Modelling Human Player Sensorial and Actuation Limitations in Artificial Players

André Gonçalo Henriques Soares

Thesis to obtain the Master of Science Degree in

Information Systems and Computer Engineering

Supervisor: Prof. Carlos António Roque Martinho

Examination Committee

November 2019

Acknowledgments

I would like to thank my parents and brother for their friendship, encouragement, caring and patience over all these years, for always being there for me through all the challenges and without whom this project would not be possible. To my girlfriend who helped, supported and encouraged me through the last hurdles of my work.

I would also like to acknowledge my dissertation supervisors Prof. Carlos António Roque Martinho for his insight, support, sharing of knowledge and understanding that has made this Thesis possible on various levels.

Last but not least, to all my friends and colleagues that helped me grow as a person and were always there for me during the good and bad times in my life.

To each and every one of you – Thank you.

Abstract

In game design, one of the most important tasks is associated with the playtesting process, as this is where game designers are able to understand if the game experience they are trying to create is indeed being passed to the players. How a player perceives and reacts to a game should be important in the game design process.

Work developed in the Deep Learning research field has proved to be a great source of information as to understand how an agent is capable of achieving great runs and scores playing various games from scratch. The capability of testing different games using a screen capture artificial player powered by Convolutional Neural Network (CNN) allows for a good understanding of how an agent is capable of extracting important features from the game screen without additional information.

This Thesis, took the work developed in the Deep Reinforcement Learning field applied to Atari environments, mainly Deep-Q Networks, modulated different types of player limitations, tested and documented the results achieved. The results seem to indicate the existence of different types of playing patterns for different limitations.

Keywords

Game Design; Playtesting; General Video Game Playing; Deep Learning; Reinforcement Learning; Convolutional Neural Network; Deep Q-Network; Artificial Player; Limitation; Reverse; Delay; Skip; Patterns

Resumo

Em game design, uma das tarefas mais importantes está associada com o processo de playtesting, já que é aqui que os designers conseguem perceber e entender se a experiência de jogo que eles pretendem criar está de facto a ser transmitida aos jogadores. A forma como um jogador percebe e reage a um jogo é extremamente importante para o processo de design do jogo.

Trabalhos desenvolvidos no campo de Deep Learning provaram ser uma grande fonte de informação na forma de entender como um agente é capaz de obter boas runs e scores ao jogar vários jogos sem qualquer conhecimento prévio sobre o jogo. A possibilidade de testar um variado leque de jogos usando agentes artificiais com captura de ecrã proporcionado por uma Convolutional Neural Network, permite uma boa compreensão em como um agente é capaz de extrair features importantes a partir do ecrã do jogo sem qualquer informação adicional.

Esta Dissertação pegou em trabalhos desenvolvidos no campo de Deep Reinforcement Learning aplicado a ambientes de Atari, modulou diferente tipos de limitações, testou e documentou os resultados obtidos. Os resultados obtidos tendem a indicar a existência de diferentes padrões na forma de jogar dos agentes para cada limitação testada.

Palavras Chave

Game Design; Playtesting; General Video Game Playing; Deep Learning; Reinforcement Learning; Convolutional Neural Network; Deep Q-Network; Jogador Artificial; Limitação; Ação Contrária; Atraso Ação; Saltar Frames; Padrões

Contents

List of Figures

List of Tables

List of Algorithms

Acronyms

CNN	Convolutional Neural Network
DQN	Deep Q Network
GGP	General Game Playing
GVGP	General Video Game Playing
VGDL	Video Game Description Language
ALE	Arcade Learning Environment
GVG-AI	General Video Game AI
NEAT	Neuroevolution of Augmenting Topologies
A3C	Asynchronous Advantage Actor-Critic
PPO	Proximal Policy Optimization

1 Introduction

Contents

1.1 Motivation

Looking at the development process of a new game, playtesting is seen as an indispensable part of the game design and development process but in part somewhat costly. Game developers need to expose their games to their intended audience and only through playtesting will a level designer understand whether the components he or she carefully assembled are able to elicit the game experience they were designed for, identifying potential design flaws and gather feedback. This game experience can vary from player to player, taking into account details related to how a player can perceive and interact with a game.

Different players can perceive the game differently and the way they process what they are looking at can possibly lead to different ways of playing a game and to the formation of different strategies in comparison to other players.

For example, people with eye color deficiencies, perceive the game in a different way from someone with normal vision. The person affected by color blindness can end up developing different strategies on how to play the game. Take Splatoon 2[1], as an example. Ink is the main projectile used in the game and is also a way of moving through the game's levels. The ink used by each team have different colors and is used by each players teams to perform various actions. If a player has difficulty distinguishing between his team color and the enemies team color he might have to find different ways and strategies in order to gain advantages.

In the opposite spectre, the way a player interacts with a game can in itself be different from how another one does it and that might prompt different players to develop different strategies on how to play the same game. If we look at the comparisons of Actions per Minute (measure of how many clicks and key presses a player can perform in sixty seconds) between professional players and casual players of the game Starcraft 2[2], there's a big disparity in numbers. According to an article by Wong [1] a professional player averages about 300 APM going up to 400 and beyond, while a casual player averages around 60 going up to 100 APM. In this case the strategies applied by professional players will be different from casual players, since they are able to act and react faster.

The fact that the earliest video game players are aging can be seen as a possible limiting factor in terms of reflex capabilities that should be taken into account during the design process of new games, as developers try to appeal to an older fan base. Although in recent years the gaming industry has become more friendly and accessible towards people with limitations like eye color deficiency with for example, giving them multiple color blind modes in game, it can still be an issue designing games that take all perception problems into account. So, it might be beneficial to take these conditions into consideration during the game design process.

[1]Splatoon 2, Nintendo, 2017, Nintendo Switch, `https://splatoon.nintendo.com/`
[2]StarCraft II: Wings of Liberty, Blizzard Entertainment, 2010, Dustin Browder and Chris Sigaty, PC, `https://starcraft2.com/`

Having the game developers capable of exposing their games to different types of artificial players as a way of simulating a specific type of audience might bring good results in their chase for the best design options. The area of General Game Playing (GGP) has specialized itself in producing artificial agents capable of playing games they never been in contact before, with a variety of approaches, ranging from different types of frameworks to different types of learning methods.

1.2 Problem

Knowing that the playtesting process is a time and resource consuming task, in this work it's interesting to understand how and if it's possible to provide the game designer with support during the level design process. What if the game designer isn't continuously forced to test his work with real players every time he needs to perform changes to his game and instead be able to use a model that can replicate a human like behaviour? Or if the game designer wants to test his work with someone that may have some kind of physical limitation?

In order for that to happen we need to understand if there is a way of replicating players limitations in an artificial manner, using the game screen as its main source of information. And so the problem we intend on tackling is as follows:

Is it possible to accurately modulate a certain type of player limitations through an artificial agent?

Can this model replicate the nuances of different types of player limitations, be it either in how the game information is perceived in the form of input and how the following processed information is passed and executed as output of the game information

1.3 Hypothesis

In order to solve the problem described, we'll be looking at replicating an artificial player capable of having a level of resemblance between itself and a real human player as the basis to our hypothesis.

Since we are trying to understand how the manipulation of input and output of game information influences the type of results and strategies the artificial player might have, our proposed model needs a way of how to process the game information in a way similar to how a human processes what he sees on screen while playing a game. For that we need to implement a deep learning method that allows the agent to learn from the game screen.

And so, our proposed hypothesis consists on using a model that allows us to replicate different types of sensorial and actuation limitations on top of the deep learning algorithm we use to create the artificial player. Using specific scenarios(delays, lag, action limitations), we want to show that this type of model is capable of presenting results that resemble a limited player.

1.4 Contributions

This project's work aims at the possibility of providing the game designers with a set of tools that might facilitate their work during the playtesting process. The capacity of executing various sets of tests using our proposed model to create the artificial player, before diving into real human players tests could allow the game designers the ability to predict in part a portion of what might happen during the playtesting with real players, allow them to plan and make changes in advance and to know what aspects they should be focusing, taking into account what they most deem important.

It might get used for simulating certain types of players or to simulate certain types of situations. Other contributions include a collection of papers and descriptions relevant to the state of the art studied all of which have helped arrive at the current solution.

1.5 Organization of the Document

This thesis is is organized as follows: in Chapter 1 we introduce the motivation behind the work, the problem raised by it, and the hypotheses based on the research done. The chapter ends with the document's delimitation. In chapter 2 we give a bit of the background of the areas in which this thesis is operating. In a first section, there's a look at the area of GGP, the ideas and frameworks surrounding it, followed by a section that presents the competition borne from it. After that, a section dedicated to the presentation of various deep learning algorithms, their applications and finally, a summary of the research done and a discussion about how it influenced the decisions made in the implementation and evaluation of the solution. In chapter 3 we look at how to replicate the chosen algorithm original results as well as describing the model proposed. Chapter 4 presents the results obtained using our model and trained agents, with the according analyses. Finally, in Chapter Chapter 5 the implications of the work results are discussed, and some routes in which the work can be continued and improved upon in the future.

2 Related Work

Contents

In order to form and develop a solution to the main problem exposed and to backup the hypothesis given, there had to be an extensive research focusing on previous works and approaches taken in the area, as well as crucial information that grants the justification as to why the path taken to solve the problem is the one pursued and not other possibilities. For that, this chapter is focused on taking a look at the various areas of this study, explain them and seek further knowledge on the subject.

First, a description of the GGP area field, an area to which this work pretends to provide some contribution, followed by a description of various deep learning based algorithms usually used in order to create successful learning artificial agents as well as various examples of their implementation.

2.1 General Game Playing

General Game Playing [2] is seen as the idea of having an Artificial Intelligence capable of playing a game it has never been in contact with before, without any kind of human help. This concept is put to action by having a General Game Player, which is an artificial agent, that learns to play the game it is put in by itself. As described by Genesereth and Love, these agents are systems capable of accepting the descriptions and the list of possible actions of a game which means they don't know what are the game's rules beforehand [2]. With the information that is given, the agent uses it to evaluate what is the best possible course of action to take in order to play the game. By continuously playing the game, the agent will be able to learn from its past experiences and develop new strategies that allow it to have a better performance and edge closer to the way a human player might have played that game.

This type of approach is not easy to achieve because the work that the artificial agent has to accomplish can't be done by preconceived algorithms for specific games. Also, the artificial agent must be able to find the best approach to the problem, while improving its efficiency.

The GGP approach prevents the developers from designing multiple agents for various different games, and instead the developer should only focus on developing a single agent capable of recognizing, adapting and playing the different types of games to which it is exposed.

Through the years the usual approach to create GGP artificial players was to use minimax based game-tree search methods with self learned evaluation heuristic functions, until the works of people like Björnsson and Finnsson that successfully applied Monte-Carlo Tree Search methods in their approach, creating an artificial player called CadiaPlayer, that won 2 GGP competitions [3].

Nowadays MCTS is a common tree-search method used while developing GGP artificial players and by combining Monte Carlo simulation with value and policy networks, Silver et al. [4] created an artificial player called AlphaGo that was able to achieve 99.8% win rate against other Go artificial players as well as defeating the human European Go champion in a match, a feat that made AlphaGo the first artificial player to beat a human in a full-size game of Go.

2.1.1 General Video Game Playing

The next logical step in the GGP approach was to expand beyond the usual board game types of games and into more varied types of video games, forming what is now known as General Video Game Playing (GVGP), as is exposed in the work of Levine et al. [5].

In its core the concept of GVGP is the application of the previous presented concepts of GGP into the video game world, which means the objective is to have artificial agents capable of playing video games they have never played before. The agent must understand how the game reacts to the actions it performs and how these actions affect the rewards it gets from performing said actions and ultimately understand how to beat and win the game. The agent should be provided at least with the current state of the world it is in and what actions it might perform, with the rest being up to the agent to decide. This allows the agent to be put in several different games, with unique stories, characters and goals across them with the only constant being the method the agent uses to interact with the current game it's playing.

2.1.2 Frameworks

In order to facilitate the development and testing of different game-playing artificial agents Levine et al. proposed to create a GVGP framework capable of recognizing 2D games written in Video Game Description Language (VGDL) to be used as various benchmark problems for the various game-playing agents [5]. This VGDL while not totally related to this research is described thoroughly in the work of Schaul [6] where he describes the language, its implementation as well as how a wide range of 2D games is generated.

2.1.2.A GVG-AI Framework

In the General Video Game AI (GVG-AI) framework, for most cases the input is provided in an object-oriented manner, where the current game state knowledge possessed by the agent is passed through an observation object that contains a list of observations, past history of events and current step counter.

With this framework as its base Levine et al. proposed to also create and hold a GVGP competition that ended becoming the annual GVG-AI Competition [5].

2.1.2.B Arcade Learning Environment

The GVGP framework was in part inspired in the work done by Bellemare et al. [7] where they developed a framework called Arcade Learning Environment (ALE) that provides an interface to a large number of Atari 2600 games [8](see Fig. 2.1 for example), but in the case of the artificial agents developed using the ALE framework, they are given as their input the game screen capture and the score counter and

Figure 2.1: PacMan and Atari 2600

from that they should be able to say which set of outputs (here it's a set of buttons corresponding to the Atari controller) determine the next action to execute in game.

Although the next section research is more focused on the GVG-AI framework and competition, which is focused in a more symbolic type of description of the games that passes to the artificial agents, it was important to make note of the way the Arcade Learning Environment framework processes its agents inputs through the use of screen capture, gathering for that the information from a framebuffer type approach, since it's the way the work of this thesis focused on passing the information to the artificial agent.

2.1.3 Breakout

From a wide range of possible testing games, the game that's used to test the solution proposed is "Breakout" which was released for the Atari console[1] [8].

"Breakout" begins with eight rows of bricks, with each two rows with a different color, with the color order beginning from the bottom to top as blue, green, yellow, orange and red. The game's screen is presented in Fig. 2.2. By using a single ball, the player must knock down as many bricks as possible, using the walls and/or the paddle controlled by the player as a way of ricocheting the ball against the bricks and eliminate them. If the paddle misses the ball's rebound, the player will lose a life. For each run, the player has three lifes to try to clear two screens of bricks. The yellow bricks earn one point each, the green bricks earn three points, the orange bricks earn five points and the the red bricks earn seven points each. The player controlled paddle shrinks to one-half its size after the ball has broken through

[1] Breakout, Atari Inc., 1976, Nolan Bushnell and Steve Bristow and Steve Wozniak, Atari 2600

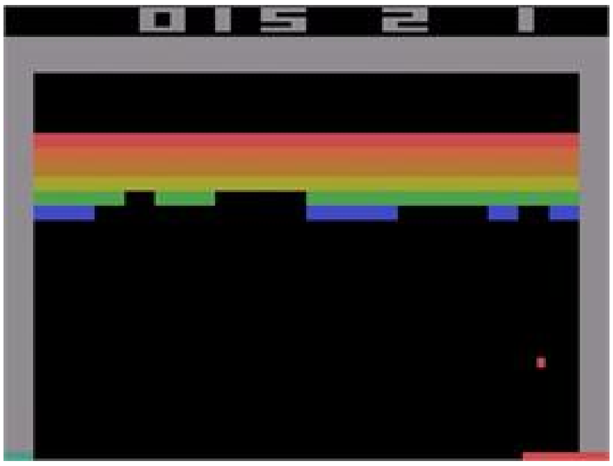

Figure 2.2: Breakout's game screen

the red row and hit the upper wall, with the ball speed increasing speed after four hits, twelve hits and after hitting an orange or red row. This was the chosen game for some reasons. The game ball has different trajectories and different speeds that could influence the way a player with physical limitations or with slower reaction interacts with it and affect their capability of achieving a better highscore after finishing the game. Another reason is attached to the fact that this game has been used in various implementations of Deep Learning and has a wide range of testing results available to cross check and compare.

2.2 General Video Game AI Competition

Set up in 2014 the GVG-AI Competition explored the problem of creating controllers for GVGP, as previously stated, and how developers could create a single agent that is capable of playing any game it is given [6]. Throughout the competition, created agents are run in a number of video games in order to observe if they show a type of general intelligence and behaviour that can be associated to a possible human player behaviour. Since its initial conception, the framework has been expanded in order to meet the demand of different research directions. So in its current iteration, as described by Perez-Liebana et al. [9], the competition covers 3 major branches:

- Agents capable of playing multiple unknown games with/without access to the game simulations;
- Agents capable of designing new game levels;
- Agents capable of generating game rules.

The Level and rule generation branch [10] focus heavily on the content creation problems and the traditionally used algorithms for this type of problem, being as well highly tied to the Procedural Content

Generation concept, which is not a concept we wanted to explore for this research and problem at hand. Instead, this works research focuses more in the corresponding branch of Agents capable of playing multiple unknown games with/without access to the game simulations, since it promotes research in model-free reinforcement learning techniques and neuroevolution.

Taking into account that the artificial agent this work plans on having doesn't have any kind of forward model that allows it to plan ahead its actions, the alternative was to look at algorithms that give the agent the capacity of learning games by repeatably playing them. Looking at the GVG-AI Competition description by Perez-Liebana et al. [9], there's a list of past years competition entries and some possible approaches that tackled the same type of challenge tackled in this work that based themselves in the use of neuroevolution and reinforcement learning techniques.

As a finishing note to this topic, it's crucial to emphasize that the objective of this work is in no way to create an artificial player capable of competing in this competition but it's nonetheless important to know in what basis was this competition built on, what have been the current proposed solutions and hypothesis that have tackled the problem of creating this type of artificial agents and how it helped selecting the kind of agent to be implemented and modeled as well as forming my solution.

2.3 Learning Approaches

In this section a series of learning algorithms fit for the problem are explored in order to replicate the artificial player of the work. Looking at the problem at hand, we want to replicate limited players, which led to us to discard the possibility of applying supervised learning methods to create the agent. That would require a type of preconceived data that to our knowledge doesn't exist or is not accurately dated. Taking this facts into account, and knowing that the agent should be learning from scratch and not from preconceived data, the focus is on Reinforcement Learning algorithms since the agent in the absence of data, must learn to achieve a goal in an uncertain, potentially complex environment.

Not all of the next approaches were taken into account while developing the work of the thesis, but were nonetheless important to note as state of the art and in tracing the path to the best possible solution.

2.3.1 Neuroevolution of Augmenting Topologies

Neuroevolution is an AI field that consists on combining an Artificial Neural Network with an evolutionary algorithm or more specifically achieve the evolution of neural networks using genetic algorithms. It is seen as a good way of creating artificial agents capable of simulating human behaviour.

The Neuroevolution of Augmenting Topologies (NEAT) concept consists of a genetic algorithm which generates efficient Artificial Neural Networks from a very simple starting network. This is the product of the research done by Stanley and Miikkulainen [11] in their entitled “Evolving Neural Networks through Augmenting Topologies” which will primarily be quoted and referenced in order to explain how the algorithm works.

First of all, in neuroevolution the Neural Networks are composed of interconnected processing units called neurons and the connection between these neurons are called synaptic weights. Each one of these neurons can be classified as an input, hidden or output node, all of which will vary depending on the task we want to assign them. The architecture of the Neural Networks is characterized by the structure of their neurons and connections, in what is called topology. Usually in neuroevolution the topology of the network is fixed, which means it fixes a topology and focuses only on evolving the strength of the connection weights, contrary to what Stanley and Miikkulainen [11] tried to achieve in their NEAT algorithm. Here they wanted to prove that evolving topologies along weights provide an advantage over evolving weight on a fixed-topology and that evolving structure along with connection weights can significantly enhance the performance of Neural Networks [11]. Looking at the results achieved they were able to suggest that the evolution of structure can be used to gain efficency over the evolution of fixed topologies.

According to the authors the NEAT method consists of solutions to three fundamental challenges in evolving Neural Network Topology: “Is there a genetic representation that allows disparate topologies to cross over in a meaningful way?”, “How can topological innovation that needs a few generations to be optimized be protected so that it does not disappear from the population prematurely?”, “How can topologies be minimized throughout evolution without the need for a specially contrived fitness function that measures complexity?”.

In terms of concrete applications of NEAT, a relevant work created by a former video games speed runner, consisting on an artificial player trained to learn how to play Super Mario World[2]. This artificial agent used an implementation of the NEAT algorithm using a graphical world representation as well as predetermined important game features as inputs[3]. During the numerous iterations and generations of the neural network it’s possible to see the evolution of the neural network’s topology while the algorithm selects the most fit generations. After a period of time that the artificial player spent training, it was able to beat a level of the game.

Some works approached the use of neuroevolution to GVGP, with NEAT being used as one of the neuroevolution approaches, using various types of state representation, just like in Hausknecht, et al. work [12]. Here they reached the conclusion that direct-encoding methods such as NEAT work best on low-dimensional, pre-processed object and compact state representations while indirect-encoding methods allow scaling to higher-dimensional representations such raw-pixel representations, suggesting that neuroevolution is a promissing approach to GVGP.

[2]Super Mario World, Nintendo, 1990, Shigeru Miyamoto, SNES

[3]Seth Bling, “MarI/O”, `https://www.youtube.com/watch?v=qv6UVOQ0F44`, `https://pastebin.com/ZZmSNaHX`

2.3.2 Convolutional Neural Networks

Since this work tries to emphasize the aspect of having the capability of recognizing the screen game as the input to the artificial agent, this research needed to find a way on how to extract information from what the artificial agent is able to see on the screen, which led to the concept of Convolutional Neural Network (CNN). In the realm of Neural Networks, CNNs are a type of Neural Network usually responsible for tasks like image recognition and classification (image data feature extraction). It was popularized and brought to attention in the work of Krizhevsky et al. [13] where they trained a large CNN in order to classify 1 million high-resolution images with smaller error rates than the competition they faced in the year of that work.

Here I will describe the architecture of CNN, based on an explanation by Cornelisse, Daphne[4] that it's focused on image recognition. The CNN image classifications receives an input image that it classifies under certain categories. This inputs are passed as arrays of pixels to the CNN and depend on the image input resolution. So the layers of a CNN are organized according to 3 dimensions: width, height and depth, which means that a neuron in one connection does not necessarily have to connect to every neuron contained in the next layer.

The hidden layers of a CNN are the component responsible for the convolution and pooling operations of a Deep Neural Network, which are performed in order to detect features followed by a classification process where fully connected layers serve as classifier for the previously extracted features as represented in Fig. 2.3.

First the convolution process is applied on the input by using a filter over that input, where at every area a matrix multiplication is executed and its sums are used to create a feature map. The filter is applied according to a stride size, which usually is 1, meaning that the filter is applied pixel by pixel. After executing multiple convolutions on the received input using different filters resulting in different feature maps, these are put together and form the convolution layer's final output. As a way of making the convolution layer output non-linear, it is passed through an activation function (in this case an ReLu function, f(x) = max(0,x), which is the commonly used). Since the resulting feature map is smaller than the input received, usually padding is used in order keep the spatial size constant, improve performance and to secure the filter and stride fit the input. It's usual to add a pooling layer after a convolution layer in order to continuously reduce the dimension, parameters and computation along the network resulting in smaller training times and preventing overfitting. For a visual representation of the various CNN layers see Fig. 2.3.

After these processes, the classification task is formed by a set of fully connected layers where their neurons are connected to all the previous layer neurons in order to, after training be able to classify the

[4]Cornelisse, Daphnee: "An intuitive guide to Convolutional Neural Networks", https://medium.freecodecamp.org/an-intuitive-guide-to-convolutional-neural-networks-260c2de0a050 (Last accessed 19 Dec 2018)

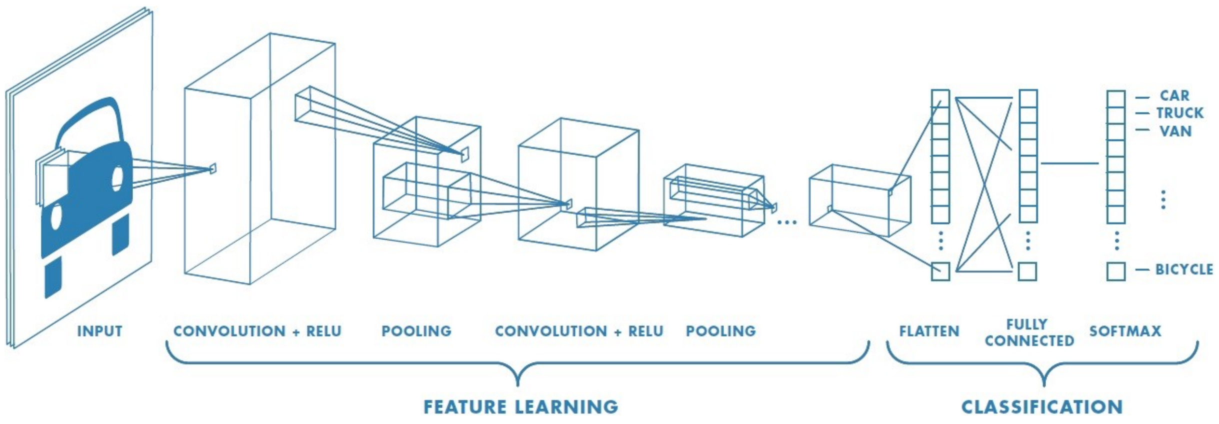

Figure 2.3: Convolutional Neural Network architecture, A Comprehensive Guide to Convolutional Neural Networks by Sumit Saha

outputs.

In terms of CNN applications, recently Gatys et al. [14] showed how a CNN can be used to create a picture that combines the content of one picture with the artistic style of another picture. As demonstrated by the authors, it could recreate pictures of real places using the style of a painter like Van Gogh. They used CNN in order to extract the content features from the picture to recreate and to extract the style features from the pictures they wanted to replicate the style.

In the context of GVGP, Kunanusont et al. [15] used CNN in order to extract important features from a game using screen capture in a work closely based and related to Mnih et al. [16, 17] work, which I will be describing more thoroughly in the context of another learning algorithm in the Deep Q Network subsection.

2.3.3 Deep Q-Network

Another take in the Deep and Reinforcement Learning fields is the Deep Q Network (DQN), which is the product of Mnih et al. [16, 17] work at a then called startup DeepMind[5]. Born out of a first idea by Lange and Riedmiller [18] that also merged deep learning and reinforcement learning in a visual input learning task.

In the Mnih et al. [16, 17] proposed model the neural network used is the previously described Convolution Neural Network, trained with a variant of the reinforcement learning technique Q-learning, in which the CNN is able to successfully learn control policies by receiving as input different Atari 2600 games screen's raw pixels from the Arcade Learning Environment [7], and return as output a q-value function estimating future rewards.

Taking into account that is often possible to learn better representations than handcrafted features

[5]DeepMind, `https://deepmind.com/`(Last accessed: 20 Dec 2018)

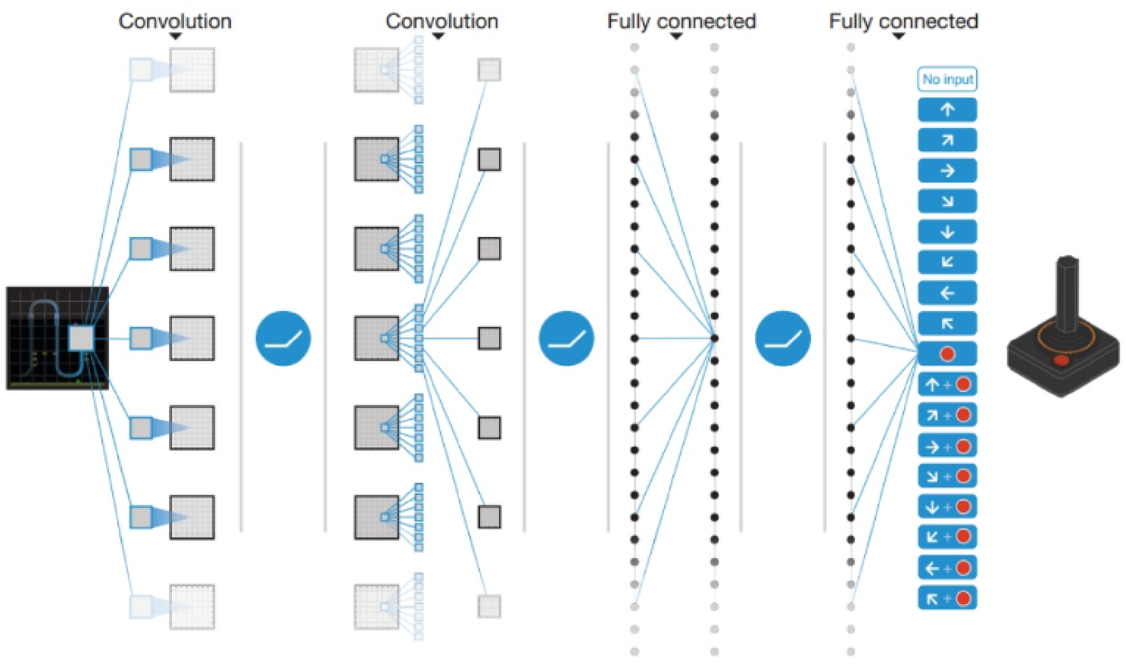

Figure 2.4: Deep Q-Network network architecture, Human-level control through deep reinforcement learning by Mnih, V., Kavukcuoglu, K., Silver, D. et al.

from previous works with CNN [13], Mnih et al. [16, 17] set their goal as to connect a reinforcement learning algorithm to a deep neural network which would operate directly on RGB images and efficiently process training data using stochastic gradient updates.

With Fig. 2.4 and the algorithm pseudocode 2.1 as reference we take a deeper look at the Deep Q-Network structure since this will approach will be used to develop our work.

The model applies to the raw Atari input frames some preprocessing methods in order to reduce its dimensionality, turning it less demanding from a computational aspect. The game frames are converted from a RGB representation to gray-scale and down-sampled. As a final result the input representation is given by cropping an 84x84 region of the image that captures a small size of the playing area.

The exact network structure as described by the authors consists of a first hidden layer which convolves 16 8×8 filters with stride 4 with the input image and applies a rectifier nonlinearity. As for the second hidden layer, it convolves 32 4×4 filters with stride 2, again followed by a rectifier nonlinearity. The final hidden layer is a fully-connected layer and consists of 256 rectifier units. The output layer is a fully-connected linear layer with a single output for each valid action [16, 17]. This outputs have the corresponding action predicted Q-values for the input state.

In their approach, they use a technique called experience replay, where all the observation and information is packed together in what is called “experience” at each time-step, to be stored in an experience

Algorithm 2.1: Deep Q-Network, Playing Atari with Deep Reinforcement Learning by Mnih

```
Initialize replay memory $D$ to capacity $N$
Initialize action-value function $Q$ with random weights $\theta$
Initialize target action-value function $Q_t$ with weights $\theta^-$ = $\theta$
for episode = $1, M$ do
    Initialize target action-value function $Q_t$ with weights $\theta^-$ = $\theta$
    for $t = 1, T$ do
        With probability $\epsilon$ select a random action $a_t$
        otherwise select $a_t = argmax_a Q(\phi(s1), a; \theta)$
        Execute action $a_t$ in emulator and observe reward $r_t$ and image $x_{t+1}$
        Set $s_{t+1} = s_t, a_t, x_{t+1}$ and preprocess $\phi_{t+1} = \phi(s_{t+1})$
        Store transition $(\phi_t, a_t, r_t, \phi_{t+1})$ in $D$
        Sample random minibatch of transitions $(\phi_j, a_j, r_j, \phi_{j+1})$ from $D$
        if episode terminates at step j+1 then
            $y_j = r_j$
        else
            $y_j$ = $r_j + \gamma max_{a'} \ Q_t(\phi_{j+1}, a'; \theta^-)$

        Perform a gradient descent step on $(y_j$ - $Q(\phi_j, a_j; \theta))$ with respect to the network
          parameters $\theta$
        Every $C$ steps reset $Q_t = Q$
```

pool. Some drawn at random experiences in the pool are sampled to update the network using Q-learning according to the last 4 frames of the selected experience, in what is called “experience replay”.

Knowing that in reinforcement learning successive states are similar, there’s a risk of the network forgetting what a state is like in which it has been in some time. By applying replay experience the network will still be able to access past frames. Although the replay is a good way of accessing past experiences it is limited to the last N experiences and samples uniformly at random while updating, which in some aspects is limiting since the memory is unable to form differentiating important transitions and needs to constantly be overwritten with newer experiences.

After the “experience replay” process the agent selects an action taking into account an E-greedy policy, which means that sometimes it picks actions randomly in a way that the model can learn about an action it doesn’t think is optimal, and sometimes pick actions according to the model so that it moves forward in the game and learns different states.

Recently Kunanusont et al. [15] tried to create a screen learning agent for GVG-AI framework based on what the work of Mnih et al. [16, 17] achieved. Here they applied the Deep Q-Network approach as well, with some improvements achieving results that suggested that their proposed screen capture learning agent had the potential to learn many different games using only a single learning algorithm. Also a look around some websites related to deep learning shows some works like an article where they instruct on how to create an agent that learns to play Doom[6][7] using the same principles referred before.

2.3.4 A3C

Another reinforcement learning algorithm developed by DeepMind's Mnih et al. [19], Asynchronous Advantage Actor-Critic (A3C) has quickly became the go to algorithm for new challenging problems with complex state and action spaces. It is faster and simpler compared to DQN and achieves better scores on the standard Deep Reinforcement Learning tasks.

The algorithm can be separated into 3 major components, that are called the 3 As of A3C which are represented in Fig. 2.5 architecture.

Actor-Critic

Similar to how the DQN works, the structure of the A3C algorithm presents a major change. In a regular DQN, there would only be one output which corresponded to the q-values of the possible different actions. In A3C case, there are two outputs that combine the benefit of a value-iteration method and policy-iteration methods. The network estimates both a value function V(s), how good a certain state is to be in and a policy $\pi(s)$ with set of action probability outputs.

The agent uses the value estimate (the critic section) in order to update its policy (the actor section) in a better way comparing to traditional policy gradient methods.

Asynchronous

Unlike DQN, where a single agent is represented by a single neural network interacting with a single environment, A3C makes use of multiple agents with each agent having its own network parameters and a copy of the environment.

In A3C there is a global network, and multiple worker agents which each have their own set of network parameters. Each agent interacts with its own copy of the environment at the same time as the other agents are interacting with their environments.

This approach performs better than a single agent because the experience of each separate agent is independent of the others, which means each agent has a more diversified training data contributing to an overall diverser experience.

[6]Simonini, Thomas: "An introduction to Deep Q-Learning: Let's play Doom". https://medium.freecodecamp.org/an-introduction-to-deep-q-learning-lets-play-doom-54d02d8017d8 (Last accessed: 20 Dec 2018)

[7]Doom, id Software,1993, Tom Hall and John Romero and Sandy Peterson and American Mcgee and Shawn Green

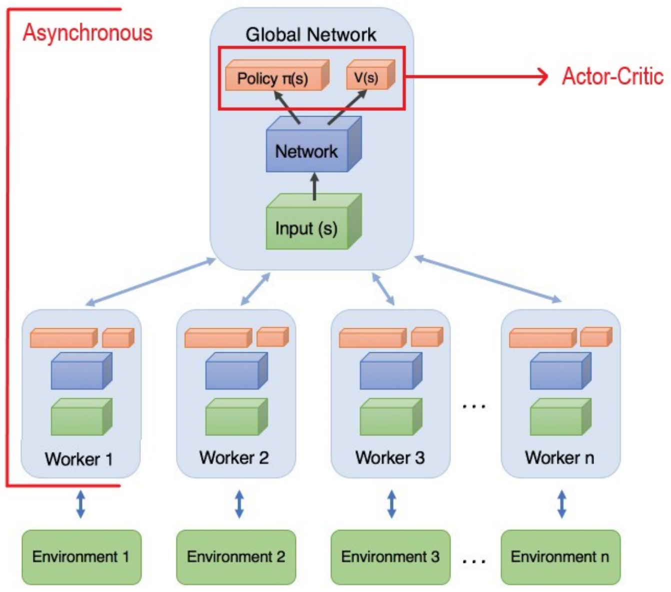

Figure 2.5: Diagram of A3C architecture

Advantage

The algorithm wants to figure out the difference between a taken action Q-value and the value of the state the agent is in. Simplifying it wants to know how much better is the q-value in relation to the known value. The point of having this calculation is to always get better q-values in order to obtain increased rewards. According to this calculations, if the advantage is high the Neural Network enforces this behaviour and updates its weights in order to keep repeating these same actions, in case they're lower it tries to prevent these actions form occurring again.

2.3.5 Proximal Policy Optimization

Obtaining good results using policy gradient methods is challenging due to their sensitive choice of step size (too small might turn the progress extremely slow and too large might cause performance issues and noise) and often poor sample efficiency, leading to the execution of millions of steps in order to learn simple tasks.

The OpenAI team of Schulman et al. proposed a new policy gradient methods for reinforcement learning which alternate between sampling data through interaction with the environment, and optimizing an objective function using stochastic gradient ascent [20]. The proposition included an objective function that enables multiple iterations of small batch updates. To this methods they named it Proximal Policy Optimization (PPO).

According to the authors blog on the OpenAI website [8], PPO strikes a balance between ease of implementation, sample complexity, and ease of tuning, trying to compute an update at each step that minimizes the cost function while ensuring the deviation from the previous policy is relatively small. This way there's less discrepancy in training at the cost of some bias, but a smoother training.

PPO achieves the data efficiency and reliable performance of TRPO [21], using only first-order optimization. Schulman et al. proposed a novel objective with clipped probability ratios, which form a pessimistic estimate of the policy performance. In order to optimize, it alternates between sampling data from the policy and performing several iterations of optimization on the sampled data.

When applying PPO on the network architecture with shared parameters for both policy (actor) and value (critic) functions, in addition to the clipped reward, the objective function is augmented with an error term on the value estimation and an entropy term to encourage sufficient exploration.

The algorithm has been tested on a set of benchmark tasks and also on the Atari domain against other well-tuned implementations of policy gradient methods. It proved to have better overall performance according to the authors.

2.4 Discussion

Looking at this section, a varied series of relevant method based approaches to the paradigm of artificial intelligence being able to play games without having prior knowledge about them and a series of deep learning algorithms were presented, allowing for a wider look at the field of research applied.

Taking into account the algorithms presented as well as their pros and cons, one of them was to be selected as the basis for this thesis model and work. Knowing that the NeuroEvolution approach wasn't really viable, due to the necessity of having a screen capture processing input method which is not present on the explored NeuroEvolution, the choice was made to have the DQN algorithm as the base going forward in detriment of the other options. Although DQN might not be the most friendly at a computational weight level, but the wide variety of implementations already document and implemented as well as the fact that the use of multi-threading needed to implement the other algorithms ended up being a problem to implement and support, the decision was made to move forward with the DQN.

With this in mind, the implementation of the DQN, the results and the proposed model will be described in the following section.

[8]https://openai.com/blog/openai-baselines-ppo/ (last accessed 13 Oct 2019)

3 Implementation

Contents

In this chapter, we will approach the base on which our solution to the problem presented in the Introduction chapter is rested. We will start by presenting the technologies that were used in order to replicate the reinforcement learning algorithm used. Then, we present the results achieved by training and testing the network in our environment of choice as well as comparing the results with the documented results on the Mnih et al. [16, 17] work papers.

3.1 Algorithm Specifications

The DQN agent implementation was provided on the keras-rl library and it follows the agent described in the work of Mnih et al [16, 17]. As previously discussed, the game used for testing the agent implementation was Breakout and was replicated using the gym environment 'BreakoutDeterministic-v4'. The gym tool offers a wide range of environments and the one stated has some particularities. The deterministic in the name stands for a fixed frameskip of 4 frames, which means that the agent only train every 4 game frames and the v4 for a 0 probability of repeating the previously selected action on the current state (v0 has a 0.25 probability of repeating the previous selected action no matter what action was selected for the new state). And with that, the frame skipping described in the paper is already implemented as part of the environment.

The frames are extracted from the framebuffer and processed according to the same procedure present in the Mnih paper. First the frames RGB representation is converted to gray-scale and down-sampled, after which the final input representation is obtained by cropping an 84×84 image that captures the playing area. As they explained this is done in order to allow the use of GPU implemented 2D convolutions.

A crucial part of the DQN agent rests on its network, and following the model description of Mnih the implemented network is composed by 5 layers. It all starts with the 84x84x4 image that serves as the input, followed by 3 hidden convolutional layers a hidden fully connected layer and a final fully connected layer with a single output for each valid action. All of this layers are followed by a rectifier nonlinearity as previously discussed. More specifically the first layer convolves 32 filters of 8x8 with sride 4, the second convolves 64 filters of 4x4 with sride 2, the third convolves 64 filters of 3x3 with sride 1 and the first fully connected layer consists of 512 rectifier units.

Since we replicated the Breakout game, the number of valid actions is set to 4 ['no-op', 'fire', 'right', 'left']. The crucial ones are the 'no-op' action (referred to from here on out as 'Don't Move') that means the slider does not move and 'left', 'right' which move the slider in each direction. The 'fire' action can be seen has doing the same as the 'Don't Move' action but it does carry the function of restarting the game when the player loses a life.

The memory used to stash the experiences collected through the training is a sequential one. While the experiences are stored in the experience replay in a sequential fashion, when they are used to train the network, random sampling is perform to extract minibatches. In our training case the replay memory has a maximum of one million most recent frames stored and each minibatch has fifty thousand frames.

Although the agent trained in the Mnih et al. paper was trained for a total sum of 5M steps, in our case we settled for 1.75M training steps. It is not the best training time but is capable of showing results and allowed a reduction on time spent training multiple iterations of different networks.

In terms of policy, we use eps-greedy action selection. For the policy, a random action is selected according to probability eps and over the course of one million steps the eps slowly decreases from 1.0 to 0.1 in order for the agent to first explore the environment, due to high eps, and gradually for it to commit to what he recognizes.

3.2 Replicating Results

In order to guarantee the accuracy of the implementation and results, we needed to run the code on our setup, so we could cross check it against the results and data presented in the Mnih, et al paper.

For this we performed a series of tests that allowed us to check how similar our results are. We trained the network following the previous declared specifications and from that we extracted the game high scores, the reward per episode evolution and the mean q value through the network training process and also an average of the actions performed.

Looking at Fig. 3.1, we have the average Q and average reward for the Mnih et al paper, with the values ordered by training epochs. In the paper each epoch corresponds to 50000 steps of training(due to minibatch size), so we looked to limit our view of this data to the max of 40 epochs (2M steps), since we trained our network for only around 1.75M steps. Each episode corresponds to a run of the game, which includes playing until the player as no more lifes and has to restart. The reward function returns increasingly better results according to the actions that lead to a change in the games score. Looking at the data, in order to have a accurate depiction of a DQN network we should have an average Q value of around 2.5 and an average reward per episode of over 60 for 1.75M training steps.

Now looking at the data we got from training our network the results are in part similar to what was expected according to the paper data. The average Q value evolves in a similar fashion to the corresponding paper data, while for the average reward per episode case the results are not always similar. Initially the reward progression is in line with the papers values but, although that for the 35 training epoch mark of the paper it had a somewhat close value (around 60 to ours 51 average), it's possible to see that it also oscillates around higher values at this point.

The difference in values might be due to different software versions and hardware configurations since the progression of values seems similar, therefor we will use our obtained results as our baseline

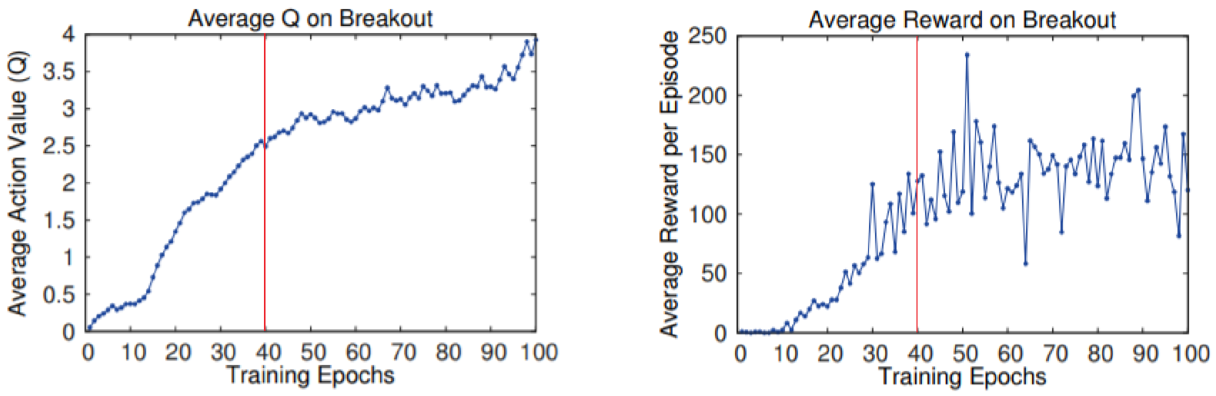

Figure 3.1: Average reward and q value from Mnih et al. paper

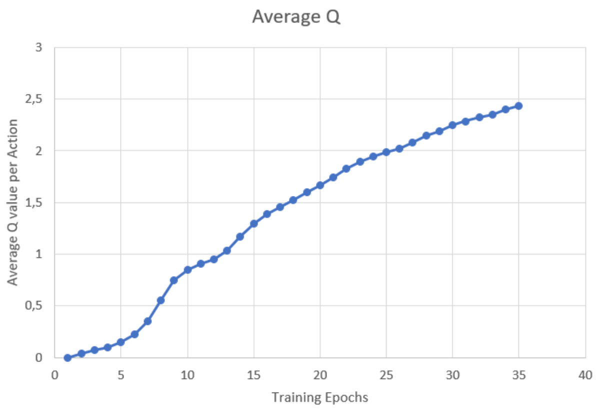

Figure 3.2: Average Q for 1.75M steps training

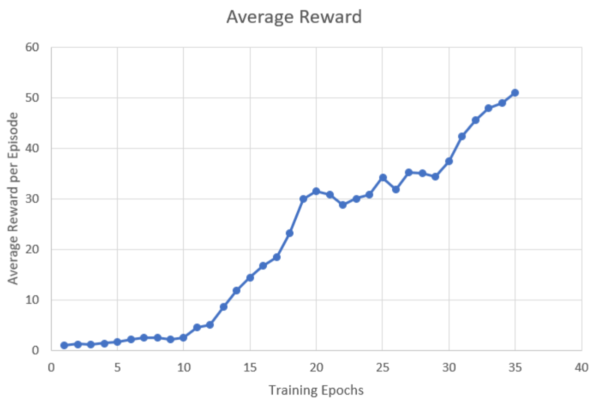

Figure 3.3: Average reward for 1.75M steps training

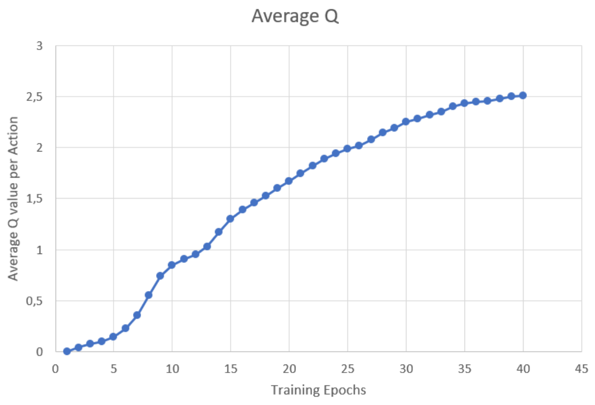

Figure 3.4: Average Q for 2M steps training

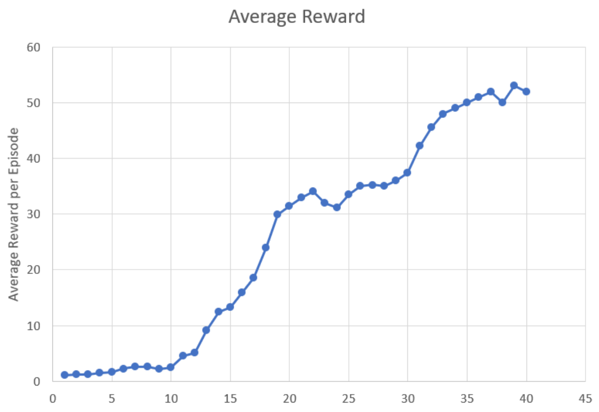

Figure 3.5: Average reward for 2M steps training

moving forward.

As a final note, in order to reduce the time spent training all the networks, we tested the 1.75M training step network head to head with a 2M training step network to conclude if the difference between training steps would produce significant impacts on the final rewards and high scores. Looking at Figures 3.2,

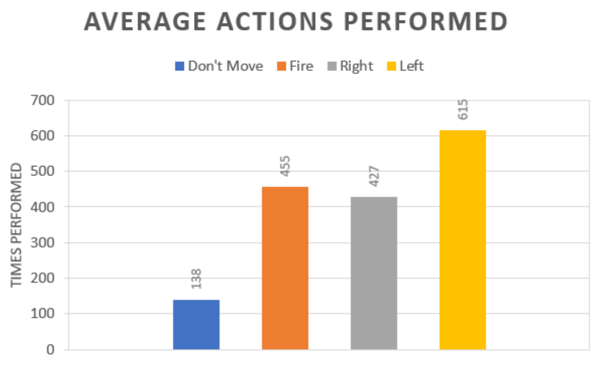

Figure 3.6: Actions Performed for Standard Training

3.3, 3.4 and 3.5 we can conclude that the difference between training steps isn't significant that would require testing with the bigger steps and so we decided to stick to 1.75M steps.

This way we are able to affirm that the trained network is able to in part replicate the paper results and we are able to propose our solution on top of this DQN implementation and our baseline results.

3.2.1 Additional Data

We now present some additional data for future reference.

First of all, a graph that represents the high score evolution throughout the training process, as a way of showing that after a certain amount of training steps (0.75M), exists a less steep evolution on the agents capability of reaching higher scores. This is also in line with the slower increase of the average reward per episode showcased in Fig. 3.3. Next, an average of the actions performed by the agent after the training has concluded (see Fig. 3.6), offering an understanding of the agents preferred actions in order to achieve the best rewards and high scores. The action 'Don't Move' and 'Fire' will a lot of times be agglomerated on the same category of action, since they have the same effect on the environment most of the time.

Moving forward, the previous data collected is the base to which we pretend to compare and analyze the next trained agents, following our solution model.

3.3 Solution Model

After forming the baseline to our work, we were able to take the algorithm implementation and apply our hypothesis on top of it

Following the DQN architecture[1] by Mnih, we propose the model present on Fig.3.7. Here we see

[1]DQN Lecture, https://drive.google.com/file/d/0BxXI_RttTZAhVUhpbDhiSUFFNjg/view, Last accessed: 27 October 2019

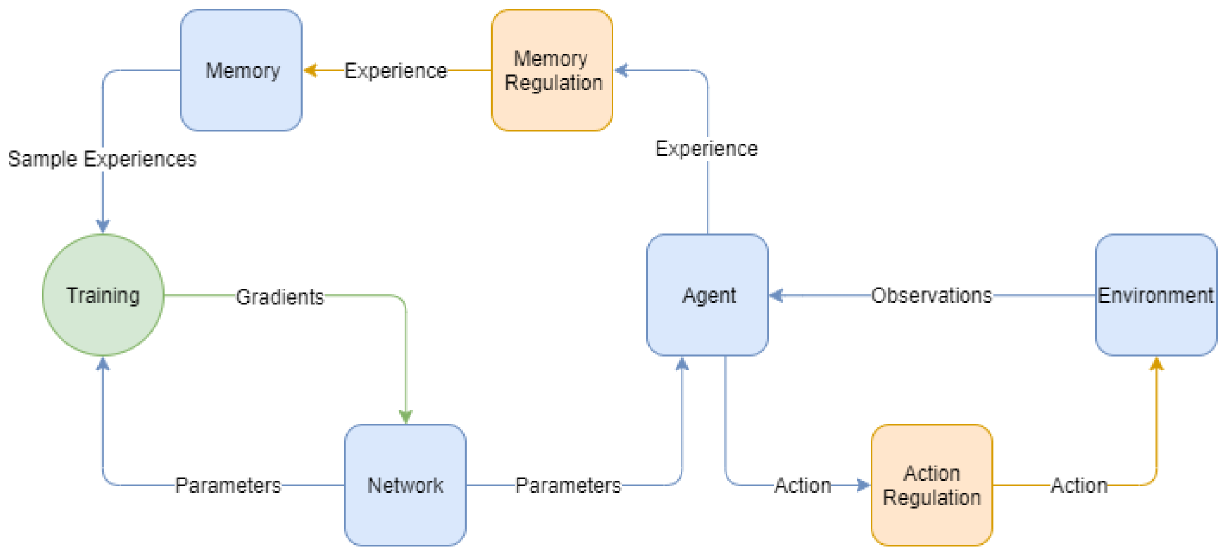

Figure 3.7: Solution Model Architecture

2 major differences in comparison to the original architecture. The original model elements are represented in green and blue, while our model is represented by the orange states.

The blue and green states represent the procedures a normal DQN implementation would take. And so, on top of that model, first we have a state where we treat the action that the network passes to be executed on the environment. This means we are able to regulate the action how we deem fit to simulate some types of limitations we want to enforce on the agent. This action is previously selected by the agent, and might suffer some type of change when passed through our state (for example, a moving action becoming a non moving action).

Next is a state that regulates the storage of experiences on Memory, limiting the set of experiences the agent can save and sample through it's experience replay. We wanted to limit what the agent would remember from executing different actions for different states, limiting what it counts as training information.

With this model our intention was to try and model limitations in a way we formulated our hypothesis, focusing on simulating errors on the execution of certain types of actions, applying delays on the execution of actions and force a possible visual misses of certain states. These limitations, procedures and results are further explained in the next chapter.

With this model we hope to comprehend how these limitations can influence the network we are training, how does it behave and what kind of results it achieves in comparison to the original network and between different limited networks.

3.4 Technology

The development language used for the replication of the algorithm and further implementations and tests was Python. In order to simulate the Atari and game environment chosen to replicate, we used a Python library called Gym[2] developed by the OpenAI company. Gym is a toolkit for developing and comparing reinforcement learning algorithms and capable of simulating large numbers of reinforcement learning environments, including Atari games as we wanted for the solution. In order to replicate the Neural Network of the reinforcement learning algorithm, we used the Keras API library[3] with the Tensorflow GPU implementation[4][5] as its base.

The algorithm implementation is provided by Plappert, Matthias Keras-RL [22]. This library implements some state-of-the art deep reinforcement learning algorithms in Python and is integrated with Keras, working with OpenAI Gym, providing easy evaluation and tinkering of different algorithms.

[2]Gym, https://gym.openai.com/
[3]Keras, https://keras.io/
[4]Tensorflow, https://www.tensorflow.org/
[5]Tensorflow GPU, https://www.tensorflow.org/install/gpu

4 Results and Analysis

Contents

Following the model structure proposed, in this chapter we present the results of the various types of manipulations deemed fit to test, with multiple iterations between themselves. All the networks trained and presented from here on further were trained following the same basis as the original network, albeit following our model structure for specific situations. Each network was trained 2 times, due to time constraints. The first and second set of test results use only part of our model, more specifically the action regulation module while the third set of test results takes full use of our model disposition, engulfing the action and memory regulation steps.

We display the graphs displaying the training actions performed to understand the trained agents patterns of play, as well as the values of rewards and highscores for each of the training executed. We decided to showcase the both since highscore simply shows the final score obtained while the reward represents the total reward the agent achieved per episode of play. They are related but the values might differ, although smaller rewards tend to have the same value as the highscore.

4.1 Action Change

Here we have a set of action limiting implementations that we base on the principle of having an error probability associated with the execution of a certain type of action. This way, we can formulate an idea of what happens when the action intent doesn't go according to plan. This is a way of either simulating the execution and try of last second changed actions, the limitation of performing an action, the execution of an action contrary to what it was supposed to happen or at a more simple level, a possible problem associated with hardware malfunction.

In order to better understand the impact these changes carry, we apply different degrees of error probability throughout the networks testing. The error percentages were set at 10%, 30% and 50%.

In this section we split the action change study into 3 main subsections. First, the possibility of an action when chosen reversing to the opposite one following the error probability set for the testing. Next, fruit of curiosity and to see how it would perform against the prospect of both actions reversing, we simply test the possibility of only one of the actions reversing (in this case the 'Right' action) according to the error probability. To finish off, a set of training networks for testing what happens when both actions have a probability of becoming the action 'Don't Move'.

4.1.1 Parametrization

For the following training procedures, it was guaranteed that all the iterations of the training had the same values and errors to ensure the results were coherent between themselves.

So in the first scenario (Both Actions Reverse) after calculating the action that would lead to the best reward according to the network, if it happened to be an action that resulted in movement ('Right','Left') it would therefor be subjected to another decision step process. According to the error probability in play, the action was reversed to its counter-part nullifying the action initially calculated in favour of its reverse. In the cases the action wasn't reversed the training process resumed as normally according to the algorithm in play.

Following a procedure identical to the first scenario in One Action Reverses scenario case, if the action calculated results in the action movement 'Right' it goes through the error probability phase previously described and is subject to a reversal if the error probability deems it so.

Finishing the scenarios, is the training according to Both Actions become 'Don't Move' philosophy, where in case the action calculated is either 'Right' or 'Left' it has a chance of becoming a 'Don't Move' action after calculating. Note that it was decided to set the actions to 'Don't Move' and not 'Fire'. Although they are in most part the same thing (both have the same effect on the players paddle), the 'Fire' action is used to restart the game every time the ball gets past the paddle.

4.1.2 Both Actions Reverse

Table 4.1: Both Actions Reverse Trained Agent Results

Limitation	Value	Reward	NºSteps	Highscore	Don't Move	Fire	Right	Left
Standard	-	51	1646	105	541	124	487	494
Both Actions Reverse	10%	37	1300	67	297	97	250	656
Both Actions Reverse	30%	31	1143	46	0	359	270	514
Both Actions Reverse	50%	22	846	28	69	78	239	460

Results

Looking at the data obtained training this agent, it's possible to see a clear worse performance, with the reward and highscore(see Table 4.1 and Figures 4.1 and 4.2) values achieved by the agent dropping when compared to the original implementation results. Subsequently the number of steps the agent is able to take during an episode of a game is also lower, since it loses quicker.

When observing the progression it has for different error probability values, it's possible to see that the drop off in values between the 10% and 50% error probability (37-22) is as accentuated as it was between the standard agent and with the 10% error probability (51 to 37 reward) with the same pattern standing for the highscores (105-67 and 67-28).

In terms of actions performed by the agent(Fig. 4.3) we can state that the 10% action distribution resembles a lot the standard action distribution, having a good balance between non moving and moving actions, albeit with more focus on the 'Left' action. As the error probability increases we see a number of notable changes to the actions performed. First of all starting at 30% the slight decrease of non moving

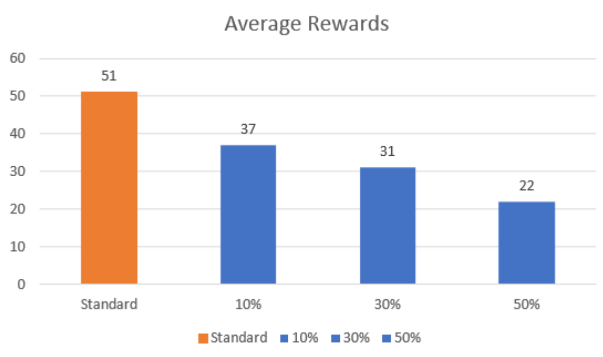

Figure 4.1: Average reward for Both Actions Reverse

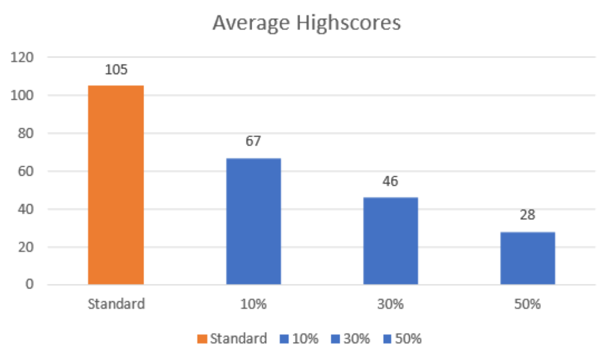

Figure 4.2: Average highscore for Both Actions Reverse

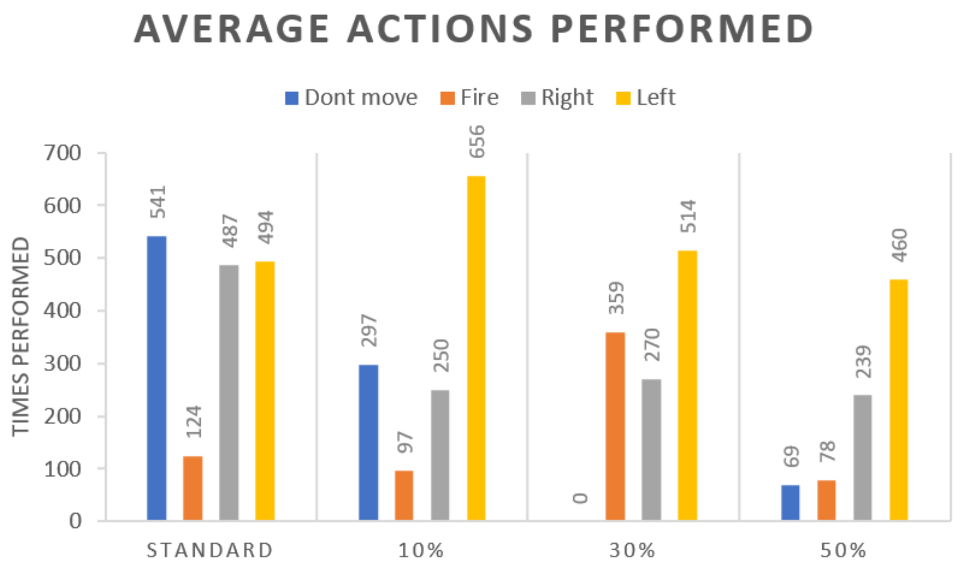

Figure 4.3: Action Distribution for Both actions reverse Training

actions further highly accentuated by the 50% mark, where they are almost non existent in comparison to the moving actions. Worth highlighting the decrease in the difference between the 'Right' and 'Left' action from 10% to 30% but not from 30% to 50% where the proportions are maintain.

Analyses

Based on the results obtained we know that the discrepancy between the standard values and this scenario training values is quite significant, which leads us to believe that the unpredictability and uncertainty associated with reversing both actions provides the agent a big hurdle to overcome when trying to train for the best possible outcomes. Although the values indicate the agents worse capability to obtain better results in this conditions, it is possible to see how he tries to adapt to those limitations.

If we look at the actions the agent performs in Fig. 4.3, it's possible to state that despite the number of steps the agent is capable of taking per episode is reduced, it still is able to maintain a similar proportion of moving actions as in the standard and lower error probability iterations. He does that at the expense of performing non moving actions and although it isn't able to achieve great results, we can see that it tried to adapt to it's imposed limitations.

To finish off, the action 'Left' sees a decrease in the number of times it was executed. Knowing that both actions can reverse and that one of the actions maintained its proportions while the other decrease, leads us to believe that the agent tried to execute more 'Right' actions knowing they would have an higher chance of reversing, turning them into 'Left' actions.

4.1.3 One Action Reverse

Table 4.2: One Action Reverse Trained Agent Results

Limitation	Value	Reward	NºSteps	Highscore	Don't Move	Fire	Right	Left
Standard	-	51	1646	105	541	124	487	494
One Action Reverse	10%	48	1635	90	137	457	427	614
One Action Reverse	30%	47	1447	92	699	288	265	225
One Action Reverse	50%	44	1230	107	341	393	186	310

Results

For this scenario we obtained some interesting results. Firstly the training still achieves worse results in terms of reward values obtained comparing to the standard training, albeit these results represent a minimal drop off in comparison (see Table 4.2 and Fig. 4.4). Looking at the whole spectrum we can see that the reward progression throughout the increasing higher error probabilities displays only a drop of under 10 reward value. And looking at the high score values (see Fig. 4.5) it's possible to state that after initially dropping off in relation to the standard training, they progressively see an increase in value while increasing the error probabilities.

In terms of steps executed by the agent(see Fig. 4.6), the number of steps per episode doesn't see a big reduction between the standard training and the different reversion error probabilities, and thus it's no surprise that for the lowest of error probabilities the actions performed are still similar to the standard trained agent. As for the other error probabilities, it's visible a clear increase in the number of non moving actions performed at the expense of moving actions. The moving actions tend to also follow the same line of execution as the 10% error percentage one, although for the 30% case it seems to be more even.

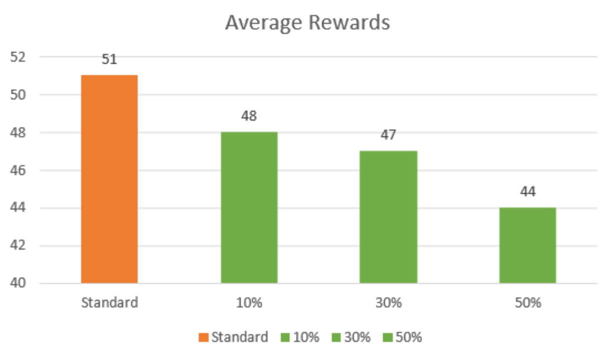

Figure 4.4: Average reward for One Action Reverse

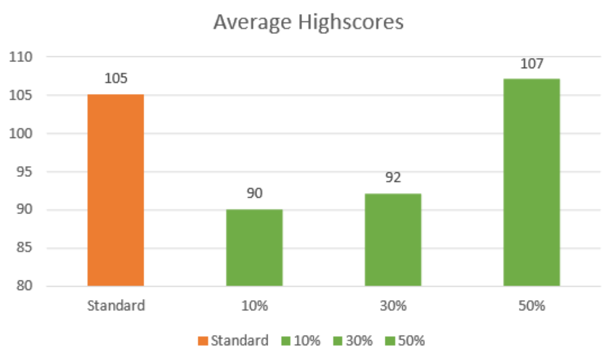

Figure 4.5: Average highscore for One Action Reverse

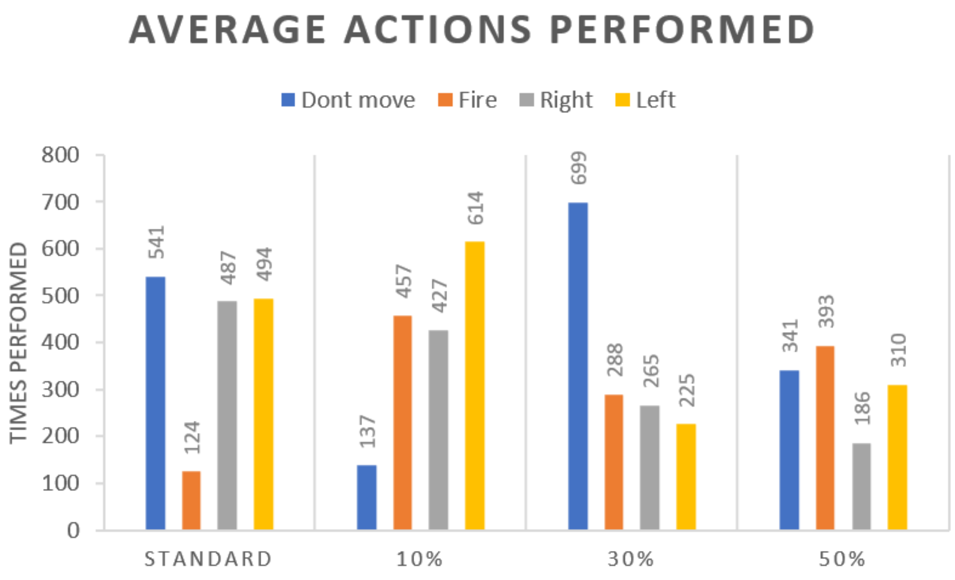

Figure 4.6: Action Distribution for One action reverse Training

Analyses

Taking into account the previous Change Both Actions scenario results we can indeed conclude that the agent is capable of better adapting to the limitations in play for this scenario where it is only required to reverse one action. The unpredictability is still there but in smaller quantities, with the fact that exists only one action capable of being reversed, making the decision process less uncertain.

As stated before, the action distribution for 10% seems to be identical to the previous tests, seemingly indicating that the uncertainty to this point caused on the agent the same type of reaction as it did in previous tests. After that, the pattern the agent seems to apply is different when compared to before. The agent clearly shows signs of preference towards non moving actions, indicating that it would rather have the paddle stay in the same position and only act on situations it would be more certain of the outcome and trying to reduce the possible error while applying the 'Right' action.

Although only one type of action can reverse scenario might not be as important to replicate real human limitations, it can still represent some kind of impact at the game design level, for example by having one action execution more facilitated than the other. Seeing as the highscore obtained for 50% error probability is higher than standard values, it might be interesting to explore a game with this types of characteristics.

The scenario also can be used as a benchmark as to how much impact the both actions reversing has on an player. In certain situations a game designer might want to know how an agent can respond when a limitation is applied to simply one action of its game, and how influential it might be on the gameplay.

4.1.4 Both Actions become 'Don't Move'

Table 4.3: Both Actions Become 'Don't Move' Trained Agent Results

Limitation	Value	Reward	NºSteps	Highscore	Don't Move	Fire	Right	Left
Standard	-	51	1646	105	541	124	487	494
Both become 'Don't Move'	10%	45	1506	84	85	96	628	697
Both become 'Don't Move'	30%	41	1517	77	505	112	430	470
Both become 'Don't Move'	50%	41	1476	71	445	85	495	451

Results

Finishing off, the results here are in line with the previous tests, showing off decreasing values for average reward and high scores(see Table 4.3 and Figures 4.7 and 4.8) but more similar to the One Action reversing scenario instead of Both Actions reversing since the drop off is not very steep. More

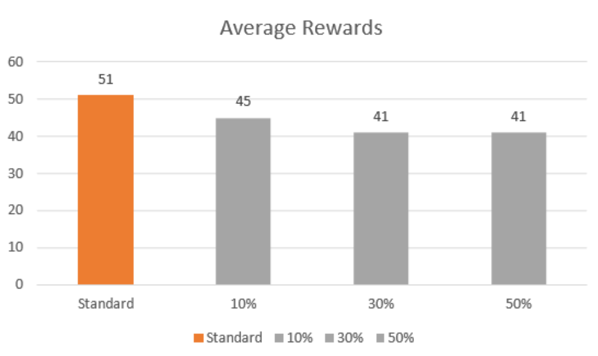

Figure 4.7: Average reward for Both Actions Becoming 'Don't Move'

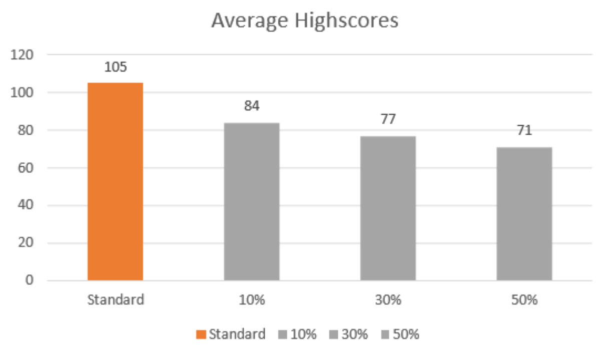

Figure 4.8: Average highscore for Both Actions Becoming 'Don't Move'

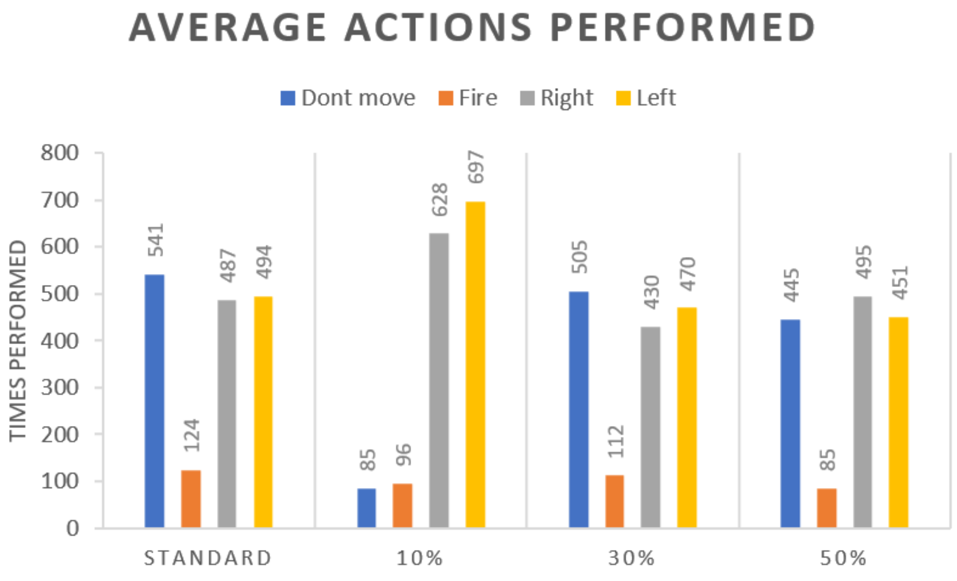

Figure 4.9: Action Distribution for Both actions become 'Don't Move' Training

specifically, in terms of highscore it doesn't have the increase we saw happen in the previous test, settling for a steady decrease in value.

The number of steps executed by the training agent are very consistent between error probabilities and very close to the standard values.

In terms of action distribution (Fig. 4.9), for 10% error they don't seem to be as in line with previous tests as they were before. Here we see a high abundance in moving actions, with very little execution of non moving actions. From there on out, the executed actions paradigm changes a bit as the moving actions decrease in terms of executions, while the non moving actions ('Don't Move' mainly) increase.

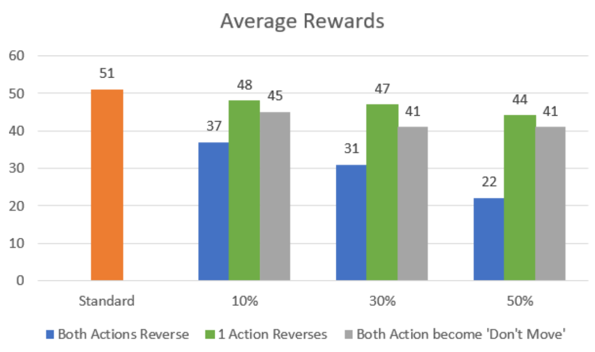

Figure 4.10: Reward distribution for 3 scenarios

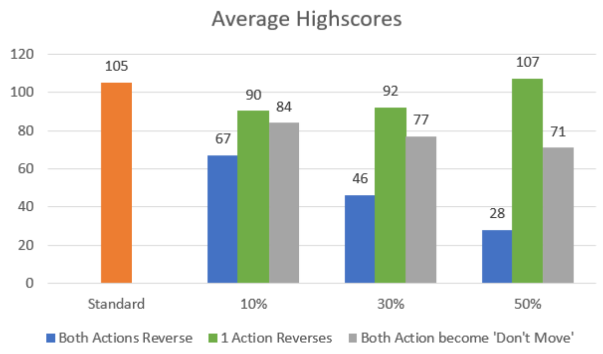

Figure 4.11: Highscore distribution for 3 scenarios

Analyses

As happened in the training scenario before, for these conditions the agent was able to adapt in similar fashion to the limitations in play. The maintenance of close to standard numbers of steps per episode is a clear evidence of that, albeit the rewards and high scores are not up to par with the standard iteration.

The actions executed for the 10% error probability are intriguing. Here we have a high volume of moving actions whereas the non moving actions represent a small percentage of the total actions executed, when we expected the contrary to happen. If we set our attention onto the other percentages we might have an idea as to why this is the case. For the higher percentages, it's still visible a high percentage of moving actions in play, albeit a decrease in comparison to the 10% training. The speculation is that the increase in volume of non movement actions and the conservation of the execution of high number of moving actions is due to the fact that the agent executes a higher number of actions that lead to movement in order to counter the existence of a probability of changing the actions to 'Don't Move'.

Following this logic, it's possible to assume that the agent was already putting in practice this counter measure on the 10% error probability training.

4.1.5 Final Notes

Finishing this section off, we have a representation of all of the rewards and highscores on the same graphs to show which have the supposedly best results (Figures 4.10 and 4.11). As previously stated the One Action reverse limitation trained agent seems to achieved a similar level of results to the ones of the Standard baseline, following a different pattern of play. From these trained agents we are already able to understand that it's possible to obtain different patterns of play for different iterations of the same limitation range.

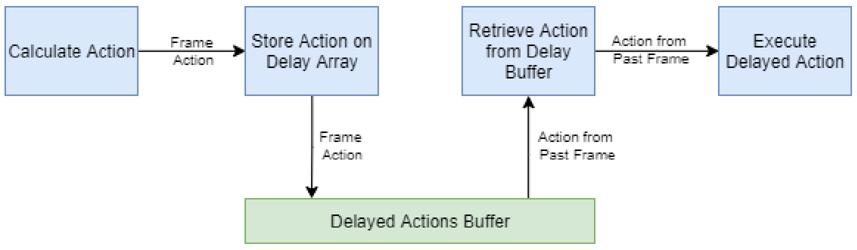

Figure 4.12: Delay Process for each Frame

4.2 Action Delay

For this type of limitation, we hoped to replicate a delay in the execution of an action by the player. This might be result of slower reaction times on players or movement deficiencies. For that, every state we calculate which action is to be executed on the environment and store it onto an array of delayed actions. From then on the delayed actions stored are to be executed in future frames of the game according to the delay we define beforehand. On Fig. 4.13 we display a visual guide of the process that happens on every frame for processing the current calculated actions and the delayed actions.

4.2.1 Parametrization

As with all the scenarios tested before, it's ensured that all of the training processes have the same base values. In this delay scenario, the action selection is affected by a delay value set by the user. The delay values range from 1 to 5 and an extra delay of 8 frames. The delay values were selected in order to understand the effect a delay has between 2 training steps (1 to 4 delayed frames, 5 and 8 delayed frames).

Table 4.4: Action Delay Trained Agent Results

Limitation	Value	Reward	NºSteps	Highscore	Don't Move	Fire	Right	Left
Standard	-	51	1646	105	541	124	487	494
Frame Delay	1	27	991	39	318	205	211	257
Frame Delay	2	19	720	28	194	18	86	422
Frame Delay	3	13	627	16	17	45	311	254
Frame Delay	4	25	958	37	138	477	123	220
Frame Delay	5	19	757	25	280	119	148	210
Frame Delay	8	15	580	27	225	6	83	266

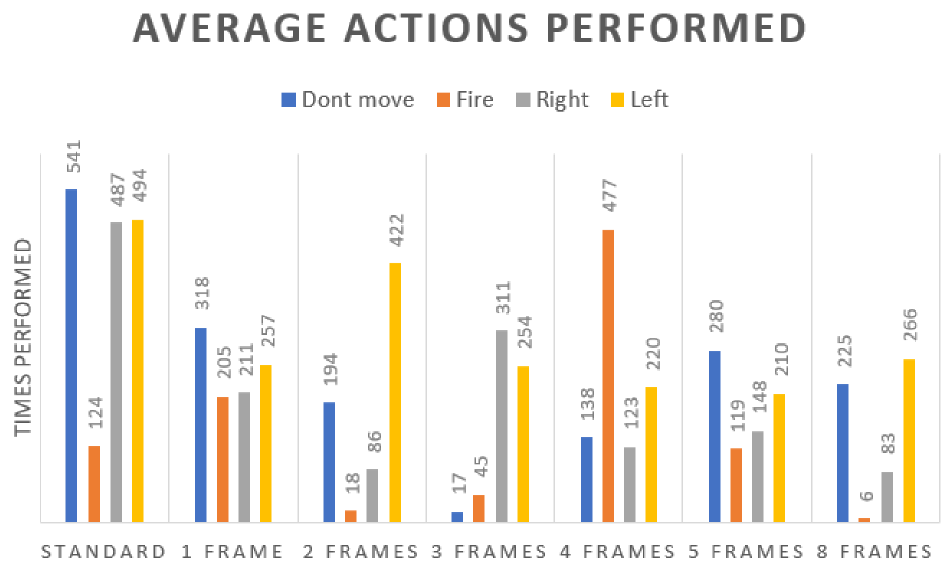

Figure 4.13: Action Distribution for various delay training

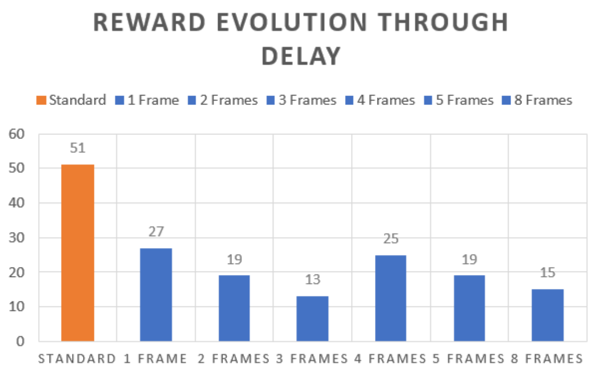

Figure 4.14: Average reward for delay training

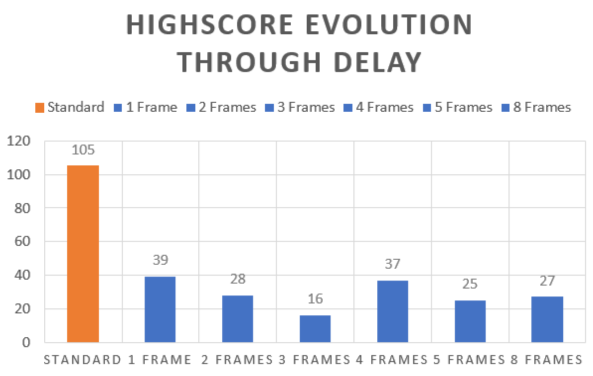

Figure 4.15: Average highscore for delay training

4.2.2 Results

The results achieved while applying delay are indeed much worse than the standard values. The rewards and high scores obtained (see Table 4.4 and Figures 4.17 and 4.18) are much smaller, as well as the number of steps executed on each episode.

Although the drop off is high for this type of limitation, it's visible that for 1 and 5 frame delays the action distribution it's in line with what we have been expecting from standard values, albeit with bigger emphasis on non moving actions. Beginning on the 2 frames delay the action distribution is more erratic, having for 2 frames delay a high percentage of 'Left' actions while the other actions saw a drop off in executions. For 3 frames it's visible the preference given to moving actions in detriment of non moving actions. For the highest delay value tested we have a similar distribution to the 2 delay frame, with a smaller set of number of steps, all of which is displayed on Fig. 4.13.

4.2.3 Analyses

The delay implemented seems to cause some problems to the training agent. The 1 frame delay is the best in terms of results, which seems plausible because we are only forcing the agent to pass it's intended action to the next frame. The actions executed for the 1 frame delay also represent the best adaptation to the delay by the agent, since it has a good distribution of actions, while the higher delay frames don't.

In the 2 frame delay the agent prefers to execute the 'Left' action a larger number of times, trying to maybe compensate the possible incapability of following normal standards and reaching the ball. If these results are confirmed as patterns adopted by real players, it would be of extreme importance to the game design, since such a small delay would be important of taking into consideration during the game design process.

For the 3 frame delay the high percentage of moving actions indicate that the agent is trying to do its best to achieve higher reward values, knowing that with the delay it might not be possible. By forcing more moving actions may lead the agent to reach the ball in more occasions.

We can state that when the delay spreads into another training step (4 delay frame), it exists an increase in reward values. This seems to indicate that by pushing the actions 1 training step forward allows the agent to better adapt to the delay in place. On this case, it decided that using more non moving actions would allow for better results, something that looking at previous results and the 1 frame delay might be true. From 4 frames forward the rewards decrease again and spreading the delay into another training step(8 delay frame) might be too much for the agent to handle and adapt.

4.3 Miss Frames

With this limitations we wanted to simulate a visual type of limitation a player might have, or a simple visual miscue. How would an agent react if from time to time it couldn't observe the environment? In order to simulate this, every state we have a probability of having what we called a visual miss. This consists of telling the agent that the frame in question is not to be actuated and observed. We do that by forcing the action to be executed on that state to be either 'Don't Move' or 'Fire', because the environment always forces an action to be executed. And additionally we don't store that state and information of the action in the memory of experiences. This way it's as if we never saw the current frame and so we simulate a missed frame.

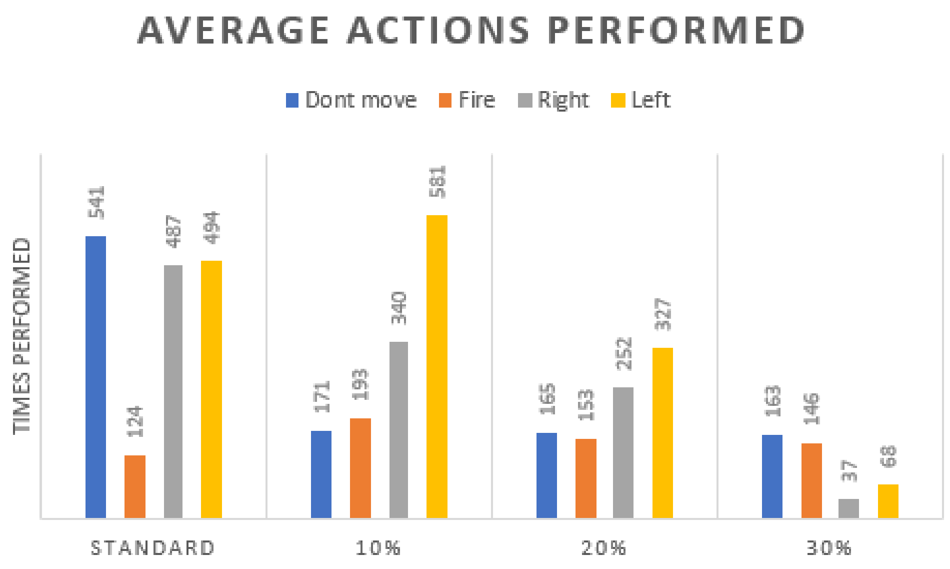

Figure 4.16: Action Distribution for various miss frames

4.3.1 Parametrization

As with all the scenarios tested before, it's ensured that all of the training processes have the same base values. In this scenario, the probability of occurring a miss on the frame ranges from 10% to 30%. We decided to cap the probability at 30% since values much bigger than that would cause too big of an impact on the results.

Table 4.5: Miss Frame Trained Agent Results

Limitation	Value	Reward	NºSteps	Highscore	Don't Move	Fire	Right	Left
Standard	-	51	1646	105	541	124	487	494
Miss Frame	10%	42	1285	96	171	193	340	581
Miss Frame	20%	27	897	27	165	153	252	327
Miss Frame	30%	8	414	8	163	146	37	68

4.3.2 Results

The results obtained for this limitation scenario brought some problems with them. Through various test runs they didn't seem to be always as coherent between themselves, achieving a wide range of results. On Table 4.5 we display the averages we were able to calculate. The values seem to indicate a fast decline in reward and highscore values when the probabilities increase and in terms of action distribution (Figure 4.16) the decrease of moving actions is evident, while the non moving actions tend to maintain the same volume of execution.

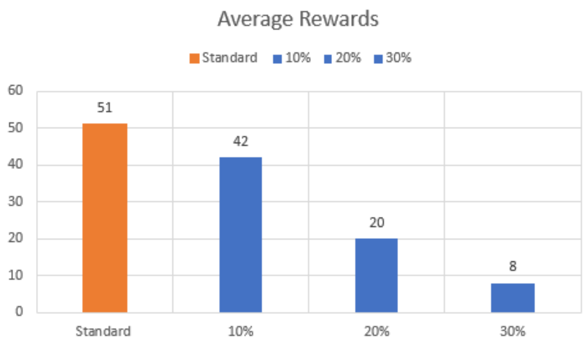

Figure 4.17: Average reward for missing frames

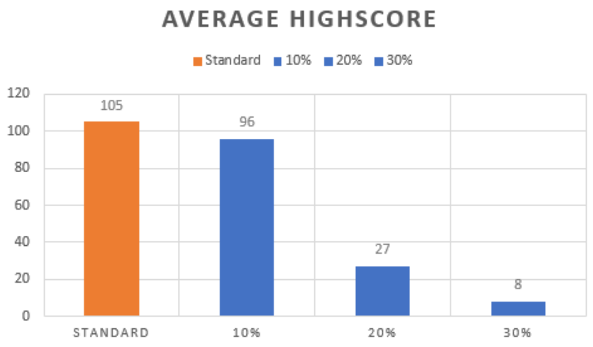

Figure 4.18: Average highscore for missing frames

4.3.3 Analyses

Looking back at the training process, we are inclined to believe that the variation of results we obtained where due to the fact that we are simulating a wide range of frames being missed. That might be the cause to why with higher percentages the results tend to dip in performance so abruptly. If we look at the action distribution, we see a pattern of play that evolves with the increase of miss probabilities. This pattern seems to be maintained throughout the various probabilities, in which the agent forces the execution of non moving actions in detriment of moving actions. The results obtained show that this pattern doesn't seem to achieve great results but it's probably the best option the agent found to counter the increasing probability of missing more and more frames.

4.4 Overview

Looking back at all the results achieved (Table 4.6, we can state that the best performing limitations are related to the agent being able to adapt to different types of action changes. The agent found various ways of trying to overcome the limitations, leading to better results overall. While the other limitations implemented might not have as great results in terms of scores, they clearly showcase that the constraints led to changes on the agent patterns of play in order to try and achieve the best results possible in the environment settings it is forced to be. The delay frames limitations show us that the agent might have a limit as to how much we can delay its actions and have it achieve reasonable results and the missing frame scenario suggests that with only a small percentage of missed frames a player might be able to achieve reasonable scores, since for higher percentages the game becomes somewhat unplayable.

Table 4.6: Trained Agents Results

Limitation	Value	Reward	NºSteps	Highscore	Don't Move	Fire	Right	Left
Standard	-	51	1646	105	541	124	487	494
Frame Delay	1	27	991	39	318	205	211	257
Frame Delay	2	19	720	28	194	18	86	422
Frame Delay	3	13	627	16	17	45	311	254
Frame Delay	4	25	958	37	138	477	123	220
Frame Delay	5	19	757	25	280	119	148	210
Frame Delay	8	15	580	27	225	6	83	266
Both Actions Reverse	10%	37	1300	67	297	97	250	656
Both Actions Reverse	30%	31	1143	46	0	359	270	514
Both Actions Reverse	50%	22	846	28	69	78	239	460
One Action Reverse	10%	48	1635	90	137	457	427	614
One Action Reverse	30%	47	1447	92	699	288	265	225
One Action Reverse	50%	44	1230	107	341	393	186	310
Both become 'Don't Move'	10%	45	1506	84	85	96	628	697
Both become 'Don't Move'	30%	41	1517	77	505	112	430	470
Both become 'Don't Move'	50%	41	1476	71	445	85	495	451
Miss Frame	10%	42	1285	96	171	193	340	581
Miss Frame	20%	27	897	27	165	153	252	327
Miss Frame	30%	8	414	8	163	146	37	68

5 Conclusion

Contents

In this chapter we will attempt to summarize our work, including the solution proposed, the various limitations and networks training and the conclusions obtained from the results achieved. To finish off, we present some possibilities on how to improve and confirm the work and solution that we propose.

5.1 Conclusions

This thesis work began with the intent of understanding if it was possible to accurately modulate certain types of player limitations using deep learning algorithms. With that work goal in mind we studied the field of General Game Playing and also delve into the state of the art deep learning algorithms and techniques in order to find the base to propose our solution.

We implemented the deep learning algorithm we saw most fit(DQN) and proposed and developed our model solution applied on top of the algorithm. On our model we recreated a series of player limitations and tested our model training various set of DQN networks following the limitations in play (Actions have a probability of changing, Actions are delayed a number of frames and the agent has a probability of missing certain frames).

The results achieved allowed us to conclude a series of strategies we believe would be visible in real human players and real playing limitations. In the case of changing actions the agent found various ways of countering the limitations imposed. The strategies ranged from forcing the executions of a certain type of action in order to compensate for the probability of changing actions, to showing preference in having the games paddle stay in one place and only move when it was most rewarding, compensating the possibility of actions changing a high number of times.

For the delay the conclusions are not as clear, but we know that the agent was able to better adapt to delays that spread exactly from one training step to another while the intermediates tended to achieve worse results due to cross training actions being executed between 2 training steps.

The missing frames tests don't provide us we great conclusions, since the results obtained varied for the same training, but taking into account the averages, the agent tried to adapt to the situation in play by reducing the moving actions, therefore changing its pattern of play as well.

If these observations and patterns are indeed confirmed, through user tests, we could affirm we have found a good way of simulating various types of player limitations using Deep Learning methods, through the implementation of our proposed model.

5.2 Future Work

The work done in this thesis proposed a model capable of simulating and representing different types of limitations using deep learning algorithms. Although the results achieved seem promising, they are merely theoretical. In order to prove the accuracy of the results obtained, the future works needs to focus on testing the same type of limitation in real players. Be it physically limited players, or simply through the applications of limitations into the games core mechanics. As an example, taking the results of the moving actions becoming non moving actions, it's concluded that the agent tries to force an execution of more moving actions to compensate for the high probability of action changes. By forcing this limitation on testing players, we should be able to observe the same type of results or since our training was limited, the same type of strategies as we were able to obtain.

If the results are indeed confirmed through user testing, it should also open the door to a wider variety of limitations to be implemented and explored, as well as testing with other types of games and algorithms.

Bibliography

[1] K. Wong, "Starcraft 2 and the quest for the highest apm," Last accessed 19 Dec 2018. [Online]. Available: https://www.engadget.com/2014/10/24/starcraft-2-and-the-quest-for-the-highest-apm/

[2] M. Genesereth and N. Love, "General game playing: Overview of the aaai competition," *AI Magazine*, vol. 26, no. 2, p. 62–72, 2005.

[3] Y. Björnsson and H. Finnsson, "Cadiaplayer: A simulation-based general game player," in *IEEE Transactions on Computational Intelligence and AI in Games*, vol. 1, 2009, pp. 4–15.

[4] D. Silver, A. Huang, C. J. Maddison, A. Guez, L. Sifre, G. van den Driessche, J. Schrittwieser, I. Antonoglou, V. Panneershelvam, M. Lanctot, S. Dieleman, D. Grewe, J. Nham, N. Kalchbrenner, I. Sutskever, T. Lillicrap, M. Leach, K. Kavukcuoglu, T. Graepel, and D. Hassabis, "Mastering the game of go with deep neural networks and tree search," *Nature*, vol. 529, pp. 484–489, 2016.

[5] J. Levine, C. B. Congdon, M. Ebner, G. Kendall, S. M. Lucas, R. Miikkulainen, T. Schaul, and T. Thompson, "General Video Game Playing," in *Artificial and Computational Intelligence in Games*, ser. Dagstuhl Follow-Ups, S. M. Lucas, M. Mateas, M. Preuss, P. Spronck, and J. Togelius, Eds. Schloss Dagstuhl–Leibniz-Zentrum fuer Informatik, 2013, vol. 6, pp. 77–83.

[6] T. Schaul, "A video game description language for model-based or interactive learning," in *Proceedings of the IEEE Conference on Computational Intelligence in Games*. IEEE Press, 2013.

[7] M. G. Bellemare, Y. Naddaf, J. Veness, and M. Bowling, "The arcade learning environment: An evaluation platform for general agents," *CoRR*, vol. abs/1207.4708, 2012. [Online]. Available: http://arxiv.org/abs/1207.4708

[8] J. Miner, "Atari 2600," Atari Inc, 1977.

[9] D. Perez-Liebana, J. Liu, A. Khalifa, R. D. Gaina, J. Togelius, and S. M. Lucas, "General video game AI: a multi-track framework for evaluating agents, games and content generation algorithms," *CoRR*, vol. abs/1802.10363, 2018. [Online]. Available: http://arxiv.org/abs/1802.10363

[10] A. Khalifa, D. Perez-Liebana, S. M. Lucas, and J. Togelius, "General video game level generation," in *GECCO*, 2016.

[11] K. O. Stanley and R. Miikkulainen, "Evolving neural networks through augmenting topologies," *Evolutionary Computation*, vol. 10, no. 2, pp. 99–127, 2002.

[12] M. Hausknecht, J. Lehman, R. Miikkulainen, and P. Stone, "A neuroevolution approach to general atari game playing," *IEEE Transactions on Computational Intelligence and AI in Games*, 2013.

[13] A. Krizhevsky, I. Sutskever, and G. E. Hinton, "Imagenet classification with deep convolutional neural networks," in *Advances in Neural Information Processing Systems 25*, F. Pereira, C. J. C. Burges, L. Bottou, and K. Q. Weinberger, Eds. Curran Associates, Inc., 2012, pp. 1097–1105.

[14] L. A. Gatys, A. S. Ecker, and M. Bethge, "A neural algorithm of artistic style," *CoRR*, vol. abs/1508.06576, 2015. [Online]. Available: http://arxiv.org/abs/1508.06576

[15] K. Kunanusont, S. M. Lucas, and D. P. Liebana, "General video game AI: learning from screen capture," *CoRR*, vol. abs/1704.06945, 2017. [Online]. Available: http://arxiv.org/abs/1704.06945

[16] V. Mnih, K. Kavukcuoglu, D. Silver, A. Graves, I. Antonoglou, D. Wierstra, and M. A. Riedmiller, "Playing atari with deep reinforcement learning," *CoRR*, vol. abs/1312.5602, 2013. [Online]. Available: http://arxiv.org/abs/1312.5602

[17] V. Mnih, K. Kavukcuoglu, D. Silver, A. A. Rusu, J. Veness, M. G. Bellemare, A. Graves, M. Riedmiller, A. K. Fidjeland, G. Ostrovski, S. Petersen, C. Beattie, A. Sadik, I. Antonoglou, H. King, D. Kumaran, D. Wierstra, S. Legg, and D. Hassabis, "Human-level control through deep reinforcement learning," *Nature*, vol. 518, no. 7540, pp. 529–533, 2015.

[18] S. Lange and M. Riedmiller, "Deep auto-encoder neural networks in reinforcement learning," in *The 2010 International Joint Conference on Neural Networks (IJCNN)*, 2010, pp. 1–8.

[19] V. Mnih, A. P. Badia, M. Mirza, A. Graves, T. P. Lillicrap, T. Harley, D. Silver, and K. Kavukcuoglu, "Asynchronous methods for deep reinforcement learning," *CoRR*, vol. abs/1602.01783, 2016. [Online]. Available: http://arxiv.org/abs/1602.01783

[20] J. Schulman, F. Wolski, P. Dhariwal, A. Radford, and O. Klimov, "Proximal policy optimization algorithms," *CoRR*, vol. abs/1707.06347, 2017. [Online]. Available: http://arxiv.org/abs/1707.06347

[21] J. Schulman, S. Levine, P. Moritz, M. I. Jordan, and P. Abbeel, "Trust region policy optimization," *CoRR*, vol. abs/1502.05477, 2015. [Online]. Available: http://arxiv.org/abs/1502.05477

[22] M. Plappert, "keras-rl," https://github.com/keras-rl/keras-rl, 2016.