Exercise 7
Double Deep Q-Networks

"Two Days Late, I'm sorry" - Edition

In this exercise we will start from our DQN implementation from last exercise and extend it to the method of Double Deep Q-Networks (Double DQN). Remember that in the last exercise, we used

$$Y_t^{\mathrm{Q}} \equiv R_{t+1} + \gamma \max_a Q\left(S_{t+1}, a; \boldsymbol{\theta}_t^-\right) \tag{1}$$

as our target to calculate the mean-squared Bellman error (MSBE), where $\boldsymbol{\theta}_t^-$ means the parameter set of the second target network.

It can be shown mathematically, that using such max operation to estimate the value of the next state can lead to optimistic overestimation, as we use the same function approximator to select and evaluate an action. The original idea of Double Q-Learning Double Q-Learning, introduced in 20.., is to decouple the selection from the evaluation by learning two Q-value estimators at the same time. Because we already have two Q-networks in the DQN method, we can directly take advantage of this. Instead of training two DQN networks with one target network each (four networks in total), Double DQN follows a more simple approach, where we use the main network to select the best action based on its value estimates, but use the value estimate of the target network for that action. This means that our update targets change into:

$$Y_t^{\mathrm{DoubleDQN}} \equiv R_{t+1} + \gamma Q\left(S_{t+1}, \operatorname{argmax} Q\left(S_{t+1}, a; \boldsymbol{\theta}_t\right), \boldsymbol{\theta}_t^-\right), \tag{2}$$

where $\boldsymbol{\theta}_t$ means the main network parameter set. Eventhough it seems complicated at first, it is actually really easy to adapt our implementation to this idea with a few lines of code changes.

To spice things up, we will further use our Double DQN implementation to train agents on the famous Atari environment. In the following, we will focus on the game of Pong, but you are free to choose other Atari games for your agent to learn playing. Be sure to use the `requirements.txt` file we ship with the code skeleton through running `pip install -r requirements.txt`. Should you encounter problems after installing the Atari Python packages on Windows, please try out the following solution. Alternatively, try to install Atari thorugh conda, i.e. `conda install -c conda-forge atari_py` (provided you are using Anaconda). This is the only exercise, where additional compute power will be helpful. If your computer has a NVIDIA GPU with Cuda capabilities, please install the GPU compiled version of torch.

Programming Tasks:

1. **Implementing Double DQN** As elaborated above, change the implementation, such that the new, double method is used to calculate the value estimation error. We marked the respective areas requiring adaptation inside the code. Be careful to not accidentally backpropagate gradients through

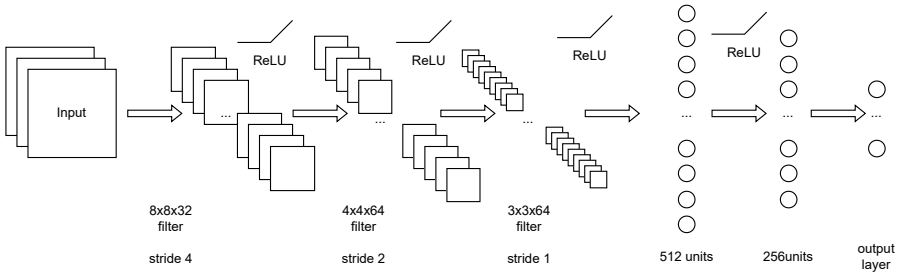

Figure 1: Original DQN network architecture for the Atari environment.

the main network as part of the action selection process. Have a look at the `calculate_msbe` function for this task.

You should first verify the correctness of your implementation on the `Cartpole-v1` environment before moving on. Here, the learning speed will generally not drastically improve compared to vanilla DQN. After you successfully tested out your implementation, you can move on to having some fun with Atari.

2. **Network Architecture** The Atari environments can either be observed in RAM or RGB mode. We advice to start with the RGB mode and maybe test out the RAM mode afterwards. In contrast to the vector observation space from Cartpole, a simple feed-forward network does not suffice to estimate state-action values, as for images we usually need CNNs to learn proper models. Implement the architecture as seen in Figure 1 inside the `DQNNetworkVisual` class. What is not shown in the picture is that you will also require ReLU layers.

3. **Additional Adaptations** Besides target networks, training DQNs and RL agents in general involves some engineering and tricks. Implement the following training features:

   - `learning_start`: It is often helpful to collect some experience and fill the replay buffer a little bit before starting the actual optimization.
   - `train_freq`: Similarly, it makes sense to adapt the training frequency (the number of timesteps to run the environment before performing an update step).

4. You are now ready to train. Compared to Cartpole, this might take some time.