FACULTY
OF INFORMATION
TECHNOLOGY
CTU IN PRAGUE

Assignment of master's thesis

Title:	Deep Reinforcement Learning for Super Mario Bros
Student:	Bc. Ondřej Schejbal
Supervisor:	Ing. Daniel Vašata, Ph.D.
Study program:	Informatics
Branch / specialization:	Knowledge Engineering
Department:	Department of Applied Mathematics
Validity:	until the end of summer semester 2022/2023

Instructions

Deep neural networks used for reinforcement learning are achieving significant performance in various tasks. The aim of this thesis is to focus on playing the Super Mario Bros game using state-of-the-art Deep Reinforcement Learning methods.

Detailed description:
- Select an appropriate tool that enables the learning framework to interact with the game's environment. It means that the agent can get all necessary information about the current state of the Super Mario Bros game in a suitable format.
- Research the recent deep reinforcement learning methods and select at least two of them that should be suitable for a given task.
- Implement the learning framework and selected methods.
- Measure, compare and discuss the performance of the learning process. Observe trained agents' performance on different game levels, which he has not been trained on. Try to fine-tune the methods and their hyperparameters to get the best results.

Electronically approved by Ing. Karel Klouda, Ph.D. on 8 February 2022 in Prague.

FACULTY
OF INFORMATION
TECHNOLOGY
CTU IN PRAGUE

Master's thesis

Deep Reinforcement Learning for Super Mario Bros.

Bc. Ondřej Schejbal

Department of Applied Mathematics
Supervisor: Ing. Daniel Vašata, Ph.D.

May 3, 2022

Acknowledgements

Firstly, I would like to thank Ing. Daniel Vašata, Ph.D. for supervising me during the completion of this thesis. His advice and recommendations during our consultations were a great help during my work. I am also grateful that he was understanding of and cooperated with me in the first months when I was concluding my studies abroad in Estonia.

My sincere thanks also go to my family and friends for endless support and patience during my studies. I would have never been able to achieve any of this without you.

Declaration

In Velké Popovice on May 3, 2022 . . . .

Citation of this thesis

Schejbal, Ondřej. *Deep Reinforcement Learning for Super Mario Bros.*. Master's thesis. Czech Technical University in Prague, Faculty of Information Technology, 2022. Also available from: ⟨`https://github.com/schejond/diploma-thesis/blob/master/schejbal_diploma_thesis.pdf`⟩.

Abstract

Within this master's thesis, a fine-tuned reinforcement learning model capable of preparing an intelligent agent able to play the Super Mario Bros. game has been created. Its architecture is based on conducted research on current state-of-the-art reinforcement learning techniques where the most relevant models for this type of task have been compared between each other. In order to compare the models, research and description of tools that allow the model to interact with the game had been done. Based on the comparison results, the most suitable approach was selected. Experiments with applying various modifications to the selected model have been done in order to find the most suitable modifications for the Super Mario Bros. game. The fine-tuned model has been used to train an intelligent agent, whose performances were tested on the level he was trained on and also on two levels that he had never seen before. The agent's performances were really good and showed nice behavioral patterns, mainly on the level he was trained on, as his performance on the unseen levels was understandably worse.

Keywords Deep Reinforcement Learning, Deep Q-learning, Asynchronous Advantage Actor-Critic, Twin Delayed Deep Deterministic policy gradient algorithm, AI agent, OpenAI Gym, Super Mario Bros.

Abstrakt

V rámci této diplomové práce byl připraven odladěný model posilovaného učení, který je schopný natrénování inteligentního agenta způsobilého hrát hru Super Mario Bros.. Jeho architektura je založena na provedeném průzkumu aktuálních state-of-the-art technik posilovaného učení, kde mezi sebou byly porovnány modely, které jsou pro tento typ úlohy nejvíce relevantní. Pro možnost porovnání modelů byl proveden průzkum a popis nástrojů, které umožňují interakci modelů s hrou. Na základě výsledků porovnání modelů byla vybrána nejvhodnější metoda. Následně byly provedeny experimenty s aplikováním rozmanitých modifikací na vybraný model za účelem najít nejvhodnější úpravy pro hru Super Mario Bros.. Odladěný model byl následně použit k natrénování inteligentního agenta, jehož výkony byly otestovány na úrovni, na které byl natrénován a také na dalších dvou úrovních, které nikdy neviděl. Výkony agenta byly velmi dobré a ukázaly pěkné vzorce chování hlavně na úrovních, na kterých byl natrénován, ačkoliv jeho výkon na neznámých úrovních byl pochopitelně horší.

Klíčová slova Hluboké posilované učení, Deep Q-learning, Asynchronous Advantage Actor-Critic, Twin Delayed Deep Deterministic policy gradient algorithm, AI agent, OpenAI Gym, Super Mario Bros.

Contents

List of Figures

List of Tables

List of Algorithms

List of Source Codes

Chapter 1

Introduction

Artificial Intelligence is one of the biggest buzzwords in the digital world and it is also one of the most progressively studied and developed areas of computer science. Reinforcement learning (RL) is a process of teaching a computer agent to simulate human-like behavior and be able to respond to its environment and overcome various obstacles and tasks. Reinforcement learning tasks have in common the fact that at the start, the agent knows nothing about the environment and has to learn the values of each action for each possible state by repeatably playing the game and utilizing the numerical reward values obtained for each performed action. Reinforcement learning gained more popularity in recent years thanks to the developments in available computational power as most of the RL tasks are really complex and, therefore, computationally expensive.

This thesis focuses on studying and describing the state-of-the-art techniques used for training an intelligent agent and then observing their efficiency in the Super Mario Bros. game. The Super Mario Bros. was chosen because it represents a complex RL task for which no direct model has been developed yet (at least to our knowledge). Most of the state-of-the-art models were trained on Atari-based games, where the majority of these games have a static, closed environment, where few obstacles are changing their position, but the agent has limited possibilities when it comes to movement and is not able to move from the closed environment. Compared to this, the Super Mario Bros. game is way more complex as not only the obstacles in the environment are moving, but the background of the environment itself shifts with each move of the agent towards the end. The environment also contains hidden obstacles that become only visible upon interaction with the agent which can act as a surprise factor for the neural network as it can be hard to predict.

We have found the complexity of this task interesting and decided to test how current state-of-the-art approaches can handle this complex task. We expect that since none of the models were prepared for the Super Mario Bros. game that we will also be able to come up with modifications that will be more beneficial for teaching the agent to play this game.

1.1 Goals

The main goal of this thesis is to compare the performance of current state-of-the-art RL models on the Super Mario Bros. game and then prepare a modified version of one of the models, which will be fine-tuned for the game and further improve the agent performances.

The first goal of this thesis is to perform research and describe the current state-of-the-art RL approaches which could be suitable for training agent to play the Super Mario Bros. game. The next goal is to find an appropriate tool that will enable the interaction of the models with the environment while also providing some reward signals that the model can use for improving the agent's decision-making. The tool should also enable showing the visual outputs for human evaluation of the trained agent. A follow-up objective is to implement at least two selected RL methods from the researched ones and to prepare a custom modified model, which will be fine-tuned for the Super Mario Bros. game. The final goal of this thesis is to compare and discuss the performance of the learning process of the implemented models and perform a series of test runs of the trained agent's performances on the trained game level and also on different game levels, which he has not been trained on.

Chapter 2

Existing approaches

One of our goals is to prepare an agent that can react to the game states and predict the best action to perform for the current game state. This chapter focuses on studying common reinforcement learning approaches used for controlling and training the agent directly from high-dimensional sensory inputs. These inputs are obtained from the selected tool which are discussed in Chapter 3. We discuss common pre-processing practices for these inputs and their benefits along with the advanced techniques which improve the training of the model. At last, we focus on the most used deep learning approaches in this field and discuss their pros and cons.

2.1 Basic approaches

We are dealing with a task where the so-called agent interacts with the environment with an aim to always choose the best action to take for the current state he is in. The reinforcement learning approach is a perfect candidate for dealing with this kind of task.

In reinforcement learning, the agent first starts with no knowledge about the environment and then learns by trial and error by playing the game over and over again. After each action the agent takes, he receives a reward that reflects if the action led to positive or negative improvement. Based on the result, he updates his internal action table, which helps him to prioritize more beneficial actions in the future. For more details, see e.g. [23].

In the subsections below, we explain basic techniques and approaches used for tasks of this kind, but first, we introduce fundamental parts of the system.

We work with an environment that is represented by the game itself. In our thesis, we have worked with Super Mario Bros.[1] game. The agent in our environment is the character that we can control, so in this case, it's Mario. The environment has a defined set of actions that the agent can perform.

[1] `www.mariowiki.com/Super_Mario_Bros.`.

These vary from basic movements to jumps and combinations of these actions. Actions can be often limited to only a subset of actions which leads to a lower amount of possibilities for the agent but can be beneficial in terms of dimension reduction and learning speed.

Another key term is the reward function and current game state. The reward function plays a crucial role in the training process as it is able to provide the agent with a reward for any game state the agent can be in. The reward can be based on various factors. For example, the distance of the agent's position from the start of the level, the number of coins collected, etc.

The most crucial part of the reinforcement learning approach is the agent's memory and constructs on which he bases his decisions of action selection. These constructs are usually represented by some high-dimensional tables that update their value over time by using an appropriate update policy. We discuss the most common approaches in the sections below.

2.1.1 Q-learning

One of the common concepts in reinforcement learning is Q-learning which is based on so-called Q-values. [13] Each Q-value is meant to represent the beneficial value of selecting given action in a given state. Q-values are gradually updated with each step of the agent with the aim to help the agent determine the optimal action for the given state of the game. We discuss the updates of the Q-values below, but first, we need to introduce the term Q-table.

The Q-table represents the agent's experience, sometimes we also refer to it as the agent's memory. Q-table's dimensions are the number of actions by the number of possible states in the game. In each cell, the table contains the Q-value, which represents how good is the selection of the given action for the current state. At the start, the values are all initialized to the same value, usually zero, representing that the agent knows nothing about the state-action game space. In each step, the agent looks into the table, checks all the actions for his current position, and chooses the one with the biggest Q-value. If there are more of them with the same value, he chooses one of them randomly.

The agent can also select a random action once in a while to prevent getting stuck in local optima and more explore the state-action space. A simple strategy for tackling the exploration-exploitation trade-off is the Epsilon Greedy Exploration Strategy. In this strategy, we use the hyper-parameter value noted as ϵ, which is called the exploration rate. At each step, the agent generates a random number from interval $[0, 1)$. If the generated value is greater than the value of ϵ then the agent performs a so-called greedy action, which means that he chooses the next action randomly without considering the Q-values. It is a common practice to start with a high value of ϵ and then gradually decrease it with the number of epochs so the agent tends to less rely on randomness in the latter training stages.

During the training phase, the agent learns each state-action pair's expected reward by trial-and-error and updates the relevant Q-values with new ones. The updates are based on the update policy. For Q-learning, the standard policy used for the updates is based on the Bellman equation which is explained in Section 2.1.2.

2.1.2 Bellman equation

The core principle of Bellman's equation is in breaking down the complex problem into simpler, recursive friendly subproblems and finding their optimal solution. Because of this, it is omnipresent in the majority of the solutions for RL problems.

Bellman equation is a well-known construct that is used in other contexts as well. [23] It is mostly known for its application in dynamic programming, where it is used as a necessary condition for optimality. The Bellman equation is also beneficial for most ML problems because its form is natural for iterative processing.

When working with Q-values we have a need for an update policy, which is based on the reward function and is able to work with the current agent's state while also considering future actions and converges to an optimal value. Bellman's equation satisfies all these requirements.

Bellman's equation takes the form:

$$Q(s_t, a_t) = (1-\alpha)Q(s_t, a_t) + \alpha \cdot (R_t + \gamma \cdot \max_a Q(s_{t+1}, a)), \tag{1}$$

where $Q(s, a)$ is the Q-value for the given state s when action a is chosen as the next step, s_t denotes the state of observation at given time step t, a_t is the action, that the agent can perform in the given state at time step t, R_t is the reward gained for taking action a at time step t and state s, α is the learning rate and γ is the discount factor.

[25]

In summary, we can say that the Bellman's equation decomposes the value function into two parts:

a) the immediate reward

b) the discounted future reward

The equation simplifies the computation such that we can find the optimal solution to a complex problem by breaking it down into simpler, recursive subproblems. In each step, the agent chooses an action from the Q-table, which has the maximal value according to the formula above.

The learning rate hyper-parameter $\alpha \in [0, 1]$ controls speed of the learning process. With small values of α, the model chooses an action with the biggest Q-value while ignoring the values obtained from the reward function. With bigger values of α, the reward function impacts the update of the Q-value more significantly. The low values are used when executing and observing the trained agent's performance, where we don't want the agent to learn anymore, just perform based on learned Q-values. In the training phase, we use this to control the amount of impact the reward function has on each Q-value update.

The discount factor $\gamma \in [0, 1]$ plays an important role in the learning process. If $\gamma = 0$, then the agent is only interested in the immediate reward and ignores the long-term return. This is often useful for later stages of the game, where it's more beneficial to prioritize immediate return over the long-term reward, because the game is more likely to end soon (the time is running out, the agent is near the finish line, etc.), so it's a common practice to gradually decrease the value of the γ hyper-parameter. [24]

2.2 Deep Q-learning

Our selected environment to work with, the Super Mario Bros. game, concludes of a very high variety of possible combinations of states and actions that can be taken in these states. Constructing a table like it's done in a normal Q-learning approach would require a table of such a big size, that the complexity of our trial-and-error approach would become very high. In scenarios like these, the so-called deep reinforcement learning approach can be applied. [17] Deep Q-learning can be also found in literature denoted as DQN. In this section, we will describe the details of the DQN approach.

The "deep" portion of reinforcement learning refers to the fact that we implement the reinforcement learning principles with the help of multiple layers of neural networks. These are supposed to replicate the structure of a human brain. The idea is to construct neural networks in such a way that it enables the agent to make more human-like decisions. Therefore, we will prepare a neural network that will be estimating different Q-values for each action. The difference between traditional Q-learning and deep Q-learning can be seen in Figure 2.1. We can see that the deep learning approach only requires the agent's current state as the input and uses function approximation to predict Q-values of every available action for the provided state as the output. Therefore, the input shape of the neural network is based on the shape of the state observation image after pre-processing and the output shape is equal to the number of all possible actions.

In other words, instead of using the tabular method for Q-value estimation, we can now estimate them using the function approximation, which not only gets rid of the necessity to store all state-value pairs in a table, but it gives agent the option to determine the Q-values of actions for the states it has never seen before or has only partial knowledge about, by using the values of similar or neighbor states.

The loss function ℓ for training then takes a very similar form as the policy for updating the Q-table, like it's described in [6]

$$\ell = (R_t + \gamma \cdot \max_a Q(s_{t+1}, a) - Q(s_t, a_t))^2, \tag{2}$$

where $Q(s_t, a_t)$ denotes the Q-value of performing the action a in state s at time step t.

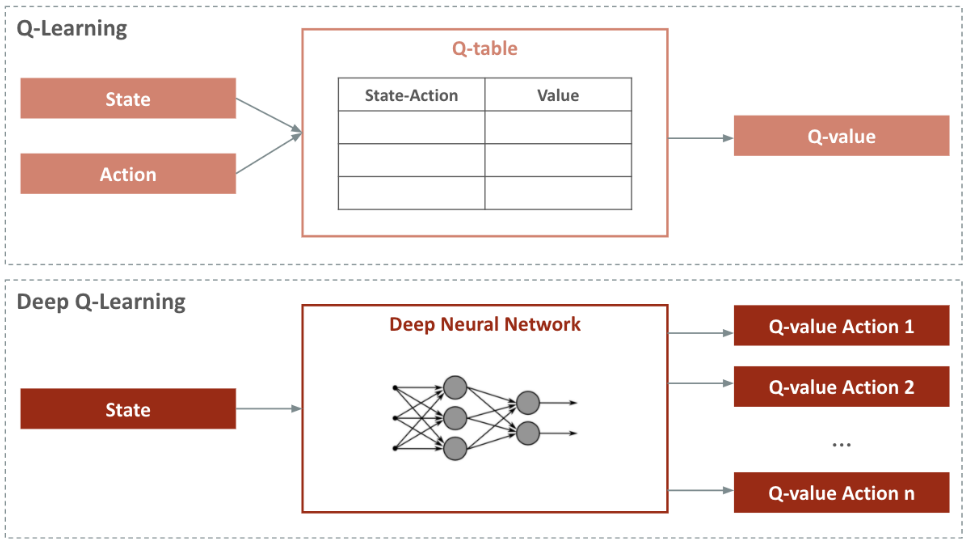


Figure 2.1: Comparison of basic Q-learning and Deep Q-learning principles. Picture from `https://miro.medium.com/max/3600/1*T4iXI6_jbaqVnsQwli-fog.png`.

Generally said, the DQN updates the network's weights in each step by computing the target (based on the Bellman's equation) and current values (based on the current predictions of the network) and feeding them to the the loss function. The the network's weights are then updated by back-propagating the computed loss value.

The introduction of the neural network with multiple layers allows us to predict Q-values more effectively, even for large environments with many possible actions for each state. It also enables us to use additional mechanisms, which would normally not be suitable for a simple Q-table approach, and these mechanisms allow us to achieve better performances. Mechanisms and approaches commonly used when working with deep reinforcement learning are described in the following subsections.

2.2.1 Pre-processing of the environment

Since we are dealing with neural networks, it's good not to omit some kind of general optimization to decrease the complexity of the model as much as possible. One of the most common ways to do it is by pre-processing the input that we feed to the neural network, which means reducing, modifying, or optimizing the state representation. When working with neural networks, you always want to study the input data and try to get rid of unnecessary features, lower the dimension of the input all in order to speed up the training of the neural network and the prediction evaluation process while losing as little information, by performing the pre-processing, as possible. In this section, we will describe commonly used pre-processing approaches which are used to deal with similar tasks and which we also decided to include in our system. [17, 18] More details about our implementation of the pre-processing techniques are described in Chapter 4.

For the Super Mario Bros. game, we have decided to use an implementation of the OpenAI's Gym environment on the Nintendo Entertainment System (NES) using the nes-py emulator (read more about it in Chapter 3). It provides us with all necessary state-related data. The state of the environment is represented as a two-dimensional RGB[2] pixel array with dimensions 240 by 256.

As part of our pre-processing, we have resized the RGB image into 84 by 84 and then we have converted it into grayscale representation. By converting the original image into grayscale, we have reduced the input image's channel size from three to only one, or in order words, decreased the dimension complexity of the input. We have done this based on our research, where we have found out that the majority of projects focusing on pre-processing similar game environments found that changing the dimensions to 84 by 84 decreases the input size while preserving all the essential information. For example, in the paper "Comparison of Deep Reinforcement Learning Approaches for Intelligent Game Playing" [10] they have used these modifications for pre-processing of Atari games. After resizing the input array, it is a good practice to normalize the pixel values into $(0, 1)$ interval.

Since performing almost any action takes the agent some time, it is also helpful to skip a few frames with each action. For example, when the agent wants to perform the jump action, it takes some time for the Mario character to actually perform the jump and land in the game. To tackle these scenarios, k frames (usually $k = 4$ is used) are skipped, and a total reward value is computed as a sum of all k skipped frames reward values for the given action. This technique makes the agent choose action on every k-th frame instead of on every frame, which saves a lot of computational time because running the emulator for multiple steps with the same action requires much less computational power than having the neural network to select an action for each consecutive frame. This technique allows the agent to play roughly k times more games without significantly increasing the run time. A similar idea is then also applied to avoid sending a copy of the same frame multiple

[2]RGB is an additive color model in which the red, green, and blue colors are added together in various ways to reproduce a broad array of colors.

times into the neural network. As Volodymyr Mnih described in [16], where he introduced the DQN model, the pre-processing steps are applied to the m most recent frames, and they are then stacked to produce the input to the Q-function. The m value is commonly set to $m = 4$, but he notes that the algorithm is robust to different values of m (for example, 3 or 5).

You can see the results of our pre-processing in Figure 2.2. By performing the pre-processing steps described above, we were able to reduce the number of computations required from the network during the training phase, and it also helped to speed up the training process.

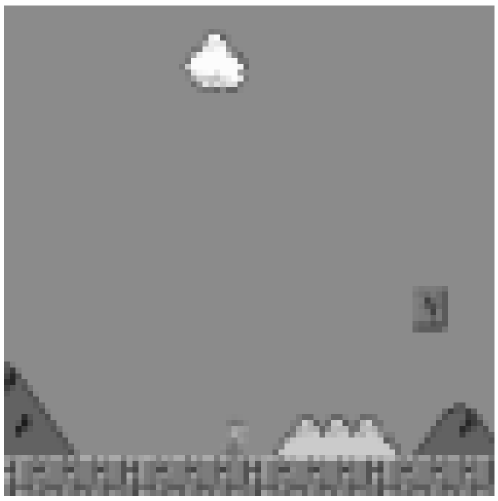


Figure 2.2: Super Mario Bros. game screen after pre-processing (downsampled to 84x84 grayscale).

2.2.2 Two separate networks technique

Using one neural network for both computing the predicted Q-value and also using it as the target value in the loss function often leads to unpleasant instabilities. In fact, the learning algorithm often becomes unstable and diverges.

Let us first explain why updating the Q-value may suddenly lead to an unstable algorithm in DQN when it's fine to use it in a basic Q-learning approach. With Q-learning, you are updating exactly one state-action value at each time step, whereas with DQN you are updating many. The problem here is that this can cause a situation, where the updated action can affect the action values for the very next state which the agent will be in instead of guaranteeing state stability like it is the case in Q-learning. This scenario, unfortunately, happens all the time with DQN when using standard deep neural network architectures. This effect is called catastrophic forgetting[3]

[3]More information on catastrophic forgetting problem can be found here: `https://towardsdatascience.com/guiding-forgetful-machines-72d1b8949138`.

and may cause endless loops in the learning mechanism.

In order to prevent all this from happening, a second neural network is introduced. Therefore, we are working with two neural networks that both have the same architecture, and we call them local and target networks. The local network's weights are updated in each step of our training process. Because of that, the local network is sometimes referred to as the online network. The target network is a periodical copy of the local network. Target network allows the learning mechanism to perform a defined number of training steps with current Q-values for each action before actually saving the newly computed values and then starting to use them in the update process. This gives the network an option to consider more actions that have taken place recently instead of updating in each iteration. This is all done in order to find better values before it starts using them to take actions. By this, we are escaping the unpleasant forgetting dilemma which has been mentioned above.

We, therefore, have two separate Q-networks. We use $Q(s, a; \theta)$ notation for the local network and $Q(s, a; \theta^-)$ for the target network. At the i-th iteration, the current θ_i parameters are updated with the aim to minimize the mean-squared Bellman error with respect to saved θ^- parameters, by optimizing the following loss function,

$$L_i(\theta_i) = \mathrm{E}[(R + \gamma \cdot \max_{a'} Q(s', a'; \theta_i^-) - Q(s, a; \theta_i))^2] \tag{3}$$

as further described in [18]. Here, the notation of s and s' is used, but it denotes the same thing as s_t and s_{t+1} that were used in previous formulas. This alternative notation is sometimes used to make complex formulas more readable. Furthermore, for computing the target Y_t^{DQN} (value used for computing the loss function's value) we would get the following formula,

$$Y_t^{DQN} = R_{t+1} + \gamma \cdot \max_a Q(s_{t+1}, a; \theta_t^-). \tag{4}$$

So, in a few words, the two networks technique introduces the target network, and it is a copy of the local network. After a defined number of steps of the learning phase during which only the local network is updated, we copy all the weight values of the local network to the target network. The number of steps after we copy the weights is a hyper-parameter of our system. The most commonly used number of steps after which the weight values are copied is around a few thousand, but its always good to experiment with the parameter's actual value to get a value that suits the current problem the most.

2.2.3 Experience replay

Imagine playing the Super Mario Bros. game (or generally any game based on a similar gameplay style). After you press the jump command, the playable character (Mario in our case) performs a jump. While mid-air, you can still turn to the sides and alter the direction of your jump a little, but mostly your available actions while mid-air will become very limited. This all means that we mostly know the real benefit of our selected initial action only after the jump is finished and Mario lands on a solid surface again. So between the start of the jump and landing, many actions may occur, but all of them will have similar reward value for our agent. These situations make learning from each executed action less effective.

The experience replay is a mechanism based on storing and replaying game states. The game states are commonly stored as a tuple of 4 values - state, action, reward, and next state. To perform experience replay we store the agent's experiences $e_t = (S_t, A_t, R_t, S_{t+1})$ at each time-step t in a data set $D_k = \{e_1, \ldots, e_k\}$. During the learning phase, we apply Q-learning updates on samples of experience $(S, A, R, S^{'}) \sim U(D)$, drawn uniformly at random from the pool of stored samples.

That means that after the first k actions, the local network's weights are updated with a sample batch of experiences from the memorized last k experiences. Therefore k is another hyper-parameter we need to introduce into our model, and its value needs to be fine-tuned. The introduction of this technique does not require us to change the loss function formula since the only difference is only which S, A, R, $S^{'}$, $A^{'}$ values we feed into it.

This trick helps to speed up the Deep Q-Learning's learning process because having the right model parameter update frequency is important. If you update model weights too often (e.g., after every step), the algorithm will learn very slowly because, in most cases, not much has changed (see the example at the start of this section) between neighbor situations. The same goes for the other scenario. When you choose the size of the experience batch too large, then this mechanism can miss selecting values, which can be beneficial because it will have to choose randomly from a large batch. Therefore the chance of selecting a less optimal update can occur with a higher probability. Another advantage of this technique is caused by the fact that Q-learning updates are incremental and do not converge quickly, so multiple passes with the same data are beneficial, especially when there is low variance in immediate outcomes given the same state, action pair. More details and analysis of this technique can be found in [16].

2.3 Q-learning modifications

We did research on Q-learning modifications in order to find the one which would be the most relevant for our problem. We have found various works all inspired by the Q-learning algorithm principle. Q-learning inspiration led to creation of similar algorithms, such as Delayed Q-learning [22], Phased Q-learning [11], and Fitted Q-iteration [5], etc. These modifications have in common their purpose. They have been created to speed up the convergence rates of the original algorithm. But one of the modifications stood out from the other. It is called Double Q-learning, and it was created by Hado van Hasselt as a solution to the Q-learning's overestimation problem. In this section, this problem is further described along with the basic idea behind Double Q-learning, and its deep neural network adaptation.

2.3.1 Double Q-learning

Even though Q-learning is one of the most popular reinforcement learning algorithms, it can perform poorly in some stochastic environments. Hado van Hasselt focuses on this problem in his paper Double Q-Learning [8] and points out that Q-learning's performance is negatively influenced by the large overestimation of action values caused by the maximization approach in the Q-value update formula. Because of this, the algorithm is known to sometimes learn unrealistically high action values, which leads to poor performances of the standard Q-learning technique.

To briefly demonstrate the problem of the algorithm, let us present you with a simple example inspired from [20]. Consider a Markov decision process having four states: A, B, C, and D. Two of these states, C and D, are terminal. State A is the starting state, where the agent has the possibility of taking two actions, either Right or Left. The Right action gives him zero reward points (we will denote the reward value as R), and the agent then lands in terminal state C. The Left action moves the agent to state B also with zero reward value. State B offers the agent several actions, and all of them move the agent to terminal state D. However, the reward R of each action from B to D has a random value from a normal distribution $\mathcal{N}(-0.5, 1)$. The visualization of the described scenario is shown in Figure 2.3.

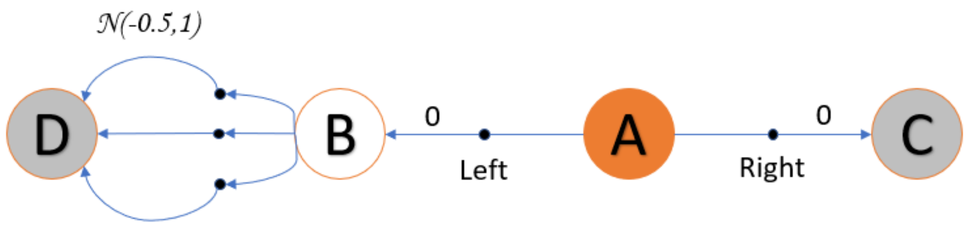


Figure 2.3: Example describing the overestimation which occurs in Q-learning. Image taken from [20].

The expected value of R is in our example given to be $\mathrm{E}(R) = -0.5$. This means that over a large number of experiments, the average value of R is less than zero. Based on this assumption, it is clear that performing the Left action from state A is always a bad idea. However, because some of the values of R are positive, Q-learning will incorrectly consider performing the Left action from state A as it maximizes the reward. In reality, this is a bad decision because even if it works for some episodes, it is guaranteed to bring a negative reward in the long run.

As an outcome of previous observations, it's evident that the main cause of the value overestimation can be solved by focusing on the maximization operator. The problem with the operator is that in both standard Q-learning and DQN update formulas, as shown in (1) and (2), the same values are used to select and evaluate the action. This makes the algorithm more likely to select overestimated value in each step, resulting in overoptimistic value estimates as was proved in [9]. To resolve this issue, Hado van Hasselt has proposed the Double Q-Learning method in his paper [8]. The main idea was to decouple the selection from the evaluation. In his work, he introduced the terms Single Estimator and Double Estimator and mathematically described and proved that the Single Estimator used in the standard Q-learning is biased when estimating the optimal value. He then proposed an alternative approach, the use of the Double Estimator, which uses two estimators instead of one. Based on its principles, he introduces the Double Q-learning algorithm, which is able to handle the overestimation issue.

His proposed algorithm, the Double Q-learning, stores two Q functions instead of one and can be seen in Algorithm 1. Each Q function is updated with a value from the other Q function for the next state. It is important to note that both Q functions learn from separate sets of experiences. To select an action for the agent to perform, one can use any of the value functions. The Q functions are updated on the same problem, but with a different set of experience samples, and because of that, it can be considered an unbiased estimate for the value of this action. Because of storing two Q functions, the algorithm is not less data-efficient than standard Q-learning.

Algorithm 1: Double Q-Learning

```
Initialize $Q^A$,$Q^B$, $s$
repeat
    Choose $a$, based on $Q^A(s,\cdot)$ and $Q^B(s,\cdot)$, observe $r$, $s'$
    Choose (e.g. random) either UPDATE($A$) or UPDATE($B$)
    if UPDATE(A) then
        Define $a^* = \operatorname{argmax}_a Q^A(s',a)$
        $Q^A(s,a) \leftarrow Q^A(s,a) + \alpha(s,a) \cdot (r + \gamma Q^B(s',a^*) - Q^A(s,a))$
    end
    else if UPDATE(B) then
        Define $b^* = \operatorname{argmax}_a Q^B(s',a)$
        $Q^B(s,a) \leftarrow Q^B(s,a) + \alpha(s,a) \cdot (r + \gamma Q^A(s',b^*) - Q^B(s,a))$
    end
    $s \leftarrow s'$
until end
```

Hasselt has also analyzed and compared Double Q-learning to the standard Q-learning approach and confirmed that his algorithm does not suffer from the overestimation bias as the Q-learning does and also noted that Double Q-learning sometimes underestimates the action values, which is more pleasant than overestimating them.

2.3.2 Double Deep Q-learning

Double Deep Q-learning (also referred to as Double DQN or DDQN) is the outcome of applying the main idea behind the Double Q-learning algorithm to the deep neural network version of the Q-learning problem. As Hado van Hasselt describes in [9], the overoptimization problem also occurs when it comes to the DQN approach. In his earlier work, [8] he proposed the Double Q-learning algorithm for solving the issue, and he tried to apply the same concepts to the neural network approach.

As a natural way of implementing his Double estimator principle (see Section 2.3.1) seemed to be to take advantage of the already commonly applied concept of two separate networks (we talk about it in Section 2.2.2). The target network in the DQN architecture provides a natural candidate for the second value function without the need to introduce additional networks. He, therefore, proposes to evaluate the greedy policy according to the local network but use the target network to estimate its value. The resulting algorithm is called Double DQN. Its update is the same as for DQN, but it differs when it comes to the computation of the target value Y_t^{DQN}. The formula for computing target value Y_t^{DQN}

$$Y_t^{DQN} \equiv R_{t+1} + \gamma \cdot \max_a Q(s_{t+1}, a; \theta_t^-) \tag{5}$$

is replaced with

$$Y_t^{DoubleDQN} \equiv R_{t+1} + \gamma \cdot Q \cdot (s_{t+1}, \operatorname*{argmax}_a Q(s_{t+1}, a; \theta_t), \theta_t^-). \tag{6}$$

It's good to compare the original Double Q-learning solution with Double DQN. In the first one, two value functions learn by randomly assigning the experiences to update one of the two value functions. This means that we are working with two sets of weights, θ, and θ'. In each update step, one group of weights determines the greedy policy, and the other determines its value. But in Double DQN, the weights of the second (local) network θ_t' are replaced with the weights of the target network θ_t^- for the evaluation of the policy (prediction of the next action for the agent). A more detailed description of the differences can be found in [9].

Hado van Hasselt in detail measured and compared the performance of Double DQN and standard DQN implementations on many different Atari games[4]. Since many of the games he tested on had not only different environments but also required different play styles from the agent, we can consider his observations game independent and, therefore, a sufficient measuring technique. After observing the test outcomes, he pointed out how the learning curves of DQN consistently end up much higher than the actual discounted value of the best-learned policy.

In conclusion, it is clear that the Double DQN algorithm improves over DQN both in terms of value accuracy and in terms of policy quality. According to Hasselt's research, reducing overestimation can significantly benefit the overall stability of the learning process. [9]

2.4 Asynchronous Advantage Actor-Critic

Even though Q-learning is the most popular and used approach when tackling the reinforcement learning problems, there are also other approaches, where some of them are worth mentioning, because they could bring better performances for our system and agent respectively. For example the Temporal Difference Learning, Advantage Actor-Critic, Asynchronous Advantage Actor-Critic, Deep Deterministic Policy Gradient, and many more. Brief description of the techniques mentioned above can be found in Deepanshu Mehta's paper *State-of-the-Art Reinforcement Learning Algorithms* [4]. One of them is the Asynchronous Advantage Actor-Critic model, often denoted as A3C, which was first proposed in 2016 by Volodymyr Mnih in [15]. A3C is considered to be the main competitor for the state-of-the-art reinforcement learning model after the Q-learning approach. In the subsections below, we will discuss the principles of the A3C approach and also how we can tackle the overestimation issue which it suffers from.

2.4.1 A3C vs. DQN

In this section we will first discuss the similarities and differences of the A3C approach when compared to the Q-learning, to which we have dedicated most of our focus in the previous sections, and then we will describe the core principles and ideas behind A3C in more detail.

The main principles and building blocks of the DQN and A3C are the same. Both have the same requirements for the environment functionality and both benefit from pre-processing of the environment in the same way. Also, both use Q-Learning as an off-policy control method to find the optimal policy. While implementing DQN we can implement the local and target networks and apply the concept of experience replay (see Sections 2.2.2 and 2.2.3), but when it comes to A3C we can't apply these techniques directly. But the com-

[4]Some of their famous games that he used in his research can be found here: `https://en.wikipedia.org/wiki/List_of_Atari_SA_video_games`.

bined key idea of introducing separate networks and learning from multiple experiences is included in A3C, where the experiences are collected and processed asynchronously by multiple independent workers in parallel on their own copies of the environment.

For both approaches, the training phase depends heavily on the system's choice of the hyperparameters (as it generally does in most RL tasks). These hyperparameters are similar for both of them. For A3C they are the number of experiences to use for each training, the frequency of training step, learning rate value, number of workers and value of the discount factor (used in updates of the Q-values) and many others. The training phase with the set parameters then has the same flow for both approaches. So in conclusion on what's similar between the two approaches we can say that their core concepts are the same and for now it looks like they are almost identical, so lets discuss the difference, which A3C brings to tackle the reinforcement learning problem.

The main difference is that DQN learns an action value function and defines the policy from that value function, while A3C learns both the policy and value function. The name A3C concludes of the terms Actor and Critic, which represent the main components of this approach. Actor represents the policy and Critic represents the value function. Both of them are implemented as fully connected linear layers on top of the network. When Actor and Critic are computing their outputs to compute policy and value for the network's weights update, they process the input game state through the shared neural network architecture and then process the output each through their individual fully connected layer. So, in short, we can say that both Actor and Critic are separate fully connected linear layers on top of the shared DQN architecture. In the learning phase, the Critic is used to update the Actor, or in other words, the value function is used to update the policy.

The principle, which we have just described is by itself called the Advantage Actor-Critic, A2C in short. The difference between A2C and A3C is that in A3C, the actor-critic concept is enhanced by the introduction of multiple workers, each represented by the same deep neural network architecture. Each worker explores its own independent copy of the environment with its own parameters for the neural network. They each learn experiences and update the so-called global network (see Figure 2.4).

Workers in A3C act concurrently and optimize the deep neural network through asynchronous gradient descent. After computing the gradients, they send the updates to the global network after every t_{max} actions (steps) or when a terminal state is reached. Whenever the global network is updated, it propagates new weights to the workers to guarantee they share a common policy. Two cost functions are associated with the two DQN outputs (from Actor and Critic) of each worker. As described in [1], for the policy function it is

$$f_\pi(\theta) = \log \pi(a_t|s_t;\theta)(R_t - V(s_t;\theta_t)) + \beta H(\pi(s_t;\theta)), \tag{7}$$

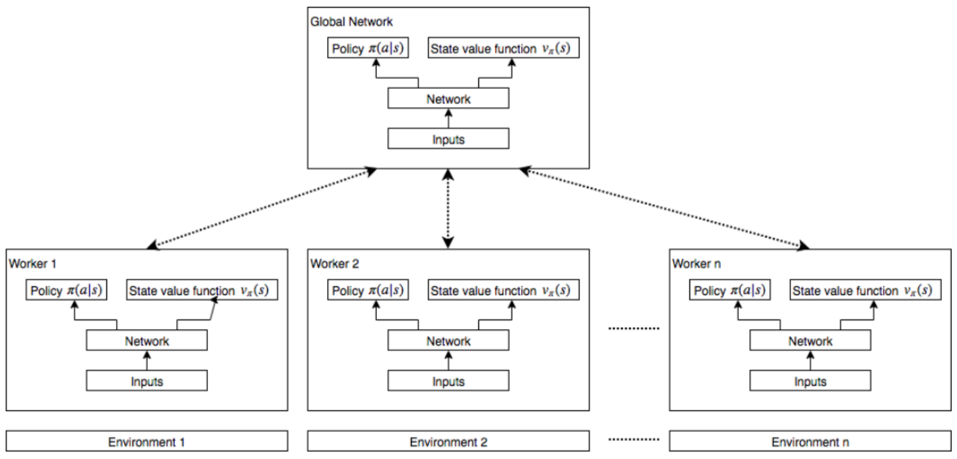


Figure 2.4: Visualization of an A3C model principle. Taken from [10]

where θ_t are the values of the parameters θ at time t, $R_t = \sum_{i=0}^{k-1} \gamma^i r_{t+i} + \gamma^k V(s_{t+k}; \theta_t)$ is the estimated discounted reward in the time interval from t to $t+k$ and k is upper-bounded by t_{max}, while $H(\pi(s_t; \theta))$ is an entropy term, used to favor exploration during the training process. $V(s_t; \theta)$ is the value function with parameters θ, where the notation $V^\pi(s_t)$ is defined as $V^\pi(s_t) = \mathrm{E}[R_t|s_t]$ which is the expected return for following the policy π in state s_t. The factor β controls the strength of the entropy regularization term. The cost function for the estimated value function is:

$$f_v(\theta) = (R_t - V(s_t; \theta))^2. \tag{8}$$

Training is then performed by collecting the gradients $\nabla\theta$ from both of the cost functions. The computed gradients can be either shared or separated between worker threads but Mnih in his work [15] notes that the shared implementation is known to be more robust.

2.4.2 Dealing with overestimation in A3C

When it comes to RL approaches tackling similar tasks, some issues are shared for most of them. Since A3C and DQN both use similar architectures and update policies, it is no surprise that they also share some weak points. As we have discussed in Section 2.3, one of the more severe issues of the DQN approach is the overestimation bias of learned Q-values. Even though A3C may seem resistant to overestimation thanks to the fact that multiple workers are solving the issue almost independently, the overestimation is still present with the A3C approach, as Scott Fujimoto proves in his work [7]. In this section, we will talk about the overestimation bias in the actor-critic concept and what adjustments need to be made to minimize it as much as possible.

A3C uses gradient descent for updating the policy and even though the overestimation is very low within each update, it is still present and can be a cause of concern in some tasks. This seemingly minor overestimation may gradually, over many steps (updates), develop into a more significant bias. Also, inaccurate value estimates in most cases lead to poor policy updates. Fujimoto in his research notes in [7] claims that he evaluated several approaches (most of them inspired by the techniques which are effective against overestimation in the DQN approach) for dealing with the reduction of the overestimation but found them ineffective in an actor-critic setting. Apart from other, he was also aware of the Double DQN algorithm created by Van Hasselt in [9], but specifically mentions that even though this technique helped to overcome overestimation in DQN, it is unfortunately ineffective in A3C and that is mainly because of the slow-changing policy. The current and target value estimates remain too similar to avoid overestimation bias. Further in his work, he describes how he was able to find a solution for the issue thanks to a much simpler Double Q-learning algorithm (see Section 2.3.1) instead of using its network-based alternative.

As Fujimoto noted in [7], the Double Q-learning can be adapted to an actor-critic format by using a pair of independently trained critics in each worker instead of using only one critic per worker. Inspired by the Double Q-learning algorithm, he proposed a modified version of it which he named Clipped Double Q-learning. Based on its principles he then proposed Twin Delayed Deep Deterministic policy gradient algorithm with which he was able to minimize the overestimation issue of the A3C model.

Now we will explain Fujimoto's Clipped Double Q-learning algorithm with a focus on the core idea behind it. If we denote a pair of actors as $(\pi_{\phi_1}, \pi_{\phi_2})$ and pair of critics as $(Q_{\theta_1}, Q_{\theta_2})$, where π_{ϕ_1} is optimized with respect to Q_{θ_1} and π_{ϕ_2} with respect to Q_{θ_2}, we then get the formulas for its respective update estimates as

$$\begin{aligned} y_1 &= R + \gamma Q_{\theta'_2}(s', \pi_{\phi_1}(s')) \\ y_2 &= R + \gamma Q_{\theta'_1}(s', \pi_{\phi_2}(s')). \end{aligned} \tag{9}$$

We can see that π_{ϕ_1} optimizes with respect to Q_{θ_1}. If we would use an independent estimate in the target update of Q_{θ_1} we would successfully avoid the bias introduced by the policy update thus it would help us with minimizing the overall overestimations. Sadly, the critics cannot be considered entirely independent due to the fact that in the formula we use the opposite critic in the learning targets. Because of that the value of $Q_{\theta_2}(s, \pi_{\phi_1}(s))$ will be greater than $Q_{\theta_1}(s, \pi_{\phi_1}(s))$ for some states s. This is problematic because $Q_{\theta_1}(s, \pi_{\phi_1}(s))$ generally overestimates the true value. To address this problem, Fujimoto proposed to simply upper-bound the less biased value estimate Q_{θ_2} by the biased estimate Q_{θ_1}. If we then take the minimum between the two estimates, we get the update estimate formula for the Clipped Double

Q-learning algorithm:

$$y_1 = R + \gamma \min_{i=1,2} Q_{\theta'_i}(s', \pi_{\phi_1}(s')). \tag{10}$$

With Clipped Double Q-learning, the value target cannot introduce any additional overestimation when compared to the standard Q-learning solution. This proposed update formula may cause a completely opposite effect on the estimated values as it may induce an underestimation bias. But in fact, having the underestimation bias is far more preferable to the overestimation one because, unlike overestimated actions, the value of underestimated actions will not be explicitly propagated through the policy update.

The adjustment explained above, together with the application of the target network, delayed policy update (the core benefits of the two separate networks technique explained in Section 2.2.2), and the target policy smoothing regularization (inspired by Sutton and Barton's [23]) led to the creation of the Twin Delayed Deep Deterministic policy gradient algorithm, called TD3 in short. TD3 consists of one actor and a pair of critics. In every step, the pair of critics is updated with respect to the minimum target value of the actions which are selected by the neural network using the target policy:

$$y = R + \gamma \min_{i=1,2} Q_{\theta'_i}(s', \pi_{\phi'}(s') + \epsilon), \tag{11}$$

where $\epsilon \sim \texttt{clip}(\mathcal{N}(0, \sigma), -c, c)$ is the added noise that is clipped with the purpose to keep the target value close to the original action and c represents hyperparameter of the model. Every d iterations (in the proposal paper [7] originally set to $d = 2$), the policy is updated with respect to Q_{θ_1} following the deterministic policy gradient algorithm introduced by Silver in [21]. Full pseudocode of the TD3 can be seen in Algorithm 2.

Algorithm 2: Twin Delayed Deep Deterministic policy gradient

```
Initialize critic networks $Q_{\theta_1}$, $Q_{\theta_2}$, and actor network $\pi_\phi$
with random parameters $\theta_1$, $\theta_2$, $\phi$
Initialize target networks $\theta'_1 \leftarrow \theta_1$, $\theta'_2 \leftarrow \theta_2$, $\phi' \leftarrow \phi$
Initialize replay buffer $\mathcal{B}$
for $t = 1$ to $T$ do
    Select action with exploration noise $a \sim \pi_\phi(s) + \epsilon$,
    $\epsilon \sim \mathcal{N}(0, \sigma)$ and observer reward $R$ and new state $s'$
    Store experience tuple $(s, a, R, s')$ in $\mathcal{B}$

    Sample mini-batch of $N$ experiences $(s, a, R, s')$ from $\mathcal{B}$
    $\tilde{a} \leftarrow \pi_{\phi'}(s') + \epsilon$, $\epsilon \sim \texttt{clip}(\mathcal{N}(0, \tilde{\sigma}), -c, c)$
    $y \leftarrow R + \gamma \min_{i=1,2} Q_{\theta'_i}(s', \tilde{a})$
    Update critics $\theta_i \leftarrow \mathrm{argmin}_{\theta_i} N^{-1} \sum (y - Q_{\theta_i}(s, a))^2$
    if $t$ mod $d$ then
        Update $\phi$ by the deterministic policy gradient:
        $\nabla_\phi J(\phi) = N^{-1} \sum \nabla_a Q_{\theta_1}(s, a)|_{a=\pi_\phi(s)} \nabla_\phi \pi_\phi(s)$
        Update target networks:
        $\theta'_i \leftarrow \tau\theta_i + (1 - \tau)\theta'_i$
        $\phi' \leftarrow \tau\phi + (1 - \tau)\phi'$
    end
end
```

Chapter 3

Tools for environment interaction

One of the goals of this thesis was to select an appropriate tool that enables the learning framework to interact with the game's environment. Optimally, we wanted to find a tool that would require only a little, ideally no adjustments from us, so we could focus solely on the RL part of the task and not the interface implementation.

In this chapter are described the considered tools, the requirements we set when searching for the ideal tool, which would enable us to interact with the game environment, and which one we decided to use in the end. We then describe how it works and what information it can provide for our agent to work with, and we also explain in detail the format of the provided information.

Before describing the available tools, we first describe Super Mario Bros. core game mechanisms, movements, and basic goals of the game in Section 3.1, because that's what was studied first in order to understand exactly what we need. After we had been introduced to the game principles, we knew that we needed to find a tool that would allow us to run the game, easily input action commands to control Mario (the agent) and at the same time be able to receive output which would be in a suitable format and can reflect the current state of the environment. Starting from Section 3.2, we have described some available tools which we found and the one which proved to be the most suitable for our game.

3.1 Super Mario Bros.

Super Mario Bros. is a very old platform video game from 1985. It was created by Nintendo[5] and is still, by some, recognized as one of the greatest video games of all time.

[5]`https://www.nintendo.com`.

In the game, the player controls the main character called Mario, and travels with him through different variations of environments, also called levels of the game. The goal of each level is to get Mario to the end of the environment with at least one health point (HP) remaining. Mario has three HP at the very start of the game, and whenever he loses one, the current environment (level) is restarted, and he needs to go through it again from the start. The whole game ends when Mario runs out of lives.

Apart from Mario, there are a few types of obstacles in the environment as well. One group of obstacles is the static ones. Those can be some barriers of various heights which can be jumped over or some pits of various widths, which also need to be jumped over. Barriers themselves are harmless and require only a proper jump to overcome, but if Mario falls into a pit, then he loses one life point. Then there are also moving obstacles, the foes. These can take various forms and can perform various actions like jumping, firing projectiles, and other. The important thing to note is that if Mario touches an enemy or its projectile then he loses one life point. Mario also has a way, how he can destroy the enemies. A very common technique that works on almost all enemies is landing on top of them, which destroys them.

Mario has a defined set of actions that he can perform. The standard controls are moving to either right or left, jumping, ducking, and dashing. Of course, various tricks and very efficient game-plays can be performed by combining (performing) multiple standard actions simultaneously. For example, executing jump and move right actions together is a standard move required to jump over obstacles.

3.2 DeepMind Lab

One of the tools considered was DeepMind Lab. It is an interesting open-source project made with the aim to prepare a flexible and powerful tool for training agents in various environments in order to push the boundaries of Artificial Intelligence (AI). The project's goal was to develop a system, where agents can learn to solve any complex problem without needing to be taught how.

The documentation of the tool mentions the possibility of performing various RL tasks with it like navigating in mazes, traversing dangerous passages while avoiding falling off cliffs, or quickly learning and remembering random procedurally generated environments. It also notes that it is made to learn agents automatically from the raw inputs and reward signals from the environment while also allowing high customization, which is exactly what we were looking for. In DeepMind Lab's GitHub repository[6] various environment implementations can be found with different agent configurations for performing a rich spectrum of tasks.

[6] `https://github.com/deepmind/lab`.

Sadly none of the available environments was similar to the one we were looking for. That was mainly due to the fact that this tool is made for three-dimensional environments and our game environment is two-dimensional. This would really complicate things because adjusting the tool and creating our own environment for the game would completely transform the complexity of our task. More details about the tool can be seen in its paper [2].

3.3 Psychlab

Since the DeepMind Lab seemed like a very useful tool, we decided to dig more deeply into it and tried to find if there exists a modification that would be more suitable for two-dimensional environments. We found that many projects and adjustments were built on top of it, but the only one which we thought was worth mentioning was the Psychlab.

Psychlab is an open-source platform built on top of DeepMind Lab with the intention to better understand the behaviors of artificial agents. Because of that, it allows to directly apply methods from fields like cognitive psychology to study the behaviors of artificial agents in a controlled environment. In order to do that, they used the DeepMind Lab to model an environment that consists of an agent sitting in front of a virtual computer monitor and responds to the onscreen tasks.

Even though this project was done with the aim to study the psychological behavior of the agents, we found it interesting (and potentially useful) that their environment transformed the three-dimensional DeepMind Lab into some sort of two-dimensional one. Unsurprisingly, not even for this project is an environment of Super Mario Bros. implemented, and in order to use this tool, we would need to modify Psychlab's environment and implement the game itself with all the input and output mechanisms into the virtual computer monitor. More details about the Psychlab tool can be found in their paper [14].

Since our main focus is set on studying and implementing RL principles, implementing such an environment would consume a lot of our time on something which is irrelevant to the goals of this thesis. Luckily, a much more suitable tool has been found, which proved to be ideal for our task, and it is described in the next section.

3.4 Gym

The Gym is an open-source project made directly for RL tasks created by OpenAI organization[7]. Gym is a toolkit and library for developing and comparing reinforcement learning algorithms. It is a collection of environments

[7]`https://openai.com`.

(RL tasks) that can be used to test desired RL algorithms. It makes no requirements on the form of the agent and is compatible with all python libraries which are commonly used in Machine Learning (ML) tasks. Gym is very easy to install and provides an API[8] which serves as a shared interface for all its environments, making it very easy to get familiar with using different environments within the Gym ecosystem. More details about the OpenAI's Gym project can be found in [3].

To show how easy it is to use, an example is provided in Code 1 which runs an instance of the CartPole game[9]. Notice that to run an environment

```
import gym
env = gym.make('CartPole-v0')
env.reset()
for _ in range(1000):
        env.render()
        # take a random action
        env.step(env.action_space.sample())
env.close()
```

Code 1: Gym - CartPole example from `https://gym.openai.com/docs`.

only a few simple commands are needed. One to create the environment, one to reset it to its initial values, one to close the environment at the end, and two commands which are often used in a loop to perform an action and render it to the user. These shared commands can be used for any environment and make using Gym environments very user-friendly. Note that the step function always allows the agent to perform an action from a defined set of actions, and it returns information about the environment upon performing given action. This information of course differs for each environment, and the same goes for the action set.

The installation of the Gym library can be done via PIP[10] and the installation steps along with some examples of how to use the library can be found in their GitHub repository[11].

So, in conclusion, the Gym library is a very strong tool that enables us to use its interface and solely focus on the implementation of the RL algorithms, agent's behavior, etc. It also works and is mostly used for two-dimension environments, which was the reason why we didn't go with DeepMind's Lab (see Section 3.2). Gym provides a few environments already in its repository, but it also offers the option for others to implement any other environment they

[8] API stands for Application Programming Interface.
[9] CartPole is a game in which you try to balance the pole as long as possible.
[10] PIP is a package manager for Python packages.
[11] `https://github.com/openai/gym`.

want under the same license (more about it in Section 3.4.3) by using its shared interface.

3.4.1 Gym Retro

The only thing, which was left for us to do was to find a Gym environment implementation of the Super Mario Bros. game. We found out that there is a GitHub repository[12] that is also from OpenAI and is called Gym Retro which lets users turn classic video games into Gym environments. It uses various emulators that make the visualizations of the games possible on almost any device. More details about Gym Retro can be found in [19].

Even though that it contains around a thousand game environments, there is no Super Mario Bros. environment present, but after further searching, we found another GitHub repository, which uses the OpenAI's repositories and implements environments for Super Mario Bros. and even Super Mario Bros. 2. Since this tool is exactly what we were looking for, we have described it more deeply in the next section.

3.4.2 gym-super-mario-bros

GitHub repository gym-super-mario-bros[13] is an implementation of an OpenAI's Gym environment for Super Mario Bros. and Super Mario Bros. 2 created by Christian Kauten. The environment's GUI[14] runs on The Nintendo Entertainment System (NES) using the nes-py emulator[15].

Environment implementation can be easily installed using PIP. Since it is an implementation of OpenAI's Gym environment, it follows Gym's shared interface and, therefore, the game can be controlled with the same commands as other environments implemented under it (see code snippet and shared commands description in Section 3.4).

In addition to the shared commands, when initializing the environment, the user can choose from three sets of actions, which the agent will have available during the game. This gives us the option to control the size of the action space if needed. The defined action sets are named RIGHT_ONLY, SIMPLE_MOVEMENT, and COMPLEX_MOVEMENT. Details of the actions list can be found in the actions.py file in the GitHub repository, but in short, we can say that the RIGHT_ONLY consists of only five actions, where four of them represent pressing the right button alone or in combination with another button. SIMPLE_MOVEMENT consists of basic control actions for the game, which are usually presented to the user in the game

[12]`https://github.com/openai/retro`.
[13]`https://github.com/Kautenja/gym-super-mario-bros`.
[14]GUI stands for Graphical User Interface.
[15]`https://github.com/Kautenja/nes-py`.

Table 3.1: Contains of the info dictionary returned by the step function

Key	**Type**	**Description**
coins	int	The number of collected coins
flag_get	bool	True if Mario reached a flag or an ax
life	int	The number of lives left, i.e., {3, 2, 1}
score	int	The cumulative in-game score
stage	int	The current stage, i.e., {1, ..., 4}
status	str	Mario's status, i.e., {'small', 'tall', 'fireball'}
time	int	The time left on the clock
world	int	The current world, i.e., {1, ..., 8}
x_pos	int	Mario's x position in the stage (from the left)
y_pos	int	Mario's y position in the stage (from the bottom)

tutorial. COMPLEX_MOVEMENT contains the same actions as the SIMPLE_MOVEMENT action set, but in addition to them, it also contains many other actions which are combinations of multiple buttons pressed at the same time. Also, it is good to mention that each action set includes the NOOP action, which doesn't make the agent do anything. It is just an empty action that can be useful for checking the agent's state.

Step function

The step function is the most important function for controlling the agent. It takes one argument, which is the action to perform from the action list with which the environment was initialized. After sending the action command to the game, the step function outputs four variables which are called state, reward, done, and info.

State is a two-dimensional array that holds all the pixel values of our game screen, which the agent sees. Done is a Boolean[16] variable which indicates the state of the game. True value signalizes that the environment reached its terminal state, which is either when the agent wins the current level or when he runs out of lives (loses the game). Info is a dictionary that contains the information described in the Table 3.1. Reward is an integer value computed from the game's current state. In the next paragraph, we will explain how the reward value is computed in the gym-super-mario-bros library, but it is important to mention that the reward function is a crucial part of the RL algorithm, and therefore it is good to consider changing it if the performance of the trained agent is not up to expectations.

The reward function which is implemented in the library assumes that the objective of the game is to move as far right as possible, as fast as possible,

[16]Boolean is a data type, where its variables have one of two possible values (usually denoted True and False), which represent the two truth values of logic in algebra.

without dying, which makes sense because to win the current level you always have to get to the most right point, where the finish flag is. To model this, three separate variables compose the reward: v, c, and d. Their summed value is then clipped into the range $(-15, 15)$. The formula for the reward value therefore takes the form:

$$R = \texttt{clip}(v + c + d, -15, 15). \tag{12}$$

Variable v represents the difference in the agent's x-position values between consecutive states which reflects the current speed of the agent. From the v value we can also get the information about the direction in which the agent is moving. If $v > 0$ then the agent is moving right, $v < 0$ means that he is moving left and $v = 0$ means he is not moving.

Variable c serves as a penalty factor in the reward function to prevent the agent from standing still. It represents the number of passed game clock ticks between the frames and is either equal to zero or to a negative number. We can understand the value of $c = 0$ as that there has been no clock tick between the frames and if $c < 0$ then there has been a clock tick.

Variable d is a death penalty that penalizes the agent for dying in the current state. This penalty encourages the agent to avoid death. The penalty value for being alive is equal to 0, and for being dead the value of d equals to -15.

Custom environment modifications

In section 3.4, we have talked about the shared interface and methods which OpenAI's Gym offers. But we didn't mention that you can also modify the behavior of these methods. We have mentioned that you need to implement the behavior of these methods when creating a new environment representation within the Gym's library, but it is also useful for our situation, where we have the environment already implemented, but we want to modify it just a little bit to make it easier for us to work with it.

In Section 2.2.1, we have talked about pre-processing. One of its steps is to reduce the size of the input which we feed to our neural network. Thanks to Gym's library, we can use the functionality of so-called wrappers to modify the behavior of some pre-defined methods for the Mario environment. In order to do that we have to implement a so-called wrapper class which inherits one of the gym.Wrapper, gym.ObservationWrapper, gym.RewardWrapper or gym.ActionWrapper classes. Implementing such wrapper enables you to redefine different methods which are shared within the Gym's environment.

For us, the most useful was to implement wrapper classes that inherit the gym.ObservationWrapper class which allowed us to modify the observations (the game states) which are returned after performing any action. We have implemented multiple classes which inherit this wrapper to apply the pre-processing steps for each state observation returned.

We have also implemented a wrapper class that inherits the gym.Wrapper which allowed us to modify the step function, which we needed to modify in order to apply the frame skipping principle in our pre-processing (see Section 2.2.1 for details). In Code 2 you can see the implementation of the frame skipping technique. For a more detailed description about the wrap-

```python
"""
    Reimplements the step() function
    We perform the action 'skip'-times and collect
    total gained reward over skipped frames
"""
class SkipFramesInEnv(gym.Wrapper):
    def __init__(self, env = None, skip = 4):
        super().__init__(env)
        self._skip = skip

    def step(self, action):
        total_reward = 0.0
        done = False
        for _ in range(self._skip):
            state, reward, done, info = self.env.step(action)
            total_reward += reward
            if done:
                break

        return state, total_reward, done, info
```

Code 2: Gym wrapper class for re-defining the step function using the functionality of Gym's wrappers.

pers see [12].

Simple gameplay example

To test and demonstrate the environment's functionality, we have prepared a working example of controlling the Mario character programmatically. Our implementation can be seen in Code 3. The code initializes the environment, loads the SIMPLE_MOVEMENT action list, and performs the simple right action for a while, and then it waits two seconds before it closes the visual output window for the user.

```python
from nes_py.wrappers import JoypadSpace
import gym_super_mario_bros
from gym_super_mario_bros.actions import SIMPLE_MOVEMENT
import time

# load word 1 stage 1 v0 (standard ROM)
env = gym_super_mario_bros.make('SuperMarioBros-1-1-v0')
env = JoypadSpace(env, SIMPLE_MOVEMENT)
env.reset()
env.render()

for step in range(1000):
    if done:
        env.reset()

    state, reward, done, info = env.step(1) # the RIGHT action
    env.render()

# wait two seconds before closing the window
time.sleep(2)
env.close()
```

Code 3: Game play example using gym-super-mario-bros library.

3.4.3 Legal use

OpenAI's Gym is licensed under the MIT license[17]. Permission is hereby granted, free of charge, to any person obtaining a copy of its software and associated documentation files to deal with it without limitation to the rights to use, copy, modify, merge, publish, distribute, sublicense, and/or sell copies of it. So, for our educational purpose, we can use it without any restrictions and the only condition to use their library is that in the license file we will include their copyright notice[18].

Repository gym-super-mario-bros is based on the Gym's code, so its license contains the same copyright notice which we mentioned above, and we will also have to include it. Apart from that, it restricts the use of this library and code only for educational purposes. Since that is our case, even here we won't have any problems with using the library.

[17]More details about the license can be read at `https://opensource.org/licenses/MIT`.
[18]`https://github.com/openai/gym/blob/master/LICENSE.md`.

Chapter 4

Architecture of selected models

This chapter introduces the baseline architectures of the selected approaches along with their modified, more advanced versions. By baseline architecture we mean the architecture which was introduced along with the first mention and description of the given approach. We then describe how we have implemented them and also various modifications and techniques of them. These techniques are based on our research in Chapter 2.

The first part of this chapter is dedicated to the implementation and variations of DQN (see Section 2.2 for details about the technique), and the second part is devoted to A3C implementation (see Section 2.4) and TD3, which is based on its principles.

It's good to note that in this chapter the focus is on the description of the architectures and implementation details of the selected models. In Chapter 5, we are then further focusing on the comparison of these models performances, and describe our own experiments with fine-tuning of the selected model and their effects on the agent's performances along with the modifications which led to the best results.

4.1 Implementation of pre-processing steps

For every implemented agent (model), we needed to prepare the input data in a suitable way. This meant applying the pre-processing steps which we have described in its specific Section 2.2.1.

We have fully utilized the use of the Gym's wrapper functionality (read more about it in Section 3.4.2) and decided to prepare a separate class for every pre-processing step and modification, which we wanted to apply to the environment observation state provided by the gym-super-mario-bros repository (described in Section 3.4.2). Implementing each individual step as a separate class also helped us to divide individual logical blocks and also made it possible, if needed, to apply just some of them and, therefore, always have the option to modify the input data in flexible ways.

When it comes to the form of the input state representation, the so-called observation, it was originally provided as a two-dimensional array of raw pixel values. In order to make the modifications easier, we have decided first to create a class that converts every two-dimensional observation array into PyTorch tensor[19]. This way, we were able to use methods from the torch transforms[20] module which made it very easy to manipulate with the image itself. Using the transforms module, we have then implemented pre-processing steps, which were modifying the form of the observation image data. The implemented classes, therefore, work with the tensors. They are `TransformToGrayEnv` for converting the observations into grayscale, `RescaleTo84x84Env` for resizing of the image into provided shape, or by default into the 84 by 84 shape and the `NormalizePixelsInEnvObservations` which normalizes the pixel values into $(0, 1)$ interval.

We also needed to implement two techniques which are both based on skipping frames and enabling fewer computations required from the neural network. The first one modifies the step function of the environment, and its implementation can be found in the `SkipFramesInEnv` class. It changes the behavior of the step function so that when the environment's character (Mario) receives an action to perform, it performs the action in a loop given number of times (this number is by default set to four based on the original idea proposed by Mnih in [16]) and collects the sum of rewards from each action performed. It returns the observation state where the agent ends together with the computed sum of rewards.

After applying all the modifications mentioned above, we had observation of shape 84 by 84. To avoid processing copies of the same frame multiple times, we have used the built-in `LazyFrames` class from the OpenAI's Gym library. It stacks m consecutive frames (m is by default set to $m = 4$) on top of each other, resolving in data of size 4 by 84 by 84. `LazyFrames` works with frames after all the pre-processing modifications have been applied to them. Inconveniently, it outputs the data in the LazyFrames object format, which is not ideal. After some reading through its documentation, we found out that the class exposes `__array__`, which means that the object can be converted into a NumPy array[21] without any problem. So, at the end of our pre-processing, the LazyFrames object is always converted into a NumPy array. This means that even though we are modifying each observation of the environment, the outputs are still in the same data type as they originally

[19]Tensor is a multi-dimensional matrix containing elements of a single data type.

[20]Transforms module contains functions for performing common image transformations. Their list and details can be found at `https://pytorch.org/vision/stable/transforms.html`.

[21]NumPy array is a grid of values, all of the same type. More at `https://numpy.org/doc/stable/reference/generated/numpy.array.html`.

were. Then, later in our code, whenever the data is fed into the neural network model, we just simply convert it again into PyTorch tensor.

Since we ended up defining a lot of classes, we have decided to put them all into one python file, which we named `preprocessing_methods.py`. For each model, we can then just import the pre-processing methods we want to apply and use them rather than copying the same code over and over again. We have also implemented function `create_mario_env`, which takes the environment instance as an argument and optionally also the action set, which is the game environment supposed to be loaded with. The function applies all the implemented pre-processing steps and returns the modified environment. This makes the code easier to read if we want to use all the methods because we can just import and call one function and don't need to worry about anything else.

4.2 Q-learning based models

For the Q-learning algorithm, we have decided to implement and observe three versions of it. In this section are described the specifics of these implementations and how they differ.

First is the description of a plain DQN implementation, which will serve as a baseline that will allow us to observe the effects of the modifications on the agent's performance. Then we will describe a model for which we have applied all the additional pre-processing steps and advanced techniques which we found during our research and that are all described in the Section 2.2, specifically in its subsections. At last, we will describe the Double DQN model and how the implementation differs from the previous models that don't deal with the overestimation problem.

4.2.1 DQN

The idea of introducing neural networks and, therefore, creating the Deep Q-learning model was proposed by Volodymyr Mnih in [16], and the main reason was to create a new way how to approach the reinforcement learning tasks. The concept of Q-learning was already known, but it didn't scale well enough with most of the RL tasks, where the environment was often huge with a big set of available actions, which caused the original Q-learning (tabular approach) to consume huge amounts of memory while also relying on enormous computational power when computing the Q value from such a huge table. And that problem was solved by the introduction of neural networks, which enable much faster computations and better scalability. Mnih named his model Deep Q-network, DQN in short. Since this was the first mention of the deep learning approach itself, we have decided to use it as an initial model for our research. In the following paragraphs, we describe its details that have been implemented.

Mnih studied the model's performance on Atari games and he found it useful to apply some common pre-processing steps to each game environment. We have talked about these common techniques in Section 2.2.1 and about how we implemented them in Section 4.1. Mnih in his work converted the environment states into grayscale, resized them into 84 by 84 and also applied both frame skipping mechanisms. This means that he used almost all the same pre-processing steps, which are commonly used for this type of RL task and were described in Section 2.2.1. The only pre-processing step he did not apply was the normalization of the pixel values after grayscale conversion and resizing, so we have also not used it when creating this version of the DQN model.

When it comes to the neural network's architecture, Mnih used a network that expects the input of shape 84 by 84 by 4 which is also the shape of our data after applying the pre-processing steps. The neural network then consists of three convolutional layers and two fully connected layers. The first hidden layer convolves 32 filters with a kernel of size 8 by 8 with a stride equal to 4 with the input image and then applies a rectifier function[22] (ReLu). The first layer is followed by the second hidden layer, which convolves 64 filters with a kernel of size 4 by 4 with stride 2, which then also applies the ReLu function. The third hidden convolutional layer convolves 64 filters of 3 by 3 kernels with stride equal to 1, followed again by a ReLu function. Then is the final hidden layer, which is a fully connected layer type and consists of 512 rectifier units. The output layer is then a fully-connected linear layer with a single output for each valid action available in the selected action set. Simple visualization of the network architecture, which we have just described and Mnih used in his first DQN introduction, is shown in Figure 4.1. Notice that the only place where our implementation of the network differs is in the output shape, where his architecture predicted Q-values for 18 actions, but in our environment where we have fewer actions available, our architecture outputs 5, 7, or 12 Q-values, depending on the action set used.

Once the network has been implemented, the agent who will be playing the game needed to be implemented as well. When initializing the instance of the agent, he loads and prepares all necessary internal variables. The variables we provide him with include the learning parameters which consist of the starting exploration rate, the exploration rate decay, the minimal value of exploration rate, the discount factor, and the learning rate.

The exploration rate related parameters are used when the agent is deciding whether to perform a greedy action or to choose an action based on the prediction of the network. After each update of the network's weights, the exploration value is set to be equal to either the minimal exploration rate or

[22]Rectifier function, also known as Rectified Linear Unit (ReLu), is an activation function commonly used inside neurons in the neural networks. Its formula and details can be found in `https://machinelearningmastery.com/rectified-linear-activation-function-for-deep-learning-neural-networks/`.

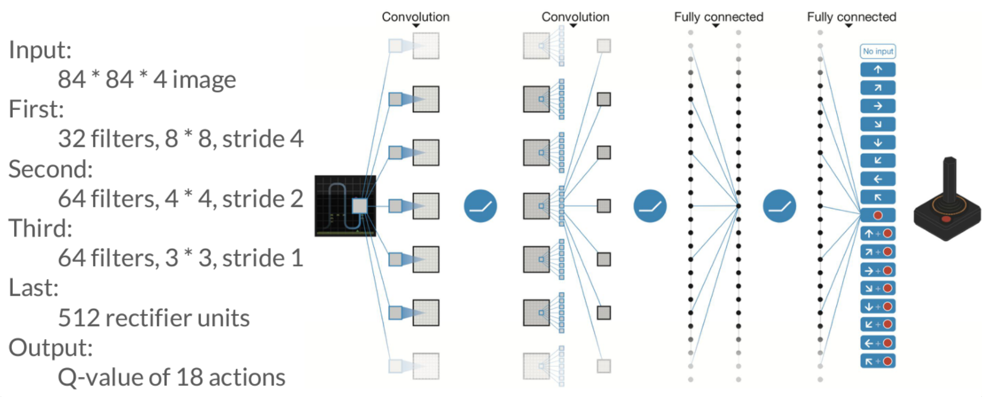


Figure 4.1: Visualization of DQN's network architecture originally proposed in [16] by Volodymyr Mnih. In our implementation, the number of actions differs so we modified the output shape. Source: `https://courses.engr.illinois.edu/cs546/sp2018/Slides/Apr05_Minh.pdf`.

the current exploration value multiplied by exploration decay, whichever is greater.

The discount factor is often denoted as γ in the mathematical formulas which we introduced in Chapter 2, and it is used during the update of the deep neural network's weight. We have set the discount factor to $\gamma = 0.99$.

When the agent initializes its variables it also initializes its neural network. The network has the form which is described above. Our implementation uses the Adam optimizer[23] with the provided learning rate. The network uses Huber loss[24] (called SmoothL1Loss loss in the torch environment) as the loss function when performing the backward propagation. We have set the value of the learning rate to the value of $\alpha = 0.00025$, which was also used by Mnih in his research. We have set the values of learning rate and discount factor the same for all our modifications so we can observe the sole effect of the applied techniques when we compare this baseline model to its modifications.

The core functionalities which the agent needs to have are commonly called (in similar RL tasks) *act* and *update*. Act method takes a pre-processed observation state of the environment as an argument. The agent then generates a random number from the interval $[0, 1)$ and if the number is lower than the current exploration rate value, then he performs a greedy (random) action. Otherwise, he provides the neural network with the observation state and performs the action for which the network predicted the highest Q-value.

After performing the action returned from the act method, the update method is called. It takes the previous observation state and the returned

[23]Optimizer based on the Adam algorithm. More at `https://pytorch.org/docs/stable/generated/torch.optim.Adam.html`.

[24]Huber loss is a loss function that is not so sensitive to outliers in data. It is a combination of the mean squared error and the absolute value loss functions.

values after performing the action. These include the new state, done flag (signalizing if the game ended), and reward value. Based on these arguments, the agent computes current and target values which are then provided to the loss function. Then the gradients are computed and the weights are optimized using the selected optimizer. The update process, along with the formulas, is explained in detail in Chapter 2.

Apart from these two methods we also implemented additional methods that allow the agent to save computed values into variables and methods for saving them permanently into computer memory if needed.

The implementation of the DQN model we have just described can be found in the DQN folder with all its relevant files, where the source code can be found in the `DQN.ipynb` Jupyter notebook[25].

4.2.2 Enhanced DQN

Our second model we decided to name Enhanced DQN because, in core, it is similar to the implementation of the original DQN model described in the previous Section 4.2.1, but it is enhanced by the advanced techniques which we discovered during our research and described in Chapter 2.

The original DQN doesn't use normalization of the pixel values in its pre-processing steps so we have included the pixel normalization for this model. This allowed us to use function `create_mario_env`, which we implemented, and it conveniently applies all the pre-processing steps at once.

After pre-processing was solved, we needed to implement two other things. The two network technique, which introduces another deep neural network and the experience replay mechanism.

The two network mechanisms requires the agent to hold two separate network instances called local and target networks which both have the same architecture. In code, the agent uses the local network within his act method for predicting the Q-values of the actions to use for a given state, but the most crucial place in the two network principle's implementation is during the update of the network's weights. Whenever weights of the network are updated, we use the target network for computing the target value for the loss function, and we use the local network for computing the current value. After the loss function processes the computed target and local values, the weights of the local network are updated. We keep a step counter, and with each update of the local network weights, we increment it by one. Every one thousand steps, we update the target network's weights with the current values of the weights of the local network. The number of steps after which we update the target network's weights is another training parameter of our agent. We have decided to use the value of one thousand because it provides enough steps for

[25]Web-based interactive computational environment, which is often used in ML for creating python-based runnable source code notebooks.

the technique to take effect. It is also not so high, so we can see the impact even with the limited computational resources which we had at our disposal.

The experience replay mechanism introduces a buffer of experiences for the agent to work with. Experience is the outcome after an agent performs an action. We refer to the buffer size as batch size, and it's commonly set to the value of 32 for similar tasks. Each action the agent decides to perform is put into the buffer, and the agent does not update the network's weights until the buffer is full. The buffer works in FIFO[26] mode, so after the buffer is full, whenever a new experience is added to the buffer, the oldest experience present in the buffer is forgotten. After the agent performs enough actions to fill the buffer, he will start performing the weights updates. For every update, the agent uses bootstrapping statistical technique[27] to get a batch of experiences from the buffer. The agent uses this whole batch of experiences to update the local network's weights. To better illustrate the mechanism described above, we have put the core lines of the agent's experience replay mechanism implementation into Code 4.

The implementation of the agent is in `EnhancedDQN.ipynb` Jupyter notebook, which is situated in the EnhancedDQN folder where also all computed data files from the model's training are present. We have also decided to implement it in a way where you can easily choose whether you want to use the two networks enhancement or not by simply providing a Boolean value when initializing the code execution.

4.2.3 Double DQN

The double DQN model is a slight modification of the original DQN model, which is able to deal with the overestimation problem, which the original model often suffers from. We have described this in detail in Section 2.4.2.

When it comes to its implementation, we have decided to use the same code as for the Enhanced DQN version which is described above. The only difference is in the agent's update method when computing the loss value and, based on it, the gradients for updating the weights of the network. The loss function (which computes the loss value) takes two values. The current value and the target value. The target value is based on the Bellman's equation (see Section 2.1.2), and in our previously implemented model, the Enhanced DQN, this formula uses the target network when determining the Q-value of the next action (as can be seen in Code 4), but in order to apply the Double DQN principle in our implementation, we have switched usage of the target network and instead used the local network for computing the target value for the loss function.

[26]FIFO stands for First In, First Out inventories of data.

[27]Bootstrapping chooses random samples from the collection. This means that the samples can be selected more than once.

```python
def experience_replay(self):
    # copy local net's weights every 1000 steps
    if self.step % 1000 == 0:
        self.copy_model()

    if self._batch_size > self.experience_cnt_in_memory:
        return

    # get random batch of experiences
    ACTION,REWARD,STATE,NEXT_STATE,DONE = batch_experiences()

    self.optimizer.zero_grad()
    # apply the Q-update based on the whole batch
    target = REWARD + torch.mul(
        (gamma *
            target_net(NEXT_STATE).max(1).values.unsqueeze(1)),
        1 - DONE)

    # local net approximation of the Q-value
    current = self.local_net(STATE).gather(1, ACTION.long())

    loss = self.l1(current, target)
    loss.backward() # compute gradients
    self.optimizer.step() # backpropagate the error

    # update exploration rate after weight update
    self.exploration_rate *= self.exploration_decay
    self.exploration_rate = max(self.exploration_rate,
                                self.exploration_min)
```

Code 4: Simplified code example of the experience replay mechanism implementation.

Again, the implementation of the model along with the parameters of the trained network can be found in its dedicated folder called DDQN and the model implementation in `DDQN.ipynb` Jupyter notebook.

4.3 Actor-Critic based models

In our later sections of Chapter 2 we talked about two models based on the actor-critic approach. In this section are described the specifics of A3C and

TD3 (which are both based on the actor-critic approach) implementations and how they differ.

Generally said, the main difference between A3C and DQN implementation and their runtimes is that A3C is created for running multiple workers in parallel. Each worker runs on a separate thread, allowing simultaneous computations and potentially a bigger chance of learning the correct behavior for the agent while keeping a similar execution time as the DQN. This also indicates that the code of A3C implementation differs significantly when compared to the DQN implementations. In contrast to the multi-threaded A3C model, we will also describe the TD3 approach (see Section 2.4.2), which was introduced as one thread algorithm but consists of multiple neural networks trained in each step, resulting in enormously bigger computational requirements when compared to A3C and DQN.

4.3.1 A3C

The first mention of A3C implementation was by Volodymyr Mnih in [15]. We used his work as our reference point and decided to create the same neural network architecture and modify it to be suitable for our task and the shape of the pre-processed data.

The main idea behind A3C was the introduction of the actor-critic concept, which aims to improve the learning phase. Its principles are described in Section 2.4, but in short, we can say that it introduces another neural network called critic with the same architecture as the actor network (network for next action prediction). Critic serves as a judge and mentor for the actor network, which is trained with each step, similar to how the neural network is trained in the DQN implementation. The critic network is trained alongside the actor during each of the actor network's update.

The actor-critic principle was already introduced along with the Advantage Actor-Critic approach (A2C), but A3C introduced another important thing which is the idea of workers. Each worker consists of an independent copy of the same neural network architectures of actor and critic networks. Since each worker trains independently in its own environment instance, it allows them to perform their computations in parallel. Workers then share their trained network weights with each other by updating the global shared model.

As mentioned above, both actor and critic networks share the same neural network architecture. We had decided to use the same architecture which was used when A3C was introduced for the first time in [15]. The network was only modified so it accepts the input of shape 84 by 84 by 4 (shape of our observation states after pre-processing). The network then consists of 4 hidden convolutional layers, where each convolves 32 filters with a kernel of size 3 by 3, with stride set to the value of 2 and padding set to one. After each convolutional layer, a rectifier function (ReLu) is applied. The convolutional layers are followed by a hidden Long Short-Term Memory (LSTM) recurrent layer with a hidden state of size 512. This hidden state is then processed by a fully connected linear layer that consists of 512 rectifier units. It is important to note that actor and critic share the convolutional and LSTM layers, but each has its own linear layer on top of them, thus training their own weights in the last layer. The actor network's output shape is based on the number of actions available to the agent, while the critic always outputs only one value which is used for computing the actor's loss value (criticizing the actor).

Since A3C is an asynchronous algorithm, a function that performs the behavior of one individual worker was created after defining the neural networks architectures. Because of the limitations of python multiprocessing in Jupyter notebooks[28] we had to put the worker's code into a separate python script file which we called `trainAgent.py`. For each worker, one thread is assigned to execute the code inside the mentioned script. Apart from the training worker process, we have also implemented code, which loads the shared model's weights, and plays the game using them while rendering the image state to the user, but this piece of code is not relevant during the training phase. It was used just for testing.

Each worker creates its own instance of the game environment and its local copy of the actor and critic neural networks and starts its independent training epochs. In each epoch, each worker plays one run of the game. It first loads the weights of the shared model into its local model and then performs individual steps based on the outputs of the actor's network. The worker collects actor and critics networks outputs and the rewards he gained during the whole epoch, and at its end, he computes the loss functions value considering all these gathered data and, based on it, updates the weights of the global shared model.

There was also a need for a script that manages that all asynchronously running workers are intact, creates and holds the shared global networks, and also sets up all necessary training parameters for each worker. Its implementation can be found in Jupyter notebook `A3C.ipynb` in the A3C folder along with the final trained networks parameters. The collection of all the data from our training process was assigned to only one of the workers to keep consistency in the data and prevent individual workers from overwriting data to each other. We were able to assign the saving process to only one worker also thanks to the fact that each worker updates the global model after each epoch.

When it comes to the training parameters, we had used the same values which were used when the model was proposed. Apart from the parameters for the number of epochs and the maximal number of steps per epoch, we had to set up the learning rate (which we set to 0.0001), γ, τ parameters, and also the entropy coefficient. Parameters γ and τ are used to compute the Generalized Advantage Estimator (GAE), which is used when computing the

[28]See `https://stackoverflow.com/q/47313732/9675818` for details.

loss value of the actor network, and we have set them to $\gamma = 0.9$ and $\tau = 1$. The entropy coefficient greatly limits the effect of the accumulated entropy loss, which is computed in each step of a given epoch, as it is multiplied by it when we use it in the computation of the overall loss of the architecture for its update. We have set the entropy coefficient value to 0.01.

4.3.2 TD3

Twin Delayed Deep Deterministic policy gradient algorithm (TD3) takes advantage of the actor-critic principle, which was introduced with the A3C approach, and solves the overestimation issue which A3C suffers from in most tasks. Even though it shares the same core principle, TD3 is not designed for direct parallel implementation (at least in its original version). This allowed us to follow similar design patterns with which the DQN-based algorithms (see Section 4.2) were implemented, so the approach and style of the implementations are quite similar and differ only in the neural network architecture and implementation of the agent's update method logic. All other pieces of code (initialization of the environment, saving and loading of trained models or other collected data) are the same, or with just slight necessary modifications.

When it comes to neural networks, the TD3 agent keeps and trains multiple instances simultaneously. He holds actor network, actor target network, critic network, and critic target network. Actors network consists of three fully connected linear layers, the first two consisting of 256 rectifier units, and the last layer's neurons use the hyperbolic tangent as its activation function. We are aware that using only these layers may not be the most suitable for our problem, but our goal was to compare the agent's performance using the originally proposed implementations of the selected approaches, and that is why we decided to stick with only linear layers as that is how Fujimoto prepared the model when he first introduced it in [7]. The critic network uses the same architecture, but it is more complicated. A critic network is made of two individual neural networks, which each represent one critic (see Section 2.4.2, where we describe why TD3 uses two critics instead of one), and each of these critics has the same network architecture which we described above (three fully connected linear layers network). The only difference is that the last linear layer's neurons don't have any activation function. The critic network then provides an option to process the input by only one critic network, or by both separately and return results from both networks. The critic network also differs from the actor network in the input shape as it expects concatenated observation state with the probabilities for the action to select (the output from the actor), while the actor network only excepts the observation state on the input. Critic's implementation is shown in Code 5.

When it comes to the act method of the agent, there are only slight modifications in TD3 when compared to the DQN implementation, as it mostly consists of feeding the actor network with the current state of the agent, re-

```python
class Critic(nn.Module):
    def __init__(self, input_shape, action_cnt):
        super().__init__()
        # critic 1 architecture
        self.lin1 = nn.Sequential(
            nn.Linear(input_shape[1] + action_cnt, 256),
            nn.ReLU(),
            nn.Linear(256, 256),
            nn.ReLU(),
            nn.Linear(256, 1)
        )

        # critic 2 architecture
        self.lin2 = nn.Sequential(
            nn.Linear(input_shape[1] + action_cnt, 256),
            nn.ReLU(),
            nn.Linear(256, 256),
            nn.ReLU(),
            nn.Linear(256, 1)
        )

    def forward(self, next_state, next_action):
        next_state_action = torch.cat([next_state,
            next_action], 2)
        out_1 = self.lin1(next_state_action)
        out_2 = self.lin2(next_state_action)
        return out_1, out_2

    def forward_first_critic(self, next_state, next_action):
        next_state_action = torch.cat([next_state,
            next_action], 2)
        return self.lin1(next_state_action)
```

Code 5: Implementation of the Critic neural network in TD3.

turning an array with Q-values for each action followed by the agent's choice of the action with the biggest value. Apart from the network computation, a noise sampled from normal distribution $\mathcal{N} \sim (0, \sigma)$ is added to the returned values, where the value of noise variance is set to $\sigma = 0.1$. Then, the returned values are clipped into the selected range. In our implementation, it was based on the action state space maximal values.

The update method required the most significant focus, as TD3 consists of a lot of mechanisms, all happening right after the agent performs an action. TD3 works with the experience replay technique (see Section 2.2.3), so during each update, a batch of experiences is sampled. This batch is then processed by the actor target network. The predicted next actions for the sampled next states are then processed by both critics of the critic target network, and the lower value from them is selected and used for computation of the target Q-value, where the γ discount factor parameter is used (we have set it to 0.9 as we did for DQN models). The target Q-value is then used to compute the loss value for the critic network.

After every defined number of iterations (usually set to a low number, we have set it to two), the actor network weights are updated based on the output of only the first critic from the critic network. After updating the actor, the target network's weights are synced with the local networks, where the weights of the local networks are combined with the old weight values of the target networks using the τ regularization parameter. In our implementation, τ is set to $\tau = 0.005$, and for a better understanding of its usage, we have attached Code 6, where this update is performed.

```
for param, target_param in zip(
    self.critic.parameters(),
    self.critic_target_net.parameters()):
        target_param.data.copy_(
            self.tau * param.data
                + (1 - self.tau) * target_param.data)

for param, target_param in zip(
    self.actor.parameters(),
    self.actor_target_net.parameters()):
    target_param.data.copy_(
        self.tau * param.data
        + (1 - self.tau) * target_param.data)
```

Code 6: TD3 - Update of the weights of target networks using the τ parameter to combine weights of the local network and old target network.

Implementation of TD3 agent is located in the TD3 folder in Jupyter Notebook sharing the same name.

Chapter 5

Experiments

After successful implementation of all models which we thought could be efficient for training an intelligent agent for the Super Mario Bros. game, we have performed a series of tests in which selected metrics were collected with the aim to compare the models and decide which is the most suitable for our task. Implementation details of these models are described in Chapter 4.

In this chapter the observed details of the training process that were applied to all the implemented models are described, and then their individual performances are compared and discussed. We also mention difficulties that were found during testing, and we also note where is room for improvement. In the later sections, we describe the experiments and possible modifications of the tested approaches and discuss whether they have improved the model or not. We then trained the agent with the fine-tuned final model and tested his performance on the level that it was trained on and on different levels he had not seen before.

5.1 Training conditions

This section focuses on describing the details of testing conditions which were set for the implemented models. It also describes the metrics which were selected to collect during training and later used for comparing the performance of the selected models.

First, we needed to determine the number of epochs for which we would train all our models. Since our resources were limited, we needed to set the number of training epochs high enough so the learning techniques and modifications could take effect while also having reasonable execution times because we needed to be able to execute many tests and later try different custom modifications. With this in mind, we have at first tried to train our models for twenty thousand epochs, but even the simplest model (DQN) took us more than 24 hours to compute this amount of epochs. Based on this we decided that this amount is too much and decided to determine the number of epochs based on the model, which requires the most computational power. Based on the number of neural networks, the most complex model seemed to be the TD3 model, which updates multiple (up to four) networks in each iteration. Even though we have implemented and executed TD3 on fast GPU[29], its computations were so slow that we were forced to limit the number of training epochs to only 2 thousand. We are aware that this number is low for such a complex task and that these models can be really demanding on the number of epochs needed for the agent to perform human-like, but even with the most powerful resources, we managed to get our hands on, training two thousand epochs of the TD3 model took us around twelve hours.

TD3 was not the only extremely slow model, as A3C proved to be the slowest model to train. Even though that the A3C model is implemented to run on multiple threads, we were running into long execution times because it was the only model which we were not able to implement and execute to run on GPU. This led to two thousand epochs running for more than 24 hours for the A3C model.

Training parameters of each model and their values are described in their respective sections in Chapter 4. Their values were used the same as they were set when they were introduced by their respective authors. Some models, which are based on the same approach (for example, the three implemented DQN-based models) use the same values of the parameters. We have decided to keep these values because our goal was first to observe the performance of these approaches on the game itself using the parameters their creators found to be the most useful. Since none of the authors tested their model on the Super Mario Bros. game, we created equal testing conditions for the models. Thanks to this, we were able to get an objective opinion on which model suits this problem the most.

Since not all the models are based on the same principles, values that are computed during training may differ a lot, so we needed to choose metrics that are shared for all the models and, therefore, would be the most suitable for comparing them. As all the models use and work with the same environment, the most suitable metric to use proved to be the reward that is returned after performing any action (also referred to as performing a step). As there can be a various number of steps in each epoch (agent can die or get stuck on the first obstacle), we have collected the total sum of rewards for each epoch and observed this number itself along with the total sum of rewards divided by the number of steps in the given epoch. In graphs, we refer to the later mentioned metric as an average reward per epoch. We have also bound every collected data point to the action set in which the training was done. This allowed us to compare model performances for each action set. We talk about the observed differences for each action set in Section 5.2.

[29]GPU stands for the graphics processing unit.

We have also computed the average selected action's value in each epoch by collecting the values from the network that predicted the actions (from which the agent always chooses which action to perform), but this metric is not comparable for all the implemented models. The same problem occurred when we tried to collect the computed loss function's value in each update step, where the problem was that some models (for example TD3) use multiple neural networks so it was not clear which one should be chosen for the comparison. Also, most of the networks have different architectures and don't even use the same loss function, so comparing all models with this value proved to be pointless.

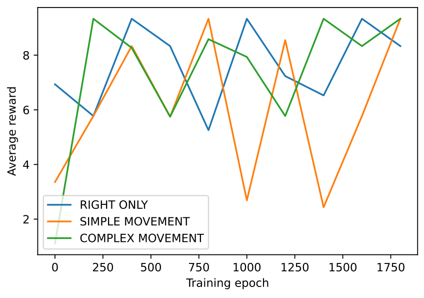
5.2 Comparison of Action sets

In this section, we discuss the observed effects of using different action sets on agents' performance during the training.

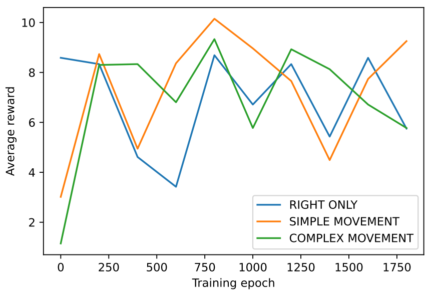
In the Super Mario Bros. environment, there are three available actions sets. RIGHT_ONLY with 5 available actions, SIMPLE_MOVEMENT with 7 actions and COMPLEX_MOVEMENT with 12 actions[30]. For the trained agent, the higher number of actions represents higher complexity. Therefore, one would naturally expect it to lead to higher computation times, but from our observations, we noticed that the execution times of their individual training differ insignificantly. For example, for the DQN-based models, the training time of the action set with the most actions was only 6 minutes longer than the execution time of the action set with the lowest number of actions.

In Figure 5.1 we show the measured average reward value per epoch during the training phase for each model separately. For each model, the graph shows the performance of the agent for each action set. We can see that in most models (DQN, Enhanced DQN, and A3C), the collected values are similar and it looks like the agent overall performs the same independently on the used action set, but from the measured values of TD3 and Double DQN models, we can see some differences. Particularly for the Double DQN model we can see that in later epochs, the RIGHT_ONLY and COMPLEX_MOVEMENT average reward values decrease with each epoch. In contrast, the agent who used the SIMPLE_MOVEMENT action set actually started to improve his average gained reward. This observation led us to use the SIMPLE_MOVEMENT action set for our model comparison and also to use it in our later experiments, even though we agree that the observed difference was small and therefore using any of the mentioned action sets should produce comparable results.

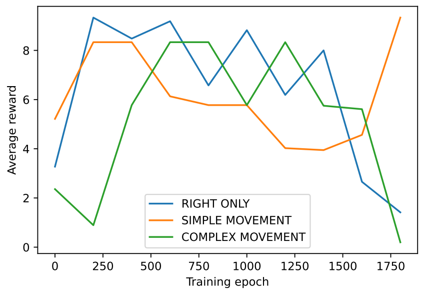
[30]Their definitions can be found at `https://github.com/Kautenja/gym-super-mario-bros/blob/master/gym_super_mario_bros/actions.py`



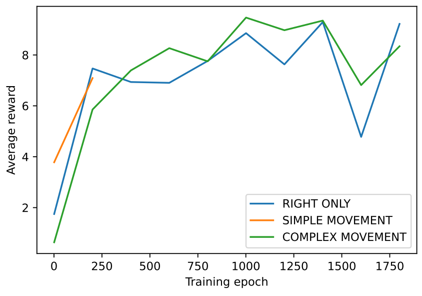
(a) DQN action set comparison.



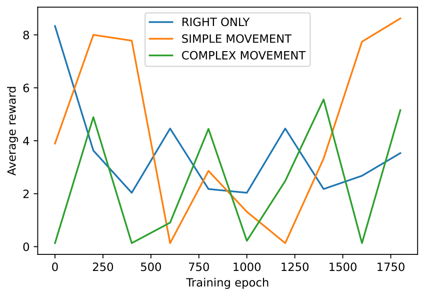
(b) Enhanced DQN action set comparison.



(c) Double DQN action set comparison.



(d) A3C action set comparison.



(e) TD3 action set comparison.

Figure 5.1: Comparison of agents performance for different action sets. The comparison is done separately for each model in its respective subfigure.

5.3 Model comparison

After determining which action set to use, we have trained the agent for each of the implemented models, and in this section we compare their performances during the training.

The measured data can be found in Figures 5.2 and 5.3, where we show the average and total reward the agent obtained in each training epoch. Each figure contains data for all the models to make the comparison easier. When we look at Figure 5.2, we can see that the values for DQN-based models are similar, with simple DQN having the biggest peeks in consecutive observed values and the Enhanced DQN model performing slightly better than the other two DQN-based models.

From all five models, it seems that the TD3 model has the worst performances as the graph indicates that the agent gains on average a very small reward value for his steps when compared to other methods, and even though with later epochs TD3 shows better values which were comparable to the other models we can say that its performances are the poorest from the observed models. In contrast to TD3, the A3C-based agent shows performance comparable to the DQN-based models, and it even has much less peaks in its values and the value seems to be constantly rising.

In the second comparison graph (Figure 5.3), we can observe huge differences in values between epochs. The TD3-based agent again scored the worst values in this metric, while A3C and Enhanced DQN-based agents again were able to perform slightly better than agents based on other models even though we have to admit that obtained data points are messy and based on only Figure 5.3 we would not be able to determine any outcomes.

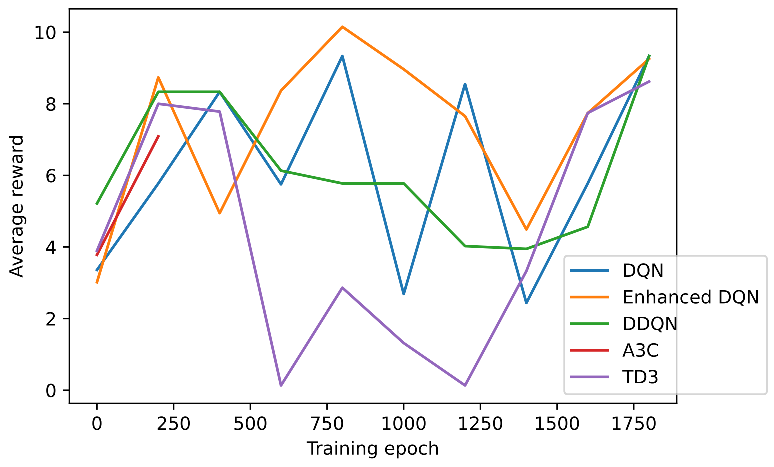


Figure 5.2: Comparison of implemented models average reward gained in each training epoch.

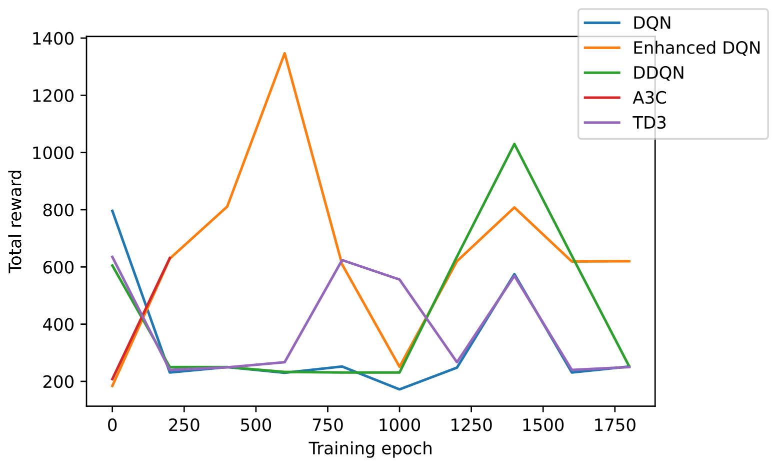
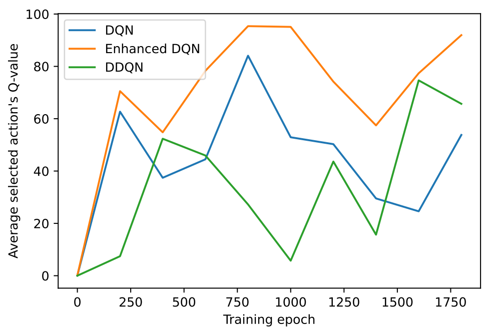


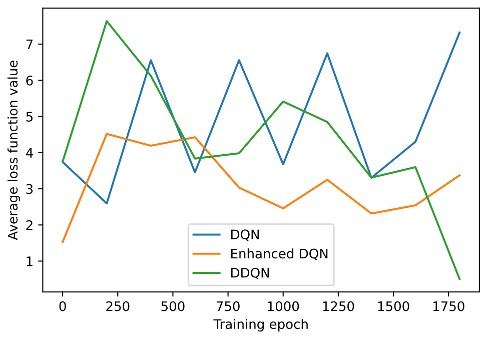
Figure 5.3: Comparison of implemented models total reward gained in each training epoch.

From the results discussed above, it looks like that from the DQN-based agents, the one which is based on the Enhanced DQN implementation performs slightly better. To confirm this, we decided to take a look at two other metrics. In Figure 5.4 we compare the DQN-based models. More precisely, we compare their average selected action's Q-value per epoch and average loss function value per epoch in its Subfigures 5.4a and 5.4b. The first Subfigure 5.4a comparing the average Q-value of the action selected by the agent clearly indicates that the Enhanced DQN model provides the agent with actions with overall higher Q-values. This does not necessarily mean that the favored actions are correct for the current state of the agent, but it indicates that the model is more sure about the action, and since all the models gained similar rewards during the training phase (see Figure 5.2) with Enhanced DQN obtaining slighter higher values, this confirms the assumption that Enhanced DQN is the best performing model from the observed DQN-based models. In the second Sub-figure 5.4b, the values are more noisy, but even here we can observe that the Enhanced DQN performs better than other displayed models as it has the lowest computed loss value over most of the observed training epochs.

Apart from the shown graphs, it is important to note that only the A3C-based agent was able to complete the game successfully during the training phase. No other agent was able to reach the winning flag at the end of the level during the training, and that is mainly due to the low number of training epochs. Even though the low number of epochs is very inconvenient and it may have caused that some models did not fully train and have not obtained the results they normally would with a larger epoch count, as there was nothing we could do because we just did not have the sufficient computational power



(a) Average selected action's Q-value per training epoch.



(b) Average loss function value per training epoch.

Figure 5.4: Comparison of metrics specific for DQN-based models.

at our disposal, we have decided to consider the data which we were able to collect as sufficient enough to compare the models.

A3C is the only model which is designed to train multiple agents simultaneously. This led us to perform a test that determines what the adequate number of threads for the Super Mario Bros. game is. Our aim was to choose the minimal number of threads needed, which would preserve the best performance of the trained agent. We have trained the model with two, four, six, and ten threads and then compared the agent's performances during each training. Our conclusion from the experiment was that the optimal number of threads is six because all the workers (represented by individual threads) were able to successfully complete the level within the observed two thousand steps at least once. Four of them were even able to complete the level for the first time in the first six hundred epochs, which indicates very efficient utilization of the multiple workers principle, but since each worker was able to clear the level no more than twice it is clear that the network still did not learn properly and it was due to some randomness during the specific runs. Nevertheless, the agent trained with six threads showed the best performance during the training.

In conclusion from the collected data points, we have determined that the TD3 model is the least efficient for our chosen game and based not only on the data but also on the fact that A3C was the only model able to successfully complete the given level of the game during the two thousand epochs, we have concluded that A3C and Enhanced DQN models are the most suitable for training an agent which is able to play the Super Mario Bros. game (from the models which we have tested). Sadly, A3C is also the model that takes the most time to train, as the two thousand epochs took us around 20 hours to execute.

Since one of our goals was to try fine-tuning one of the models, we have decided to experiment and create modifications for the Enhanced DQN model because, from our observations, it proved to be among the most suitable models for the game, and each training phase took the model about 2 hours to execute. This made it possible for us to perform multiple modifications, train the agent for each of them and then compare the collected metrics in a reasonable time.

After we had executed enough experiments and decided which modifications were most beneficial for the agent, we applied the modifications to the A3C model and then observed and discussed the effects of these changes on it. We discuss our experiments and their results, along with results from the test runs of final trained agents, in the following sections.

5.4 Model fine-tuning

After comparing the models we have decided to conclude a series of experiments with the aim to improve the selected model, in particular the performance of the agent who is based on it. Based on our previous observations we have decided to fine-tune the Enhanced DQN model, which performance and time to execute the training phase made it the ideal candidate for this part of our research. In this section, we will describe what modifications we have experimented with, what modification provided the best performance during the agent's training, and in the later paragraphs, we will discuss which of these modifications brought positive improvement to the agent's performance and which did not.

Our first experiments concluded of observing the effect of various modifications to the neural network used by the agent. The original network's implementation concludes of 3 hidden convolutional layers followed by a linear, fully connected hidden layer, so for most of our experimental phase we have focused on modifying and observing the effects of changing the size of these layers (modifying the neuron count of the layers). We have also observed the effects of removing up to two convolutional layers and also the effects of adding up to three additional convolutional layers. We have also experimented with different kernel sizes in the convolutional layers.

Apart from modifying the architecture of the neural networks, we have also focused on introducing more control over the randomness during the model's training process. In the original implementation, the agent either performs an action based on the neural network's prediction, or it performs a completely random action[31]. Our reason behind this idea was to limit the randomness just a little bit, so the agent still explores new actions in the first phases of

[31]Performing a random action is based on the randomly generated number which has to be lower than the current value of the exploration rate parameter which decays with each taken step. This makes the agent explore the state space more in the initial phases of the training process. More details about the action selection of the agent can be found in Section 4.2.1

the training but also guarantees that the actions are not completely random. We did this by taking the Q-values predicted by the neural network and used these values to construct a categorical distribution[32] where the probability of an action being selected was based on the predicted Q-value and then we chose a random sample from this distribution. We have also tried adding noise to the predicted Q-values from the neural network from a normal distribution to little bit alter the predicted values and make the process of choosing the next action less deterministic in the training phase.

While we were experimenting and observing the effects of the randomness in the initial parts of the training, we have noticed that the exploration rate parameter value decays to its allowed minimal value after the agent performs just about a thousand training steps. This caused that majority of the training phase actually did not contain almost any randomness at all, which we thought could be harmful for the agent's learning. In order to prevent that, we have experimented with setting the exploration decay parameter to higher values or setting the minimal possible exploration rate value to higher values.

Another thing that we have experimented with was changing the pre-processing steps applied to each observed state. We have tried to skip different steps used to pre-process the state image. For example, we have tried to skip the conversion to gray-scale or to omit the resizing of the observation state image and leave it in its original size.

After observing the effects of all our separate modifications and combinations of them, we were able to conclude which modifications are beneficial for the agent's performance in the Super Mario Bros. game environment and which are not. Introducing randomness in any form we have tried and described above did not bring any improvement and it actually made agent performance slightly worse. Even modifying the value of the exploration rate related parameters did not seem to affect the agent's overall learning performance.

From experiments with modifications of the architecture of the neural network, we have observed that modifying the size of the linear layer does not have any significant impact, but modifying the convolutional layers has. Removing and adding convolutional layers or changing parameters of individual layers showed to affect the agent's performance. We have found out that the most useful modification, which we have also decided to use in our final version of the model, was the removal of the second hidden convolutional layer and changing the number of filters convolved in the first layer from 32 to 64 (for details about the initial neural network architecture see Chapter 4). For this modification, the agent has kept obtaining almost identical rewards during the training phase and it allowed faster execution time of the training phase

[32]Categorical distribution (also called Bernoulli distribution) is a probability distribution used for generating a pseudo-random variable. This variable can take value of one of the $\mathcal{K}$ possible values and the probability of choosing each value is specified for each category separately.

(and generally having fewer neurons to train while keeping the performance quality is always good). In some parts of the training, the agent based on the modified model gained even more reward points.

From our experiments with modifications to the pre-processing steps, we have found out that letting the neural network work with the original size of the image had the most positive effect from all the modifications which we have experimented with. This means that even though resizing and minimizing the dimensions of the game state image had a positive effect for Atari games on which the DQN model was originally fine-tuned, for our game this actually made the agent's performances worse. The difference between the fine-tuned model and the original Enhanced DQN implementation in the observed average reward value per epoch can be seen in Figure 5.5.

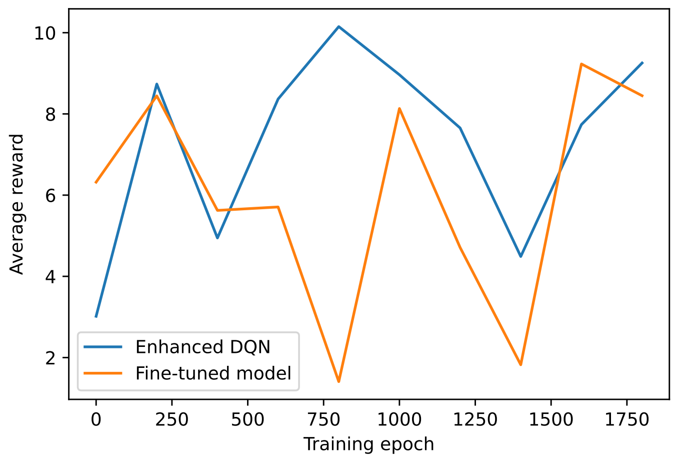


Figure 5.5: Comparison of the average reward gained per epoch of the Enhanced DQN model with our fine-tuned model.

Since all our previous comparisons and experiments were realized with a limited number of epochs due to the limited resources we had which led to the fact that almost all the models were not able to train an agent compatible of finishing the level, we have decided to perform one final and larger test of our fine-tuned model with a greater number of training epochs. We have trained an agent for ten thousand epochs, which is five times more than during our experiments. We have then observed his performances during our evaluation phase, which is described in Section 5.5.

Since we had much more training epochs, we were also able to set some training parameters to higher values. We have increased the number of steps after which the local network's weights are copied into the target network from one thousand to five thousand and we have also increased the experience replay memory buffer size to 500 steps. We have also lowered the minimal value of the exploration rate parameter in the second half of the training epochs from

2 percent to 1 percent. In the last 1500 epochs, we gradually lowered it up to 0.3 percent.

5.5 Evaluation of the trained agent

In previous sections was introduced our fine-tuned version of the Enhanced DQN model. We have decided to train an agent for a larger number of epochs and observe his performances while playing the Super Mario Bros. game. We have decided to perform this larger training and tests not only for our fine-tuned Enhanced DQN model, but we have also applied the same modifications to the A3C model and observed its change in performance. In this section is described the test phase setup which was used and our observations of the agent's behavior for each model separately.

To test the trained agent's performance, we have decided to display the game's visual output and observe the agent's behavior. Apart from observing his average behavior in the environment, we have also focused on the farthest x-coordinate which he was able to reach and how often he reached this farthest position. We have also tried observing how large game score was the agent able to gain during these runs, but it proved to be an ineffective metric when it comes to evaluating an agent's performance as it is more important to get as far as possible in the game than to collect more coins and kill enemies, which does not win you the game.

5.5.1 Evaluation of the Modified DQN model

We have trained the agent on our fine-tuned model for ten thousand epochs which took approximately 70 hours. This length was mostly due to the fact that the agent was continuously able to get farther in the level, which increased the time of each individual run (one successful run takes the agent approximately 25 seconds) and the total run-time in general. Even though that the number of epochs should still be much larger[33] we wanted to observe and describe the behavior of the trained agent, even if it won't be perfect.

The first interesting fact that we observed is that the agent was able to successfully complete the level many times during this bigger training. After three thousand epochs, the agent started successfully finishing the level approximately every 30 epochs, and this frequency slowly increased. In later epochs of the training, we have observed that the agent was able to successfully finish the game in almost every second run and occasionally even in a few games in a row. The longest consecutive successful run was five epochs long. As the agent has not started reaching the end in every epoch during the later stages of training, it indicates that even though his performance had become really good, he still needed to perform some random steps in order to finish

[33]For example, Mnih in [15] trained his model on over a hundred thousand epochs.

the level successfully. We again mention that the exploration rate was set to 0.3 percent chance in the later stages which indicates that the agent did not rely on the randomness, but he most likely needed for it to happen in order to finish the level. This observation from the training phase is a great sign, which indicates that the model actually improves itself with our modifications and the larger number of training epochs, as none of the DQN-based agents was able to reach the end during the two thousand epoch training phases.

As expected, the agent's behavior is deterministic and he always performs the action which the network evaluates as the most beneficial for his current state, which resulted in each run of the agent being the same. Agent's performance exceeded our expectations as the agent has learned to play the game so well that he was able to swiftly jump over obstacles while dodging or even killing some enemies and he was able to clear the whole level. It was interesting to observe some trained behavior of the agent. For example, when he is running and has a pit in front of him, he always performs a perfectly timed jump action, or when he is running and an enemy is going against him, he has learned to jump over him. These are the types of behavioral patterns that we aimed to achieve.

In conclusion, our fine-tuned model, after 10 000 epochs, was able to learn how to clear the whole level successfully and we were able to observe nice behavioral patterns. We could still see the agent getting stuck for a few seconds by some static obstacles before he decided to jump over them. This indicates that even 10 000 epochs are not enough for the model to fully train the agent to clear the level perfectly but we were still able to see that the model was able to utilize itself for the game. Therefore our applied modifications were suitable for the given task as we were able to successfully train the agent for the Super Mario Bros. level.

5.5.2 Evaluation of the Modified A3C model

In our first comparison of the implemented models during training in Section 5.3 we have picked A3C as the model which seems to be the most effective for the low number of epochs, but since its training time was by far the slowest we have decided to experiment and fine-tune the Enhanced DQN model whose much lower training times gave us the option to perform multiple various experiments. After introducing the fine-tuned version of the Enhanced DQN we have decided to try applying the same modifications to the A3C model and observe if they also improve its performance. In this section we describe the observed training performance differences for the modified model and also the observations from when we let the A3C-based agent play the game.

Application of the same modifications as we applied to the Enhanced DQN were easy to implement as the neural network uses similar architecture. Unfortunately, we were forced to train the modified A3C model for only two thousand epochs, the same number that we used in the initial tests, and that was due to the fact that one of the applied modifications was to let the model work with the full-sized image of the states and this increased the execution time drastically as it took almost four days to train the A3C-based agent.

The performance of the agent during the training turned out to be much worse for the modified model than it was for the original implementation. We have used six worker threads as it was the amount that initially brought us the best training performances as most of the agents were able to clear the level within the first six hundred epochs, and most of them were able to finish it even twice. Unfortunately, this was not the case for the modified A3C model as all the workers were able to complete the level only once during the two thousand epochs. When we compared the average gained reward per epoch (see Figure 5.6), we could see that the modified model performed worse than the original one as the measured values are overall smaller for the observed period and much worse in the later epochs.

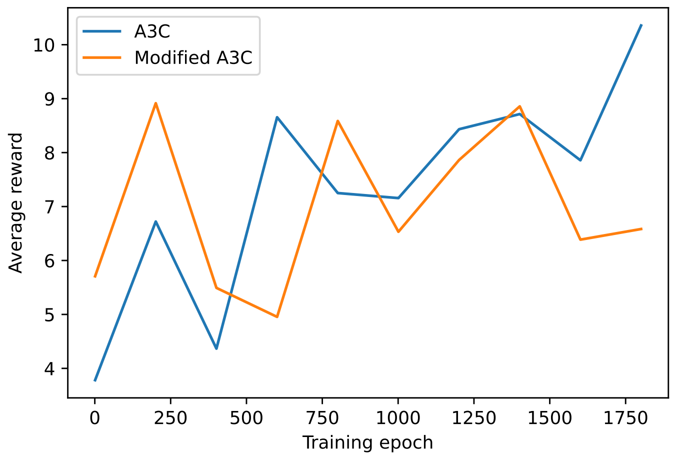


Figure 5.6: Comparison of the average reward gained per epoch of the A3C model with its modified version (inspired by our fine-tuned Enhanced DQN model).

We have then tried to observe the learned behavior of the agent in the game, but it turned out to be a huge disappointment as the agent did not show any sign of human-like behavior, and he basically only ran into the first enemy he encountered. So, in conclusion, even though the A3C model showed signs of being the most suitable model for the Super Mario Bros. game based on the observations from the initial training phase, the model was not able to learn the agent enough. We admit that the model could give better results if trained for more epochs (as indicated in results for the fine-tuned Enhanced DQN model trained for much more epochs), but we did not have proper resources for executing such tests. Also, it is clear that our modifications, which improved the performance of the Enhanced DQN, did not work at all for the A3C model as they made its training performance worse.

5.5.3 Agent's performance on unseen levels

In this section we describe the performances of our agent trained by the fine-tuned Enhanced DQN model on different levels for which he was not trained.

The Super Mario Bros. game offers eight worlds, each containing four levels. During our research we were training the agent on the first level of the first world. Our goal was to train the agent on one level and then test his reactions and behavior for levels that he had not seen before, but there was one catch, as some levels have different environment texture[34]. After observing the agent playing the second and third level in the first world, we have quickly realized that the different textures of the level confuse the neural network significantly. The agent's actions looked completely random and he was not able to overcome almost any obstacles. He has also shown no signs of learned reaction patterns which we described in Section 5.5. In one level with different textures the agent did not even perform any action. He was just standing on the starting position until the time ran out. Because we still wanted to measure the agent's performances we have selected two other levels from different worlds, whose environments have the same textures as the one we trained our agent on, and we have decided to observe his performances in them. The levels which we decided to use in the end were the first level of the second world and the first level of the fourth world. It's good to note that the world number does not have a significant effect on the difficulty of its levels, and it is only relevant to the story of the game, which is irrelevant for our agent, but of course, it still may contain new elements which the agent has never seen before like new type of enemy.

While observing the agents performances we have noticed that allowing the agent to perform random actions (even with very low probability) enabled him to follow the learned patterns from the network while sometimes performing a random action which resulted in better overall behavior of the agent in states which he has not learned how to act properly. It also often helped him in some situations. For example, when he got stuck and needed to perform a bigger jump than the network predicted. The agent also looked more human-like as its repeated runs were not the same. We are aware that introducing randomness into the agent's actual gameplay drifts away from the idea of his decision making being purely experience-based, but it actually resulted in more pleasant average game plays from the agent in the levels which he has not seen before.

We have observed the effects of randomness on the agent's overall behavior in 50 test epochs for different probability values of performing a random action

[34]For example, some levels have blue background while other have a purely black background. Also, the colors and shapes of obstacles are different for different texture styles.

and we have concluded that letting the agent have a one percent randomness rate is ideal because the randomness occurs very rarely and it actually resulted in occasionally better plays from the agent. With no randomness at all the agent's performances on the unseen levels were miserable. With only one percent randomness rate, the agent's performances improved significantly. Because the randomness can occur any time during the run, it also led to occasional worse plays, but these scenarios were interesting because they showed the agent's behavior in states in which he would not be able to find himself if it wasn't for the random actions. The agents were able to obtain similar results for every 50 epoch test run we have executed, which indicates that the randomness is small enough that on average it results in almost deterministic-like behavior. When setting the agent's randomness rate to higher values (we have experimented with up to 10 percent randomness rate), it caused that his actions were too much random and each 50 epoch test had completely different results.

On both of the levels that we have tested the agent on, he has performed miserably. Without allowing the agent to perform any random steps he always went in the opposite direction than the end of the level was and got stuck there. As this behavior was really strange, we tried to find out why would the trained neural network predict for the agent to go left. We have found out that on the level that the agent was trained on, the level starts with a clear blue sky in the background and ends with a winning flag which is at the end of the level right before a castle. Other levels of the same texture start with the castle being shown in the background (indicating that we have come from the previous level through the castle) and it can be the reason why the agent is acting so strange. He is most likely trying to go to the left because he has learned that on the left side from the castle is the finish line and he will get the largest reward for winning the game. The introduction of randomness helped significantly, but only in rare cases. In order for the agent to show signs of the learned behavior (jumping over enemies etc.), he needed to perform a series of actions at the very start of these levels, which got him just far enough that the castle was not visible anymore. Once the castle was not visible to the agent, he started showing signs of his learned behavior. In the first level of world two, the agent was able to get 300 pixels from the start in his best run, which means that he was able only to overcome the first enemy. In the second observed level (level 1 of world 4), the agent was able to perform better once he got away from the castle. In his best run, he was able to get up to the 1310 x-coordinate, which is approximately in the middle of the level. His behavior was much better and it showed some of the learned behavioral patterns that we have seen on the level he was trained on.

In conclusion, the agent's overall performance was terrible at first, but after determining the root of the issue and the introduction of randomness, we were able to observe some progress on the level. The agent was able to overcome some obstacles even though he had never played these levels. This is a positive sign as most of the implemented models were not able to train an agent capable of successfully overcoming multiple obstacles even for the level he was trained on. Without allowing the agent to perform random actions the results were terrible and the agent was not capable of playing any of the unseen levels. It would be interesting to train an agent on the level where the castle is shown in the background at the start to see if it would improve his performances on other unseen levels.

Chapter 6

Conclusion

This master's thesis focuses on preparing an intelligent agent for playing the Super Mario Bros. game. One of the main goals of the analysis part of this thesis was to research the most relevant state-of-the-art reinforcement learning approaches which could be suitable for this game and describe them. This was successfully achieved as the two most relevant techniques were selected and described in detail while also their weak points were exploited and solutions to them discussed.

The next goal was to find an appropriate tool that could be used for the interaction of the learning framework with the environment. Multiple tools that could be used were found, described, and from them was selected the one that was the most convenient, easy to use, and required almost no modifications for the game.

The main goal of this thesis was to implement selected models, train an agent for each of them and compare the agent's performances during the learning process. In the end, five variations of two selected RL approaches were implemented. For each of them, an agent was trained while their respective rewards gained during the training were collected. Then the performances of these models were compared, and the ones that suit the most for the Super Mario Bros. game were chosen. These models were the Enhanced DQN and the A3C.

After the model comparison, experiments with various modifications to the Enhanced DQN model were done in order to fine-tune it for the game. It was successfully determined that using a smaller neural network and letting it work with the original sized state representation image were the modifications that improved the agent's performance the most during its training. The agent was then trained for a greater number of epochs and improvements in his behavior were described. The observed effects of these modifications were then successfully applied to the A3C model, for which they proved to be unsuitable and made the agent's performances worse.

During the experiments we were having problems with executing large number of epochs that the models demand. This was caused not only by the limited resources we had access to but also by the fact that each action performed by the agent takes some time to execute[35]. But even these complications did not stop us from being able to successfully compare the models and later experiment with fine-tuning the selected one. It only caused that most of the initial agents were not trained enough to show human-like performances.

As the last goal, the in-game performance of the agent trained on the fine-tuned model was observed in the level that he was trained on and also in two different levels that the agents had never seen before. The agent showed great behavioral patterns in the level he was trained on and was able to clear the whole level with ease. In the levels, which the agent had not seen before, his performances were really bad, but we were able to determine the cause of the issue and find a solution for it. When the agent was allowed to perform random actions with just one percent chance, his performances increased greatly while keeping consistent results between multiple test runs. He was also able to get much farther, and whenever he performed a random action he kept showing all the learned behavioral patterns making his reactions look human-like.

[35]For example, the jump action takes some time until the agent lands and can effectively perform the next action.

Bibliography

[1] Mohammad Babaeizadeh et al. “Reinforcement Learning through Asynchronous Advantage Actor-Critic on a GPU”. In: *arXiv:1611.06256 [cs]* (Mar. 2017). arXiv: 1611.06256. URL: `http://arxiv.org/abs/1611.06256` (visited on 02/16/2022).

[2] Charles Beattie et al. “DeepMind Lab”. In: *arXiv:1612.03801 [cs]* (Dec. 2016). arXiv: 1612.03801. URL: `http://arxiv.org/abs/1612.03801` (visited on 02/21/2022).

[3] Greg Brockman et al. “OpenAI Gym”. In: *arXiv:1606.01540 [cs]* (June 2016). arXiv: 1606.01540. URL: `http://arxiv.org/abs/1606.01540` (visited on 04/03/2022).

[4] Deepanshu Mehta and Panjab University (UIET). “State-of-the-Art Reinforcement Learning Algorithms”. en. In: *International Journal of Engineering Research and* V8.12 (Jan. 2020), IJERTV8IS120332. ISSN: 2278-0181. DOI: `10.17577/IJERTV8IS120332`. URL: `https://www.ijert.org/state-of-the-art-reinforcement-learning-algorithms` (visited on 02/16/2022).

[5] Damien Ernst, Pierre Geurts, and Louis Wehenkel. “Tree-based batch mode reinforcement learning”. In: *Journal of Machine Learning Research* 6 (2005), pp. 503–556.

[6] Lina Faik. *Deep Q Network: Combining Deep & Reinforcement Learning.* en. Aug. 2021. URL: `https://towardsdatascience.com/deep-q-network-combining-deep-reinforcement-learning-a5616bcfc207` (visited on 02/12/2022).

[7] Scott Fujimoto, Herke van Hoof, and David Meger. “Addressing Function Approximation Error in Actor-Critic Methods”. In: *arXiv [cs, stat]: 1802.09477* (Oct. 2018). arXiv: 1802.09477. URL: `http://arxiv.org/abs/1802.09477` (visited on 02/16/2022).

[8] Hado Hasselt. “Double Q-learning”. In: *Advances in Neural Information Processing Systems*. Ed. by J. Lafferty et al. Vol. 23. Curran Associates, Inc., 2010. URL: `https://proceedings.neurips.cc/paper/2010/file/091d584fced301b442654dd8c23b3fc9-Paper.pdf`.

[9] Hado van Hasselt, Arthur Guez, and David Silver. “Deep Reinforcement Learning with Double Q-learning”. In: *arXiv:1509.06461 [cs]* (Dec. 2015). arXiv: 1509.06461. URL: `http://arxiv.org/abs/1509.06461` (visited on 02/12/2022).

[10] Anoop Jeerige, Doina Bein, and Abhishek Verma. “Comparison of Deep Reinforcement Learning Approaches for Intelligent Game Playing”. In: *2019 IEEE 9th Annual Computing and Communication Workshop and Conference (CCWC)*. Las Vegas, NV, USA: IEEE, Jan. 2019. ISBN: 9781728105543. DOI: `10.1109/CCWC.2019.8666545`. URL: `https://ieeexplore.ieee.org/document/8666545/` (visited on 02/12/2022).

[11] Michael Kearns and Satinder Singh. “Finite-Sample Convergence Rates for Q-Learning and Indirect Algorithms”. In: *Advances in Neural Information Processing Systems*. Ed. by M. Kearns, S. Solla, and D. Cohn. Vol. 11. MIT Press, 1999. URL: `https://proceedings.neurips.cc/paper/1998/file/99adff456950dd9629a5260c4de21858-Paper.pdf`.

[12] Alexander van de Kleut. *Gym Wrappers*. en-US. May 2019. URL: `https://alexandervandekleut.github.io/gym-wrappers/` (visited on 03/04/2022).

[13] Ajitesh Kumar. *Reinforcement Learning Real-world examples*. en-US. Jan. 2022. URL: `https://vitalflux.com/reinforcement-learning-real-world-examples/` (visited on 04/28/2022).

[14] Joel Z. Leibo et al. “Psychlab: A Psychology Laboratory for Deep Reinforcement Learning Agents”. In: *arXiv:1801.08116 [cs, q-bio]* (Feb. 2018). arXiv: 1801.08116. URL: `http://arxiv.org/abs/1801.08116` (visited on 02/21/2022).

[15] Volodymyr Mnih et al. “Asynchronous Methods for Deep Reinforcement Learning”. In: *Proceedings of The 33rd International Conference on Machine Learning*. Ed. by Maria Florina Balcan and Kilian Q. Weinberger. Vol. 48. Proceedings of Machine Learning Research. New York, New York, USA: PMLR, June 2016, pp. 1928–1937. URL: `https://proceedings.mlr.press/v48/mniha16.html`.

[16] Volodymyr Mnih et al. “Human-level control through deep reinforcement learning”. en. In: *Nature* 518.7540 (Feb. 2015), pp. 529–533. ISSN: 0028-0836, 1476-4687. DOI: `10.1038/nature14236`. URL: `http://www.nature.com/articles/nature14236` (visited on 02/12/2022).

[17] Volodymyr Mnih et al. "Playing Atari with Deep Reinforcement Learning". In: *arXiv:1312.5602 [cs]* (Dec. 2013). arXiv: 1312.5602. URL: `http://arxiv.org/abs/1312.5602` (visited on 04/28/2022).

[18] Arun Nair et al. "Massively Parallel Methods for Deep Reinforcement Learning". In: *arXiv:1507.04296 [cs]* (July 2015). arXiv: 1507.04296. URL: `http://arxiv.org/abs/1507.04296` (visited on 02/12/2022).

[19] Alex Nichol et al. "Gotta Learn Fast: A New Benchmark for Generalization in RL". In: *arXiv:1804.03720 [cs, stat]* (Apr. 2018). arXiv: 1804.03720. URL: `http://arxiv.org/abs/1804.03720` (visited on 05/03/2022).

[20] Ziad SALLOUM. *Double Q-Learning the Easy Way.* en. Dec. 2021. URL: `https://towardsdatascience.com/double-q-learning-the-easy-way-a924c4085ec3` (visited on 02/12/2022).

[21] David Silver et al. "Deterministic Policy Gradient Algorithms". In: *Proceedings of the 31st International Conference on Machine Learning.* Ed. by Eric P. Xing and Tony Jebara. Vol. 32. Proceedings of Machine Learning Research 1. Bejing, China: PMLR, June 2014, pp. 387–395. URL: `https://proceedings.mlr.press/v32/silver14.html`.

[22] Alexander L. Strehl et al. "PAC model-free reinforcement learning". en. In: *Proceedings of the 23rd international conference on Machine learning - ICML '06*. Pittsburgh, Pennsylvania: ACM Press, 2006, pp. 881–888. ISBN: 9781595933836. DOI: `10.1145/1143844.1143955`. URL: `http://portal.acm.org/citation.cfm?doid=1143844.1143955` (visited on 02/12/2022).

[23] Richard S. Sutton and Andrew G. Barto. *Reinforcement learning: an introduction.* Adaptive computation and machine learning. Cambridge, Mass: MIT Press, 1998. ISBN: 9780262193986.

[24] Jordi TORRES.AI. *The Bellman Equation.* Sept. 2021. URL: `https://towardsdatascience.com/the-bellman-equation-59258a0d3fa7`.

[25] Mike Wang. *Deep Q-learning Tutorial: minDQN.* 2020. URL: `https://towardsdatascience.com/deep-q-learning-tutorial-mindqn-2a4c855abffc`.

Appendix **A**

Acronyms

RL	Reinforcement Learning
DQN	Deep Q-learning
NES	Nintendo Entertainment System
A2C	Advantage Actor-Critic
A3C	Asynchronous Advantage Actor-Critic
TD3	Twin Delayed Deep Deterministic policy gradient algorithm
HP	Health
AI	Artificial Intelligence
ML	Machine Learning
API	Application Programming Interface
NES	Nintendo Entertainment System
GUI	Graphical User Interface
ReLu	Rectified Linear Unit
FIFO	First In, First Out
LSTM	Long Short-Term Memory
GAE	Generalized Advantage Estimator
GPU	Graphics Processing Unit

Appendix **B**

Contents of enclosed DVD

Below you can find the structure of the attached DVD, which contains all the files related to this master's thesis.

```
│ A3C ................................. Folder with A3C model related files
│ └── MyA3C ................ Folder with files related to modified A3C model
├── DDQN ........................ Folder with Double DQN model related files
├── DQN ................................. Folder with DQN model related files
├── EnhancedDQN ............. Folder with Enhanced DQN model related files
├── MyModel ............ Folder with files related to the fine-tuned final model
│   ├── Experiments ...................... Folder with experiment related files
│   ├── MyDQN.ipynb .................... Source code for the fine-tuned model
│   └── MyDQN_target.pt ....... Weights of the trained agents neural network
├── TD3 ................................. Folder with TD3 model related files
├── dict_utils.py ............... Functions for handling pandas dictionaries
├── Graphs_action_set_comparison.ipynb ... Action set comparison graphs
├── Graphs_experiments.ipynb .................. Experiment related graphs
├── Graphs_model_comparison.ipynb ............. Model comparison graphs
├── LICENSE ........................................... Legal use information
├── preprocessing_methods.py ....... Environment pre-processing methods
├── README.md ................. General information about the diploma thesis
├── requirements.py ............... Information about used python libraries
├── schejbal_diploma_thesis.pdf ............. Master's thesis text in PDF
└── schejbal-diploma-thesis.zip ....................... LaTeX source files
```