Report - (Project 2, Continuous-Control)

Learning Algorithm

DDPG is an actor-critic algorithm for solving continuous control problems in reinforcement learning. It is an extension of the deterministic policy gradient algorithm and combines ideas from the Q-learning and actor-critic algorithms.

Here are the summarized basic steps of the DDPG algorithm [1]:

1. Initialize the critic network and actor network with random weights.
2. Sample a mini-batch of experiences from the replay buffer, which is a memory of past experiences.
3. Use the actor network to choose an action for each state in the mini-batch.
4. Calculate the target value for each state-action pair in the mini-batch using the critic network and the Bellman equation. The target value is a measure of how good the action is in that state, taking into account the expected future rewards.
5. Train the critic network to minimize the mean squared error between the predicted target values and the actual rewards.
6. Use the critic network to compute the gradient of the target value with respect to the action, and use that gradient to update the actor network. This helps the actor network learn to choose actions that will result in high target values.
7. Update the weights of the actor and critic networks using stochastic gradient descent or another optimization algorithm.
8. Repeat steps 2-7 for a certain number of iterations or until convergence.

DDPG is effective in solving continuous control problems because it can learn a deterministic policy that directly maps states to actions. This can be more efficient than learning a stochastic policy and sampling actions from it. Additionally, the use of a replay buffer and target network helps stabilize the learning process and prevent the networks from overfitting to the data.

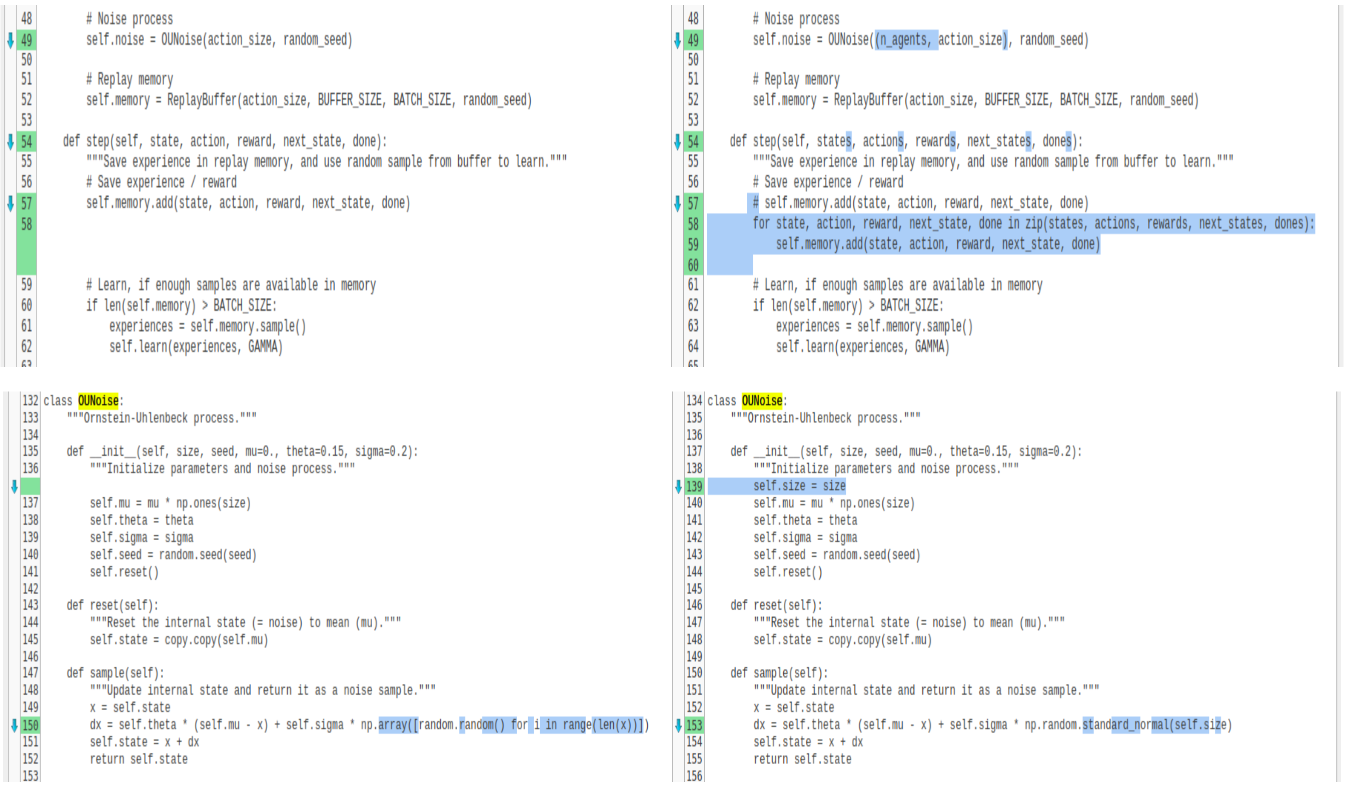
There are two sets of neural networks for both the actor and critic: the local network and the target network. The local networks are used to select actions and make predictions during each time step of training, while the target networks are used to compute the target values for training. Target networks are updated more slowly than the local networks, and this helps stabilize the training process by providing a more stable target for the critic during training. By using a combination of both types of networks, the DDPG algorithm can find an optimal policy for the given task by balancing the need for exploration and learning with the need for stability and accuracy in the training process.

Udacity at GitHub - deep-reinforcement-learning/ddpg-pendulum/ provide a good base framework in solving the continuous control of multiple agents in this project 2. The first thing that I did was amend the DDPG code to work for multiple agents, to solve version 2 of the environment. The original DDPG code in the DRLND GitHub repository utilizes only a single agent, and with each step:

- the agent adds its experience to the replay buffer, and
- the (local) actor and critic networks are updated, using a sample from the replay buffer.

So, in order to make the code work with 20 agents, I modified the code so that after each step:

- each agent adds its experience to a replay buffer that is shared by all agents, and
- the (local) actor and critic networks are updated 20 times in a row (one for each agent), using 20 different samples from the replay buffer.



```
        # Noise process
        self.noise = OUNoise(action_size, random_seed)

        # Replay memory
        self.memory = ReplayBuffer(action_size, BUFFER_SIZE, BATCH_SIZE, random_seed)

    def step(self, state, action, reward, next_state, done):
        """Save experience in replay memory, and use random sample from buffer to learn."""
        # Save experience / reward
        self.memory.add(state, action, reward, next_state, done)



        # Learn, if enough samples are available in memory
        if len(self.memory) > BATCH_SIZE:
            experiences = self.memory.sample()
            self.learn(experiences, GAMMA)
```

```
        # Noise process
        self.noise = OUNoise((n_agents, action_size), random_seed)

        # Replay memory
        self.memory = ReplayBuffer(action_size, BUFFER_SIZE, BATCH_SIZE, random_seed)

    def step(self, states, actions, rewards, next_states, dones):
        """Save experience in replay memory, and use random sample from buffer to learn."""
        # Save experience / reward
        # self.memory.add(state, action, reward, next_state, done)
        for state, action, reward, next_state, done in zip(states, actions, rewards, next_states, dones):
            self.memory.add(state, action, reward, next_state, done)

        # Learn, if enough samples are available in memory
        if len(self.memory) > BATCH_SIZE:
            experiences = self.memory.sample()
            self.learn(experiences, GAMMA)
```

```
class OUNoise:
    """Ornstein-Uhlenbeck process."""

    def __init__(self, size, seed, mu=0., theta=0.15, sigma=0.2):
        """Initialize parameters and noise process."""

        self.mu = mu * np.ones(size)
        self.theta = theta
        self.sigma = sigma
        self.seed = random.seed(seed)
        self.reset()

    def reset(self):
        """Reset the internal state (= noise) to mean (mu)."""
        self.state = copy.copy(self.mu)

    def sample(self):
        """Update internal state and return it as a noise sample."""
        x = self.state
        dx = self.theta * (self.mu - x) + self.sigma * np.array([random.random() for i in range(len(x))])
        self.state = x + dx
        return self.state
```

```
class OUNoise:
    """Ornstein-Uhlenbeck process."""

    def __init__(self, size, seed, mu=0., theta=0.15, sigma=0.2):
        """Initialize parameters and noise process."""
        self.size = size
        self.mu = mu * np.ones(size)
        self.theta = theta
        self.sigma = sigma
        self.seed = random.seed(seed)
        self.reset()

    def reset(self):
        """Reset the internal state (= noise) to mean (mu)."""
        self.state = copy.copy(self.mu)

    def sample(self):
        """Update internal state and return it as a noise sample."""
        x = self.state
        dx = self.theta * (self.mu - x) + self.sigma * np.random.standard_normal(self.size)
        self.state = x + dx
        return self.state
```

Image 1: The modification on the code segment of "ddpg_agent.py"

Model Architecture

The algorithm employs two deep-neural-networks (which consist of 1 actor and 1 critic). The structure is as follow:

Actor model:

Input layer: 33 (state size) neurons

1st hidden layer: 128 neurons, [relu activation function]

2nd hidden layer: 128 neurons, [relu activation function]

Output layer: 4 (action size) neurons, [tanh activation function]

Critic model:

Input layer: 33 (state size) neurons

1st hidden layer: (128 + 4) neurons (where the additional 4 are the action size), [relu activation function]

2nd hidden layer: 128 neurons, [relu activation function]

Output layer: 1 neuron

Hyperparameters Tuning

Setting 1

- num of episodes: 1000
- max timestep: 300
- replay buffer size: 10000
- minibatch size: 128
- gamma, discount factor: 0.99
- tau, soft update for target networks factor: 1e-3
- Learning rate of the actor: 1e-4
- Learning rate of the critic: 1e-3

```
Episode 100     Average Score: 0.65
Episode 200     Average Score: 3.10
Episode 300     Average Score: 5.31
Episode 400     Average Score: 6.50
Episode 500     Average Score: 8.59
Episode 600     Average Score: 8.87
Episode 700     Average Score: 9.21
Episode 800     Average Score: 9.37
Episode 900     Average Score: 9.86
Episode 1000    Average Score: 9.99
```

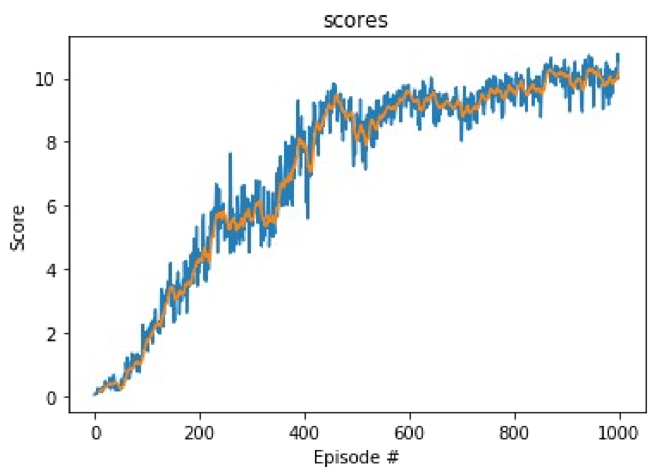


Image 1: The performance result of agents with setting 1

Setting 2

- num of episodes: 1000
- max timestep: 300
- replay buffer size: 10000
- minibatch size: 128
- gamma, discount factor: 0.99
- tau, soft update for target networks factor: 1e-3
- Learning rate of the actor: 1e-2
- Learning rate of the critic: 1e-3

Result:

Episode 100 Average Score: 0.01

Episode 200 Average Score: 0.01

It was found that large learning rate of the actor causes the agents cannot learn and progress at all!

Setting 3

- num of episodes: 1000
- max timestep: 1000
- replay buffer size: 10000
- minibatch size: 128
- gamma, discount factor: 0.99
- tau, soft update for target networks factor: 1e-3
- Learning rate of the actor: 1e-3
- Learning rate of the critic: 1e-3

For multi-agents environment, when the time-steps given are small, the environment cannot experience with rich different combination set of states and actions of separate multiple agents, hence the only little amount of information are learnt before the end of 1 episode. To solve this, "max_t" is set to be a larger value to improve the experience.

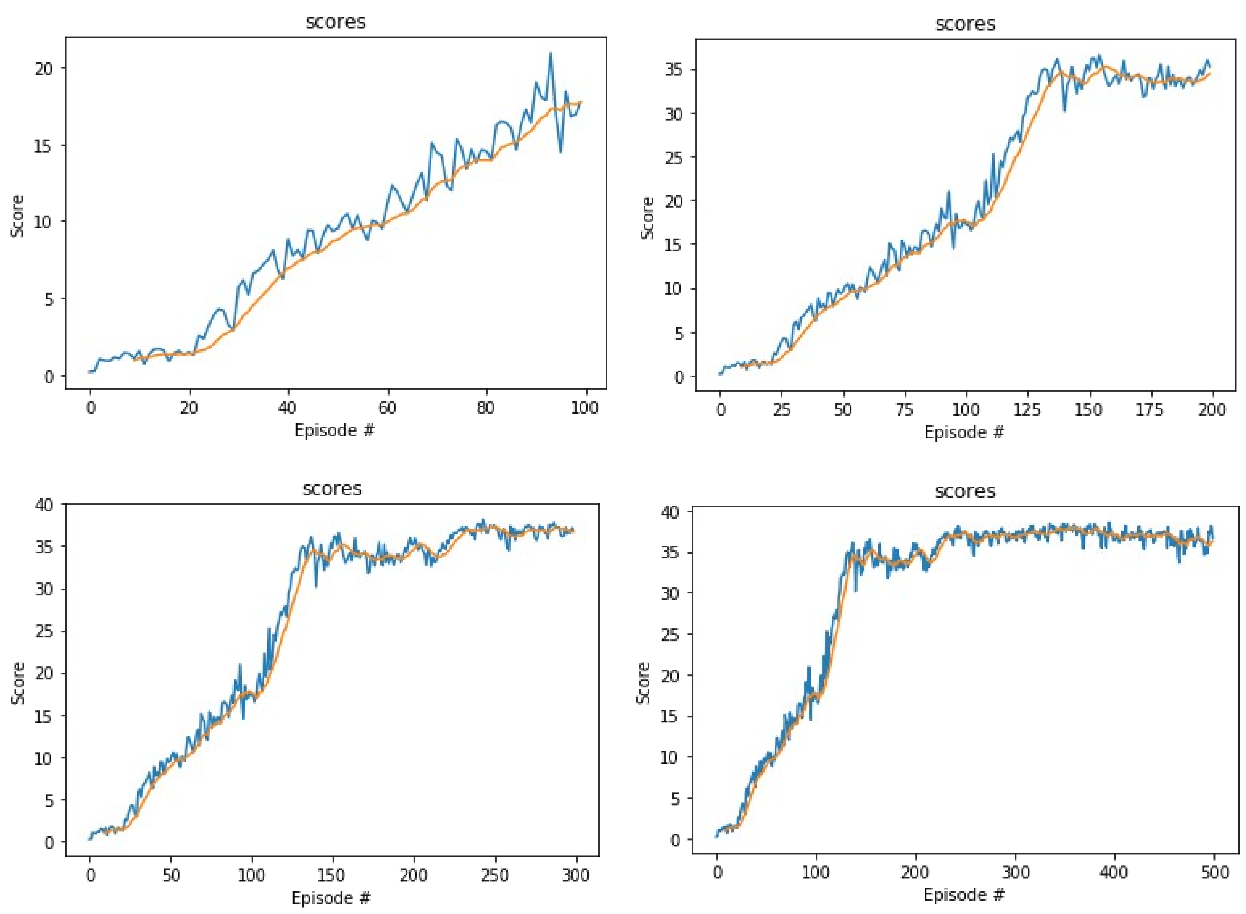


Image 1: The performance result of agents with setting 3 over (a) 100 episodes; (b) 200 episodes; (c) 300 episodes; (d) 500 episodes;

The agents solve the Reacher environment after the episode of 130, and then maintaining the average score (taken across 20 agents) of around 37 afterward.

Ideas for Future Work

Prioritized Experience Replay is encouraged to be implemented so as to replay important transitions more frequently, and therefore learn more efficiently [2]. Moreover, it is worth to try the algorithms like PPO, A3C, and D4PG that use multiple (non-interacting, parallel) copies of the same agent to distribute the task of gathering experience, and then compare with the performance with DDPG.

Reference

1. https://arxiv.org/abs/1509.02971
2. https://arxiv.org/abs/1511.05952