Assignment 2

Due Date: November 25, 2024

Exercises

In this set of exercises, you will implement the DQN algorithm and several of its extensions. Your task is to implement the algorithms, evaluate them on benchmark environments, and answer questions about their performance and properties. **You may work alone or in groups of two.** If you work in a pair, include both names and UČOs in your report.

To pass the homework, you need to get **at least 70 points** out of 100 points.

Implementation (max. 40 points)

Complete the classes `DQNNet` and `DQNTrainer` in the `hw2.py` file. Submit the modified source file to the file vault (odevzdávárna) called *code_hw2* in the IS.

Gym Environments In contrast to the previous assignment, no environment wrapper will be used this time. Instead, you will interact directly with the environments from the OpenAI Gym library. An example interaction is shown in the `train` function of `DQNTrainer`. For further details, see the documentation. You will evaluate your algorithms on the following environments: CartPole-v1, Acrobot-v1, and LunarLander-v2.

Logging As in the previous homework, you can use the `Logger` class to log data during training to a CSV file.

Interface You will complete the trainer class for DQN, which is required to have the constructor and the `train` method. The `train` method should train a new `DQNPolicy` from scratch in a provided environment and return it.

The signatures (**positional** arguments, **return type**) of the prepared `train` method and the constructor should remain unchanged for evaluation purposes, however, you are free to add your own methods, or your own training (hyper) parameters via keyword arguments. Once again, make sure you provide default values for these parameters since we will only supply the positional parameters during the evaluation.

We have also provided several helper functions that should aid you in starting the assignment. These illustrate some core Pytorch functionality - such as how to train your model and some useful tensor operations. However, feel free to delete these methods, or modify them as you like, since they are not part of the required interface.

You can also check out this PyTorch tutorial, which gives some more examples.

DQN and Its Extensions

[25 points] Complete the `train` method of trainer class `DQNTrainer` to implement the DQN algorithm. It should support three modes `'dqn'`, `'dqn+target'`, and `'ddqn'` specified by the argument `mode`. See the Algorithms Overview section for a brief description of the differences. Use an ε-greedy policy for exploration.

[**5 points**] Modify the `train` method of the `DQNTrainer` class to support linear and/or exponential decay of the exploration rate ε over time. The initial value of ε is specified by the argument `initial_eps`. The final value of ε is specified by the argument `final_eps`. You can experiment with various decay schedules.

[**10 points**] Modify the `train` method of the `DQNTrainer` class to support n-step bootstrapping. The number of steps is specified by the argument `n_steps`.

Evaluation (max. 60 points)

Submit your solutions to the following exercises in a single PDF file to the file vault (odevzdávárna) called *report_hw2* in the IS. Evaluate your implementation on environments `CartPole-v1`, `Acrobot-v1`, and `LunarLander-v2`.

- Run the algorithm for 50000 steps on the `CartPole` and `Acrobot` environments and for 100000 steps on `Lunar Lander`. Experiment with different hyperparameters. Use the discount factor $\gamma = 0.99$, but the other hyperparameters are up to you.[1] We have provided reasonable default values for the other parameters in the source code to get you started easily. However, please keep the batch size ≤ 128, so that we are able to evaluate your implementation.

  With a reasonable amount of hyperparameter tuning, you should, on average, achieve approximately the following mean undiscounted returns in each environment: 400 on `CartPole`, -100 on `Acrobot`, and 200 on `Lunar Lander`.
- Repeat the training process multiple times and aggregate the results.
- [**7 points**] Plot how the mean discounted and undiscounted return of the trained policy and value estimate $\max_a Q_\theta(s_0, a)$ of the initial state change over time. Use whatever plot you think is the most informative. However, the plot should ideally display more than merely average values [**+4 points**].
- [**10 points**] Describe the obtained figures.

Answer the following questions in your report. Make conjectures based on your experimental results and your understanding of the algorithms. Keep your responses brief and concise.[2]

- [**15 points**] Compare the results for individual DQN modes. Explain the differences in performance.
- [**12 points**] What is the impact of the decay of the exploration rate on the performance of the DQN algorithm?
- [**12 points**] What is the impact of the n-step bootstrapping on the performance of the DQN algorithm? Experiment with various values of n.

[1] You do not have to train all combinations of the extensions. It is up to you to select a reasonable subset.

[2] The assignment is intentionally designed to be open-ended, encouraging you to experiment with the algorithms and consider their properties. If you would feel more comfortable with additional structural guidelines, three reasonable bullet points for each question should be sufficient.

Algorithms Overview

Deep Q-Network (Lecture 7)

DQN is a model-free reinforcement learning algorithm based on Q-Learning that leverages neural network approximators for estimating the Q-function. The line of research on DQN-based algorithms includes several features that improve the overall stability of the learning, namely: replay buffers, target networks, and double Q-learning.

Vanilla DQN (`'DQN'`) The simplest version of DQN uses a replay buffer to store transitions experienced in the past. Every step through the environment (s, a, r, s') is stored in the buffer for later training. Whenever the number of samples in the buffer exceeds its capacity, the oldest samples are discarded.

Every K steps through the environment, the algorithm samples the buffer for a mini-batch of size B to train the Q-network. Let us define an MSE loss which is a differentiable equivalent of Bellman's equation:

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

where $t_i = (s_i, a_i, r_i, s'_i)$ is the i-th sampled transition and y_i is the target, calculated as follows:

$$y_i = \begin{cases} r_i \text{ if } s'_i \text{ is terminal,} \\ r_i + \gamma \cdot \max_{a'} Q_{\theta'}(s'_i, a') \text{ otherwise .} \end{cases}$$

The vanilla DQN update proceeds by taking a semi-gradient descent step in the direction $\nabla_\theta L(\theta, \theta')$ evaluated at $\theta' = \theta$. Note that y_i is treated as a constant when calculating the gradient with respect to θ even though θ' is set to θ later; thus the name "semi"-gradient. The update is in line with the Bellman update which computes the new Q-values estimate based on the old one.

DQN with Target Network (`'DQN+target'`) Unlike the discrete Bellman operator, the semi-gradient descent needs to perform multiple updates to converge to an optimal minimizer $\theta^\dagger := \text{argmin}_\theta L(\theta, \theta')$. This results in a stability issue, since changing parameters θ changes parameters $\theta' := \theta$; and therefore, both target values y and expected loss minimizer $\theta^\dagger$ constantly shift as we optimize θ. The DQN algorithm with the target network resolves this issue by decoupling θ' and θ. It maintains a separate lagged target network with parameters θ', which are updated less frequently than the original network with parameters θ. To update θ, the algorithm performs a gradient descent step on the loss $L(\theta, \theta')$, this time with a fixed value of θ'.

There are two basic approaches to update the target parameters θ':

1. Hard updates – every M learning steps, copy parameters $\theta' \leftarrow \theta$.
2. Soft updates, or "Polyak Averaging" – every learning step, set $\theta' = \rho \cdot \theta + (1 - \rho) \cdot \theta'$, where ρ is a sufficiently small constant (e.g. 0.005).

Hard update every M steps and soft update with $\rho := 1/M$ should be somewhat comparable in terms of the effect on the learning process. You can implement either, or experiment with both.

Double DQN (`'DoubleDQN'`) DQN also suffers from a shortcoming of the classic Q-Learning algorithm, which is maximization bias. In order to combat overestimation, the selection of maximizing action and realizing its value is split between the target network and the main network, similar to the idea of double Q-learning. In DDQN, the target values are computed according to the following formula:

$$y_i = \begin{cases} r_i \text{ if } s'_i \text{ is terminal,} \\ r_i + \gamma Q_{\theta'}(s'_i, \text{argmax}_{a'} Q_\theta(s'_i, a')) \text{ otherwise.} \end{cases}$$

Note where θ and θ' are used in the formula.

n-step Bootstrapping All of the above equations generalize to n-step returns as well. Instead of using a single transition to calculate each target y_i, we consider a window of n successive transitions $\{s_{i,j}, a_{i,j}, r_{i,j}, s'_{i,j}\}_{j=0}^{n-1}$ from the buffer[3]. Let l be the least index such that $s'_{i,l}$ is terminal, or n if no such index exists. We calculate the n-step return target y_i as follows:

$$y_i = \begin{cases} \sum_{j=0}^{l} \gamma^j r_{i,j} \text{ if } l < n, \\ \sum_{j=0}^{n-1} \gamma^j r_{i,j} + \gamma^n \max_{a'} Q_{\theta'}(s'_{i,n-1}, a') \text{ otherwise.} \end{cases}$$

The last bootstrap term should be adjusted based on the mode of operation, as discussed above.

[3] $s_{i,j+1} = s'_{i,j}$