# Continuous_Control based on DDPG

Project Objectives:

The project demonstrates how policy-based methods can be used to learn the optimal policy in a model-free Reinforcement Learning setting using a Unity environment, in which a double-jointed arm can move to target locations.

Reinforcement learning-related projects need to first understand the three elements of environmental state, action, and reward.

In this environment, a double-jointed arm can move to target locations. A reward of +0.1 is provided for each step that the agent's hand is in the goal location. Thus, the goal of your agent is to maintain its position at the target location for as many time steps as possible.
The observation space consists of 33 variables corresponding to position, rotation, velocity, and angular velocities of the arm. Each action is a vector with four numbers, corresponding to torque applicable to two joints. Every entry in the action vector should be a number between -1 and 1.

In this project I chose to solve the first environmental problem, even though the agent scored +30 on average in 100 consecutive episodes.

Project Flow:

The DDPG algorithm can be understood as an improved algorithm of the actor-reviewer algorithm, so its parameter update steps are similar to the actor-reviewer algorithm. However, because the idea of DQN is introduced, both neural networks have a target network. As shown in Figure 1, the part in the virtual box is the target network and online network of the corresponding network. First, the target Q network of the reviewer network calculates the value function $Q(s_i, a_i')$ according to the next state and the action of the target strategy network output of the actor network, and then passes the The output $Q(s_i, a_i)$ forms an error function to update the parameters, thereby affecting its output action value, and finally updating the parameters of the online strategy network in the actor network.

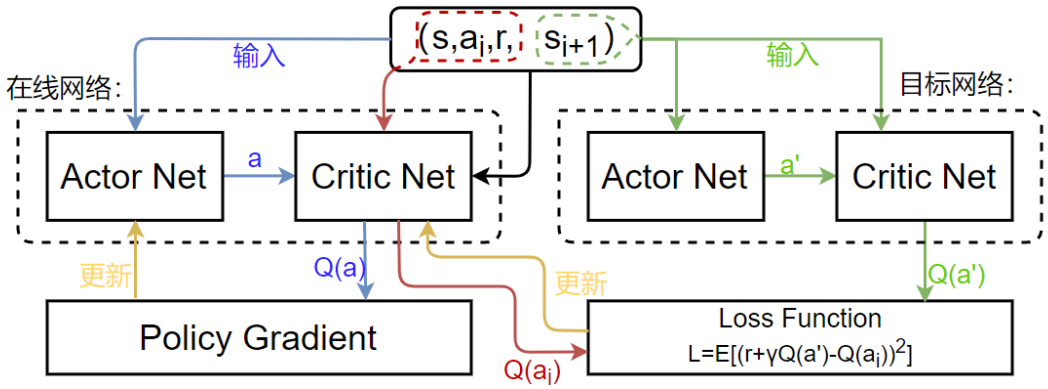



Fig. 1 DDPG algorithm framework

Critic Network Update

The Critic network uses a neural network to approximate the value function. The input is a state action pair $([S_t, A_t])$ and the output is an action value $(Q(S_t, A_t))$. The update method is similar to the DQN algorithm. The formula of the loss function L is:

$$\mathrm{L} = \frac{1}{N}\sum_i (y_i - Q(s_i, a_i|\theta^Q))^2$$

In the above formula, the target Q function $y_i$ can be used as a label for supervised learning. The gradient descent algorithm is used to iteratively update the weights of the neural network parameters $\theta^Q$ and $\theta^{Q'}$, so that the gradient of the online Q network can be obtained, $\nabla_{\theta^Q}\mathrm{L}$. The formula is:

$$y_i = r_i + \gamma \mathrm{Q}'(s_{i+1}, \mu'(s_{i+1}|\theta^{\mu'})|\theta^{Q'})$$

Combining the above two formulas together can calculate the average error of each parameter of the online network in the critic network.

$$\mathrm{L} = \frac{1}{N}\sum_i (r_i + \gamma \mathrm{Q}'(s_{i+1}, \mu'(s_{i+1}|\theta^{\mu'})|\theta^{Q'}) - Q(s_i, a_i|\theta^Q))^2$$

（2）Actor Network Update

The actor network uses another neural network to approximate and parameterize the strategy. The update method is similar to the strategy gradient algorithm. The input is the state $(S_{t+1})$ and the output is the action $(A_{t+1})$. In order to find the optimal strategy, it is necessary to know the gradient formula of the online strategy in the actor network, and iteratively optimize the parameters $\theta^\mu$ and $\theta^{\mu'}$ of the neural network.

Here, the function $J(\mu)$ is used to measure the quality of the output function μ of the offline strategy. By obtaining μ to make J (μ) reach the maximum value, the formula is as follows:

$$J_\beta(\mu) = E_{s\sim\rho^\beta}[Q^\mu(s, \mu(s))]$$

The above formula represents the expected value of the $Q^\mu(s, \mu(s))$ distribution function $\rho^\beta$ produced by the agent's β strategy under state s. $Q^\mu(s, \mu(s))$ represents the Q value generated when the action is selected according to the parameters of $\theta^\mu$ in each state s, so $J_\beta(\mu)$ is the inverse of $\theta^\mu$, the gradient algorithm formula of the deterministic strategy for:

$$\nabla_{\theta^\mu} J_\beta(\mu) \approx E_{s\sim\rho^\beta}[\nabla_a Q(s, a|\theta^Q)|_{a=\mu(s)} \cdot \nabla_{\theta^\mu}\mu(s|\theta^\mu)]$$

At this time, the policy gradient becomes the expected value $(\nabla_\alpha Q \cdot \nabla_{\theta^\mu}\mu)$ when it is distributed according to the distribution function $\rho^\beta$. in the state s. Monte Carlo method is used to estimate this expected value. The conversion process $(s_i, a_i, r_i, s_{i+1})$ in the empirical playback pool is based on the behavior generated by the β strategy. The distribution function

of these data is $\rho^{\beta}$.

So when randomly sampling from it, using mini-batch data to substitute the above formula according to Monte Carlo method can be used as an unbiased estimate of the above expectations, and the formula for the policy gradient is rewritten as:

$$\nabla_{\theta^{\mu}} J_{\beta}(\mu) \approx \frac{1}{N}\sum_{i}(\nabla_{\beta} Q(s,a|\theta^{Q})\,|_{s=s_i,a=\mu(s_i)} \cdot \nabla_{\theta^{\mu}}\mu(s|\theta^{\mu})|_{s=s_i})$$

Among them, $\nabla_{\beta} Q(s,a|\theta^{Q})$comes from the critic network. The meaning is how the action of the actor network moves this time to obtain a larger Q value. $\nabla_{\theta_{\mu}}\mu(s|\theta^{\mu})$ comes from the actor network. The significance is how the actor network will modify its own parameters to make the agent more likely to do this action. The significance of the two together is that the actor network must modify the action parameters in a direction that is likely to obtain a larger Q value.

（3）Online network and Target network

If the DDPG algorithm uses only a single neural network to approximate as the traditional method, because the algorithm needs to calculate the Q value and fit the policy gradient while updating the gradient of its own Q network, it will cause the learning process to be unstable and difficult to converge. . By creating such a target network, you can control the amount of network parameter changes, and use a delayed update mechanism to make the algorithm more stable and easy to converge, but it will also slow the learning process. Based on this, the DDPG algorithm will clone a neural network with the same structure for the strategy network and the Q network.

$$\text{policy network:}\begin{cases} online: \mu(s|\theta^{\mu}) \rightarrow gradient\ update\ \theta^{\mu} \\ target: \mu'(s|\theta^{\mu'}) \rightarrow soft\ update\ \theta^{\mu'} \end{cases}$$

$$Q\ \text{network:}\begin{cases} online: Q(\mathrm{s,a},\theta^{Q}) \rightarrow gradient\ update\ \theta^{Q} \\ target: Q'^{(\mathrm{s,a},\theta^{Q'})} \rightarrow soft\ update\ \theta^{Q'} \end{cases}$$

After the batchsize data is learned by the agent, the parameters in the neural network are updated by gradient descent, and then the parameters in the target network are updated by soft update.

$$soft\ update_{\tau=0.001}: \begin{cases} \theta^{Q\prime} \leftarrow \tau\theta^{Q} + (1-\tau)\theta^{Q\prime} \\ \theta^{\mu\prime} \leftarrow \tau\theta^{\mu} + (1-\tau)\theta^{\mu\prime} \end{cases}$$

Code composition:

In the code part, I mainly refer to the completed multi-agent report to write, because the core of the MADDPG algorithm is DDPG.

My code is mainly composed of four parts, namely: neural network, hyperparameter , agent and training. By referring to the code of MADDPG in Udacity, I use its same model to build neural networks and agents. In the agent code part, there are mainly three parts: behavior function, learning function, and soft update. The main action is to update the weights through the evaluation of the Critic network after the agent takes action through the

Actor network.

In the code defined by the agent, there is an important hyperparameter epsilon that controls the agent's exploration. When the value is closer to 1, the more important it is the future total reward. When the value is equal to 0, the agent only Value the rewards of the current state, regardless of the future. The multi-agent algorithm model can use the Epislon-greedy strategy to control the appropriate exploration environment of the agent. This method reduces the probability for exploration with increasing time. This project uses the SGD optimizer to update the learning rate by multiplying the diminishing factors in each period. Here, a method similar to Epsilon Greedy is used to achieve it. Finally, Ornstein-Uhlenbeck process and empirical playback function are used.

To solve the problem of gradient explosion, the model uses the torch.nn. utils.clip_grad_norm function to implement gradient clipping, and sets the range of the gradient to 1. Prevents them from growing exponentially by setting an upper limit on the size of parameter updates. This can make the agent more stable and fast learning.

In order to explore the action space more, the EPS_START parameter is set to 1.0, so the chance of detecting a signal can be increased. When the noise drops to zero, this extra signal seems to improve learning in subsequent training.

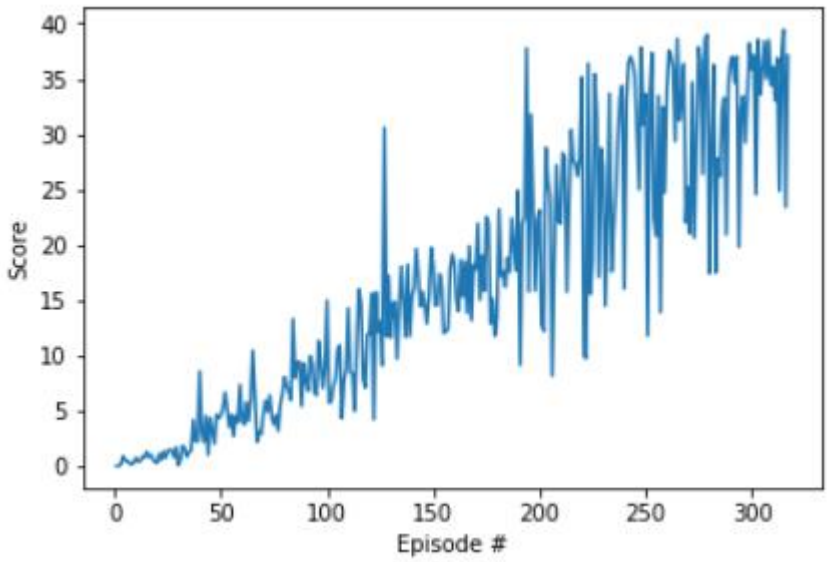
In the training part, by setting the learning interval, multiple learnings per plot are performed to produce faster convergence speed and higher scores. But setting the learning interval, which slows down training, is a question worth weighing. Finally, I set the learning interval to 1, so that the agent will perform the learning steps in each plot, so that it samples experience from the experience pool and runs the learning function of the agent part.

About hyperparameters:

```
EXPERIENCE_POOL_SIZE = 100000 # define the experience pool size
BATCH_SIZE = 128 # minibatch size
LR_ACTOR = 0.0002 # actor network learning rate
LR_CRITIC = 0.0003 # critic network learning rate
WEIGHT_DECAY = 0 # L2 weight decay
LEARN_EVERY = 20 # learning interval
GAMMA = 0.99 # discount factor
TAU = 0.001 # soft update parameters
OU_SIGMA = 0.2 # Ornstein-Uhlenbeck noise parameter, volatility
OU_THETA = 0.15 # Ornstein-Uhlenbeck noise parameter, mean recovery speed
EPS_START = 1.0 # Initial value of ε during noise attenuation
EPS_DECAY = 0.00001 #to decay exploration as it learns
EPS_FINAL = 0.05 # final value of ε after attenuation
GOAL_SCORE = 30 # goal score
WINDOW_LENGTH = 100 # limited number of actions
PRINT_EVERY = 10 # print the score every tenth time
ADD_NOISE = True # add noise
```

Operation result:

The following figure shows the final training results.




After 317 episodes of training, the agent can solve the problem of the project environment. The highest score is 38.3 and the highest average is 30.03

Future Development:

Algorithmically:

DDPG achieves DQN and decreases convergence speed in tasks in continuous action space, but it cannot use random environment problems. Then, I think the most important improvement of reinforcement learning is the use of transfer learning. If the agent can continuously improve through memory, then general artificial intelligence is just around the corner. Finally, the priority experience playback algorithm is not used in this model training. The priority experience reply does not learn the data stored in the experience pool through random selection, but chooses the experience based on the priority value related to the error size.

At Work:

I will consider using DDPG and MADDPG for medical diagnosis. By treating the patient's physical state as a dynamic environment and each external drug dose as an action, I will explore the application of artificial intelligence in medicine in this way.