dqn copy

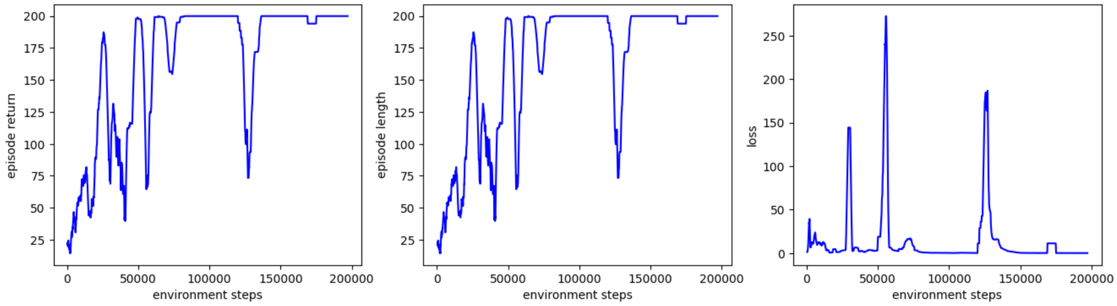
December 7, 2023

0.1 A2.1a) Run the given online Q-learning algorithm

Go through the implementation in the given Jupyter Notebook. Run online Q-learning, that is, use the `QLearningExperiment` with the `QLearner` class on the `CartPole-v1` environment for 200k steps in the environment.

[ ]:

```
```



0.2 A2.1b) Use a replay buffer in Q-learning

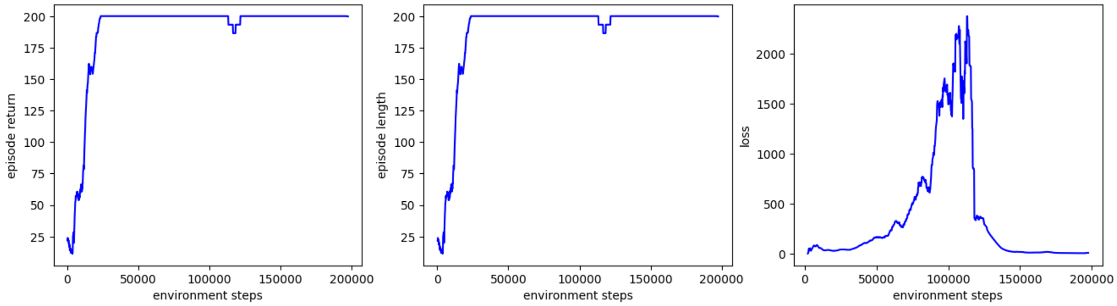
Implement online Q-learning with an experience replay buffer by extending the given skeleton of the `DQNExperiment` class. Train your implementation again in the `CartPole-v1` environment for 200k steps.

[ ]:

```python
class DQNExperiment(QLearningExperiment):
    """Experiment that perfoms DQN. You can provide your own learner."""

    def __init__(self, params, model, learner=None, **kwargs):
        super().__init__(params, model, learner=learner, **kwargs)
        self.use_last_episode = params.get("use_last_episode", True)
        self.replay_buffer = TransitionBatch(
            params.get("replay_buffer_size", int(1e5)),
            self.runner.transition_format(),
            batch_size=params.get("batch_size", 1024),
        )
    def _learn_from_episode(self, episode):
        total_loss = 0
        self.replay_buffer.add(episode["buffer"])
        if len(self.replay_buffer) < self.replay_buffer.batch_size:
            return None
        sample = self.replay_buffer.sample()
        for i in range(self.grad_repeats):
            total_loss += self.learner.train(sample)
        return total_loss / self.grad_repeats
```

[ ]:



0.3 A2.1c) Implement target networks with hard updates

Extend the `QLearning` class with target-networks that use a hard update rule. Train your implementation again in the `CartPole-v1` environment for 200k steps.

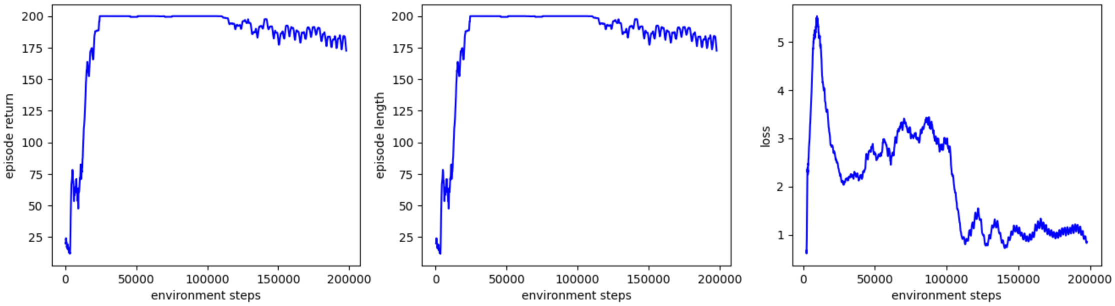
[ ]:

```
class QLearnerHardTarget(QLearner):
    def __init__(self, model, params={}):
        super().__init__(model, params)
        self.target_update = params.get("target_update", "hard")
        self.target_update_interval = params.get("target_update_interval", 200)
        self.target_update_calls = 0
        if params.get("target_model", True):
            self.target_model = deepcopy(model)
            for p in self.target_model.parameters():
                p.requires_grad = False
        assert (
            self.target_model is None
            or self.target_update == "soft"
            or self.target_update == "copy"
        ), 'If a target model is specified, it needs to be updated using the␣
 ↪"soft" or "copy" options.'

    def q_values(self, states, target=False):
        if target == True and self.target_model is not None:
            return self.target_model(states)
        else:
            return self.model(states)

    def target_model_update(self):
        if self.target_model is None:
            pass
        self.target_update_calls += 1
        if self.target_update_calls % self.target_update_interval == 0:
            self.target_model.load_state_dict(self.model.state_dict())
```

[ ]:



0.4 A2.1d) Implement target networks with soft updates

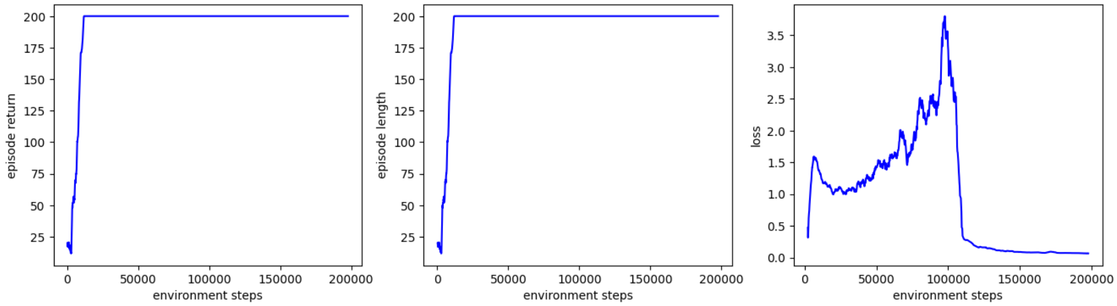
Extend your implementation with a soft-update rule for the target network and test it in the environment `CartPole-v1` for $200k$ steps.

[ ]:

```
class QLearnerSoftTarget(QLearnerHardTarget):
    def __init__(self, model, params={}):
        super().__init__(model, params)
        self.target_update = params.get("target_update", "soft")
        self.soft_target_update_param = params.get("soft_target_update_param",␣
 ↪0.1)

    def target_model_update(self):
        if self.target_model is None:
            pass
        self.target_update_calls += 1
        for t_param, param in zip(
            self.target_model.parameters(), self.model.parameters()
        ):
            t_param.data.copy_(
                (1.0 - self.soft_target_update_param) * t_param.data
                + self.soft_target_update_param * param.data
            )
```

[ ]:



0.5 A2.1e) Implement double Q-learning

Extend your implementation with double-Q-learning and test it in the `CartPole-v1` environment for 200k steps.

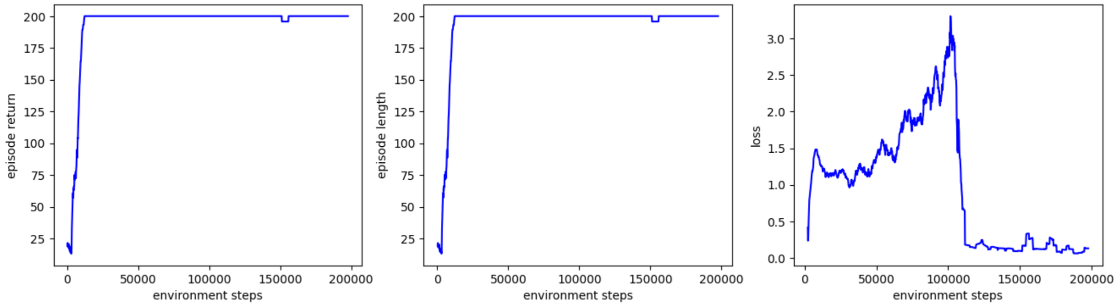
[ ]:

```
class DoubleQLearner(QLearnerSoftTarget):
    def __init__(self, model, params={}):
        super().__init__(model, params)
        self.double_q = params.get("double_q", True)

    def _next_state_values(self, batch):
        with th.no_grad():
            qvalues = self.q_values(batch["next_states"], target=False)
            _, actions = qvalues.max(dim=-1, keepdim=True)
            eval_qvalues = self.q_values(batch["next_states"], target=True)

            return eval_qvalues.gather(dim=-1, index=actions)
```

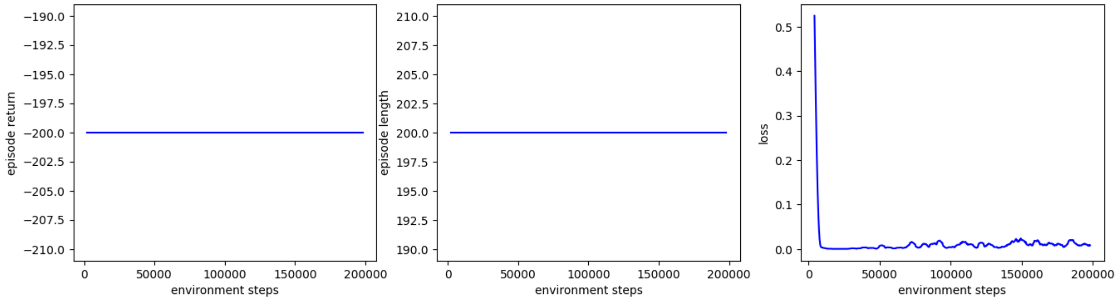
[ ]:



0.6 A2.1f) Run double Q-learning on MountainCar

Run your implementation of `DoubleQLearner` on the `MountainCar-v0` environment for 200k steps. In all likelihood, your agent will not be able to solve the task (reach the goal), and learning should not pick up at all. Explain why that is.

[ ]:

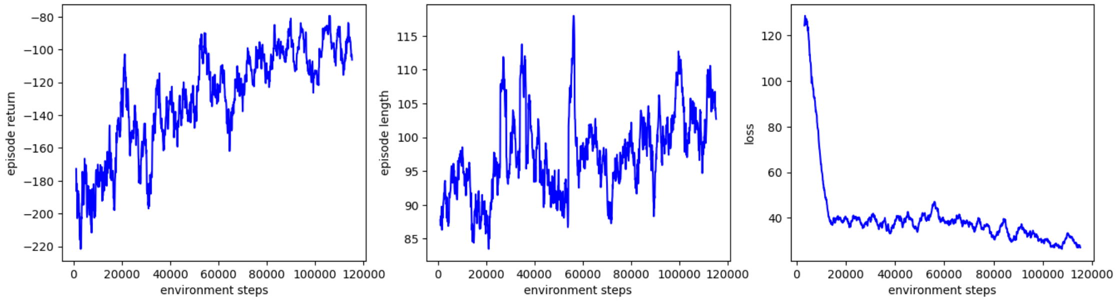


The only positive reward in MountainClimber comes from reaching the right top of the hill, which requires a coordinated movement. The chances of a random agent reaching it are very low, so there is no prior experience the model could learn from. A possible solution would be to increase the initial exploration phase, i.e. increase the lower size limit of the replay buffer.

0.7 A2.1g) Run double Q-learning on LunarLander

Run your implementation on the `LunarLander-v2` environment for at least 5 million environment steps (more is better). This can take a while, expect 1-3 hours of computation time. Do you get similar results as shown in the lecture?

[ ]:



```
The Kernel crashed while executing code in the the current cell or a previous␣
 ↪cell. Please review the code in the cell(s) to identify a possible cause of␣
 ↪the failure. Click <a href='https://aka.ms/vscodeJupyterKernelCrash'>here</a>␣
 ↪for more info. View Jupyter <a href='command:jupyter.viewOutput'>log</a> for␣
 ↪further details.
```

Because of the memory leak in the Gym environment, the simulation crashed around 120k steps in.