Learning to play Pong with DQN

Casper Norrbin, Joel Sikström

May 2023

1 Deep Q-Network

Deep Q-Network (DQN), as first introduced by V. Minh et al. [1], combines the ideas of Q-learning with a deep neural network to learn a policy for playing games directly from raw pixel inputs. The new idea in DQN is the use of a deep neural network to estimate Q-values of the possible actions in a given state. The neural network takes raw pixels inputs from the game and outputs Q-values for each action. This allows the agent to learn a policy from sensory input that can be complex and high-dimensional. To avoid overfitting to recent experiences when training the agent, DQN also uses experience replay, a method that randomly samples past experiences and uses them to update the network weights.

A key innovation that was used to improve the learning process was to use a target network to stabilise the learning process. V. Minh et al. used a separate neural network, which they called the target network, to estimate target Q-values that were used in the Q-learning updates. Updating the target network less frequently than the main network allows the Q-learning updates to be more stable and reduce the risk of weights in the network oscillating and negatively impacting the learning process.

In summary, V. Minh et al. have demonstrated how combining deep neural networks with reinforcement learning can improve the agent's ability to learn environments. Furthermore, they highlight the importance of using methods such as experience replay and target networks to stabilise the learning process to make it possible to learn a complex and high-dimensional policy from rudimentary inputs.

2 Experiments

We have implemented a version of DQN in Python, as described by V. Minh et al. [1], and trained two models, one on the gymnasium environment Cart Pole [2] and one on the gymnasium environment for the Atari game Pong [3].

The hyperparameters listed in Table 1 are the starting parameters that have been used when training the models for the respective environments. We will also explore how changing some of the parameters might affect the learning process, both through results and impact on time for the agent to learn.

Parameter	**Cart Pole**	**Pong**
Replay memory size	50000	100000
Number of episodes	1000	1000
Minibatch size	32	32
Target network update frequency	100	1000
Training frequency	1	4
Discount factor	0.99	0.99
Learning rate	0.0004	0.0004
Initial exploration	1	1
Final exploration	0.01	0.01
Final exploration frame	10000	10000
Number of actions	2	2

Table 1: Initial hyperparameters used for Cart Pole and Pong environments.

2.1 Cart Pole

The Cart Pole environment is where a pole is attached by an unactuated joint to a cart, which moves along a frictionless track. The possible actions in the environment are $\mathcal{A} = \{0, 1\}$, where 0 pushes the cart to the left and 1 to the right. The state space is described by the cart's position and velocity and the pole's angle and angular velocity. The goal is to keep the pole upright for as long as possible, incentivised by a +1 reward for every step that is taken. The episode ends when the pole tips over, the car leaves the screen or automatically after 500 episodes.

Figure 1 shows the mean return of evaluating the agent after n episodes of training using the hyper-parameters in Table 1. We can observe that the agent got the best return after about 600 episodes, consistently getting +200 reward over multiple episodes. Training the model from scratch multiple times consistently gives the maximum mean reward after about 600 episodes, after which we can observe that the agent rarely learns a better policy, and if it does, the increased mean return is small.

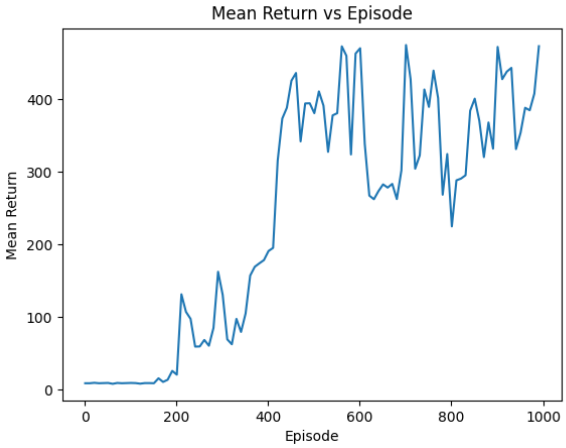
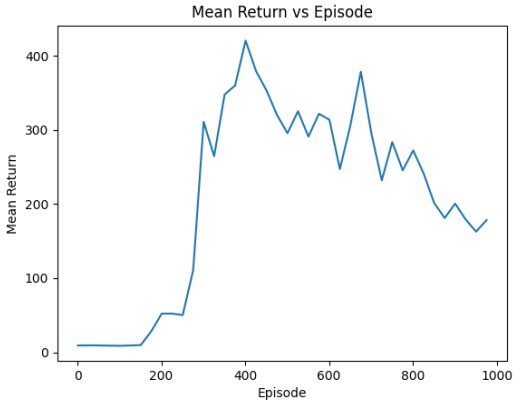


Figure 1: Mean return of 5 runs after training up to n episodes.

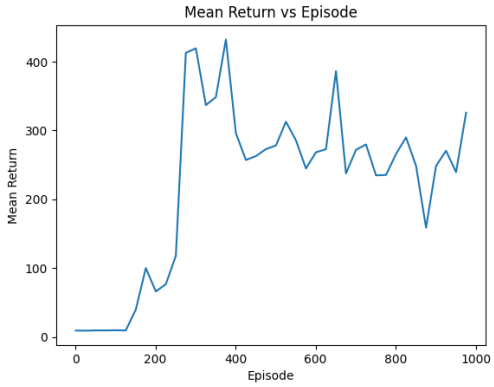
Figure 2 shows the mean return of evaluating the agent after n episodes of training using the hyper-parameters in Table 1, but with a learning rate of 10^{-3} and varying training frequency.

Figure 2: Mean return of 5 runs after training up to n episodes using different training frequencies.

Figure 3 shows the mean return of evaluating the agent after n episodes of training with the hyperparameters listed in Table 1 but with a batch size of 64 and 128 respectively. Using a batch size of 64, we can observe that the best reward is reached after about 400 episodes, while a batch size of 128 reaches its best reward after about 300 episodes.



(a) batch size = 64



(b) batch size = 128

Figure 3: (a) batch size 64 (b) batch size 128.

2.2 Pong

The Pong environment is a replica of the classic Atari game "Pong". The player moves a paddle up and down, with the goal to bounce a ball to the enemy, who also controls a paddle on the other side of the field. If one misses the ball, the other player receives a point and a new round is started. The possible actions are $\mathcal{A} = \{0, 1, 2, 3, 4, 5\}$, but there are only three different ways to act in the environment. Actions 0 and 1 do not move the paddle, actions 2 and 4 move the paddle up, and actions 3 and 5 move the paddle down.

The state space is described by the screen output of the game, which is 210×160 pixels of RGB values. To simplify the input to the agent, the observations are grayscaled and resized to 84×84 pixels. This reduces the complexity of the observation, without removing much information. Additionally, multiple frames are stacked and delivered to the agent together. One frame of the game does not contain enough information to act on, as it is impossible to know which direction the ball is travelling.

Figure 4 shows the mean return of evaluating the agent after n episodes of training using the hyperparameters in Table 1. We can observe that the mean return of the agent trends up as the number of episodes increases. The best return of -5.0 is given at episode 900, but there is a large possibility that a better agent would be found if the number of episodes increased.

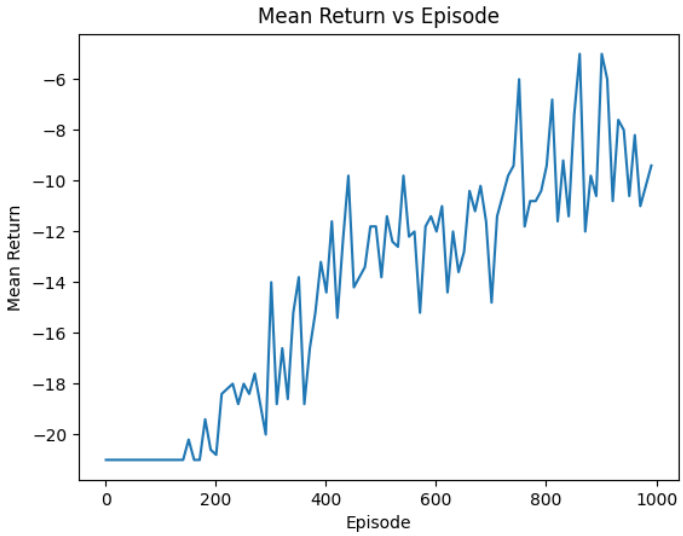


Figure 4: Mean return of 5 runs after training up to n episodes.

3 Discussion

3.1 Cart Pole

When implementing DQN, we trained exclusively on the much simpler Cart Pole environment to make sure that our code was functioning as it should before moving on to the more complex Pong environment. When the agent was able to achieve close to the maximum reward in an episode (475) within 1000 episodes, we felt that we were ready to move on to Pong.

Changing the training frequency to be less often, setting n to 4 or 8 instead of the default 1 results in the agent updating its Q-network less often. As can be seen in Figure 2, $n = 1$ reached its best policy earlier than $n = 4$, which reached its best policy earlier than $n = 8$. Although it was much quicker to train the agent using a higher value for n, it resulted in the agent taking longer to learn a good policy.

Initially when training we used a batch size of 32, meaning that 32 random samples from the replay memory were used when optimising the Q-network. Using this batch size, the agent managed to perform reasonably well and solved the environment relatively quickly. However, using a small batch size may lead to overfitting since the agent learns from only a small subset of the data at each training step. Changing the batch size to 64 resulted in the agent learning a good policy after fewer episodes, at around episode 400, and changing the batch size to 128 resulted in a good policy at around 350 episodes. It is clear that the agent learned a good policy after fewer episodes using a higher batch size. However, training took significantly longer since more computations have to be made during optimisation. Additionally, we can also observe that a better policy is rarely reached after the first peak w reached, as we observed with a batch size of 32.

Another hyperparameter which has a significant impact on learning but one that we have not experimented much with is the learning rate. Choosing a higher learning rate can help the agent learn more quickly, but can also lead to overfitting since larger steps are taken. Conversely, choosing a low learning rate can instead avoid overfitting, but make the learning process much slower.

3.2 Pong

For us, training a model on Pong took a significant amount of time to be able to perform actions that would produce a somewhat decent reward. Rendering a video replay from the agent playing Pong makes it clear that the agent is on the right track, managing to kick the ball and score points occasionally. Despite the agent not playing perfectly, we are satisfied with our results and feel confident that training the agent for a longer period would most likely result in better results.

In our Pong experiment, we limited the possible actions corresponding to either moving the paddle up or down in the game. However, looking at a video replay of the agent playing the game with the experience it had learned, we noticed that it would sometimes miss the ball by moving too fast, or "twitch" in scenarios it would have been better to stay still. For this reason, we tried adding another action corresponding to doing nothing or staying still. With three actions, the training became much slower, and no significant progress was made in 1000 episodes. It is likely the agent would eventually learn how to use all three actions, but the large increase in complexity also means that training would be slower.

3.3 Implementing the DQN

Implementing the DQN has been both interesting and challenging. It is more complex than the algorithms we had previously encountered, so it took some time to fully understand. The usage of neural networks was especially interesting, as neither of us had previously worked with them before. This added layer of complexity, as we had to understand both the neural network and the rest of the algorithm.

A consequence of this added complexity meant that debugging and troubleshooting became more difficult. A significant amount of time was spent trying to understand issues and the fixes for them. It can be hard to pinpoint the exact issue when the agent is not performing, and the addition of a neural network made the process even more opaque. The fact that the agent also took a long time to start to produce reasonable results extended that time even further.

3.4 Exploration vs. Exploitation

In future experiments, it may be suitable to explore how different methods of decreasing the ratio between exploration and exploitation may affect the learning process. Right now the agent uses a linear epsilon decay, given a value over how many episodes the decay should occur. Instead, one might consider experimenting with exponential decay or Reward Based Epsilon Decay (RBED), as explored by A. Maroti [4].

A potential downside of using linear decay of epsilon is that it may lead to over- or under-exploration. Early on in training, linear decay may reduce epsilon too quickly, resulting in the agent's ability to explore and learn an optimal policy. Also, later on in training, the agent may continue to explore longer than necessary which might lead to a slower convergence to an optimal policy.

References

[1] V. Mnih, K. Kavukcuoglu, D. Silver, A. A. Rusu, J. Veness, M. G. Bellemare, A. Graves, M. Riedmiller, A. K. Fidjeland, G. Ostrovski *et al.*, "Human-level control through deep reinforcement learning," *Nature*, vol. 518, no. 7540, pp. 529–533, Feb 2015.

[2] Farama Foundation. (2023) Cart pole. Accessed 2023-05-12. [Online]. Available: https://gymnasium.farama.org/environments/classic_control/cart_pole/

[3] ——. (2023) Pong. Accessed 2023-05-12. [Online]. Available: https://gymnasium.farama.org/environments/atari/pong/

[4] A. Maroti, "RBED: reward based epsilon decay," *CoRR*, vol. abs/1910.13701, 2019. [Online]. Available: http://arxiv.org/abs/1910.13701