Reinforcement Learning
Continuous Control - Tennis Project

[17.09.2021]

This file explains the solution to the Tennis environment found in Tennis.ipynb of the current repository.
The README.md file explains the environment and how to run the solution.

General approach

The problem features 2 agents playing tennis together, with the goal of collaborating and maximizing a common score based more or less on how long the game was.
This can be modeled as a Markov Decision Process.

We use a policy-based method to directly learn a policy, i.e. a function that outputs an action given an input state.
More precisely, we use an agent-critic method, whereby the training of an agent is backed by the training of a critic that outputs a best estimate for the total discounted rewards given an input state.
In the current context, we have multiple agents, and the critic takes as input all agent observations and all agent actions, while each agent takes as input only its own "local" observation.
More specifically, we use the (Deep Determinisitc Policy Gradient) (DDPG) algorithm, which we only slightly adapt so that, as described above, the (single!) critic takes as input the states and actions of all agents.

Networks

<u>Agent</u>

Note the agent takes as input a 24-vector, even though the state has only 8 elements. That's because the environment outputs a series of 3 consecutive observations.

```
State(24) -> Linear(24, 300) -> ReLU -> Linear(300, 300) -> ReLU
-> Linear(300, 2) -> Tanh
```

The final activation function is Tanh, to squeeze the output into the [-1, 1] action domain space.

Critic

```
States(48) -> Linear(48, 100) -\
Actions(4) ---------------------> Linear(104, 300) -> ReLU ->
Linear(300, 200) -> ReLU -> Linear(200, 1)
```

Learning algorithm

We maintain a target and online version of the Agent and the Critic networks.
The trained networks are the “online” ones, while targets are soft-copied from online networks after each iteration. “Soft-copied” means that the network weights are not copied 1-to-1, but instead slightly converge towards the ones of the source network:

```
def soft_update(self, local_model, target_model, tau):
    for target_param, local_param in zip(target_model.parameters(), local_model.parameters()):
        target_param.data.copy_(tau*local_param.data + (1.0-tau)*target_param.data)
```

As written in introduction, we basically use the DDPG algorithm, slightly modified so that the critic takes as input the states and actions of all agents.
Each step of each episode is added to an “experience buffer”.
After each step, a random batch of experiences is collected, and training is based on those.
The training of the critic is based on a minimization of distance between the Q-value (i.e. discounted total rewards) output by the current online network, and the target value (considered to be the current “best estimate”) provided by the target critic network:

```
# Train centralized critic
next_actions = torch.stack([self.actor_target(next_state) for next_state in next_states_t]).transpose(0, 1)
Q_target_next = self.critic_target(next_states, next_actions).detach()
with torch.no_grad():
    Q_target = torch.hstack([rewards[:, i].unsqueeze(1) + (gamma * Q_target_next * (1 - dones[:, i].unsquee
Q_online = self.critic_online(states, actions)

assert(Q_online.shape == (batch_size, 1))
assert(Q_target.shape == (batch_size, self.nb_agents))
self.optimizer_critic.zero_grad()
loss_critic = F.smooth_l1_loss(Q_online.expand(batch_size, self.nb_agents), Q_target)
loss_critic.backward()
self.optimizer_critic.step()
```

Then, the (single!) actor is trained by maximizing the expected discounted rewards given the action taken by the online actor, using the online critic to estimate the discounted rewards:

```
loss_actor = torch.Tensor([0])
# Train actor
self.optimizer_actor.zero_grad()
predicted_actions = torch.stack([self.actor_online(state) for state in states_t]).transpose(0, 1)
loss_actor = -self.critic_online(states, predicted_actions).mean()
loss_actor.backward()
self.optimizer_actor.step()
```

Hyperparameters

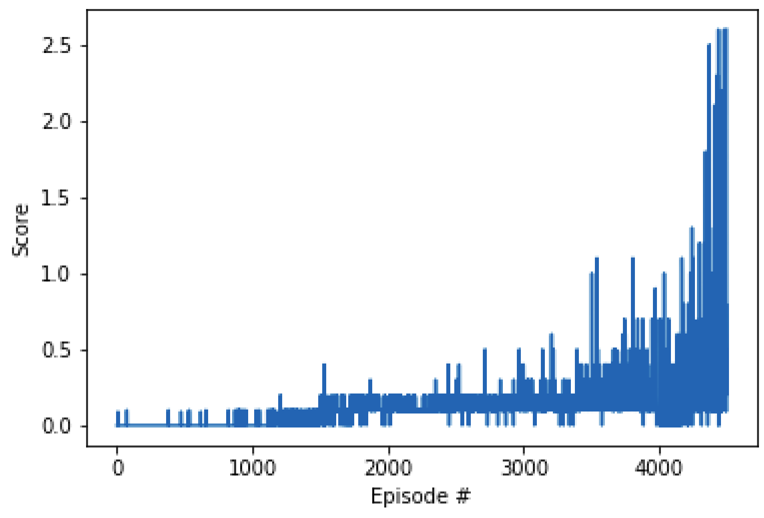
Name	Value	Comment
TAU	0.01	Rate of update of the target network weights. This is common value for TAU.
LR_ACTOR	5e-5	Learning rate for the update of weights of the Actor network during the gradient descent. This value has been decreased compared to the original value, observing too much learning instability.
LR_CRITIC	5e-5	Learning rate for the update of weights of the Critic network during the gradient descent. This value has been decreased compared to the original value, observing too much learning instability.
BUFFER_SIZE	100000	Size of the experience buffer from which to collect a random set for the Stochastic Gradient Descent. It has to be large enough to encompass older experiences, since newer experiences kick out older ones once the number of elements reaches BUFFER_SIZE.
gamma	0.99	Discount applied to a reward for 1 time step. Here, it makes sense to not have 1, i.e. give less important to future rewards, as in tennis, the consequence of an action is to some extent limited in time, and more recent ones explain more the immediate reward.
noise decay factor	0.999	Amplitude of the random noise applied to an action output by the online actor during action (i.e. not

		training). This has to be large enough at the beginning of training to allow for exploration, and decrease over time to help converging and hence “exploit” the learning (“exploration vs exploitation”)

Results

Below is a plot of rewards obtained on the successful run:

```
fig = plt.figure()
ax = fig.add_subplot(111)
plt.plot(np.arange(1, len(all_scores)+1), all_scores)
plt.ylabel('Score')
plt.xlabel('Episode #')
plt.show()
```



As for the benchmark implementation, we observe a sudden increase in scores after a quite long time of slow learning.
It took **4397 episodes** to reach an average score over 100 episodes beyond 0.5.

Improvements

A first obvious improvement is hyperparameter tuning. It was observed that slightly changing some of them had dramatic impacts on the outcome. For this project, hyperparameter tuning was done purely manually and based on experience. Tools can help automate this tuning, which we think would enable a significantly faster converging of the algorhtm.

Otherwise, a more advanced algorithm should be used. Here, we almost misuse DDPG for a multi-agent training, while DDPG is only “guaranteed” to converge if the environment always behaves the same. But here, each agent observes an environment that includes the other agent, which obviously makes the learning way more unstable. This can be understood intuitively: if an agent learns that pushing on button A turns on a light, but actually another agent controls this behavior, and at some point this other agent does not turn on the light anymore upon pushing on button A, then whatever the other agent learned is not valid anymore. In this lab, we are probably lucky there is only 1 other agent and that the behavior of agents does not change dramatically, so at the end we still manage to converge. Also, the critic learns based on the full agents states and actions, which limits the consequences of the non-stationarity of the environment.