Exercise 6
Value Function Approximation

This exercise is all about value function approximation and how to implement it in PyTorch. We will start with DQN in the first exercise, and build a more sophisticated version, i.e. Double DQN, in the next one.

1 Deep Q-Networks

We now want to implement DQN using PyTorch and find good policies for discrete action space environments. We will test our agent on the famous `CartPole` environment.

Programming Tasks:

1. **Neural Network Architecture** The neural network that will be used as a Q-function approximator has to be defined inside the `DQNNetwork` class constructor. Your task is to define a network which:

   - maps from `num_obs` (dimensionality of input observations) to `num_actions` (one Q value for every possible action)
   - uses `torch.nn.Linear` with ReLU/Tanh activations
   - has a `torch.nn.Linear` layer as output layer (try to think about why this is)

   You can decide by yourself how many layers and which intermediate feature dimensionality you use, but for easy environment like `CartPole` one or two layers and an intermediate dimensionality of 64 or 128 is more than enough (and often also works much better and converges faster).

   You can either pass inputs to the network manually through every layer or wrap all you layers into a `torch.nn.Sequential` object and do this in one call.

2. **Epsilon Decay** DQN addresses exploration with the use of ϵ-greedy policies, which performs a random action with probability ϵ, whereas the greedy action w.r.t the approximated Q-function is taken with probability $1 - \epsilon$. Implement the function `epsilon_by_timestep`, which returns:

   - `epsilon_start` for `timestep == 0`
   - `epsilon_end` for `timestep >= frames_decay`
   - a linearly decaying ϵ for `0 < timestep < frames_decay`

3. **Action Prediction** The `DQN` class represents the DQN agent. Based on the observation returned by the gym environment and a specific `epsilon` value the agent should either:

   - perform a random action with probability `epsilon`
   - perform the action which has maximum Q-value for the given observation. For this you will have to wrap the observation (`np.array`) into a `torch.Tensor`, bring it into batch format, pass it through the network, and return the index of the network outputs which has maximum value.

4. **Replay Buffer** For off-policy methods like DQN to work it has been shown that it can be beneficial to use a replay buffer, with which past experience of the agent can be stored and reused during training. As an exercise you should implement the `__init__`, `__len__`, `put` and `get` methods inside the `ReplayBuffer` class.

   - You can decide yourself how you want to implement this. One way would be to hold a list of tuples `(obs, action, reward, next_obs, done)` inside a `list` or `collections.deque` object and enforce a maximum size on it (delete oldest entries first once your replay buffer is full).
   - `get` should return *five* lists, i.e. `obs_lst`, `action_lst`, ... Should you decide to store entries as a list of tuples, you can convert them into said list format via `zip(*tuple_list)`.

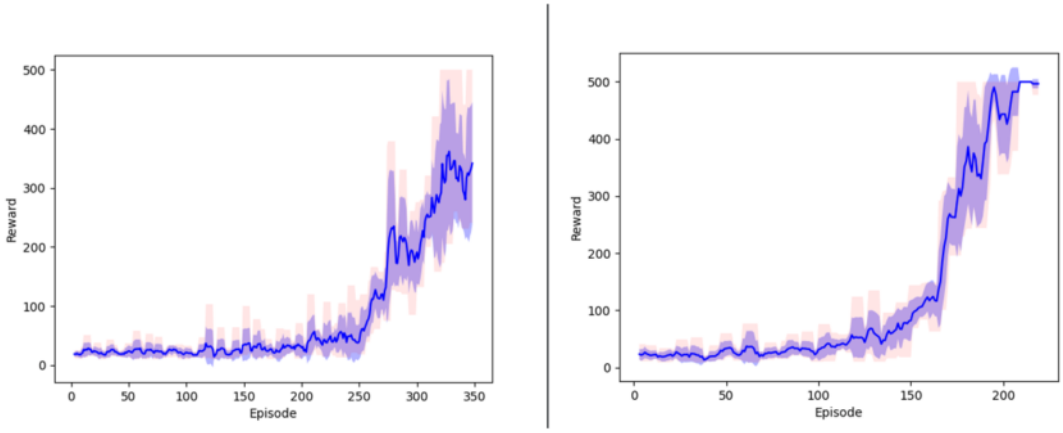

Figure 1: (left) without target network; (right) with target network; Sample reward plot we were able to achieve with our implementation on `CartPole` (hyperparameters as provided in skeleton, Linear(128) → ReLU → Linear NN architecture)

- In `get batch_size` random entries from you buffer should be returned. Use `random.sample(...)` for this.
- Don't forget to implement the `__len__` function which returns the number of entires inside your replay buffer.

5. **MSBE** In order to train your DQN method you will need to calculate the *mean-squared Bellman error* (MSBE) inside `DQN.compute_msbe_loss` and return it. More specifically, for a set $\mathcal{D}$ of `batch_size` transitions sampled from the replay buffer the MSBE is defined as:

$$L(\phi, \mathcal{D}) = \mathbb{E}_{(s,a,r,s',d)\sim\mathcal{D}}[(Q_\phi(s,a) - (r + \gamma(1-d)\max_{a'} Q_\phi(s',a')))^2],$$

where we've used a Python convention of evaluation `True` to 1 and `False` to 0.

- In order to select the Q-values from a batch tensor calculated from a batch of states, which correspond to the actions taken in the sampled transitions, you can use `torch.gather`. Alternatively you can use a loop.
- If s' is a terminal state its Q-values are per definition zero. Take this into account during your calculations.

6. **Target Network** Your network should now already be able to learn on easier problems. For more challenging problems it has been shown that it can be beneficial to use a fixed target network with which the TD target values (the $r + \gamma max_{a'} Q$ term above) are calculated.
   (a) Create a second `DQNNetwork` object inside the `DQN` class constructor.
   (b) Calculate the TD target values using only this network.
   (c) Synchronize it on construction and periodically during training using `target_net.load_state_dict(main_net.state_dict())`.

Try to play around with hyperparameters a little bit. You might also try out training your DQN in a little bit more challenging environments. Have a look at the documentation to get an overview.

2 Double Deep Q-Networks (Advanced)

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

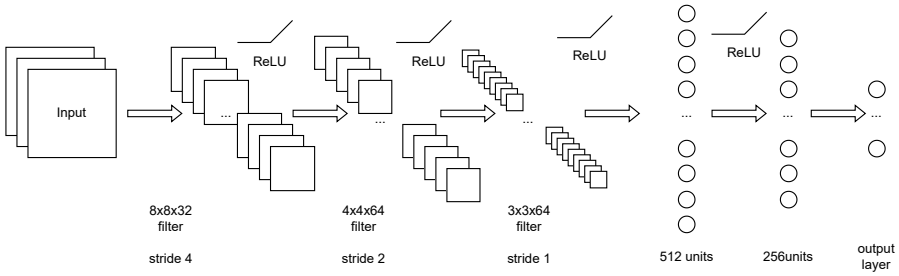

Figure 2: Original DQN network architecture for the Atari environment.

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:

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

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 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 2 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.