Reinforcement Learning KU (DAT.C307UF), WS24
Assignment 3
RL in a Grid World, Deep Reinforcement Learning

Thomas Wedenig
thomas.wedenig@tugraz.at

Teaching Assistant:	Erwin Pfeiler
Points to achieve:	30 pts
Deadline:	17.01.2025 23:59
Hand-in procedure:	You can work in groups of **at most two people**. **Exactly one** team member uploads four files to TeachCenter: **The report (.pdf)**, `rl_frozen_lake.py`, `dqn_atari.ipynb` and `frozen_lake_utils.py`. The first page of the report must be the **cover letter**. Do not upload a folder. Do not zip the files.
Plagiarism:	If detected, 0 points for all parties involved. If this happens twice, we will grade the group with "Ungültig aufgrund von Täuschung"

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? …)
- If some results or plots are not described in the report, they will **not** count towards your points.
- Code in comments will not be executed or graded. Make sure that your code runs.
- The submitted Jupyter Notebook `dqn_atari.ipynb` should be able to (approximately) reproduce all of your experiments when setting the parameters accordingly (e.g., setting `run_as_ddqn`). The code in the notebook should work if all cells are executed in-order.

The file `frozen_lake_utils.py` provides plotting utilities that you can use. You do not need to edit this file (you can however, as this file is also included in your submission).

1 Reinforcement Learning in a Grid World [7.5 Points]

We will revisit the FrozenLake[1] grid world environment from Assignment 2. In many practical problems, we do not know the MDP dynamics. Hence, we will now implement algorithms that *do not rely on knowing the MDP dynamics*.

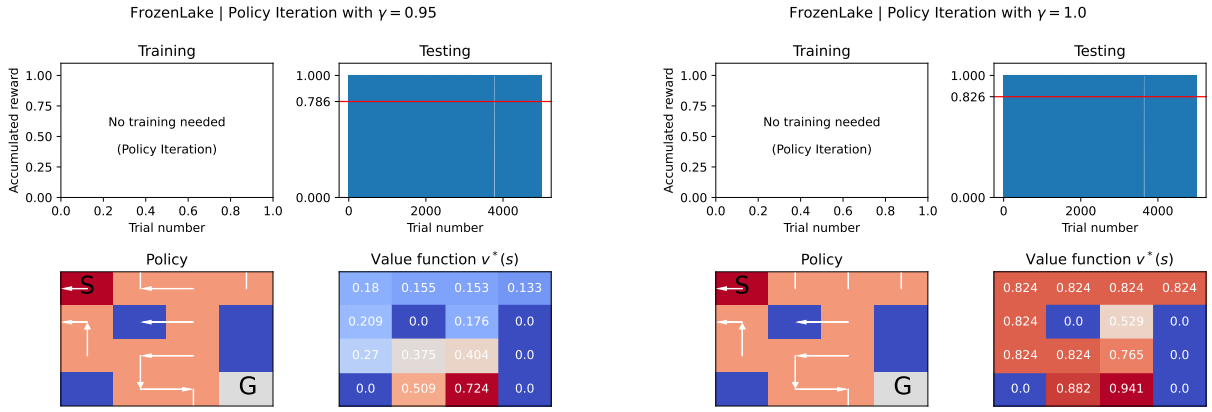

Figure 1: Value function and policy that were acquired using *Policy Iteration* with $\gamma \in \{0.95, 1\}$ in the `FrozenLake` environment (see Assignment 2). In this plot, we test the policy using 5000 episodes.

a) Carefully read the code in the file `rl_frozen_lake.py` and fill in the `TODOs`. More specifically, this requires you to implement the following algorithms, which differ in how they update their current estimate of the Q values:

- SARSA (State, Action, Reward, State, Action)
- Q-Learning
- Expected SARSA

Moreover, you are tasked to implement an ε-greedy policy. Start training your agent with $\varepsilon = 1$ and decay ε throghout training using the `eps_decay` parameter. Train your agent for $10,000$ episodes and test the policy you have found using 5000 episodes. **Important:** When simulating test episodes, choose all actions deterministically ($\varepsilon = 0$).

For each of the 3 implemented algorithms and for each $\gamma \in \{0.95, 1\}$, find good values for the hyperparameters α ("learning rate") and `eps_decay`. For each of the 6 cases, report the best hyperparameters you have found and include the plot that is saved by `plot_frozenlake_model_free_results`. Moreover, report which values of α and `eps_decay` you have tried during your hyperparameter search.

Compare the plots of the 3 different algorithms and briefly explain any differences.

b) Compare the policies you have found with *SARSA*, *Q-Learning*, and *Expected SARSA* with the policies that were found using *Policy Iteration* (see Figure 1). Also compare the corresponding value functions v. What do you find? (Make sure you compare plots where γ was the same.)

[1] `https://gymnasium.farama.org/environments/toy_text/frozen_lake/`

2 Deep Q-Learning for Atari Breakout [17.5 Points]

We will solve the Atari Breakout environment 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 network* and *experience replay*. We will use `PyTorch` for this, such that we can harness automatically computed gradients for parameter optimization.

Carefully read and understand the code skeleton in the file `dqn_atari.ipynb`. You can also open this file using Google Colab, which will give you access to GPUs to train your agent. *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 2). While one can use hand-crafted state representations to solve this environment, we will use deep convolutional neural networks that directly take the 2D scene frames as input.

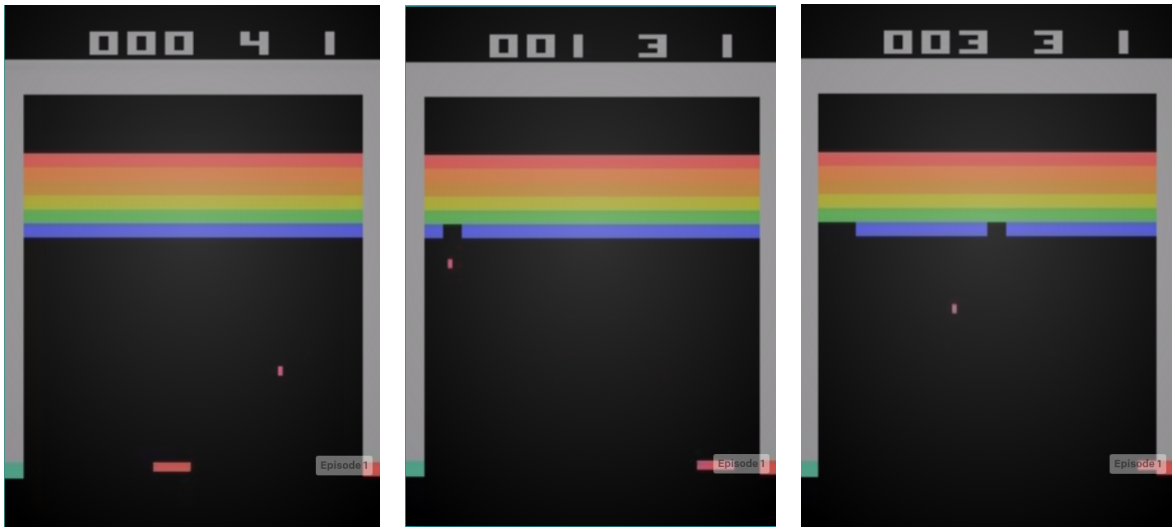

Figure 2: An illustration of the Breakout Atari environment.

a) **Using a Fixed Target Q Network:** Assume the agent is in state s_t and takes action a_t, which results in an observed reward r_{t+1} and a new state s_{t+1}. The parameter updates for standard Q-learning (with function approximation) is then $\theta_{t+1} \leftarrow \theta_t + \alpha[y_t^{\mathrm{Q}} - Q_\theta(s_t, a_t)]\nabla_\theta Q_\theta(s_t, a_t)$, where $y_t^{\mathrm{Q}} = r_{t+1} + \gamma \max_{a'} Q_\theta(s_{t+1}, a')$. Essentially this is interpreted as stochastic gradient 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:

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

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 `TODO`s in the provided code skeleton to implement a convolutional network for the DQN. Explore different network configurations by varying the number of Conv2D layers, activations, and

batch normalization layers. Solve the task with a fixed target network and an online DQN with periodic updates to the target model. Experiment with changing the hyperparameters listed in the code skeleton and report on your results. Explain and explore the functionality of the variable `burn_in_phase`. You are free to also modify other parts of the code, like the pre-processing pipeline: For example, by default, we will convert the input frames to grayscale and downsample them (going from the original dimensionality 210×160 to 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., $(s_t, a_t, r_{t+1}, s_{t+1}) \sim \mathit{Uniform}(\mathcal{B})$. We will implement a simple replay memory with a double ended queue, to sample a minibatch of transitions from this cyclic buffer before the parameter optimization steps.

Fill the `TODO`s in the experience replay memory function to perform appropriate buffering and uniform sampling while training your models. You can reduce the buffer size/capacity if you run into GPU memory issues.

In your report, explain why we use a replay buffer in the first place. What assumption of many machine learning models is violated if we directly learn from state transitions as they occur? Briefly explain why this is the case.

c) **Optimization Configurations**: We conventionally used a mean-squared error loss. Explore the use of the Huber loss[2] function. Mathematically explain the differences between these two functions. Investigate the difference you emprically observe when using the `HuberLoss` function from `PyTorch` (as opposed to the mean squared error loss).

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

3 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 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:

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

Set the boolean variable `run_as_ddqn` to `True` and implement the DDQN extension. 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) **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.