Solving The Pendulum Swing-up Problem With Deep Reinforcement Q Learning

Michael Cornelisse (s1059020)
Radboud University
michael.cornelisse@ru.nl

Abstract

This paper applies Deep Q-Learning (DQN) with a replay buffer and epsilon-greedy exploration to the pendulum swing-up problem. The DQN successfully learned to balance the pendulum upright, as evidenced by increasing rewards and decreasing loss during training. Despite the limitations of a discretized action space, the study demonstrates DQN's potential for control tasks and encourages further research into continuous action methods for enhanced performance.

1 Introduction

Reinforcement Learning (RL) aims to learn optimal actions in an environment by maximizing a numerical reward signal. Deep Reinforcement Learning (Deep RL) combines Reinforcement Learning with deep neural networks to learn policies in high-dimensional or continuous environments. Another RL approach is *Q-Learning*, which learns a state action value function $Q(s, a)$. The Deep Q-learning algorithm (DQN) extends classical Q-learning by using neural networks as function approximators for $Q(s, a)$ and using techniques such as replay buffers and target networks for increased stability.

In this project, I focus on the *pendulum swing-up* task, a fundamental control problem in the RL domain. The goal is to move the pendulum from its stable downward equilibrium to an upright position and keep it balanced. Although this environment typically uses continuous action space, I discretize the torque control to enable a DQN-style algorithm. I will investigate if a Deep Q-Learning approach with an experience replay buffer and ϵ-greedy exploration can solve the Pendulum-v1 swing-up problem under these discretized conditions.

1.1 Research Question

To what extent can a Deep Q-Learning approach, using a replay buffer and ϵ-greedy exploration, successfully learn the pendulum swing-up task in a discretized action setting?

2 Methods

2.1 Pendulum Environment

I use the "Pendulum-v1 environment from Gymnax [1], which simulates a pendulum with continuous dynamics. The environment represents the pendulum's state as a three-dimensional vector $(\cos(\theta), \sin(\theta), \dot{\theta})$, where $\cos(\theta)$ and $\sin(\theta)$ capture its orientation and $\dot{\theta}$ denotes its angular velocity. The environment runs in an episodic manner, were each episode lasts up to 200 steps, after which the environment resets. To accommodate Q-learning, I discretize the continuous torque $\in [-2, 2]$ into three discrete values $\{-1.0, 0.0, 1.0\}$. The goal of the DQN algorithm is to swing the pendulum to the upright position minimizing $\theta = \pi$ and keep it balanced there.

Preprint. Under review.

2.2 Q-Network Architecture

To approximate the Q-function $Q_\theta(s, a)$, I define a fully connected feed-forward neural network (MLP) consisting of two hidden layers of size 128 with ReLU activation, and an output layer of size 3, corresponding to the discrete action space. I initialize the parameters with small Gaussian noise (scale $\approx 10^{-2}$). More formally, if $\mathbf{x}$ is the pendulum state, then:

$$Q_\theta(x, a) = \text{MLP}(x; \theta) \in \mathbb{R}^{|\mathcal{A}|}$$

At each forward pass, the network receives the current observation $\mathbf{x}$ as input and outputs a vector of Q-values $Q_\theta(\mathbf{x}, a)$ for each discrete action $a \in \mathcal{A}$, where $\mathcal{A} = \{-1, 0, +1\}$.

2.3 DQN Algorithm

Our algorithm is based on the DQN framework introduced by Mnih et al. [2]. I use the Q-network described in Section 2.2 to approximate the optimal state-action value function $Q_\theta(s, a)$.

During each training iteration, the network interacts with the environment and collect experiences. Action selection follows a ϵ-greedy policy derived from the Q-network's output. Specifically, with probability ϵ, it selects a random action to encourage exploration. Otherwise, with probability $1 - \epsilon$, it selects the action that maximizes the Q-value for the current state, that is, the action that is predicted to yield the highest expected return: $a = \arg\max_{a'} Q_\theta(s, a')$.

The collected experiences are stored as transitions of the form $(s_t, a_t, r_t, s_{t+1}, d_t)$ in a replay buffer. This buffer acts as a memory, storing a history of the interactions. The use of a replay buffer helps to decorrelate consecutive samples, which is important for stabilizing the learning process. Here, s_t and s_{t+1} denote the current and subsequent states, respectively, a_t is the action taken, r_t is the reward received, and d_t is a boolean flag indicating whether s_{t+1} is a terminal state (i.e., $d_t = 1$ if the episode ended at s_{t+1}, and $d_t = 0$ otherwise).

To update the Q-network, it samples mini-batches of transitions uniformly at random from the replay buffer. For each transition $(s_i, a_i, r_i, s_{i+1}, d_i)$ in a mini-batch, the one-step temporal difference (TD) is computed, denoted by y_i:

$$y_i = r_i + \gamma\,(1 - d_i)\,\max_{a'} Q_{\theta^-}(s_{i+1}, a')\,.$$

Here, $\gamma \in [0, 1]$ represents the discount factor, which determines the importance of future rewards, and d_i indicates whether the episode terminated. The term θ^- represents the parameters of a separate target network, which is periodically updated with the parameters of the online network θ to improve training stability.

The loss function, $\mathcal{L}(\theta)$, computes the discrepancy between the Q-network's predicted Q-values and the corresponding TD targets over the entire mini-batch, using the Mean Squared Error (MSE):

$$\mathcal{L}(\theta) = \frac{1}{B}\sum_{i=1}^{B}\Big(Q_\theta(s_i, a_i) - y_i\Big)^2,$$

where B is the batch size, we then update the online network parameters θ by performing gradient descent on the loss function using back-propagation.

Instead of directly copying the online network parameters θ to the target network parameters θ^- periodically, a soft-update is performed, also known as Polyak averaging, after each gradient step. This soft update is performed as follows:

$$\theta^- \leftarrow \tau\,\theta + (1 - \tau)\,\theta^-,$$

where $\tau \in (0, 1)$ is a small value (e.g., $\tau = 0.001$) that controls the update rate. This gradual update of the target network helps mitigate the instability that can arise from chasing a rapidly moving target during training.

2.4 Implementation Details

The experimental pipeline was implemented in Python within a Jupyter notebook environment. JAX [3] was used to run the computations on the CPU and Optax [4] for automatic differentiation and optimization.

2.5 Model Training

The training procedure for the Deep Q-Learning algorithm is outlined in Algorithm 1. This algorithm initializes the Q-network and target network parameters, manages the replay buffer, and iteratively updates the network based on interactions with the environment.

Algorithm 1: Training Loop

Input : Q-network parameters θ, target network parameters $\theta^- \leftarrow \theta$, replay buffer capacity N
Output : Trained Q-network
Initialize Q-network parameters θ and target network parameters $\theta^- \leftarrow \theta$;
Initialize Optimizer;
Initialize replay buffer with capacity N;
for *each episode* **do**
  Reset the environment to obtain initial state s_0;
  for *each step in the episode* **do**
    Select action a_t via ϵ-greedy over $Q_\theta(s_t, a)$;
    Execute a_t in the environment, observe reward r_t and next state s_{t+1};
    Store $(s_t, a_t, r_t, s_{t+1}, d_t)$ in the replay buffer;
    if *replay buffer has enough samples* ***and*** *warm-up period is over* **then**
      Sample a mini-batch of transitions;
      Compute TD targets and the loss;
      Perform a gradient update on θ;
      Perform a soft update on θ^-;
    end
    $s_t \leftarrow s_{t+1}$;
    Terminate if done;
  end
end

In our experiments this procedure was repeated for 600 episodes (120,000 steps). The model was trained using the Adam optimizer, and we used the following hyper-parameters: a learning rate α of 5×10^4, and a discount factor γ of 0.99. The replay buffer size was limited to 10,000 transitions, and mini-batches of 128 experiences were sampled during training. To encourage exploration, an ϵ-greedy policy is employed, with initial ϵ set to 0.9 and linearly decayed to 0.05 over the first 1,000 steps, after a warm-up phase consisting of 1,000 steps where actions were selected randomly. To improve the stability of the learning process, the target network was softly updated with a rate, τ, of 0.005.

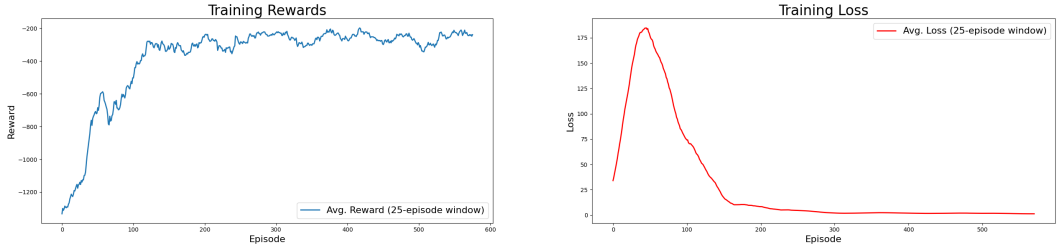
3 Results and Discussion

The model was trained multiple times using varying numbers of episodes, different layer sizes (32, 64, 128, and 256), multiple learning rates (1e-4, 3e-4, and 5e-4), various epsilon decay steps (1,000; 5,000; 10,000; and 25,000), and different batch sizes (32, 64, and 128). I found that 600 episodes yielded the best results across most configurations, whereas the model failed to converge when trained with only 500 episodes. The optimal performance was achieved with a layer size of 128, a learning rate of 5e-4, a batch size of 128, and 1,000 epsilon decay steps.

Figure 1a presents the training episode returns over 600 episodes. I smoothed the rewards with a 25-episode moving average. The average reward starts at a low value of around -1200 but the reward curve exhibits an initial steep increase, followed by a more gradual improvement, and eventually plateauing around episode 300. This indicated that the network was able to learn a reasonably good

policy for swinging up and balancing the pendulum. However, the presence of some fluctuations in the later stages of the training process suggest that the learned policy might not be perfectly stable or optimal.

Figure 1b presents the average training loss over the total episodes. Initially, the loss values are high, but a rapid decrease is observed, followed by stabilization near zero after roughly 200 episodes. The initial high loss are likely because, the networks initial predictions are initially far off from the TD targets. As the network leans and the Q-values more accurate, the loss decreases. The rapid decline in loss coincides with the rapid increase in rewards observed in Figure 1a, suggesting a correlation between loss reduction and performance improvement.



(a) Training rewards averaged over 25 episodes.

(b) Training losses averaged over 25 episodes.

Figure 1: Training performance of the DQN model, showing average reward and loss over 25-episode windows.

Figure 2 presents the final position of the pendulum after the network has been successfully trained. The pendulum is depicted in a fully upright position and balanced state, with the tip at coordinates (0, 1.0) and the pivot at (0, 0.0). This depiction confirms that the agent has effectively learned to swing the pendulum upward and maintain its stability in the inverted position.

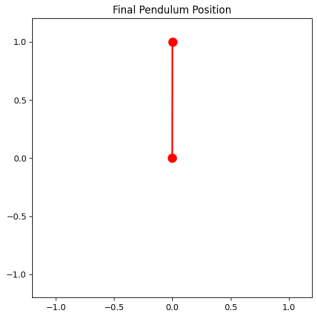


Figure 2: Pendulum in upright position after training.

It is worth mentioning that the choice of the network architecture and hyper parameters, likely had a important role in the networks learning process. While we have used standard values based on the literature [5], further tuning of these parameters could potentially lead to even better performance or faster convergence.

One limitation of our approach is the discretization of the action space. While it enables the use of a DQN-style algorithm, it might limit the precision with which the agent can control the pendulum. A continuous action space, coupled with an actor-critic architecture as proposed by Bi et al [6], could potentially achieve better performance.

4 Conclusion

In this project I implemented DQN algorithm based on the framework outlined by Mnih et al. [2]. The results show that a Deep Q algorithm with a replay buffer and epsilon-greedy exploration can reasonably the pendulum swing-up task with a discretized action space. However, the discrete action space limits control precision, suggesting that future research should explore continuous action methods to enhance performance.

References

[1] Robert Tjarko Lange. gymnax: A JAX-based reinforcement learning environment library, 2022.

[2] Volodymyr Mnih, Koray Kavukcuoglu, David Silver, Alex Graves, Ioannis Antonoglou, Daan Wierstra, and Martin Riedmiller. Playing atari with deep reinforcement learning, 2013.

[3] James Bradbury, Roy Frostig, Peter Hawkins, Matthew James Johnson, Chris Leary, Dougal Maclaurin, George Necula, Adam Paszke, Jake VanderPlas, Skye Wanderman-Milne, and Qiao Zhang. JAX: composable transformations of Python+NumPy programs, 2018.

[4] DeepMind, Igor Babuschkin, Kate Baumli, Alison Bell, Surya Bhupatiraju, Jake Bruce, Peter Buchlovsky, David Budden, Trevor Cai, Aidan Clark, Ivo Danihelka, Antoine Dedieu, Claudio Fantacci, Jonathan Godwin, Chris Jones, Ross Hemsley, Tom Hennigan, Matteo Hessel, Shaobo Hou, Steven Kapturowski, Thomas Keck, Iurii Kemaev, Michael King, Markus Kunesch, Lena Martens, Hamza Merzic, Vladimir Mikulik, Tamara Norman, George Papamakarios, John Quan, Roman Ring, Francisco Ruiz, Alvaro Sanchez, Laurent Sartran, Rosalia Schneider, Eren Sezener, Stephen Spencer, Srivatsan Srinivasan, Miloš Stanojević, Wojciech Stokowiec, Luyu Wang, Guangyao Zhou, and Fabio Viola. The DeepMind JAX Ecosystem, 2020.

[5] Reinforcement Learning (DQN) Tutorial &x2014; PyTorch Tutorials 2.5.0+cu124 documentation — pytorch.org. `https://pytorch.org/tutorials/intermediate/reinforcement_q_learning.html`.

[6] Yifei Bi, Xinyi Chen, and Caihui Xiao. A deep reinforcement learning approach towards pendulum swing-up problem based on tf-agents, 2021.