Exploring Learning Techniques for Playing Atari Games

Malandrakis Eftychios-Angelos
student-id: 35363431

Abstract

The aim of this project is to explore and analyze different techniques, on how an agent can learn how to play more complicated games, using reinforcement learning methods. The main focus of this research will be Atari's video game Ms Pac-Man, as it is considered one of the most complicated games for an AI agent, because of the many completely different states that it may has.

1 Introduction

Reinforcement learning, provides a learning framework for algorithms that are as close as possible to the human way of learning from experiences. However, for real world problems, that task of reinforcement learning is still very challenging, as the algorithms needs to represent sometimes a very big environment from high-dimensional inputs, while they need to generalize these inputs of past experiences for new situations. In this project we will work, on developing an algorithm that plays Atari's 2600 Ms Pac-Man, one of the most challenging games for the AI algorithms. The reason why Ms Pac-Man is so challenging, is because there is a huge amount of different states, where a small difference in rather similar states, can lead in a very different optimal action. To build the algorithm, we use python's openAI Gym module, where it simulates the environment of the Atari games that we will use, returning the games' ram for each state. Then, we use a deep neural network that it is trained from past experiences and it predicts new states.

2 Similar Work

Even though there are many research works related with reinforcement learning applications for playing Atari games, however there isn't a lot of work related to the game Ms Pac-Man. As we previously stated, Ms Pac-Man is a video game with many different states, that it's too difficult for an agent to learn how to identify similarities and differences between them.

The most related work is from Mnih et. al 2013 [2] and Mnih et. al 2015 [3], where they introduced the idea of the Deep Q-Network. In these papers, among other games, they also tested the DQN on Ms Pac-Man, where after running for 50 million frames, it achieved 2300 score. The idea behind DQN is to approximate the Q-function with a deep neural network. Since it was published, there has been a lot of attention on the DQN method and many improvements have been proposed.

Also, a year later Hado et. al 2016 [4] introduced the Double DQN, while it achieved greater scores for most of the games compared to DQN, Ms Pac-Man was among the very few games that the original algorithm performed better. The idea of the double DQN is to use different Q-values, to evaluate and select an action from a specific state, using two different DQN with different weights. This is proven to help in many cases the overestimation of Q-values, which is a known issue of DQN.

Deep Recursive Q-Network is another proposed improvement, which is a combination of a long short-term memory (LSTM) and a Deep Q-Network. This is designed so that it can handle the loss of information in a better way than the original DQN [5]. One more highly promising improvement is the prioritized experience replay, which sets priorities for the actions stored in the memory. Then converting these priorities into probabilities,

it chooses which actions to use to train the neural network [6] [7].

All of these improvements of the original DQN algorithm, have already been proven to improve the performance on many Atari games. In this project we will implement a DQN to solve Ms Pac-Man and we will work to identify why is it considered among the most challenging games for reinforcement learning algorithms.

3 Methods

3.1 About Deep Q-learning

Ms Pac-Man and most of the Atari games can have a huge number of different states. As it is not possible for an agent to pass through all these states several times, to learn the best actions for each state, we have to build an approximation function that it will be able to approximate future states and predict the best action for each state. To do this we will use a novel method called "Deep Q-Network" that was proposed in 2015, from Google's DeepMind research center. Deep Q-Network (DQN) combines a neural network, with the idea behind Q-learning, where an agent passes through different states multiple times, using and updating a Q-value function, which provides the rewards of different actions for each state. Each time the agent takes an action from a specific state and it receives its reward (or punishment), it updates the Q-value function, using the bellman equation:

$$Q(s,a) = r + \gamma * \max\left(Q(s',a')\right)$$

Where, $\boldsymbol{Q}$ is the $Q - function$ which gives us the reward of an action $\boldsymbol{a}$ for a state $\boldsymbol{s}$. $\boldsymbol{r}$ is the current value of the Q function for the state $\boldsymbol{s}$, given an action $\boldsymbol{a}$. $\boldsymbol{s}$ and $\boldsymbol{s'}$ are the current state and the future state respectively, according to the action $\boldsymbol{a}$ that was taken. $\boldsymbol{\gamma}$ which is called the discount factor, is a constant number between 0 and 1, that reduces the reward of the future actions, so that the Q-function can converge faster. In the case of DQN, the algorithm stores the data of the current state, the next state, the action taken, the reward and if the game is over. Then, at the end of each episode, it selects some random data, updates their Q-values with the Bellman equations and use them, to fit the model.

During the first episodes, the model is not able to predict any action, as it doesn't have any data yet. For this reason, we will let the agent play selecting random action, until we have 50000 actions in our memory. Then we will use the $\epsilon - greedy$ algorithm. $\epsilon - greedy$, is a simple way to let the agent explore new actions, while it also exploits the experience that it has gain so far. At the beginning of each episode we set a parameter $\boldsymbol{\epsilon}$, which is the probability given to the algorithm to select a random action. In our case, we originally set the parameter ϵ equal to 1, which means that the algorithm will select actions solely by chance. Then, at the end of each episode, we multiply the algorithm with a value slightly smaller than 1. In our training algorithms, we have set this value as 0.9995, while we also set a minimum value $\epsilon = 0.05$, that the algorithm may reach. Considering that in most of the cases we have trained the agent for 10000 episodes, it means that ϵ will reach 0.05 after approximately 6000 episodes.

3.2 Game Environment

For the game's environment, we will use python module "gym", from openAI. It is an environment that implements many Atari games, in framework suitable for reinforcement learning. For all of the games that we will train, we will get as state parameter, games' ram memory at 128 bytes. Of course, this isn't perfectly suitable for many cases, as we are not able to decode games' ram and hence its rather difficult to identify some states that we might needed to set a different reward. For example, when playing Ms Pac-Man, we are not able to identify when Pac-Man is eaten by a ghost, with the exception when the game is over. Therefore, in this case we are not able to set a negative reward that will "teach" the agent that it's bad to be eaten by ghosts. The same is also the case for a few other games. Game's ram memory is a sequence of 128 numbers ranging from 0 to 255. To improve model's performance, we will normalize it's values by dividing it with 255.

Regarding the main game that we will work in this project, the goal is to eat as many yellow and white dots as possible, without being eaten by ghosts. For each episode the agent will have three chances to complete its tasks. For playing the game, the first part is to set the right reward system for the agent and find the right parameters for the DQN algorithm. The rewards awarded by the game are 10 points for each dot eaten, 50 point for each "power pellet" and $100 * 2^n$ for the $\boldsymbol{n}$ ghosts

eaten, a few seconds after a "power pellet" ($n \geq 1$) . At first, we tried to train the game using only the rewards awarded by the environment. However, it resulted the agent wouldn't learn that it is bad to be eaten by ghosts (as there wasn't any punishment for it), and it was spending time by remaining in the same state, or by repeating paths. To improve this, we changed the rewards received by the agent, so that it will be punished with 150 points for game over and with 2 points for not receiving any reward. We have also tried to punish the agent with a much bigger penalty for game-over actions, such as -1500. However, this resulted to overfitting the approximation function. In terms of reinforcement learning, overfitting means that the agent will avoid, or it will take some actions, because he wrongly correlates them with actions he was awarded with a very high or a very low score. In this case, it would avoid for example reaching a place where it was previously eaten by a ghost that resulted to a game-over, even if there is no ghost nearby, for the exact state.

3.3 Hyperparameter Optimization

The next important task was to set the right parameters for the DQN model. In our case the hyper parameters are: The discount factor in bellman equation, the minimum value of the $\epsilon - greedy$ algorithm and the value that decays it for each episode, the size of the memory storing the actions, the number of episodes, or the total running time, the data points per episodes that are used to train the approximation function and the learning rate of the neural network.

Parameter	Tests	Best value
game-over punishment	-30, -100, -50, -1500	-150
NN learning rate	10^{-3},10^{-5}, 2.5 10^{-4}	10^{-4}
gamma discount factor	0.9, 0.93, 0.98, 0.99	0.95
epsilon decay	0.9975, 0.99975	0.99965
memory length	10^4, 5 10^4, 10^6	10^5
training batch size	256, 128, 64	32
NN Optimizer	RMSprop	Adam

Table 1: Test cases for selecting the optimal parameters for our algorithm. The optimal parameters were selected based on the final average score of the classifiers and the number of episodes needed to reach this score.

We didn't perform a systematic grid search, because the computational time would be too much, considering the system capabilities. Instead, we ran more than 40 different test cases, with at least 10000 episodes each, that helped us choose the most suitable parameters. Above is a table with the different tests implemented to choose the right parameters. The Neural Network used, is a 4-layer network with its hidden layers containing 512, 512 and 128 units for the respective hidden layers and using a rectified linear unit (ReLU), for each of its activation functions.

3.4 Memory Prioritization

To improve the way that the algorithm chooses the data that will train the model after each episode, instead of choosing the data completely randomly, we will use different priorities for each action. We define priority as:

$$p_i = (\delta_i + \epsilon)$$

where, $\boldsymbol{\delta_i}$ is the error of the transition $\boldsymbol{i}$ and $\boldsymbol{\epsilon}$ is constant to help us make sure that every transition will have a priority greater than 0. Usually the error is defined as:

$$\delta_i = \big(reward - \max\big(Q(s', a')\big)\big)^2$$

The squared difference between action's predicted and actual reward. However, in our case we will take as error the absolute difference of the two values, as we would like to avoid huge differences between the actions' priorities.

$$\delta_i = \sqrt{\big(reward - \max\big(Q(s', a')\big)\big)^2}$$

For instance, if the predicted reward would be -2, but instead it was a game over (-150 actual reward), the squared difference would be $152^2 = 23104$. Instead, using the absolute difference, it would be $|150 - (-2)| = 152$. Finally, we convert these priorities to probabilities, dividing all the priorities with their summation.

$$P_i = \frac{p_i}{\sum_k p_k}$$

These are the probabilities that will be used by the algorithm, to choose the data that will train the approximation model, after each episode.

3.5 Adaptive Rewarding System

Another way to help the algorithm converge faster to an optimal policy, is to adapt the actions' rewards, considering the performance of the whole episode. We introduce the normalized episode performance as:

$$epf = \frac{eps - avgs}{eps + avgs}$$

where, $\boldsymbol{eps}$ is the final score of the episode and $\boldsymbol{avgs}$ is the average score of the last 100 episodes. For a reward $\boldsymbol{r_{s,a}}$ awarded from a state $\boldsymbol{s}$, choosing an action $\boldsymbol{a}$, the reward that will be stored in the memory will be:

$$r'_{s,a} = r_{s,a} * (epf + 1)$$

$\boldsymbol{epf}$, will always have values between -1 and 1, so the reward boosting will be between -100% and 100%. However, this is recommended only for a number of episodes at the beginning of the learning process. For example, for a 5% or 10% of the total episodes. Otherwise, it's very likely that the values predicted from the approximation function, will not converge to an optimal policy.

3.6 Data Collection

For more complicated games, like Atari games, it's common to have high variance for different episodes' scores. For this reason and to avoid, misleading scores, we will test the agents for 100 episodes and we will compare the collected data. According to the original paper that introduced the DQN, for each game, the agent was trained for at least 1 day. In our case, most of the training periods were lasted between 3 to 6 hours in a consumer computer. Hence, we would expect that for longer training periods we would have a more accurate approximation function.

4 Results

4.1 Testing Cases

At first, we let the agent play for 100 episodes, selecting actions only by chance, to be able to compare our results with this score. The average score of these games was 218.

Then, we trained the model, using only the reward system awarded from the environment. After, testing many different parameters, we found out that setting up the minimum value of ϵ to 0.2, it would achieve an average score of more than 600.

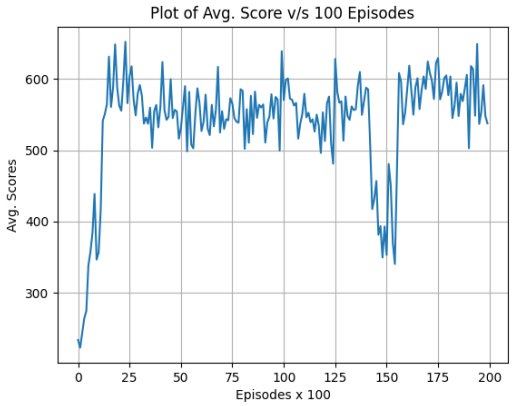

Figure 1: Performance plot for the training process of the first case. Each point represents the average score for 100 games.

However, when we actually tested the agent, it was always playing the same actions. That means, the agent was overfitted, and it was always selecting the same actions, without considering its environment. During the last training episodes, because of the 20% chance to select a random move, sometimes, it was selecting some good random actions, resulting to an average score of more than 600. When tested without doing any random actions, it was always achieving 215.

To improve the learning process, we defined that an action that would result to game-over, would be a -150 reward for the agent. Also, the agent would receive -2 punishment, if it doesn't receive any reward. Otherwise, it would receive the environment's reward. These changes, were aiming to teach the agent to avoid the ghosts and avoid repeating the same path, or stuck in a specific place. After testing many different cases to find the best parameters for the model, we ended up with a model that achieved an average score of 703. Figure 2 represents the performance plot for the second case, where each data-point represents 100

training episodes. It's also interesting, that it achieves almost 500 after less than 2000 episodes.

Next, we implement the memory prioritization and the adaptive rewarding system techniques. However, tests' results for both of these methods performed worse than the previous case. In the first case, it follows the same path every time, similarly with the first case, with a standard score 210, while in the second case performs slightly better with an average score 468.

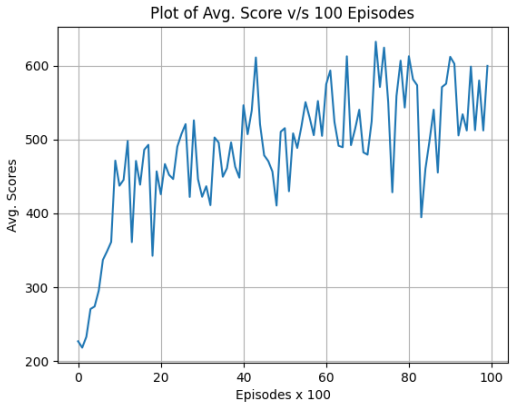

Figure 2: Performance plot for the training process of the second case, using a different rewarding system, which "punish" the agent for loosing the game and for repeating the same actions without achieving any reward.

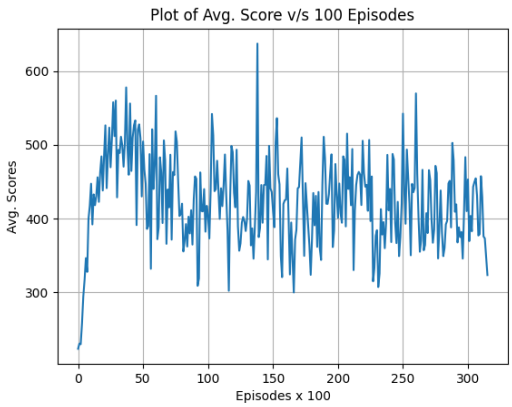

Figure 3: Performance plot for the training process of the third case, implementing the memory prioritization.

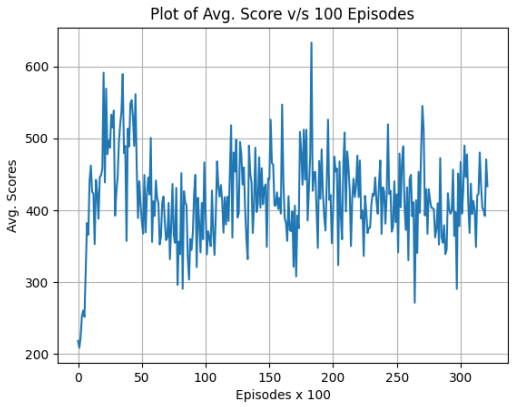

Figure 4: Performance plot for the training process of the fourth case, implementing the memory prioritization with the adaptive awarding system.

The above cases presented in the figures achieved very similar performance, with the second one having slightly higher average score both for training and testing. However, they still perform very poor compared to the second case (figure 2).

4.2 Comparison Between the Cases

For comparing the 5 different cases, first we will use a boxplot to compare their test episodes. From the figure 5, it seems that the second case it is the most interesting one, not just because of the highest median, but also because there are a few episodes that it achieved score more than 3500. Regarding the case 4, it's interesting that it achieved a few episodes with scores more than 1000 and it's first and third quartiles are very close to each other.

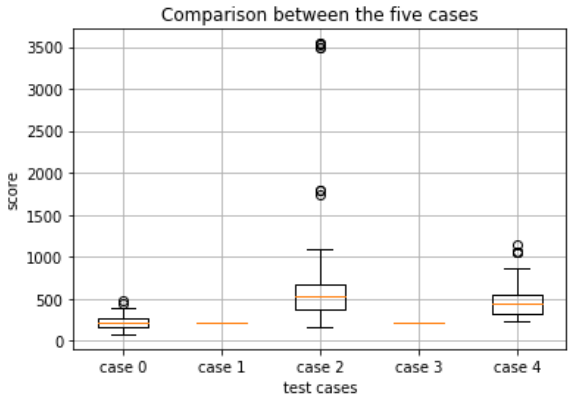

Figure 5: Comparison figure between the five cases. Boxplots whiskeys are equal to 1.5 IQR (IQR = Q3-Q1)

Next, there is a comparison table that presents the average score, its standard deviation, the upper

and lower quartiles and the maximum and minimum values of each test case. Also running a $t-test$ for the data of the cases 2 and 4, the $p-value$ received is 0.002. That means, that we have evidence to reject the hypothesis that the two datasets have the same mean value.

#	AVG	STD	MIN	MAX	Q1	Q3
0	215	72	80	480	160	262
1	210	0	210	210	210	210
2	703	708	170	3540	377	680
3	210	0	210	210	210	210
4	468	201	240	1150	320	550

Table 2: statistical values of the test cases. Average score, Standard deviation, Min & Max scores, First quartile, third quartile.

5 Conclusions

Building a reinforcement learning algorithm to train Ms Pac-Man turned out to be a very challenging task. First, because small changes on different parameters would result to have a huge impact on the final outcome, while there is no standard "best values" to choose (e.g. for longer training sessions a value closer to 1 would be more sensible). The second reason is because it's very challenging to train the agent with an effective way, but without overfitting. Overfitting is one of the biggest issues that reinforcement learning systems are facing when dealing with real world problems, as it is very easy for an agent to start following standard paths, not because they are chosen as the optimal actions for each state, but because it predicts them wrong, based on its limited memory. Also, we have seen that even though, the memory prioritization technique can lead to significant score improvements in many reinforcement learning algorithms, in the case of Ms Pac-Man, it's more possible to make the algorithm overfit faster. As we can see on the figure 3, during the training session, the algorithm reaches a point that achieves scores of more than 550, but then, the average score values start to decline. Regarding the adaptive rewarding system proposed, it indeed improves the learning outcome in the fourth case, however, there wasn't enough time to run more cases. Hence, we don't have enough evidence to support this hypothesis. Finally, regarding our choice to use game' ram instead of the output frames to train the classifier, it was most probably a bad decision. This decision had been taken, considering that training 128 features of games ram would be a better choice than training using 80x80px images (640 features). However, decoding game's ram to recognize patterns is most probably trickier for the algorithm, compared to analyzing game images.

6 Acknowledgements

The training algorithm was originally inspired by the book "Python deep learning projects" [8], but it was built and adapted for Ms Pac-Man from scratch.

7 Extra Materials

In the results section we have summarized five cases that turned out to be the most interesting cases for some conclusions. However, we have run many other cases for this project, most of them in order to specify the hyperparameters that would give us the best results. The link bellow includes all the training cases that we ran, videos of sample episodes for each of the five cases and videos with cases, using the same algorithms for training other Atari games out of personal interest. The videos with the names of the above cases presented will be equivalent trained with these algorithms.

https://cloud.anjelo.ml/index.php/s/5rMkQM88HY7Y5cW

8 References

[1] B. Sutton, Reinforcement Learning, The MIT Press, 2018.

[2] Mnih, "Playing atari withdeep reinforcement learning," *arXiv,* 2013.

[3] Silver, "Human-level control through deepreinforcement learning," *Nature,* 2015.

[4] Hado, "Deep reinforcementlearning with double q-learning," *arXiv ,* 2016.

[5] Hausknecht, " Deep recurrent q-learning forpartially observable mdps," *arXiv ,* 2015.

[6] S. Ravichandiran, Hands-on Reinforcement Learning with Python, Packt, 2018.

[7] S. Q. S. Antonoglou, "Prioritized experience replay," *arXiv ,* 2016.

[8] R. K. Mathew Lamons, Python Deep Learning Projects, Packt, 2018.