Exercise 6
Deep Q-Networks

In this exercise you will learn how to implement DQN using PyTorch and find good policies for discrete action space environments. We will test our agent on the famous *Cart Pole* environment. Next exercise, we will build a more sophisticated version of DQN and apply it to a more interesting 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 *Cart Pole* 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_frame`, 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)`.
   - 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.

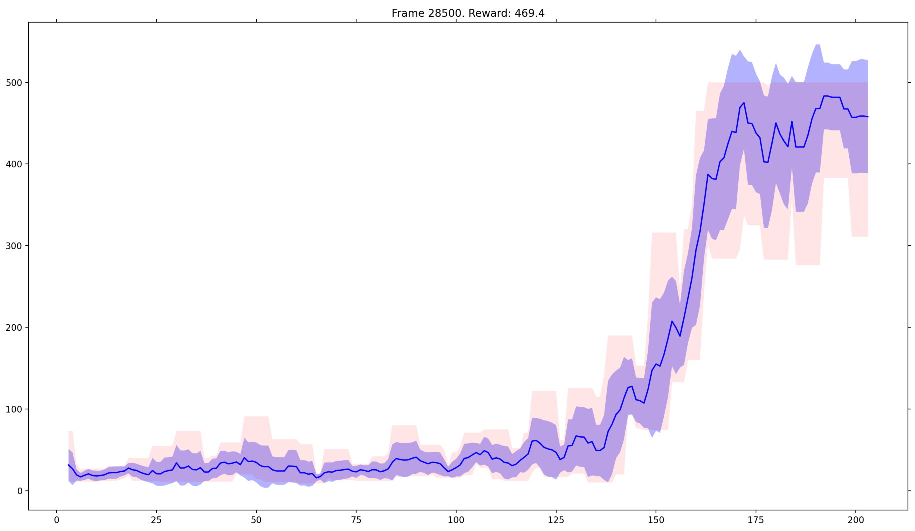

Figure 1: Sample reward plot we were able to achieve with our implementation on CartPole (hyperparameters as provided in skeleton, Linear(128) → ReLU → Linear NN architecture)

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.