Continuous Control Project - Report

Introduction

We train an agent to control a double-jointed arm and make it point in a given direction.
Our agent will interact with a Unity Reacher environment:
https://github.com/Unity-Technologies/ml-agents/blob/main/docs/Learning-Environment-Examples.md#reacher

The problem is formalized as a Markov Decision Process (MDP).
Time is discretized, and at each time step, the agent makes a decision in order to maximize the expected total discounted rewards.

We use neural networks as universal function approximators in order to map an environment state to an action.
The implementation is an adaptation of DDPG, which is considered as an Actor-Critic method.
A neural network implements the Actor, which maps a state to an Action, while another neural network implements the Critic, and evaluates the expected reward given a (state, action) tuple (as we have a continuous action space).

Model

Below is the implementation of the policy network:

```
# Define the Policy network
# It will take as input a vector of 33 elements, and output an action vector, which we normalize with tanh
# to clip vector elements between -1 and 1, as expected by the environment, which makes sense, as otherwise we
# would need to know about the concrete "physical" values behind each action value, but the environment acts as
# a black box for us.
class Policy(nn.Module):
    def __init__(self):
        super(Policy, self).__init__()
        self.hidden1 = nn.Sequential(
            nn.Linear(33, 400),
            #nn.BatchNorm1d(400),
            nn.ReLU()
        )
        self.hidden2 = nn.Sequential(
            nn.Linear(400, 300),
            #nn.BatchNorm1d(300),
            nn.ReLU()
        )
        self.output = nn.Sequential(
            nn.Linear(300, 4),
            nn.Tanh()
        )

    def forward(self, x):
        x = self.hidden1(x)
        x = self.hidden2(x)
        return self.output(x)
```

..

It has 2 hidden and 1 output layers.
The activation function (a.k.a. “non-linearity”) for hidden layers is ReLU, which is probably the most commonly used activation function. It does not squeeze values and hence enables significant gradients, avoiding the vanishing gradient problem.
The output is normalized with Tanh, in order to squeeze values within range accepted by the environment, for actions.
The number of neurons was chosen intuitively, from previous experiences on similar problems.

The Critic also has a quite common architecture:

```
# Define the Critic network
# Here, we have a continuous action space, so the critic cannot output Q-value estimates given a state,
# and instead has to take both state and action as input, and output a scalar Q-value estimate.
class Critic(nn.Module):
    def __init__(self):
        super(Critic, self).__init__()
        self.hidden1s = nn.Sequential(
            nn.Linear(33, 100),
            #nn.BatchNorm1d(100),
            nn.ReLU()
        )
        self.hidden1 = nn.Sequential(
            nn.Linear(100+4, 300),
            #nn.BatchNorm1d(300),
            nn.ReLU()
        )
        self.hidden2 = nn.Sequential(
            nn.Linear(300, 100),
            #nn.BatchNorm1d(100),
            nn.ReLU()
        )
        self.output = nn.Sequential(
            nn.Linear(100, 1)
        )

    def forward(self, state, action):
        xs = self.hidden1s(state.float())
        x = self.hidden1(torch.cat((xs, action), dim=1))
        x = self.hidden2(x)
        #x = self.hidden1s(state.float())
        #x = self.hidden1(torch.cat((x, action), dim=1))
        return self.output(x)
```

In a first implementation, the input vector was a concatenation of state and action vectors, but this did not lead into any significant learning.
The idea of first letting the state go through a first hidden layer was taken from the benchmark implementation of a similar problem:
https://github.com/udacity/deep-reinforcement-learning/blob/master/ddpg-pendulum/model.py
Note the output is not transformed, as it is an estimate of rewards, as opposed to, for instance, probabilities, for which we would need to transform the output into probabilities using softmax.

Algorithm

- while episodes are running, collect each experience tuple (state, action, next state, reward)
- every N steps, learn:
    - collect M random experience tuples
    - train the Critic using DQN:
        - update network models through gradient descent on a loss being the mean-squared distance between the current Q values (from the online critic network) and the target Q values (from the target critic network, being a soft copy of the online network from N steps before)
        - do a soft update of the target network, by moving parameters towards the ones of the online network, as opposed to doing a sheer copy

    - train the Actor (Policy):
        - compute predicted actions, for the states of the collected experience tuples, using the online policy network
        - inject those actions into the online critic network, to obtain an average of estimated rewards, given by the critic network. Since we want to maximize this average, we compute the gradient of the opposite of this average. The policy network model is updated based on this gradient. Note an advantage of using an Actor-Critic method is to get a low-bias, low-variance gradient estimate to stabilize learning and have better chances of it converging
        - note that as for the Critic, we also maintain a target network for the Actor. We also use the target network to compute the target values when training the Critic network. The Actor target network also undergoes soft updates from its peer online network every N steps.

- after the end of each episode, calculate the average score over the last 100 episodes. If it is more than 30, the problem is solved. Models are saved in a checkpoint file for future model instantiation.

- a decay factor is applied to the random noise added to the action output by the online actor network, so that there is more exploration at the beginning of training, and less over time

Hyperparameters

```
UPDATE_EVERY = 20
NB_UPDATES = 10
LR_ACTOR = 1e-4
LR_CRITIC = 1e-3
TAU = 1e-2 # for soft update of target parameters
BUFFER_SIZE = 10000
```

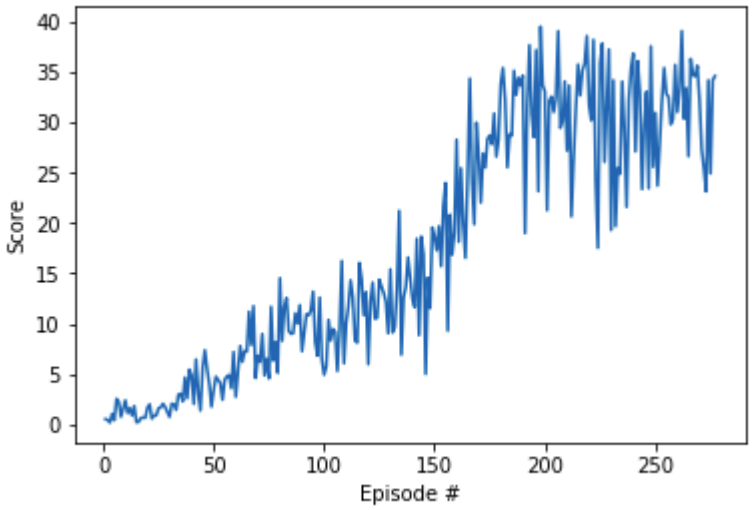
- UPDATE_EVERY: 20 experience tuples are collected before performing gradient descent to update the models, and this update is done NB_UPDATE times
- LR_ACTOR, LR_CRITIC: these are the learning rates applied on the model parameter updates, respectively for the online actor network, and the online critic network. Values in this range are very common and constitute a good balance between slow but stable learning, and faster but unstable or diverging learning.
- TAU: multiplicative factor used for the soft update of target network parameters
- BUFFER_SIZE: number of experience tuples from which tuples are sampled randomly for stochastic gradient descent. Experience tuples over all episodes accumulate in this buffer, so older experience tuples get discarded. In the present case, the buffer size can fit 10 full episodes. Retrospectively, the buffer size could have been increased to, say, 50000, as 10 episodes to learn from does not seem a lot.

Result

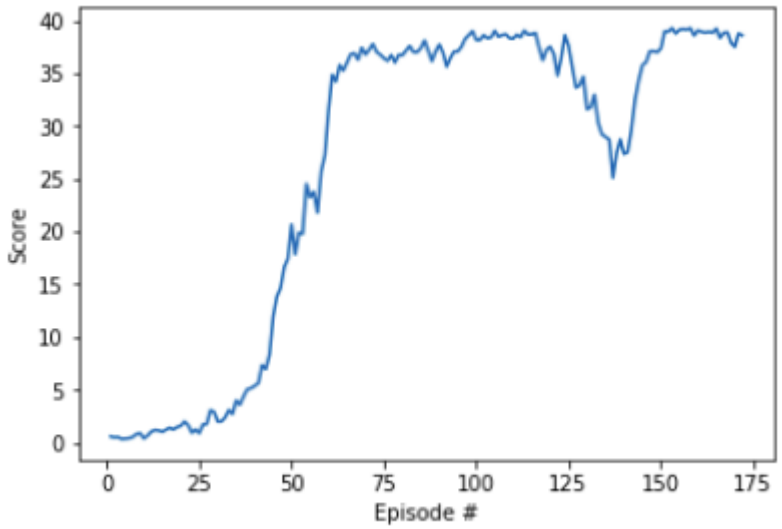
The problem is solved after 260 episodes:

```
Episode  259  - Score:  31.01999930664897
Episode  260  - Score:  33.249999256804585
Episode  260  - Mean score:  30.0326993287
Problem solved!
```

It means the average score of episodes 161 to 260 was beyond 30.



We notice the score was not as stable as expected, compared to the scores obtained in the reference implementation:



On the chart above, we observe that despite a temporary drop in scores, once the network converged beyond 30 around episode 75, it always remained around 35. Our implementation would drop from time to time below 30.

We also observe that convergence was way slower.
There was an attempt at applying batch normalization, which actually hampered training.

Discussion

- A significant amount of time was spent tweaking hyperparameters: maximum number of steps per episode, size of hidden layers, etc. The architecture of network was also modified several times, and the most important change was to inject the state into a dedicated hidden layer, apply ReLU, and concatenate this result with the input action into further layers. There was an attempt at having as critic network input a concatenation of state and action vectors, and the same but with a normalized state vector for it to be in the same range as the action vector, between -1 and 1. None of those alternatives allowed to train.

- Gradient clipping was not applied, while it had to be used in the benchmark implementation. Here, once the network converged, despite unstable scores, it did not diverge, which we could observe as we let training happen even after reaching an average of 30, so there was not felt need of adding gradient clipping.

- There is room for improvement: converge faster, and keep scores more stable once the network converges. A decay factor was applied to the noise applied to the online actor network, as an attempt to reach more stability. Very probably, further fine tuning would enable improving those two points, as it is known, also from personal experience, that even slight modifications in hyperparameters can have dramatic impacts on the outcome.