Continuous control udacity deep RL project report

Overview:

This report illustrates the conducted work to pass the continuous control udacity deep reinforcement learning project. We will present the implementation, results, and future work.

Link to the project repository: [github link]

1- Implementation details:

The project has the following structure:

- **.Isort.cfg:** config file that specifies some code setting (line_length, packages,...).
- **.pre-commit-config.yaml:** config file to specify the hooks used when running `pre-commit`.
- **core.py:** Serves as a wrapper to encapsulate the Reacher unity environment.
- **environment.yml:** contains the dependencies and packages to be able to run the code.
- **train.py:** Train the agent.
- **ddpg_agent**: ddpg implementation inspired from Udacity deep RL repository.
- **model.py**: contains the actor and critic architecture.
- **checkpoints**: directory where to store the checkpoints.
- **plots:** directory where to store the plots.
- **report.pdf**: A summary of the conducted work.

2- Learning algorithm:

In this project we used ***ddpg*** algorithm to train our agent.

Following is the pseudo-code of the algorithm that we want to implement:

Algorithm 1 DDPG algorithm

Randomly initialize critic network $Q(s, a|\theta^Q)$ and actor $\mu(s|\theta^\mu)$ with weights θ^Q and θ^μ.
Initialize target network Q' and μ' with weights $\theta^{Q'} \leftarrow \theta^Q$, $\theta^{\mu'} \leftarrow \theta^\mu$
Initialize replay buffer R
for episode = 1, M **do**
  Initialize a random process $\mathcal{N}$ for action exploration
  Receive initial observation state s_1
  for t = 1, T **do**
    Select action $a_t = \mu(s_t|\theta^\mu) + \mathcal{N}_t$ according to the current policy and exploration noise
    Execute action a_t and observe reward r_t and observe new state s_{t+1}
    Store transition (s_t, a_t, r_t, s_{t+1}) in R
    Sample a random minibatch of N transitions (s_i, a_i, r_i, s_{i+1}) from R
    Set $y_i = r_i + \gamma Q'(s_{i+1}, \mu'(s_{i+1}|\theta^{\mu'})|\theta^{Q'})$
    Update critic by minimizing the loss: $L = \frac{1}{N}\sum_i (y_i - Q(s_i, a_i|\theta^Q))^2$
    Update the actor policy using the sampled policy gradient:

$$\nabla_{\theta^\mu} J \approx \frac{1}{N} \sum_i \nabla_a Q(s, a|\theta^Q)|_{s=s_i, a=\mu(s_i)} \nabla_{\theta^\mu} \mu(s|\theta^\mu)|_{s_i}$$

    Update the target networks:

$$\theta^{Q'} \leftarrow \tau\theta^Q + (1-\tau)\theta^{Q'}$$
$$\theta^{\mu'} \leftarrow \tau\theta^\mu + (1-\tau)\theta^{\mu'}$$

  end for
end for

Taken from “Continuous Control With Deep Reinforcement Learning” (Lillicrap et al, 2015)

Actor architecture:

- 3 fully connected layers with batch normalisation on the first layer output.
- The first layers are followed by the ***Relu*** activation function and the last one with ***tanh***.

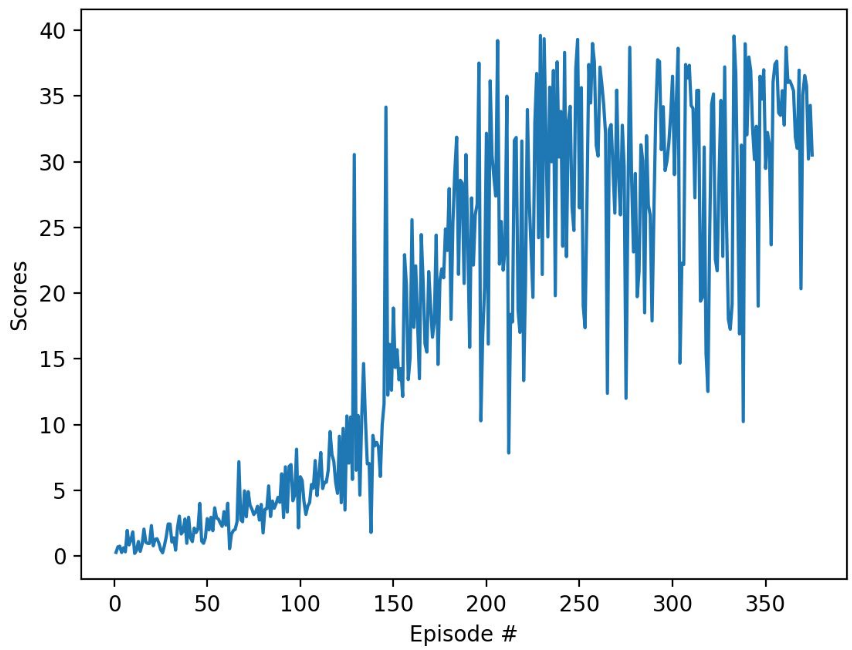
Critic architecture:

- 3 fully connected layers with batch normalisation on the first layer output.
- The first two layer are followed by ***Relu*** activation function

Features added to be able to learn properly:

- Add the batch normalisation as presented in the architecture above.
- Change the horizon value to 10000.
- Tweak the Sigma from the Ornstein-Uhlenbeck process. (**sigma=0.1**)
- Learning rate tuning. (**actor_lr= 1e-4** and **critic_lr=1e-4**)
- Normalise and clipp the gradient.

3- Results:



We were able to solve the environment and reach the score of **30** over 100 episodes after almost **370 episodes.**

4- Future work:

- It would be very interesting to work with the environment version with 20 agents. It is useful for algorithms like PPO, A3C, and D4PG that use multiple (non-interacting, parallel) copies of the same agent to distribute the task of gathering experience.