Learning from the environment:

Deep Reinforcement Learning for generalizable Data-driven decisions

Alberto Ruiz Benitez de Lugo Hernandez

School of Mathematics, Computer Science and Engineering

City, University of London

Thesis Supervisor: Eduardo Alonso

A thesis submitted for the degree of

MSc Data Science

London 2017

Abstract

Nowadays Deep Reinforcement Learning represents the cutting-edge of technology regarding Artificial Intelligence. However, there are still serious limitations to be covered, such as the absence of generalizable memory approaches to increase the learning speed. The nature of this project aims to create an Artificial Intelligence framework capable to export knowledge from one environment to another, in order to solve difficult sequential decision-making tasks with less training time.

This goal will be achieved using a memory architecture, based on neuroscience hippocampal models, to contain the knowledge learned from one environment. Next, this memory will be used to initialize a different and never seen environment, using Experience Replay and Double Q-Learning methods. This memory framework will help the agent to solve this second environment more easily, using previous information as initialization.

Contents

List of Figures

List of Tables

Introduction, Objectives and Critical Context

Nowadays we are seeing rapid improvements in Artificial Intelligence regarding end-to-end reinforcement learning task, such as games [16]. Although, there is still a lack of robust generalizable approaches [12], particularly using memory while these agents interact with the environment.

Therefore, the aim of this project is to provide a generalizable framework able to transfer the knowledge from one environment to another using memory. This topic is highly related to hippocampus models in neuroscience, particularly with hippocampal episodic control [1][16]. Currently, Deep Reinforcement Learning techniques usually require millions of interactions to achieve human-level performance [12]. This is equal to hundreds of hours playing games and it is not longer affordable with complex environments under real-time circumstances, such as autonomous cars.

Thus, the objective of this work is to create a Convolutional Neural Network framework [16] able to store playing experiences from an Atari game, based on pixel images, and use this stored information to play a different Atari game with ease. The expected result, or hypothesis, is that similar games architectures will provide good results. This hypothesis is based on the pixel input structure, that would provide strong patterns between similar games, such as "Shoot and avoid get shot" behaviors seen in many Atari games like Space Invaders.

Regarding critical context, the main bottleneck in artificial intelligence is the lack of conscience [12]. These days we call intelligence to maximize a reward or reduce an error within a particular model. However, this knowledge is not fully exportable or generalizable to other situations. The motivation of this project is to set a base to generalizable intelligence, where the information obtained through a particular system can be re-used to other environments, as real knowledge does. Therefore, this approach moves away from rigid AI models that only try to improve a particular result within a particular situation. Instead, this scheme focus on generalizable AI, with abstraction capabilities and able to connect two different situations as a human does, using consciousness.

As a result, over this project different methods has been wrapped up, such as Deep Q-Learning, Double Q-Learning and Experience Replay. In order to provide a generalizable framework that transfer information learnt from one Atari game to another. Hence, the agent will use this memory information to start acting in the new environment, instead of taking actions from scratch. The work plan to develop this framework follows the stated order shown in section 2, starting with the state representation.

Finally, regarding report structure, this work will start defining the theoretical background needed to understand the implementation in section 1, followed by the methods and techniques used to create the framework architecture in section 2. Next, the experimental results will be shown in section 3, as well as the important insight discovered during this work. Then, section 4 presents the final conclusions and future work opportunities.

1. Theoretical Background

1.1 Reinforcement Learning

Reinforcement Learning (RL) [18] is a particular area of Machine Learning based on behavioural psychology. An agent takes actions within an environment and gets certain reward in return.

These sequences of rewards define the behavior of the agent, known as policy. Many researchers suggest that this can partially be explained by associative processes within the animal kingdom, resulting from past experiences in association with pain or pleasure [13]. This may lead the behavior through positive and negative reinforcement, reward and punishment respectively.

Regarding computational theory, the concept of reinforcement Learning is applied to computational pattern-recognition. The main different between RL and other areas in Machine Learning, such as Supervised Learning, is that patterns are learned from experiences instead of labeled examples. Figure 1.1 defines this scheme.

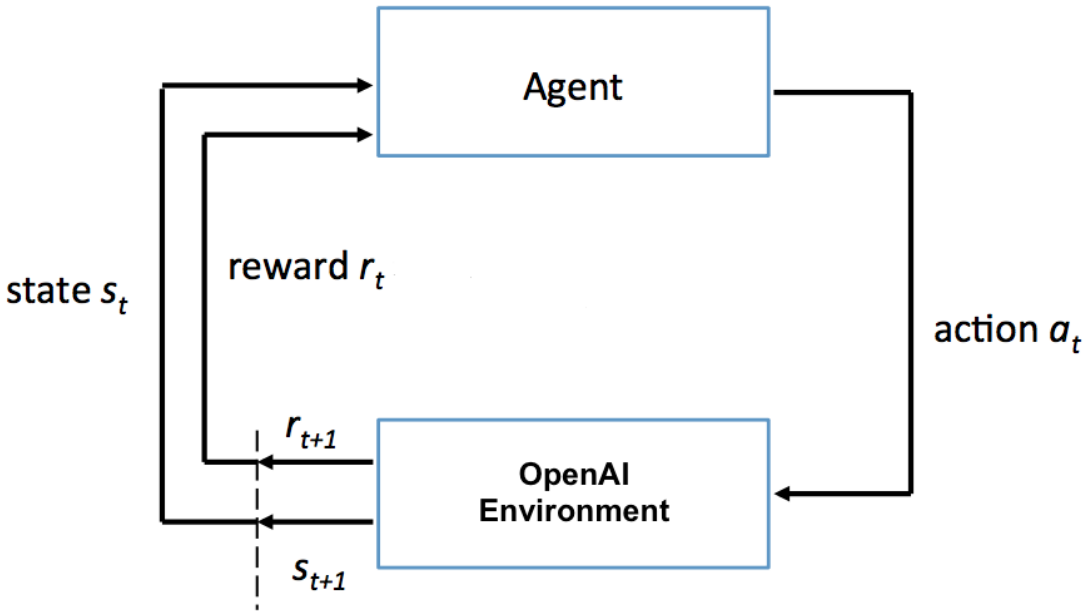

Fig 1.1 **Reinforcement Learning Cycle**. Source: OpenAI environment, Prepared.

As seen in Fig 1.1, the Reinforcement Learning agent takes actions in the environment, receiving numerical rewards in return. Then, based on the acquired rewards attached to the given observations (State), the agent updates its policy function in order to obtain the best possible action for the next iteration.

1.2 Markov Decision Process

Markov decision processes (MDP) define a mathematical framework for decision making [9]. Precisely, it could be defined as a discrete time stochastic control process. Thus, a MDP is very helpful to solve a decision problem by reducing the problem to its essential parts. These models are very similar to a Markov Chain, but the transition matrix depends on the action taken by the agent at each time step.

Therefore, MDP consist on all possible states (s), actions (a) and rewards (r) for each possible state transition. Where these states contain all the important information in the past, known as Markov Property.

Future rewards could be less probable than nearest rewards. In other words, the future reward ought to be discounted by a parameter γ. This discount rate γ satisfies $0 \leq \gamma > 1$. As a result. when the discount rate equals $\gamma = 0$, just most recent rewards are taken into account, whereas the more γapproaches to one, the future rewards will be taken into consideration. This can be defined mathematically by the next formula:

$$\sum_{t=0}^{n} = \gamma^t R_a(s_t, s_{t+1})$$

1.3 Q-Learning Algorithm

The Q-Learning Algorithm can be defined as a model-free Reinforcement Learning technique. This method is able to provide an optimal solution, or optimal policy (π), that can solve any environment represented as a Markov Chain with finite state-action pairs. Thus, trying all possible combinations and selecting the best in every state.

The algorithm works by learning an action-value function related to the rewards obtained by iteration with the environment. This collection of state-action pairs are commonly called Q-values.

$$\pi(s_t) = \arg\max_a Q(s_t, a_t)$$

Therefore, Q-Learning [18] is represented as Q(s,a), defined as the future expected reward per state (s), once an action (a) is taken. As stated above, the convergence of the algorithm always apply for any finite Markov Decision Process [4]. An optimal policy (π) is obtained by selecting the action that maximizes the expected reward among all possible actions available in the current state (s).

1.4 Convolutional Neural Networks

Convolutional Neural Networks (CNN) is a particular case of feed-forward neural networks. The CNN architecture was created to find patterns within high dimensional data, such as images. This kind of networks have been widely applied in different areas, but specially in computer vision and image recognition [7] [8]. This kind of architectures have been essential for this project, because all the environments are solved by using images inputs.

The main difference between CNNs and other fully-connected neural networks is that CNNs uses convolutions on every layer. These layers acts as filters, or kernels, in order to discover and classify patterns. The weights of these filters are updated using stochastic gradient descent.

1.5 Deep Reinforcement Learning

Q-Learning algorithms store Q-values into a data table or any other indexed data structure. However, this is not practical, nor possible, when dimensionality is too high. The algorithm would need too much memory and processing power to converge.

1.5.1 Deep Q-Learning Algorithm

Deep Q-learning algorithm (DQN) is the approach taken when such high-dimensional situations occurs. Under this architecture, the Q-values are interpolated by a CNN instead of a data structure. This approach was first implemented in DeepMind [11], where a serie of Atari games were solve using this method.

An essential fact is that Deep Q-Learning does not take the argmax Q-value for every state-action pair. Instead, this algorithm employs Convolutional Neural Networks (CNNs) as function approximators [2]. In other words, it approximates the expected Q-values, which is a prediction. This property strongly differs with original Q-Learning, where the argmax Q-values are obtained from previous pairs, stored in a data structure. As a result, convergence property does not always apply in DQL.

1.5.2 Double Q-Learning

The utilization of function approximators may incur in expecting too much, or too often, a particular value. This would provide overestimation/underestimation of certain values given a state-action pair, which is a big threat for DQL convergence.

In order to solve this problem, Double Q-Learning is implemented. This method avoids over/underestimation by comparing the main network (Q-net) against another network, Target network (Target Q-net). Under this approach [19] the performance of Deep Q-Learning algorithm (DQL) improves significantly.

1.5.3 Experience Replay

One of the biggest limitations in AI is the absence of conscience. Thus, the first step to deliver conscience in Artificial Intelligence is providing memory.

Experience Replay [17], provides a very good framework to deliver memory to our agents. This method store "memories" in a database or memory. Then, a mini-batch of experience, defined as s, a, r, s tuples, are sampled from the memory and used to determine an action to take next. The recurrent use of this "experiences", allow to compare previous state-transitions and improve the efficiency on DQL. The underneath idea of this implementation is based on hippocampus models from animal brains [1][16].

Moreover, this method was developed to improve the predictive capacity of DQL. Basically, updating the system using subsequent state transitions violates the assumption of independent and identically distributed states (i.d.d.), Experience Replay solves this problem.

1.6 Exploration vs Exploitation

Similarly to Q-Learning, there is a dilemma between exploration and exploitation in Deep Q-Learning. The final goal for an agent is to always select the most effective action based on previous experiences. Although, to achieve this objective the agent needs to see a situation never observed before. Hence, the trade-off between taking actions that may lead to new observations (exploration) and actions that would provide the best expected reward (exploitation) defines the agent behavior and its final result.

$$0 \leq \epsilon < 1$$

This situation can be solve by applying different techniques, such as ε-greedy. This method defines an ε value that balances both exploration and exploitation metrics. This parameter (ε) defines the probability of taking a random action (exploration) within the action space (number of actions possible), this provide new situations and opportunities to improve the optimal policy. Therefore, 1 - ε times the algorithm will exploit the current optimal policy, selecting the action that provides the highest expected reward (Q-value). Thus, ε-greedy method could be formulated as follows:

$$with\ probability\ \epsilon : Take\ random\ action\ (Explore)$$

$$with\ probability\ 1 - \epsilon : Selects\ best\ possible\ action\ (Exploitation)$$

2. Method Architecture

2.1 Memory based Framework

The lack of robust memory approaches that ensure generalization has encourage this project. As said, the aim of this work is to provide a generalizable framework able to transfer information from one environment to another. Fig 2.1 represents this memory based framework.

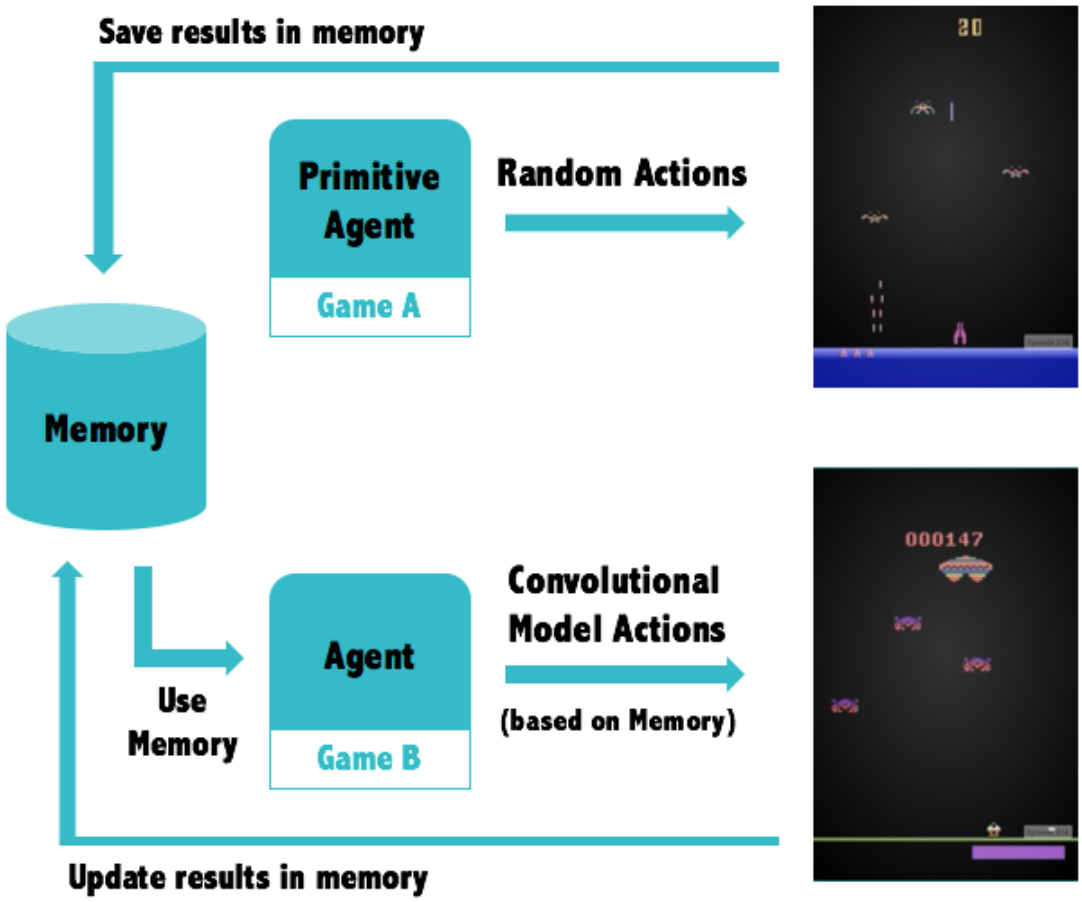

Fig 2.1 **Memory based Framework**. Source: OpenAI environments, Prepared.

As seen in Fig 2.1, there are two clear steps to generalize knowledge between environments. First, a random-action-based agent, called Primitive Agent, plays a particular game and save the results in the memory. This agent will play until the memory is full of capacity. This represents the initial knowledge that will be exported.

Next, a second agent will play a different game using a Convolutional Neural Network as model approximator [2]. However, this agent does not start the decision process of what actions to take from scratch, instead it will use the memory to initialize this process [17]. Hence, if the second game is highly similar to the first one, the information stored in the memory will be quite helpful for the CNN approximator, and therefore the learning process will be very quick. On the other hand, if the previous game differs too much from the current game, the initialization would not bee so useful. Finally, this second agent will update the new information it obtain and will replace it in the memory using priority based on results.

2.2 Atari Environments using OpenAI

The environments applied in this project come from OpenAI [14]. This platform provides a high number of environments, from simple optimization problems to arcade games such as Atari. One particular advantage is that both, actions and states, are already internally developed in OpenAI, simplifying the coding part. Just Atari games will be implemented due to similar built-in features and pixel structure, Fig 2.2.

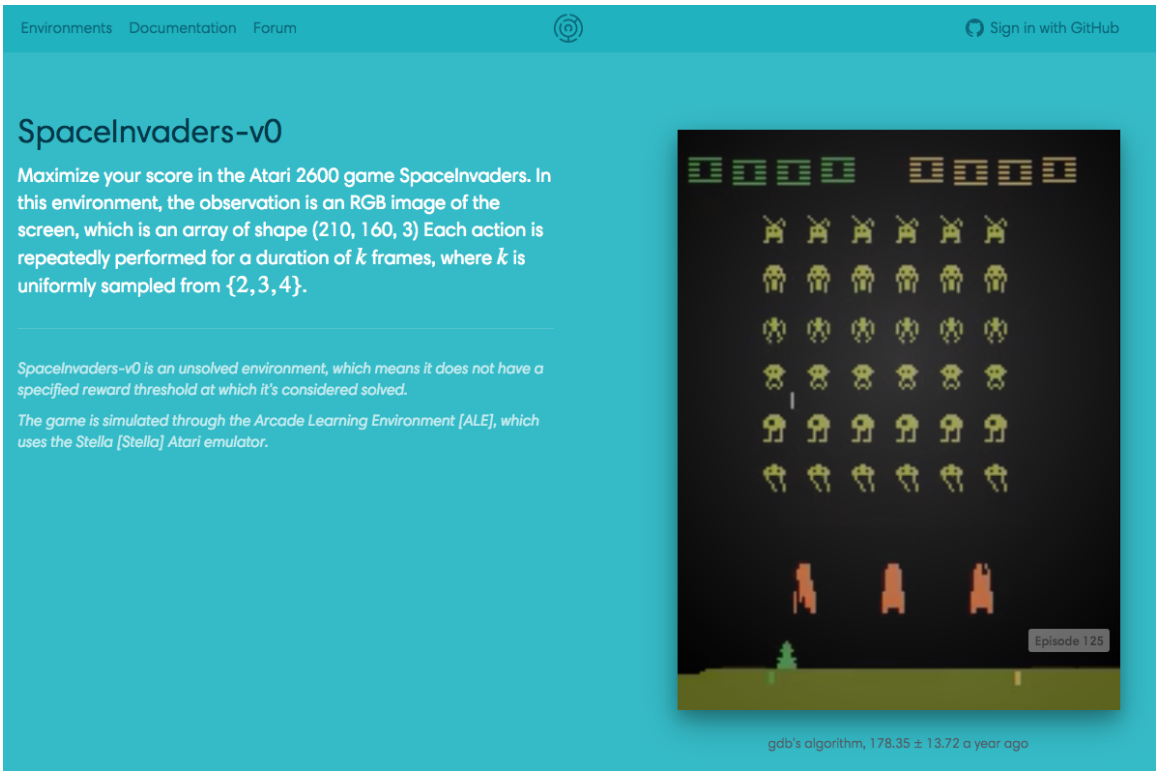

Fig 2.2 **Atari Game Description**. Source: OpenAI environments, Prepared.

2.3 State Representation

The states are represented as RGB pixel images with shape (210,160,3). Meaning 210 pixels high, 160 pixel width and 3 color channels; red, green and blue (RGB). These characteristics are the same in every Atari game, and they need to be processed to gray-scale before applying the Convolutional Neural Network (CNN).

2.3.1 Pixel Density

There are two important facts while applying this architecture, the Processing Efficiency and the Generalization Level. Both of them can be managed by controlling the pixel density from input image.

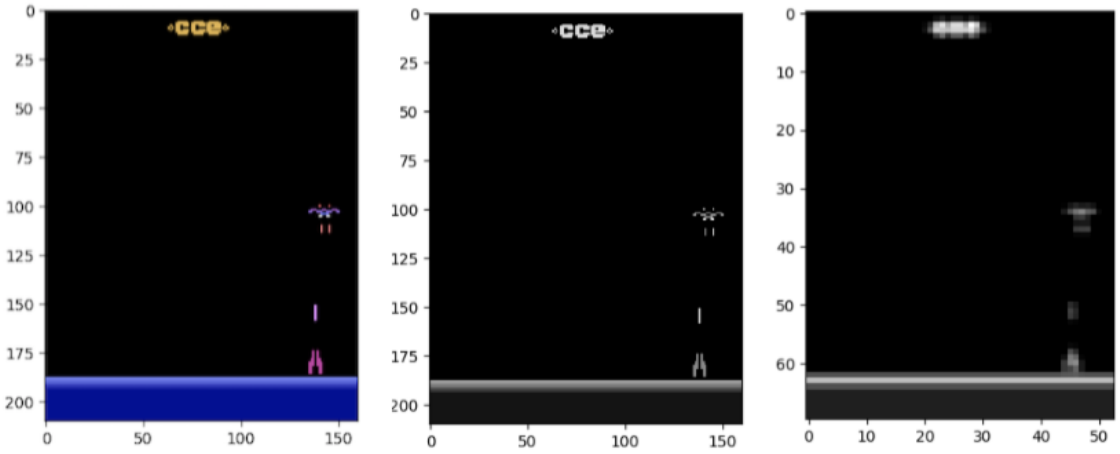

Fig 2.3 **Pixel Density example**. From left to right; Original input, Gray-scale input and Low Pixel Gray-scale input. Source: OpenAI environments, Prepared.

Thus, high pixelated images takes more time to process and store in memory than low pixel images. In the same way, the Convolutional Agent not only needs less time to evaluate low pixel states while taking decisions, but also provides more generalizable decision-making actions. As seen in Fig 2.3, as long as the image becomes more complex, turns less similar to others. As a result, low pixel inputs are more effective regarding processing speed and generalization. This is because the inputs look like more similar between each other.

2.4 Brain Architecture

The code has been developed by classes and one of the most important is the Brain. This Python Class encapsulates the Convolutional Neural Network (CNN), here the algorithm trains and predict the Q-values. Also, it applies Double Q-Learning to update the network and avoid overestimation with certain Q-values.

2.4.1 Convolutional Network Implementation

Once the Pixel Density has been reduce enough, we can obtain fast and efficient results from the Convolutional Neural Network (CNN) [11].

The states (RGB inputs) approached during this project are fairly complex, even after reducing the number of pixels. Thus, three Convolutional Hidden Layers with Rectified Linear Unit (ReLU) have been necessary to achieve good results. The first hidden layer has 32 neurons units, whereas hidden layers two and three have 64 neurons each. The last part of the networks is a fully connected layer, also using a linear activation function. Although, instead of using Gradient Descend (GD), a more sophisticated method available in Keras library has been implemented, RMSprop, which provides better results. Finally, Mean Squared Error (MSE) is implemented as loss function.

2.4.2 Double Q-Learning Update

A very important fact regarding Deep Q-Learning is that the agent tends to overestimate the Q-value function, because of the Max operator used in the formula to define the targets, as follows:

$$Q(s,a) \rightarrow r + \gamma MaxQ(s',a)$$

To illustrate this situation just imagine a particular state, such as the RGB image from an Atari game, with a finite set of possible actions. Now we find out that all these actions have the same true Q-value. However, there is noise and the estimation of these Q-values are not the same as the true values. Under this circumstances, the Max operator in the formula will select the action with highest positive value, this will be propagated in future states. As a result, this Q-value is overestimated due to the positive bias applied by the noise, and therefore affects the learning stability and convergence of the algorithm.

Hence, to solve this problem Double Q-Learning [19] has been applied to the architecture. This method uses two function approximators instead of just one, both functions learn independently. As a result, one approximator is used to maximize actions whereas the other estimates the values. The formula is shown as follows:

$$Q1(s,a) \rightarrow r + \gamma\, Q2(s', argmax\, Q1(s',a))$$

2.5 Memory Architecture

The purpose of Memory class is to store the obtained experience. This Class has been developed in concordance with a Sum Binary Tree Class. Both of them represent how the memory works and how the stored memories are applied while taking decisions. Such as the Priority schema applied for updating memories, and the Batch strategy to define how many items are taken into account.

The first issue while developing the framework was how to treat the memory. In essence, the memory needs to be huge to be able to contain as much information as possible. However, as long as the memory size increases, the complexity while taking decisions rises as well. Thus, a Batch strategy was a key solution, instead of taking all the memory items, just take a bunch of them. The method of this Batch strategy is implemented using Experience Replay, meanwhile the procedure of how to treat the items within this Batchs is developed using Sum Binary Tree method.

2.5.1 Dynamic Memory with Experience Replay

As said before, every step the agent takes an action (a) in state (s), receiving a reward (r) and then passing to a new state (s"). This pattern is defined as online learning and it is repeated providing (s, a, r, s) pairs that are stored in the memory for each iteration. This allow us to learn from them and may expect to achieve convergence in the long term. Unfortunately, this is no common in Deep Reinforcement Learning and therefore needs a technique to solve this problem.

Experience Replay is applied to solve this issue. Although, in order to fully understand this method we first need to understand the characteristics of online learning applied on memory agents. First, memory-based RL agents always store the output pairs (s, a, r, s) in every iteration. Secondly, these observations arrive in the same order they were experienced. In other words the observations are highly correlated. Because of this, the network is keen to overfit and reduce its generalization capacity.

Hence, Experience Replay sample a random bunch of stored pairs and use them to decide which actions to take next. However, the memory size is finite, only allowing a limited number of samples. As a result, once this capacity is reached we will substitute or not the old samples based on a Priority system [17]. This system depends on a parameter theta (ϑ), that defines a Proportional Priority when $\vartheta > 0$, and Uniform Priority once $\vartheta = 0$. Based on experimental results, Proportional Priority is preferred.

2.5.2 Sum Binary Tree

At the beginning of this project all the samples were sorted by priority within an array and then randomly selected. Nevertheless, this method was very slow and not too much efficient, as unsorted arrays would provide similar results.

The reason is that items with higher priority are more probable to be picked, independently they are sorted or not. Next, a data structure called Sum Binary Tree [21], was discovered and applied to the architecture. Under this scheme, our unsorted array can be treated as a leaf nodes data structure, where the value of a parent leaf is the sum of its children. This implementation is very effective and allow us to use very large memory sizes.

2.6 Agents

This project is based on two main agents, as showed in section 2.1. This architecture is partly based on Actor Mimic [15], where a multitask DQL algorithm was implemented. This basic idea has been wrapped up with Replay Memory and Double Q-Learning to be able to export the knowledge from one environment to another, rather than mimic the actions taken by other agent.

2.6.1 Primitive Agent

This agent is the first to interact. It basically plays in the first game (Game A) by taking random actions in every step. The results are then stored in the memory until reach the maximum capacity.

2.6.2 Convolutional Agent

This second agent is the one who uses the CNN to predict the highest expected return per action while playing the second game (Game B). Unlike Primitive agent, this one thinks while taking actions, using experiences stored in the memory to evaluate which actions to take next. Always keeping the goal of selecting the sequence of actions, or optimal policy (π), that maximize the results.

2.7 Rewards

As many OpenAI environments are used for this implementation, the reward structure is fairly heterogeneous. Although, they usually follow a particular structure: When the agent do anything good, such as kill an enemy, collect an object or reach an objective, a positive reward is given. However, if the agent gets killed, lose all of its points or run out of time, it receive a negative rewards. Finally, in every middle state when nothing of these happen, a reward with value of zero is obtained.

3. Experiments and Results

Once all the specifications of the algorithm have been described, we will start showing the results and insight obtained from the experiments. First, section 3.1 shows the main differences between export knowledge among different environments or not, as well as the most valuable insight about the experiments. Next, section 3.2 provides further details about the parameter optimization process and the main effects that every parameter has over the experiment. Finally, section 3.3 analyze the results according to different kind of games, rated from very similar environments to very complex and unrelated ones. Under this last section we will observe how generalizable the model is regarding the inputs chosen as memory initialization.

3.1 Game vs. Exportable Game

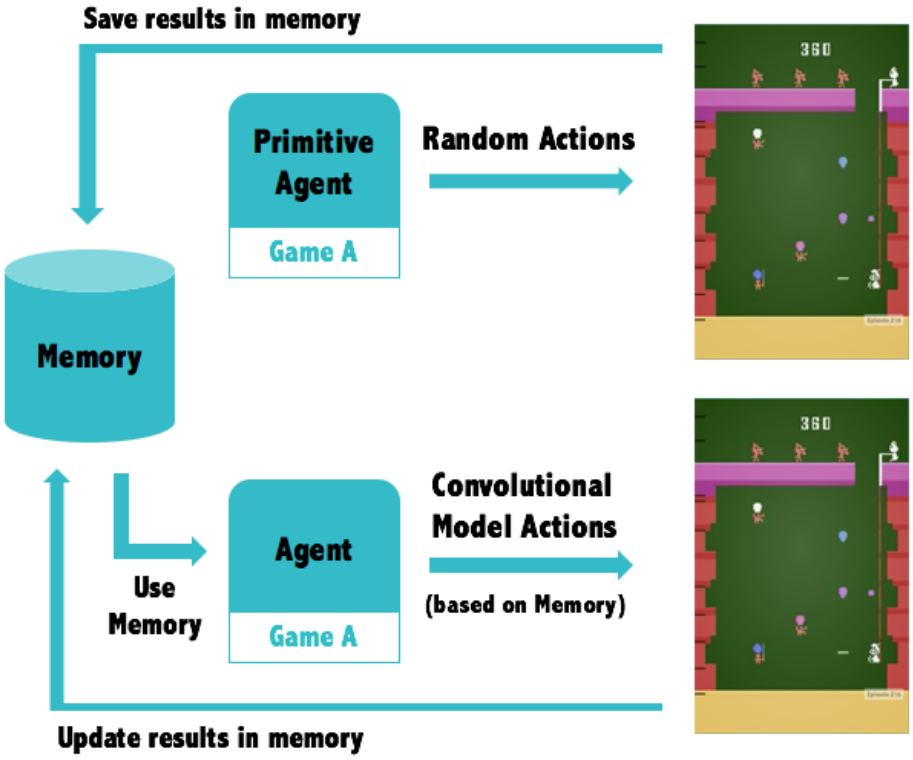

Fig 3.1 **Self-Memory based Framework**. Source: OpenAI environments, Prepared.

The background of this entire project is based on exporting knowledge from one environment to another. This can only be fully analyzed by comparing the results from a game using itself as memory initialization, Fig 3.1, against the results achieved by applying the original exportable framework, Fig 2.1 (Method architecture).

The first case, represented by Fig 3.1, is using the same game for both sides, Primitive and Convolutional. Thus, no transfer learning [6] is applied. This fact is very important, because the results obtained from this implementation should be higher than its counterpart. The main reason is that under this scheme we will play exactly the same game that we used to fill up the memory early on.

Fortunately, this hypothesis has come true in the results. As seen in Fig 3.2 and Fig 3.3, the results while applying self-Memory based Framework are clearly higher that when it is not implemented.

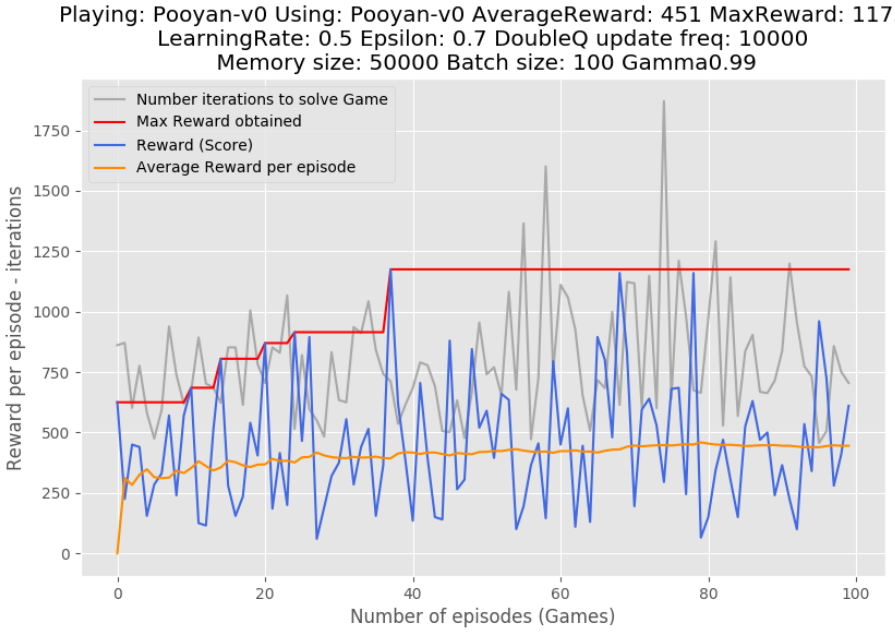

Fig 3.2 **Self-Memory Experiment Results**. Using Atari Game "Pooyan-v0" for both playing (CNN) and Memory Intit (Primitive). Source: Python 3.0, Prepared.

Table 1: Game vs Exported Game Experimental Results

Memory Framework	Average Reward	Max Reward
Self-Memory	451	1175
Exported-Memory	372	1000

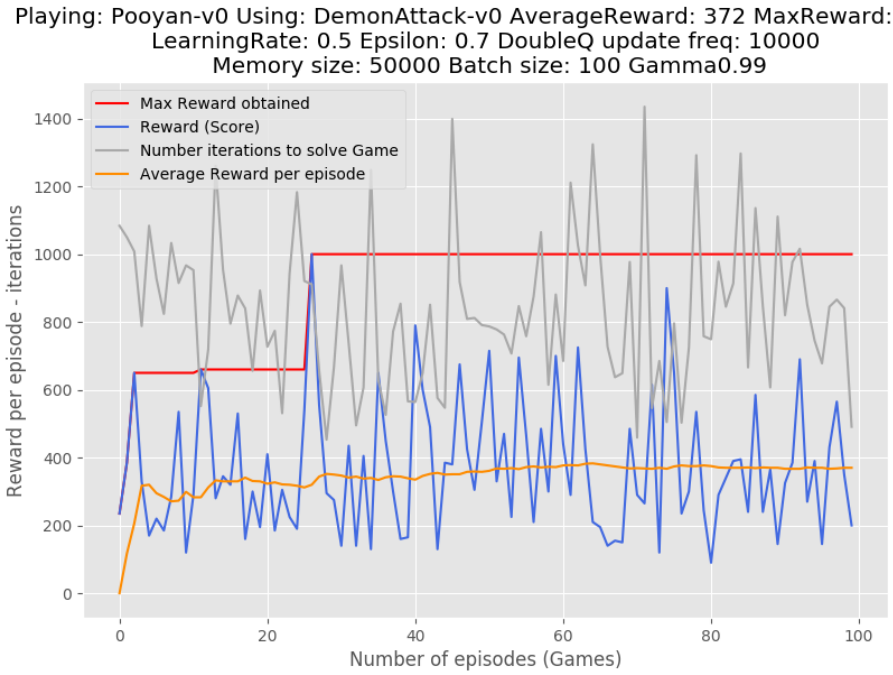

Fig 3.3 **Exported-Memory Experiment Results**. Playing Atari Game "Pooyan-v0" using "Assault-v0" as Memory Init. Source: Python 3.0, Prepared.

According to the obtained results, we can ensure that our Exported Knowledge Framework works fine. Not only because the Self-Memory results shows the expected output, higher than the Exported-Memory example. But also because the good performance shown in Fig 3.3, which output levels are very similar to Self-Memory scheme. This proves the high efficiency of the original framework, Fig 2.1.

3.1.1 Generalization and Abstraction

The Generalization Level is an important feature to take into account while developing Artificial Intelligence. In this experiment, this generalization is directly related to Abstraction Level. The reason is that the pixel density reduction applied to the input image is the main source of abstraction, and this is which allows the algorithm to set correlations between very different states, because the algorithm does not see very defined images anymore, instead it see groups of pixel spots, Fig 2.3, which makes it easy to set up correlation between states.

This idea becomes very clear when analyzing Fig 3.3. As denoted before, the system is playing "Pooyan-v0" while uses memories obtained from "Assault-v0". These two games are fairly different in color, characters size, positions and even the axis where the actions is taken; the agents should shot vertically while moving left-right for "Assault-v0", whereas it needs to shoot horizontally and move up-down for "Pooyan-v0". However, both games represent the same actions scheme, move side to side and shoot, in order to avoid get shot.

This instinct knowledge, based on neuroscience Hippocampal models [1][16], is extracted by reducing state complexity applying Pixel density reduction. However, the success of this approach also comes from the Experience Replay Memory using Multi-agent Reinforcement Learning (Primitive and Convolutional). In order to obtain a good generalization level on high dimensional and stochastic environments, both memory and batch size needs to be big enough, as seen in other experiments using StarCraft [5]. Thus, this combination of methods allows a stable training process.

3.1.2 Non-terminal State Issue: Exploration Rate

Another important issue faced during this work is the nature and structure of the Atari games. All of them has been selected by similarity, and even among their superfluous differences, all share a particular feature: They have no terminal state.

This fact is very important, at the beginning of the project a minimum exploration rate was enforced, Epsilon (ε): 0.5, in order to assure exploration rate in advanced iterations. The reason lays on this no-terminal state issue, because according to common sense, the agent should need to continue exploring, even in advanced steps, to improve the max scored. However, this was no true, as some games reach very good scores but it cannot improve this limit, no matter how much you explore. This detail comes from DeepMind first paper [], where depending on the game the result obtained changes drastically.

In order to prove this point, the code has been modified to train three different agents at the same time, Fig 3.4, using different exploration measures.

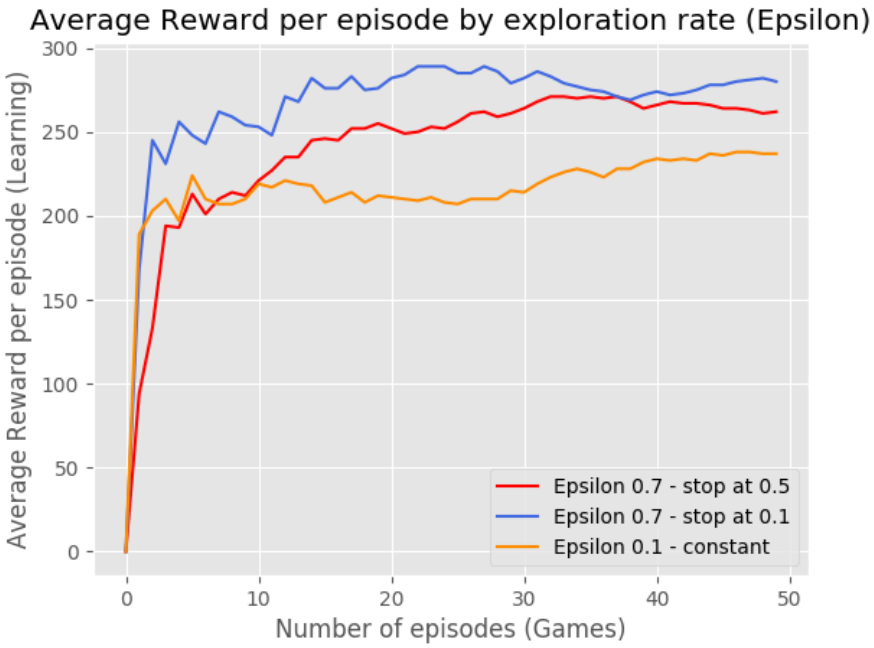

Fig 3.4 **Comparing Exploration Rates**. Different Epsilon applied. Source: Python 3.0, Prepared.

As seen in Fig 3.4, the initial idea of maintaining a high exploration rate, never going behind 0.5 (red line) is a good approach, far better than constant epsilon (orange line). Although, allowing the agent to reduce the exploration rate to low levels, such as 0.1 (blue line) is the best approach for this experiment. The reason is that once the maximum capacity of a game has been reached, the low epsilon rate allows the algorithm to maximize the best possible outcome, instead of keep looking.

3.2 Parameters Optimization

During the development all the Hyper-parameters has been tested to evaluate their repercussion on results. The next table shows the final parameters selection after optimization:

Table 2: Optimized Final Parameters

Memory Size	Batch Size	Double Q Update	Pixel Reduction
50.000	100	10.000	5
Learning Rate	**Gamma**	**Epsilon**	**Epsilon Decay**
0.5	0.99	0.7	0.01

Even with the final parameters available, it worth talking about the importance of some of them in the next section, especially those related to the memory.

3.2.1 Memory and Brain Parameters

Regarding Memory Parameters, a high storage capacity is essential to achieve good results. During the first stage of the algorithm, the Primitive Agent fills up this memory capacity. In order to avoid biases due to observation outliers the sample needs to be big enough. Therefore, Memory Size hyper-parameter has to be higher, such as 50.000, showed in table 2. The same happens with the Batch Memory, where 100 items per sample is more than enough.

Similarly, the parameters applied on the Brain Class are very important, especially theta (ϑ), "exponente" in the code. This parameters defines the priority, as seen in Prioritized Experience Replay paper from DeepMind [17]. The higher this value is, the better the architecture works. The reason is that $\vartheta = 0$ means Uniform Priority, meanwhile values near 1 define Proportional Priority and its level.

Finally, Double Q-Learning Update is necessary to avoid overestimation in particular values when the true values are the same but exists noise in the observations.

3.2.2 Agent Parameters

The hyper-parameters related to the Agents are similar to simple Q-Learning [18]. The learning rate (α) has been settle to 0.5 to ensure a stable learning process, neither too fast or slow. In the same way, Gamma (γ) is very high because we want to see many previous experiences as possible.

Additionally, Epsilon and Epsilon Decay has been given with those values according to the Exploration-Exploitation dilemma explained in section 1.6 of the project. Thus, the final architecture start by exploring with a 70 percent chance and then reduces this probability by 0.01 in every iteration, as shown in Fig 3.5.

```
def pickAction(self, this_state):
    if random.random() < self.epsilon:
        return random.randint(0, self.actions_space - 1)
    else:
        return numpy.argmax(self.brain.take_Max_Q_value(this_state))
```

Fig 3.5 **Exploration vs Explotation Code**. Either explore or exploit Q-values with the next action regarding epsilon (ε). Source: Python 3.0, Prepared.

3.3 Exportable Knowledge Results

Over this section different results will be explained according to its complexity level. In other words, all the games selected for this project have been divided into three groups regarding their similarity to shooting games. As a general rule, all the games will be tested using "DemonAttack-v0" as memory initialization.

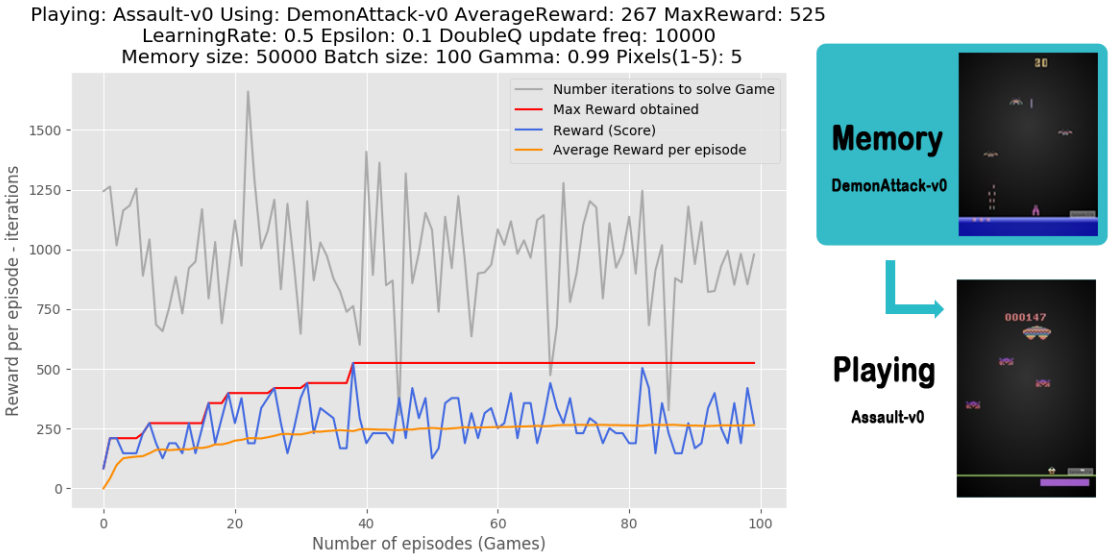

Fig 3.6 **Easy Exportable Game Results**. Playing Asssault-v0 while using DemonAttack-v0 as memory initialization. Source: Python 3.0, Prepared.

3.3.1 Easy Exportable Games

These kind of games are shooting-based games or are strictly similar to them regarding playability. This kind of games generally provide very good result, due to high concordance with the memory-based environment used to take decisions. Hence, the algorithm can extrapolate the stored knowledge more easily.

A clear example is Fig 3.6, which shows a very good performance. This happens because both games, DemonAttack-v0 and Assault-v0, are extremely similar. These games are based on up-down shooting while dodging (left-right) the enemy fire.

3.3.2 Common-Nature Games

On the other hand, the algorithm has been implemented in games that are slightly different, but representing the same structure, set of actions or task (shooting). The results obtained showed that this framework is good enough to manage this slightly complex environments.

To prove this point, Fig 3.7 represents the results obtained from playing Pooyan-v0 using DemonAttack-vo as memory initialization. The main difference between is that against the results obtained from Fig 3.6, this implementation takes more time to reach the stable value regarding "Average Reward per episode".

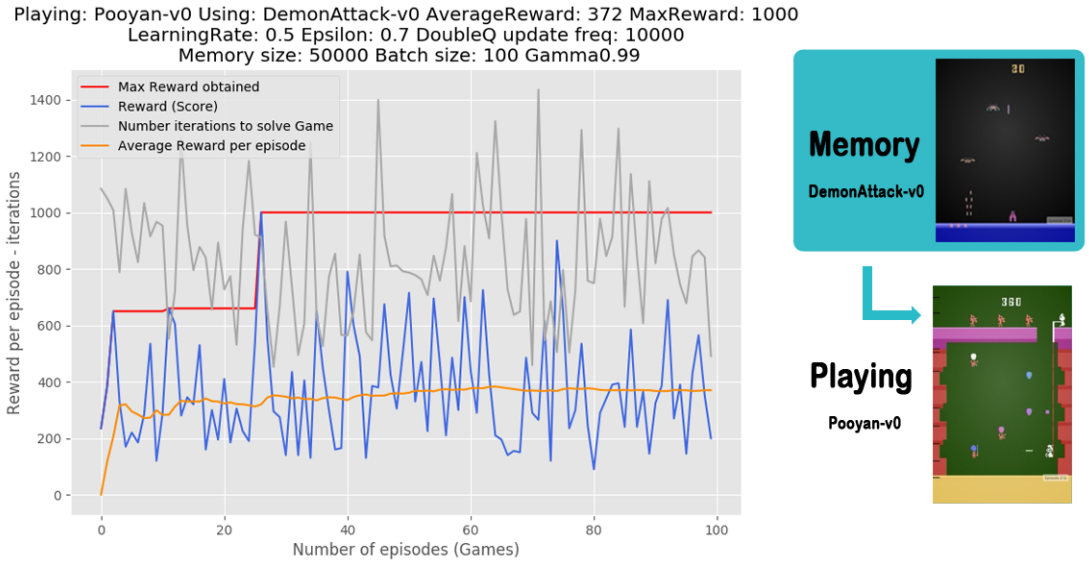

Fig 3.7 **Common-Nature Game Results**. Playing Pooyan-v0 while using DemonAttack-v0 as memory initialization. Source: Python 3.0, Prepared.

Furthermore, we can observe some perturbations in the first 30 episodes of learning. This represents that the knowledge stored in memory is not very efficient for this environment. Once the algorithm starts updating the memory using information from the new game, the results become stable.

3.3.3 Complex Games

In order to set a contrast, the algorithm has been implemented over very complex or unrelated environments, and the results are very surprising. First, this category has to be divided into two subcategories: Complex games that provide bad performance or not. Regarding the first subgroup, Fig 3.8, the bad output was expected. Basically the game is to complex or it not related enough to provide valuable results.

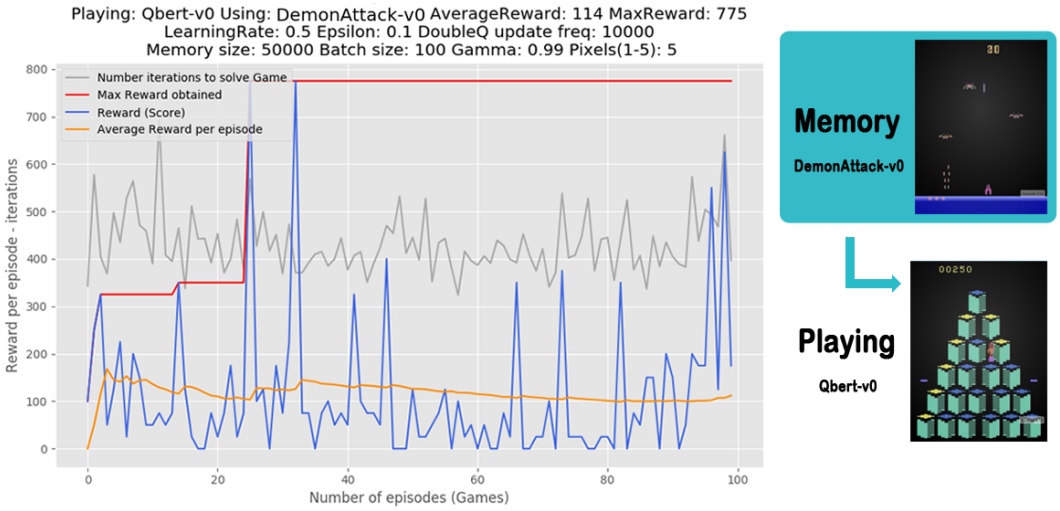

Fig 3.8 **Complex Negative Game Results**. Playing Qbert-v0 while using DemonAttack-v0 as memory initialization. Source: Python 3.0, Prepared.

However, the second subgroup was a surprised during research. Initially these positive results, Fig 3.9, were taken as strokes of luck. Fortunately, this changed when all the test showed the same results. Thus, there are components in the game that are not easy to see from a human eye, but that indeed exist in the environment.

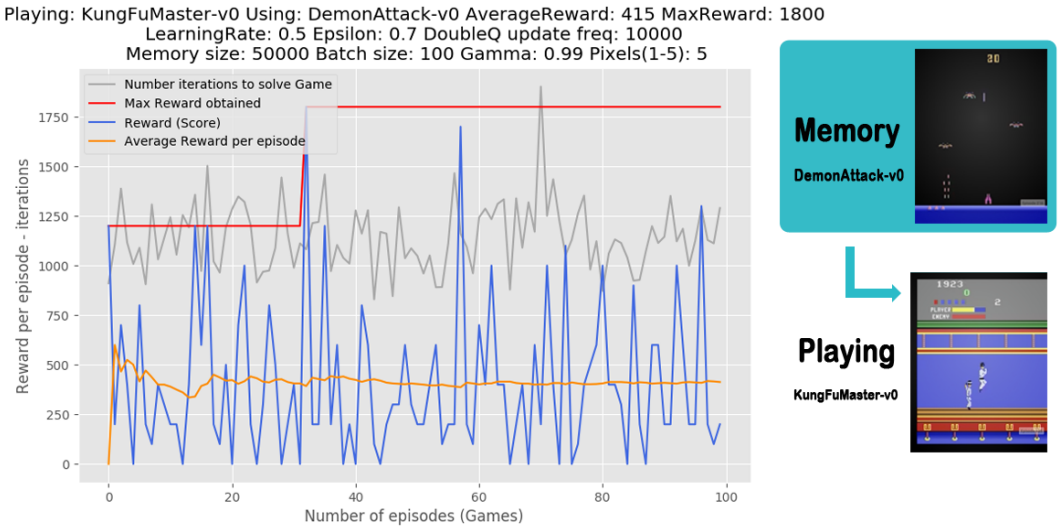

Fig 3.9 **Complex Positive Game Results**. Playing KungFuMaster-v0 while using DemonAttack-v0 as memory initialization. Source: Python 3.0, Prepared.

As a result, Fig 3.9 show a clear descent trend at the beginning, but them the results climb up again before reach 20 iterations, very quickly. This means that part of exported knowledge was useful, basically because the results stand stable since this climb. Thus, there are some correlations that the agent is using instinctively while taking actions that are not easy to see for the human eye. This fact worths a research in the future.

The main difference between the first negative subgroup (Fig 3.8) and the positive one (Fig 3.9), is the slope trend for "Average Reward per episode". Which is very stable for KungFuMaster-v0, whereas a negative trend is appreciated for Qbert-v0.

Even during the last iterations Qbert-V0 provide bad results. In fact, the slope never turns positive until episode number 95. This means that the information stored in the memory is completely useless for this particular environment. As a result, the algorithm needs to update almost all the memory samples stored to be able to provide good results.

4. Discussion Reflection, Conclusions and Future Work

This Memory-based Framework implementation has worked very effectively and have proved the starting hypothesis of the project. According to the experiment results, the combination of Double Q-Learning and Replay Memory methods seems to be the way to improve efficiency in Artificial Intelligence while exporting knowledge from one environment to another.

Therefore, as stated in the starting hypothesis, the results tends to be better when similar behavior architectures are implemented. In other words, when the objective game is played using a highly similar stored environment in the memory, the results tends to be better because of pattern similarity between states stored in memory and current states, obtained while playing the new game. As the situations are similar, the agent can interpret the patterns easily. Thus, the framework is able to understand behaviors such as "shoot and avoid get shot", commented at the beginning of this project.

Furthermore, the Pixel Density reduction used to improve both generalization and abstraction has been proven to be successful. The algorithm is able to export and fully employ information from previous environment and obtain good results. Even with games that are slightly different than the memory-based environment. As well as non-correlated or complex games. As a result, the Memory-based Framework is very helpful and it is still able to improve their performance including more cutting-edge methods, such as planning strategies using the available memory samples.

Moreover, the hippocampal models [1][16] have been a very good starting point for the idea of consciousness in Artificial Intelligence. Even not providing direct practical results, they have been the main engine for this framework implementation. The relationships between states formed by the blurred pixels images allow to generalize and bring abstraction while taking decisions, just like hippocampus in the human brain. These actions are taking instinctively by humans based on prior experiences that do not need to be highly similar between each other, but just a few. This is the main goal that this experiment has covered, the approximation to generalizable AI that somehow thinks as a human been, using consciousness.

Regarding Future Work, Experience Replay [17] or further memory-based implementations look like the right way to go, mainly if the main objective is keep focusing on consciousness, as memories are a fundamental part of it. Additionally, the discoveries obtained from the last section (3.3.3), show that Complex Games provide very good results by taking instinctively related actions that are not easy to see for the human eye, Fig 3.9. This is strongly related to neuroscience hippocampal models [1][16], and it worth enough for a future research in this field, maybe investigating how the convolutional patterns are seen inside the network.

Bibliography

[1] Blundell, C., Uria, B., Pritzel, A., Li, Y., Ruderman, A., Leibo, J.Z., Rae, J., Wierstra, D. and Hassabis, D. 2016, "Model-Free Episodic Control".

[2] Busoniu, L., De Schutter, B., Babuska, R. and Ernst, D. 2010, Reinforcement learning and dynamic programming using function approximators, CRC Press, Boca Raton, Fla;London;.

[3] Davidson, T.J., Kloosterman, F. and Wilson, M.A. 2009, "Hippocampal Replay of Extended Experience", Neuron, vol. 63, no. 4, pp. 497-507.

[4] Dulac-Arnold, G., Evans, R., van Hasselt, H., Sunehag, P., Lillicrap, T., Hunt, J., Mann, T., Weber, T., Degris, T. and Coppin, B. 2015, "Deep Reinforcement Learning in Large Discrete Action Spaces".

[5] Foerster, J., Nardelli, N., Farquhar, G., Afouras, T., Torr, P.H.S., Kohli, P. and Whiteson, S. 2017, "Stabilising Experience Replay for Deep Multi-Agent Reinforcement Learning".

[6] Hou, Y., Ong, Y., Feng, L. and Zurada, J.M. 2017, "An Evolutionary Transfer Reinforcement Learning Framework for Multiagent Systems", IEEE Transactions on Evolutionary Computation, vol. 21, no. 4, pp. 601-615.

[7] IEEE/IET Electronic Library (IEL), IEEE Staff and IEEE/IET Electronic Library (IEL) Conference Proceedings 2015, 2015 IEEE Conference on Computer Vision and Pattern Recognition (CVPR), IEEE, Piscataway.

[8] Klette, R. and SpringerLink eBook Collection 2014, Concise Computer Vision: An Introduction into Theory and Algorithms, Springer London, London.

[9] Krishnamurthy, V. 2015, "Reinforcement Learning: Stochastic Approximation Algorithms for Markov Decision Processes".

[10] Lim, S.H., Xu, H. and Mannor, S. 2016, "Reinforcement Learning in Robust Markov Decision Processes", Mathematics of Operations Research, vol. 41, no. 4, pp. 1325-1353.

[11] Mnih, V., Kavukcuoglu, K., Silver, D., Graves, A., Antonoglou, I., Wierstra, D. and Riedmiller, M. 2013, "Playing Atari with Deep Reinforcement Learning".

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

[13] Niv, Y. 2009, "Reinforcement learning in the brain", Journal of Mathematical Psychology, vol. 53, no. 3, pp. 139-154.

[14] OpenAI environments. Retrieved from https://gym.openai.com/envs

[15] Parisotto, E., Ba, J.L. and Salakhutdinov, R. 2015, "Actor-Mimic: Deep Multitask and Transfer Reinforcement Learning".

[16] Pritzel, A., Uria, B., Srinivasan, S., Puigdomnech, A., Vinyals, O., Hassabis, D., Wierstra, D. and Blundell, C. 2017, "Neural Episodic Control".

[17] Schaul, T., Quan, J., Antonoglou, I. and Silver, D. 2015, "Prioritized Experience Replay".

[18] Sutton, R.S. and Barto, A.G. 1998, Reinforcement learning: an introduction, MIT Press, Cambridge, Mass;London.

[19] Van Hasselt, H., Guez, A. and Silver, D. 2015, "Deep Reinforcement Learning with Double Q-learning".

[20] Wang, Z., Schaul, T., Hessel, M., van Hasselt, H., Lanctot, M. and de Freitas, N. 2015, "Dueling Network Architectures for Deep Reinforcement Learning", .

[21] Zhu, R., Zeng, D. and Kosorok, M.R. 2015, "Reinforcement Learning Trees", Journal of the American Statistical Association, vol. 110, no. 512, pp. 1770-1784.