Reinforcement Learning KU (708.062) WS23

Assignment 3

Deep Reinforcement Learning

Assigned on:	15.12.2023 12:30
Q&A Session:	19.01.2024 12:30
Deadline:	**29.01.2024 23:55**
Submission:	Submit your report (pdf) and Python code files (solved code skeleton). via TeachCenter: `https://tc.tugraz.at/main/course/view.php?id=3110`
Group size:	Groups of two students are allowed for this task.

General remarks

Your submission will be graded based on:

- Clarity and correctness (Is your code doing what it should be doing?)
- Include intermediary steps, and textually explain your reasoning process.
- Quality of your plots (Is everything clearly legible/visible? Are axes labeled? . . .)
- Both the submitted report and the code. If some results or plots are not described in the report, they will **not** count towards your points.
- Your submission should run with **Python 3.7+** and **gym version 0.25.2**
- Code in comments will not be executed or graded. Make sure that your code runs.

1 Deep Q-Learning for Atari Breakout [5 Points]

We will solve the Breakout environment from the OpenAI Gym Framework, using off-policy Q-learning with non-linear function approximation via deep neural networks. We will use the deep Q-learning (DQN) framework, as introduced by [Mnih et al., 2015], which exploits two important mechanisms, namely a *fixed target Q* and *experience replay.* We will use PyTorch for this, such that we can harness automatically computed gradients for parameter optimization.

Open the file `assignment3_dqn_breakout.ipynb` code skeleton which you can run on Google Colab with GPUs. This file creates the ALE/Breakout-v5 environment. *Breakout* is the well-known Atari game environment where the player moves a board at the bottom of the screen that returns a ball to destroy the blocks at the top of the screen (see Figure 1). To solve the Q function learning problem, while one can use a very large set inputs to the agent that represent the environment state, we will use deep convolutional neural networks which can solve the same problem by only looking at the 2D scene frames.

a) **Using a Fixed Target Q Network:** Parameter updates for standard Q-learning (with function approximation) after taking an action a_t in state s_t and resulting in the state s_{t+1} by observing the reward r_{t+1} is $\theta_{t+1} \leftarrow \theta_t + \alpha[y_t^{\mathrm{Q}} - Q_\theta(s_t, a_t)]\nabla_\theta Q_\theta(s_t, a_t)$, where $y_t^{\mathrm{Q}} = r_{t+1} + \gamma \max_{a'} Q_\theta(s_{t+1}, a')$. Essentially this is interpreted as stochastic gradient

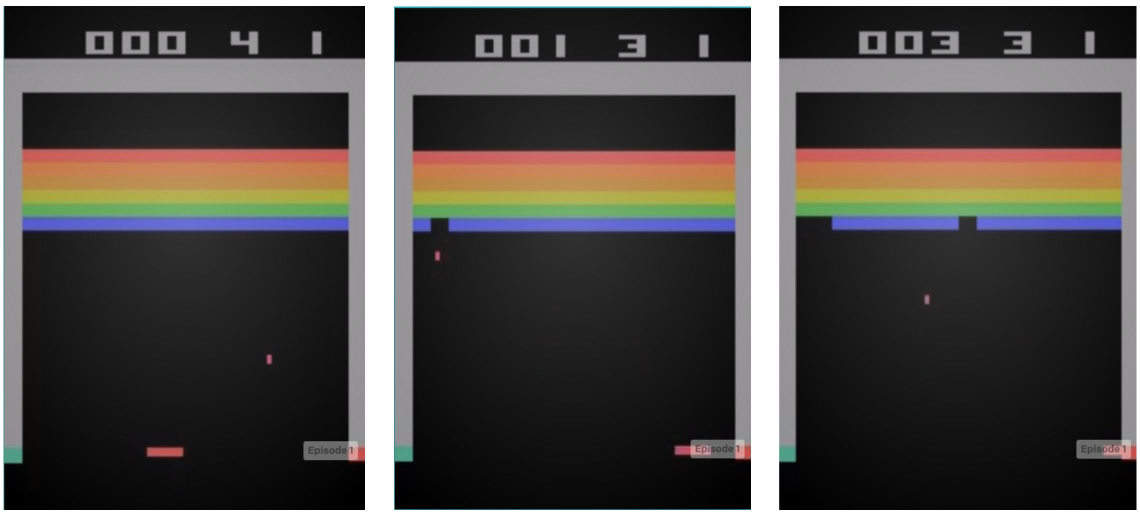

Figure 1: An illustration of the Breakout Atari environment.

descent optimization by updating the $Q_\theta(s_t, a_t)$ towards y_t^{Q} using a mean-squared error loss. To overcome the instability in optimization that can be caused when small updates to $Q_\theta(s_t, a_t)$ change the policy and potentially the target data distribution, DQNs implement a fixed target Q network as an important component.

The fixed target Q network with parameters $\bar{\theta}$ has the identical convolutional architecture as the online Q network, except that we will not update the parameters $\bar{\theta}$ at each step but copy the parameters from the online Q network after every k steps (by setting $\bar{\theta} = \theta$), and keep it fixed otherwise. In this case y_t^{DQN} for stochastic gradient descent optimization will be written as:

$$y_t^{\text{DQN}} = r_{t+1} + \gamma \max_{a'} Q_{\bar{\theta}}(s_{t+1}, a'), \tag{1}$$

resulting in the following parameter updates with mean-squared error loss:

$$\theta_{t+1} \leftarrow \theta_t + \alpha[r_{t+1} + \gamma \max_{a} Q_{\bar{\theta}}(s_{t+1}, a) - Q_\theta(s_t, a_t)]\nabla_\theta Q_\theta(s_t, a_t). \tag{2}$$

Fill the TODOs in the provided code skeleton to implement a convolutional network for the DQN (i.e., explore different configurations by varying the number of Conv2D layers, activations or batch normalization), and solving the task with a fixed target and an online DQN with periodic updates to the target model. Change the provided parameter values if needed. Explain and explore what is the functionality of the variable `burn_in_phase`. You will also see some utility functions to be completed (or modified if you wish) that will pre-process input frames to reduce the input complexity by converting them to grayscale and downsampling of original frames with dimensionality 210×160 into frames of 84×84.

b) **Experience Replay Memory:** We will now implement an experience replay memory by storing state transitions $(s_t, a_t, r_{t+1}, s_{t+1})$ that the agent observes, which allows us to reuse this data by uniformly sampling from the buffer $\mathcal{B}$ later, i.e., $(s_t, a_t, r_{t+1}, s_{t+1}) \sim U(\mathcal{B})$. This framework overcomes the problem of repeatedly observing correlated data in our sequence of observations. We will implement a simple replay memory with a double ended queue, to sample an uncorrelated minibatch of transitions from this cyclic buffer before the parameter optimization steps.

Fill the TODOs in the experience replay memory function to perform appropriate buffering and uniform sampling while training your models consistently through the rest of the pipeline. You can reduce the buffer size/capacity if you run into GPU memory issues.

c) **Optimization Configurations**: We conventionally used a mean-squared error loss. Explore the use of a Huber loss[1] function. Explain mathematically what is the main difference between these two? Then implement these two separately and investigate how different do they perform. Using the model you implemented, observe the influence of initialization for the parameter ϵ. Report the accumulated reward obtained in the test episodes with initial values for $\epsilon \in \{0, 0.5, 1\}$ during training. What is the best value? Is it still possible to solve the task with $\epsilon = 0$ and why?

2 Combining Improvements for DQNs [5 Points]

We will now sequentially incorporate improvements to the baseline DQN model we just implemented. For each model and additional improvement mechanism that will be implemented, make sure to report your results with clear plots where you present the increasing obtained accumulated reward by the agent over episodes.

a) **Double Q-Learning:** This approach proposed by [van Hasselt et al., 2016] exploits two Q functions to disentangle the max operation in Eq. (2), where one model will be used to determine the greedy action selection at s_{t+1} and another one to evaluate this selected action. This will eventually help reducing the overestimation of Q values and allow better and faster learning. To combine this idea with DQNs, we can use our target Q network with parameters $\bar{\theta}$ (which we have been periodically updating the weights of) as a proxy for the second Q function.

We will decompose the max operation in Eq. (1) where the action is both selected and evaluated by the target Q network, and instead perform action selection using the online Q network with parameters θ, and evaluate this action on the target Q network. Hence, in this case y_t^{DDQN} will be:

$$y_t^{\text{DDQN}} = r_{t+1} + \gamma Q_{\bar{\theta}}(s_{t+1}, \arg\max_{a'} Q_\theta(s_{t+1}, a')). \tag{3}$$

Create a boolean variable `run_as_ddqn` at the top of your script which would make your model run as a Double DQN when `True`. You will only need to modify the loss function computation with the new y_t^{DDQN}, as stated in Eq. (3). Train your model and compare the performance of DDQN with respect to the DQN model.

b) **Multi-step Learning:** Previously we have been accumulating a single reward r_{t+1} and then choosing a greedy action at the next step. Alternatively, we can use multiple steps to accumulate rewards. The truncated n-step return from a given state s_t is given by

$$r_t^{(n)} = \sum_{k=0}^{n-1} \gamma^{(k)} r_{t+k+1}. \tag{4}$$

Note that for $n = 1$ this corresponds to the standard $r + 1$ single reward setting. Accordingly, implement a multi-step learning variant for your DQN by adapting y_t^{DQN} into:

$$y_t^{\text{n-DQN}} = r_t^{(n)} + \gamma^{(n)} \max_{a'} Q_{\bar{\theta}}(s_{t+n}, a'), \tag{5}$$

which can be incorporated into our usual mean-squared error or Huber loss function to optimize $Q_\theta(s_t, a_t)$, as well as also combined with a DDQN via Eq. (3). Perform multi-step learning with $n = 3$ steps. Compare the n-step learning approach with respect to a baseline DQN. Incorporate this addition to the DDQN as well (i.e., n-step DDQN) and report comparisons. *Hint: For the n-step learning variant you might want to adapt the experience replay function too.*

[1] https://pytorch.org/docs/stable/generated/torch.nn.HuberLoss.html

c) **Prioritized Experience Replay:** Your implementation of DQN so far uniformly samples experiences from the replay buffer. It would make sense however to sample more frequently those transitions from which there is much to learn [Schaul et al., 2016]. Discuss in your report (verbally and mathematically) an example to how one can implement such a prioritized experience sampling function for the DQN (e.g., proportional to which measure/criterion should we sample from the experiences). For this part, you do not need to implement anything. Explain and discuss in detail why your choice makes sense.