Lecturer: Henning Sprekeler

Deep Q-Learning

The report for this project as well as the source code should be handed in through the Moodle interface. You can discuss your advances and hurdles with the project instructor until the end of the lecture period during the weekly supervision meetings, or by sending them an email.

1 Summary

The goal of this miniproject is to implement a set of Deep Q-Learning algorithms and to analyse their learning dynamics. In order to do so, you will first implement a discrete Reinforcement Learning environment using `Pycolab`[1] and train several Deep Reinforcement Learning (DRL) agents using `PyTorch`[2]. You will perform hyperparameter experiments and gain insights into the robustness of the learning dynamics as well as the general design of experiments in DRL.

Note I: Most of the content of this miniproject will be thoroughly discussed during the lecture on the 14th of June 2024. Until then please read the paper by Mnih et al. (2015).

Note II: Depending on your computational resources, the project might occupy your computer for a while. In order to circumvent any problems, please have a look at Google Colab.

2 Detailed instructions

2.1 Installation

Along with this project description, we provide a Python Notebook `skeleton_dql.ipynb`, which walks you through the setup of the project. Start by opening the notebook and installing the following packages : `gym==0.15.4`, `pycolab==1.2`, `torch==1.2.0`. Furthermore, we provide example code for a simple environment that you can use as a template for your own implementation. To install this environment, unzip the file `gym-grid.zip`, change into the directory `gym-grid` and execute `pip install -e .` .

2.2 Environment

General remarks: Throughout the Deep Reinforcement Learning research community it is common to design virtual environments in order to test the performance of algorithms. These environments define benchmark tasks which allow researchers to probe the capabilities of the trained agents. Classical examples include the ATARI benchmark (Bellemare et al., 2013) as well as continuous control environments (MuJoCo, Todorov et al., 2012). The OpenAI gym (`https://gym.openai.com/`) style has become a standardised interface for virtual environments. For example, a transition in an environment `env` by an agent `agent` can be executed by the following code.

[1] An open-source project by DeepMind which allows for flexible implementation of RL environments in the OpenAI gym style: `https://github.com/deepmind/pycolab`.
[2] Please consult MHBF computer exercise 1 for a more detailed introduction to training Deep Neural Networks.

```
env = gym.make('<env_name>')       # Create the environment
state = env.reset()                # Reset the environment
action = agent.get_action(state) # Obtain action from agent
next_state, rew, done, _ = env.step(action) # Step Transition
```

We will follow this methodology in this miniproject. We will focus on two simplistic gridworld environments with high-dimensional state representations (i.e., pixel arrays and not tabular states):

1. To familiarise yourself with the general structure of `Pycolab` environment, start by visualising the state representation of an environment for a Linear Track, in which an agent (in red) can move around within walls (grey) and collect food (yellow) or a diamond (turquoise). Visualize this environment in the `skeleton_dql` notebook. Inspect the definition of the environment in `gym_grid/linear_track.py`. Note that the plotted pixel array $s \in \mathbb{R}^{3\times32\times4}$ is not merely a visualization of the environment, but the actual state representation accessible to the agent. The 4 channels encode the 4 unique objects in the environment (agent, walls, food, diamond). The agent has 3 action choices ($\mathcal{A} = \{\textit{left, right, stay}\}$), which lead to deterministic state transitions. Visualise a example trajectory of a random agent using the notebook.

2. *Deadly Gridworld*: Implement a more complex gridworld (Fig. 1, right). Extend the action space to include *up* and *down*. Enlarge the state space ($s \in \mathbb{R}^{6\times14\times5}$) and extend the previous example by a patroller (green) and poison (black). The patroller keeps walking from top to bottom and back. A collision with the agent leads to the termination of the episode. *Hint*: For the implementation, have a look here: https://tinyurl.com/w6d8wvs.

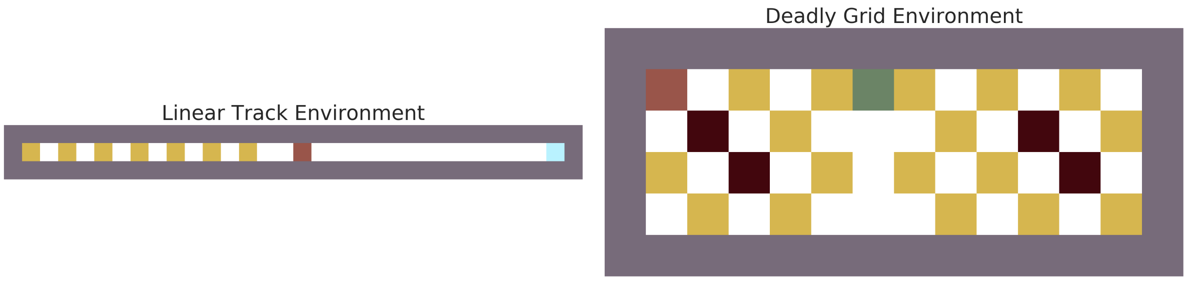

Figure 1: RL environments. In every trial, the agent (red box) starts at the same start position. It can collect small positive (+100) / large positive (+10000) / negative rewards (-100) by stepping onto yellow/turquoise/black boxes. The rewards are consumed and disappear afterwards. The green patroller has to be circumvented to maximise the reward of the agent. Once all rewards of one kind (yellow or turquoise) are collected or the patroller "catches" the agent, the episode terminates.

2.3 Deep Q-Learning Agents

General remarks: The breakthrough moment of DRL can be dated back to Deep Q Networks (DQNs; Mnih et al., 2013, 2015). We will start by implementing a base learning loop for an RL agent that uses function approximation to estimate the action value function, $Q(s, a) \approx Q(s, a; \theta)$. Afterwards, we will successively add details to the agent & analyse the improvements for both the *Linear Track* and the *Deadly Gridworld* environments. Set the random seeds (e.g. in `numpy`, `random`, `torch`) for exact replicability of your results.

1. *Base Learning Loop*: In order to train a Deep Q-Learning agent, we require a set of ingredients. These include the following:

   - Agent architecture: In Pytorch, define a Multi-Layer Perceptron (MLP) with the parameters listed in Section 2.5. Make sure that the dimensions align with the given setting. An MLP for the *Linear Track* environment should have input dimension 384 (flattened input vector - $3 \times 32 \times 4$) and an output dimension of $|\mathcal{A}| = 3$.
   - Action-Perception Loop: Implement a training loop, which generates episodes of transitions $\{< s_t, a_t, r_{t+1}, s_{t+1} >\}_{t=1}^{T}$ based on $Q(s_t, a_t; \theta_k)$ (given parameters θ_k at optimisation iteration k). Obtain the action value estimates by performing a forward pass through the network with the state input s_t. The greedy action is then simply given by the maximum over the output of the network. Ensure that the agent sufficiently explores the environment. The simplest way to do so is ϵ-greedy. Reduce (anneal) the amount of exploration during the course of training (starting from 1, $\epsilon \to 0$). Terminate an episode after $T = 50$ (*Linear Track*) or $T = 100$ (*Deadly Gridworld*) steps (or earlier, when the agent hits the patroller or has collected the rewards).
   - Learning: Deep Q-Learning formulates the refinement of action value estimates as an optimisation problem, in which the agent iteratively minimises a regression loss, e.g. the Mean Squared Bellman Error (MSBE)[3]:

     $$\mathcal{L}_{MSBE} = \mathbb{E}_{s_t, a_t, r_{t+1}, s_{t+1}}[(Q(s_t, a_t; \theta_k) - Y_k^t)^2] \text{ with } Y_k^t = r_{t+1} + \gamma \max_{a' \in \mathcal{A}} Q(s_{t+1}, a'; \theta_k) \,. \tag{1}$$

     After each step transition, learning is performed by calculating a gradient estimate of the loss function with respect to the parameters θ_k and performing a gradient descent step:

     $$\theta_{k+1} = \theta_k + \eta \nabla_\theta \mathcal{L}_{MSBE} \,.$$

     Getting a good approximation of the average in the MSBE loss is one of the central problems of RL. Implement the MSBE loss function such that the set of transitions over which the average is calculated is provided as an argument.

     Due to memory constraints, the first implementations (Fitted Q-Learning; Gordon, 1996) used an online gradient estimate from the most recent transition to estimate the gradient. Implement such an online update and analyse the learning stability for both environments. Choose a reasonable discount parameter γ (i.e., $\gamma \in [0.95, 0.99]$ - see default hyperparameter section). Choose an optimisation algorithm (i.e., Adam) and set the learning rate $\eta = 0.001$. Use an implementation from `torch.optim`. *Note*: It is important to detach the target values from the computation graph. Discuss in your final report why this is the case.
   - Evaluation: Evaluate the performance of the agent regularly after a set of learning updates (e.g., every 500 steps), by running a couple of episodes of the current agent with relatively weak exploration (i.e., $\epsilon = 0.01$) . Average the resulting cumulated and discounted episode rewards $\sum_{t=0}^{T} \gamma^t r_t$ (also known as the *returns*) and plot them as a function of time. The agent should improve over time [4].

[3] Please be careful when computing target values Y_k^t for terminal state transitions. In this case the target boils down to $Y_k^t = r_{t+1}$. *Hint*: This can easily down by masking the bootstrap estimate with $(1 - \texttt{done})$.

[4] You should be able to see performance improvements on the *Linear Track*/*Deadly Gridworld* env after training for 20,000/100,000 updates. On a 2,7 GHz Quad-Core Intel Core i7 processor this should take approximately 45-90 secs/350-500 secs.

2. *Experience Replay Buffer* (Lin, 1992): Since Fitted Q-Learning is an online algorithm, it uses each transition tuple $< s, a, r, s' >$ exactly once to update $Q(s, a; \theta_k)$. This can be sample inefficient. Alternatively, one can store transitions in a buffer/dataset $\mathcal{D} = \{< s, a, r, s' >\}$. The buffer has a limited capacity $N^{replay} \in \mathbb{N}^+$. As we fill the buffer, the oldest transitions will be discarded. During learning, the buffer is used by uniformly sampling a "mini-batch" of transitions from the buffer and computing a batch-average of $\nabla_\theta \mathcal{L}_{MSBE}$. Implement such a buffer for batch-size 128 & capacity of 1000. Compare the learning dynamics with and without the replay buffer for both environments.

3. *Target Networks* (Mnih et al., 2013, 2015): Deep networks are powerful function approximators. Unfortunately, their generalisation ability can lead to strong learning instabilities. By implicitly changing the target values Y_k^t at every time step, Fitted Q-Learning may fail to converge. This can be stabilised by using a second slowly changing target network with parameters θ^- to compute target values:

$$Y_k^{t,DQN} = r_{t+1} + \gamma \max_{a' \in \mathcal{A}} Q(s_{t+1}, a'; \theta_k^-) \ .$$

The target network is regularly updated after a fixed amount of iterations, by copying the parameters of the main network $\theta^- \leftarrow \theta$. Implement such a separate target network. Update the target network every $C = 500$ update steps.

4. *Loss Function Comparison*: Next to the MSBE objective, additionally implement the smooth L1 Huber loss, e.g. using the `torch.nn.SmoothL1Loss` with default settings:

$$\mathcal{L}_{SmoothL1} = \mathbb{E}_{s_t, a_t, r_{t+1}, s_{t+1}} \Big[\begin{Bmatrix} 0.5(Q(s_t, a_t; \theta_k) - Y_k^t)^2, & \text{if } |Q(s_t, a_t; \theta_k) - Y_k^t| < 1 \\ |Q(s_t, a_t; \theta_k) - Y_k^t| - 0.5, & \text{else} \end{Bmatrix} \Big]. \tag{2}$$

This objective is less sensitive to large temporal difference errors, which can destabilize the learning dynamics. It has been used in several DQN-follow-up research papers. How do MSBE and the L1 objective compare on the different environments (track and grid)?

5. *Hyperparameter Intuition*: The implemented algorithms have a significant number of hyperparameters (e.g., network architecture, optimisation algorithm, learning rate, batch size, replay buffer capacity, frequency of target network updates, exploration strategy, etc.). Choose a subset of such and vary them within a reasonable range. Retrain the agents and visualize your results appropriately. For example: What happens if you don't detach the target values during training? What are interactions between learning rate and batch-size? What happens when you use a shallow network without hidden layers?

6. *Behavioural Analysis*: Visualise episodes (rollouts) of the Deep RL agents at different stages of learning and for different discount parameters ($\gamma \in [0.7, 0.8, 0.9, 0.99]$). This can be done by saving the agent's weight parameters and later reloading them. Can you think of different interpretations for the discount factor?

7. **Bonus:** *Prioritized Experience Replay (PER)* (Schaul et al., 2015): Until now, batches of transitions were sampled uniformly at random. Intuitively, one might improve sample efficiency by prioritising important transitions in the sampling process. Schaul et al. (2015) propose to sample transitions proportionately to their associated *learning progress*, defined as the magnitude of the TD error (i.e. $|Q(s_i, a_i; \theta_k) - Y_k^i|$). The i-th sample from the replay buffer is then sampled with probability

$$P(i) = \frac{p_i^\alpha}{\sum_k p_k^\alpha},$$

where $p_i = |Q(s_i, a_i; \theta_k) - Y_k^i| + \kappa$ are the so-called priorities and $\alpha \in [0, 1]$ controls the amount of prioritisation. κ is a small constant (i.e. $1e - 10$) and ensures that each transition has a non-zero probability of being sampled. When implementing PER, make sure that a new transition is sampled at least once by setting its priority higher than the maximum priority in the buffer. Set $\alpha = 0.6$. At each step, only update the priorities that are sampled at the current step. The prioritized sampling introduces a biased version of the MSBE objective in equation 1. This bias can be corrected by *importance* reweighting the temporal difference errors:

$$\mathcal{L}_{PER} = \mathbb{E}_{i \sim P(i)}[\tilde{w}_i (Q(s_i, a_i; \theta_k) - Y_k^i)^2]$$

where $\tilde{w}_i = \frac{w_i}{max_i w_i}$ with $w_i = \left(\frac{1}{N^{replay}} \frac{1}{P(i)}\right)^\beta$ are importance weights. The parameter $\beta \in [0, 1]$ mitigates the impact of the prioritization and is usually linearly increased to 1 (usually starting at 0.4). Implement the PER scheme & evaluate the performance for different buffer capacities.

2.4 Report

- Write a report that presents the results of the previous implementations. Marks will be based on the clarity of the presentation and on the thoroughness of the analysis.
- Please discuss the importance of state representation. What state representation would you choose to solve the *Linear Track* problem with simple tabular Q-learning?
- Give detailed arguments and intuitive explanations for your results. Furthermore, visualise the behaviour of the trained agents over the course of training. Throughout the report and implementation make sure to cite the appropriate sources as well as references.

2.5 Default Hyperparameters

DQN - 1-Hidden Layer Multilayer Perceptron/Learning Hyperparameters:

Hyperparameter	Value	Hyperparameter	Value
Discount γ	0.99	# Hidden Units	128
Learning Rate η	0.001	# Hidden Layers	1
Buffer Size N^{replay}	1000	Optimizer	Adam
Target Updates C	500	Batch-Size	128
ϵ-Decay (80% steps)	Linear	# Eval Episodes	10 (every 1k steps)
Activation Function	ReLU		

References

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

Gordon, G. J. (1996): "Stable fitted reinforcement learning," in *Advances in neural information processing systems*, 1052–1058.

Lin, L.-J. (1992): "Self-improving reactive agents based on reinforcement learning, planning and teaching," *Machine learning*, 8, 293–321.

Mnih, V., K. Kavukcuoglu, D. Silver, A. Graves, I. Antonoglou, D. Wierstra, and M. Riedmiller (2013): "Playing atari with deep reinforcement learning," *arXiv preprint arXiv:1312.5602*.

Mnih, V., K. Kavukcuoglu, D. Silver, A. A. Rusu, J. Veness, M. G. Bellemare, A. Graves, M. Riedmiller, A. K. Fidjeland, G. Ostrovski, et al. (2015): "Human-level control through deep reinforcement learning," *Nature*, 518, 529.

Schaul, T., J. Quan, I. Antonoglou, and D. Silver (2015): "Prioritized experience replay," *arXiv preprint arXiv:1511.05952*.

Todorov, E., T. Erez, and Y. Tassa (2012): "Mujoco: A physics engine for model-based control," in *2012 IEEE/RSJ International Conference on Intelligent Robots and Systems*, IEEE, 5026–5033.