Reinforcement Learning Assignment 2: Deep Q-Learning

Georgios Doukeris (s3536149) Evan Meltz (s3817911) Giulia Rivetti (s4026543)

Abstract

Traditional tabular methods for decision-making in discrete state spaces cannot be applied in complex environments where the state space is high-dimensional. In such cases Deep Q-Learning Networks (DQN) can be used, since they employ deep neural networks for function approximation. In this report we have implemented a DQN agent specifically for the Cartpole environment, where the agent needs to learn how to balance a stick on a cart. After having performed hyper-parameter tuning to optimise the agent's performance, we have also conducted some experiments by employing several exploration strategies, DQN implementations, e.g. double DQN and dueling DQN, as well as DQN components, such as experience replay and target network. From our findings we have observed that DQN demonstrates effective learning in the Cartpole environment.

1. Introduction

Reinforcement Learning (RL) algorithms make use of tabular methods for decision-making tasks in discrete, low-dimensional state spaces, as seen in the previous assignment. However, these methods fall short in complex environments with high-dimensional state spaces. To address this issue, in this assignment the focus is switched to the use of deep neural networks for function approximation, specifically using Deep Q-Learning (DQN). The environment used in this assignment is the Cartpole environment, which utilises a cart that moves along a horizontal axis, where the goal is to move the cart whilst always ensuring the pole stays upright. Our experiments include implementing and comparing various architectural choices, exploration strategies, as well as the novel components of DQN, such as experience replay and target networks.

In Section 2, we discuss the Cartpole environment and its components, laying the groundwork for our DQN agent's implementation. Section 3 presents the methodology behind the DQN agent, detailing the algorithm and the architecture. Then, in section 4.1 we have tuned different hyper-parameters to optimise the performance of our agent; in section 4.2 we have performed an ablation study to see how disabling the usage of the experience replay and target network affected the performance of the model; sections 4.3 and 4.4 shows the results of employing different exploration strategies and different DQN implementations respectively.

.

2. Environment

The Cartpole environment is a classic reinforcement learning problem, where there is a pole attached to a cart, which can move along a frictionless track. The pole starts upright and the goal is to prevent it from falling over by controlling the cart. The observation or state s_t that can be retrieved by

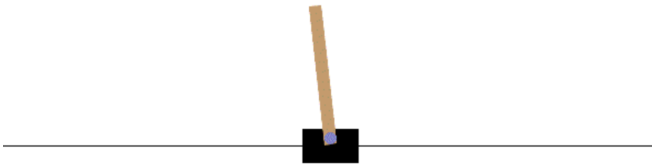

Figure 1. Illustration of the Cartpole environment. Source: (OpenAI)

the environment, represents the position and velocity of the cart, as well as the angle and angular velocity of the pole. Within this environment, the agent can take two different actions: push the cart to the right ($a_t = 1$) or to the left ($a_t = 0$) (OpenAI). At each time-step that the pole stays upright, the agent receives a reward $r_{t+1} = 1$. An episode ends when either the pole tips over a certain angle limit, the cart moves outside of the environment edges, or a maximum number of steps is reached.

The goal of the agent is to learn a policy $\pi(a_t|s_t)$ so as to maximize the sum of rewards in an episode $\sum_{t=0}^{T} \gamma^t \cdot r_t$, where γ is the discount factor that discounts future rewards relative to immediate rewards, in order to let the agent focus on obtaining rewards quickly (TensorFlow).

3. DQN Agent

DQN (Deep Q-Networks) is a reinforcement learning algorithm developed by DeepMind in 2015, which combines

the principles of deep neural networks with Q-learning, enabling agents to learn optimal policies in complex environments (Dhumne). It allows for the achievement of stable deep reinforcement learning by breaking correlations between subsequent states, and slowing down changes to parameters in the training process. These improvements are accomplished as a result of two methods: experience replay and infrequent weight updates (Plaat, 2023).
In order to tackle the Cartpole problem, we have implemented a DQN agent that learns to move the cart in such a way that the pole stays upright.

Algorithm Algorithm 1 (Mnih et al., 2013) shows the pseudo-code that we have followed to implement our DQN agent. First of all, we have initialized the core components of the DQN:

- the experience replay buffer, which is a cache of previously explored states(Plaat, 2023) - see Paragraph 4.2 for further details;
- a neural network, that takes in an observation and outputs an expected total reward for each possible action it can take;
- a target network, which is a copy of the neural network that only updates at certain time-steps (Bailey) - see Paragraph 4.2 for a more detailed explanation.

Algorithm 1 Deep Q-Learning Pseudocode

Initialize replay memory to capacity C
Initialize action-value function Q with random weights θ
Initialize target action-value function Q with weights $\theta^- = \theta$
for episode=1, M **do**
  Reset the environment to obtain initial state s_t
  for t=1, T **do**
    Select an action a_t ▷ under ϵ-greedy, annealing ϵ-greedy or softmax policies
    Execute action a_t and observe next state s_{t+1}, reward r_t and done flag
    Store transition (s_t, a_t, r_t, s_{t+1}, *done*) in replay memory
    Sample a random minibatch of transitions (s_i, a_i, r_i, s_{i+1}, *done*) from replay memory
    Set $y_i = r_i + \gamma \cdot max_a \hat{Q}(s_{i+1}, a'; \theta^-)$
    Perform a gradient descent step on $[y_i - Q(s_i, a_i; \theta)]^2$ with respect to the network parameters θ
    Every n steps reset $Q = Q$
  end for
end for

In our implementation, we have performed 1000 episodes, and for each of them we have selected an action based on the initial state following either the ϵ-greedy policy, the annealing ϵ-greedy policy or the softmax policy. Then, based on the selected action, we have retrieved the reward and the next state from the environment and stored them in the experience replay buffer alongside the action we took, the current state and a flag indicating whether the current episode is terminated.
During the learning or optimization phase, the agent randomly samples transitions from the replay buffer. Then, the learning target y_i is computed by summing the current reward obtained from the replay buffer (r) and the target network's estimate of future reward given the next observation ($Q(s, a; \theta_i)$). Finally, we compute the mean squared error between the learning target and the Q-network's estimate reward, obtaining the loss function for our DQN (Bailey). All these steps can be summarised by the following equation:

$$L_i(\theta_i) = \mathbb{E}_{s,a,r,s' \sim \rho(.)} \left[(y_i - Q(s, a; \theta_i))^2 \right] \quad (1)$$

where $y_i = r + \gamma \cdot max_{a'} Q(s', a'; \theta_{i-1})$ represents the (temporal difference) target (TensorFlow), namely the target value derived from future rewards discounted by the discount factor γ.

Q-network architecture The Q-network model that we have implemented is a feed-forward neural network with three fully connected layers; the network outputs a Q-value for each action, allowing the agent to decide which action to take based on the highest predicted Q-value. During the forward pass, the input to each layer is propagated through the network, undergoing linear transformations followed by ReLU activations (PyTorch).

Policies Policies in reinforcement learning define the strategy by which an agent decides on actions based on its current state, aiming to maximize cumulative rewards over time. They map from perceived states of the environment to actions to be taken when in those states.

ϵ-greedy The *ϵ-greedy policy* is a straightforward yet effective strategy in reinforcement learning that balances the act of exploring new actions with exploiting the best-known actions. This policy primarily exploits by choosing the action with the highest estimated reward (or Q-value) but occasionally explores by selecting an action at random. The decision between exploration and exploitation is governed by a parameter ϵ, a small probability.

The policy can be formally described as follows:

$$a = \begin{cases} \text{a random action} & \text{with probability } \epsilon \\ \arg\max_a Q(a, s) & \text{with probability } 1 - \epsilon \end{cases} \quad (2)$$

where:

- a is the action selected by the policy in state s,
- $Q(a, s)$ represents the Q-value for taking action a in state s, which estimates the expected reward,
- ϵ is a small positive value (typically between 0 and 1) that controls the rate of exploration. A higher ϵ increases the probability of taking a random action, thereby encouraging exploration, whereas a lower ϵ favors exploitation of the best-known action based on the Q-values.

The ϵ-greedy policy provides a simple mechanism to ensure that the agent does not become overly focused on the actions known to yield high rewards and continues to explore other potentially better actions. The choice of ϵ is crucial: too high an ϵ may lead to excessive exploration at the cost of exploiting known good actions, while too low an ϵ may prevent the agent from discovering better strategies. Often, ϵ is decayed over time, starting from a higher value to encourage initial exploration and gradually reduced to favor exploitation as the agent learns more about the environment.

Softmax The *softmax policy* is a mechanism in reinforcement learning that balances between exploring new actions and exploiting the known efficacies of actions within a given state. It operates by converting the Q-values, which estimate the expected rewards of taking certain actions from a specific state, into a probability distribution over all possible actions. The conversion utilizes the softmax function, described as follows:

Given a set of Q-values $\{Q(a, s)\}$ for all actions a in a state s, the probability of selecting action a, denoted as $P(a|s)$, is computed using the softmax function:

$$P(a|s) = \frac{\exp\left(\frac{Q(a,s)}{\tau}\right)}{\sum_b \exp\left(\frac{Q(b,s)}{\tau}\right)} \tag{3}$$

where:

- $P(a|s)$ is the probability of selecting action a when in state s,
- $Q(a, s)$ represents the Q-value for taking action a in state s, indicating the expected reward,
- τ is the temperature parameter controlling the level of exploration. A high τ value results in a more uniform distribution of action probabilities, promoting exploration. Conversely, a low τ value makes the distribution more peaked on the actions with the highest Q-values, favoring exploitation.
- The denominator $\sum_b \exp\left(\frac{Q(b,s)}{\tau}\right)$ serves as a normalization factor ensuring that the probabilities sum up to 1.

The temperature parameter τ plays a critical role in the softmax policy, determining the trade-off between exploration and exploitation. By adjusting τ, the policy can be made more explorative or more exploitative, depending on the requirements of the learning task or the stage of training.

Annealing ϵ-greedy The *annealing ϵ-greedy policy* enhances the basic ϵ-greedy policy by dynamically adjusting the exploration rate ϵ over time, allowing the agent to smoothly transition from exploration to exploitation. This is particularly useful in reinforcement learning, where the balance between exploring new actions and exploiting known actions is crucial for efficient learning. The annealing approach starts with a higher probability of exploration, which is gradually reduced according to a predefined schedule.

The policy operates as follows:

$$a = \begin{cases} \text{a random action} & \text{with probability } \epsilon(t) \\ \arg\max_a Q(a, s) & \text{with probability } 1 - \epsilon(t) \end{cases} \tag{4}$$

where:

- a is the action selected by the policy in state s,
- $Q(a, s)$ denotes the Q-value for taking action a in state s, an estimate of the expected reward,
- $\epsilon(t)$ is the exploration rate at time t, which decreases over time according to a predefined annealing schedule, such as linear or exponential decay.

The annealing schedule for $\epsilon(t)$ is crucial and can take various forms, including linear decay, where ϵ is reduced by a constant amount each step, or exponential decay, where ϵ decreases by a percentage of its current value. The choice of schedule should be tailored to the specific requirements of the learning task and the environment.

The advantage of the annealing ϵ-greedy policy is that it allows the agent to explore widely initially when its knowledge about the environment is limited, thus avoiding premature convergence to suboptimal policies. As learning progresses and the agent becomes more confident in its action-value estimates, the policy gradually shifts towards exploitation, maximizing the use of the agent's accumulated knowledge to achieve better performance.

Implementing an annealing strategy for ϵ requires careful consideration of the annealing schedule and the initial and

final values of ϵ, ensuring that the agent maintains a balance between exploration and exploitation throughout the learning process. (Sutton & Barto, 2018)

4. Experiments

4.1. Hyperparameter Tuning

The performance of networks can significantly vary depending on the hyperparameters used and therefore tuning them is a crucial factor. We consider the following parameters to be of paramount importance: network architecture (*number of hidden layers* and *number of neurons for the hidden layers*), *learning rate*, *exploration factor*, *policy*, *batch size* and *discount factor*.

- **Network architecture:** The architecture of the neural network plays a central role in the agent's ability to learn and generalize from the environment. Therefore, we have experimented with different numbers of hidden layers and different numbers of neurons per hidden layer. Indeed the number of layers in a neural network significantly affects the efficiency of the model: a deeper network can capture more complex representations of the input state space, improving performance. However, excessive layers may lead to overfitting. For our specific task, we have found that 4 layers lead to the best performance. Similar to the number of layers, the number of neurons is critical for avoiding overfitting while ensuring sufficient representation of inputs. In our case, the optimal number of neurons is 256.

- **Learning rate:** The learning rate controls the magnitude of parameter updates during training and directly impacts the convergence of the neural network. A higher learning rate allows for faster convergence but can cause instability in training. On the other hand, a lower learning rate can lead to more stable learning dynamics but reach convergence more slowly. The optimal learning rate value that we have found for our task is 0.0001.

- **Exploration factor ϵ:** This parameter manages the exploration-exploitation trade-off, where exploration favours discovering new strategies, whereas exploitation indicates leveraging known strategies. Balancing exploration and exploitation is crucial for effective learning, as too much exploration may impede to reach convergence, while too much exploitation may result in suboptimal policies. Within the Cartpole environment we observed that the optimal value for the exploration factor is 0.1.

- **Policy:** The policy plays a critical role in the performance of the algorithm, as it determines how the agent explores its environment to discover optimal actions. We have experimented with three different exploration strategies: ϵ-greedy, annealing ϵ-greedy and softmax (see section 4.3 for more details). We identified annealing ϵ-greedy as the best policy due to its effectiveness in balancing exploration and exploitation.

- **Batch Size:** The batch size determines the number of samples used in each training iteration of the neural network, thus it affects the stability and speed of training by influencing the variance of weight updates. A larger batch size can provide more stable updates to the network's weights, leading to smoother convergence during training, but on the other hand it requires more memory and computational resources, which can slow down training. Conversely, smaller batch sizes may introduce more variance into weight updates, leading to less stable training behavior. Our DQN shows better performance with a batch size of 128.

- **Discount factor:** The discount factor γ, influences the importance of future rewards relative to immediate rewards. A higher γ prioritises long-term rewards, encouraging the agent to make decisions that maximise cumulative rewards over time. Instead, a lower discount factor value prioritises more short-term rewards. The optimal value is found to be 0.95 for the discount factor.

4.2. Ablation Study

To fully understand what each component contributes to the models overall performance, we can perform an ablation study. This is a commonly used methodology which involves systematically removing components to compare how the model performs with and without them. For this experiment the components we looked at were experience replay and target network.

Experience Replay This is essentially the act of sampling a small batch of tuples from the replay buffer in order to learn. It can be extremely beneficial within RL, particularly for DQN as it addresses two primary challenges:

- **Correlation between consecutive samples:** Consecutive samples are generally highly correlated, resulting in inefficient and unstable learning. However, by using experience replay this can be mitigated by breaking the correlations among consecutive training samples.

- **Reusing experience for multiple updates:** Experience replay allows for the efficient reuse of past experiences, meaning that at each step of interaction with the environment the data that is generated can be used for learning more than once. (Torres, 2020)

This is implemented using a deque (memory) with a fixed size, which acts as the replay buffer. This buffer stores tuples of experience in the form of (state, action, reward, next state, done). During training, once an action is taken and the outcome and reward are observed, the experience is stored in the replay buffer. Then, before optimization, a batch of experiences is randomly sampled from the replay buffer in order to break correlations between consecutive samples. These experiences are used to calculate loss as well as update the network weights.

Target Network A target network is also commonly used in DQN models in order to stabilize training. This is because it provides stable target Q-values for the learning updates. During training, the agent attempts to minimize the difference between the predicted Q-values and these target Q-values, computed using the reward from the environment and estimated future rewards. In a DQN without a target network, the Q-values used as targets are generated by the same network currently being updated. This is problematic as every update to the network changes the Q-values, making them non-stationary. This can lead to instability in training, as the network can end up going in circles.

In our implementation, the target network and primary Q-network are initialised by creating a copy of the Q-network's architecture and weights. The target network will have the same weights as the primary Q-network and is not updated during the forward passes. Then, during training, the target network's Q-values are used to compute the target for the Q-learning updates. The target network's weights are also updated with the weights from the primary Q-network periodically. This occurs at every target network update step in order to ensure the target network evolves alongside the primary network but without the risk of chasing a moving target.

From Figure 2, it is easily observable that DQN with both features performs the best, demonstrating the most significant improvement at the earliest timestep and the highest overall rewards. This outcome is logical, considering that Experience Replay enhances the diversity of the training data, helping to break the correlation between consecutive samples, whereas the target network provides a stable set of Q-values against which the network's predictions can be compared. Together, these features prevent the value estimates from becoming non-stationary as well as ensuring a more robust generalization across the state-action space.

The DQN with only the target network (DQN-TN) demonstrates an improved final performance over the configuration without any of the two features (DQN-ER-TN), which suggests that having stable target values through the target network plays a more crucial role in the cartpole environment. The stability of the target network appears to be instrumental in allowing the network to learn effectively and achieve higher rewards.

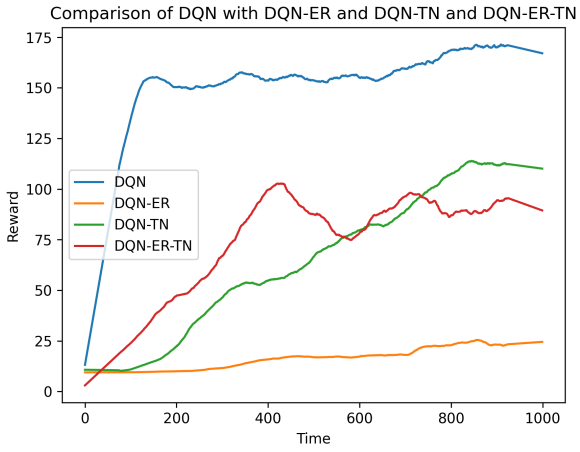

Figure 2. The plot shows the return per episode of the DQN, under four different configurations: utilizing both experience replay and a target network, employing solely a target network, employing solely experience replay, and omitting both features. The learning curves were obtained by executing 20 repetitions and the Hyperparameters were unchanged.

Interestingly, while the configuration without experience replay and target network (DQN-ER-TN) does not reach as high a final performance as DQN-TN, its learning curve suggests an earlier increase in performance. This shows a rapid initial learning phase, which then peaks early and drops.

Lastly, it is also apparent from the graph that the configuration using just experience replay (DQN-ER) is the least effective, with only a marginal increase in performance over time. In the context of the cartpole environment, this underlines the limited effectiveness of experience replay when used in isolation, without the benefits of the target network to guide the learning process with a stable set of Q-values.

4.3. Exploration Strategies

One of the preliminary questions we addressed before conducting our experiments was the selection of an appropriate exploration method for the DQN algorithm. We conducted experiments running the DQN algorithm over 1000 episodes, employing various exploration methods, namely ϵ-greedy, annealing ϵ-greedy, and softmax. For each exploration method, we utilized the optimal combination of parameters determined through extensive hyperparameter tuning. This approach ensured that each exploration method was evaluated under conditions that showcased its maximum potential performance.

In *Figure 2*, we observe that the annealing ϵ-greedy method significantly outperforms the other two methods. This outcome aligns with our expectations, considering the operational mechanics of annealing ϵ-greedy. Specifically, beginning the exploration process with a higher ϵ value is advantageous, as it promotes extensive exploration while the agent is relatively "uninformed" about the environment. Subsequently, the ϵ value gradually decreases towards zero, shifting the agent's behaviour towards exploiting its accumulated knowledge.

The ϵ-greedy method demonstrates a similar initial behaviour to annealing ϵ-greedy, yet its effectiveness diminishes over time. This indicates that maintaining a constant balance between exploration and exploitation throughout all episodes is not optimal. Furthermore, the softmax policy, at the beginning is outperformed by ϵ-greedy. The reason for this could be over the overexploitation tendecies of the softmax policy. The softmax method selects actions based on their predicted Q-values, with the exploration-exploitation balance regulated by a temperature parameter. A high temperature leads to nearly equal probabilities for all actions, fostering too much exploration at the expense of exploitation. Conversely, a low temperature results in a disproportionate preference for the action with the highest Q-value, leading to excessive exploitation. Consequently, this imbalance renders the policies suboptimal, hindering the efficiency of the learning process.

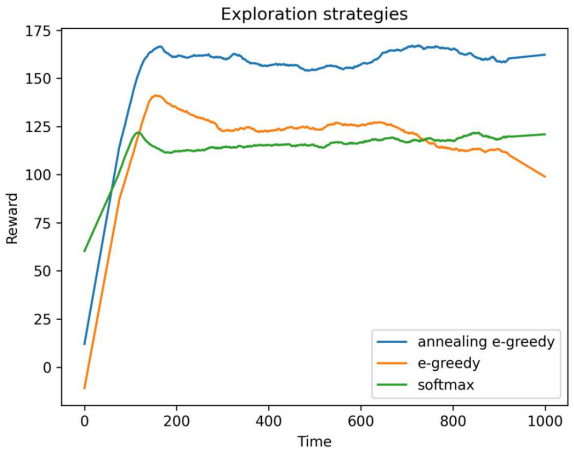

Figure 3. Graph that shows how different exploration strategies achieve higher reward over more iterations.

4.4. DQN Improvements

The power of DQN has pushed many reinforcement learning researchers to improve the algorithm even further and many refinements were achieved; this is the case for both the *double DQN* and the *dueling DQN*.

Double DQN One of the problems of the DQN is that, being an off-policy algorithm, it overestimates the true rewards. To address this issue, Double DQN was introduced: its main idea is to decouple the action selection from the action evaluation. The Bellman equation used in the DQN algorithm is substituted by the following equation:

$$Q(s,a;\theta) = r + \gamma \cdot Q(s', argmax_{a'} Q(s',a';\theta);\theta') \quad (5)$$

As the equation shows, the main neural network decides which one is the best next action a' among all the available next actions, and then the target neural network evaluates this action to know its Q-value (Ausin).

As for the architecture, it is similar to that of the DQN, but the main difference lies in how the target Q-value is computed: instead of directly using the maximum Q-value from the target network, double DQN employs the online network to select the best action and then uses the target network to evaluate that action.

Dueling DQN Dueling DQN was developed with the goal of standardizing the action values by the introduction of the so-called advantage function. In particular, in the dueling DQN algorithm, the same neural network splits its last layer in two parts, one of them to estimate the state value function for state s and the other one to estimate the advantage function for each action a, and at the end it combines both parts into a single output, which will estimate the Q-values (Ausin). At first, one might be tempted to translate this into the following formula:

$$Q(s,a;\theta,\alpha,\beta) = V(s;\theta,\beta) + A(s,a;\theta,\alpha) \quad (6)$$

where θ are the parameters of the neural network, α and β are the parameters of the advantage and value streams respectively. However, equation 6 is unidentifiable, in the sense that given Q we cannot recover V and A uniquely. To address this issue, we can force the advantage function estimator to have zero advantage at the chosen action:

$$Q(s,a;\theta,\alpha,\beta) = V(s;\theta,\beta) + \left(A(s,a;\theta,\alpha) - \max_{a' \in |A|} A(s,a';\theta,\alpha) \right) \quad (7)$$

where $\max_{a' \in |A|} A(s,a';\theta,\alpha)$ represents the maximum advantage value among all possible actions a' in state s.

The stream $V(s;\theta,\beta)$ provides an estimate of the value function, while the other stream produces an estimate of the advantage function. An alternative module replaces the max operator with an average (Wang et al., 5-04-2016):

$$Q(s,a;\theta,\alpha,\beta) = V(s;\theta,\beta) + \left(A(s,a;\theta,\alpha) - \frac{1}{|A|} \sum_{a'} A(s,a';\theta,\alpha) \right) \quad (8)$$

where $\frac{1}{|A|}\sum_{a'} A(s, a'; \theta, \alpha)$ represents the average advantage value among all possible actions a' in state s.

The architecture of the dueling DQN is characterized as follows. The feature layer processes the input state to extract relevant features; it consists of a series of fully connected layers with ReLU activation functions. The value stream estimates the state value function ($V(s; \theta, \beta)$), representing the value of being in a particular state regardless of the action taken. The advantage stream estimates the advantage function ($A(s, a; \theta, \alpha)$), representing the advantage of taking each action in a given state. Then, the outputs of the value stream and the advantage stream are combined to compute the Q-values for each action (Yoon).

Convolutional Dueling DQN We have also implemented a dueling DQN with convolutional layers: in this case, instead of processing raw state inputs with fully connected layers, the network begins with a stack of convolutional layers, which extract hierarchical features from the input states, enabling the model to learn spatial relationships (Yoon). Then, the feature maps are flattened into a vector representation to be fed into the subsequent fully connected layers, and the arcchitecture proceeds in the same way as for the standard dueling DQN that we have explained before.

Models comparison After having developed the DQN agent, we have decided to compare its performance with the double DQN and dueling DQN and therefore we have additionally implemented these two algorithms. Figure 4

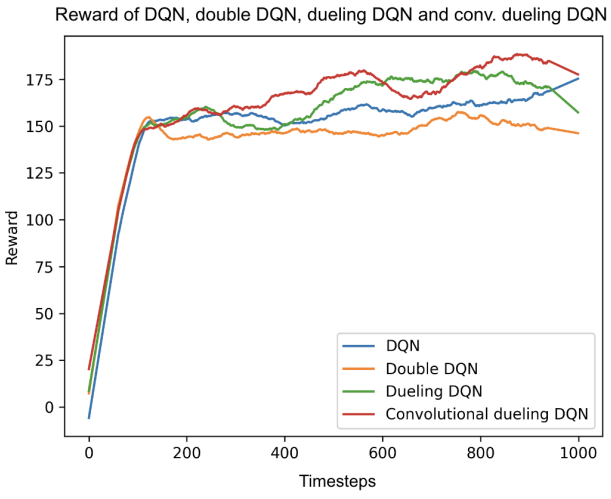

Figure 4. The plot shows the return per episode of the DQN, double DQN, dueling DQN and convolutional dueling DQN. The learning curves were obtained by executing 20 repetitions.

shows the reward per timestep curves obtained by running for 20 repetitions (each of 1000 timesteps) four different networks: DQN, double DQN, dueling DQN and dueling DQN with convolutions.

From the plot we can observe that, even though all of the models are able to achieve good performance, the convolutional dueling DQN is the one that reaches the highest reward value per episode. Indeed, the use of convolutional layers allows it to efficiently extract features from the state observations, reducing the dimensionality of the input space and enabling faster learning, as can be noticed from the plot where the convolutional dueling DQN is indeed the first model that starts to effectively learn. Moreover, convolutional layers detect spatial patterns and relationships between pixels, enabling the model to understand the spatial arrangement of objects in the environment, such as the position of the cart relative to the pole, thus leading to more effective learning.

High performance are also achieved by the dueling DQN, which was expected since it is in general more efficient than traditional Q-learning algorithms because it only needs to learn a single value for each state and a single advantage value for each action, rather than a value for every state-action pair (Jayakody). However, both dueling DQN models show many fluctuations, which can be attributed to their more complex architectures that make them even more sensitive to the choice of hyperparameters, thus potentially leading to a more unstable learning curve (Jayakody).

As for the double DQN, we can observe that, after reaching a peak at about 150 reward per episode, the learning curve seems to stabilize to a certain episode reward without improving any further: this behavior can be attributed to the addition of an extra hyperparameter to control the frequency of updating the target network (Shost). If not properly tuned, it can lead to suboptimal learning and therefore obstruct the agent's ability to estimate action values.

By turning instead to look at the plot of the standard DQN, it can be noticed that it has a slower learning compared to other models, since, due to its simpler architecture, it may take longer to learn complex representations of the environment. On the other hand, the DQN learning curve shows a more steady increase in reward, which lets us hypothesize that with more timesteps it could even reach better rewards than the other models, which are more complex and therefore also more prone to overfitting or having a more erratic behavior.

5. Conclusion

In this report, our primary goal was exploring the effectiveness of DQN in the high-dimensional and dynamic Cartpole environment. Through a series of experiments, we tuned hyperparameters, investigated different exploration strategies, as well as experimented with multiple DQN configurations, including the double and dueling DQN models.

Our findings have revealed that DQNs are indeed capable of learning and performing in complex environments. The ablation study illustrated in Figure 2, highlighted the contributions of experience replay and target networks to the overall performance of DQN models. While the combination of both led to the highest rewards, the stability provided by the target network alone proved to be a strong contributor to effective learning in this environment.

Further experimentation with exploration strategies showed the annealing ϵ-greedy method's superiority, as well as the importance of a balanced approach to exploration and exploitation over time. When comparing architectural improvements, the convolutional dueling DQN was the top performer, utilizing the benefits of feature extraction capabilities via the convolutional layers.

Overall, the experiments conducted validate the utility of DQN in a complex control task like balancing a pole on a cart and the insights gained from the ablation study as well as the performance metrics of the various DQN models provide valuable insights into designing and optimizing DQNs for similar tasks in the future.

References

Ausin, M. S. Introduction to reinforcement learning. part 4: Double dqn and dueling dqn. https://markelsanz14.medium.com/introduction-to-reinforcement-learning-part-4-double-dqn-and-dueling-dqn-b349c9a61ea1. 14-04-2020.

Bailey, J. Deep q-networks explained. https://www.lesswrong.com/posts/kyvCNgx9oAwJCuevo/deep-q-networks-explained#DQN_Technical_Explanation__Section_4_. 13-09-2022.

Dhumne, S. Deep q-network (dqn). https://medium.com/@shruti.dhumne/deep-q-network-dqn-90e1a8799871. 30-06-2023.

Jayakody, D. Dueling networks - a quick introduction (with code). https://dilithjay.com/blog/dueling-networks-a-quick-introduction-with-code#advantages-of-dueling-networks. 24-12-2022.

Mnih, V., Kavukcuoglu, K., Silver, D., Graves, A., Antonoglou, I., Wierstra, D., and Riedmiller, M. Playing atari with deep reinforcement learning. *IBM Journal of Research and Development*, 2013.

OpenAI. Gym documentation: Cart pole. https://www.gymlibrary.dev/environments/classic_control/cart_pole/. August 2022.

Plaat, A. (ed.). *Deep Reinforcement Learning*. Springer Nature, 2023.

PyTorch. Reinforcement learning (dqn) tutorial. https://pytorch.org/tutorials/intermediate/reinforcement_q_learning.html.

Shost, M. What are the benefits and drawbacks of using double dqn over vanilla dqn? https://www.linkedin.com/advice/3/what-benefits-drawbacks-using-double-dqn-over-van~:text=Additionally%2C%20training%20and%20updating%20both,preferable%20in%20reinforcement%20learning%20tasks. 2024.

Sutton, R. S. and Barto, A. G. *Reinforcement learning: An introduction*. 2018.

TensorFlow. Introduction to rl and deep q networks. https://www.tensorflow.org/agents/tutorials/0_intro_rl. 2023.

Torres, J. Deep q-network (dqn)-ii experience replay and target networks. *Towards Data Science*, 2020.

Wang, Z., Schaul, T., Hessel, M., van Hasselt, H., Lanctot, M., and de Freitas, N. Dueling network architectures for deep reinforcement learning. *Google DeepMind, London, UK*, 5-04-2016.

Yoon, C. Dueling deep q networks. https://towardsdatascience.com/dueling-deep-q-networks-81ffab672751. 19-19-2019.