Reinforcement Learning and Decision Making Under Uncertainties - Snake Project Report

Carrel Vincent, Fontana Jonas

June 3, 2022

Abstract

Since the introduction of Deep Reinforcement Learning by DeepMind, the use of CNN to play games as been everywhere, both in publications and on the internet. It looks cool, but does it always works? When is the reward too sparse? While revolutionary, the idea is still difficult to apply to challenging scenarios where the agent is changing in form (the snake is getting longer) and the reward are rare (in the original game, a reward is only given when eating a target).

In this project, we build a new Gym environment reprsenting the famous Nokia game Snake. We implement 4 different algorithms (SARSA, Q-Learning, TreeSearch and Deep Q-Learning) and compare the performances of the different models in different arena. Then we pay a closer look at the capabilities of the Deep Q-Learning methods to extrapolate to new arenas. We discuss the shortcomings in the training setup and propose different avenue to consolidate the model and obtain an agent able to perform in different environment.

The source code of our program can be found at https://github.com/Fontanjo/RL_Project

Contents

List of Figures

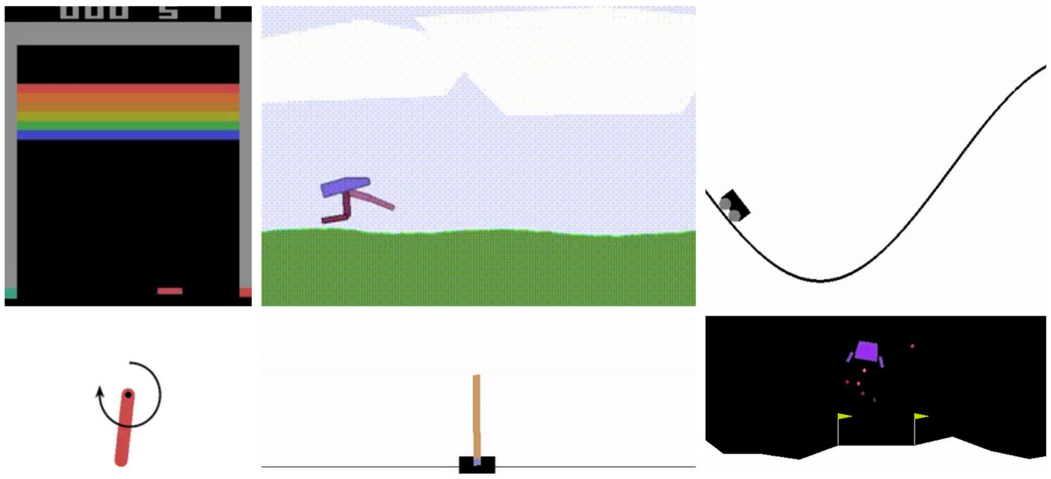


Figure 1: An example of gym environments ([Kathuria, 2021]).

1 Introduction

Over the last 15 years, the field of Machine Learning has seen major breakthroughs with the introduction of Deep Neural Networks. It gained a global attention when it was able to obtain state-of-the-art performance in almost every applications, should it be image recognition, pixel segmentation or NLP tasks. The first to apply the same concept to the Reinforcement Learning have been DeepMind [Mnih et al., 2013], when they were able to approximate the Q-Value function with their Deep Q-learning network (DQN) and trained agents capable of solving multiple Atari games.

We continue on their footstep and build a new Gym environment of the well-known game Snake. We created two different versions, one with a limited state space, where the information is not complete, but it allows us to use classical methods such as Q-Learning or SARSA. The second one return the raw representation of the game, a (n x n) pixel matrix which is too big to be treated with traditional methods. We apply Deep Q-Learning to this environment, and compare the performance against the baseline methods working in the reduced state space. In the second part, we analyse the effect of the arena, by implementing different type of boards, and compare the snakes behaviors for all methods.

2 Gym environment

Gym ([Brockman et al., 2016]) is a toolkit created by OpenAI to facilitate development and comparison of Reinforcement Learning algorithms. It provides a variety of environments, ranging from board games to Atari games, as well as 2D and 3D (robot) simulators.

In addition, gym offers the possibility for the user to create a custom environment. The requirements are minimal, and a new environment should expose at least the following four methods:

- A constructor (__init__()) to instantiate the environment
- A reset() method to reset the environment to the initial state
- A step() method, which takes as input an action and return the new state, together with a reward and whether or not the simulation has finished
- A render method, which shows the environment to the user

A custom class satisfying this requirements can easily be built into a gym environment and used to develop/compare RL algorithms. Figure 1 shows an example of existing gym environments.

2.1 Environment requirements

The first part of the project consisted in the creation of our custom environment. Even though no constraints where posed, we wanted an environment that could be solved by a classical RL algorithm (e.g. Q-Learning, SARSA, etc.) with "decent" results, but at the same time required a more sophisticated approach for an optimal solution. A straightforward way to find such problems is to consider environments where the observation space is very big, but can be approximated with a smaller number of states (usually sacrificing the performance).

Let's consider an example. Imagine you have to train an agent to play blackjack. The initial situation consists of 2 cards for the player and 1 visible plus 1 hidden cards for the dealer. Even if we ignore the hidden card, we have 132'000 combinations (in a standard deck there are 52 cards). And this considering only the initial round (in the following more cards are distributed). However, we can reduce this number thanks to some considerations:

- Having a queen of spades or a queen of hearts does not make any difference. Nor does it having a king or a jack (all figures have the same value in blackjack)
- The order in which I receive the first 2 cards is also not relevant, since the game only starts after I received both

This already greatly reduces the number of possible states. In addition, we could also discuss whether considering both cards is useful, or if instead it would be enough to consider the sum. In fact, the final score of a player is determined by only the sum of its cards. So having a 10 and a 8, or two 9, is the same. However, knowing which cards already appeared slightly changes the probability of the next card, possibly impacting the best decision (consider the case where you need a 1. If all four 1s have already appeared, you will probably not ask for another card, since it is impossible that you get what you want).

In this case, we can reduce the number of states if in exchange we accept a loss of information, lowering the upper bound of the performance of our algorithm.

It is also to notice that the process of reducing the number of states and thus simplifying the problem is done by the developer, and not from the algorithm itself. The resulting method is thus not directly generalizable without further human interventions.

2.2 Snake environment

We decided to implement an environment simulating the classical game "snake", which became famous in 1997 with the mobile phone Nokia 6110. However, the idea originated already in 1976, in the game "Blockade" developed by the British company Gremlin Interactive ([Krishnankutty, 2020]).

Snake is a simple game in which a small snake moves on a board. The goal of the snake is to eat some fruits that appears on the board, and at the same time avoid collisions with itself or with the walls. Every time the snake eats, its tail grows, making the game harder (it gets more likely to collide with itself).

We can very quickly see that in the original setting, the possible states are too big in order to solve the problem with a classical RL algorithm. Already with a 10x10 board, each tile might contain a wall, a piece of the snake, or be empty. This means $3^{10\cdot 10} \approx 5 \cdot 10^{47}$. And we still have to consider the head position and orientation, as well as the target (fruit) position. This is clearly not feasible except for very small boards.

To tackle this problem, there are two possible ways. The first one is to use a more advances algorithm, for example using a convolutional neural network (CNN) as feature extractor. But this is not a concerning of the environment (meaning that we don't need to modify the implementation in order to allow it). The second alternative is to reduce the amount of information, as shown in [sid sr, 2020]. In particular, the environment only returns a state determined by the following elements:

- The target position w.r.t. the snake head (can be in front, front-right, etc.)
- The presence of an obstacle in the 7 tiles surrounding the snake head (the tile behind the head is not considered, as there is always an obstacle - the fist piece of body!)

In this way, we drastically reduce the number of possible states. This doesn't come for free: firstly, we only know the target direction, but not the distance; second and more important, we have only a restraint visual on the board. This results in the impossibility of avoiding dead ends (e.g. when the snake surrounds itself).

In the following, we describe the various alternatives implemented and offered from our environment when instantiated. A more precise description can be found in the README file of the environment.

States: For the state, we implemented both the option discussed above. The state mode 'matrix' returns a 2D array representing the board, including the walls, the head (and its orientation, with a different value for up/down/right/left), the body parts, and the target. The state mode 'states' returns instead an integer, encoding only the direction of the target (relative to the head position and orientation) and the presence of an obstacle in the tiles surrounding the snake head. The possible values for this mode ranges from 0 to 1023, making the application of a classical RL algorithm fully feasible.

Rewards: For the rewards, we implemented three different options. The first option, that we call 'normal', gives a positive reward whenever the snake eats the target, and a negative reward when it dies (i.e. collides with a wall or with its body). Optionally there can also be a rewards for each step the snake does without dying, but by default this is 0. In this setup, the rewards are very sparse, only appearing when the snake finally eats a target, or die. Most actions have a reward of 0, and we can only know "after the fact" (if a target was eaten or if the snake dies) it the last few actions were good or not. In the 'extended' reward mode, we also give positive rewards when the snake takes moves approaching the target, and negative rewards when it takes moves taking it away from the target. This should serve as intermediate rewards, making it easier for an algorithm to learn a good policy. Finally, we added a third option, 'adaptive', which is similar to the 'extended' mode, but the absolute value of the reward given when moving towards and away from the target is linearly decreased with respect to the size of the snake. The reflection behind this mode is that the longer the snake get, the more complicated it is to perform evasion moves, avoid blocking itself. It becomes less important to go "directly toward the target".

Actions: At every step, there are only 3 possible actions: continue straight forward, turn right, turn left. The fact that they are relative to the head orientation is the reason why this last variable is not encoded in the 'states' mode (and why it is not necessary). For debugging (and entertaining) purposes, we implemented the possibility to play in what we call 'human mode', where the human can play controlling the snake with the keyboard arrows. To facilitate it, in this case the arrows direction are the classical ones (up for going up, right for going right, etc.), and not relative to the head orientation anymore. The environments takes case of converting it to the right action, simplifying the user experience (invalid actions, such as trying to go immediately down while going up, are simply ignored).

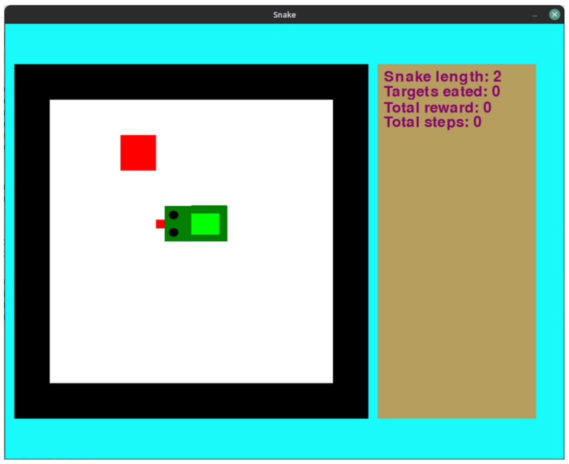
Shape: The environment not only proposes the classical "square arena", but we also added three other possibilities, to compare the performance of the different methods in varying environments. The "double_v" and "double_h" shape arena are the same classical square arena, but a line of wall is crossing vertically/horizontally the whole terrain, leaving only a small 1-2 wide gap in the middle to go from one half of the arena to the other. The "Shuriken" shaped arena is a combination of both, walls are reaching, from the 4 cardinal directions toward the center, separating the arena and leaving only a small bottleneck in the middle. Figure 2 shows the four shapes (in all cases with a 10x10 board and solid borders).

3 Algorithms

3.1 Baseline

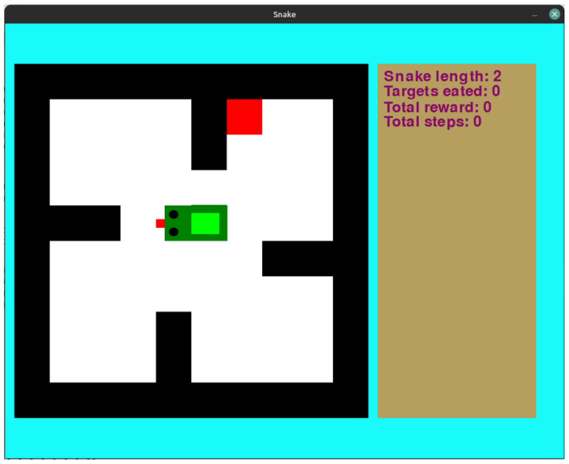
In order to evaluate whether the effort of developing a more complex algorithms is worth, we first test some of the classical RL algorithms to have a baseline (a minimal performance to use as comparison).

Q-Learning: Q-Learning is one of the simplest and most well known algorithms in reinforcement learning [Dimitrakakis and Ortner, 2022]. It is an *off-policy* algorithm, meaning that the values are learned independently from the policy followed. The algorithm maintains a table with the value of each (state, action) combination. At every step, the agent performs an action, receives a reward, and updates the table according to the Bellman equation:



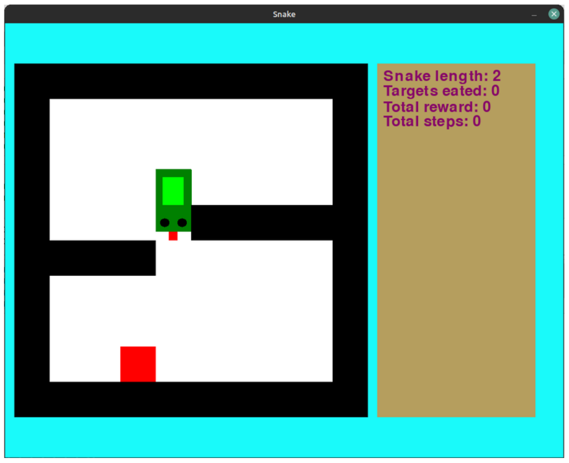
```
Snake length: 2
Targets eaten: 0
Total reward: 0
Total steps: 0
```

(a) Normal shape



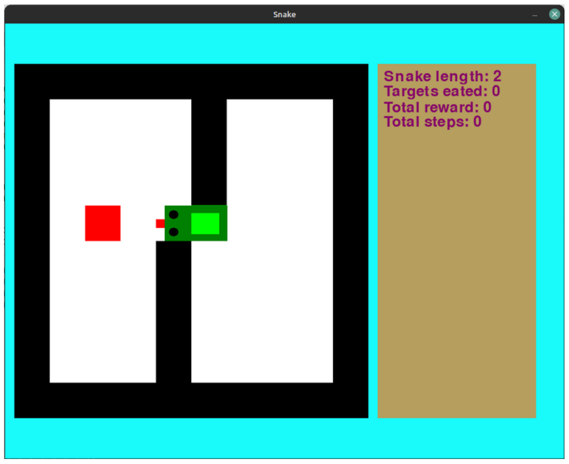
```
Snake length: 2
Targets eaten: 0
Total reward: 0
Total steps: 0
```

(b) Shuriken shape



```
Snake length: 2
Targets eaten: 0
Total reward: 0
Total steps: 0
```

(c) Double_v shape



```
Snake length: 2
Targets eaten: 0
Total reward: 0
Total steps: 0
```

(d) Double_h shape

Figure 2: Example of possible shapes for the Snake environment (10x10 board and solid borders)

$$\underbrace{Q(s,a)}_{\substack{\text{New}\\\text{Q-Value}}} = \underbrace{Q(s,a)}_{\substack{\text{Current}\\\text{Q-Value}}} + \underset{\substack{\text{Learning}\\\text{rate}}}{\alpha} \Big[\underbrace{R(s,a)}_{\text{Reward}} + \underset{\substack{\text{Discount}\\\text{rate}}}{\gamma} \overbrace{\max_a Q(s',a)}^{\substack{\text{Maximum predicted reward, given}\\\text{new state and all possible actions}}} - Q(s,a) \Big]$$

SARSA: The name SARSA comes from the elements considered at every step from the algorithm: current state S_t, current action A_t, reward given for the current state and action R, next state S_{t+1}, and next action A_{t+1}. The algorithm is very similar to Q-Learning, but instead of choosing the next action to maximize the Q-value, we chose it according to the current policy. This is why SARSA is an *on-policy* algorithm. The update equation of SARSA is the following:

$$\underbrace{Q(s,a)}_{\substack{\text{New}\\\text{Q-Value}}} = \underbrace{Q(s,a)}_{\substack{\text{Current}\\\text{Q-Value}}} + \underset{\substack{\text{Learning}\\\text{rate}}}{\alpha} \Big[\underbrace{R(s,a)}_{\text{Reward}} + \underset{\substack{\text{Discount}\\\text{rate}}}{\gamma} \overbrace{Q(s',a')}^{\substack{\text{Predicted reward, given}\\\text{new state and action}}} - Q(s,a) \Big]$$

TreeSearch: Since the environment is almost completely deterministic (except for the placement of the new targets), a much simpler and intuitive approach is to simulate at every step all the possible actions, and choose the one which gives the best reward. But most of the time all the three possible actions will give the same reward (nothing), since not at every step you can eat a target (or die). Therefore, the next logical step is to "look further" in the possible actions, meaning that for every possible reachable state, you consider the states reachable from there, an so on. This is equivalent at building a tree of actions, where the first path might be something like "go left - go left - ... - go left - go left", the second "go left - go left - ... - go left - continue straight", and so on. Since at every step we have 3 possible actions, to "inspect" the next N steps we will need to evaluate 3^N paths, and this before taking every single choice. Even though this method could potentially solve the problem of being trapped by foreseeing the danger, its complexity severely limits its applicability.

3.2 Convolutional Deep Q Learning

Both SARSA and Q-Learning works on the reduced state space composed of 1024 different situations, and therefore have only a limited information about the current situation. Intuitively, these models should lead to a "reach the target asap" strategy, and will lack the global understanding of the board to realise when a turn might lead them into a dead-end. These simple behaviors should become less and less reliable as the snake gets longer. Depending on the situation, it might be beneficial to first do a detour before reaching the target to avoid getting stuck, but the global view necessary to take such decision is lacking in these models. On the other hand, TreeSearch has severe limitation on the exploration depth. Even though it might still be a good solution for a board of medium size, it becomes much less efficient in a bigger board where the target might be too far from the snake to be spotted. In both cases, we need to find another method.

The first approach could be to encode all the possibilities into the state space, but it quickly becomes uncomputable. For a 10x10 arena, we have 100 square which could all take 8 different values (Empty, Wall, Head (4 different directions), Body, Target), so 8^{100} states to encode. This represent way too many states to be solved by a classical Q-Learning or SARSA approach. Instead, we use the Deep Q-Learning approach[Mnih et al., 2013] and train a model taking directly the raw data as input. Our hypothesis is that such agent should have enough information about the complete situation to detect and avoid the pitfalls described before.

3.2.1 Model

For the Deep Q-Learning approach, we are using the states mode "matrix", where the state is a full 10x10 matrix. Each pixel contain a different number depending on what is present in that position: a wall, nothing, the target, the body or the head of the snake. Each head direction is encoded as a different number, to simplify the CNN architecture. We don't need to process multiple following frames to infer the direction of the snake, and can directly work on a single snapshot of the current environment. Considering that each convolution has a tendency of "blurring" some small details, we decided to double this matrix, and use a (20 x 20 x 1) array as input to the CNN. It ensures for example that the target is a (2x2) square instead of being a single pixel. We are not sure about the relevancy of this trick, and considering the time it takes to train each model, we could not perform an ablation study on that approach specifically. Based on different literature and previous experience with CNN [Mnih et al., 2013], the architecture is composed of two convolutional layers (a third one would reduce too much our input image which was already quite small, a second justification of the initial doubling), a flattening layer leading into an hidden fully-connected layer with 128 neurons. All these layers use ReLu as activation function, and the output layer, a fully connected softmax layer, output the action to take (continue, left or right). Two models are used, the "Q-model" and the "target-model", which is used to estimate the Q-values of the next states.

4 Experiments

In this section, we present the technical details of the algorithms and the experiments conducted.

4.1 Baseline

We trained both Q-Learning and SARSA on each of the three environments "Classic", "Shuriken" and "Double_v". In all cases, we trained the algorithm for 100'000 iterations, and every 1'000 iterations we draw 5'000 random past steps and play them again (replay buffer). We always started with an epsilon of 1 (meaning that the algorithm only takes random choices), and decrease it at each iteration in order to reach the minimal value of 0.1 after 70'000 iterations.

For the TreeSearch, as mentioned in 3.1 the main problem is the complexity. This can be partially faced using multithreading, since every path of the search tree is independent. However, creating a new thread entails a certain overheat, and in addition it makes no sense (and is even harmful) to generate more thread than those that the machine can afford. After some tests, we determined that the best choice was to create new threads for the first 2 levels (i.e. for each of the first 3 choices, and than again for the 3 subsequent choices in each case). This creates a total of 12 threads (which could be reduced to 9 by reusing the first 3). Then, for each combination of the first 2 choices, additional 5 choices are inspected (for a total depth of 7). Thus, before taking a choice, 2187 different developments of the game are inspected. This might not seem much, but it should be considered that in order to simulate the next move, the program must first copy the entire environment. By doing this, the average time per move is more than 0.25s (against less than 0.0002s of SARSA and QL, once trained). If we were to go deeper in the search tree, the time required would be too much to be useful for the real snake game (where a default move is taken if no input comes before a given threshold).

4.2 CNN method

We trained three different models of the same architecture, once in each of the following environment: "Classic", "Shuriken" and "Double_V". Each model is trained for 50'000 iterations. During the first 5'000 epochs, the action is totally random to explore at maximum, then the action is selected by an epsilon greedy algorithm decaying from 1 to 0.0002 over the next 10'000 epochs. Finally, we set the epsilon at 0.0002 during the rest of the training, to ensure that it is almost always the optimal action selected, but we still have a chance to break out of a loop where the snake would be stucked. Each action is stored with its reward, corresponding state and future state in the replay buffer, which can contains at max 1'000'000 tuples (s_t, a_t, r_t, S_{t+1}). Every 4 actions, we update the Q-model by sampling a batch of 64 tuples randomly in our replay buffer, estimating the Q-value with our target-network, and performing gradient descent following the Deep Reinforcement Learning theory. The tuples are sampled randomly to avoid the high correlation between tuples if we would only consider the last x frames. Every 10'000 actions, the target-model is updated simply by settings its weights to the current weights of the Q-model.

When looking at figure 3, we can see that the model is still steadily improving over time. Unfortunately, we could not train a new model for longer due to limited computation power. Each of

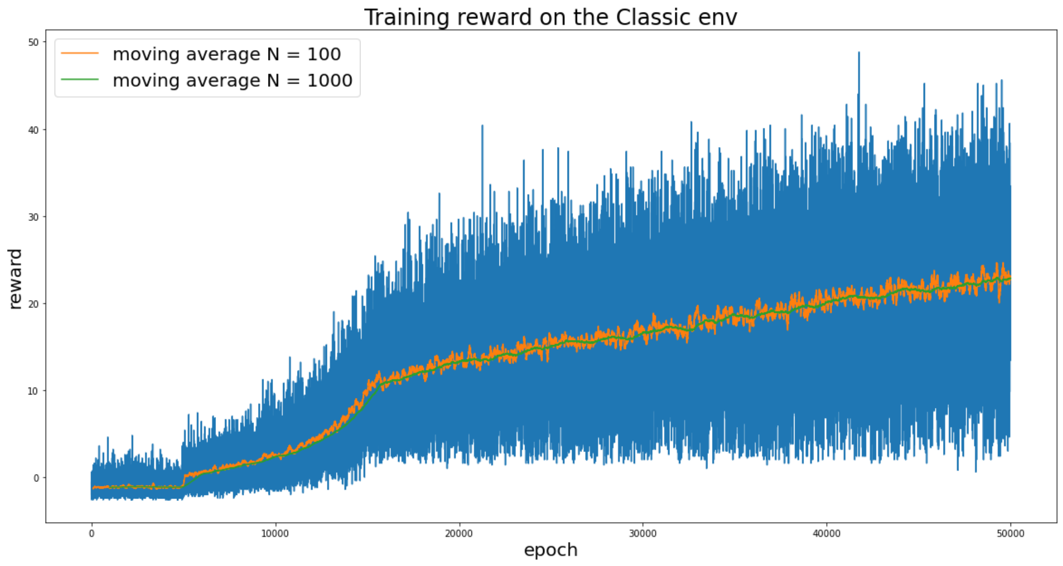


Figure 3: Average reward of the model training in the Classic environment

the models took approximately 40h to train on a single machine. Some shortcomings in the training setup were spotted during the different results analysis. The replay buffer containing all (most) of the past action, the huge majority of them will be actions performed in the beginning of the game, when the snake is still small. By sampling randomly over the whole history, we end up training our model mostly on the same repetitive frames, and while the model is "specialized" in the early game (almost always reliably reaching a correct score), it is extremely difficult to train the rare situations when the snake get longer and longer, and we end up in a negative loop: not performing well when long $\rightarrow$ not many saved experience while being long $\rightarrow$ no training done for the long situation $\rightarrow$ not performing well when long. When discussing possible fixes, we argue it could be massively beneficiary for the training to begin the game with the snake measuring a random length. By doing so, at least in the beginning, we'd ensure that enough different experiences are present rapidly in the replay buffer, and every situation would be sampled regularly. The second idea would be to implement two (multiples?) replay buffer, each storing different experience. The separation could be done on the length of the snake. The sampling would ensure that it picks enough tuples from the "long" replay buffer, so each time, the model sees enough "long snake situations" to get better not only at the beginning of the game. The threshold for storing an action in the corresponding replay buffer could be based on the average performance over the last X iterations, such that our "long" replay buffer would always contains situations which are better than the current average performance, and therefore rare.

Figure 4 clearly shows the different steps of the training. The first 5'000 steps where the actions are random, then the steep increase where the reward seems to be limited only by the action being random "too many times", nerfing the model, and finally the linear improvement. Both models trained in the "Double_v" and in the "Shuriken" environment seems to have reached their asymptotic limit, with the "Double_v" apparently being the most difficult environment to train in.

5 Results

In the following section we present the results obtained using the configurations mentioned in 4.

5.1 Performance

First we compare the performance of our CNN model with the different baselines.

We run the different models for 200 games (exept for TreeSearch, which we run for 50 games only since the simulation requires much longer, and considering also that the results are much more stable), and we record the mean and the standard deviation of the number of targets eaten during each runs (considering the "score" obtained in the official game is the number of target eaten before dying). When

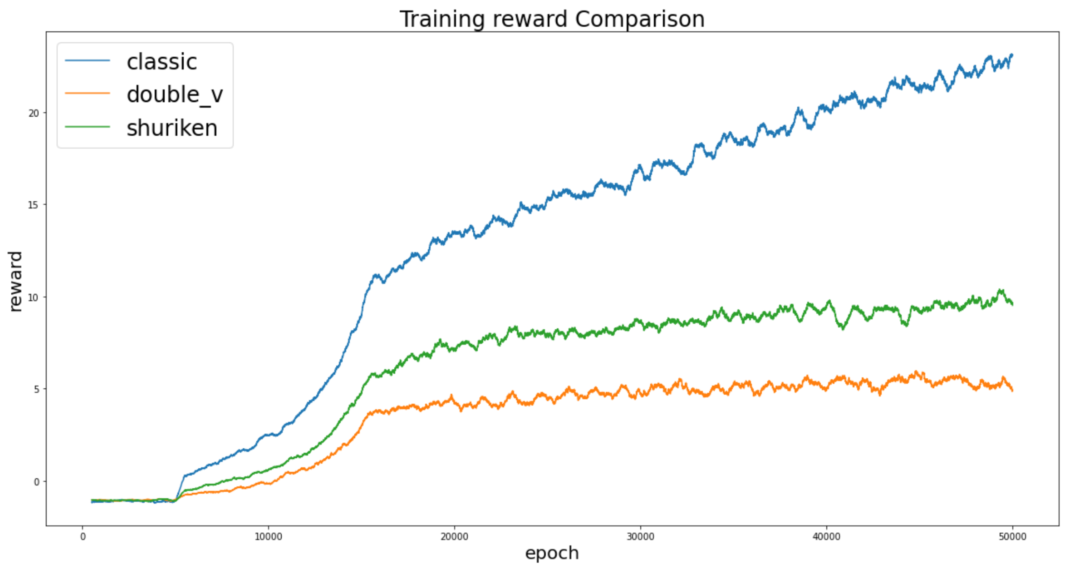


Figure 4: Average reward of the models training in the different environments

focusing first on the classic setup (model trained and tested in the "Normal" arena), we can see that all results are quite similar (except for TreeSearch, which outperforms the other methods but requires several order of magnitude more time before every decision). Depending on the runs, the average is around 15 for all models, but we noted that both SARSA and Q-Learning were constantly having a maximum run way higher than the Deep Q-Learning, in the 30-35 range versus Deep Q-Learning who was never able to beat 22-23. The disparity in the maximum values, coupled with the standard deviation being twice smaller for the Deep Q-Learning compared to the other makes us believe that the DQN model is way more stable, robust, and can constantly reaches a decent performance, in contrast to the others models being way more volatile and sensitive to the target's position randomness. It is also interesting to note that all the models operating on the restricted states spaces are not really affected by the different setups. For Q-Learning and SARSA, the performances in the different environments are similar, no matter on which environment the model was trained. It is because the model doesn't not even know the arena is shaped differently. With the information available, it is not capable of realising that the walls are placed differently, it always only focus on the short field of view around the head.

This is to put in relation with the discussion about the training setup. With the current replay buffer, we are the training the model to be a "specialist of the early game", which he seems to achieve, with arguably better performance than the baseline methods during the first few targets (smaller standard deviation for a similar mean → bad runs are more rare). Considering that the model was still far from converging, combined with the possible amelioration to the training setup, we believe it should be possible to increase the effective range of the model, and obtain the same kind of robustness (standard deviation < 4) for a way higher average reward. It is obviously impossible to predict up to where, but the results are encouraging to pursue in this direction and maybe overtake by a large margin the current baselines on this challenging environment.

		Training env		
	# targets	Normal	Shuriken	Double_v
Test env	Normal	16.9 ± 6.5	17.1 ± 6.1	15.5 ± 6.1
	Shuriken	6.4 ± 2.3	7.4 ± 2.7	6.5 ± 3
	Double_v	4.6 ± 3.1	6.4 ± 3.3	7.6 ± 5

Table 1: Mean (and std) of targets eaten using **QL** given the training environment and the test environment.

		Training env		
	# targets	Normal	Shuriken	Double_v
Test env	Normal	14.9 ± 6.2	12.6 ± 5	16.3 ± 7
	Shuriken	4.7 ± 2.5	5.8 ± 2.7	6.8 ± 3.4
	Double_v	4 ± 2.9	6.6 ± 3.7	6.3 ± 4.3

Table 2: Mean (and std) of targets eaten using **SARSA** given the training environment and the test environment.

		Training env		
	# targets	Normal	Shuriken	Double_v
Test env	Normal	27.6 ± 7.4		
	Shuriken		12.6 ± 2.4	
	Double_v			16.2 ± 3.6

Table 3: Mean (and std) of targets eaten using **TreeSearch** for each environment. Since this method has no training part, it does not make any sense to "test on a different environment". It is to notice that this method requires up to 0.35 seconds for each move.

		Training env		
	# targets	Normal	Shuriken	Double_v
Test env	Normal	13.6 ± 3.3	5.5 ± 2.9	3.9 ± 2.4
	Shuriken	0.5 ± 0.8	7.4 ± 0.7	0.3 ± 0.9
	Double_v	0.4 ± 0.9	1.2 ± 3.1	4.1 ± 3

Table 4: Mean (and std) of targets eaten using **Deep Q-Learning** given the training environment and the test environment.

5.2 New environment

In the second part of the analysis, we wanted to analyse the impact of the arena on our different models. What would append to a snake trained in the classic square arena if we let it go in a differently shaped arena? Are all models affected the same? We also compare the ability of our models to generalize to other arena.

Before conducting the experiment, our assumption was that the Deep Q-Learning would clearly have been the best. The CNN component in the model should allows to extract and understand that a wall present in the middle of the arena is exactly the same, and should lead to the same escape route, than a wall on the border. On the contrary, we were not sure at all about the performance of the other models, considering that they only have partial information. Multiple situations could actually looks similar in its restricted field of observation, for example a wall directly in front but the target in the wall's direction, is it because of the tail of the snake, a "middle" wall, or is it on the side, ultimately it doesn't really matter. The action should still be similar. That is why our analysis is mostly focused on the (disappointing) performances of the Deep Q-Learning model in the different arenas.

Our first assumption that the Deep Q-Learning model should be able to extrapolate well was completely erroneous, as seen on figure 5. The model trained in the "Classic" environment is extremely bad in all the other terrains, averaging less than 1 target per runs. Our hypothesis is that the model actually doesn't really consider the walls on the sides, and simply learn to avoid taking an action going toward the outside of the arena, but it doesn't link the act of dying with the action of moving in a wall, but rather with the act of having the snake going outside. So when faced with walls in the middle of the arena, it doesn't really take them into account and end up dying really early.

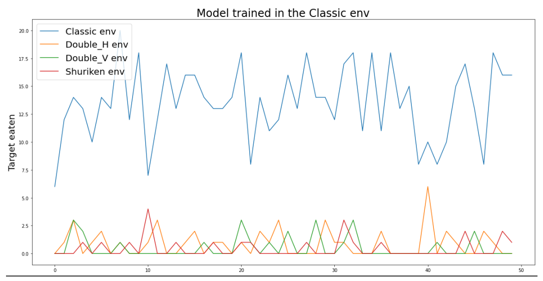


Figure 5: Classic Model performance

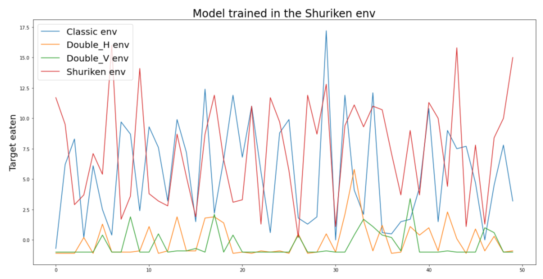


Figure 6: Shuriken Model performances

On the contrary, the model trained on the Shuriken arena is the best (or the least worse, its performance are definitely not "good") at extrapolating and obtaining decent result on other shape. As seen in figure 6, the model obtains almost similar performance in the classic environment, a completely new setting for him, to his performance in the original Shuriken shaped arena. The model trained in the Double_V arena is similar, with having its best performance in its original shape, and a decent showing in the easier Classic setup. While we were expecting the Double_V model to perform really well in the Double_H setting, considering that a convolution should be able to treat similarly a vertical or horizontal wall, that is not the case at all. Figure 7 confirms that none of the models are working in the completely new situation. The Shuriken model seems to be slightly better than the others, and considering the other results, it is coherent as it is the only model which also faced some horizontal separations during the training, but it is hard to say if that is really the case or if it was just some lucky runs.

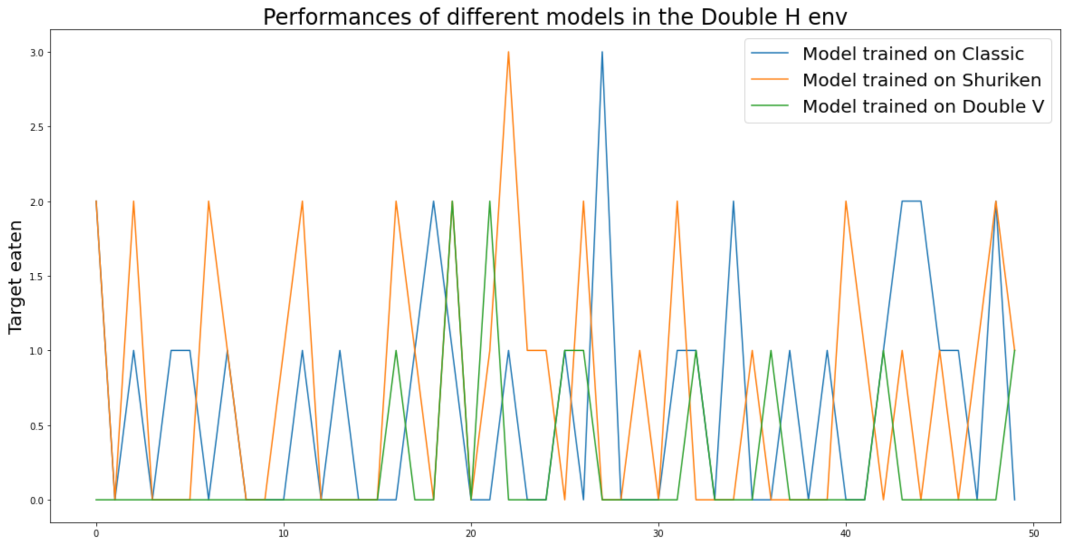


Figure 7: Performance of different Models in a new setting

We also performed a visual analysis to try to understand/have an idea about the different behavior learned. For example, we ran multiple times the models trained on the Shuriken arena in the Classic settings. It was pretty clear by observing that the snake was still avoiding the positions where a wall was present during the training, even if nothing was here anymore. It also struggled immensely/was almost incapable of eating a target which was positioned in a corresponding cell containing a wall in the ”Shuriken” setup. This is in relation with the discussion above, transferring the Classic Model to other setups. The snake does not understand the concept of ”wall $\rightarrow$ death”. Rather, it creates a geometrical representation of the arena, and learns which positions are acceptable for the snake to go, and which are not. The model overfits and does not pay attention to the walls themselves, the consequences of going into them, and it only remember and avoid their position in the arena. The effect is the same and it doesn't matter when we only run the agent in the same environment it was trained on. But if further researches should be done in the transferability of the Snake agent, some randomness of the arena should be introduced during the training. Two possible approach would be to randomly select which of the environment is used for each new iterations, or the environment could be the Classic setup with some random position being filled with wall during each new iteration. In both situations, it is important that the model makes the connection ”wall $\rightarrow$ death” rather than ”that specific cell of the arena $\rightarrow$ death”. We'd also suggest looking into the different replay buffer to train more regularly on the ”rare but important” situation, as well as train long enough until the agent actually converges, to see its peak performance.

6 Conclusion

In this project, we first build our own Gym environment of the game snake. We implement two different state space, a simpler one which can be used with any model as the states are just integer between 0 and 1023, and a more complicated matrix containing the whole board. We also made sure to implement different reward systems to allow to bypass the sparsity of reward in the traditional game which was a major limitation when training model on the Snake game. We train multiples models on the different arena available: some baseline algorithms like Q-Learning, SARSA, using the reduced state space and a Deep Q-Learning approach with a CNN considering the whole board as state space. We also test the performance of a deterministic TreeSearch algorithm. We compare the performances on the ”Classic” environment, then further dive into the ability of the CNN model to generalize to different arena. We train the same model on differently shaped boards, and compare the performance on all the possible arena. Considering the observation that the model trained on the simple arena is not able to run as soon as some obstacles appears, it is clear that the training methods were not optimal. It is interesting to note that the snake trained in the most complicated terrain (Shuriken) is logically the one obtaining the best performance in ”out of comfort situations”, but because it simply learned a geographic representation of the arena, it continues to avoid the cells (now empty) which were containing a wall previously.

Going forward, it should be interesting to conduct a way longer training of the model, until convergence if possible. The training setup should be modified and take into account the different observations of this project. Currently, the model is lacking training example when the snake is long. We proposed as solution to make sure that the agent can learn more effectively from situation where the snake is already quite long, and therefore make sure that we can reach top (max value) performance equal or better than the baseline methods. If the focus is kept on the transferability of the Snake agent, then some randomness should be including in the position of the walls to make sure that the model doesn't not simply learn to avoid some cells.

References

[Brockman et al., 2016] Brockman, G., Cheung, V., Pettersson, L., Schneider, J., Schulman, J., Tang, J., and Zaremba, W. (2016). Openai gym.

[Dimitrakakis and Ortner, 2022] Dimitrakakis, C. and Ortner, R. (2022). *Decision Making Under Uncertainty and Reinforcement Learning*. University of Gothenburg, Chalmers.

[Kathuria, 2021] Kathuria, A. (2021). Creating custom environments in openai gym. https://blog.paperspace.com/creating-custom-environments-openai-gym/.

[Krishnankutty, 2020] Krishnankutty, P. (2020). Nokia's snake, the mobile game that became an entire generation's obsession. https://theprint.in/features/nokias-snake-the-mobile-game-that-became-an-entire-generations-obsession/462873/.

[Mnih et al., 2013] Mnih, V., Kavukcuoglu, K., Silver, D., Graves, A., Antonoglou, I., Wierstra, D., and Riedmiller, M. (2013). Playing atari with deep reinforcement learning.

[sid sr, 2020] sid sr (2020). Q-snake. https://sid-sr.github.io/Q-Snake/.