UNIVERSITÉ JEAN MONNET

MLDM COHORT 2022-2024
ADVANCED MACHINE LEARNING
FINAL PROJECT

Bandits, Reinforcement learning

Students :
Mohamed MOUDJAHED
Franck SIRGUEY

Professor :
Amaury HABRARD

February 1, 2024

Contents

1 Bandit Algorithm **2**
1.1 Introduction . . . 2
1.2 Algorithms Description . . . 2
1.3 Experimental setup . . . 3
1.4 Algorithm evaluations . . . 3
1.5 Algorithm analysis . . . 4
1.6 Conclusion . . . 5

2 Reinforcement Learning **6**
2.1 Game environment . . . 6
2.2 Q-learning . . . 7
2.2.1 Introduction . . . 7
2.2.2 Implementation of our Qlearning class . . . 7
2.2.3 Experiments . . . 8
2.3 Deep-Q-learning . . . 10
2.3.1 Introduction . . . 10
2.3.2 Using the keras-rl library . . . 10
2.3.3 Experiments . . . 10
2.4 Conclusion . . . 12

1 Bandit Algorithm

1.1 Introduction

The bandit problem consists of choosing among several arms each offering an unknown random reward, with the objective of optimizing the total reward over a given period. In our exploration of the bandit problem, we are interested in evaluating the effectiveness of different algorithmic strategies against a set of arms. Two types of reward distributions will be tested: one where the rewards are determined according to a uniform distribution in the interval [0, 10], and the other following a Gaussian distribution with a mean of 5 and a standard deviation of 2.

We chose to implement and compare four specific algorithms: Incremental Uniform, UCB, Simple Epsilon-Greedy, and EXP3, as previously studied in class. The objective of our study is to conduct an experimental analysis on the performance of these algorithms in various situations, by observing their ability to identify the most profitable arm.

The tests will be carried out on m trials, and for each trial, each algorithm will be used n times. This will allow the cumulative and simple regret to be assessed for each algorithm, thus giving an estimate of their overall effectiveness.

1.2 Algorithms Description

1. **Incremental Uniform Algorithm:** This algorithm explores each arm equally, accumulating the rewards of each option after each draw. In each iteration, a reward is randomly generated for each arm, based on a normal distribution centered around the arm's value, and this reward is added to the cumulative total for that arm. At the end of all draws, the arm with the largest total reward is selected as the best.

2. **Upper Confidence Bound (UCB) Algorithm:** UCB is a strategy that balances exploration and exploitation. For each arm, it calculates a score based on the average reward obtained so far and an exploration term that depends on the number of times the arm has been played. The arm with the highest score is selected for the next drawing. This score favors arms with high rewards, but also those less often played.

3. **Simple Epsilon-Greedy Algorithm:** In the epsilon-greedy approach, the algorithm primarily chooses the arm with the best known average reward, but with probability 'epsilon' it selects an arm at random. This strategy ensures that the algorithm is not entirely exploitative and always has some chance of exploring other options. Over the course of draws, the algorithm adjusts its knowledge about each arm based on the rewards received, allowing better identification of the best arm while retaining a certain amount of random exploration.

4. **Exp3 Algorithm:** The Exp3 algorithm is designed for situations where arm rewards are non-stationary, that is, they can change over time. It assigns weights to each arm, and the probability of selecting an arm is proportional to its weight. After each draw, the weights are updated exponentially based on the rewards received, with adjustment by an 'epsilon' factor which controls the learning rate. This allows the algorithm to quickly adapt to changes in arm rewards, favoring recently

performing arms while maintaining a chance for less performing arms to be chosen and re-evaluated.

1.3 Experimental setup

We created a synthetic test environment, where we could vary the number of arms of the bandit. We assigned to each arm a random value from a distribution that we could choose between 2 modes, either uniform (ranging from 0 to 10) or Gaussian (mean of 5 and standard deviation of 2).

In the table 1, we have the different setup we did to observe the different influence of algorithms such as Incremental Uniform, UCB, Simple Epsilon Greedy, and EXP3.

	Incremental Uniform	**UCB**	**Simple Epsilon Greedy**	**EXP3**
Distribution	Uniform/Gaussian	Uniform/Gaussian	Uniform/Gaussian	Uniform/Gaussian
Epsilon	-	-	0.1, 0.2	0.1, 0.2
Arm Numbers	3/10/50	3/10/50	3/10/50	3/10/50
Trials	10	10	10	10
Arm Pulls	150	150	150	150

Table 1: Experimental setup for different algorithms across various settings.

1.4 Algorithm evaluations

To evaluate the effectiveness of the different strategies, we calculated and analyzed indicators such as cumulative regret and simple regret. To this end, we have plotted the different results to facilitate analysis.

As a reminder :

1. **Simple Regret:** The simple regret at a given time is the difference between the reward of the best arm (the one with the highest average reward) and the reward of the chosen arm at that time. It measures the instantaneous loss due to not playing the best arm at that precise moment.

2. **Cumulative Regret:** Cumulative regret, for its part, is the sum of simple regrets over all the draws. It measures the total loss accumulated over time due to consistently not playing the best arm.

1.5 Algorithm analysis

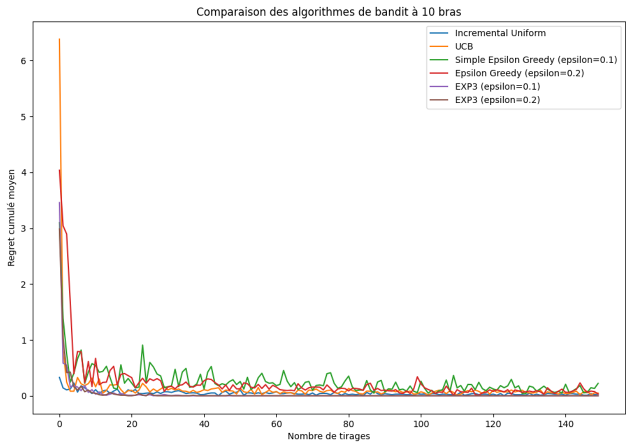


Figure 1: Cumulative Average Regret in a uniform distribution

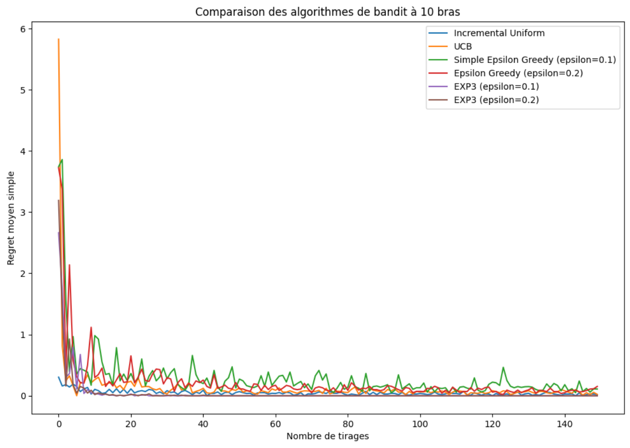


Figure 2: Simple Average Regret in a uniform distribution

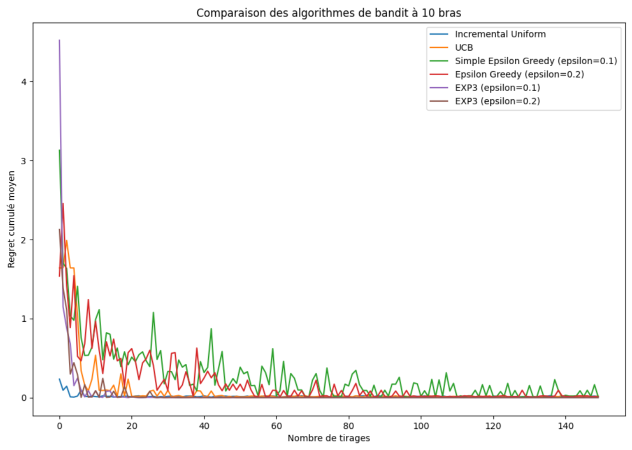


Figure 3: Cumulative Average Regret in a Gaussian Distribution

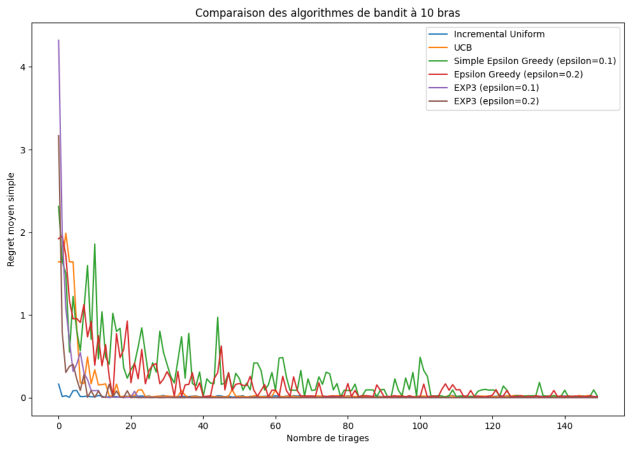


Figure 4: Simple Average Regret in a Gaussian Distribution

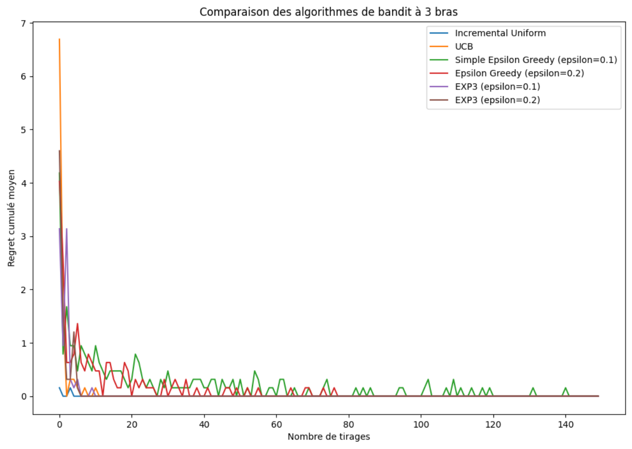


Figure 5: Cumulative Average Regret with 3 Arms in a uniform distribution

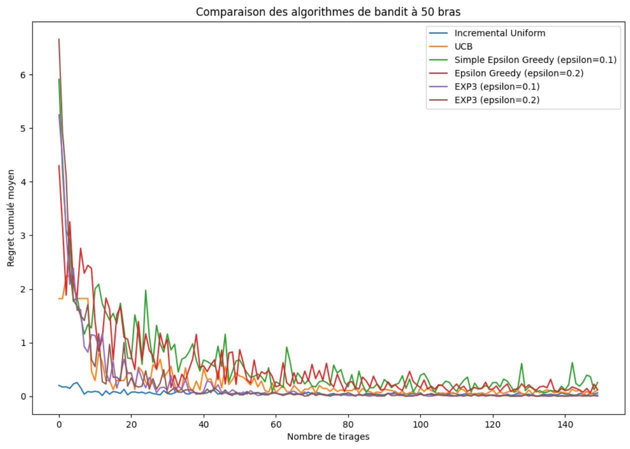


Figure 6: Cumulative Average Regret with 50 Arms in a uniform distribution

1. **First Experimentation**
   In a uniform distribution with 10 arms, EXP3 with an epsilon of 0.1 or 0.2 shows the lowest regret and quickly converges towards 0, usually around 30 pulls. Incremental Uniform, UCB, and Epsilon Greedy (with an epsilon of 0.1) exhibit higher regrets, but Epsilon Greedy with an epsilon of 0.2 performs better than its variant with an epsilon of 0.1. EXP3 is particularly effective due to its ability to balance exploration and exploitation in a stable and uniform environment.

2. **Second Experimentation**
   With a Gaussian distribution of 10 arms, UCB and Incremental Uniform converge better towards 0 compared to a uniform distribution. EXP3 (with epsilon of 0.1 or 0.2) and Incremental Uniform exhibit the lowest regret and converge quickly towards 0. UCB takes longer to converge. Epsilon Greedy methods, especially with an epsilon of 0.2, take longer to converge. The Gaussian distribution favors strategies like UCB and Incremental Uniform which better adapt to distributions with more pronounced variations.

3. **Third Experimentation**
   In a uniform distribution with 3 arms, the convergence to a low regret is quicker for all algorithms compared to that with 10 arms. This result is expected as fewer arms mean fewer choices to evaluate to find the best arm.

4. **Fourth Experimentation**
   With 50 arms in a uniform distribution, all algorithms struggle to achieve a regret close to 0. Incremental Uniform and EXP3 manage it, with Incremental Uniform converging more quickly. Epsilon Greedy and UCB have higher regrets. The presence of more arms makes the task more challenging for all algorithms, as they have to evaluate a greater number of options before finding the most rewarding ones.

1.6 Conclusion

In our study of the one-armed bandit problem, we found that the effectiveness of algorithms strongly depends on context. EXP3, in particular, performed well in both uniform and Gaussian distributions, thanks to its ability to balance exploration and exploitation. The UCB and Incremental Uniform algorithms also stood out in the Gaussian distribution, indicating effective adaptation to environments with more pronounced variations. The number of arms also affects performance, with faster convergence to low regret in scenarios with fewer arms. These results underline the importance of choosing an algorithm adapted not only to the distribution of rewards but also to the number of arms available.

2 Reinforcement Learning

2.1 Game environment

Our project consists of a simplified adaptation of the famous game PACMAN, developed in accordance with the project specifications. As a reminder, the objective is to navigate the PACMAN through a maze to collect pacdots while avoiding ghosts. Here are the key features of our version :

- **Movements:** The PACMAN and the ghosts can move in four directions: up, down, left, right.
- **Ghost Movement:** The ghosts move randomly. The randomness only takes into account the places where a ghost can move, so not in a block or in a corner of the matrix.
- **Interaction with pacdots:** When the PACMAN encounters a pacdot, the game ends and the score increases by 10 points.
- **Interaction with ghosts:** If the PACMAN encounters a ghost, the game ends and the score decreases by 10 points.
- **Obstacle management:** The PACMAN cannot cross blocks. An attempt to do so results in a loss of 1 point.

Our environmental class offers several configuration options :

- **Configuration choice:** Ability to choose a predefined or random game configuration, specifying the size of the terrain, the percentage of blocks, as well as the number of pacdots and ghosts.
- **Ghost Control:** Option to enable or disable ghost movement.

We have also integrated three standard methods used in simulation environments such as those offered by OpenAI Gym :

- `step(action)`: Performs a given action and returns the new state of the environment, the reward obtained, an indicator if the game is over, and additional information.
- `reset()`: Resets the environment to its starting state.
- `render()`: Displays the current state of the environment.

The presence of these methods is essential to ensure compatibility with specialized reinforcement learning libraries, such as keras-rl.

2.2 Q-learning

2.2.1 Introduction

Q-Learning is a machine learning technique that falls under reinforcement learning. In this approach, an agent learns to make decisions by discovering which actions maximize cumulative reward in a given environment. Q-Learning is a policy-free method, which means that it learns the value of the best possible action (optimal policy) independently of the action chosen by the agent during learning.

2.2.2 Implementation of our Qlearning class

In the context of PACMAN, the agent uses Q-Learning to learn the best strategy to navigate the game's maze. It does this by estimating the Q-value function, which gives the expected value of each state-action pair. At each step, the agent chooses an action, receives a reward from the environment, and updates the Q value based on this reward and estimated future Q values. This update is done according to a specific formula, which incorporates a discount factor to weight the importance of future rewards. We reused the approach seen in class:

Algorithm 1 Q-learning Algorithm

```
1: $Q \leftarrow InitialQ$ (initialization, e.g., to 0)
2: for $episode \leftarrow 0$ to $TotalEpisodes$ do
3:     $currentState \leftarrow \text{ResetEnvironment}()$
4:     for each step of episode $episode$ do
5:         $action \leftarrow \text{ChooseAction}(currentState, Q, \epsilon)$ - choose action with probability $\epsilon$
6:         $reward \leftarrow \text{GetReward}(currentState, action)$
7:         $nextState \leftarrow \text{NextState}(currentState, action)$
8:         $Q(currentState, action) \leftarrow Q(currentState, action) + \alpha[reward + \gamma \max_{action'} Q(nextState, action') - Q(currentState, action)]$
9:         $currentState \leftarrow nextState$
10:    end for
11: end for
12: return $Q$
```

- **Initialization :** The QLearning class is initialized with the game environment, as well as key parameters: the `alpha` learning rate, the `gamma` discount factor, and the `epsilon` exploration parameter. A Q table (`q_table`) is created to store the Q values for each state-action pair.
- **Choice of action :** At each step, the agent chooses an action. With probability `epsilon`, the agent explores by choosing a random action. Otherwise, it exploits its current knowledge by choosing the action with the highest Q value for the current state.
- **Table update Q :** After executing an action, the agent observes the reward and the new state. The Q table is updated using the Q-Learning update formula seen in class, which incorporates the reward received and the estimate of future value.

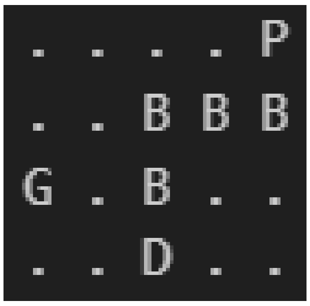
– **Training (`train`) :** The train method runs game episodes, where in each episode the environment is reset and the agent interacts with it until the game ends. Total rewards per episode are recorded for later analysis

Before explaining the experiments it is necessary to understand how the action is chosen. Our function, `ChooseAction`, takes the current state, the exploration rate (ϵ), the Q-table, and the environment as inputs. It first retrieves all possible actions from the environment. Then, it decides whether to explore (select a random action) or exploit (choose the best action based on Q-values). If exploring, it randomly selects from the possible actions. If exploiting, it calculates the Q-values for all possible actions, identifies the best actions (those with the maximum Q-value), and randomly selects one from these best actions.

Algorithm 2 ChooseAction Function

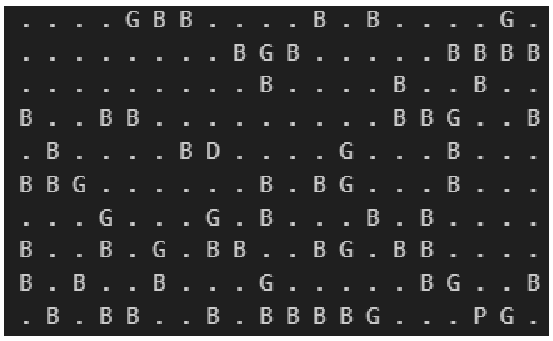
```
1:  function ChooseAction($state, \epsilon, Q, environment$)
2:      $possibleActions \leftarrow environment.GetPossibleActions()$
3:      if random.random() $< \epsilon$ then
4:          return random.choice($possibleActions$)
5:      else
6:          $qValues \leftarrow action : Q.GetQValue(state, action)$ for $action$ in $possibleActions$
7:          $maxQValue \leftarrow$ max($qValues.values()$)
8:          $bestActions \leftarrow$ [$action$ for $action, qValue$ in $qValues.items()$ if $qValue == maxQValue$]
9:          return random.choice($bestActions$)
10:     end if
11: end function
```

2.2.3 Experiments



```
. . . . P
. . B B B
G . B . .
. . D . .
```

Figure 7: Environment 1, identical to the statement



```
. . . . G B B . . . . B . B . . . . G .
. . . . . . . . B G B . . . . . B B B B
. . . . . . . . . B . . . . B . . B . .
B . . B B . . . . . . . . . B B G . . B
. B . . . . B D . . . . G . . . B . . .
B B G . . . . . . B . B G . . . B . . .
. . . G . . . G . B . . . B . B . . . .
B . . B . G . B B . . B G . B B . . . .
B . B . . B . . . G . . . . . B G . . B
. B . B B . . B . B B B B G . . . P G .
```

Figure 8: Environment 2, 10x20 with 15 ghosts, 1 pac-dot and 25% of the terrain occupied by blocks (placement is random)

As a reminder : P is the pac-man, D is a pac-dot, B is a block and G is a ghost

Without random movement

In simple environments, a Q-learning agent with an epsilon of 0 can often succeed, because it can quickly learn the best strategy among a small number of options. However, in more complex scenarios with many possible states and actions, the lack of exploration can severely limit the agent. He risks missing out on better and more effective strategies, remaining stuck in repetitive or unsuccessful behaviors, especially if known actions do not lead to positive results. In the examples that I have just shown, I only show the cases where our model succeeds in finding the best policy, but it may be that by restarting the model that our agent is unable to complete a game (in this case there we are immersed in an infinite loop) or that it does not find the best reward due to lack of exploration (for example by including a random character to our agent).

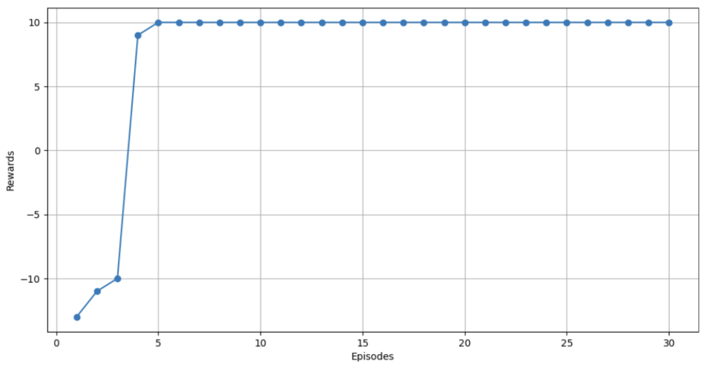


Figure 9: Environment 1, Rewarding episodes

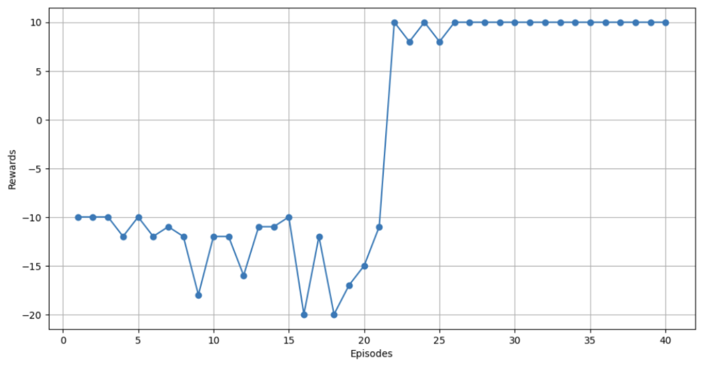


Figure 10: Environment 2, Rewarding episodes

With random movement

Setting epsilon to 0.1 in a Q-learning algorithm means that the agent will have a 10% probability of choosing a random action at each step, rather than always following the best known action. This introduces a certain level of exploration, allowing the agent to discover new strategies that might be more effective. In complex environments, this ability to explore occasionally can help the agent avoid getting stuck in sub-optimal behaviours and potentially find better solutions or paths that it might not otherwise have discovered. The random nature of the agent therefore leads us to certain undesirable movements but which tend to find an optimal policy.

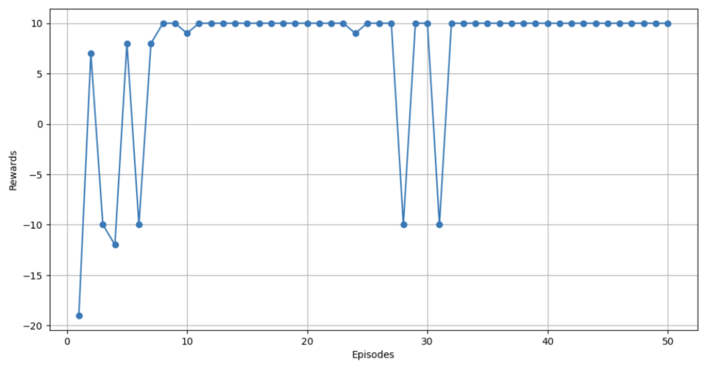


Figure 11: Environment 1, Rewarding episodes, with random movement

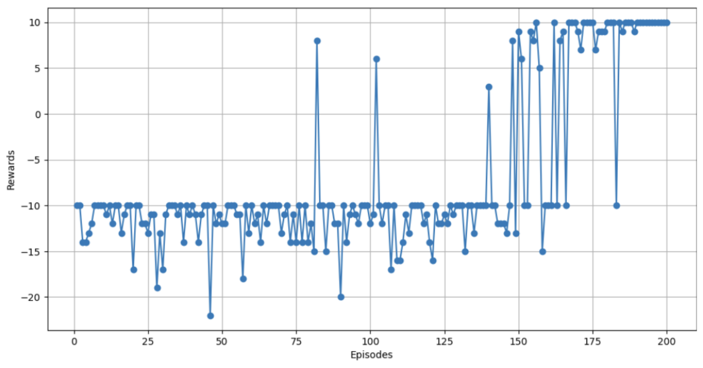


Figure 12: Environment 2, Rewarding episodes, with random movement, randomness is stopped at episode 190

2.3 Deep-Q-learning

2.3.1 Introduction

Deep Q-Learning is a reinforcement learning technique that combines deep neural networks with the Q-learning algorithm. As a reminder, the basic idea of Q-learning is to find a function (called the "Q-function") that evaluates the quality of a given action in a given state, i.e. it estimates the future "reward" that can be obtained by taking this action. In Deep Q-Learning, this Q-function is approximated using a neural network, which makes it possible to deal with problems with a very large and complex space of states and actions.

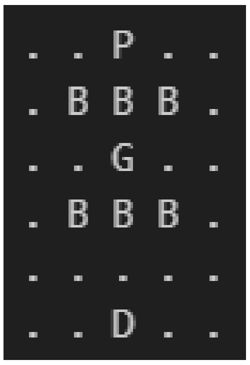
2.3.2 Using the keras-rl library

Keras-RL is a library that facilitates the creation of reinforcement learning agents using neural networks with Keras. In my code, I first create a neural model with Keras to estimate the value of actions in each state. Next, I define my agent's environment, specifying its possible states and actions. I also configure the learning parameters, such as the discount rate and the learning rate. After building and compiling my DQN agent with this model, I'm ready to train it in the environment to learn how to maximise rewards.

For my DQN agent designed for the Pac-Man game, I use a model that starts with a Flatten layer to transform the game's configuration matrix into a one-dimensional vector. Then there are two Dense layers of 64 neurons each, with ReLU activation, to process this input. The output of the model is a Dense layer with 4 neurons, corresponding to the 4 possible actions in Pac-Man (up, down, left, right), and this layer uses linear activation to provide the final decision on which action to take.

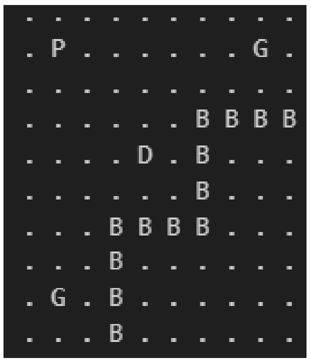
My model is trained by action, not by episode. So if I specify `nb_steps=200000`, training will take place over 200,000 actions. In addition, the parameter epsilon, used for the exploration policy, decreases progressively from 1 to 0 over the duration specified in number of steps, allowing the agent to explore at the beginning and to exploit more and more what it has learnt as the training progresses.

2.3.3 Experiments



```
. . P . .
. B B B .
. . G . .
. B B B .
. . . . .
. . D . .
```

Figure 13: Environment 3, Customised environment, 1 ghost, 1 pacdot



```
. . . . . . . . . .
. P . . . . . . G .
. . . . . . . . . .
. . . . . . B B B B
. . . . D . B . . .
. . . . . . B . . .
. . . B B B B . . .
. . . B . . . . . .
. G . B . . . . . .
. . . B . . . . . .
```

Figure 14: Environment 4, Customised environment, 2 ghosts, 1 pacdot

Without ghost movement

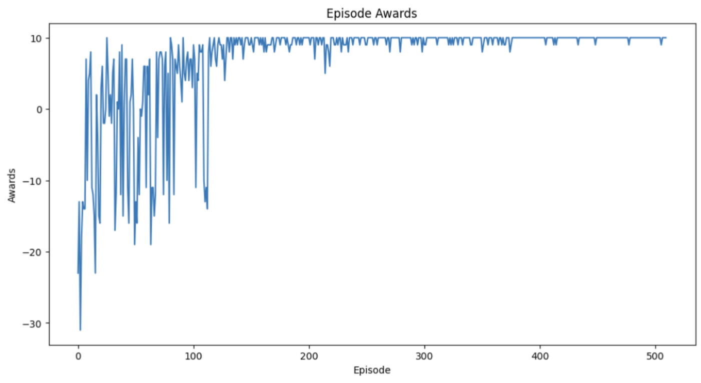


Figure 15: Environment 3 (the one on the statement), Rewarding episodes, eps_nb_steps=10000

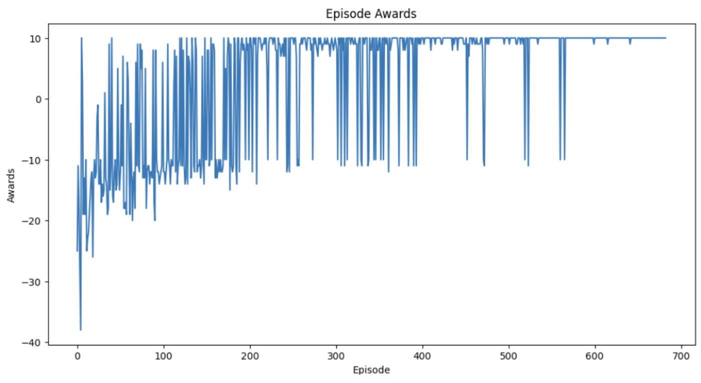


Figure 16: Environment 3, Rewarding episodes, eps_nb_steps=10000

In my tests with an environment where the ghosts remained motionless, my Deep Q-Learning agent learned an optimal policy in less than 700 episodes. This speed of convergence was due to the predictability and simplicity of the environment, which made learning and identifying effective strategies much more straightforward and less complex.

With ghost movement

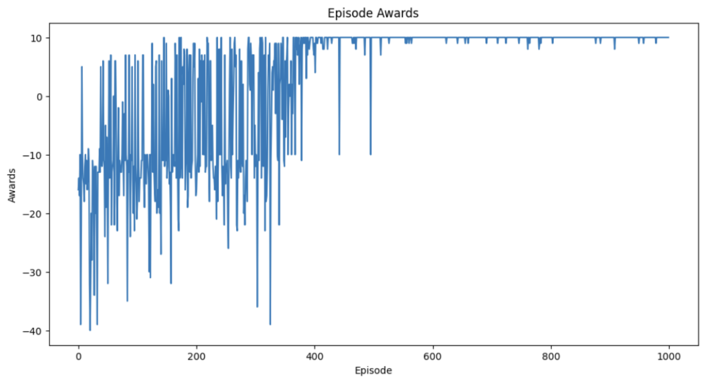


Figure 17: Environment 1 (the one on the statement), Rewarding episodes with ghost movement, eps_nb_steps=20000

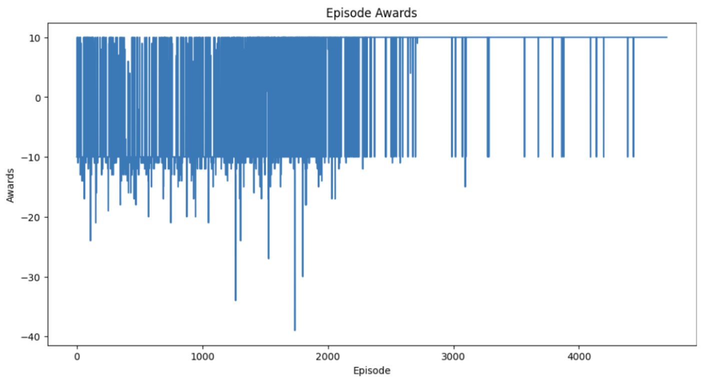


Figure 18: Environment 4, Rewarding episodes with ghost movement, eps_nb_steps=20000

When I changed my Pac-Man environment so that the ghosts moved around, I found it more difficult to get my Deep Q-Learning agent to converge on an optimal policy. The mobility of the ghosts considerably increased the complexity and uncertainty of the environment. Each movement of the ghosts changed the conditions of the environment, forcing me to constantly adapt my strategies. This increased variability meant that strategies that worked in one episode might not be effective in the next. As a result, I needed many more episodes to get good results, as my agent had to learn and adapt to a much wider range of possible situations.

The introduction of a second ghost into the Pac-Man game, using Deep Q-Learning, significantly increases the complexity of the model. Whereas the environment with a single ghost required around 450 episodes to converge to an optimal policy, the addition of an extra ghost increased this number to around 2,500 episodes. This significant increase in the number of episodes reflects the increased difficulty of the model to learn and adapt to a more complex environment. I have deliberately chosen an example with two phantoms, which seems simple enough to reflect the effectiveness of deepQlearning, although this method has its limitations.

2.4 Conclusion

In a simple Pac-Man environment where the ghosts don't move, traditional Q-Learning can be effective, as the state space is relatively small and manageable with a Q-table. Here, the agent learns to navigate and collect points while avoiding static ghosts. On the other hand, in a more complex version of Pac-Man where the ghosts move and the environment changes dynamically, Deep Q-Learning (DQL) is preferable. DQL uses neural networks to process the much larger state space and movement patterns of the ghosts, allowing the agent to learn more sophisticated strategies to maximise rewards in this more unpredictable and complex environment.