Reinforcement Learning
Assignment 1: Q-Learning - Tabular & Deep

Daniël Zee (s2063131) [1]

Abstract

In this assignment, we study the use of Deep Q-Networks (DQN) for approximating the optimal state-value function in the CartPole-v1 environment. A naive DQN agent using a neural network function approximator was implemented and used for evaluating the effect of hyperparameters, including learning rate, update frequency, epsilon decay, and network size. We then extended the naive agent to full DQN, introducing Experience Replay and a Target Network. Our experiments showed that the combination of both ER and TN significantly improves stability and enables faster convergence to the optimal return.

1. Introduction

Q-Learning is one of the most well known algorithms in the field of reinforcement learning (RL). In tabular implementations, Q-Learning is able to learn the optimal policy π^* for an agent in an environment specified as an Markov decision process (MDP). This is achieved by keeping track of a tabular representation of the state-action values $Q(s, a)$ during training, for every state s and action a in the state and action spaces of the environment. However, when state spaces for environments are very large, this approach becomes infeasible due to memory constrains and the increased time it would take for an agent to explore the entire state space. For continuous states spaces, it would even be impossible. For these scenarios, a function approximation of the state-action function is required, replacing the tabular representation by a fixed set of parameters θ, such that $Q_\theta(s, a) \approx Q(s, a)$.

The most prominent function approximators used for this purpose are neural networks. Mnih et al. (Mnih et al., 2013; 2015) were the first to successfully implement Q-learning using a neural network function approximator in an algorithm called Deep Q-Network (DQN). Besides the use of a neural network, two more extensions were introduced to increase the learning stability: Experience Replay (ER) and Target Network (TN).

For this assignment, we have implemented a Q-learning agent which extends the tabular algorithm using a neural network approximator, and performed an ablation study to test the effect of hyperparameter configurations. We then extended this naive agent to full DQN by adding ER and TN, and studied the changes in performance using combinations of these extensions. The environment used for our experiments was CartPole-v1, by Gymnasium (Towers et al., 2024).

2. Theory

The goal of Q-learning is to derive the optimal action-value function $Q^*(s, a)$. This function returns the expected future return, the discounted sum of future rewards, of taking action a in state s following the Bellman equation,

$$Q^*(s, a) = \mathbb{E}_{s' \sim T(s'|s,a)} \left[r + \gamma \max_{a'} Q^*(s', a') \right] \quad (1)$$

with T being the transition funcion in the environment, r being the reward received after taking action a in state s, and γ being the discount factor used to decrease the contribution of future rewards. Greedily selecting the action with the highest expected return in $Q^*(s, a)$ then gives us the optimal policy π^*. Q-learning calculates $Q^*(s, a)$ using one-step temporal difference learning (TD). One step TD methods update $Q(s, a)$ based on the current estimate of $Q(s', a')$, without waiting for a full episode to end. This process of updating our estimates based on other estimates is called bootstrapping. The update for Q-learning at every training step looks as follows,

$$Q(s, a) \leftarrow Q(s, a) + \alpha \left[r + \gamma \max_{a'} Q(s', a') - Q(s, a) \right] \quad (2)$$

with α being the learning rate. The agent explores the environment following a behavior policy which balances

[1]Leiden Institute of Advanced Computer Science (LIACS), Leiden University, Leiden, The Netherlands. Correspondence to: Daniël Zee <d.f.zee@umail.leidenuniv.nl>.

Proceedings of the 42^{nd} International Conference on Machine Learning, Vancouver, Canada. PMLR 267, 2025.

exploration and exploitation. As this policy is different than π^*, this is called off-policy learning.

DQN uses a neural network as an approximator to estimate $Q^*(s, a)$, also called a Q-network. To train the Q-network, we can´t directly apply the Bellman update rule in equation 2, because neural networks are optimized using gradient-based methods rather than direct assignment. We therefore require a differentiable loss function. The loss function then becomes the Mean Squared Error (MSE) between the predicted state-action value and the target value. The parameters weights θ are optimized by minimizing the loss function at each update iteration i,

$$\begin{aligned} y_i &= \mathbb{E}_{s' \sim T(s'|s,a)} \left[r + \gamma \max_{a'} \hat{Q}_{\theta_{i-1}}(s', a') \right] \\ L_i(\theta_i) &= \mathbb{E}_{s,a \sim \rho(s,a)} \left[(y_i - Q_{\theta_i}(s, a))^2 \right] \end{aligned} \tag{3}$$

where y_i denotes the target value and ρ the behavior distribution over s and a. The gradient over the loss function is then used move θ in the direction which minimizes the loss function, using a learning rate α.

As update targets depend on the previous networks weights θ_{i-1}, the loss function of DQN minimizes a moving target. In the tabular case, this bootstrapping behavior would still lead to convergence, as we are able to calculate the exact state-action function because all possible states can be visited during learning. For continuous problems where a Q-network is used, this is not the case, and convergence is therefore not guarantied. The risk over overshooting a moving target using function approximation is real, resulting in an unstable optimization process. DQN tries to mitigate this bootstrapping instability using infrequent weight updates (Mnih et al., 2015). A second Q-network, called the Target Network $\hat{Q}$, is used for generating the target values in y_i. After a set number of parameter updates in Q, the weights θ of Q are then copied to $\hat{Q}$. As a result, the weights of the target network now move slower than the network on which we act during learning, reducing the instability caused by moving targets.

The loss function in equation 3 takes the average of errors for a certain set of of state-action pairs observed in the environments. If we were to update θ after every step the agent takes in the environment, this would be a single state-action pair. But updates can also be performed after a number of sequential steps in the environment, resulting in a loss function with more complete gradient information, optimizing for a broader part of the state space. Subsequent training states are however highly correlated, which can introduce bias in the learning for more frequently visited areas in the state space. This can even result in behavior being forgotten for less frequently visited areas in the state space if the agent focuses too much on exploitation, with little exploration. DQN tries to break this correlation of states during learning using Experience Replay (Mnih et al., 2013). A replay buffer is introduced, storing a set number of the previously explored state-action pairs, together with their rewards. Every update step, we now calculate the loss function using a randomly sampled batch from the replay buffer. This breaks the correlation between training examples, resulting in an improvement of the state space coverage during updates.

Algorithm 1 shows the pseudocode for DQN with Experience Replay and Target Network.

Algorithm 1 Deep Q-Network with Experience Replay and Target Network

Initialize replay buffer $\mathcal{D}$ with size N
Initialize online network $Q(s, a; \theta)$ with random weights θ
Initialize target network $Q_{\text{target}}(s, a; \theta^-)$ with $\theta^- \leftarrow \theta$
for episode = 1 to M **do**
  Initialize state s_0
  for t = 1 to T **do**
    Select action a_t using ϵ-greedy policy:

$$a_t = \begin{cases} \text{random action} & \text{with probability } \epsilon \\ \arg\max_a Q(s_t, a; \theta) & \text{otherwise} \end{cases}$$

    Execute action a_t, observe reward r_t, next state s_{t+1} and termination status d_t
    Store transition $(s_t, a_t, r_t, s_{t+1}, d_t)$ in $\mathcal{D}$
    Sample random minibatch with size B of transitions from $\mathcal{D}$
    Compute target:

$$y_t = \begin{cases} r_t & \text{if } d_t = 1 \\ r_t + \gamma \max_a Q_{\text{target}}(s_{t+1}, a; \theta^-) & \text{otherwise} \end{cases}$$

    Perform gradient descent on loss with respect to the online network weights θ:

$$L(\theta) = \frac{1}{B} \sum_i (y_t - Q(s_t, a_t; \theta))^2$$

    Every C steps update target network weights:

$$\theta^- \leftarrow \theta$$

  end for
end for

3. Experiments

We have performed experiments using a modular implementation of a DQN agent, where ER and TN can separately be

turned on or off, in order to study the effect these extensions have on the stability of learning. On the agent with both extensions turned off, which we call Naive DQN, an ablation study was performed over most hyperparameters to study their impact on learning stability.

3.1. Environment

All experiments were performed on the CartPole-v1 environment (Towers et al., 2024). This environment simulates a classic control problem where a pole is attached to a cart, which moves along a frictionless track. The goal is to balance the pole by applying forces in the left and right direction on the cart. The action space is binary and corresponds to an actions for pushing the cart to the left and to the right. The state space consists of 4 continuous values: the cart position, the cart velocity, the pole angle, and the pole angular velocity. A default reward $+1$ is given for every step taken, with an episode time limit of 500 steps. An episode is terminated if either the pole angle is greater that $\pm 12^\circ$ or the cart position is greater than ± 2.4.

3.2. Naive DQN

To study the effects of hyperparameters configurations on the performance of our Naive DQN implementation, we ran each agent for a total of 10^6 environment steps. After every 1000 environment steps, we evaluated the learned Q-network by calculated the episode return over the greedy policy from a generated starting state. Each configuration was run for 5 repetitions, over which the average evaluation returns and confidence intervals were calculated and plotted. Both the average evaluation returns and confidence intervals have been smoothed using a Savitzky-Golay filter with window size 31.

The following hyperparameters were included in our ablation study:

1. Learning rate (α). This determines the size of the step that is taken in the direction of the gradient of the loss function. For all experiments, the Adam (short for Adaptive Moment Estimation) optimizer was using as gradient descent method (Kingma & Ba, 2014).

2. Epsilon decay rate. Our agent uses the ϵ-greedy policy with a decaying epsilon value to move from high exploration to high exploitation. The initial epsilon value is 1 cross all experiments and is exponentially decreased after every training episode by multiplication with the epsilon decay rate.

3. Update frequency. Also called the update-to-data ratio, this specifies the number of sequential environment steps that are taken before updating the Q-network parameters.

4. Hidden layer dimension. Our Q-network is represented by a relatively simple 2 layer feed forward neural network with 4 inputs, one for each state space dimension, and 2 outputs, one for each action. The number of neurons in the hidden layers is specified with this hyperparameter.

Figure 1 shows the learning curves for our Naive DQN agent using three different learning rates. We see that for both $\alpha = 0.001$ and $\alpha = 0.01$, the agent learns much quicker in the early stages. But eventually, for all three configurations, the evaluation returns start to collapse and learning becomes unstable. We do however see that it takes longer for $\alpha = 0.001$ to collapse, which is to be expected as the parameter weights are changed more slowly. This instability could be the result of the bias introduced in the learning when epsilon decreases, and exploitation increases, resulting in the agent getting stuck in a local optimum and overfitting for the most frequently visited state-action pairs.

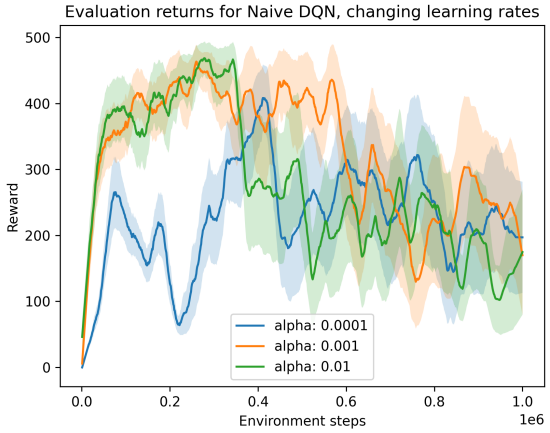

Figure 1. Effects of different learning rate values on the greedy evaluation over the Q-network during learning, using an epsilon decay rate of 0.9995, update frequency of 4, and hidden layer dimension of 128.

Figure 2 shows the learning curves using three different epsilon decay rates. As suspected, increasing the decay rate to 0.9999 increases the learning stability significantly. The final epsilon value after training has only dropped down to around 0.4, meaning that there is still a significant amount of exploration in the final phases of the learning process. The lack of exploitation does however mean that it takes a long time for the agent to reliably reach the full return value of 500, as the agent does not strongly capitalize on the estimated successes in previously visited regions of the state space. Decreasing the decay rate to 0.999 results in an even earlier collapse of the evaluation returns, which is in line with out assumptions.

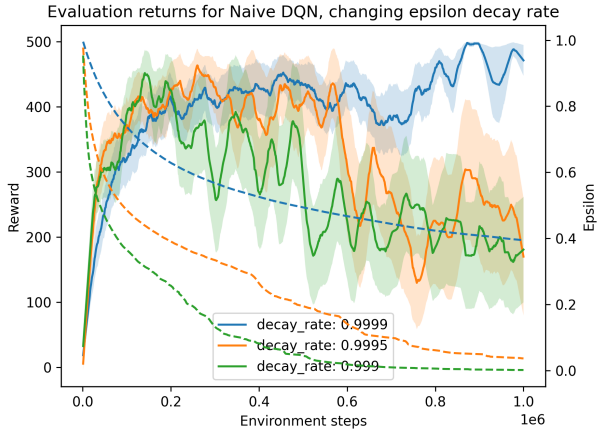

Figure 2. Effects of different epsilon decay rate values on the greedy evaluation over the Q-network during learning, using a learning rate of 0.001, update frequency of 4, and hidden layer dimension of 128. The epsilon decay during learning is shown in the same color.

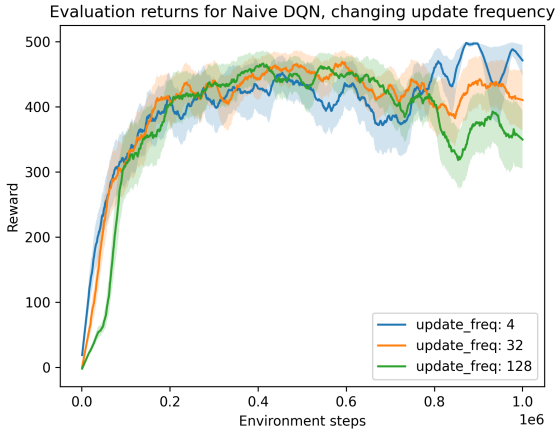

Figure 3. Effects of different update frequency values on the greedy evaluation over the Q-network during learning, using a learning rate of 0.001, epsilon decay rate of 0.9999, and hidden layer dimension of 128.

Figure 3 shows the learning curves using three different update frequency sizes, using the most stable learning rate and epsilon decay rate found thus far. We observe that increasing the update frequency results in slower learning in the early stages. This can be explained by the fact that there are more environment steps between updated of the Q-network, making the agent slower to adapt to changes. The upside however is that each update is performed using a more complete gradient landscape, as more state-action pairs are included in the loss function calculation. Increasing the update frequency does seem to hurt the learning process in the final stages however. A possible explanation for this could be that when the agent is close to the optimal policy, the infrequent updates might come too slow for the agent to act on the small but meaningful improvements required to reach the optimal return.

Figure 4 shows the learning curves using three different number of neurons in the hidden layers of the Q-network. The required size of a neural network to give a good approximation of the desired function is usually dependent on the dimensionality of the input and output (Karsoliya & Azad, 2012). As CartPole is a relatively simple environment, decreasing the number of neurons in the hidden layers should be possible without hurting performance. This assumption is confirmed by inspecting the learning curve using 64 neurons per hidden layer. Decreasing even further to 16 neurons per hidden layer does however start to negatively impact the learning results, indicating that at this size, the Q-network is too small to capture the complexity of the theoretical state-value function we are trying to approximate.

3.3. Full DQN

We now extend our Naive DQN agent to include ER and TN. Each of these extensions can be included independently from each other in our implementation. To study the effects of each individual extension on the learning stability, we tested each of the 4 configurations, with and without the extensions. The hyperparameters for this experiment were kept fixed at $\alpha = 0.001$, an epsilon decay rate of 0.995, an update frequency of 4, and 128 hidden layer neurons. A lower epsilon decay rate compared to the most stable rate for Naive DQN was chosen to test if Full DQN performs better at higher levels of exploitation. ER and TN however also introduce new hyperparameters:

1. Replay memory size. This determines the size of the replay buffer. State-action pairs are added and removed from the buffer in FIFO order, when the buffer is full. For this experiment, a value of 10000 was used.

2. Minibatch size. This determines the size of the mini-batch sampled from the replay buffer at each update step. For this experiment, a value of 32 was used.

3. Target network update frequency. This specified the interval between the parameter weight updates in the target network $\hat{Q}$. For our implementation, this is the number of training episodes between updates. For this experiment, a value of 5 was used.

Figure 5 shows the learning curves for the 4 configurations of extension inclusions. We see that including just ER increases the learning speed in the early stages, which is to

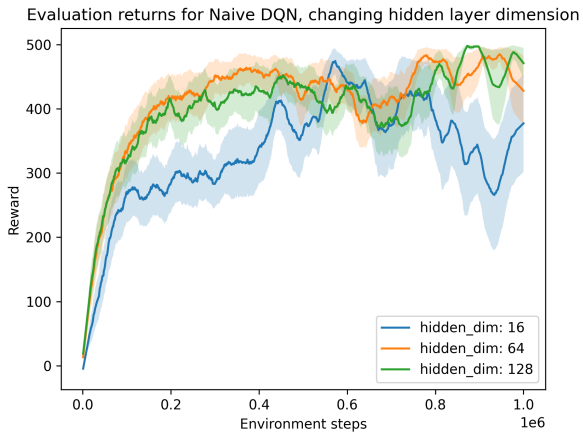

Figure 4. Effects of different hidden layer dimension values on the greedy evaluation over the Q-network during learning, using a learning rate of 0.001, epsilon decay rate of 0.9999, and update frequency of 4.

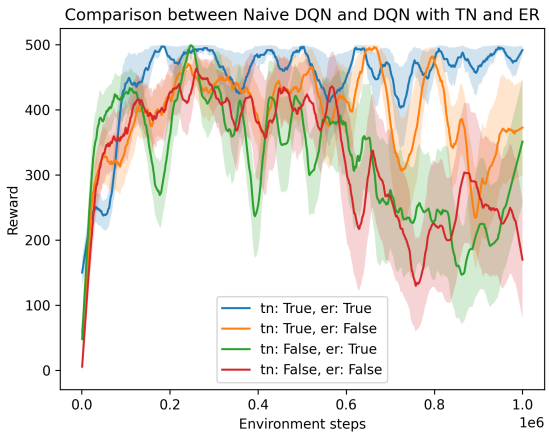

Figure 5. Effects of extending Naive DQN with Target Network and Experience Replay on the learning stability, using a learning rate of 0.001, epsilon decay of 0.9995, update frequency of 4, hidden layer dimension of 128, replay memory size of 10000, minibatch size of 32, and target network updates after every 5 training episodes.

be expected as we are calculating the loss function over 32 state-action pairs instead of just 4. In the long run however, the stability stays similar to Naive DQN, indicating that just ER is not enough to mitigate all factors causing learning instability. Including just TN paints a similar, but slightly more optimistic picture. Here we observe that while the learning still diverges at the end, it does so at a slower pace, indicating that some of the causes of instability have been improved. When combining TN and ER, we observe that while ER on its own did not improve stability, together they complement each other nicely, completely mitigating the downward trend of the evaluation returns.

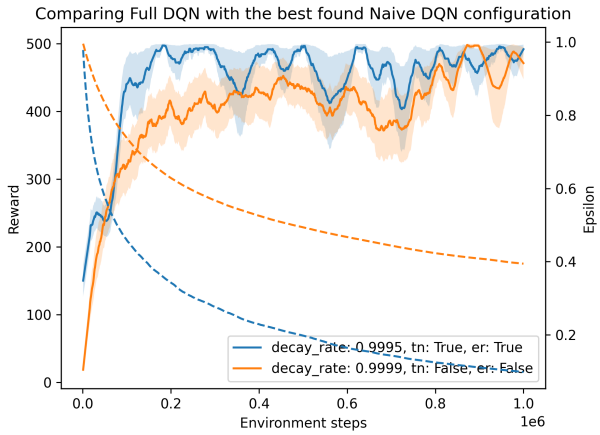

Figure 6. Comparing the learning curves and the epsilon decay between the best performing Naive DQN configuration and full DQN.

4. Discussion

Our experiments showed that Naive DQN does not reliably converge without running the risk of divergent learning for most hyperparameter configurations. We have observed that for Naive DQN to keep stable evaluation returns during training, a very slow transition from exploration to exploitation was required, ending at a much lower rate of exploitation usually required to converge to the optimal reward. Increasing the update frequency of the Q network decreases the algorithms runtime as there are less update steps needed to be computed. However, a negative impact on learning in the final stages was observed when increasing the update frequency. We have also seen that the number of neurons in the hidden layers of the Q-network can be lowered to 64 without degrading the learning results, indicating that this network size is enough to capture the complexity required for this environment. Increasing the hidden layer dimension to 128 showed no significant change in the algorithms runtime however.

The experiments on Full DQN showed that TN and ER alone are not enough to combat the learning instabilities observed using a lower epsilon decay rate. Together however, they are nicely able to keep the learning stable around the optimal episode return. Figure 6 shows the best performing Naive DQN learning curve next to the one for Full DQN, also showing the epsilon decay scheme. Because Full DQN is able to increase exploitation much more quickly, the optimal evaluation return can be reached much sooner.

In order to gain more insight into the impact hyperparameter configurations have on the learning stability, more experiments would need to be performed using different configurations. We could also experiment with a linear epsilon decay scheme, comparing to the exponential one used here. Different neural network topologies could also be explored, using more or fewer hidden layers or with an asymmetric number of neurons in the hidden layers. Finally, an ablation study for the hyperparameters associated to TN and ER would be insightful, as these were kept constant during our experiments.

5. Conclusion

We studied the use of DQN for approximating the optimal state-value function for the CartPole environment. By implementing a modular DQN agent, where we can toggle the use of Target Network and Experience Replay, we have shown that a Naive DQN agent, without these extensions, suffers from divergent learning when moving too quickly from exploration to exploitation. A Full DQN agent was shown to overcome this by breaking the correlation of subsequent states during learning and limiting the instability caused by moving targets during bootstrapping. This as a result allows the Full DQN agent to be trained on a more aggressive epsilon decay scheme, resulting in a quick and stable convergence to the optimal epeside return.

References

Karsoliya, S. and Azad, M. A. K. Approximating number of hidden layer neurons in multiple hidden layer bpnn architecture. 2012. URL https://api.semanticscholar.org/CorpusID:38704853.

Kingma, D. and Ba, J. Adam: A method for stochastic optimization. *International Conference on Learning Representations*, 12 2014.

Mnih, V., Kavukcuoglu, K., Silver, D., Graves, A., Antonoglou, I., Wierstra, D., and Riedmiller, M. Playing atari with deep reinforcement learning. 12 2013.

Mnih, V., Kavukcuoglu, K., Silver, D., Rusu, A. A., Veness, J., Bellemare, M. G., Graves, A., Riedmiller, M. A., Fidjeland, A. K., Ostrovski, G., Petersen, S., Beattie, C., Sadik, A., Antonoglou, I., King, H., Kumaran, D., Wierstra, D., Legg, S., and Hassabis, D. Human-level control through deep reinforcement learning. *Nature*, 518:529–533, 2015. URL https://api.semanticscholar.org/CorpusID:205242740.

Towers, M., Kwiatkowski, A., Terry, J., Balis, J., Cola, G., Deleu, T., Goulão, M., Kallinteris, A., Krimmel, M., KG, A., Perez-Vicente, R., Pierré, A., Schulhoff, S., Tai, J., Tan, H., and Younis, O. Gymnasium: A standard interface for reinforcement learning environments, 07 2024.