Exercises for "Deep Reinforcement Learning"

Winter Term 2024 Sheet 5

Deep Reinforcement Learning and Policy Gradient

Issued: 09.12.2024, **Due:** 12.01.2025, **Discussion:** 15.01.2025

In this exercise sheet, you will learn about function approximation and how it enables reinforcement learning to scale to environments with high-dimensional or continuous state and action spaces. By using deep neural networks as function approximators, reinforcement learning can provide solutions to tasks where traditional tabular methods fail due to the curse of dimensionality or the need for generalization across states and actions.

Task 5.1 Function Approximation with Value Functions on Feature-Level: 4 PPoints = 2 + 2

In previous exercises you have used value tables to map between states and their respective state- or action-value. For this task you will represent this value function by an artificial neural network to approximate the action-value function $Q(s, a)$ to solve the classic control problem given by the Cart Pole Environment (see Figure 1).

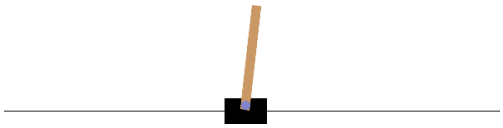

Figure 1: The CartPole Environment from Gymnasium (`gymnasium.make("CartPole-v1")`) presents a control problem that involves a continuous state space with a discrete action space.

(a) **Deep Q-learning** (off-policy TD control) for estimating $\pi \approx \pi_*$

Your Q-Learning algorithm form the previous exercise sheet utilizes a Q-table to directly map each state-action pair to a value. In the deep Q-learning algorithm this table is replaced with a neural network, $Q(s, a; \theta)$, where θ represents the network's parameters. The network takes a state as input and outputs Q-values for all possible actions.

- *Algorithm*: Modify your existing Q-learning algorithm to a deep Q-learning algorithm by adjusting the training and update process. Instead of using the Q-table for lookups, you need to feed the state into the neural network to generate value estimates and generate actions. In vanilla Q-Learning, the agent updates the Q-table entry for a state-action pair via:

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

With Deep Q-Learning, this update translates into minimizing the loss function using supervised learning techniques. The target value $y = r + \gamma \max_{a'} Q(s', a'; \theta)$ is used for network optimization:

$$L(\theta) = (y - Q(s, a; \theta))^2$$

To improve the network loss on this objective function, the parameters θ are adjusted via backpropagation and gradient descent. (Check out the **Guidance** for more help on parameter choices for implementation)

- *Visualization*: Since deep reinforcement learning employs function approximators that aim to reduce the error of a predefined loss function, you now need to monitor 2 learning processes. The first one is the convergence of the reinforcement learning algorithm towards the optimal value function and policy, and the other is the reduction of the function approximators loss. Accordingly, you have to visualize the agents training progress by plotting cumulative rewards (return G) per episode and monitor the loss to ensure the network is learning.
- *Reflection*: Function approximation and large state spaces make visualization of the resulting value functions unintuitive. Propose an idea to visualize the value function effectively and provide a brief rationale for your chosen approach.

(b) **Extensions to the Q-learning Algorithm**

- *Algorithm* - Extend your existing implementation with one of the following techniques, presented in the lectures:
  - *Replay Buffer*: Save $(S, A, R, S', done)$ samples for delayed and randomised training. Store the agent's experiences at each time step in a replay buffer. Sample a random mini-batch from the buffer to break the correlation between consecutive samples.
  - *Target Q-Learning*: Use a target network to compute stable Q-value targets during training. Every N steps, clone your main network to create a target network. Use this target network to compute Q-value targets, which helps in reducing the oscillations during training.
- *Visualization*: Similar to the previous task, you have to visualise the cumulative rewards and the loss. For a detailed comparison, the results of the previous task which uses a vanilla Q-learning approach and the current one have to be plotted alongside. Additionally, compare the final policies with your proposed idea from the previous task.
- *Reflection*: How do modifications introduced to the Q-Learning algorithm, such as the use of neural networks, replay buffers, and target networks, contribute to improved learning outcomes? Why can these enhancements lead to potentially more accurate approximations of action values $Q(s, a)$, better handling of exploration-exploitation trade-offs, and greater stability and scalability compared to the traditional Q-table approach? Can you perceive these changes in your vanilla and advanced Q-learning implementations?

Guidance - Start by setting the parameters according to the parameters used in the original publication [2]:

- *ANN Structure* - Multi Layer Perceptron (MLP) with two hidden layers $(64, 32)$
- *ϵ-greedy strategy* - during first million frames scale ϵ from 1.0 to 0.1
- *Mini-Batch Size* - 32 samples
- *Replay Buffer Size* - 1 million frames
- *Training Time* - approximately 50 million frames (around 38 days of continuous gaming)

Task 5.2 Function Approximation with Value Functions on Pixel-Level: 3 Points = 2 + 1

Atari games like Breakout (https://ale.farama.org/environments/breakout/) present a challenge to reinforcement learning algorithms that we have not addressed thus far. The state space consists of raw pixel data, a high-dimensional input representation. A common technique from recent years is to use Convolutional Neuronal Networks (CNNs) to encode the complex high-dimensional state space representation into a lower dimensional latent space, that retains the most essential informations from the original data. Based on this lower dimensional representation a Multi-Layered Perceptron can discriminate between states and learn Q-value estimates.

Before you proceed to the task, ensure that you have installed the 'Arcade Learning Environments' (ALE; https://ale.farama.org/getting-started/) provided by the developers of Gymnasium [1]. If you followed the instructions, you are able to instantiate the Breakout environment by importing the library and calling `gymnasium.make("ALE/Breakout-v5")` (see Figure 2).

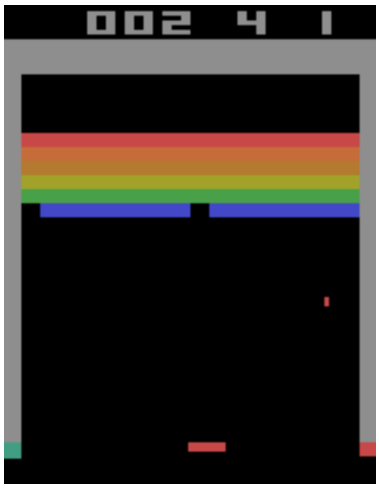

Figure 2: Illustration of the classic Arcade game 'Breakout', where players control a paddle to bounce a ball and break bricks. The goal of this game is to break all bricks without loosing the ball.

The state space in the Breakout environment is represented by an RGB image of the game screen. This is defined as a Box space with dimensions (210, 160, 3), indicating the height, width, and colour channels (RGB) of the image. Each of the three pixel values in the grid is an integer (`uint8`) ranging from 0 to 255, representing the respective colour intensity. The action space is a Discrete space with 4 possible actions, each represented by an integer: $0 \rightarrow$ 'NOOP' (No operation, maintain the current state), $1 \rightarrow$ 'FIRE' (Launch the ball if it is not already in play), $2 \rightarrow$ 'RIGHT' (Move the paddle to the right) and $3 \rightarrow$ 'LEFT' (Move the paddle to the left).

(a) **Introduction to CNNs for Reinforcement Learning**

Feature engineering is an integral part of reinforcement learning, as it enables learning in high-dimensional spaces. By transforming raw observations, such as pixel data, into structured or compressed representations, it allows agents to focus on relevant information for decision-making.

- *Algorithm*: Implement a Deep Q-Learning Algorithm using a CNN to process the high-dimensional pixel data from the Atari Breakout Game. Learn the optimal value function $Q_*(s, a)$ to derive the policy π_*.

  To prepare the RAW images for encoding by the CNN, you have to create a preprocessing pipeline that includes the steps outlined in the **Guidance** section. For feature extraction and value estimation, build a CNN-based model following the specified architecture outlined in the **Guidance** section as well. The equations from
- *Visualization*: Plot the cumulative rewards over episodes during training to assess the agent's learning progress.

- *Reflection*: Feature engineering with CNNs transformed the high-dimensional raw pixel data into a lower dimensional representation. How did this enhance the agent's ability to learn the optimal policies in the Breakout environment?

Guidance - For your orientation you can adhere to the preprocessing steps and parameters specified by Mnih et. al. [2].

CNN Architecture:

The CNN architecture consists of a feature extractor that distills input data into essential features. Following this, the Q-value head computes Q-values, approximating the expected future rewards for possible actions.

1. Feature Extractor
    - Convolutional Layer, 32 filters, 8×8, stride 4, ReLU
    - Convolutional Layer, 64 filters, 4×4, stride 2, ReLU
    - Convolutional Layer, 64 filters, 3×3, stride 1, ReLU
2. Q-Value Head
    - Fully-Connected Layer, 512 Units, ReLU
    - Output: Fully-Connected Layer, n_actions Units

Preprocessing:

For processing the observations from the environment, you can use the dedicated wrapper for Atari games (Link). To stack the most recent observations, have a look at the frame stack wrapper (Link)

- *Frameskip* - instead of using every frame, 4 frames are skipped
- *Observation Noise/Variance* - use max value from consecutive frames ($max(t, t_{-1})$) to remove artifacts (e.g. flickering)
- *Feature Engineering* - extract luminance from each RGB frame (e.g. conversion to Grayscale or the LAB colour space and extracting the luminance channel) and rescale resulting frame to shape 84×84
- *State Representation* - stack 4 of the resulting frames as input to the network to capture environment dynamics

Task 5.3 Function Approximation with Policies - Policy Gradient Algorithm: 4 Points = 2 + 2

Policy gradient methods are a powerful class of algorithms in reinforcement learning, designed to optimize an agents policy directly. In this exercise, you will apply these methods to the CartPole environment (`gymnasium.make("CartPole-v1")`) from Gymnasium. Starting with the REINFORCE algorithm, a Monte Carlo-based policy gradient method, you will learn how to estimate gradients and improve the policy iteratively. You will then extend this implementation by introducing a baseline - a practical technique to reduce variance and improve the stability of gradient estimates. For a visual explanation of these ideas, we recommend watching this video: https://www.youtube.com/watch?v=5eSh5F8gjWU.

Policy gradient methods aim to update the probability distribution of actions in a given state towards those actions with higher expected rewards. For the following a finite discrete action space and a stochastic policy based on the Softmax function is assumed:

$$\sigma(\mathbf{z})_i = \frac{e^{z_i}}{\sum_{j=1}^{K} e^{z_j}}$$

(a) **REINFORCE** (Monte-Carlo) Policy-Gradient Control (episodic) for π_*

The REINFORCE algorithm is a policy gradient method that directly estimates returns from complete episodes and uses these estimates to update the policy parameter θ. For gradient based optimisation the fundamental idea of Monte-Carlo Methods can be applied. The expectation of the sample gradient is equal to the actual gradient:

$$\begin{aligned}\nabla J(\theta) &= \mathbb{E}_\pi[q^\pi(S_t, A_t)\nabla \ln \pi(A_t|S_t, \theta_t)] \quad | \; q^\pi(S_t, A_t) = \mathbb{E}_\pi[G_t|S_t, A_t] \\ &= \mathbb{E}_\pi[G_t \nabla \ln \pi(A_t|S_t, \theta_t)]\end{aligned}$$

- *Algorithm*: Implement the REINFORCE algorithm. Initialise a policy network π^θ with random parameters θ to map states to probabilities of actions. Update the policy parameters (θ) at each time step via gradient ascent using:

$$\theta_{t+1} \leftarrow \theta_t + \alpha G_t \nabla \ln \pi(A_t|S_t, \theta)$$

- *Visualization*: Visualize the training progress by plotting cumulative rewards per episode and monitor the loss of the policy network to ensure the network is learning. Visualize your policy with your in task 5.1 proposed idea.
- *Reflection*: Value-based methods rely on ϵ-greedy or soft policies to ensure exploration. How does the REINFORCE algorithm facilitate exploration in the learning process? Explain the method used in this task and possible alternatives.

(b) **REINFORCE with Baseline** (episodic), to estimate $\pi \approx \pi_*$

In many reinforcement learning tasks, variance in gradient estimates can be high, leading to unstable learning. This is especially the case for Monte Carlo Methods. Using a baseline leaves the expected value of the update unchanged (low bias), but it can have a large effect on reducing its variance. Numerous possibilities exist (see [3]), but the common practice is to subtract a value function function estimate $V(S_t; \phi)$ from the total return to calculate the advantage $A_t = G_t - V(S_t; \phi)$. With this addition the gradient calculation changes to:

$$\nabla J(\theta) = \mathbb{E}_\pi[A_t \nabla \ln \pi(A_t|S_t, \theta_t)]$$

- *Algorithm*: Modify the REINFORCE implementation to include a baseline. Incorporate a state-value function $V(S_t, \phi)$ to calculate the baseline as a separate neuronal network (MLP with 2 hidden layers). Adjust the return computation to the advantage function $A(S_t, A_t) = G_t - V(S_t; \phi)$.

For the policy gradient update use the advantage function instead of the return to update the policy parameters:

$$\theta_{t+1} \leftarrow \theta_t + \alpha A(S_t, A_t) \nabla \ln \pi(A_t|S_t, \theta)$$

Train $V(S_t, \phi)$ to approximate G_t using the mean-squared error loss:

$$L(\phi) = \frac{1}{2}\mathbb{E}[(G_t - V(S_t; \phi))^2)]$$

Compute the gradient of the loss function for the value function:

$$\nabla L(\phi) = -\mathbb{E}[(G_t - V(S_t; \phi))\nabla V(S, \phi)]$$

To update the parameters of the value-function use gradient descent, with β as learning rate for the value-function update step:

$$\phi_{t+1} \leftarrow \phi_t + \beta(G_t - V(S_t; \phi))\nabla V(S_t; \phi)$$

- *Visualization*: Similar to the previous task, you have to visualise the cumulative rewards and the loss. For a detailed comparison of both REINFORCE algorithms, the results of the vanilla REINFORCE algorithm and the current one have to be plotted alongside.
- *Reflection*: How do changes in the state value $V(S_t)$ influence the resulting Advantage values? Consider cases with high and low action values. How would different baseline values influence the Advantage calculation and eventually the gradient calculation?

References

[1] M. G. Bellemare, Y. Naddaf, J. Veness, and M. Bowling. The arcade learning environment: An evaluation platform for general agents. *Journal of Artificial Intelligence Research*, 47:253–279, jun 2013.

[2] Volodymyr Mnih, Koray Kavukcuoglu, David Silver, Andrei A. Rusu, Joel Veness, Marc G. Bellemare, Alex Graves, Martin Riedmiller, Andreas K. Fidjeland, Georg Ostrovski, Stig Petersen, Charles Beattie, Amir Sadik, Ioannis Antonoglou, Helen King, Dharshan Kumaran, Daan Wierstra, Shane Legg, and Demis Hassabis. Human-level control through deep reinforcement learning. *Nature*, 518(7540):529–533, 02 2015.

[3] John Schulman, Philipp Moritz, Sergey Levine, Michael Jordan, and Pieter Abbeel. High-Dimensional Continuous Control Using Generalized Advantage Estimation.