# Deep Reinforcement Learning - assignment part 1

**Thijs Lukkien** [1] **Carlos Brito** [1]

## Abstract

Reinforcement learning is a subfield of machine learning where the goal is to optimize a given policy to maximize the rewards returned over time. The goal of this paper is to explore 5 different algorithms within this field and compare them. For this we will provide a short description of each and the methods we used to train and test them. Finally, discussing and concluding what results we obtained.

## 1. Introduction

In this project, we will be attempting to implement five Deep Reinforcement Learning (DRL) algorithms in the game of Catch. The goal of Reinforcement Learning (RL) is to learn policies that maximize the rewards in a sequential decision problem. The RL process is modeled after a Makarov Decision Process (MDP). The agent observes the environment, selects an action based on a policy and then receives a reward before passing to the next state. Then the goal of RL is to train a policy so that the reward return over time is optimized.

In this paper we will first describe the environment that was used to train and test our RL algorithms. Then we will provide a description of the algorithms that were implemented. Followed by describing the methods that we used to train and test out models. Additionally, we will provide the results that we obtained and discuss them. Finally, we will provide a conclusion to the results we obtained.

## 2. Environment

The environment that is used for this work is the *Catch* environment (Mnih et al., 2014). In the game of catch an agent performs certain actions with a paddle to catch a ball that is falling vertically. This is performed within an environment that is represented with a $21 \times 21$ grid, where the paddle is located on the bottom half of the grid. In this grid, the ball is falling with a vertical speed of $V_y = -1 \quad cell/s$ and has a vertical speed of $V_x \in \{-2, -1, 0, 1, 2\}$. The paddle may perform three possible actions: left, right, or no action. This environment has been implemented using the *Gymnasium* environment (Towers et al., 2024) as a base. Gymnasium is an open-source Python library that is widely used to study reinforcement learning algorithms. For this assignment, the grid has been scaled up by a factor of 4, resulting in a $84 \times 84$ grid.

*Figure 1.* Representation of the game of Catch

## 3. Networks

In this section, we will describe the architectures that have been implemented within the code. However, we will first explain the background knowledge on which the five implemented algorithms are built.

### 3.1. Makarov Decision Process (MDP)

MDPs are used for solving sequential decision-making problems in stochastic environments and are defined by the tuple $\mathcal{M} := (\mathcal{S}, \mathcal{A}, \mathcal{P}, \Re, \gamma)$ where:

$\mathcal{S}$ is the state space
$\mathcal{A}$ is the action space
$\mathcal{P} : \mathcal{S} \times \mathcal{A} \times \mathcal{S} \to [0, 1]$ is the transition function
$\Re : \mathcal{S} \times \mathcal{A} \times \mathcal{S} \to \mathbb{R}$ is the reward function returning $r_t$
$\gamma$ is the discount factor

This provides a formal framework in which (D)RL algorithms can learn to navigate towards optimal rewards, such as catching the ball.

### 3.2. Q-Learning

Q-Learning is an off-policy learning algorithm that uses equation 1 to update each state-action pair.

$$Q(s_t, a_t) := Q(s_t, a_t) + \alpha \left[ r_t + \gamma \max_{a \in \mathcal{A}} Q(s_{t+1}, a) - Q(s_t, a_t) \right] \quad (1)$$

Where $s \in \mathcal{S}$ and $s_t$ and $a_t$ correspond to a certain state and action at a point in time. This state-action pair is assigned a Q-value that represents a 'benefit score' of taking that action in that state. The main issues of Q-learning is its poor generalizability, memory consumption, and performance overestimation. The first two of these problems we aim to solve with Deep Q-Networks (DQNs).

### 3.3. Deep Q-Network

DQNs can generalize better, as this method uses a model to estimate Q-values where otherwise the tabular version of Q-learning would need to remember it. This makes DQNs a continuous method, that does not *need* to have seen a sample in order to predict the best action.

The goal of the DQN is to train a Convolutional Neural Network (CNN) using the objective function seen in 2. This is used to obtain the best $Q$ values possible and then by using the $\max$ operator we can choose the best possible action in a greedy way.

$$\mathcal{L}(\theta) = \mathbb{E}_{\langle s_t, a_t, r_t, s_{t+1} \rangle \sim U(D)} \left[ \left( r_t + \gamma \max_{a \in \mathcal{A}} Q\left(s_{t+1}, a; \theta^-\right) - Q\left(s_t, a_t; \theta\right) \right)^2 \right] \quad (2)$$

### 3.4. DQN Novel Components

#### 3.4.1. Experience Replay Buffer ($D$)

This novel component was introduced by Lin (Lin, 1992) in 1992. The concept is that throughout the learning we save all the trajectories that we obtain after interacting with the environment as seen in equation 3.

$$\tau = \langle s_t, a_t, r_t, s_{t+1} \rangle \quad (3)$$

From these trajectories, we save them in a matrix denoted as $D$.This matrix can be thought of as a Queue since when it is full the older trajectories are removed to make space for the new trajectories.

#### 3.4.2. Minibatch Training

This technique allows the agent to learn by sampling the experience replay buffer uniformly.

$$\mathbb{E}_{\sim U(D)} \langle s_t, a_t, r_t, s_{t+1} \rangle \quad (4)$$

The batch of samples are constructed so that the expectation over these trajectories is minimized.

#### 3.4.3. Target Network $\theta^-$

When sampling the trajectories from the buffer each trajectory must be related to its Temporal Difference (TD) target. To calculate this value another function approximation $\theta^-$ is introduced and its only goal is to estimate equation 5.

$$r_t + \gamma \max_{a \in \mathcal{A}} Q\left(s_{t+1}, \theta^-\right) \quad (5)$$

This network does not get explicitly trained. However, every certain amount of iteration we update $\theta^- \leftarrow \theta$

#### 3.4.4. Reward Clipping

This technique was used in order to normalize the rewards and penalties across the different atari environments that the DQN was originally designed to perform. This was achieved by setting the positive rewards of $r_t$ to 1, the negative rewards to -1, and the rewards with 0 unchanged. This allowed the network to avoid large updates so that it could perform updates in a smoother way.

### 3.5. Deep Double Q-Network

The most important quality of the DQN is that it makes use of the $\max$ operator. When training the agent we will obtain as many Q values as actions that the agent can perform. After calculating this value we then choose the action that has the highest Q value. Due to this choice, the algorithm suffers from an overestimation bias in equation 6.

$$\max_{a \in \mathcal{A}} Q(s_{t+1}, a) \quad (6)$$

The reason is that the maximum Q value that is used corresponds to the maximum value of the state and not the maximum expected value. In order to solve this issue Double Q-learning is introduced by using two separate $Q$ functions and then randomly preferring one or the other during training.

The DDQN goes further and provides a generalization of this idea in equation 7.

$$y_t^{DDQN} = r_t + \gamma Q\left(s_{t+1}, \underset{a \in \mathcal{A}}{\operatorname{argmax}} Q(s_{t+1}, a; \theta); \theta^-\right). \quad (7)$$

### 3.6. Dueling Architecture

The dueling architecture goes further than just learning the $Q^\pi(s, a)$ function and attempts to estimate the state values,

advantages, and state-action values simultaneously. To accomplish this three different streams are needed to estimate each of the mentioned value functions. These streams are initialized with their own parameters and are represented by $\theta^{(.)}$

$$Q(s,a;\theta^{(1)},\theta^{(2)},\theta^{(3)} = V(s,\theta^{(1)},\theta^{(3)}) + (A(s,a,\theta^{(1)},\theta^{(2)}) - \max_{a_{t+1}\in\mathcal{A}} A(s,a_{t+1},\theta^{(1)},\theta^{(2)}) \tag{8}$$

It is important to note that the state value function is only estimated and is not learned.

### 3.7. Deep Quality-Value Learning

For Deep Quality-Value learning, two sets of parameters are learned. One for the state-action value and one for the state value. Similarly to a DQN, a target network is used to stabilize learning. Differently, however, is that the value parameters are copied such that $V^- \leftarrow V$. $Q$ and $V$ are both learned using a separate loss function (and thus also optimizer)

$$\mathcal{L}(\phi) = \mathbb{E}_{\langle s_t,a_t,r_t,s_{t+1}\rangle\sim U(D)} \left[\left(r_t + \gamma V\left(s_{t+1},\phi^-\right) - V\left(s_t,\phi\right)\right)^2\right] \tag{9}$$

$$\mathcal{L}(\theta) = \mathbb{E}_{\langle s_t,a_t,r_t,s_{t+1}\rangle\sim U(D)} \left[\left(r_t + \gamma V\left(s_{t+1},\phi^-\right) - Q\left(s_t,a_t;\theta\right)\right)^2\right] \tag{10}$$

Here $\phi$ is used in the network used to construct the temporal difference errors. It is important to note that we are learning approximations of the state-value and state-action values using the equations 9, 10.

### 3.8. Deep Quality-Value Max Learning

The DQV Max learning differs from the DQV since the calculation of two separate targets is required. Furthermore, $\theta$ and $\phi$ are also used in different places within the DQV max algorithm. This algorithm uses the following equations 11, 12.

$$\mathcal{L}(\phi) = \mathbb{E}_{\langle s_t,a_t,r_t,s_{t+1}\rangle\sim U(D)} \left[\left(r_t + \gamma \max_{a\in\mathcal{A}} Q\left(s_{t+1},a,\theta^-\right) - V\left(s_t,\phi\right)\right)^2\right] \tag{11}$$

$$\mathcal{L}(\theta) = \mathbb{E}_{\langle s_t,a_t,r_t,s_{t+1}\rangle\sim U(D)} \left[\left(r_t + \gamma V\left(s_{t+1},\phi\right) - V\left(s_t,a_t;\theta\right)\right)^2\right] \tag{12}$$

## 4. Methods

In order to test the different algorithms within the game of Catch we have used the following experimental setup:

We have set the number of episodes that each run will have to 12000. Then every 100 episodes we take a copy of our $\theta$ model and we test this model for 50 episodes. After running this test we averaged the rewards we obtained and then returned this value that was later saved in a csv file. This method resulted in 120 average rewards of this test model. In the figures, we can see a solid line that represents the average value of the reward and around this line, we have the standard deviation represented as a cloud around this line.

The code that was developed proved to be computationally intensive and for this reason it was run within the Habrok HPC cluster.

## 5. Results

In this section, we will present the results that we have obtained after following the method presented previously.

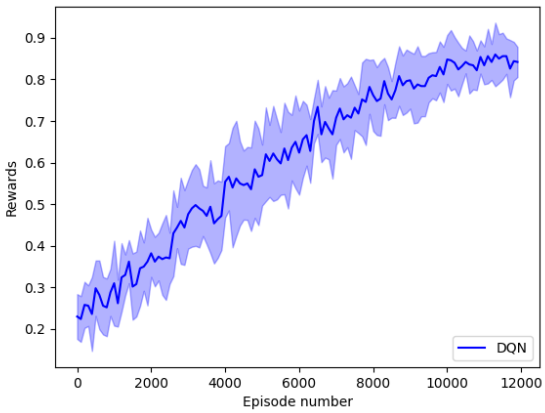


*Figure 2.* Rewards of the DQN algorithm after 10 runs, within each run the model after 100 eps (of the 12000) was then tested for 50 eps, these values were returned and then averaged out

## 6. Discussion

After analyzing the plots we can see that in general the models seem to have not fully finished training yet. This will make the final discussion about the performance of each one. However, if we observe figure 7 we can see that on average the Dueling model had the best average rewards of the models that were implemented. Furthermore, the DDQN model returned on average the lowest reward of all the models.

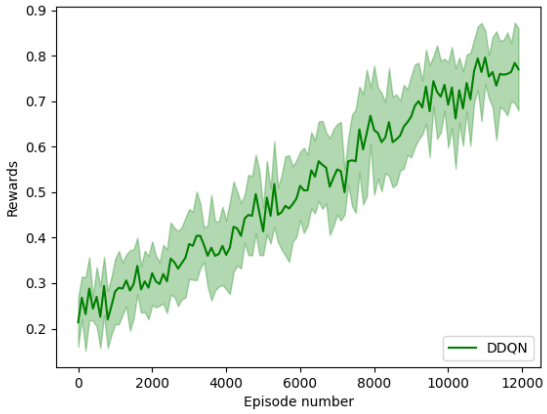


*Figure 3.* Rewards of the DDQN algorithm after 10 runs, within each run the model after 100 eps (of the 12000) was then tested for 50 eps, these values were returned and then averaged out

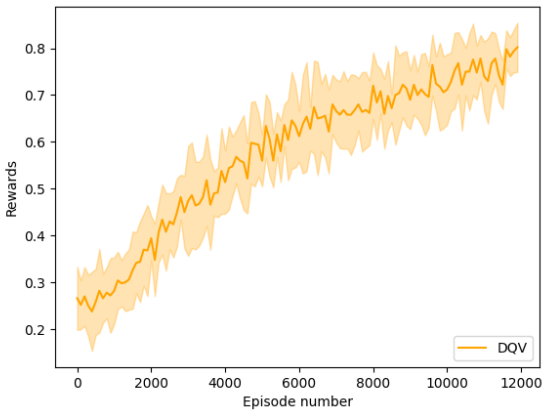


*Figure 5.* Rewards of the DQV algorithm after 10 runs, within each run the model after 100 eps (of the 12000) was then tested for 50 eps, these values were returned and then averaged out

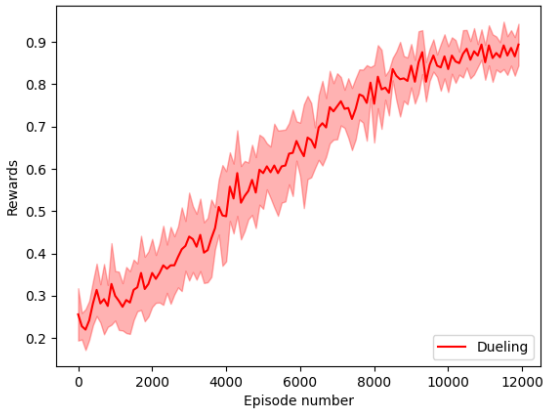


*Figure 4.* Rewards of the Dueling algorithm after 10 runs, within each run the model after 100 eps (of the 12000) was then tested for 50 eps, these values were returned and then averaged out

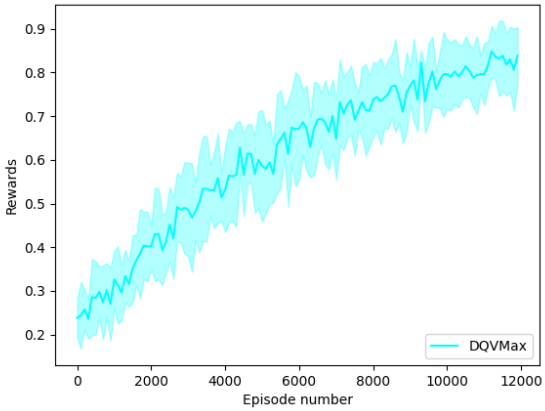


*Figure 6.* Rewards of the DQV Max algorithm after 10 runs, within each run the model after 100 eps (of the 12000) was then tested for 50 eps, these values were returned and then averaged out

Observing the standard deviation of each algorithm we can see that most of the models have more or less the same value. The DDQN model however taking into consideration the standard deviation had the lowest potential reward return.

We can observe from the figures that all the models were able to learn from the environment and perhaps with more episodes they may have been able to plateau and arrive at a more stable return value. For this reason we can recommend that more episodes should be used so that the models arrive at a maximized return value.

## 7. Conclusion

In conclusion, we have implemented 5 different DRL models using the Catch environment. Then we have evaluated each model by running each model 10 times each. These runs had 12000 episodes each and after 100 episodes a copy of the $\theta$ is used to test the model for 50 episodes. These values produced 120 rewards after each run and used to create the figures presented. We can see that the models have not fully finished training. Nevertheless, the dueling model seems to perform the best while DDQN had on average the worst performance.

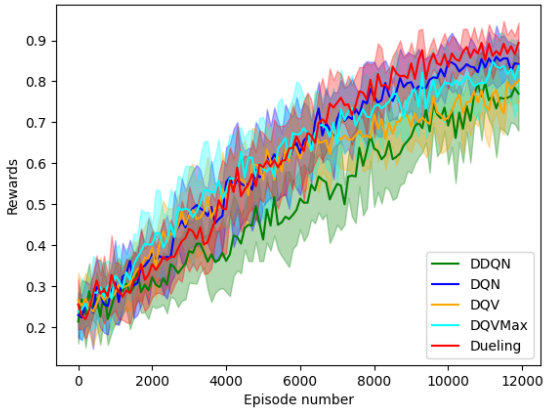


*Figure 7.* General comparison of the rewards after 10 runs, within each run of the models after 100 eps (of the 12000) were then tested for 50 eps, these values were returned and then averaged out

## References

Lin, L.-J. Self-improving reactive agents based on reinforcement learning, planning and teaching, May 1992. URL http://dx.doi.org/10.1007/BF00992699.

Mnih, V., Heess, N., Graves, A., and Kavukcuoglu, K. Recurrent models of visual attention, 2014.

Towers, M., Terry, J. K., Kwiatkowski, A., Balis, J. U., Cola, G., Deleu, T., Goulão, M., Kallinteris, A., KG, A., Krimmel, M., Perez-Vicente, R., Pierré, A., Schulhoff, S., Tai, J. J., Tan, A. J. S., and Younis, O. G. Gymnasium, February 2024. URL https://doi.org/10.5281/zenodo.10655021.