Playing NES through the use of a DQN

Lucas Mageste de Almeida   *Esther Luna Colombini*

Relatório Técnico - IC-PFG-18-02
Projeto Final de Graduação
2018 - Julho

UNIVERSIDADE ESTADUAL DE CAMPINAS
INSTITUTO DE COMPUTAÇÃO

Playing NES through the use of a DQN

Lucas Mageste de Almeida[*] Esther Luna Colombini[†]

17/07/2018

Abstract

This work uses the deep learning model known as DDQN to learn control policies for a variety of NES games. The input given to the model are raw pixels of the game screen and the output is a value function estimating future rewards. The learning process uses reinforcement learning exclusively and the same set of hyper-parameters for every game, with no adjustments. This work provides initial insight into RL used for NES games and provides full-support for the training of any other NES game using DDQN as its underlying learning agent.

1 Introduction

Progress in Artificial Intelligence (AI) has been continuously increasing over the last years [2]. Among the trends followed in the field, artificial neural networks are currently a top trend, mainly because of advancements in large data handling, computing power and the arrival of techniques such as backpropagation and deep learning neural networks [5, 3].

A neural network traditionally starts as a *tabula rasa*, and must be fed a training data set and learn patterns from it to successfully solve the problem it was designed to address for a given testing data set. The learning process and, hence, the description of ways in which to let a machine learn from a training set can be divided into three categories: supervised, unsupervised and reinforcement learning (RL) [13].

In supervised learning, the training data set is labeled by a human supervisor before being fed to the network, that is, the machine already knows the output for that given data set, and only needs to figure out the steps and patterns that produce the desired output given the input. The main advantage of supervised learning is the fact that the output for the training set is already known and hence, it is much easier to design the machine to learn from the data set, while the main disadvantage is the fact that there is a need of a human supervisor in order to treat a training data set first before using it. It is widely used in many Machine Learning (ML) systems across the world given its ease of design.

In unsupervised learning, on the other hand, the training data is not labeled by a human supervisor, and hence, the machine needs to learn from the raw training data without any

[*]Universidade Estadual de Campinas

[†]Institute of Computing, University of Campinas, 13081-970 Campinas, SP

additional information. The main advantage for unsupervised learning is the fact that there is no need to treat a training data set prior to using it and also that the machine is able to think independently from the way a human thinks, learning its own way of handling data in the learning process, without any human bias. As the main disadvantage we have the increased difficulty in designing unsupervised algorithms in comparison to supervised ones for many AI problems, which is reflected in its low usage in ML problems across the world.

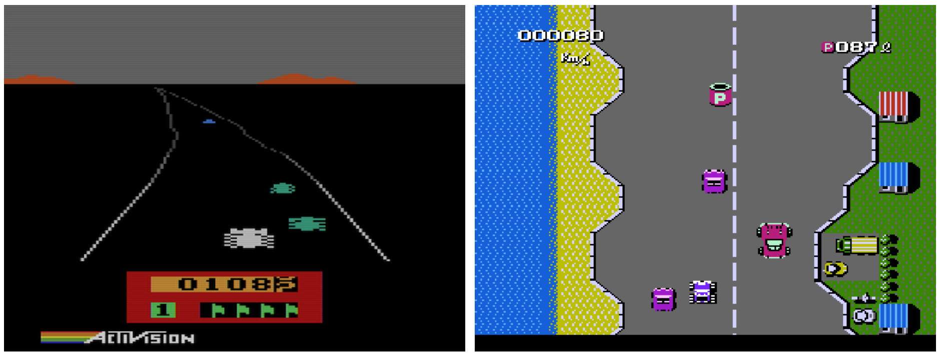
Reinforcement Learning (RL) tries to emulate human behavior and the fact that humans can learn from a completely clean slate, that is, acquiring knowledge from experience and perception without depending on inherited expertise or memory [15]. Mainly, it programs the machine (also known as learning agent) to take actions in an environment to maximize some notion of cumulative reward. The environment is mainly treated as a Markov decision process (MDP) [23] and although similar to a dynamic programming approach, a reinforcement learning approach does not assume knowledge of an exact mathematical model of the MDP and target large MDPs where exact methods become unfeasible. A direct consequence of this approach is that the training data set for the learning agent is entirely generated by the training agent itself and its actions on the environment and hence does not need a human expert to label it; a virtually infinite training data set is generated on demand and tagged according to the agent's reward system in order for the agent to progress further in its learning process, always looking for a greater reward.

A game (and moreover a computer game) is an excellent candidate for a reinforcement learning approach since every game has a set of rules, a set of available commands and a victory condition or some score system, which the player wants to achieve or maximize.

Google's Deepmind group used these concepts to create the first deep learning model (namely Deep Q-Networks, also knows as DQNs) to successfully learn control policies directly from high-dimensional sensory input using reinforcement learning. Their model was trained in a set of Atari 2600 games and was able to deliver state-of-the-art results without any adjustment of the architecture or the learning algorithm [17].

Many improvements were presented in the literature since DQN has been released, including Double DQN (DDQN) [8], a DQN algorithm that decouples the action selection and evaluation in the target computation step to avoid the overoptimism of DQN, what demonstrated to lead to more stable results.

Given the state-of-the-art results observed for Atari 2600 games, the primary purpose of this work is to adapt the DDQN to be used on Nintendo Entertainment System (NES) games and evaluate its performance. The Atari games are known for their simplicity and almost complete absence of information unrelated to the play-ability (that is, graphics that only appeal to the player eye and whose lack would not affect the player's performance), while NES games have a bigger screen size, broader color palette and commands and are, in general, far more complex and have far more "irrelevant" graphics (as we can see in Figure 1).



(a) Enduro, from Atari 2600 (b) Bump 'n' Jump, from NES

Figure 1: The figure shows a racing game from each platform. Notice how (b) is a far more complex game when compared to (a) taking detail and scenery into account. Additionally, the player in (b) has more freedom in their actions, being able to move horizontally, vertically and also being able to jump, while in (a) the player can only move horizontally

1.1 Objectives

The main objective of this work is to fully adapt the original DQN's Atari 2600 code to work with any NES game. For that, we will prepare a database of games from different genres so they can be learned through the algorithm. With this algorithm in hand, we aim at showing that the algorithms can be trained for different games and that corresponding metrics necessary to evaluate the techniques are available.

2 Revised Bibliography

Artificial Neural Networks (ANNs) are computing systems whose original goal was to emulate how a human brain works. As such, they are a collection of units which, in turn, are connected to each other, called artificial neurons, as a neuron in a human brain. Each connection between two artificial neurons, as synapses in a human brain, can transmit a signal from one artificial neuron to another. A typical process is that each artificial neuron gets signals from other artificial neurons, process them into its signal and then transmits it to more artificial neurons. Each ANN must, before being used to solve a task, be trained for that given task. This process requires that a large number of inputs are available and, during training, updates occur on the weights, i.e., on each connection between artificial neurons, to approximate the desired output.

According to the nature of connections between artificial neurons in an ANN, it can be classified as a feedforward, whose graph is a directed acyclic graph, or a recurrent neural network, whose graph is a cycled graph.

ANNs were successful in a variety of problems since their conception, for example in computer vision, speech translation and game playing [14]. They are still extensively explored and were responsible for much of the current increased interest in AI.

The concept of layers is essential for ANNs. A layer is a collection of artificial neurons which are not connected to each other but are, in turn, connected to the ANN's next layer's artificial neurons. An ANN typically consists of an input layer, which receives the raw input from the system, an output layer, which provides the desired output to a given input and an arbitrary number of hidden layers between the input and output layers, which further process the data before feeding it to the output layer. An ANN that has more than one hidden layer is called a Deep Artificial Neural Network or DNN.

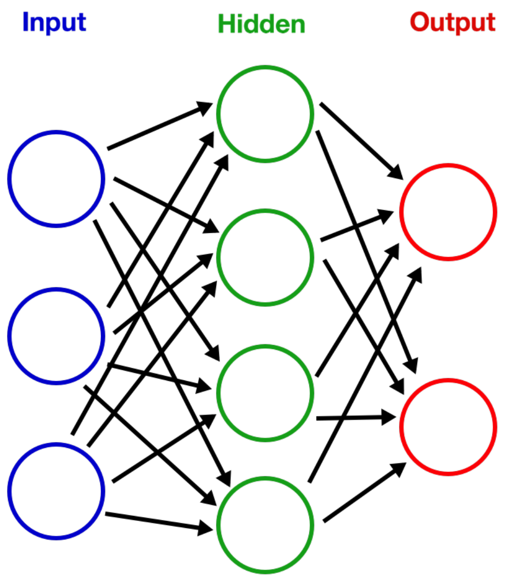
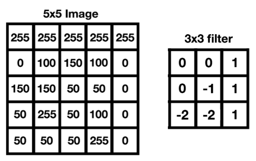


Figure 2: A representation of a Feedforward Artificial Network with a single hidden layer. The input, hidden and output layers are fully-connected and have, respectively, three, four and two artificial neurons

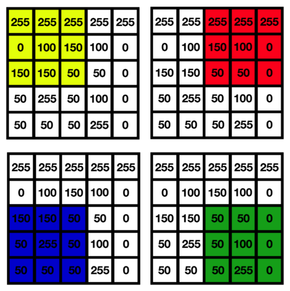
In its initial conception, layers in neural networks were fully-connected to the next layer, that is, all artificial neurons of a previous layer have a link to each of the neurons of the next layer. This approach initially worked, but as problems grew in complexity, so did the need for more hidden layers to process data. The computational cost for fully-connected layers, however, are substantial, since for two layers with n artificial neurons each, at least n^2 operations (n signals for each neuron in the second layer) need to be done to transfer data from one layer to the other. Because of this, many solutions were unfeasible precisely due to the time necessary to train a given network.

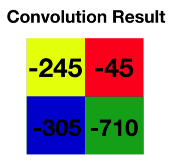
One of the ways to make ANNs' training less computational-heavy is through the use of convolutional layers [1]. Instead of being fully-connected, for convolutional layers, each artificial neuron in the next layer typically gets signals only from a small range of spatially-close artificial neurons in the previous layer. This process uses a filter over the last layer for each of the neurons in the next one.

To illustrate how a convolutional layer works, let us suppose we have an input layer, where each neuron represents each pixel of a gray-scale image of 5x5 size. To produce a convolutional layer, we may create a 3x3 filter and roll it through the image, from left to right and top-down. This filter applies an operation in a 3x3 portion of the image, and the result goes to the neuron of the next layer as its input. How fast this filter rolls (also known as its stride), that is, how many pixels (or neurons) it walks over at each new calculation step, affects the number of neurons we shall have at the next layer and hence, the number of operations we need to perform to propagate data from one layer to another. A typical stride value is 2, and as such we would pass this filter on this 5x5 image only four times, therefore generating only 4 artificial neurons for the next layer, instead of the 25 of the previous layer. At each layer, multiple filters may be passed on the last layer (generating multidimensional artificial neurons), and each filter used typically identifies a pattern. At first, the patterns are very rudimentary, representing the edges in an image: vertical, horizontal, diagonal, etc. In the next layer, those edges may be convoluted and identified as shapes: circles, squares, triangles. As a general rule, at each layer, the abstraction increases further, and more complex shapes and patterns may be identified in the image. Figure 3 depicts an example of how convolutional filters work.



(a) Image being analyzed and filter used in the convolutional layer


(b) Filter being applied to the image with its stride value of 2



(c) Result from the convolution operation to be used by the network's next layer

Figure 3: The figure shows the process of how a convolutional filter works, starting from an input layer representing an image 5x5, where each artificial neuron's value is equal to the image's greyscale pixel value. (a) shows both the filter and the image it is being applied on. (b) shows the four positions the filter is being applied to the image, since the filter has a stride of 2. This specific filter is applied by multiplying the filter's values to the corresponding positions in the image and adding the results. (c) shows the final result of the convolution process, where four new values are calculated from the previous twenty-five and which shall be used as the values of the artificial neurons of the next layer

Although the use of convolutional layers results in loss of data, the computational power needed to process a convolutional layer is much lesser than a fully-connected, allowing to concatenate several more hidden convolutional layers than their fully-connected counterparts for the same computational cost without compromising their efficiency. One effect of the use of convolutional networks in the network is the spatial-locality, that is, the patterns or shapes are identified only if the information that composes them are close to each other. But, since the filter is applied to every neuron region equally, this also means that the features or patterns may be located anywhere within the image, what has an excellent effect for image classification problems. For example, if we wanted to identify a nose in an image, all the information that translates as a nose needs to be next to each other in the image for it to be considered a nose or the ANN could identify a nose in an image with two scattered nostrils.

A DQN, as proposed in the original article, is a deep feedforward convolutional network that uses reinforcement learning instead of a supervised training approach. In the tasks, an agent interacts with an environment through a sequence of observations, actions, and rewards. The agent's goal is to select actions in a way to maximize cumulative future reward, or in other words, to use the ANN to approximate the reinforcement learning optimal action-value function.

The action-value function used in the original DQN article is fairly straightforward: it is the maximum sum of rewards r_t discounted by γ at each timestep t, achievable by a behavior policy $\pi = P(a|s)$, after making an observation (s) and taking an action (a)[16].

$$Q^{*}(s,a) = \underset{\pi}{max}\mathbb{E}[r_t + \gamma r_{t+1} + \gamma^2 r_{t+2} + ...|s = s_t, a = a_t, \pi] \tag{1}$$

The original formula has an overestimation problem since it selects a maximum value. Suppose, for example, one of the given estimates is very noisy and differs from its actual value. Since the formula picks the maximum value, it will end up selecting a value with a high positive error, propagating it further to other states [11].

To address this problem, the function used in the particular version of DQN used in this work is the one known as Double Learning [7]. Instead of using only one Q function, it uses two - Q_1 and Q_2 - which are independently learned. One of them is used to estimate the maximizing action and the other to estimate its value and either of them are update according to:

$$Q_1(s,a) = r + \gamma Q_2(s^{'}, argmax_a Q_1(s^{'}, a)) \tag{2}$$

$$Q_2(s,a) = r + \gamma Q_1(s^{'}, argmax_a Q_2(s^{'}, a)) \tag{3}$$

Decoupling the maximizing action from its value eliminates the overestimation problem found in the original algorithm of the DQN [9].

By taking advantage of the fact that in a DQN there are already two networks: Q and $\tilde{Q}$ (the target value) it is possible to simplify the algorithm above, although the networks are not entirely independent.

$$Q(s,a) = r + \gamma \tilde{Q}(s^{'}, argmax_a Q(s^{'}, a)) \tag{4}$$

This is the action-value function used in this version of DQN, also known as DDQN (Double Deep Q-Network), and although performance is not always improved, the stability of the learning process is greatly enhanced [7].

When a nonlinear function approximator (such as an ANN) is used to represent the action-value (also known as Q in DQN) function, reinforcement learning tends to be unstable or even diverge [18], mostly due to correlation between data and the fact that even small updates to the Q value may change the learning policy significantly [16].

To deal with these problems, a mechanism termed experience replay, which randomizes over the data, removing correlations in the observation sequence and also an iterative update that adjusts Q towards target values that are only updated periodically, effectively reducing correlations with the target, is in place.

It is not the purpose of this work to give a minute explanation about how DQN works but to explain its basic concepts. For more in-depth details about the operation of DQN, the reading of the original article is greatly encouraged.

The Deep Reinforcement Learning approach, of which DQN is a part of, has been consistently and successfully used to perform many complicated tasks ever since its development.

Using DQN as its underlying learning mechanism,the Deepmind group developed an agent (AlphaGo Zero) that learned only through self-play was able to achieve a win-ratio of 99.8% against other AIs in the game Go and further beating European Go champion by 5 games to 0, the first time an AI has done so [20].

An agent (Giraffe) was also created for playing the popular board game chess, where it achieved performance compared to the top 2.2% players of the world using only self-plays in order to learn [12].

A Learning Environment for the game Starcraft II was developed in Python (PySC2) by the Deepmind group in partnership with the game's company Blizzard [22]. Starcraft II is one of the biggest and most successful games of all time and players have competed in Starcraft tournaments for over twenty years. A good portion of StarCraft's longevity is due to the rich, multi-layered gameplay, which also makes it an ideal environment for AI research. Along with PySC2, many RL mini-games within the Starcraft II game were created by Deepmind where developers may evaluate their AIs built using PySC2. One such example is a Mineral Shard collector AI made using PySC2 [21].

DeepMimic uses Deep Reinforcement Learning to learn robust control policies capable of imitating a broad range of example motion clips, while also learning complex recoveries, adapting to changes in morphology, and accomplishing user-specified goals. The results are demonstrated in multiple characters and a large variety of skills, including locomotion, acrobatics, and martial arts [19].

Recently, a novel model called Reinforcement Learning with Decision Trees (RL-DT) was introduced, that uses decision trees to learn the model by generalizing the relative effect of actions across states. Such a model was tested in an environment of robots performing penalty kicks and trying to score goals. The algorithm was very successful in teaching the robot to score a goal quickly [10].

The original DQN article had a wrapper to communicate with an Atari 2600 emulator and thus allow the agent to effectively play games. A wrapper for the FCEUX Emulator for NES was developed by Ehren Brav in order to train an agent to play the game Super Mario Bros. Although the implementation utilized DDQN at its core, it was not essentially pure reinforcement learning, since the author had to apply an extra virtual reinforcement to the agent: a reward is given to the agent in case Mario moves forwards and a penalty in case he moves backward. Since the objective of a Mario level is to reach its end, which is always located in the far right, said reward makes sense, but it is not extracted from the game score itself. This exemplifies how poorly correlated the score in Super Mario Bros. is to the actual progress in a level: a player can actually run through the entire level without obtaining any score (which is tied to power-ups, coins and enemies killed) and still pass a level [4].

The adaptation of Ehren Brav's code which utilizes DDQN in its latest version (3.0) as its underlying learning agent is the base of this work.

3 Methodology

Many aspects of the project needed to be taken into consideration, besides the learning process itself, which used DDQN at its core. For simplicity's sake, the DDQN was not modified in any way, besides its hyperparameters, like its learning and discount rate or the number of hidden layers. But apart from it, it was still mandatory that any NES game could be supported by our algorithm (but that we focus on only a chosen few, best candidates for RL), that the training of an agent could be resumed from some failure (since the training process is very time-consuming) and that critical information about the development of the agent saved during its training in order to evaluate its progress.

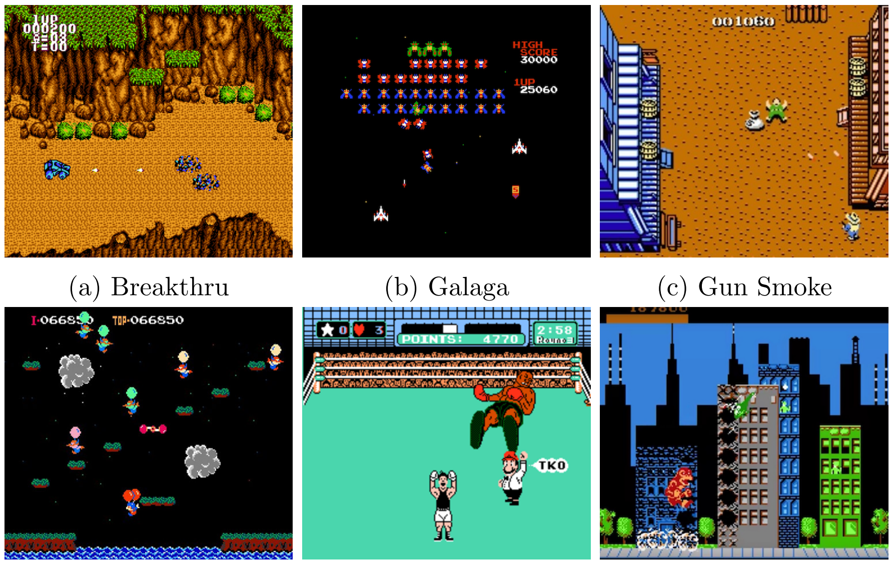
3.1 Game Database selection

In this work, the selection of games considered those where the score in them reflects real progress in the game. Therefore, exploration games were mostly not taken into account (like Super Mario Bros.), while fighting (like Double Dragon), racing (like Bump 'n' Jump) and shooter (like Galaga) games were outstanding candidates, what allows us to not use any approach besides a pure RL one.

The selection happened upon watching hundreds of NES games gameplay videos and selecting those that felt most suited to the RL approach. The 17 games supported and used for this work were: Bump 'n' Jump, Spyhunter, Rampage, Kungfu, Galaga, Joust, Lifeforce, Breakthru, Gradius, Contra, Double Dragon, Gun Smoke, Hudson's Adventure Island, Mach Rider, Punch-Out!!, Balloon Fight and Rad Racer. Some of them are presented in Figure 4.

3.2 DQN integration with NES

The initial integration of the DQN with the NES FCEUX emulator was a work of Ehren Brav. It was mainly a matter of providing an interface between the emulator and the



(a) Breakthru (b) Galaga (c) Gun Smoke

(d) Balloon Fight (e) Punch-Out!! (f) Rampage

Figure 4: Six of the database games chosen for testing the algorithm

network. More specifically, the network needed to get the game's information: for example, the screen pixels used as input, while the emulator required the output from the system, that is, the actions the network propose as the best ones for that given screen frame.

Since only this exchange of information is necessary, adapting the original article's DQN, that worked with an Atari 2600 emulator, to an NES emulator, was a task that did not need to change the actual code related to the network nor the functions call used between these models. It was merely to rewrite all necessary functions on the emulator side, including the needed ones for the interface, taking the Atari's implementation as a model. For example, acquiring the game's score and pressing a button (which must be the one provided by the network) are some functions that need to be implemented at the emulator's side, but their counterparts at the network's side may remain unchanged.

3.3 Providing support for NES games

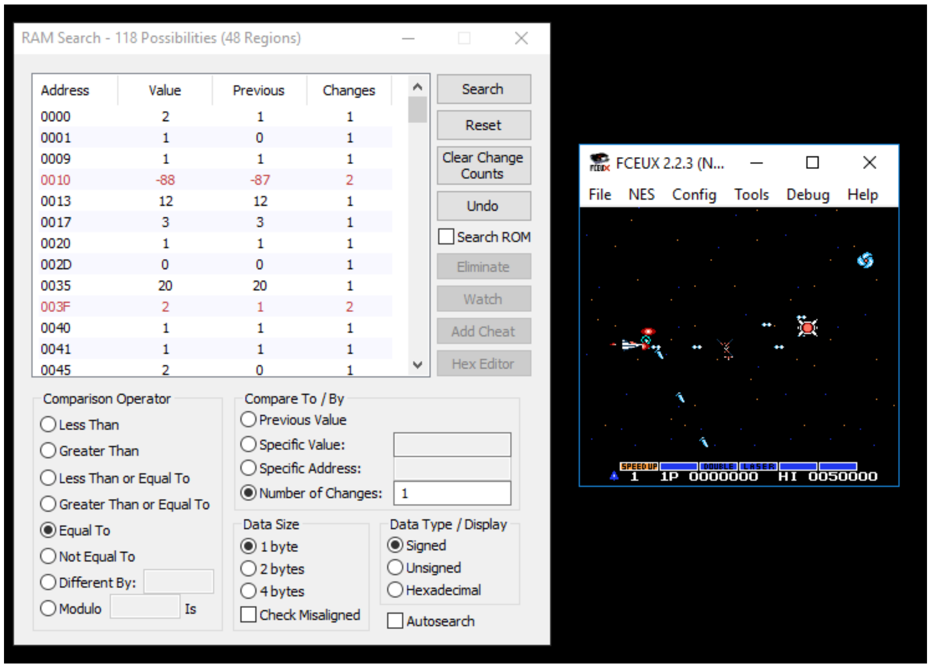
For this work, Ehren Brav's algorithm (and mainly the emulator's side functions) was adapted to support any game in the NES (given that information about its score, controls and game-over indication) and perform in a reinforcement learning approach. This was accomplished through the modularization of the functions responsible for acquiring the score, lives (when possible) and a game-over indication, as well as giving a list of the controls available for each game and skipping each game's title screen to be able to play the game itself.

In regards to rewards, our pure RL approach dictates that nothing should be used besides the raw pixels from the screen for the agent to learn. It means that the only rewards received by the agent are in the form of the game's score, which the agent will always pursue to maximize. Therefore, this score needs to be acquired, converted to value and used by our agent. The agent could obtain the score from the pixels and then use its value for a reward, but the process would require very high accuracy and a tailored solution for each game according to the font used, the pixels they occupy, the screen position and so on. Instead of doing so, we acquired the score directly from the RAM of each game, which is a more straightforward process than the one described above, but that yields the same results. The same procedure was done to acquire the remaining lives for a game (when possible).

Besides the score, concern was also the game-over indication. The games' behavior when the user enters into the game-over screen is very erratic: the user may be prompted to press some specific button combination to go back to gaming, they may go back to the title screen, they may simply be put back into the game with no action needed, etc. Besides this, there was also the problem of identifying when a game-over occurs in a game to start a routine to replay the game. To address these two problems, our solution was to identify the game-over through the RAM of each game, just like with the score and, besides this, for simplicity's sake, to reset the game as soon as a game-over is reached. In this way, the entire tailored routine needed for each game to replay is out.

Although for some very famous NES games, like Super Mario Bros. and Castlevania, which are intensely modified by the gaming community in order to change their graphics, levels, etc. (a process known as modding or hacking), much of their RAM was already mapped and widely available in forums and websites specialized at game hacking like Datacrystal, for other games the information was simply not found. So, to acquire it, a RAM searching tool within the FCEUX NES Emulator (Figure 5) was used to monitor the RAM of each game used in the database and find out which bytes corresponded to the **game's score** and **lives** and which byte could represent a **game-over indication**.

Finding such information is not a very straightforward task. The RAM in NES games has 1024 bytes, and any could be the information we are looking for since the way the data is stored does not necessarily reflect what happens on the screen. For example, the game Gun Smoke saves its score as digits in RAM, but instead of counting from 0 to 9 for each digit, it counts from 88 (which means 0) to 97 (which means 9). Therefore, much trial and error was needed to identify the information we were looking for, especially the ones stored in an elusive way: at first the focus was at finding exact values, for example, the value 5 in the RAM when we want to see the byte that dictates where is the digit 5 in the game's score, but this approach more than often yielded zero possibilities in the long run. After much trial, the usual solution that worked well was to look for changes in the value, not the exact amount, we were looking for. Therefore, to narrow our possible candidates, it is mandatory that we keep track of the changes in the game's information we are looking for and check which bytes have behaved the way we were expecting. A typical routine performed to acquire each score of the games used for this work was to keep track of all RAM byte values at first, then perform an action that increases the score, pausing the game and removing from our possible candidates all of the bytes that have not changed at all or changed more than one time between those two game instances. A handy routine was also to perform many actions unrelated to the information we are looking for (like exploring the stage in a game that does not reward a player for it when looking for the score information),



RAM Search - 118 Possibilities (48 Regions)

Address	Value	Previous	Changes
0000	2	1	1
0001	1	0	1
0009	1	1	1
0010	-88	-87	2
0013	12	12	1
0017	3	3	1
0020	1	1	1
002D	0	0	1
0035	20	20	1
003F	2	1	2
0040	1	1	1
0041	1	1	1
0045	2	0	1

Comparison Operator: Less Than, Greater Than, Less Than or Equal To, Greater Than or Equal To, [✓ Equal To], Not Equal To, Different By:, Modulo Is

Compare To / By: Previous Value, Specific Value:, Specific Address:, [✓ Number of Changes:] 1

Data Size: [✓ 1 byte], 2 bytes, 4 bytes, Check Misaligned

Data Type / Display: [✓ Signed], Unsigned, Hexadecimal

Autosearch

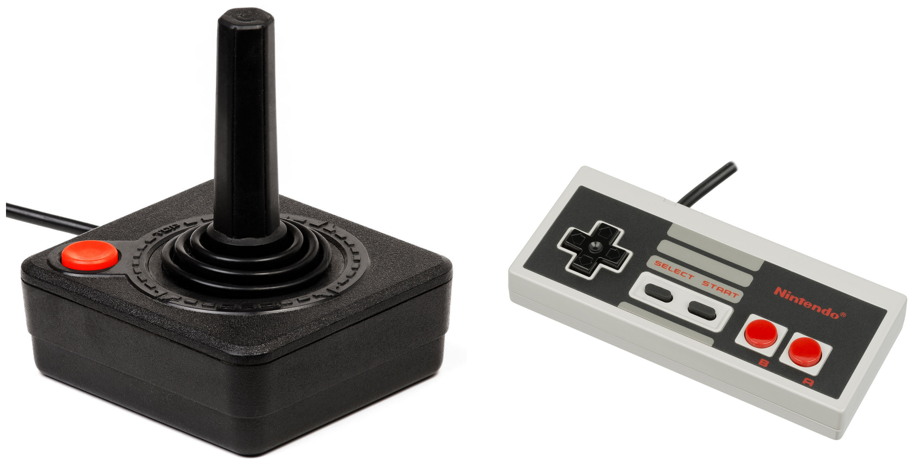
Figure 5: The RAM search tool embedded within the FCEUX Emulator for Windows OS running the game Gradius

then pause the game and removing all bytes from our candidates that did change. By doing both routines a couple of times, one can narrow the candidates to a few dozen or fewer byte positions and be able to track their behavior as the game is being played, eventually finding out where and how the information is stored for that specific game. The RAM search tool in FCEUX handles the pausing, filtering and monitoring of the game's RAM bytes nicely, so the process for finding each game's information, although tedious, was faster than imagined.

By repeating the process above for every game used in the proposed game database, their information was successfully mapped and ready to be used by the learning agent. This is though, unfortunately, a limitation in our work: that for every new game supported, its RAM needs to be scanned in order to find such information and doing so is not very trivial.

The **controls** for each supported game also need to be provided, that is, how many unique commands are there for that game. Finding the commands are relatively trivial, one needs to press all buttons and also their possible combinations to find out which commands do different actions in each game. The NES joystick (Figure 6), unlike modern ones, does not support analogical commands. Therefore all of its buttons have only two states: activated or not. In regards to directions, traditionally the player may move with the directional button, which has four courses (up, down, left, right) but they could also be combined in order to move the player to four possible extra directions (up and left, up and right, down and left, down and right). In regards to actions, the buttons traditionally related to them are two: A and B; they may be pressed individually or together. Besides the directional and action buttons, there are also the start button and the select button, which are mostly unrelated to gameplay, being used in menus. So, in total, there are nine possible actions related to movement (the eight directions above and not pressing any button at all) and four possible actions related to game actions (pushing: A; B; A and B; none at all). When combined, they yield a maximum of thirty-six possible actions, offering much more freedom than Atari's ten possible operations (five for directions, that is: staying idle and pressing up, down, left or right; and two for actions, that is: staying idle or pressing the action button). Upon testing all these button combinations for each game and checking the game's behavior, it is relatively easy to identify which combinations provide unique actions in each game and updating the algorithm's code to support these actions for each supported game.

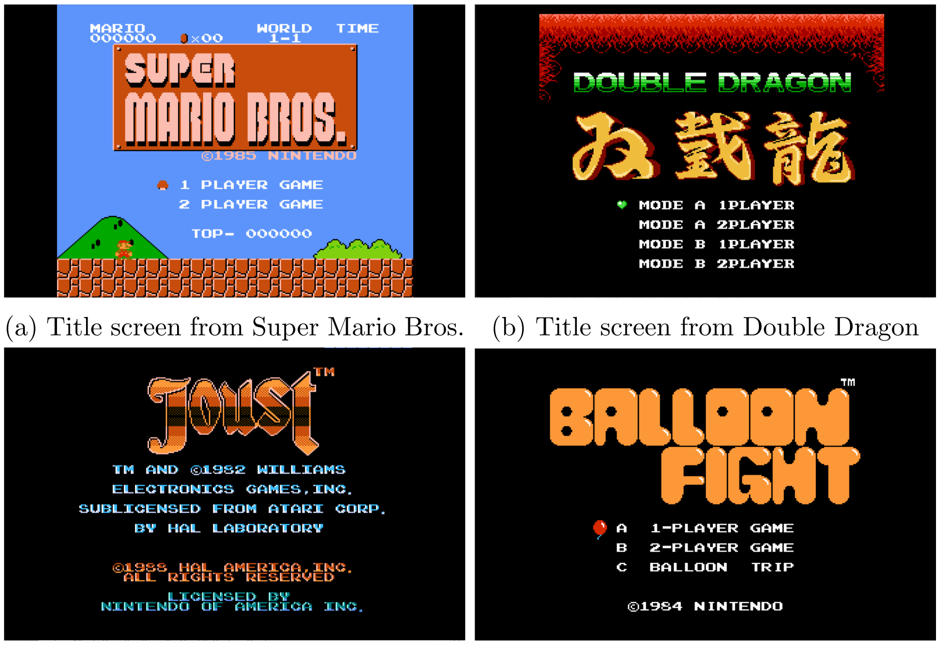
As a side note, it would actually be possible to train all games allowing all of these control combinations for every game, independent of its behavior. There could be multiple commands with the same effects which would probably add degrees of divergence and instability to the network, since it aims to select the best action for a given state and some of these actions would have an identical outcome.



(a) Original joystick from ATARI 2600 (b) Original joystick from NES

Figure 6: The figure shows both original controllers for the NES and Atari 2600 platforms. The directional stick in (a) may seem like an analog controller, but it is actually a digital one.

To skip the title screen (Figure 7) there were mainly two possibilities considered: pressing buttons specific to each game to get to the actual game screen or loading a save state (a file that holds all bytes of the RAM in a given point of the game) at the beginning of the actual game screen. The former, although more laborious method, was chosen, mainly because of the "randomness" (pseudo-randomness in reality) that we lose when loading a save state and is very important for machine learning to prevent over-fitting. Luckily, the majority of NES games has almost always the same routine to start its gameplay: wait for the title screen to appear, press A (which selects the option for a one-player game) and wait until the game begins. This routine worked for many of the games in our database. For those few ones that it did not work, there are methods invoked, specific to each game., what limits our approach since in order to support a new game that does not follow this routine, a new routine needs to be added via code, but creating a method is very straightforward and relatively simple for someone with little programming experience.



(a) Title screen from Super Mario Bros. (b) Title screen from Double Dragon

(c) Title screen from Joust (d) Title screen from Balloon Fight

Figure 7: (a), (b) and (d) are very similar title screens, although from very different genres. (c), on the other hand, has no options at all in the title screen, since there is just one mode in the game which is started by pressing a button. So, in reality, the routine to start the game in those four games above are identical.

With all these considerations, all of the games in our database were supported by the algorithm and, mostly, any other NES game could be supported by following the steps previously mentioned.

3.4 Resuming agent training from a failure

As already stated, the process of training the network is very time-consuming, so the need for failure handling and resuming training from a given point is essential. To address both of these problems, not only the network is saved periodically into a file, but its hyper-parameters, set in the training's initialization, are also kept in a separate file in order to resume training with the same hyper-parameters in case a failure occurs or if the training needs to be interrupted and resumed at any time.

Saving the network file also allows the evaluation of the agent's performance in a game at any given point by calling a test routine, which does not update the network file, just uses an action with the largest Q-value for each particular instant of the game and plays it, in order to witness the network's performance.

3.5 Progress over the training process

To be able to check the progress of the agent over time during the training process, an evaluation routine (much like the test routine) is called frequently to evaluate the progress of the agent. Upon the routine's end, a few relevant parameters that are related to the agent's development are stored:

- **the loss of the network**, which is defined as how much the network has deviated from the start of the evaluation to its end. This parameter should ideally decrease over time, since the network adapts over time to better predict which actions will yield more rewards and the more the network is trained the better at predicting it is and the lesser it should deviate from a previous state;
- **the mean Q-value**, defined as the average maximum value of Q for all states in the network.This should ideally increase over time, since the network should select a specific action for each instance of a game, in contrast with a random approach, which is the behavior of an untrained network;
- **the mean score** in the game achieved during the evaluation process, which may take multiple gameplays. This value is a much more practical way to evaluate the performance.

Now, besides the qualitative result of testing the final trained network by using it to play the game, the data above shares meaningful information about the whole training process.

3.6 Proposed Model

Taking into account what has been mentioned above, the final model used for this work was developed, divided into two parts: the model representing how a new game gets supported (as shown in Figure 8) and the model describing the learning process (as shown in Figure 9).

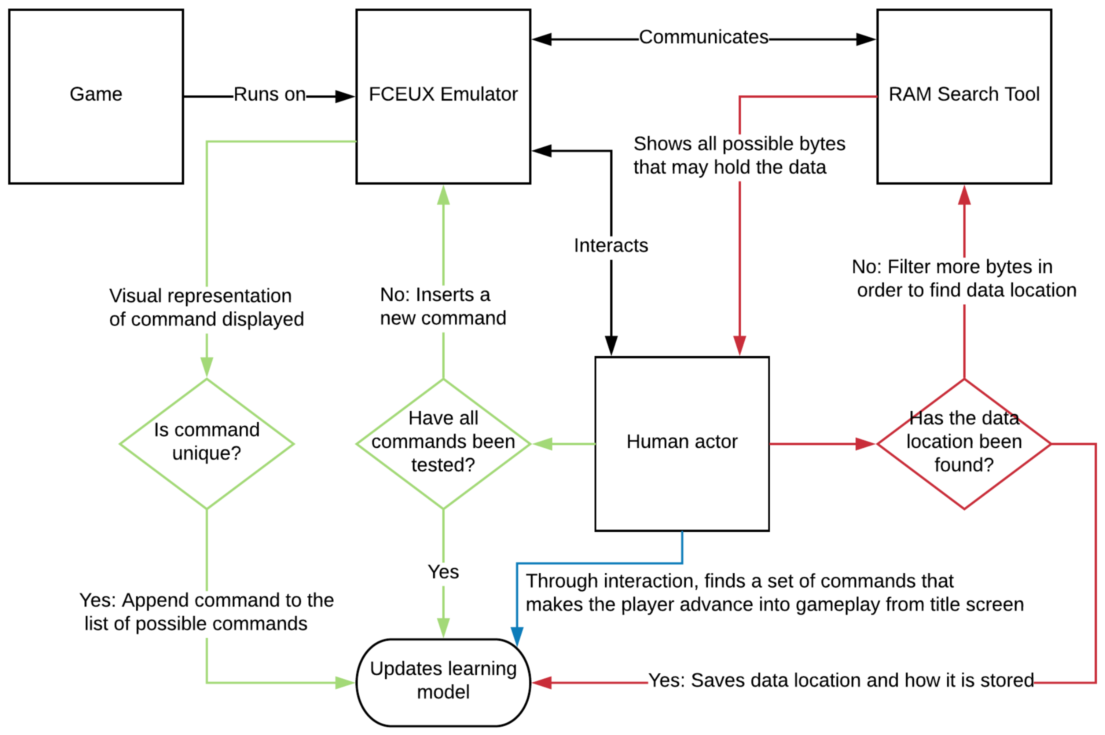

Game
Runs on
FCEUX Emulator
Communicates
RAM Search Tool
Shows all possible bytes that may hold the data
Interacts
No: Filter more bytes in order to find data location
Visual representation of command displayed
No: Inserts a new command
Is command unique?
Have all commands been tested?
Human actor
Has the data location been found?
Yes
Yes: Append command to the list of possible commands
Through interaction, finds a set of commands that makes the player advance into gameplay from title screen
Updates learning model
Yes: Saves data location and how it is stored

Figure 8: The model shows the process to support a new game. Each color represents a flux of steps that are done in order to acquire relevant data from the game. It requires the intervention of a human supervisor.

4 Experiments and Results

At first, after executing the necessary modifications to support the game database, a few games were tested using the same hyper-parameters as Ehren Brav used in his implementation.

The initial results were not very promising. Although the network did indeed learn, as can be seen by the data below, the learning rate was prolonged and very inferior to that of a human player, even a beginner. The training process of the network for the game Galaga may be seen in Figure 10 below.

Since the training process takes quite some time to be done and the fact that the agent indicated that it could learn further with more training time, some modification to the network was considered to accelerate the training process. Considering hyper-parameters, an adaptive learning rate α which decreased at each epoch k and a discount rate γ which increased at each epoch k until the value of 0.99 was found to achieve better results for Atari games [6].

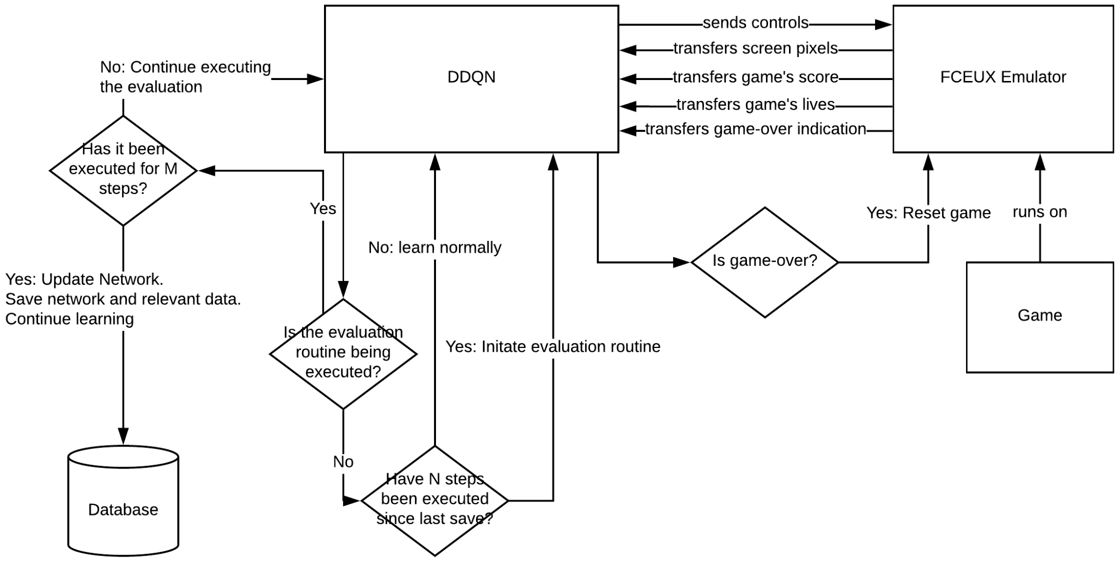
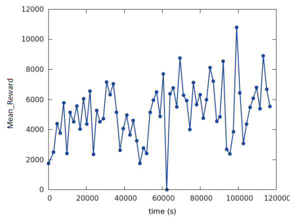
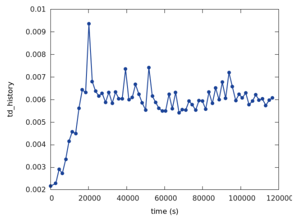

DDQN
FCEUX Emulator
sends controls
transfers screen pixels
transfers game's score
transfers game's lives
transfers game-over indication
No: Continue executing the evaluation
Has it been executed for M steps?
Yes
Yes: Update Network. Save network and relevant data. Continue learning
Database
Is the evaluation routine being executed?
No: learn normally
Yes: Initate evaluation routine
No
Have N steps been executed since last save?
Is game-over?
Yes: Reset game
runs on
Game

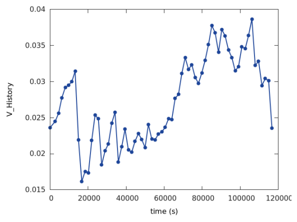
Figure 9: The model represents the learning process for any game supported by the algorithm. It is totally automatized and does not require the intervention of a human supervisor.



(a) Mean Reward for game Galaga over training



(b) Loss of the network over time during training



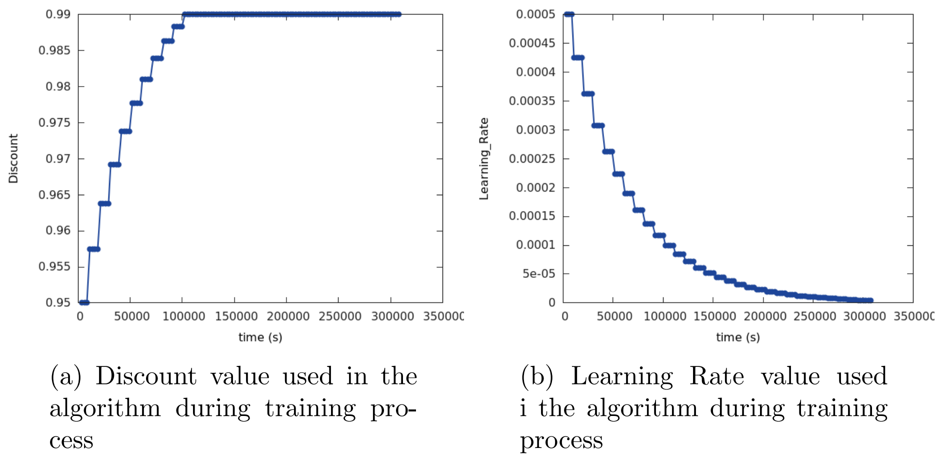
(c) Mean Q-value of the network over time during training

Figure 10: The figure shows the learning process for a pure RL approach in game Galaga without any modifications to the original algorithm. Although the mean score seems to be improving over time, in case a linear fit is done for (a), figure (b), that although initially increasing its value, as expected, suffers a sudden decrease, and figure (c), that does not decrease over time (as expected), have an erratic behavior

$$\alpha_{k+1} = \alpha_k * 0.98, \alpha_0 = 0.0005 \tag{5}$$

$$\gamma_{k+1} = 1 - 0.98 * (1 - \gamma_k), \gamma_0 = 0.95 \tag{6}$$

The initial values for α and γ were set, respectively, at 0.0005 (double the original value of 0.00025) and 0.95 (the original value was fixed at 0.99). An epoch was set at 200.000 learning steps of the algorithm, so the values above would be updated after these many steps. The effects may be seen in Figure 11 below.



(a) Discount value used in the algorithm during training process

(b) Learning Rate value used i the algorithm during training process

Figure 11: Figures (a) and (b) show the behavior of discount and learning rate over time during training for any game. Both of them increase and decrease exponentially, as can be expected from their functions, but only (a) stops increasing after reaching a limit (0.99), while (b) continues to decrease over time, as long as floating point precision allows it to.

After the modification above the network achieved better learning rates quicker than before, although results were still far from these expected of a human player. Below we may see the results for the same game Galaga in Figure 12.

To analyze the effects of the approach described above in a game after much more training steps, when the learning rate approaches 0, the same game Galaga was trained again and its results are shared below in Figure 13.

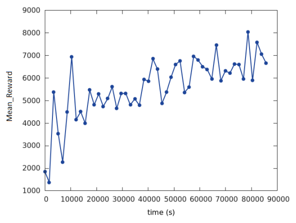
Other games of the database were tested, but none of the games tested had a performance comparable to that of a human player and evolved in a much more erratic way than Galaga, probably due to the fact that Galaga was the game with the most simple set of commands and with the simplest graphics out of all games selected and hence, less complex.

All of the tests above were conducted in an Intel(R) Core(TM) i7-7700 CPU @ 3.60GHz, 16GB of RAM, and an NVIDIA GTX Corporation Device 1080 running Ubuntu 16.04.4 LTS.

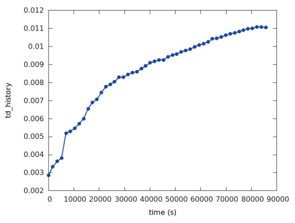
5 Conclusion

Although the pure RL approach worked, its results were not very encouraging, probably due to NES games superior complexity against Atari 2600 games.

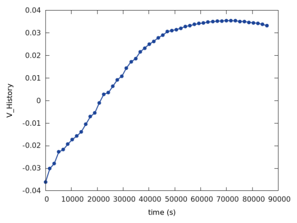
Although results may at first seem discouraging, the interface and methods necessary



(a) Mean Reward for game Galaga over training

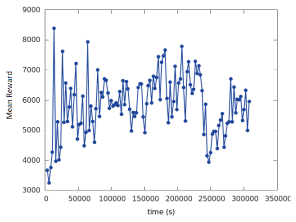


(b) Loss of the network over time during training

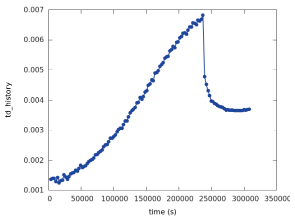


(c) Mean Q-value of the network over time during training

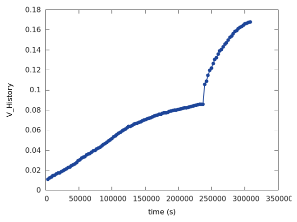
Figure 12: The figure shows the learning process for a pure RL approach in game Galaga with a decreasing LR and increasing Discount up to 0.99. (a), (b) and (c) show a much less erratic behavior now, and although (a) did not achieve a score as high as the previous approach this time, the deviation of the curve is much lesser than before and certainly increasing over time. (b) now, although with a clear behavior, deviates from the expected, increasing over time, while it should be decreasing. (c), on the other hand, shows a consistent increasing behavior, just like expected



(a) Mean Reward for game Galaga over training



(b) Loss of the network over time during training



(c) Mean Q-value of the network over time during training

Figure 13: The figure shows the learning process for a pure RL approach in game Galaga with a decreasing LR and increasing Discount up to 0.99 after many learning steps. Notice how after 200,000 seconds of training, (a), (b) and (c) deviate behavior drastically, with (a) suddenly decreasing to start increasing again, (b) suddenly decreasing as expected and (c) increasing with a higher rate. Said behavior may be attributed to the learning rate, as Figure 11 shows, to a learning rate very close to the value of 0. It could be that the steps taken during training were too large and it was not able to pursue a specific local maximum among the possibilities, instead switching over and over, while with smaller steps the algorithm was able to concentrate its efforts into achieving a specific local maximum.

to test any of the games above and any other game in the NES are now open source and available and future works with different approaches and more training time are greatly encouraged.

More specifically, for future works, it is advised that:

- the number of the network's layers are increased to further increase abstraction;
- the initial values and epoch of LR and discount should also be modified and the results analyzed;
- the networks are given much more time to train.

Unfortunately, the options above were out of scope for this work due to time constraints.

The algorithm described in this work is available at https://github.com/lmageste/DeepQNetwork.

References

[1] *A Beginner's Guide to Deep Convolutional Neural Networks (CNNs)*. 2017. URL: `https://deeplearning4j.org/convolutionalnetwork`.

[2] *AI Sector Maturing As Funding Shifts to Later Stage*. 2018. URL: `https://www.venturescanner.com/blog/2018/ai-sector-maturing-as-funding-shifts-to-later-stage`.

[3] Sarah Allidina. *The rise of artificial intelligence in 6 charts*. 2016. URL: `https://www.raconteur.net/business/the-rise-of-artificial-intelligence-in-6-charts`.

[4] Ehren Brav. *Teaching Your Computer To Play Super Mario Bros. – A Fork of the Google DeepMind Atari Machine Learning Project*. 2016. URL: `http://www.ehrenbrav.com/2016/08/teaching-your-computer-to-play-super-mario-bros-a-fork-of-the-google-deepmind-atari-machine-learning-project/`.

[5] Luke Dormehl. *What is an artificial neural network? Here's everything you need to know*. 2018. URL: `https://www.digitaltrends.com/cool-tech/what-is-an-artificial-neural-network/`.

[6] Vincent François-Lavet, Raphaël Fonteneau, and Damien Ernst. "How to Discount Deep Reinforcement Learning: Towards New Dynamic Strategies". In: *CoRR* abs/1512.02011 (2015). arXiv: `1512.02011`. URL: `http://arxiv.org/abs/1512.02011`.

[7] Hado V. Hasselt. "Double Q-learning". In: *Advances in Neural Information Processing Systems 23*. Ed. by J. D. Lafferty et al. Curran Associates, Inc., 2010, pp. 2613–2621. URL: `http://papers.nips.cc/paper/3964-double-q-learning.pdf`.

[8] Hado van Hasselt, Arthur Guez, and David Silver. "Deep Reinforcement Learning with Double Q-learning". In: *CoRR* abs/1509.06461 (2015). arXiv: `1509.06461`. URL: `http://arxiv.org/abs/1509.06461`.

[9] Hado van Hasselt, Arthur Guez, and David Silver. "Deep Reinforcement Learning with Double Q-learning". In: *CoRR* abs/1509.06461 (2015). arXiv: `1509.06461`. URL: `http://arxiv.org/abs/1509.06461`.

[10] Todd Hester, Michael Quinlan, and Peter Stone. "Generalized Model Learning for Reinforcement Learning on a Humanoid Robot". In: (May 2010), pp. 2369–2374.

[11] Jaromír Janisch. *Let's make a DQN: Double learning and prioritized experience replay.* 2016. URL: `https://jaromiru.com/2016/11/07/lets-make-a-dqn-double-learning-and-prioritized-experience-replay/`.

[12] Matthew Lai. "Giraffe: Using Deep Reinforcement Learning to Play Chess". In: *CoRR* abs/1509.01549 (2015). arXiv: `1509.01549`. URL: `http://arxiv.org/abs/1509.01549`.

[13] Ronald van Loon. *Machine Learning Explained: Understanding Supervised, Unsupervised, and Reinforcement Learning.* 2018. URL: `https://www.datasciencecentral.com/profiles/blogs/machine-learning-explained-understanding-supervised-unsupervised`.

[14] Dean Macri. *An Introduction to Neural Networks With an Application to Games.* 2017. URL: `https://software.intel.com/en-us/articles/an-introduction-to-neural-networks-with-an-application-to-games`.

[15] Rahul Matthan. *Tabula Rasa.* 2017. URL: `https://www.livemint.com/Opinion/cJq5l1OkKI6xpJavwcYO2O/Tabula-Rasa.html`.

[16] Volodymyr Mnih et al. "Human-level control through deep reinforcement learning". In: *Nature* 518 (Feb. 2015), 529 EP -. URL: `http://dx.doi.org/10.1038/nature14236`.

[17] Volodymyr Mnih et al. "Playing Atari with Deep Reinforcement Learning". In: *CoRR* abs/1312.5602 (2013). arXiv: `1312.5602`. URL: `http://arxiv.org/abs/1312.5602`.

[18] Joseph O'Neill et al. "Play it again: reactivation of waking experience and memory". In: *Trends in Neurosciences* 33.5 (2010), pp. 220–229. ISSN: 0166-2236. DOI: `https://doi.org/10.1016/j.tins.2010.01.006`. URL: `http://www.sciencedirect.com/science/article/pii/S0166223610000172`.

[19] Xue Bin Peng et al. "DeepMimic: Example-Guided Deep Reinforcement Learning of Physics-Based Character Skills". In: *ACM Transactions on Graphics (Proc. SIGGRAPH 2018 - to appear)* 37.4 (2018).

[20] David Silver et al. "Mastering the game of Go without human knowledge". In: *Nature* 550 (Oct. 2017), 354 EP -. URL: `http://dx.doi.org/10.1038/nature24270`.

[21] Chris Hoyean Song. *StarCraft II - pysc2 Deep Reinforcement Learning Examples.* 2018. URL: `https://github.com/chris-chris/pysc2-examples`.

[22] Oriol Vinyals et al. "StarCraft II: A New Challenge for Reinforcement Learning". In: *CoRR* abs/1708.04782 (2017). arXiv: `1708.04782`. URL: `http://arxiv.org/abs/1708.04782`.

[23] George Weiss. "Dynamic Programming and Markov Processes. Ronald A. Howard. Technology Press and Wiley, New York, 1960. viii + 136 pp. Illus. $5.75". In: *Science* 132.3428 (1960), pp. 667–667. ISSN: 0036-8075. DOI: `10.1126/science.132.3428.667`. eprint: `http://science.sciencemag.org/content/132/3428/667.1.full.pdf`. URL: `http://science.sciencemag.org/content/132/3428/667.1`.