Udacity Deep Reinforcement Learning Project 3 - Collaboration and Competition

Jonathan Salfity | December, 2018

Collaboration and Competition

Observation Space, Reward Structure, Action Space

In this environment, two agents control rackets to bounce a ball over a net. If an agent hits the ball over the net, it receives a reward of +0.1. If an agent lets a ball hit the ground or hits the ball out of bounds, it receives a reward of -0.01. The reward structure encourages each agent to keep the ball in play.

The observation space consists of 8 variables corresponding to the position and velocity of the ball and racket. Each agent has its own observation.

The action space consists of two continuous actions, movement in the lateral and vertical direction.

The task is episodic and agents must get an average score of +0.5 over 100 consecutive episodes.

Learning Algorithm

Deep Deterministic Policy Gradient (DDPG)

The learning algorithm used was Deep Deterministic Policy Gradient (DDPG), a policy gradient reinforcement learning algorithm. DDPG relies on an actor-critic architecture. An actor is used to hold the network weights $\boldsymbol{\theta}$ for the policy function.

$$\pi_\theta(s, a) = \mathbb{P}[a \mid s, \theta]$$

The above equation represents a policy function, $\pi_\theta(s, a)$, where the policy function takes in the current state-action pair and outputs the optimal next action.

A critic is used for evaluating the policy function estimated by the actor according to the temporal difference (TD) error:

$$r_{t+1} + \gamma V^v(S_{t+1}) - V^v(S_t)$$

Where the lower-case *v* denotes the policy that the actor has decided. In this way the critic acts like a Q-Learning agent to learn the optimal action to take in a given state and the equation closely resembles the temporal difference (TD) learning equation from Q-Learning.

DDPG implements an experience replay and separate target network so there exists both 'local' and 'target' networks for both the actor and critic**.** The local networks are updated after each timestep while the target networks are 'soft' updated according to this equation:

$$\theta_{target} = \tau * \theta_{local} + (1 - \tau) * \theta_{target}$$

Where τ is a hyperparameter which can be set prior to training.

In this project, each agent has its own Actor and Critic networks. Each agent learns from a shared memory buffer that consists of the experiences from both agents. Each agent maximizes their sum of discounted future rewards independent of the other agent. Each agent learns to collaborate because of the nature of the reward function and how it guides each agent.

Training the Network

Below is the DDPG pseudo code that is found in textbooks:

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

Software Framework

A Jupyter notebook hosted on a Udacity server with GPU support was used to write and run code. PyTorch was used to build and train a deep neural network.

A large part of this project was built leveraging the DDPG algorithm implemented by Udacity in the DDPG Pendulum repository.

Deep Neural Network Model Architecture

Each agent embodied an Actor-Critic neural network architecture:

The Actor neural network was comprised of two fully connected layers containing 128 hidden units. A ReLu non-linear activation function was used between each layer with batch normalization. The output had 4 neurons and a tanh non-linearity to keep the outputs in the range [-1,1].

The model can be seen in `model.py`

```
        self.model = nn.Sequential(
             nn.Linear(state_size, fc1_units),
             nn.ReLU(),
             nn.BatchNorm1d(fc1_units),
             nn.Linear(fc1_units, fc2_units),
             nn.ReLU(),
             nn.BatchNorm1d(fc2_units),
             nn.Linear(fc2_units, action_size),
             nn.Tanh()
         )
```

The Critic neural network was comprised of two fully connected containing 128 hidden units. The output from the first layer was passed through a ReLU non-linear activation, then through a batch normalization regularization layer and finally as input to the second layer. Here it was concatenated with the 4 actions to give 132 hidden units, then through a ReLu and finally a single output node

The model can be seen in `model.py`

```
        self.model_input = nn.Sequential(
             nn.Linear(state_size, fc1_units),
             nn.ReLU(),
             nn.BatchNorm1d(fc1_units)
             )
         self.model_output = nn.Sequential(
             nn.Linear(fc1_units + action_size, fc2_units),
             nn.ReLU(),
             nn.Linear(fc2_units, 1)
             )
```

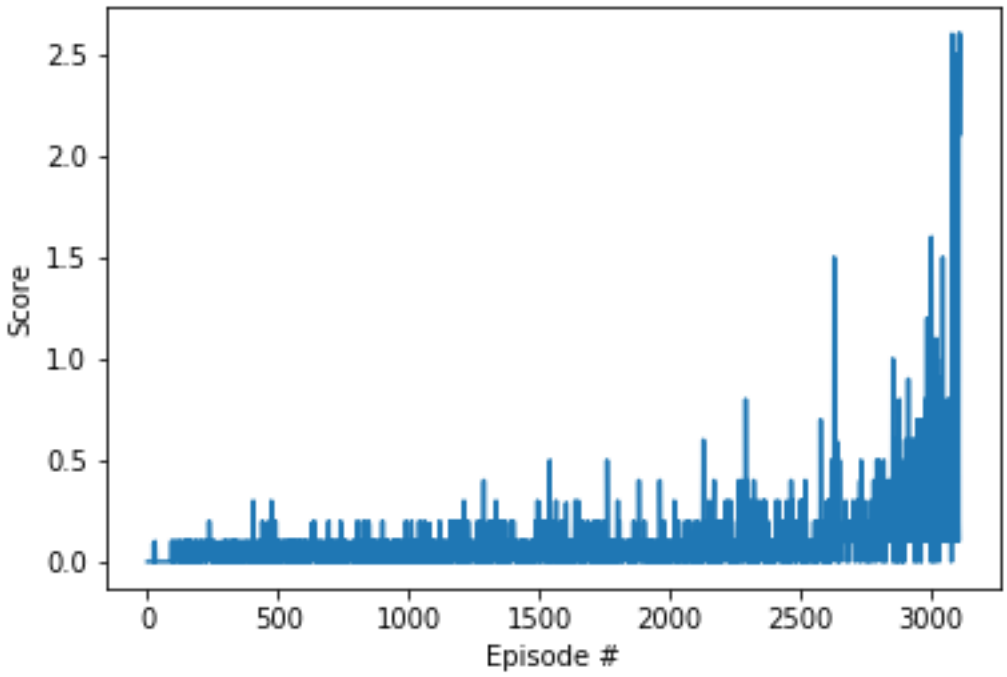
HyperParameters

```
BUFFER_SIZE = int(1e6)  # replay buffer size
BATCH_SIZE = 512         # minibatch size
GAMMA = 0.99            # discount factor
TAU = 0.1              # for soft update of target parameters
LR_ACTOR = 1e-4         # learning rate of the actor
LR_CRITIC = 3e-4        # learning rate of the critic
```

```
WEIGHT_DECAY = 0.0001    # L2 weight decay
UPDATE_EVERY = 1         # how often to update the network (first ran
with 20)
EPSILON = 1.0
```

Results

As noted by the project description, multiple agents tend to learn a bit slower and be more unstable at the beginning of training. The episode was solved around 3000 episodes.



Ideas for Future Work

The hyperparameters listed above are the biggest control knob of this experiment. A separate loop structure that sat above this training could be allowed to search over a matrix of the hyperparameter configurations.

It's clear from the char that each agent takes a long time to begin learning. Once the agents begin to learn, they learn quickly. The OU Noise process for exploration could be increased to increase the range of actions the agents take in early episodes. This experiment does not deal with many dangerous (highly penalized) scenarios, so exploration would not greatly penalize each agent. Once a more optimal trajectory is found in early stage, epsilon could be altered to exploit that path.

The structures of the neural networks and different non-linear activation functions could be altered to see if a deeper network or more hidden neurons would allow the agent to train faster.

Appendix

https://github.com/udacity/deep-reinforcement-learning/

https://github.com/udacity/deep-reinforcement-learning/tree/master/ddpg-pendulum

https://github.com/Unity-Technologies/ml-agents/blob/master/docs/Learning-Environment-Examples.md

Reinforcement Learning: An Introduction, Ricard Sutton and Andrew Barto