Reinforcment Learnig case study

Author: Thomas Lagkalis, Prof. Michail G. Lagoudakis

March 23, 2024

Abstract

Reinforcement Learning (RL) has emerged as a powerful paradigm for training intelligent agents to make sequential decisions in various environments. This project explores two fundamental RL algorithms, Q-Learning and Deep Q-Networks (DQN), by applying them to two distinct environments: the classic Cart Pole and the iconic Atari game, Pac-Man. The Cart Pole environment serves as a simple yet illustrative example, ideal for understanding basic RL concepts. In contrast, the Pac-Man environment offers a complex and visually rich domain, showcasing the scalability and versatility of RL algorithms. Through empirical evaluation and analysis, this project investigates the performance and behaviors of Q-Learning and DQN agents across these environments, investigating their strengths, limitations, and potential applications. Additionally, insights gained from comparing these algorithms in different environments provide valuable perspectives for understanding the capabilities and challenges of RL methodologies.

1 Introduction

Reinforcement learning (RL) is concerned with how an intelligent agent ought to take actions in a **dynamic environment** in order to maximize the cumulative reward. Reinforcement learning is one of three basic machine learning branches, alongside supervised learning and unsupervised learning. By providing feedback in the form of rewards, RL algorithms enable agents to navigate decision spaces, optimizing their actions to achieve desired goals over time. This project delves into RL, focusing on two algorithms: Q-Learning and Deep Q-Networks (DQN).

The primary objective of this report is to explore the capabilities and behaviors of Q-Learning and DQN agents within two distinct environments: the Cart Pole and Pac-Man. The Cart Pole environment is a classic (and basic) problem in control theory, which serves as a foundational example, offering a simple yet informative environment to understand fundamental RL concepts such as state, action, reward, and policy. On the other hand, the classic Atari game Ms Pac-Man environment presents a more complex and visually engaging example, showcasing the adaptability and scalability of RL algorithms.

The project is structured into two parts. The first part involves implementing both a Q-Learning agent and a DQN agent within the Cart Pole environment. In this part the two algorithms are compered to extract the advantages and the drawbacks of each algorithm. In the second part, the focus shifts to the implementation of a DQN agent in the more complex Ms Pac-Man environment. This part of the project explores the scalability and adaptability of RL algorithms to tackle dynamic decision-making tasks in visually rich and challenging environments with much larger state-space and action-space. Through experimentation and analysis in these two parts, this report aims to evaluate the performance, effectiveness, and computational efficiency of the Q-Learning and DQN algorithms.

2 Q-Learning and DQN

Central to RL is the notion of goal-oriented learning (fig.1). Agents are tasked with maximizing cumulative rewards over time, guiding their behavior accordingly. In order to formalize that idea, consider a computational agent moving in a discrete and finite world. At step t the agent is observing the state $s_t \in S$ and has to choose an action $a_t \in A$. Then it gets a probabilistic reward from the environment r_t, whose mean value is depended only from the state and action. The state of the world changes probabilistically to s_{t+1}. The agent's action selection is modeled as a map called policy: $\pi : S \rightarrow A$

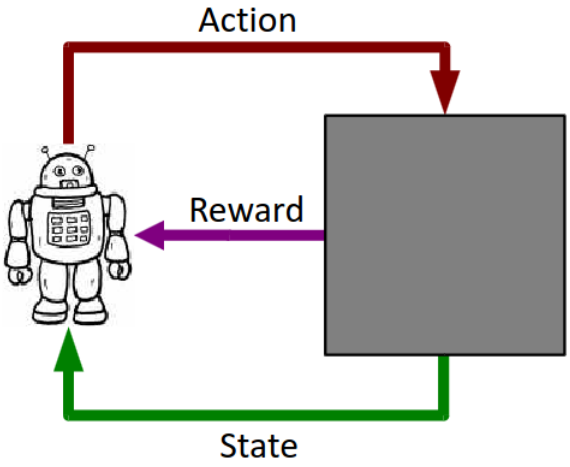

Figure 1: the basic idea of Reinforcement Learning (Source: class slides)

The state-value function $V_\pi(s)$ is defined as, expected discounted return starting with state s, i.e. $S_0 = s$, and successively following policy π. Formally,

$$V_\pi(s) = E[G|S_0 = s] = E[\sum_{t=0}^{\infty} \gamma^t r_{t+1}|S_0 = s]$$

Where γ is a discount factor. The value function could be thought of "how good" it is to be in a given state. It is proved [3] that there is at least one optimal policy π^* such that:

$$V^*(s) = V_{\pi^*}(s) = \max_a\{V_\pi(s)\}$$

We also define the Q function (a.k.a Q-values or action values), for policy π as:

$$Q^\pi(s, a) = E[G|S_0 = s, a_0 = a] = E[\sum_{t=0}^{\infty} \gamma^t r_{t+1}|S_0 = s, a_0 = a]$$

In other words, the Q value is the expected discounted reward for executing action a at state s and following policy π thereafter.

2.1 Q-Learning

The objective of Q-Learning is to estimate the Q values of an optimal policy π^* (for convenience we define these as $Q^*(s, a)$). It is proved [6] that Q-learning converges to the optimum action-values with probability 1. In order to estimate the Q-values we need to represent them as a table and populate it based on an optimal rule which guarantees the convergence to the optimal Q-values i.e.:

$$Q(s_n, a) \leftarrow Q(s_n, a)(1 - \alpha) + \alpha(r_t + \gamma \max_a Q(s_{n+1}, a))$$

if it's not a terminal state, or

$$Q(s_n, a) \leftarrow Q(s_n, a)$$

if it's a terminal state.

where $\alpha \in [0, 1]$ is the learning rate or step size. This simply determines to what extent newly acquired information overrides old information. Discount factor $\gamma \in [0, 1]$ denotes that rewards in the distant future are weighted less than rewards in the immediate future.

Algorithm 1 Q-Learning algorithm

Input

- number of episodes: numberOfEpisodes.
- initialized Q table Q(s,a) for all s,a.

Procedure

Get initial state s from the environment
for $i = 1, 2, 3, ..$ to numberOfEpisodes **do:**
while not a terminal state **do:**
sample action **a**.
make step in the environment based on action **a**
get the next sate s'
if s' is terminal state **do:**

$$Q(s, a) \leftarrow Q(s, a)$$

else do:

$$Q(s_n, a) \leftarrow Q(s_n, a)(1 - \alpha) + \alpha(r_t + \gamma \max_a Q(s_{n+1}, a))$$

end while
end for

Note: In the current implementation of the project the Q table is initialized with zeros.

Now that we have the Q table populated we can choose a policy that maximizes the expected reward by selecting the action with the maximum Q-value. So the policy for our agent is the following:

$$\pi(s) = \underset{a}{\text{argmax}}\{Q(s, a)\} \tag{1}$$

2.2 Exploration vs exploitation

When it comes to choose an action in either the Q-Learning or the DQN implementation we face the the following dilemma: *we explore the environment by randomly sampling from the action space and thus letting the agent learn new features about the environment or we capitalize on knowledge already gained by being greedy?* If the agent continues to exploit only past experiences, it is likely to get stuck in a sub-optimal policy. On the other hand, if it continues to explore without exploiting, it might never find a good policy. In order to let the agent explore the environment in the beginning and the exploit it after having gained enough experience we use the GLIE (Greedy in the Limit of Infinite Exploration) strategy to choose the action in each time step. This idea is described in the algorithm 2 and forms the ϵ-greedy policy.

Algorithm 2 ϵ-greedy policy

Input

- number of episodes to explore: exploreEpisodes.
- greedy factor: $\epsilon \in [0, 1]$.
- epsilon decay factor: $decayFactor \in [0, 1)$

Output

- action: a

Procedure

if current episode $<$ exploreEpisodes **do:**
    return random a
else if random number $\in [0, 1] < \epsilon$ **do**:
    return random a
else:
    return greedy action (eq. 1)

2.3 DQN

In the Q-Learning algorithm all the Q-Values are stored in the memory. For environments with a small state space, e.g. the cart pole environment, this is not a problem. On the other hand, in MsPacMan environment where each state corresponds to an RGB image (i.e array of dimensions $210 \times 160 \times 3$), it's impossible to store all the action-value pairs in a table in the memory. Hence, we need a way to approximate the Q value (of each action) for each state. The non-linear approximator, Deep Q-Networks (DQN) offer a solution to the problem. In order to help the agent learn about i

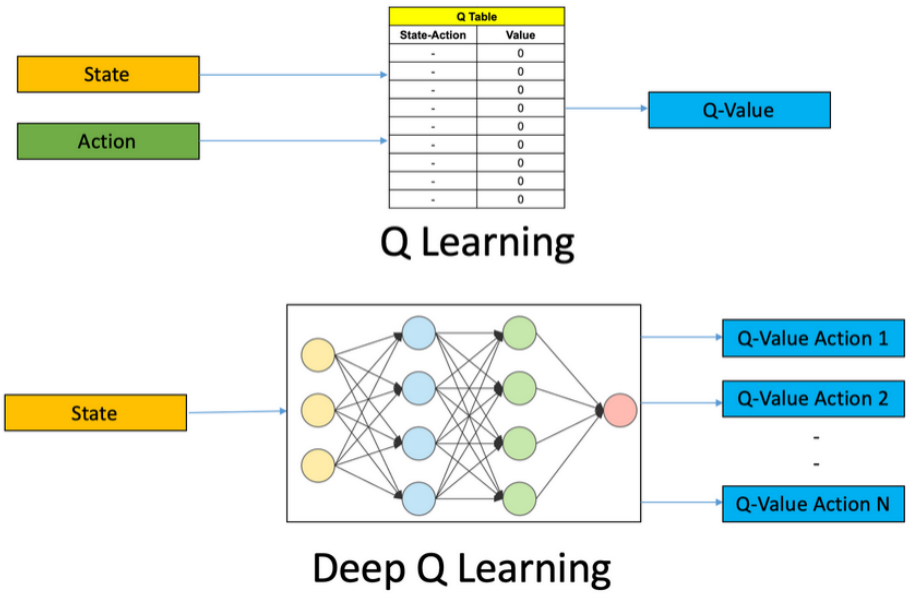

Figure 2: Source: www.analyticsvidhya.com

The use of neural networks in reinforcement learning proposes certain challenges compared to classic Deep Learning. In supervised deep learning, the target variable does not change and hence the training

is stable, which is just not true for RL. In order to apply the back-propagation algorithm [5] and train the Neural Network (NN) we need to calculate the error of an estimation based on some ground truth values. The problem is that in RL we don't have these ground truth values because the environment is constantly changing.

2.3.1 Target Network

In order to address this problem we build two NNs, the online network which makes all the predictions and the target network which servers as the ground truth predictor. The target network is not updated in each step, but it's rather updated after a predefined number of steps (or episodes). The reason for that is that we don't want to be chasing a constantly changing non-stationary target as described above. So we keep the target network as is for a number of iterations and then we synchronize it with the online network, i.e. coping the weights of the online NN to the target NN. This technic is significantly improving the stability of the training process.

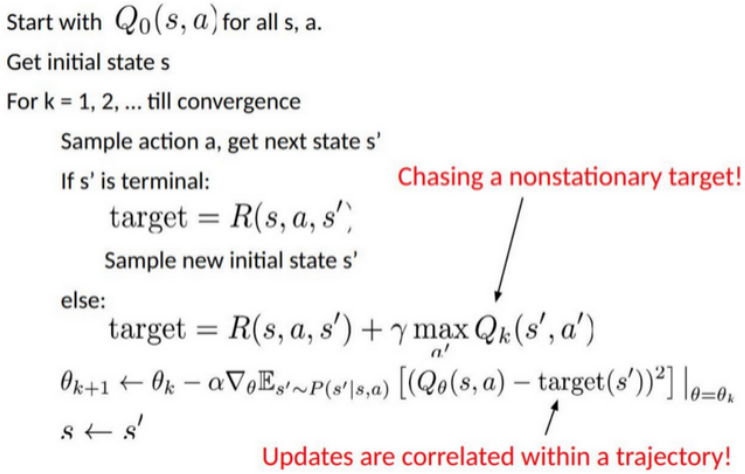

Figure 3: DQN algorithm, Source: www.analyticsvidhya.com

2.3.2 Experience replay

Another technic which is used to improve the stability of the training process is the experience replay as used in [4]. The problem which is addressed by this technic is that the samples are highly depended to each other because experience in time step t, $e_t = (s_t, a_t, r_t, s'_t)$, (where s is the state, a is the action, r is the reward and s' is the next state) depends on the previous experience e_{t-1}. So, we store the agent's experiences at each time-step, in a data set $D = e_0, e_1, ..., e_N$ pooled over many episodes into a replay memory. During each time step of the algorithm, we apply Q-learning updates, or minibatch updates, to samples of experience, drawn at random from the pool of stored samples (replay memory). That way the agent is training on random samples and avoiding over-fitting.

3 Environments description and model architecture

3.1 Cart Pole environment

A pole is attached by an un-actuated joint to a cart, which moves along a frictionless track. The pendulum is placed upright on the cart and the goal is to balance the pole by applying forces in the left and right direction on the cart [1]. A reward of +1 is provided for every time step that the pole remains upright. The episode ends when the pole is more than 15 degrees from vertical, or the cart moves more than 2.4 units from the center. The actions that are performed on the cart are :

The observation vector has 4 elements (dimension 4×1):

0 Push cart to the left
1 Push cart to the right

Index	Description	Min	Max
0	Cart Position	-4.8	4.8
1	Cart Velocity	-Inf	Inf
2	Pole Angle	-0.418 rad (-24°)	0.418 rad (24°)
3	Pole Angular Velocity	-Inf	Inf

In this environment was applied the Q-Learning and the DQN algorithm. For the DQN the neural network consists of fully connected layers with the Relu activation function. The first and the second hidden dense layers have 32 units and the output leyer has two units with a linear activation function.

3.2 Ms Pac Man

This environment is a classic game based on the well known Atari machine.The goal is to collect all of the pellets on the screen while avoiding the ghosts [2]. The rewards are based on the game score and the rules of the game. The actions dimension is 9×1 which corresponds to all the directions applied to the machine's joystick plus the 0 action i.e. do nothing. Each observation (state) is the image of the current state of the game, thus it's an RGB image with dimension $3 \times 210 \times 160$. We could use the same architecture we used in the cart pole environment but in order to extract the useful information from the image we are adding three more convolutional layers to the network before the fully connected dense layers. The first convolutional layer consists of 32 88 filters. The second has 64 4×4 filters and the third 64 3×3 filters. After these three convolutional layers comes the same dense layers as in the cart pole environment.

4 Results

In each environment the results acquired by the equivalent run are the graphs of cumulative rewards of each episode the moving average (smoothed) cumulative rewards and the epsilon through episodes.

4.1 Cart pole

In the cart pole environment the duration of the training process was 10000 episodes which lasted $\sim$ 5 hours on a conventional laptop machine. As we see in figs.4 when the exploration-only period ending at the 7000th episode the cumulative rewards increased dramatically. That's happening because the agent is no more exploring the environment but having already gained enough experience it's now exploiting it.

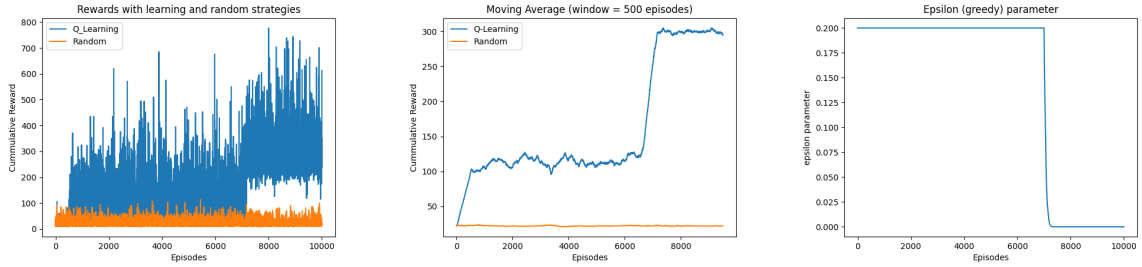

Figure 4: Results of Q-Learning in the cart pole environment

Similar behaviour is observed also with the DQN implementation (figs.5). This duration was $\sim$ 1000 episodes which lasted $\sim 1,5 days$ on the same machine. The first and obvious drawback of the DQN is that it's more time consuming and computational demanding than the Q-Learning algorithm which

often reacquires dedicated machines like GPU clusters to train the NNs on. How ever the DQN is able to be implemented in larger state-spaces like the Ms Pac Man environment.

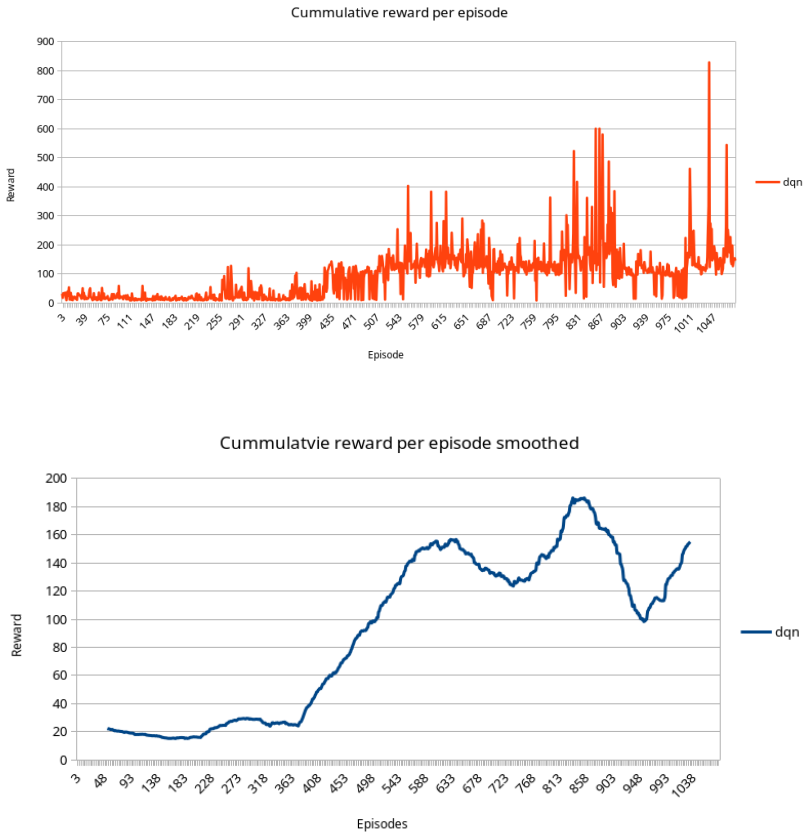

Figure 5: Results of DQn in the cart pole environment.

In the Ms Pac Man instance the correlation between the ϵ parameter decay and the cumulative rewards increase is no different than the previous instances, as expected. The convolutional layers increased the computational demands and the training lasted $\sim 3,5$ days for 3000 episodes. The agent increased it's rewards but the score kept far from a human level score, due to the small number of training episodes.

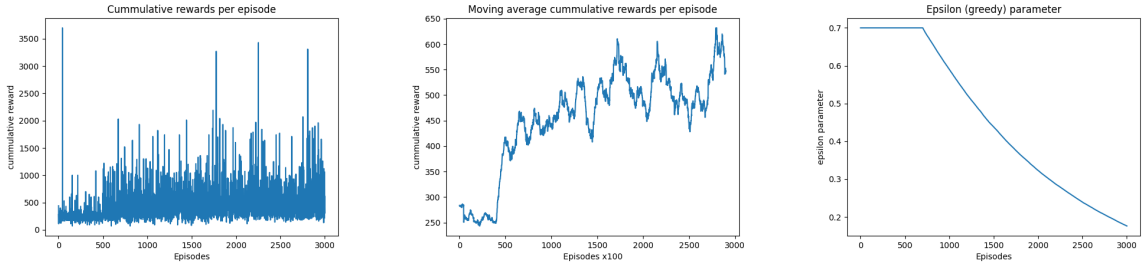

Figure 6: Results of DQN in the MsPacMan environment

5 Conclusion

In conclusion, our exploration of DQN in these two environments has highlighted their distinct advantages and disadvantages. Q-Learning, with its simplicity and ease of implementation, offers a straightforward approach to learning optimal policies in discrete state-action spaces. Its model-free nature and off-policy learning capability make it suitable for environments with discrete action spaces and small state-space, such as the Cart Pole.

On the other hand, DQN demonstrates a learning ability in handling complex, high-dimensional environments like Ms Pac-Man. By leveraging deep neural networks to approximate the action-value function, DQN imporves in learning from image inputs, enabling agents to navigate in visually rich environments effectively.

However, DQN's reliance on deep neural networks introduces certain challenges, such as instability during training and the need for extensive computational resources. Additionally, the complexity of DQN algorithms may require careful tuning of the NN parameters and good architecture design to achieve optimal performance.

References

[1] Cart pole environment,. https://www.gymlibrary.dev/environments/classic_control/cart_pole/.

[2] Mspacman environment,. https://gymnasium.farama.org/environments/atari/ms_pacman/.

[3] R. Bellman and S. Dreyfus. *Applied Dynamic Programming*. Princeton Legacy Library, 1962.

[4] Volodymyr Mnih, Koray Kavukcuoglu, David Silver, Alex Graves, Ioannis Antonoglou, Daan Wierstra, and Martin Riedmiller. Playing atari with deep reinforcement learning, 2013.

[5] Michael A. Nielsen. How the backpropagation algorithm works. *Neural Networks and Deep Learning. Determination Press.*, 2015.

[6] Dayan P. Watkins, C.J.C.H. Q-learning. *Machine Learning*, 1992.