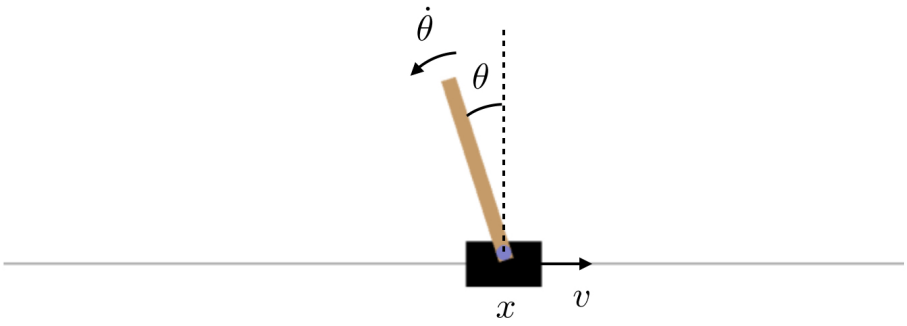
Deep Reinforcement Learning

Exercise 2: DQN

Lecturer: Armin Biess, Ph.D.

Due date: 31.8.2021

In this exercise you are asked to solve a control problem with continuous state and discrete action space using DQN. The environment is given by the OpenAI Gym CartPole environment (https://gym.openai.com/envs/CartPole-v0/). The goal of CartPole is to balance a pole connected with one joint on top of a moving cart (see figure). The state is continuous and given by four real numbers $\boldsymbol{s}$ = [cart position, cart velocity, pole angle, pole angular velocity] (cart position $\in [-4.8, 4.8]$, cart velocity $\in [-\infty, \infty]$, pole angle $\in [-0.418, 0.418]$rad and pole angular velocity $\in [-\infty, \infty]$). The actions are discrete and given by $\boldsymbol{a}$ = [push cart to the left, push cart to the right] = [0,1]. The reward is +1 for every step taken. An episode is terminated if (i) the pole angle goes beyond ± 12 degrees (± 0.2094 rad), (ii) the cart position is larger than 2.4. The score is the total reward (= total number of steps made) received in one episode. The overall performance of the agent is evaluated by using a task score. The task score is defined as the maximum of the minimum score over five *consecutive* episodes. For example, if the agent achieves in 10 episodes the following scores: 7, 9, 10, 20, 35, 15, 25, 5, 8, 8, then the task score is max[min[7, 9, 10, 20, 35], ..., min[10, 20, 35, 15, 25], ... min[15, 25, 5, 8, 8]]= 10.



IMPORTANT: An incomplete code for DQN is provided. The sections that you need to work on are marked with TODO. Please also note the provided hints for the solution in the code. The task score is already implemented.

a.) [35 pts] ***DQN***

Complete the function `train_model()` in `DQN.py` by implementing the barebone of DQN.

b.) [15 pts] ***Replay Buffer***

Add to the code a `ReplayBuffer` class that stores experiences and takes as input argument the capacity of the replay buffer. Set the capacity to 100000. Implement the following two methods

- `push`: to push experiences into the buffer
- `sample`: to sample minibatches of experiences from the buffer

Note: A class skeleton `buffer.py` is attached.

c.) [10 pts] ***Hard Target Network-Update***

Add to the code a `target_network` with a *hard* target network update frequency of 15 episodes, i.e., after every 15 episodes the weights of the policy network are copied to the target network:

$$\theta' \leftarrow \theta\,.$$

Initialize the target network with the weights/biases of the policy network.
Note: Insert the hard target update in line 228 of `DQN.py`.

d.) [10 pts] ***Soft Target Network-Update***

Add to the code a *soft* target update option, i.e., after *every* update the target weights are updated according to

$$\theta' \leftarrow \tau\theta + (1-\tau)\theta'$$

with $\tau = 0.01$ Initialize the target network with the weights/biases of the policy network.
Note: Insert the soft target update in line 210 of `DQN.py`.

e.) [25 pts] ***Effect of Replay Buffer and Target Network***

In this section you will study the effect of the replay buffer and target network in DQN. Evaluate the performance of your agent for different settings (see table) using the task score. The computation of the task score is already implemented in the code. Store the weights of the policy network every time the task score improves. One experiment consists of 200 episodes, each of maximally 500 steps. As your environment is stochastic, run five experiments for each setting and fill the average task score into the table. For settings with target networks investigate *soft* and *hard* target updates. Which difference do you observe in the loss and mean Q error?

target update	with replay with target	with replay without target	without replay with target	without replay without target
hard		x		x
none	x		x	
soft		x		x

Note: to run your DQN without replay buffer you can set the capacity of the replay buffer to 1 and the batch size to 1.

f.) [5 pts] ***Movie***

Generate a movie of your best performing agent (DQN) using the provided `cartpole_play()` function.

Remarks:

- Make sure that you have all required packages in your Python environment installed.
- Pay attention to pass correct tensor sizes and data types.
- This exercise makes use of the `PyTorch` library (https://pytorch.org/).
- A `Network` class in `model.py` is provided that builds a feedforward neural network with one hidden layer of 64 neurons with leaky ReLu activation functions and one linear output layer of two neurons. The loss function is the Huber loss, which is a generalization of the MSE error loss, discussed in class.