robust.

3 Vision on Atari Games

The Atari games provide not only environments that are challenging and diverse, but also environments with states that are screens. This allows for the addition of an adaptive sensing task that is similar to actual vision.

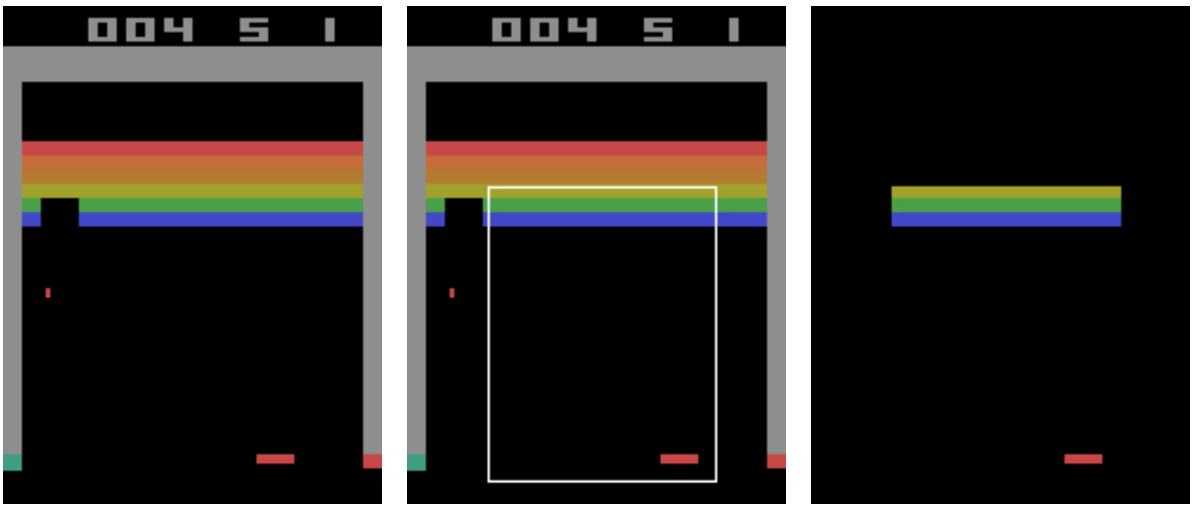
This thesis introduces vision to Atari games by modifying the screen to restrict the agent's view, and by adding visual actions such that the agent is able to move this view around. In this set-up, the agent takes the position of a player observing the screen of a video game with focused eyes, while at the same time playing it.

3.1 Modification of Screens

At each time-step, a screen is modified given a focal point, which determines the center of the agent's view, and a focal size, which determines how much the agent can see. The focal point is simply two integers specifying a pixel position on the screen. The focal size is also two integers, but they specify the visible area around the focal point, in terms of vertical and horizontal pixel lengths.

The visible part of the screen is calculated as a rectangular area with the focal point as a center. All the pixels outside of the focal area are kept on the screen but set to pixel value zero. This results in an agent that is narrow sighted in a two-dimensional sense, referred to as a myopic agent.

An example of the modification procedure to 35% visibility for a screen of Breakout can be seen in Fig. 9.



1. Original 2. With a focal frame 3. Modified

Figure 9: The modification of a screen of Breakout to 35% visibility.

3.2 Visual Actions

In this thesis the agents visual actions correspond to moving the focal point with a predetermined step-size, called a focal step-size. This step-size determines the speed at which the agent is able to move its vision, as well as the continuity between each observation.

There are a total of five visual actions for moving the focal point, which are: move up, move down, move left, move right, stand still.

At each time-step, the myopic agent observes a part of the screen and decides on the next action to move the character around, as well as the next visual action.

3.3 Partial Observability

As the modification procedure hides everything outside of the focal area, the states become partially observable. This means that the true state is hidden and that the Markov Property is violated to some degree. However, this thesis deals with an interesting dynamic to this partial observability, as every part of the true state is in theory available to the agent. This means that the agent might still be able to find policies for its vision such that the observations it sees violate the Markov Property the least. As an example, in Breakout and Pong, it is arguably more informative for the agent to observe the paddle and the ball, than only observing the displayed scoreboard at the top of the screen. The signal to noise ratio might, therefore, be large enough for a naive implementation of Deep Q-Learning.

However, it would certainly be useful to apply some of the methods that try to remedy the violation of the Markov Property, which will be introduced later on in the following section.

4 Myopic Deep Q-Learning

This thesis presents Myopic Deep Q-Learning (MyDQL), which can be used in environments with an adaptive sensing task to learn a high rewarding policy for discrete actions including sensing actions, such as the previously defined visual actions.

It is useful to define terminology to distinguish between sensing actions and actions that are able to directly affect the state of the environment. These actions are to be referred to as natural actions, and from this point on, all sensing actions will be referred to as visual actions.

4.1 Myopic Deep Q-Network

The Myopic Deep Q-Network (MyDQN) defines a near-sighted agent with the ability to control its vision. It uses a deep artificial neural network as a function approximator for the action-values of the natural actions and visual actions respectively.

The architecture of the network consists of a convolutional torso, which is a set of convolutional layers acting as a feature extractor, and two heads, which are fully connected layers that combine the extracted features to action-values. In other words, an observation of a state is first passed through multiple convolutional layers and then split into the fully connected layers of each head.

An overview of the MyDQN architecture can be seen in Fig. 10 and a detailed view can be found in the Appendix.

The separation of the natural and visual action-values in the architecture does not mean they are independent of each other, as will be explained below.

4.2 Natural Controller

The transition probabilities between observations depend on the current visual policy, which means that the natural head has to model vision as a part of its environment. Fortunately, this does not change the update rule of the natural head from the one of Deep Q-Learning, but is likely to make learning more difficult as the Q-value targets become even more non-stationary.

In other words, updates of the parameters of the natural head, including the convolutional torso, are made through stochastic gradient descent on batches of standard TD-errors:

$$\delta_t = R_{t+1} + \gamma \max_a Q^n_{\theta_n}(O_{t+1}, a) - Q^n_{\theta_n}(O_t, A_t), \tag{4.20}$$

where O_t is the observation at time t, and θ_n represents the parameters of the natural head as well as the convolutional torso.

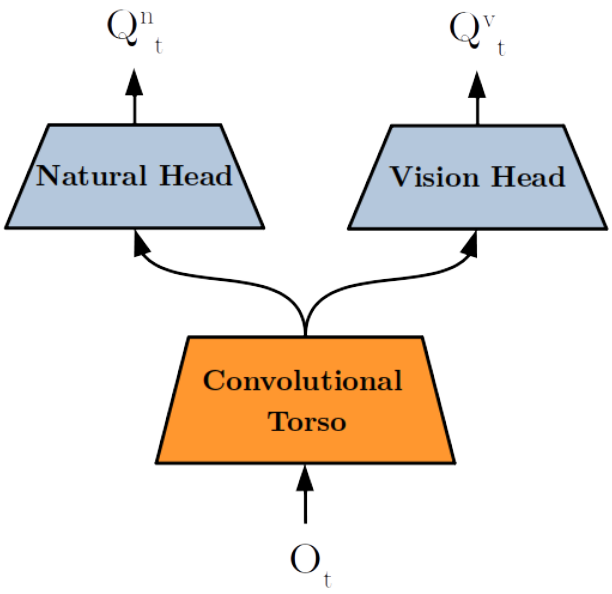


Figure 10: An overview of the MyDQN architecture, where O_t represents the observation at time t, with Q_t^n and Q_t^v being the action-values for the natural actions and visual actions respectively.

The natural TD-errors are to be modified by the improvements introduced in the Deep Q-learning section, which is omitted here for clarity.

The natural controller, i.e. the policy π_n, can as usual be derived from the action-values of the natural head, as follows

$$\pi_n(a|s) = \begin{cases} 1 & \text{if } a = \arg\max_{a \in \mathcal{A}} Q^n(s,a), \\ 0 & \text{otherwise.} \end{cases} \tag{4.21}$$

During training, π_n is to be modified to an ϵ-greedy exploration policy.

4.3 Vision Controller

An important distinction between natural actions and visual actions is that visual actions are unable to directly affect the environment, while natural actions can. This induces a slight change in how the Deep Q-learning updates are to be made for the vision head.

At a given time-step, the agent observes a part of the state and simultaneously decides on the next natural action and the next visual action. The agent then acts in the environment with the natural action and the environment transitions to a new state and the agent receives a reward, while the visual action changes the observation of the new state. The

reward that the agent received is only causal to the natural action, which means that it should not be used in the estimate of the visual action-value target.

The action-values should represent what you expect to gain if you take a specific action, which means that there should be some causal relation between the actions and the rewards in any estimate. However, the visual action that was taken is causally related to every other reward that follows, as it is partly responsible for the next observation, which affects the next natural action, and so on.

In other words, updates of the parameters of the vision head, including the convolutional torso, are to be made through backpropagation on batches of the following TD-errors:

$$\delta_t = R_{t+2} + \gamma \max_a Q^v_{\theta_v}(O_{t+1}, a) - Q^v_{\theta_v}(O_t, A_t), \tag{4.22}$$

where θ_v represents the parameters of the vision head as well as the convolutional torso.

The visual TD-errors are also to be modified by the improvements introduced in the Deep Q-learning section, which is again omitted here for clarity.

The vision controller, i.e. the policy π_v, can as usual be derived from the action-values of the vision head, as follows

$$\pi_v(a|s) = \begin{cases} 1 & \text{if } a = \arg\max_{a \in \mathcal{A}} Q^v_{\theta_v}(s, a), \\ 0 & \text{otherwise,} \end{cases} \tag{4.23}$$

As with the natural policy, the vision policy is to be modified to an ϵ-greedy exploration policy during training.

Furthermore, at the beginning of every episode, or after losing a life, the position of the focal point is randomly assigned to the screen. This helps to prevent overfitting and builds more robust visual policies.

4.4 Simultaneous Updates

To allow for a simultaneous update of the natural head and vision head, the experiences need to also include the reward after the next step. This means that an experience for the myopic agent is to be defined as $(O_t, A^n_t, A^v_t, R_{t+1}, R_{t+2}, O_{t+1})$.

Furthermore, the priorities of these experiences are to be calculated for each head separately using their respective TD-errors, and then averaged to give the new definition of the priority for an experience.

Updating the heads simultaneously by the above definition of experiences aligns the updates of the shared parameters, such as the convolutional torso. This could help stabilizing the learning of visual features.

4.5 Dealing with Partial Observability

As previously mentioned, the modification of the screens for vision introduces some degree of violation to the Markov Property. In the definition of MyDQN the partial observability was ignored, but it certainly does not have to be.

The methods that deal with partial observability all try to form a better representation of the true state. This can be done by either constructing a probability distribution over states given observations, or to simply stack past observations, and possibly past rewards and actions, and use that in place of the true state.

However, it is also possible to learn a representation, by utilizing the properties of a recurrent neural network. The architecture of MyDQN can be changed such that it becomes recurrent in a rather simple way, which is introduced below.

4.5.1 Myopic Deep Recurrent Q-Network

The Myopic Deep Recurrent Q-Network (MyDRQN) defines a near-sighted agent with short-term memory, that also has the ability to control its vision.

The MyDRQN has almost the same architecture as MyDQN but with an extra layer of LSTMs after the convolutional torso and before the natural head and vision head. This allows the hidden states of the LSTMs to remember features of past observations that are extracted from the convolutional torso. This is likely to help the agent build a representation of the true state.

An overview of the MyDRQN architecture can be seen in Fig. 11, and a detailed view can be found in Appendix A.1.

As was discussed in the section on improvements for Deep Q-learning, using an LSTM requires several changes to the distributed setting. These changes can be applied directly to MyDQL without further modifications.

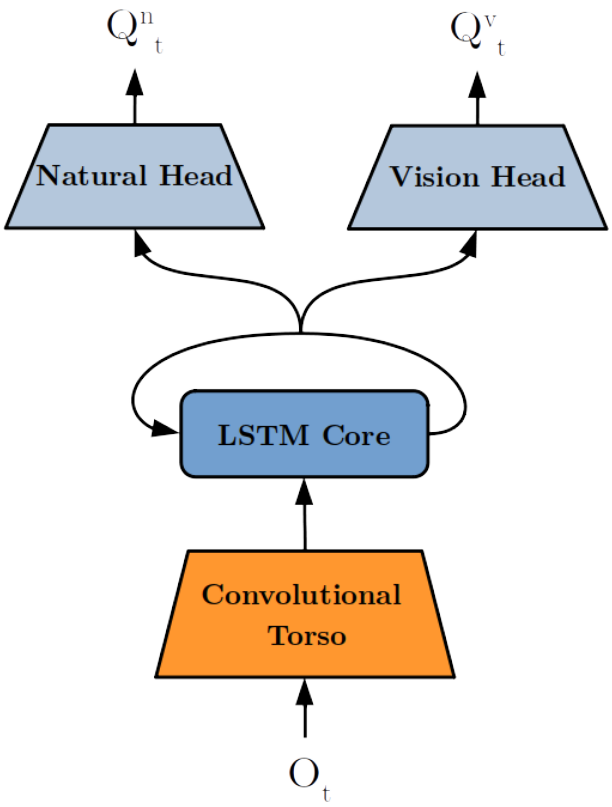


Figure 11: An overview of the MyDRQN architecture, where O_t represents the observation at time t, with Q_t^n and Q_t^v being the action-values for the natural actions and visual actions respectively.

5 Experiments

Below is an introduction to the main experiments that this thesis conducted on the three modified Atari games: Pong, Breakout, and Space Invaders.

5.1 35% Visibility

To test MyDQN and MyDRQN on the modified Atari games, an amount of visibility has to be decided, as well as a focal step-size.

A visible area of 35% was considered a reasonable medium. As this size is small enough to require the agent to actively move its vision to gather information, yet large enough to see that the algorithm actually works.

This corresponds to (50×50) pixels on preprocessed screens of size (84×84) pixels. An illustration of the amount that is visible to such a myopic agent on a non-preprocessed screen can be seen in Fig. 12.

Furthermore, the focal step-size, which translates to the speed of the agent's vision, is set at 4 pixels per action. This made sure to be fast enough to track objects yet keep some continuity between steps.

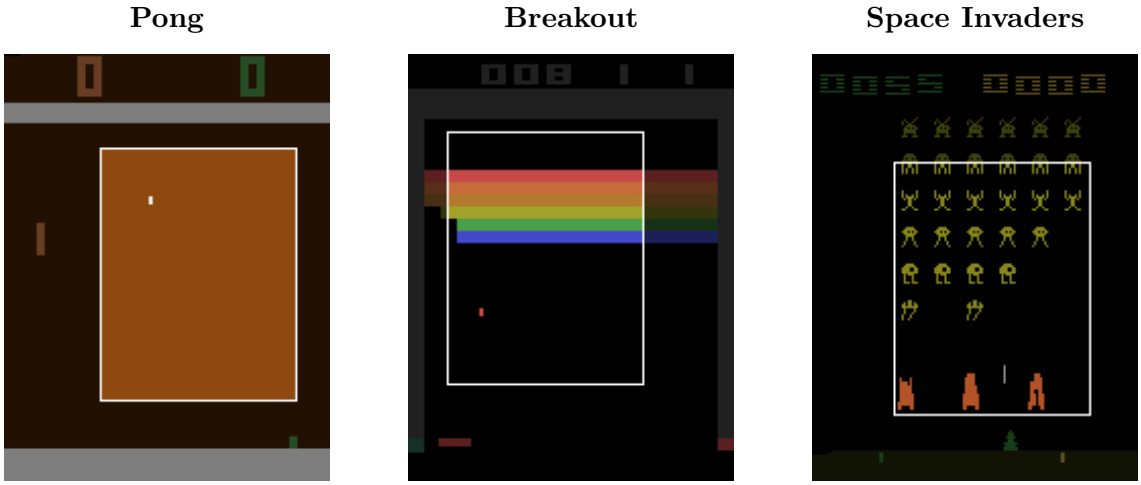


Figure 12: An illustration of the amount that is visible to a myopic agent with 35% visibility on each game. The area outside of the white rectangle is not visible to the agent.

A comparison between MyDQN and MyDRQN was then run on the three modified Atari games: Pong, Breakout, and Space Invaders using 35% visibility.

5.1.1 MyDRQN v.2

An additional experiment was conducted for MyDRQN to see how its performance would be affected with upgrades to the underlying Deep Q-Learning method. These upgrades were a change in the architecture of the heads to dueling architectures, and a change in the update rule to include 3-step returns. A detailed description of the dueling architecture and N-step returns can be found in Appendix A.3.

This version of MyDRQN is to be referred to as MyDRQN v.2.

5.2 13% Visibility

Another experiment was decided to see how the performance of MyDRQN v.2 would be affected if the visibility was reduced much further.

A visibility reduced to 13% seemed reasonably small for this experiment, which corresponds to (30×30) pixels on preprocessed screens of size (84×84) pixels. An illustration of the amount that is visible to such a myopic agent can be seen in Fig. 13.

With 13% visibility the focal point speed was increased to 6 pixels per action. This was made to make the myopic agent less restricted by its visual speed.

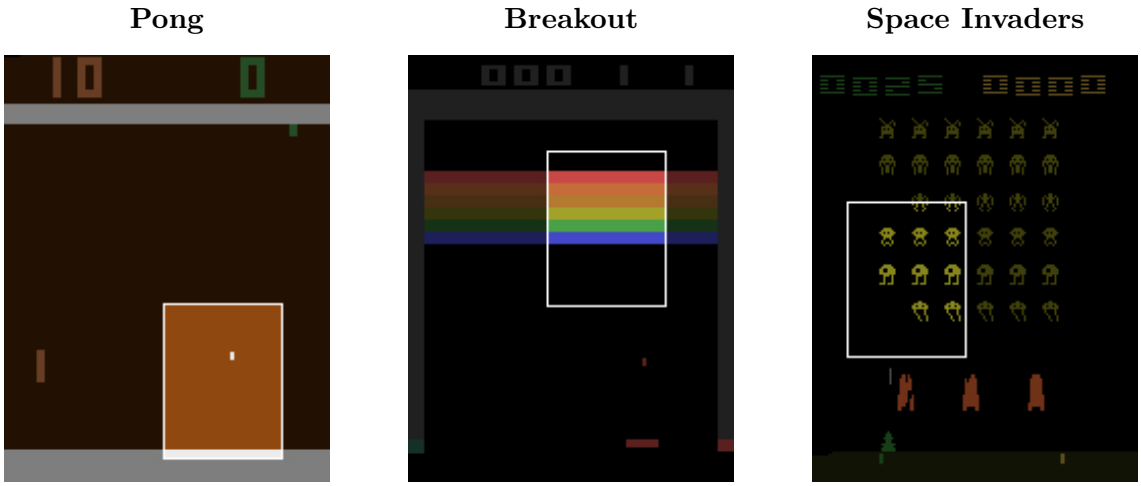


Figure 13: An illustration of the amount that is visible to the myopic agent with 13% visibility for each game. The area outside of the white rectangle is not visible to the agent.

A comparison for MyDRQN v.2 between 13% and 35% visibility was then run on the three modified Atari games: Pong, Breakout, and Space Invaders.

5.3 Hyperparameters

The majority of the hyperparameters were initially set similar to the ones used for Apex [10] and R2D2 [12], which has already proven itself on Atari games in the non-myopic setting. These were then tuned within some limits of time and computational constraints.

The hyperparameters that were tuned the most were the learning rate, the discount, and the target network update frequency. Unfortunately, the computational resources did not allow for higher values of memory consuming hyperparameters, such as the number of actors, batch size, replay memory size, and with the use of LSTMs, the sequence length.

See Appendix A.2 for the complete set of hyperparameters that was used.

The computational resources that were used for training were 4 Geforce GTX (2x1050 and 2x1650) GPUs, which shared 2 Xeon X5550 CPUs each having 4 cores and 8 threads with a total of 47 GB's of RAM. This allowed for a modest distributed setting where an agent was able to be split into one learner and seven actors all utilizing the GPU resources.

All the seven actors had a unique exploration rate taken from the following set:

$$\epsilon \in \{0.5, 0.28, 0.16, 0.09, 0.05, 0.03, 0.02\}.$$

6 Results

An agents performance, or playing strength, is presented in terms of its score per episode throughout the training process. These scores also have a rolling average of a 100 episode window applied to them, such that visualization is improved.

The figures showing the performances include a human baseline for each game, which represents average scores from 20 episodes of a group of professional game testers after 2 hours of practice [6].

All seven actors managed to gather an average of a thousand experiences per second on the Atari games, but the results below are taken from the actor using an ϵ-greedy policy with $\epsilon = 0.02$.

6.1 35% Visibility

The performance results of MyDQN and MyDRQN agents having 35% visibility can be seen in Fig. 14.

The figure shows that the MyDQN agent was struggling on Pong and Space Invaders, but performed decently on Breakout. The MyDRQN agent performed significantly better on all three games, and was still improving on Breakout and Space Invaders at the end of its training.

As this experiment was mainly a comparison, both learning processes were ended when the progress of the MyDQN agent started to slow down significantly.

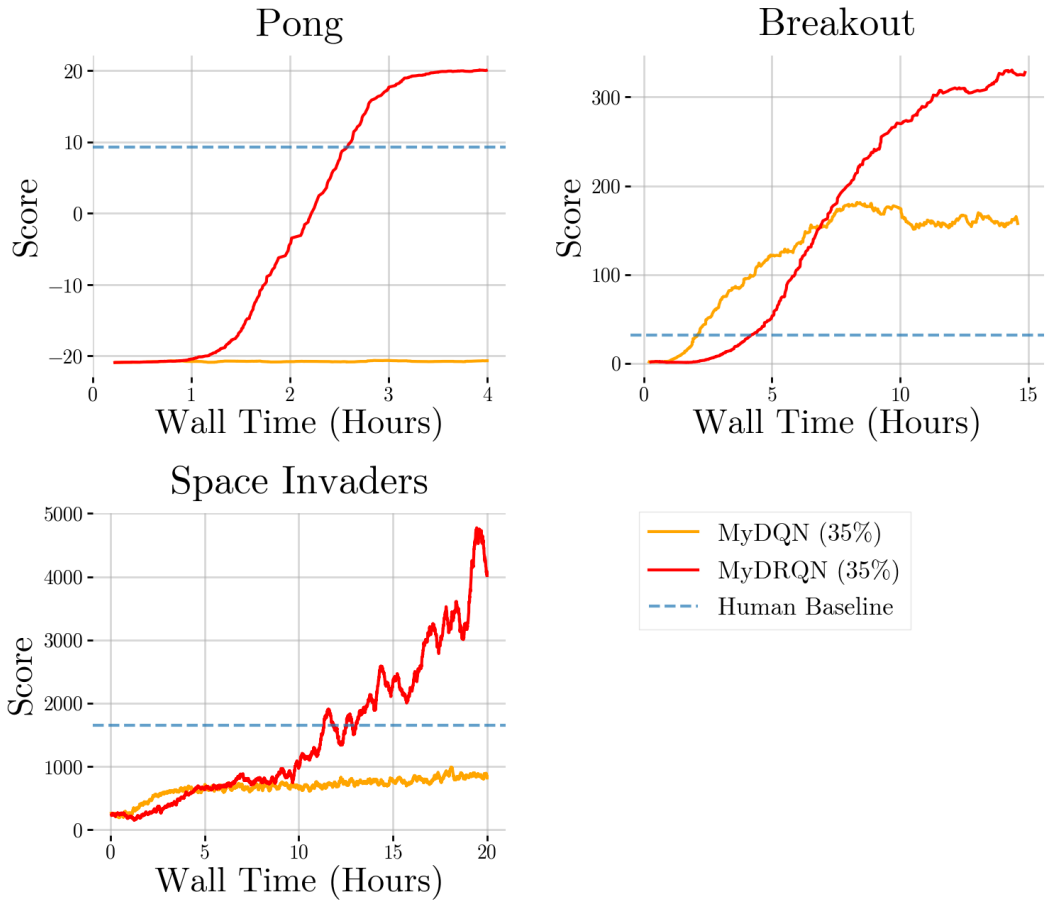


Figure 14: Episodic scores of MyDQN and MyDRQN having 35% visibility on Pong, Breakout and Space Invaders against hours of training. A rolling average of 100 episodes is applied to ease visualization.

6.1.1 MyDRQN v.2

The performance results of a MyDRQN v.2 agent having 35% visibility is shown in Fig. 15, with the results of MyDRQN added for a comparison.

As can be seen, the upgrades of MyDRQN to include 3-step returns and a dueling architecture for its heads resulted in faster learning and better performance on all three games.

The MyDRQN v.2 agent was still improving on Breakout and Space Invaders but its learning process was ended due to time constraints.

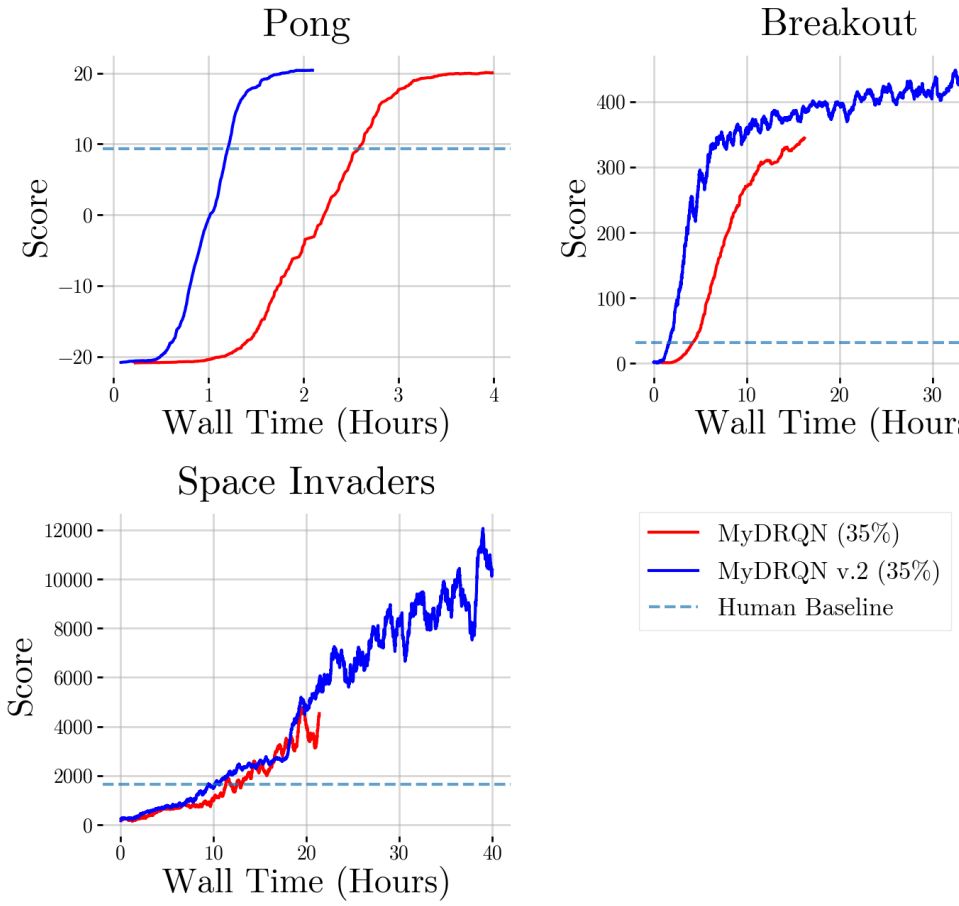


Figure 15: Episodic scores of MyDRQN and MyDRQN v.2 having 35% visibility on Pong, Breakout and Space Invaders against hours of training. A rolling average of 100 episodes is applied to ease visualization.

6.2 13% Visibility

The performance results of MyDRQN v.2 having 13% compared to 35% visibility is shown in Fig. 15.

As can be seen, the visibility reduction to 13% performed worse than 35%, which is expected. However, it still managed to fully beat Pong, and perform quite well on Breakout, and was still improving on Space Invaders, albeit quite slowly.

The learning process of the MyDRQN v.2 agent having 13% visibility was again ended due to time constraints.

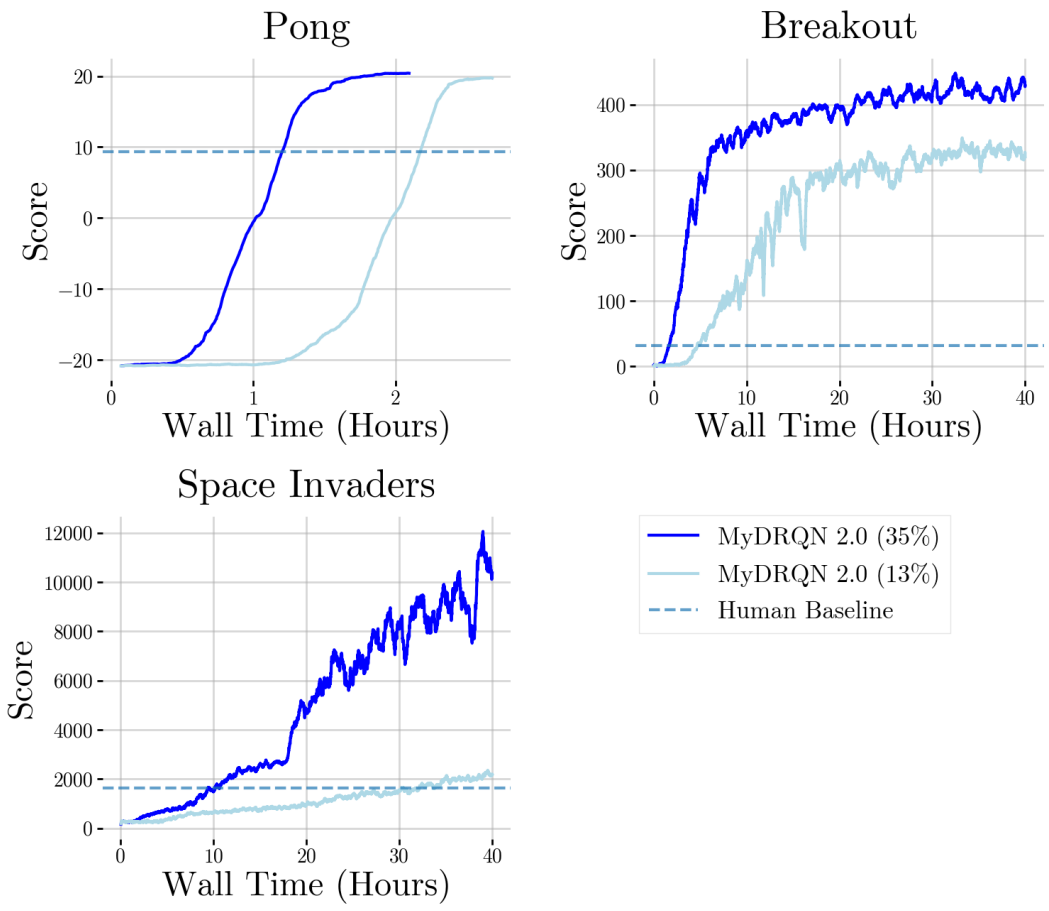


Figure 16: Episodic scores of MyDRQN v.2 having 13% and 35% visibility on Pong, Breakout and Space Invaders against hours of training. A rolling average of 100 episodes is applied to ease visualization.

7 Analysis of Agent Performance

Below is an analysis of the behaviors the MyDRQN v.2 agent having 35% visibility had learned throughout the training process on each game.

The interesting behaviors of the MyDRQN v.2 agents with 13% and 35% visibility are largely the same, but less consistent for 13% visibility.

The figures below display the behaviour of the agent from the point of view of the screen. To improve visualization only non-preprocessed screens are shown, with white rectangles representing the agent's visible portion throughout the game. Furthermore, for Pong and Breakout, a trail of the ball is added, such that its movement can be visualized.

7.1 Pong

In this environment the myopic agent learned to beat the game rather quickly. It learned how to exploit its opponent and form a strategy that wins by 21 - 0 points almost every time.

At the end of the training, the actor with $\epsilon = 0.02$ had an average score over the last 100 episodes of around 20.6 points, which is well above the human baseline of around 9.3 points.

At the beginning of the training, the agent is close to random with both its vision and paddle movements. After a while, it starts to realize the existence of the ball and its paddle, and it starts to try and keep the ball and the paddle visible on the right side of the screen. As it can observe the ball close to its paddle it starts to improve on being able to return the ball. The agent then starts to follow the ball to the other side with its vision such that it is better prepared when the opponent returns the ball. During the later stages of training, it started to realize that it is possible to exploit the opponent from the way it reacts to a newly released ball. It then found a winning strategy in which it was able to quickly execute the same type of shot over and over again.

In the winning strategy the myopic agent makes sure to keep the ball and the paddle visible most of the time. Its vision tends to follow the ball a portion of the way to the other side after a successful return, but does not care about observing the opponent. The agent's view then stops at the center of the screen, and if it observed a newly released ball going towards the opponent, it starts to maneuver its view and the paddle to the bottom right corner of the screen, as it already knows how the opponent will return the ball. As the ball goes towards the right corner the agent is able to execute a type of shot that the opponent is unable to return. Then the agent moves its view such that it can observe a new release of the ball, and does the same maneuvers again. This strategy often finishes the game with 21 - 0.

A depiction of the winning strategy can be seen in Fig. 17.

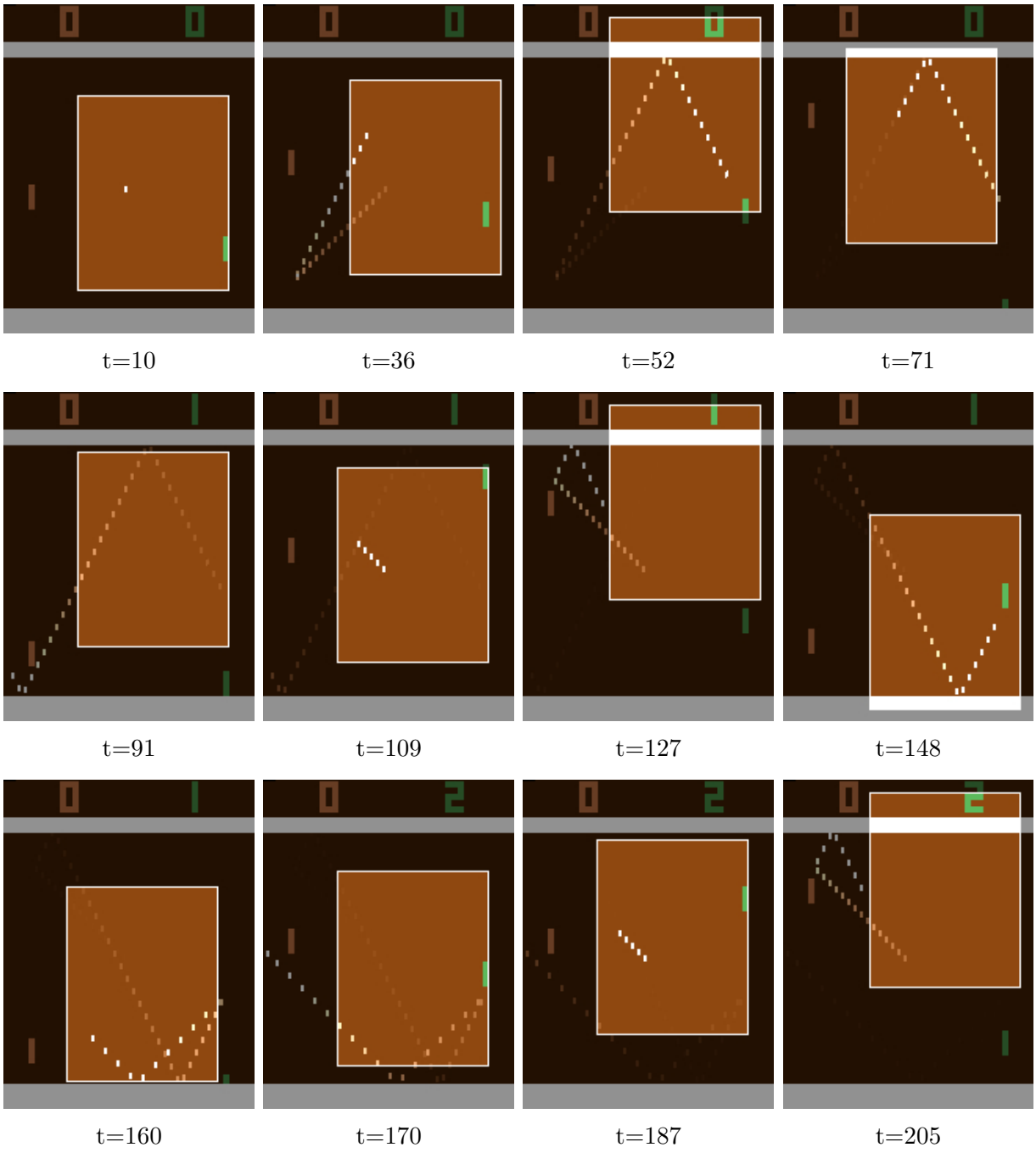


Figure 17: The myopic agent maneuvering in the beginning of an episode of Pong, such that it can execute the same shot over and over again. Notice that the screens at $t = 127$ and $t = 205$ are nearly identical, which means that the agent is near a closed trajectory to victory by 21 - 0. Also, notice how the myopic agent never needed to observe the opponent.

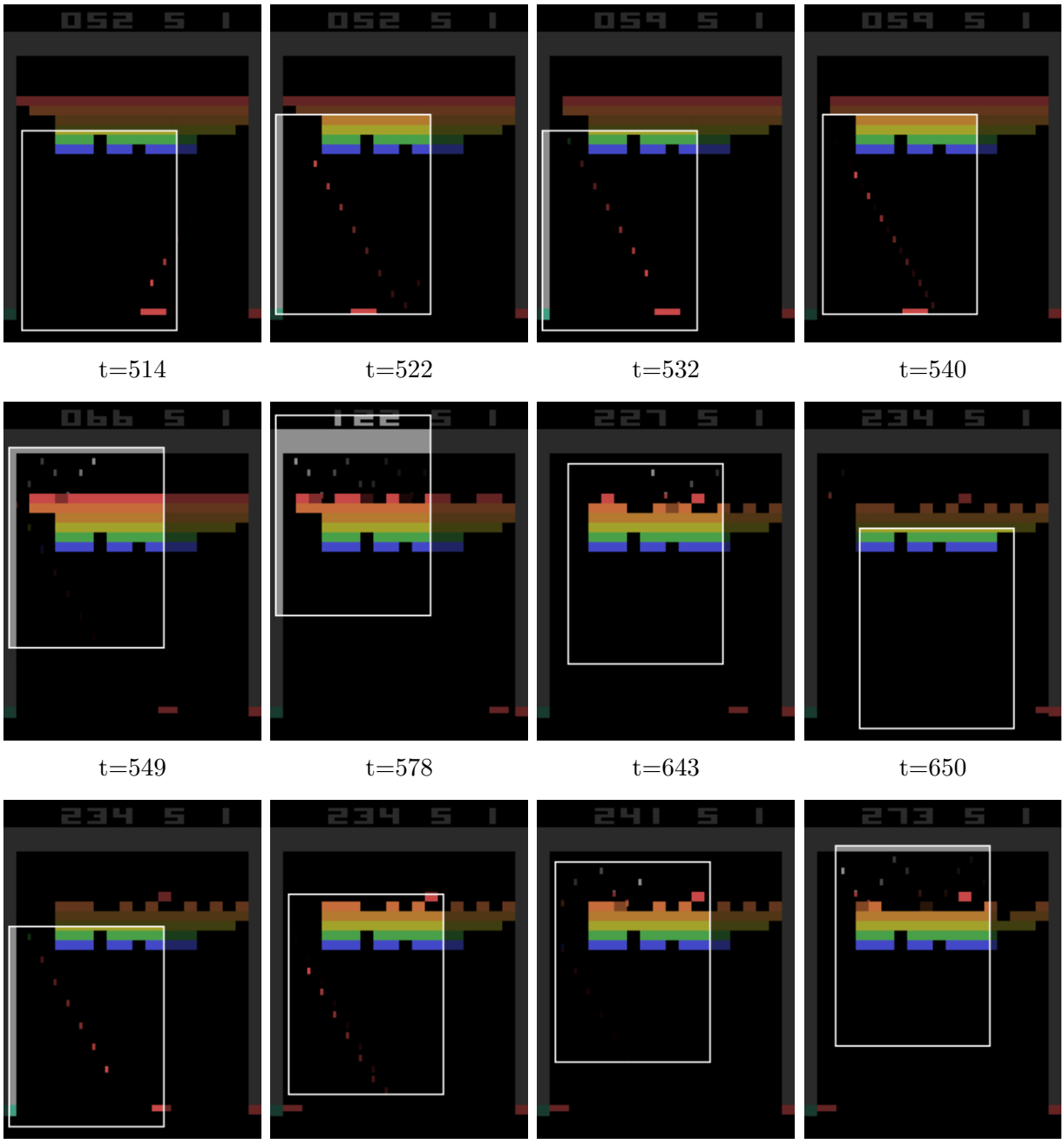
7.2 Breakout

In this environment the myopic agent was able to clear almost all of the first set of blocks consistently, and if it managed to clear all on the first set, it would also clear almost all on the second set.

At the end of training, the actor with $\epsilon = 0.02$ had an average score over the last 100 episodes of around 430 points, which is well above the human baseline of around 31.8 points.

At the beginning of training, the agent moves both the paddle and its vision close to randomly. After a while, it starts to realize that hitting the ball somehow is rewarding, so it began to keep the ball visible near the lower part of the screen. The agent then started to become consistent in returning the ball with its paddle, which increases the difficulty of keeping the ball visible, as the speed of the ball increases after hitting the third row of blocks and above. The agent then slowly adjusts to the increased speed and starts to clear even more rows. Eventually, the agent encounters the ball tunneling through the blocks, such that it bounces on the top row. This is a strategy which leads to a lot of reward in a fast manner, and the agent started to adjust its behavior such that it is more likely to make a tunnel. At the beginning of learning this strategy, the agent did not observe the ball as it tunneled but instead stayed below the tunnel keeping its paddle visible and waiting for the ball to come down. It then slowly learned that it is more successful if it also observes the ball bouncing on the top rows, such that it could better predict where and when it would come down. The agent then continued to improve upon this strategy until the end of the training.

A depiction of the tunneling strategy can be seen in Fig. 18.



t=514 t=522 t=532 t=540

t=549 t=578 t=643 t=650

t=658 t=666 t=675 t=702

Figure 18: The myopic agent performing its tunneling strategy in Breakout. Notice how its vision tries to keep the ball and the tunnel visible at the same time. Also, notice how it directly returns the ball back through the tunnel as it comes down again.

7.3 Space Invaders

In this environment the myopic agent learned to consistently beat the first level rather quickly, it soon after learned to beat the second consistently, and then the third.

At the end of the training the actor with $\epsilon = 0.02$ was able to get large amounts of score each episode and was still improving, albeit slowly. The average score over the last 100 episodes was around 10000 points, which is well above the human baseline of around 1652 points.

As usual, the myopic agent is close to random in its movements at the start. However, it quickly learned to fire and move to the right, and not soon after it learned to try and keep the laser cannon visible. It then figured out that it could survive longer and get more reward by firing the laser cannon near bunkers. Then as the agent improved its ability to keep the laser cannon visible it also started to improve on how to avoid lasers in the open, which made it start to move around the screen while firing. The agent improved its precision in hitting the aliens and avoiding the lasers while keeping the laser cannon visible. This allowed it to eventually get to the third level, from which it needed to learn a specific strategy.

At the third level, the agent had to learn how to quickly shoot the aliens in the lowest rows first, as there was little time to prevent them from invading. After a while, it managed to learn a strategy to beat the third level as well, which was to quickly observe and move the cannon to the right while shooting accurately, and then quickly change direction to the left and make sure to shoot the lowest alien first, and then continue as usual. It improved upon this strategy until the end of training such that it was able to consistently beat the first, second, and then the third level many times, in one episode.

A depiction of the myopic agent perfectly beating the third level can be seen in Fig. 19.

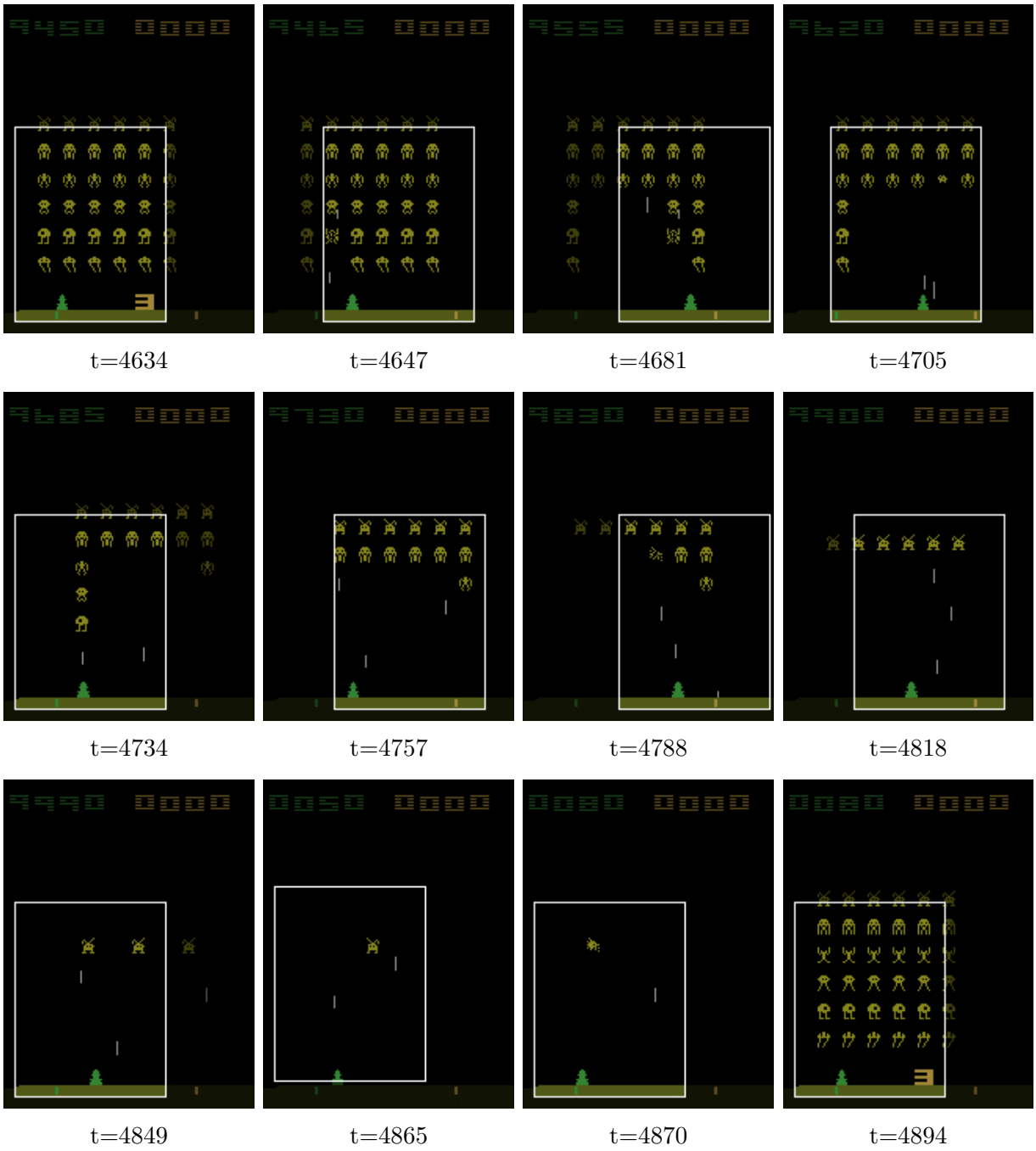


Figure 19: The myopic agent clearing the third level for the eight time in one episode of Space Invaders. Notice that the agent leaves the left column of aliens but quickly gets back to it before the formation moves down once and invades. Also, it is interesting to notice that it achieved a score higher than the game is able to display.

8 Conclusion

This thesis introduced an adaptive sensing task to Atari 2600 games, as well as a reinforcement learning algorithm that was able to learn how to solve this new task. This algorithm is referred to as Myopic Deep Q-Learning (MyDQL).

The adaptive sensing task was for the agents to learn how to observe the screen of an Atari game as they learned how to play. These myopic agents could only observe a small part of the screen at any time, but had the ability to move the visible part around.

To solve these tasks, a slightly modified Deep Q-Learning algorithm was introduced to simultaneously learn a policy for vision and character control. This is the previously mentioned MyDQL algorithm.

Furthermore, two different neural network architectures were introduced for MyDQL. The first architecture is referred to as a Myopic Deep Q-Network (MyDQN), which is a fully feed-forward neural network. The second architecture is based on MyDQN but has an added layer of LSTMs, which is referred to as a Myopic Deep Recurrent Q-Network (MyDRQN).

It was shown that on three modified Atari games: Pong, Breakout and Space Invaders, a MyDQN agent with only 35% visibility can learn how to play decently on Space Invaders, and above the human baseline on Breakout, but was struggling on Pong.

It was also shown that a MyDRQN agent performs significantly better, and was able to play close to optimally on all three environments, again only having 35% visibility. The MyDRQN agent was also shown to able to perform impressively on each game having only 13% visibility.

The difference in performance between MyDQN and MyDRQN is believed to be due to a significant amount of partial observability of the modified Atari games. The recurrent nature of MyDRQN makes it better equipped to represent the true state of the environment. This is likely enhanced by the fact that an agent has access to all parts of the screen, just not at once.

The set of all modified Atari games can be used to test reinforcement learning techniques that deal with partial observability. This can be useful as many real-world environments are partially observable, which most reinforcement learning algorithms are unequipped to deal with.

It is possible that agents that are able to perform well across all of the modified Atari 2600 games could also perform well in similar real-world environments. This could find applications within robotics, where robots often have to dynamically sense their surroundings.

Furthermore, training with higher values of hyperparameters such as the number of actors, batch size, and sequence length, is likely to add additional performance to MyDRQN, but this is left for future work.

Acknowledgments

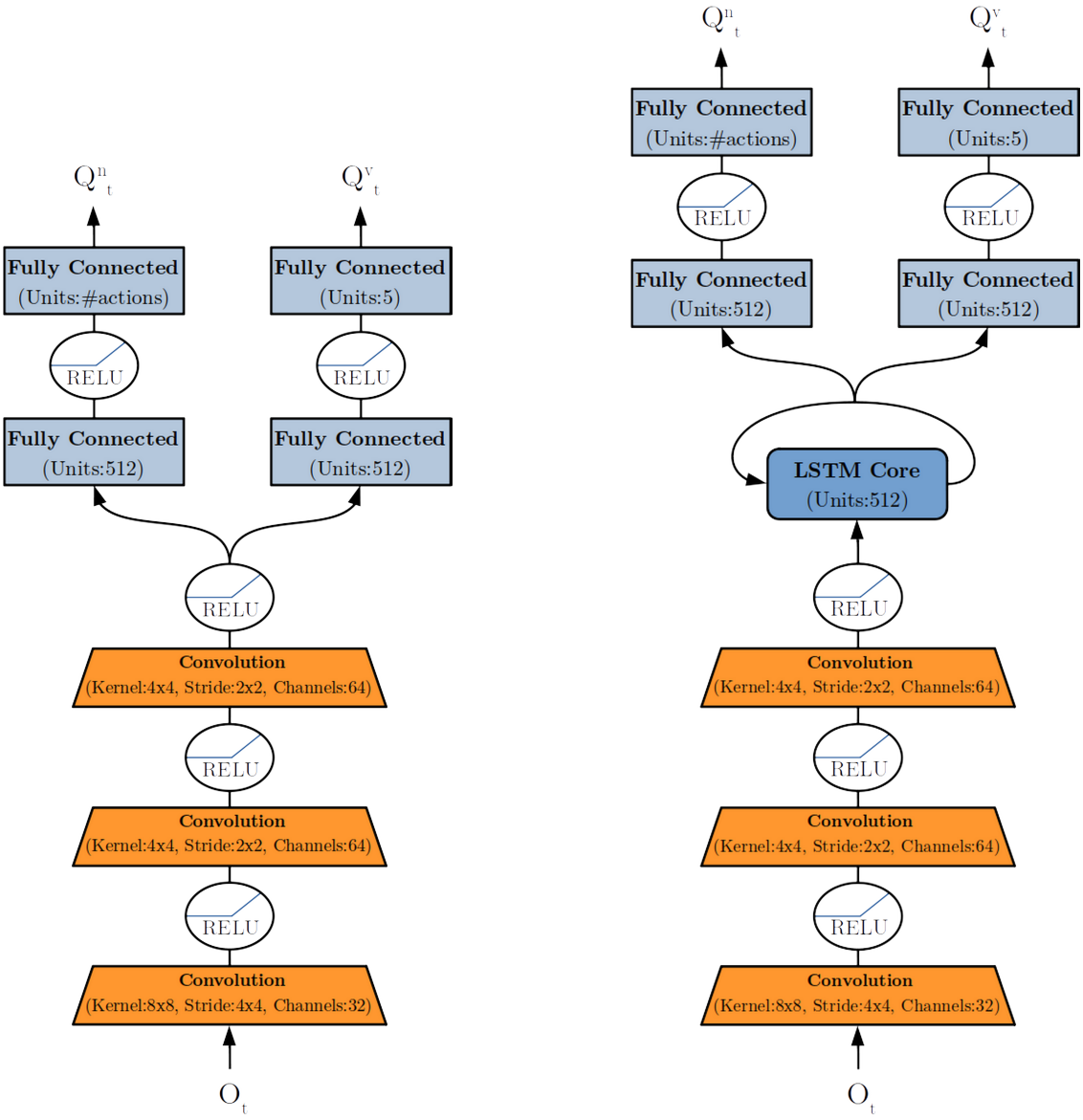
I would like to say thanks to my supervisor Mattias Ohlsson, and especially my co-supervisor Viktor Stagge, as well as everyone at Sentian.AI that helped me while making this thesis.

A Appendix

A.1 Network Architectures

The details in the architecture of MyDQN and MyDRQN is similar to the one used in [5]. However, two heads are used instead of one.

A detailed view of MyDQN and MyDRQN can be seen in Fig. 20.



(a) MyDQN (b) MyDRQN

Figure 20: A detailed view of the MyDQN and MyDRQN architectures.

A.2 Hyperparameters

This section contains the values of the hyperparameters that are used for the applications of MyDQL on the Atari 2600 games.

Table 2: Hyperparameters used for MyDQN and MyDRQN.

Hyperparameter	**Value**
Atari preprocessing	
Frame stack	4
Frame size	84×84
Gray-scale	Yes
Action hold	4
Max-pool screens	Yes
Initial no-operations	$[0, 30]$
Terminal on loss of life	Yes
Clip rewards to $\{-1, 0, 1\}$	Yes
Fire on reset	Yes
Number of actions	Game specific
Training	
Learning rate	2.5×10^{-4} (1.25×10^{-4} for MyDQRN 2.0)
Optimizer	Adam ($\epsilon = 1.5 \times 10^{-4}$)
Loss function	Huber
Batch size	32
Target network update frequency	2500 updates (400 for MyDQRN)
Prioritized Replay Memory	
Replay Size	300000
Minimal priority (ϵ)	0.01
Prioritization parameter (α)	0.6
Bias correction parameter (β)	1.0
Distributed setting	
Number of actors	7
Exploration rates (ϵ)	{0.5, 0.28, 0.16, 0.09, 0.05, 0.03, 0.02}
Actor buffer size	50
Weight synchronization	400 time-steps
Distributed setting with LSTM	
Sequence length	40
Sequence overlap	15
Burn-in steps	10
Sequence priority parameter (η)	0.9

A.3 MyDRQN v.2

In this section are the two main improvements of MyDRQN presented.

A.3.1 N-Step Returns

There are two main ways of estimating expected returns for a given state, either by Monte-Carlo (MC) methods or Temporal-Difference (TD) methods.

MC methods use the returns of full trajectories to estimate the value targets used in an update, while TD methods estimates the value targets by the first reward of the trajectory and the value estimate at the next step. This means that MC updates have to wait until a trajectory has been completed, while TD updates can be performed at every step, as is done in Q-learning.

A downside of MC methods is that the return of a full trajectory depends strongly on the current policy, which clearly changes during training. It is, therefore, difficult to benefit from a replay memory. Additionally, these methods tend to have a high degree of variance. However, a benefit is that the estimates of the expected returns are completely unbiased.

On the other hand, the TD methods tend to have low variance but are biased, as an estimate is being used to calculate another estimate.

In other words, MC methods tends to have high variance but are unbiased, while TD methods tend to have low variance but are biased. This opposition suggests that it might be possible to benefit from combining the two methods, such that both the variance and bias is at a reasonable level.

The method of N-step returns interpolates between MC methods and TD methods. Instead of using full trajectory returns to estimate the value targets, only the first N rewards are used, as well as the value estimate at the Nth step. In other words, an N-step update for Q-learning is as follows:

$$Q(S_t, A_t) \leftarrow Q(S_t, A_t) + \alpha \left[\sum_{i=0}^{N-1} \gamma^i R_{t+1+i} + \gamma^N \max_a Q(S_{t+N}, a) - Q(S_t, A_t) \right].$$

Another potential benefit to N-step returns is that for every terminal state there will be N updates that used it. This means that there will be a lot more accurate updates of trajectories leading to terminal states.

The choice of N in N-step returns is another hyperparameter added to the algorithm.

A.3.2 Dueling Architecture

The dueling architecture [14] explicitly separates the action-values into a representation of state values and action advantages.

The action advantages are defined as

$$a(s,a) = q(s,a) - v(s), \tag{A.24}$$

and serves to evaluate the advantage of each action in a given state.

A detailed view of the head of a dueling architecture can be seen in Fig. 21.

The head of the dueling architecture contains two forward streams that are to be connected to a convolutional torso. These two streams represent the networks estimate of $a(s,a)$ and $v(s)$ respectively, i.e. $V_\theta(s)$ and $Q_\theta(s,a)$. At the top of the architecture, the streams are combined to form the action-values, in the following way

$$Q_\theta(s,a) = V_\theta(s) + (A_\theta(s,a) - \bar{A}_\theta(s)), \tag{A.25}$$

where $\bar{A}_\theta(s)$ is the average of $A_\theta(s,a)$. The subtraction of the average advantage estimate is introduced purely for stability reasons.

The dueling architecture allows for the agent to learn the value of a state without involving the actions, which can be especially helpful in situations where all actions have equivalent outcomes.

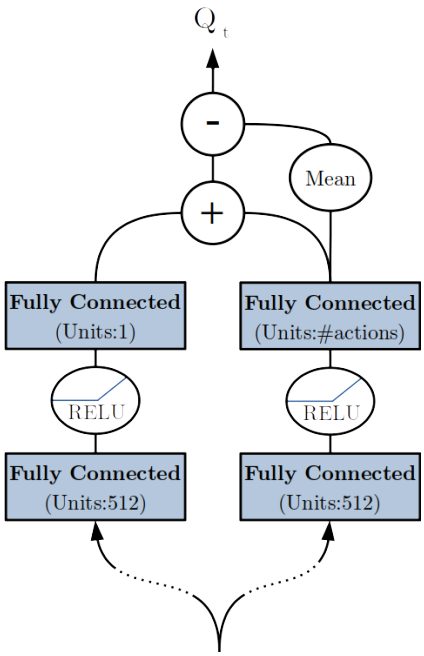


Figure 21: A detailed view of the dueling architecture.

References

[1] David Silver, Thomas Hubert, Julian Schrittwieser, Ioannis Antonoglou, Matthew Lai, Arthur Guez, Marc Lanctot, Laurent Sifre, Dharshan Kumaran, Thore Graepel, et al. Mastering chess and shogi by self-play with a general reinforcement learning algorithm. *arXiv preprint arXiv:1712.01815*, 2017.

[2] 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.

[3] Richard S Sutton and Andrew G Barto. *Reinforcement learning: An introduction*. MIT press, 2018.

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

[5] Volodymyr Mnih, Koray Kavukcuoglu, David Silver, Alex Graves, Ioannis Antonoglou, Daan Wierstra, and Martin Riedmiller. Playing atari with deep reinforcement learning. *arXiv preprint arXiv:1312.5602*, 2013.

[6] 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. Human-level control through deep reinforcement learning. *nature*, 518(7540):529–533, 2015.

[7] Hado Van Hasselt, Arthur Guez, and David Silver. Deep reinforcement learning with double q-learning. *arXiv preprint arXiv:1509.06461*, 2015.

[8] Long-Ji Lin. Reinforcement learning for robots using neural networks. Technical report, Carnegie-Mellon Univ Pittsburgh PA School of Computer Science, 1993.

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

[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.

[11] Sepp Hochreiter and Jürgen Schmidhuber. Long short-term memory. *Neural computation*, 9(8):1735–1780, 1997.

[12] Steven Kapturowski, Georg Ostrovski, John Quan, Remi Munos, and Will Dabney. Recurrent experience replay in distributed reinforcement learning. In *International conference on learning representations*, 2018.

[13] Greg Brockman, Vicki Cheung, Ludwig Pettersson, Jonas Schneider, John Schulman, Jie Tang, and Wojciech Zaremba. Openai gym. *arXiv preprint arXiv:1606.01540*, 2016.

[14] Ziyu Wang, Tom Schaul, Matteo Hessel, Hado van Hasselt, Marc Lanctot, and Nando de Freitas. Dueling network architectures for deep reinforcement learning, 2015. *URL https://arxiv. org/abs/1511.06581*, 2015.