Exercise 6

Tommaso Brumani - 100481325
ELEC-E8125 - Reinforcement Learning

November 6, 2022

Task 1

The actor-critic algorithm was implemented for the Inverted Pendulum environment by modifying the **pg_ac.py** file as follows:

- In the **Policy.__init()__** function, the following line was added:

```
# Implement actor_logstd as a learnable parameter
# Use log of std to make sure std doesn't become negative during training
self.actor_logstd = nn.parameter.Parameter(torch.tensor(np.zeros(action_dim), device=device))
```

- In the **Policy.forward()** function, the following line was added:

```
#  Create a Normal distribution with mean of 'action_mean' and standard deviation of 'action_logstd', and return the distribution
probs = Normal(action_mean, action_std)
```

- In the **PG.update()** function, the following lines were added:

```
########## Your code starts here. ##########

# calculate values for current and next states
states_values = self.value.forward(states)
next_states_values = self.value.forward(next_states)

# set next state values to zero when terminal
dones_mask = dones == False
next_states_values *= dones_mask

# calculate TD target
td_target = rewards + self.gamma * next_states_values

# calculate critic loss
critic_loss = torch.mean(torch.square(states_values -
   td_target.detach()))

# calculate advantage
adv = td_target - states_values

# normalize advantage
(std, mean) = torch.std_mean(adv)
norm_adv = (adv - mean)/std

# compute policy loss for batch
actor_loss = - torch.mean(action_probs.squeeze(-1) *
   norm_adv.detach())

# compute combined loss
comb_loss = actor_loss + critic_loss

# backpropagate gradients
comb_loss.backward()
self.optimizer.step()

# empty optimizer gradients
self.optimizer.zero_grad()

########## Your code ends here. ##########
```

- In the **PG.get_action()** function, the following lines were added:

```
########## Your code starts here. ##########

# get action distribution following policy for give
    observation
action_distribution = self.policy.forward(x)

# select action
if evaluation:
    # mean if testing
    action = action_distribution.mean
else:
    # random sample otherwise
    action = action_distribution.sample()

# get log probability of chosen action
act_logprob = action_distribution.log_prob(action)

########## Your code ends here. ###########
```

As a result, the performance plot reported in Figure 1 was obtained.

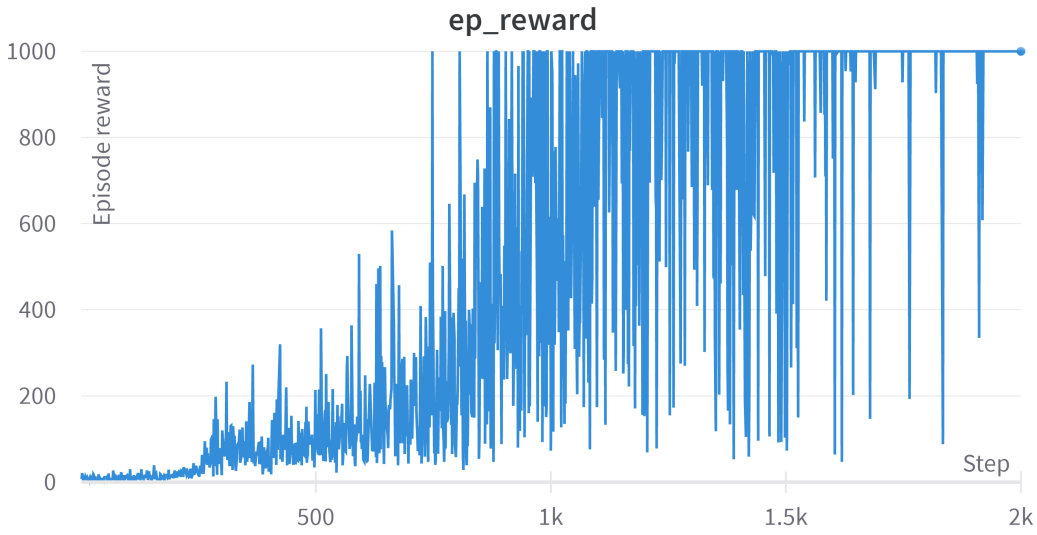

Figure 1: Performance of the PG actor-critic model.

Question 1.1

In REINFORCE with baseline, a good choice for the baseline was an estimation of the state value function $V(s_t)$, which could be learned from the data. In actor-critic methods this becomes the critic model, which is used to determine the TD target.

Question 1.2

The value of Advantage can be intuitively interpreted as the amount by which having chosen action a increases the cumulative return, with respect to the average obtainable from the current state.
In other words, it indicates how optimal the chosen action is with respect to the others.

Question 1.3

Since REINFORCE is a MC approach and the implemented actor-critic model is a TD approach, we can expect the actor-critic to be characterized by an increased bias and reduced variance with respect to the REINFORCE algorithm because of bootstrapping.

Question 1.4

Implementing a form of TD-λ could be used to control the bias/variance tradeoff by setting using values of λ ($\lambda = 0$ corresponding to the current actor-critic approach, and $\lambda = 1$ approximating the MC approach).

Task 2

The deep deterministic policy gradient algorithm was implemented for the Half Cheetah environment by modifying the **ddpg.py** file as follows:

- In the **DDPG.update()** function, the following lines were added:

```
########## Your code starts here. ##########

# compute Q target
a_next = self.pi_target(batch.next_state)
q_next =  self.q_target(batch.next_state, a_next)
q_tar = batch.reward + self.gamma * q_next * batch.not_done

# compute critic loss
q_policy = self.q(batch.state, batch.action)
critic_loss = torch.mean(torch.square(q_tar.detach() -
   q_policy))

# update critic's parameter
critic_loss.backward()
self.q_optim.step()

# compute actor loss
a_policy_new = self.pi(batch.state)
q_policy_new = self.q(batch.state, a_policy_new)
actor_loss = - torch.mean(q_policy_new)

# update actor's parameters
actor_loss.backward()
self.pi_optim.step()

# update target networks
h.soft_update_params(self.q, self.q_target, self.tau)
h.soft_update_params(self.pi, self.pi_target, self.tau)

# empty optimizer gradients
self.q_optim.zero_grad()
self.pi_optim.zero_grad()

########## Your code ends here. ##########
```

- In the **DDPG.get_action()** function, the following lines were added:

```
########## Your code starts here. ##########

# caculate action to execute
action = self.pi(x)

# if not evaluation, add gaussain noise
if not evaluation:
    action += expl_noise * torch.randn(action.shape, device
       =device)

# get correct action shape
action = action.squeeze(0)

########## Your code ends here. ##########
```

As a result, the performance plot reported in Figure 2 was obtained.

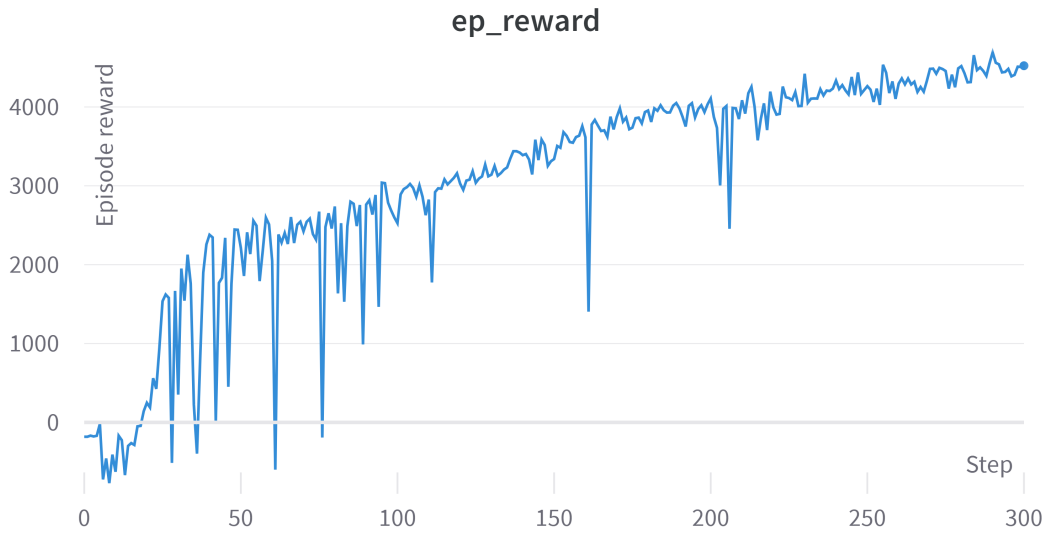

Figure 2: Performance of the DDPG model.

Question 2.1

In DDPG we can use off-policy data because instead of using a whole episode's discounted reward we are able to use the data from single timesteps thanks to bootstrapping.
This is made possible by the fact that we use the data sampled from the memory buffer to train the policy network, which in turn we use to determine the estimated optimal Q when calculating the loss functions (similarly to how in Q-learning we used a maximum over the different actions).

Question 2.2

Compared to the actor-critic method of Task 1, which uses the actor to output the mean and variance of a gaussian distribution from which actions are sampled, the DDPG algorithm's actor outputs directly a deterministic value.
While this reduces computational burden in those cases where actions are deterministic, it fails to capture those instances where actions are stochastic, and thus has reduced expressive power.
This also means that the exploration during training is dictated by previously set hyperparameters of the noise distribution, rather than on a probability distribution whose parameters are learned and updated at runtime based on data.