Deep Q-Networks

From Q-Learning to DQN applied to Snake

Erika Lena, Francesco Ortu, Alessandro Serra

https://github.com/erikalena/Deep-QNetworks

DATA SCIENCE & SCIENTIFIC COMPUTING

Overview

- ▶ QLearning
- ▶ Q-Networks
- ▶ Deep Q-Networks
    - ▶ Implementation
    - ▶ Results
    - ▶ Conclusions

Purpose

The aim of the project was the implementation of Deep-QNetworks [1].

In what follows, we try to reconstruct the path from the basic QLearning algorithm applied to a simple GridWorld problem to its application in the training of Deep Neural Networks for learning to play games.

Application

All the methods and the strategies are shown with respect to a particular case study, which is Snake. This simple game was chosen because it can be simplified enough to be treated like a GridWorld problem, but it is also suitable for the training of DQN, like the Atari games used in the original paper.

QLearning

QLearning
Tabular method

In its simplest version, Snake can be thought as a GridWorld environment:

- ▶ the state space is discrete and finite, and the agent can only move in four directions (up, down, left, right);
- ▶ if the agent hits a wall, it proceeds from the opposite side (i.e. if the agent hits the left wall, it will appear on the right side of the grid);
- ▶ the agent receives a reward of +1 for each apple eaten, and a reward of -1 for each step taken; the episode ends when the agent eats the apple.

We are in a framework in which we know the dynamics of the environment, but we do not know the model (i.e. the transition probabilities and the reward function).

In this case, we can use a tabular method to learn the optimal policy.

QLearning
Tabular method

Q-learning in an **off-policy TD control algorithm**, which allows us to learn the optimal policy from the agent's experience, without requiring a model of the environment.

It estimates the optimal action-value function, starting by Q_0 and iteratively updating it:

- it takes an action using an ϵ-greedy policy derived from Q;
- observe the reward R and next state S';
- update Q as:

$$Q(S, A) \leftarrow Q(S, A) + \alpha[R + \gamma \max_a Q(S', a) - Q(S, A)]$$

where α is the learning rate and γ is the discount factor.
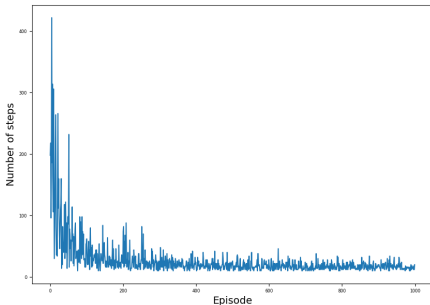
QLearning
Case study: Snake

This algorithm can be applied to our Snake environment, allowing us to learn a Qtable, which provides the Qvalue and the best action for each state, which is a tuple made of the position of the head and the position of the apple.

Configuration:

- grid 10x10
- $\alpha_0 = 0.15$
- $\gamma = 1.0$
- $\epsilon_0 = 0.4$
- $n\_episodes = 1000$

QLearning
Case study: Snake

▶ Pros:
  - it is rather fast, it took $\sim$ 2min to run;
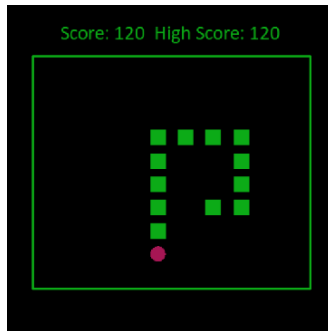 - the algorithm used is simple and explainable.

▶ Cons:
  - It is expensive in terms of memory: we need to save all the Qtable. In this very simple framework the cost is already $O(n^4)$, where n is the size of our grid.
  - We can not generalize it to an higher number of states, nor even to more complex game logic.

QLearning
Case study: Snake
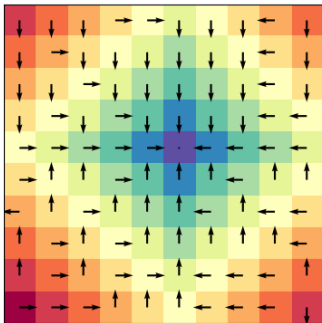
(a) *Qvalues and corresponding actions for target (5,4).*

(b) *Playing snake using this simple policy.*

8

QNetworks

QNetworks
Moving towards Neural Networks

The main problem with tabular methods is that they do not scale well with the size of the state space, which is often continuous and very large. In this case, we can use function approximation to estimate the action-value function, which is the core of the Q-learning algorithm.

One simple and naïve way of applying function approximation, to obtain a continuous representation of our state space, is to exploit the Qtable we learnt to train a NN. This is very simple, but it does not exploit the full potential of a NN and it can be done also through a linear approximation.

QNetworks
Moving towards Neural Networks

Another possibility is to exploit Neural Networks while learning, by integrating NN directly in Q-Learning algorithm. This is what in the paper is referred as **QNetworks**.

To do this we trained a small neural network, with just one hidden layer of 32 hidden nodes. During the training the network takes as input the state S (the position of the snake plus the position of the food $\rightarrow$ 4 input nodes), and it provides as output the Qvalues associated to S for each of the possible actions (4 output nodes).

At each iteration i, the NN tries to minimize the following loss function:

$L_i(\theta_i) = \mathbb{E}_{s,a \sim \rho(\cdot)}[(r + \gamma \max_{a'} Q(s', a'; \theta_{i-1}) - Q(s, a; \theta_i))^2]$

QNetworks
Moving towards Neural Networks

During the training, the Q-Network makes also use of a **Replay Buffer**. The mechanism of replay makes use of previous experience to make the training distribution smoother and more robust, and to alleviate the problem of correlated data.

The replay buffer is a list of tuples (s, a, r, s') which stores the agent's experience. At each step, we add the tuple (s, a, r, s') to the replay buffer; then we sample a batch of tuples from the replay buffer and we use them to train the network.

QNetworks
Case study: Snake

Configuration:

- NN: 4 input nodes, 4 output nodes, 32 hidden nodes, 1348 parameters
- $\alpha_0 = 10^{-4}$ (modified by the optimizer during training)
- $\gamma = 0.99$
- $\epsilon_0 = 1$ decreased by 0.01 every 200 episodes
- $n\_episodes = 20000$ (Solved in 16325 episodes)
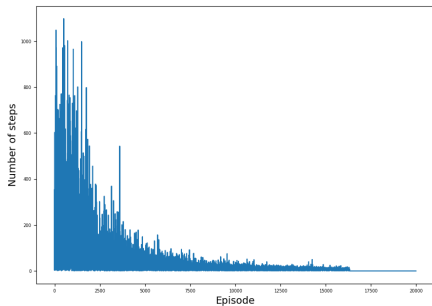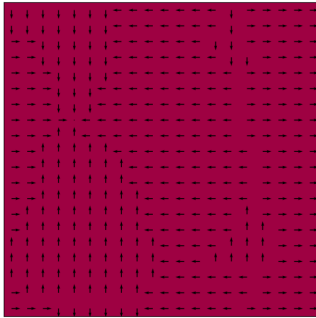
QNetworks
Case study: Snake

▶ Pros:
- we can generalize to more complex environments and game logic;
- we are still using Q-Learning, even if results become less explainable;
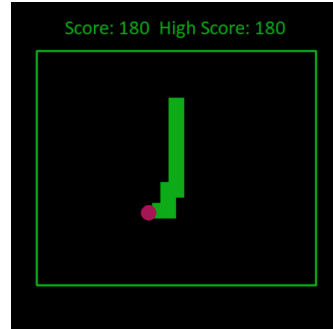- the number of parameters can be kept small enough.

▶ Cons:
- it takes more time to train $\sim$ 20min;
- Neural Networks add a lot of complexity, in particular for what concerns hyper-parameters, which need to be properly tuned.

QNetworks
Case study: Snake

(a) *Optimal actions to be taken for each state with target (7,4).*

(b) *Playing snake using trained QNetwork*

14

QNetworks
Case study: Snake

Now, we want to move towards a **full implementation** of the game, in particular by making the snake learn how not to eat itself. We need to add a negative reward each time the snake crosses itself and to change the representation of the states in order to deal with the changing body of the snake.

We can use a Neural Network like we did before, but taking in account the body (with a maximum length) of the snake.

Most critical issues:

▶ how to work with the body of the snake being the size of the input fixed;
▶ choosing the hyper-parameters;
▶ choosing when to **end an episode**;
▶ choosing the **rewards**.

QNetworks
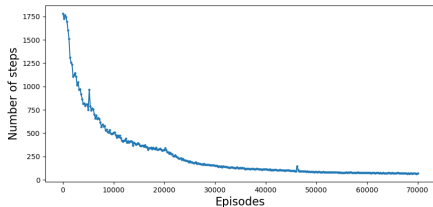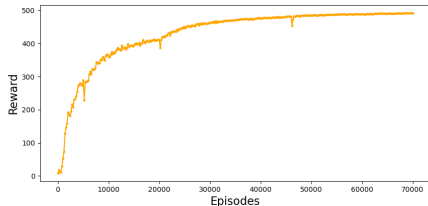## Case study: Snake

After many trials we decided to:

- ▶ assign zero reward for the snake movement;
- ▶ assign -1 for each time the snake eats itself, but **do not end** the episode;
- ▶ to tune the positive reward with respect to the mean number of steps necessary to conclude an episode;
- ▶ to end an episode when the food eaten was = 10, to limit the training time;
- ▶ to carefully tune the decay of epsilon with respect to the number of steps taken by the first episodes.

QNetworks
Case study: Snake

Configuration:

- NN: 24 input nodes (body length = 10), 4 output nodes, 32 hidden nodes, 1988 parameters
- $\alpha_0 = 10^{-4}$ (modified by the optimizer during training)
- $\gamma = 0.99$
- $\epsilon = 1$ decreased by 0.01 every 500000 frames
- $n\_episodes = 70000$
- Rewards:
    - movement: 0
    - food: +50
    - crossing itself: -1

QNetworks
Case study: Snake

▶ Pros:
- we can generalize to more complex environments and game logic;
- the results we obtain are useful for training following more complex models.

▶ Cons:
- it takes a lot of time $\sim 6h$ (on Intel Xeon, 2.60GHz);
- it was particularly difficult to tune hyperparameters and rewards.

Deep Q-Networks

Deep QNetworks

Why don't use more parameters?

▶ Instead of using an MLP to approximate the Q function, let's harness the power of Deep Neural Networks.

▶ Use a CNN that takes the frame of the current state as input and predicts the Q-values for each possible action.

▶ **Pros:** A more general algorithm that eliminates the need for encoding the state.

▶ **Cons:** The more sophisticated the networks are, the more computational resources are required.

Deep QNetworks
Algorithm
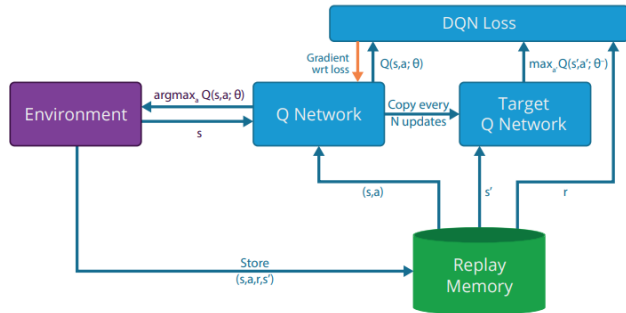
Deep QNetworks

Algorithm

Algorithm 1: Deep Q-learning Pseudocode

Data: Initialize replay memory D, Q-function Q, target Q-function $\hat{Q}$, exploration probability ε, learning rate α, discount factor γ, batch size B

Result: Trained Q-function Q

1 **for** *episode* $\leftarrow$ 1 **to** N **do**
2   Initialize state s_1;
3   **for** $t \leftarrow 1$ **to** T **do**
4   With probability ε select a random action a_t, otherwise select $a_t = \arg\max_a Q(s_t, a)$;
5   Execute action a_t in the environment and observe the next state s_{t+1} and reward r_t;
6   Store transition (s_t, a_t, r_t, s_{t+1}) in replay memory D;
7   Sample a minibatch of transitions (s_i, a_i, r_i, s_{i+1}) from D;
8   Set target value $y_i = r_i + \gamma \cdot \max_a \hat{Q}(s_{i+1}, a)$;
9   Update Q-function using gradient descent: $Q(s_i, a_i) \leftarrow Q(s_i, a_i) + \alpha \cdot (y_i - Q(s_i, a_i))$;
10   **end**
11   Update the target Q-function: $\hat{Q}(s, a) \leftarrow Q(s, a)$;
12 **end**

Deep QNetworks
Algorithm

In order to make the algorithm more stable and less likely to oscillate or diverge the Q-network had been cloned. This leads to two NNs: one online network that produces prediction for the Q function and one target network used to produce its expected value. After 10000 frames the online network update the target network with its current weights. In this way the agent is trying to minimize the distance from a target that is stationary for a limited time.

Deep QNetworks
Algorithm

▶ Pros:
- CNN are better suited to detect geometrical pattern and edges.
- The use of convolution enforce equivariance of the representations resulting in a possibly better generalization.

▶ Cons:
- They have more parameters than a MLP, thus they're more computationally demanding.

Implementation

Deep QNetworks

Implementation

- ▶ The algorithms were implemented in Python.
- ▶ Pytorch was used as framework for neural networks implementation and GPU-computation.
- ▶ The environment class inherit from Gymnasium library [2].
- ▶ The agent class manage all the interactions with the environment.

Deep QNetworks
Implementation

The agent provides a snap of each frame as torch tensor. In order to prompt the net to differentiate between *body*, *head*, *target* we denoted each of them with different values, and they sum when there's an intersection.

```
array([[0. , 0. , 0. , 0. , 0. , 0. , 0. , 0. , 0. , 1. ],
       [0. , 0. , 0. , 0. , 0. , 0. , 0. , 0.5, 0. , 0. ],
       [0. , 0. , 0. , 0. , 0. , 0. , 0. , 0. , 0. , 0. ],
       [0. , 0. , 0. , 0. , 0. , 0. , 0. , 0. , 0. , 0. ],
       [0. , 0. , 0. , 0. , 0. , 0. , 0. , 0. , 0. , 0. ],
       [0. , 0. , 0. , 0. , 0. , 0. , 0. , 0. , 0. , 0. ],
       [0. , 0. , 0. , 0. , 0. , 0. , 0. , 0. , 0. , 0. ],
       [0.1, 0. , 0. , 0. , 0. , 0. , 0. , 0. , 0. , 0.1],
       [0. , 0. , 0. , 0. , 0. , 0. , 0. , 0. , 0. , 0.2],
       [0. , 0. , 0. , 0. , 0. , 0. , 0. , 0. , 0. , 0.2]])
```

Deep Q Networks
Hyperparameters

- The parameters that have a major impact in the learning phase are the number of frames after decaying epsilon, the reward function, and the number of steps for each episode.
- We set these parameters based on previous experience with a simple model, which led us to obtain acceptable results without too many trial-and-error steps.
- Since the learning process is highly computationally demanding, we decided to use a small environment size.

Deep QNetworks

Hyperprameters

Env size	(10,10)
Buffer size	100000
Update target	10000
Update after actions	4
CNN layers	2 Convolution, 2 Linear
CNN n. parameters	52000
reward	$x_{neg}, x_{pos} \in [-100, 100]$
epsilon	$\epsilon \in [0.1, 1]$
decay step	10000

Deep Q Networks
Hardware

- ▶ We use the ORFEO cluster with the A100 GPUs.
- ▶ The computations were extremely expensive, with a training time of approximately 48 hours for the DQN algorithm.
- ▶ We suspect that a significant amount of time is spent on CPU-GPU communications, limiting our ability to fully exploit the parallel acceleration capabilities of the GPU.
- ▶ We also found that in the case of MLP networks, it is more convenient to perform all the computations on the CPU.
- ▶ The speed-up is likely to be more significant when we scale up the game (increase the grid size).

Results and Discussion

Deep Q Networks
Training analysis

- ▶ We trained the model with a max length of the snake of 10.
- ▶ Due to time constraints, we were not able to have results for an unlimited length.
- ▶ The first run took approximately 24 hours.
- ▶ By leveraging the insights gained from the first run, we expect to optimize the training process further. Specifically, we could fine-tune the epsilon decay strategy and reward function, which should lead to quicker convergence and improved results.

Deep QNetworks
Training analysis - Limited
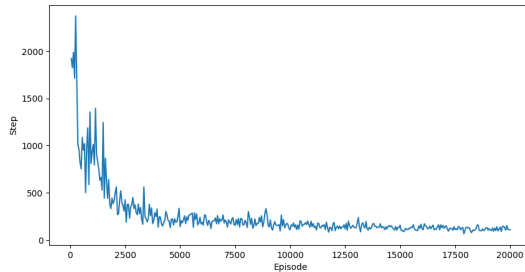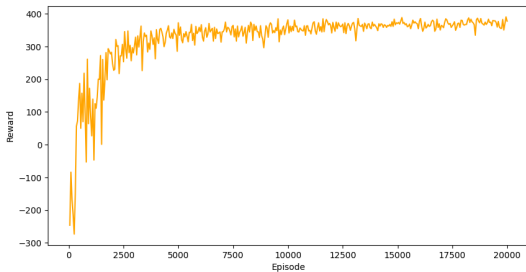
Deep Q Networks
Discussion

Major difficulties found:

- ▶ Setting the reward function.
- ▶ Deciding the best way to create the input of the CNN.
- ▶ Choosing the strategy for epsilon decay.

Deep Q Networks
Discussion

Solution adopted:

▶ Building our models incrementally has allowed us to have a control on what the snake was doing and an execution time small enough to check results and apply suitable corrections.

▶ We decide to not use multi-channel input in order to keep a smaller number of parameters, as discussed previously.

▶ The way in which we decay epsilon has a great impact on the results, but in particular on the training time, even in this case having a simpler model, to use as comparison, has been crucial.

Deep Q Networks
Discussion

Other things we have discussed in order to accelerate the training:

▶ Should the episode terminate when the snake eats itself?
  - We decided not to terminate the episode when the snake eats itself to reduce the training time. In this way, the CNN sees more states, and the behavior can be learned from the negative rewards. However this choice has other drawbacks.
▶ Should we limit the snake from going back to accelerate the training?
▶ Could vectorizing the environment lead to faster learning?
  - Since the environment is compatible with the Gym library, which supports vectorized environments, we tried running multiple episodes at the same time, but we did not see visible improvements.

Deep QNetworks
How to learn to not eat itself?

The main problem we have noticed in the learned policy is that the snake keeps eating itself. We have thought about possible solutions to this problem:

- ▶ Continue the training, which at a certain time becomes very slow.
- ▶ Add a small reward > 0 for the snake to stay alive.
- ▶ Modify the CNN in order to have more "context" (adding channels).
- ▶ Improve the tuning of hyperparameters of the network.
- ▶ Establish how to deal with wrong configurations in tensorial representation of frames.

Conclusions

Conclusions
Comparisons

Q-learning

► *Pros*: Simple and explainable. Fast to train.

► *Cons*: It does not scale.

Q-networks

► *Pros*: They can scale to large state spaces and they can generalize to complex environments. They are smaller and faster than CNNs.

► *Cons*: More complex and requires more tuning. The 2D configuration of the game has to be handled as 1D tensor.

Conclusions
Comparisons

Deep Q-Networks

► *Pros*: It can learn to play complex games and it can handle complex state spaces.

► *Cons*: It is more computational and in particular time-expensive and it requires a lot of work in properly tune the parameters and hyperparameters.

Conclusions
Possible improvements and future exploration

► Find the optimal epsilon decay strategy.

► Scale up the size of the grid.

► Scale up the neural networks adding channels.

► Optimize the code, exploit parallelism.

► Implement more advanced algorithms, like double-DQN.

Conclusions
Note on the use of AI assistant

We partially used the following AI assistant during this work:

► GitHub Copilot as AI intelligent code completion.

► ChatGPT and Bard to refactoring small part of code and to correct the slides

References I

📄 V. Mnih, K. Kavukcuoglu, D. Silver, A. Graves, I. Antonoglou, D. Wierstra, and
M. Riedmiller.
Playing atari with deep reinforcement learning, 2013.

📄 M. Towers, J. K. Terry, A. Kwiatkowski, J. U. Balis, G. d. Cola, T. Deleu, M. Goulão,
A. Kallinteris, A. KG, M. Krimmel, R. Perez-Vicente, A. Pierré, S. Schulhoff, J. J. Tai,
A. T. J. Shen, and O. G. Younis.
Gymnasium, Mar. 2023.