Online and Reinforcement Learning (2025)
Home Assignment 5

Davide Marchi 777881

Contents

1 Deep Q-Learning

1.1 Neural architecture

1.1.1 Structure

The line

```
x = torch.cat((x_input, x), dim=1)
```

takes the original input `x_input` (the raw state features) and concatenates it with the hidden representation `x` along the feature dimension. This effectively merges the initial state features with the learned features from the hidden layers. Such a design is sometimes referred to as a *skip connection* or *residual-like* connection, because the network has direct access to the original input when producing the final Q-values.

1.1.2 Activation function

We use tanh instead of the standard logistic sigmoid function because:

- tanh is zero-centered (ranging from -1 to 1), which often helps with training stability.
- The logistic (sigmoid) function saturates more easily at 0 or 1, leading to slower gradients. In contrast, tanh keeps activations in a range that often accelerates convergence in practice.

1.2 Adding the Q-learning

The missing line to compute the target **y** (right after the comment "`# Compute targets`") is shown below. We also detach the tensor so that the gradient does not flow back through the next-state values:

```
max_elements = torch.max(target_Qs, dim=1)[0].detach()
y = rewards + gamma * max_elements
```

This implements the standard Q-learning target:

$$y_i = r_i + \gamma \max_{a'} Q(s'_i, a').$$

After adding this line, the training converged to policies achieving returns above 200 in many episodes, indicating that the agent successfully learned to land.

Learning curve

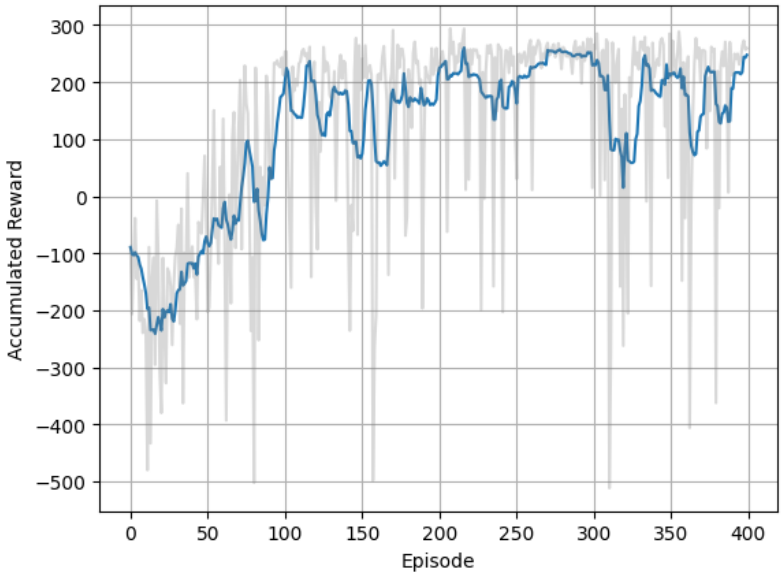

Figure 1: Accumulated reward per episode (in gray) and its smoothed average (blue).

1.3 Epsilon

The line

```
explore_p = explore_stop + (explore_start - explore_stop)*np.exp(-
decay_rate*ep)
```

implements an exponentially decaying exploration probability. At the beginning (when `ep` $= 0$), it starts near `explore_start`, and as the episode index `ep` grows, the probability `explore_p` exponentially approaches `explore_stop`. This ensures that early in training the agent explores more, and then it gradually exploits its learned policy more often.

1.4 Gather

The line

```
Q_tensor = torch.gather(output_tensor, 1, actions_tensor.unsqueeze(-
1)).squeeze()
```

selects the Q-value corresponding to the action actually taken in each state. Since `output_tensor` contains Q-values for all possible actions, we use `gather` along the action dimension to pick out the one Q-value that corresponds to the `actions_tensor`. In other words, it is a convenient way to index a batch of Q-value vectors by their chosen actions.

1.5 Target network

Code modifications

Below is the code I introduced to maintain a target network and perform Polyak averaging. First, I created `targetQN` right after creating `mainQN` and copied the parameters:

```
# Create the target network
targetQN = QNetwork(hidden_size=hidden_size)

# Copy parameters from mainQN to targetQN so they start identical
targetQN.load_state_dict(mainQN.state_dict())
```

Then, at the end of each training step (i.e., right after `optimizer.step()`), I inserted the Polyak update:

```
with torch.no_grad():
    for target_param, main_param in zip(targetQN.parameters(), mainQN.
    parameters()):
        target_param.data.copy_(tau * main_param.data + (1.0 - tau) *
    target_param.data)
```

This implements

$$\theta_{\text{target}} \leftarrow \tau\,\theta_{\text{main}} \;+\; (1-\tau)\,\theta_{\text{target}},$$

where $\tau \in (0, 1)$ is the blend factor.

The changes corrctly introduce a delayed target Q-network. We copy the main network's parameters once at initialization, then repeatedly blend them after every gradient update. This ensures that the target network changes slowly, providing more stable target values in the Q-learning loss.

Comparison with and without the target network

I kept the same hyperparameters as before and introduced the target network. The learning curve is shown below.

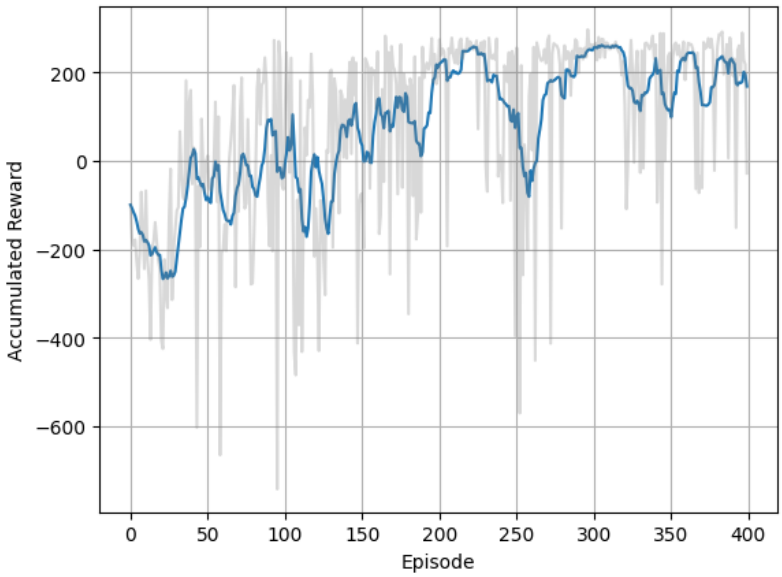

Figure 2: Accumulated reward per episode (in gray) and its smoothed average (blue) using a target network.

Empirically, we can see that the performance still reaches high returns, and in many cases the learning appears somewhat more stable. Although there can be noise and variability in individual runs, a target network typically reduces training instabilities and leads to more consistent convergence.

2 A tighter analysis of the Hedge algorithm

3 Empirical evaluation of algorithms for adversarial environments

4 Empirical comparison of UCB1 and EXP3 algorithms