Udacity Deep Reinforcement Learning Nano Degree

Report of Project 3: Collaboration and Competition

Pongsasit Thongpramoon

Section 1: Learning Algorithm

Section 2: Plot of Rewards

Section 3: Ideas for Future Work

Section 1: Learning Algorithm

Section 1.1: Deep Deterministic Policy Gradient (DDPG), algorithm overview

DDPG uses many of the same techniques found in DQN. It uses a replay buffer to train an action-value function in an off-policy manner, and target networks to stabilize training. However, DDPG also trains a policy that approximates the optimal action. Because of this, DDPG is a deterministic policy-gradient method restricted to continuous action spaces. The agent collects experiences in an online manner and stores these online experience samples into a replay buffer. On every step, the agent pulls out a mini-batch from the replay buffer that is commonly sampled uniformly at random. The agent then uses this mini-batch to calculate a bootstrapped TD target and train a Q-function. The main difference between DQN and DDPG is that while DQN uses the target Q-function for getting the greedy action using an argmax, DDPG uses a target deterministic policy function that is trained to approximate that greedy action. Instead of using the argmax of the Q-function of the next state to get the greedy action as we do in DQN, in DDPG, we directly approximate the best action in the next state using a policy function. Then, in both, we use that action with the Q-function to get the max value.

Pseudocode

Algorithm 1 Deep Deterministic Policy Gradient

Input: initial policy parameters θ, Q-function parameters ϕ, empty replay buffer $\mathcal{D}$

Set target parameters equal to main parameters $\theta_{\text{targ}} \leftarrow \theta$, $\phi_{\text{targ}} \leftarrow \phi$

repeat

  Observe state s and select action $a = \text{clip}(\mu_\theta(s) + \epsilon, a_{Low}, a_{High})$, where $\epsilon \sim \mathcal{N}$

  Execute a in the environment

  Observe next state s', reward r, and done signal d to indicate whether s' is terminal

  Store (s, a, r, s', d) in replay buffer $\mathcal{D}$

  If s' is terminal, reset environment state.

  if it's time to update **then**

    for however many updates **do**

      Randomly sample a batch of transitions, $B = \{(s, a, r, s', d)\}$ from $\mathcal{D}$

      Compute targets

$$y(r, s', d) = r + \gamma(1 - d) Q_{\phi_{\text{targ}}}(s', \mu_{\theta_{\text{targ}}}(s'))$$

      Update Q-function by one step of gradient descent using

$$\nabla_\phi \frac{1}{|B|} \sum_{(s,a,r,s',d) \in B} \left(Q_\phi(s, a) - y(r, s', d)\right)^2$$

      Update policy by one step of gradient ascent using

$$\nabla_\theta \frac{1}{|B|} \sum_{s \in B} Q_\phi(s, \mu_\theta(s))$$

      Update target networks with

$$\phi_{\text{targ}} \leftarrow \rho \phi_{\text{targ}} + (1 - \rho)\phi$$
$$\theta_{\text{targ}} \leftarrow \rho \theta_{\text{targ}} + (1 - \rho)\theta$$

    end for

  end if

until convergence

Ref: https://spinningup.openai.com/en/latest/algorithms/ddpg.html

We want to train a network that can give us the optimal action in a given state. The network must be differentiable with respect to the action. Therefore, the action must be continuous to make for efficient gradient-based learning. The objective is simple; we can use the expected Q-value using the policy network, mu. That is, the agent tries to find the action that maximizes this value. Notice that in practice, we use minimization techniques, and therefore minimize the negative of this objective.

Also notice that, in this case, we don't use target networks, but the online networks for both the policy, which is the action selection portion, and the value function (the action evaluation portion). Additionally, given that we need to sample a mini-batch of states for training the value function, we can use these same states for training the policy network. ***<u>So we have actor network and critic network.</u>***

Section 1.2: Multi Agent Deep Deterministic Policy Gradient (MADDPG), algorithm overview

Multi-Agent Deep Deterministic Policy Gradient Algorithm

For completeness, we provide the MADDPG algorithm below.

Algorithm 1: Multi-Agent Deep Deterministic Policy Gradient for N agents

for episode $= 1$ to M **do**
  Initialize a random process $\mathcal{N}$ for action exploration
  Receive initial state $\mathbf{x}$
  for $t = 1$ to max-episode-length **do**
    for each agent i, select action $a_i = \boldsymbol{\mu}_{\theta_i}(o_i) + \mathcal{N}_t$ w.r.t. the current policy and exploration
    Execute actions $a = (a_1, \ldots, a_N)$ and observe reward r and new state $\mathbf{x}'$
    Store $(\mathbf{x}, a, r, \mathbf{x}')$ in replay buffer $\mathcal{D}$
    $\mathbf{x} \leftarrow \mathbf{x}'$
    for agent $i = 1$ to N **do**
      Sample a random minibatch of S samples $(\mathbf{x}^j, a^j, r^j, \mathbf{x}'^j)$ from $\mathcal{D}$
      Set $y^j = r_i^j + \gamma Q_i^{\boldsymbol{\mu}'}(\mathbf{x}'^j, a'_1, \ldots, a'_N)|_{a'_k = \boldsymbol{\mu}'_k(o_k^j)}$
      Update critic by minimizing the loss $\mathcal{L}(\theta_i) = \frac{1}{S}\sum_j \left(y^j - Q_i^{\boldsymbol{\mu}}(\mathbf{x}^j, a_1^j, \ldots, a_N^j)\right)^2$
      Update actor using the sampled policy gradient:

$$\nabla_{\theta_i} J \approx \frac{1}{S} \sum_j \nabla_{\theta_i} \boldsymbol{\mu}_i(o_i^j) \nabla_{a_i} Q_i^{\boldsymbol{\mu}}(\mathbf{x}^j, a_1^j, \ldots, a_i, \ldots, a_N^j)\big|_{a_i = \boldsymbol{\mu}_i(o_i^j)}$$

    end for
    Update target network parameters for each agent i:

$$\theta'_i \leftarrow \tau \theta_i + (1 - \tau)\theta'_i$$

  end for
end for

During the training, the Critics networks have access to the states and actions information of both agents, while the Actors networks have only access to the information corresponding to their local agent.

Section 1.3: Model and Agent Code explanation

1. model.py
   Class Actor

Initialize the class with 3 Fully connected Neural network (without any dropout and batch normalization).

- First layer: input size =24 (state size of the environment), output size =400
- Second layer: input size =400, output size =300
- Third layer: input size =300, output size =2 (action size)
- 1 batch normalize layer for normalize the first fully connected layer.

Function: reset_parameters().

- Reset weights of parameters by fills the given 2-dimensional matrix with values drawn from a uniform distribution.

Function: forward()

- Forward input through the 1^{st} network then activate layer's output by relu function.
- Batch Normalize the output from relu function.
- Forward batch normalized output from above through the 2^{nd} network then activate layer's output by relu function.
- Batch Normalize the output from the 2^{nd} relu function.
- Forward batch normalized output from above through the 3^{rd} network then activate layer's output by tanh function.

Class Critic

Initialize the class with 3 Fully connected Neural network (without any dropout and batch normalization).

- First layer: input size =24 (state size of the environment), output size =300
- Second layer: input size =300, output size =300
- Third layer: input size =300, output size =2

Function: reset_parameters().

- Reset weights of parameters by fills the given 2-dimensional matrix with values drawn from a uniform distribution.

Function: forward()

- Concatenate state with action (get from environment) → got xs
- Feed xs into the 1^{st} fully connected layer then operate its output with relu fuction → got x.
- Batch normalization to x.
- Feed x into the 2^{nd} fully connected layer then operate its output with relu fuction → x (replace an old x)
- Feed x to the 3^{rd} fully connected layer get 2 dimensions output.

2. dqn_agent.py (Agent)

BUFFER_SIZE = int(1e5) , BATCH_SIZE = 128, GAMMA = 0.99, TAU = 1e-3 , LR = 1e-3, WEIGHT_DECAY = 0

Class Agent:

Initialize 4 instances of the actor local network and the actor target network,

critic local network and the critic target network.

Initialize the noise (Ornstein-Uhlenbeck process).

Initialize state size and action size.

function step():

Allows to store a step taken by the agent (state, action, reward, next_state, done) in the replay buffer. And if their are enough samples available in the Replay Buffer, will call function learn()

function act():

Use policy function to get action. And returns actions for the given state as per current policy.

function learn():

Update the Neural Network value parameters. Critic network update by “mse loss” of Q value from critic target network and Q value from critic local network. And Actor network update by negative value of critic network.

function soft_update():

Softly updates the value from the target Neural Network from the local network weights. (use in learn() function)

Class OUNoise:

Add noise.

3. maddpg_agents.py

Implement the MADDPG alorithm.

The `maddpg_agents` is relying on the `ddpg` class from dqn_agent.py.

- It instanciates DDPG Agents
- It provides a helper function to save the models checkpoints
- It provides the `step()` and `act()` methods
- As the Multi-Agent Actor Critic `learn()` function slightly differs from the DDPG one, a `maddpg_learn()` method is provided here. The `learn()` method updates the policy and value parameters using given batch of experience tuples.

```
Q_targets = r + γ * critic_target(next_state, actor_target(next_state))
where:
    actor_target(states) -> action
    critic_target(all_states, all_actions) -> Q-value
```

ref: https://arxiv.org/pdf/1706.02275.pdf

4. memory.py

Class ReplayBuffer:

- function add(): allows to add an experience step to the memory.
- function sample(): allows to randomly sample a batch of experience steps for the learning.

5. hyperparameters.py

Defines all the hyperparameters in constant variables.

```
import torch.nn.functional as F

# Default hyperparameters

SEED = 10                                   # Random seed

NB_EPISODES = 100000                         # Max nb of episodes
NB_STEPS = 10000                             # Max nb of steps per episodes
UPDATE_EVERY_NB_EPISODE = 4                 # Nb of episodes between learning process
MULTIPLE_LEARN_PER_UPDATE = 3               # Nb of multiple learning process performed in a row

BUFFER_SIZE = int(1e5)                      # replay buffer size
BATCH_SIZE = 200                            # minibatch size

ACTOR_FC1_UNITS = 400   #256                # Number of units for the layer 1 in the actor model
ACTOR_FC2_UNITS = 300   #128                # Number of units for the layer 2 in the actor model
CRITIC_FCS1_UNITS = 400   #256              # Number of units for the layer 1 in the critic model
CRITIC_FC2_UNITS = 300   #128               # Number of units for the layer 2 in the critic model
NON_LIN = F.relu    #F.leaky_relu           # Non linearity operator used in the model
LR_ACTOR = 1e-4     #1e-4                   # learning rate of the actor
LR_CRITIC = 5e-3    #2e-3                   # learning rate of the critic
WEIGHT_DECAY = 0    #0.0001                 # L2 weight decay

GAMMA = 0.995 #0.99                         # Discount factor
TAU = 1e-3                                  # For soft update of target parameters
CLIP_CRITIC_GRADIENT = False                # Clip gradient during Critic optimization

ADD_OU_NOISE = True                         # Add Ornstein-Uhlenbeck noise
MU = 0.                                     # Ornstein-Uhlenbeck noise parameter
THETA = 0.15                                # Ornstein-Uhlenbeck noise parameter
SIGMA = 0.2                                 # Ornstein-Uhlenbeck noise parameter
NOISE = 1.0                                 # Initial Noise Amplitude
NOISE_REDUCTION = 1.0 # 0.995               # Noise amplitude decay ratio
```

Section 2: Plot of Rewards

Section 2.1: Training code explanation

Continuous_Control20agents_train.ipynb

- Import the Necessary Packages.
- Examine the State and Action Spaces.
- Take Random Actions in the Environment (No display).
- Train an agent using Deep Deterministic Policy Gradient.

for each episode.
reset an environment, get state, initialize score to zero.

In each episode run over all of the steps until the episode done.
(Agent)
Let an actor act.

(Environment)
Get an action from the actor to the environment.
Get next state.
Get reward.
Get done status.

(Agent)
Store a step taken by the multi agents.
Update the target network weights with the current weight values from the local network.
Update by add reward to score that initialized per episode.

Check if Done then break out the steps loop.

Update scores of all episodes (list) by append the score (float).

And save model if the model can reach the score we want

Return the train function with scores (list of score of every episodes).

- Plot the scores

Section 2.2: Plot of Rewards

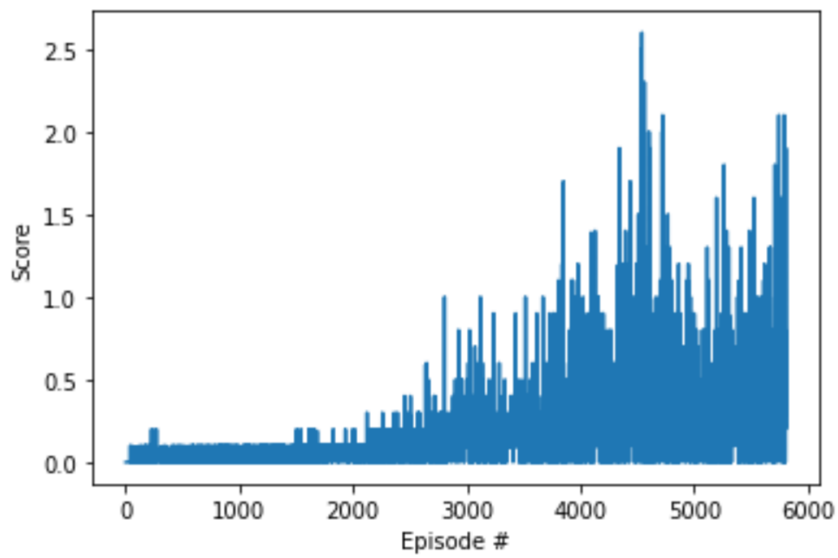
In [11]:
```
# Launch training
scores = train()
plot_training(scores)
```

```
/home/pongsasit/anaconda3/envs/drlnd/lib/python3.6/site-packages/torch/nn/functional.py:1698: UserWarning: nn.functional.tanh is deprec
ed. Use torch.tanh instead.
  warnings.warn("nn.functional.tanh is deprecated. Use torch.tanh instead.")
```

```
Episode 15      Average Score: 0.00     Episode score (max over agents): 0.00
```

```
/media/pongsasit/480/deepRlUdacity/deep-reinforcement-learning/p3_collab-compet/maddpg_agents.py:131: UserWarning: Using a target size
orch.Size([200, 2])) that is different to the input size (torch.Size([200, 1])). This will likely lead to incorrect results due to broa
asting. Please ensure they have the same size.
  critic_loss = F.mse_loss(Q_expected, Q_targets)
```

```
Episode 100     Average Score: 0.01 (nb of total steps=1653   noise=1.0000)
Episode 200     Average Score: 0.02 (nb of total steps=3478   noise=1.0000)
Episode 300     Average Score: 0.02 (nb of total steps=5454   noise=1.0000)
Episode 400     Average Score: 0.02 (nb of total steps=7187   noise=1.0000)
Episode 500     Average Score: 0.01 (nb of total steps=8919   noise=1.0000)
Episode 600     Average Score: 0.01 (nb of total steps=10554   noise=1.0000)
Episode 700     Average Score: 0.01 (nb of total steps=12286   noise=1.0000)
Episode 800     Average Score: 0.02 (nb of total steps=14168   noise=1.0000)
Episode 900     Average Score: 0.03 (nb of total steps=16407   noise=1.0000)
Episode 1000    Average Score: 0.03 (nb of total steps=18489   noise=1.0000)
Episode 1100    Average Score: 0.05 (nb of total steps=20999   noise=1.0000)
Episode 1200    Average Score: 0.04 (nb of total steps=23225   noise=1.0000)
Episode 1300    Average Score: 0.05 (nb of total steps=25515   noise=1.0000)
Episode 1400    Average Score: 0.07 (nb of total steps=28373   noise=1.0000)
Episode 1500    Average Score: 0.04 (nb of total steps=30683   noise=1.0000)
Episode 1600    Average Score: 0.06 (nb of total steps=33496   noise=1.0000)
Episode 1700    Average Score: 0.09 (nb of total steps=36800   noise=1.0000)
Episode 1800    Average Score: 0.08 (nb of total steps=39788   noise=1.0000)
Episode 1900    Average Score: 0.09 (nb of total steps=42796   noise=1.0000)
Episode 2000    Average Score: 0.09 (nb of total steps=46040   noise=1.0000)
Episode 2100    Average Score: 0.07 (nb of total steps=48780   noise=1.0000)
Episode 2200    Average Score: 0.09 (nb of total steps=52173   noise=1.0000)
Episode 2300    Average Score: 0.09 (nb of total steps=55733   noise=1.0000)
Episode 2400    Average Score: 0.10 (nb of total steps=59807   noise=1.0000)
Episode 2500    Average Score: 0.10 (nb of total steps=64316   noise=1.0000)
Episode 2600    Average Score: 0.12 (nb of total steps=69115   noise=1.0000)
Episode 2700    Average Score: 0.12 (nb of total steps=74353   noise=1.0000)
Episode 2800    Average Score: 0.12 (nb of total steps=79500   noise=1.0000)
Episode 2900    Average Score: 0.13 (nb of total steps=84780   noise=1.0000)
Episode 3000    Average Score: 0.14 (nb of total steps=90591   noise=1.0000)
Episode 3100    Average Score: 0.17 (nb of total steps=98015   noise=1.0000)
Episode 3200    Average Score: 0.21 (nb of total steps=106680   noise=1.0000)
Episode 3300    Average Score: 0.15 (nb of total steps=112777   noise=1.0000)
Episode 3400    Average Score: 0.13 (nb of total steps=118000   noise=1.0000)
Episode 3500    Average Score: 0.14 (nb of total steps=123768   noise=1.0000)
Episode 3600    Average Score: 0.16 (nb of total steps=130308   noise=1.0000)
Episode 3700    Average Score: 0.19 (nb of total steps=137549   noise=1.0000)
Episode 3800    Average Score: 0.21 (nb of total steps=146006   noise=1.0000)
Episode 3900    Average Score: 0.21 (nb of total steps=154060   noise=1.0000)
Episode 4000    Average Score: 0.27 (nb of total steps=164572   noise=1.0000)
Episode 4100    Average Score: 0.28 (nb of total steps=175692   noise=1.0000)
Episode 4200    Average Score: 0.34 (nb of total steps=189102   noise=1.0000)
Episode 4300    Average Score: 0.21 (nb of total steps=197590   noise=1.0000)
Episode 4400    Average Score: 0.37 (nb of total steps=212267   noise=1.0000)
Episode 4500    Average Score: 0.38 (nb of total steps=227512   noise=1.0000)
Episode 4600    Average Score: 0.38 (nb of total steps=242745   noise=1.0000)
Episode 4700    Average Score: 0.34 (nb of total steps=256070   noise=1.0000)
Episode 4800    Average Score: 0.40 (nb of total steps=271913   noise=1.0000)
Episode 4900    Average Score: 0.29 (nb of total steps=283740   noise=1.0000)
Episode 5000    Average Score: 0.29 (nb of total steps=295259   noise=1.0000)
Episode 5100    Average Score: 0.24 (nb of total steps=305228   noise=1.0000)
Episode 5200    Average Score: 0.28 (nb of total steps=316446   noise=1.0000)
Episode 5300    Average Score: 0.35 (nb of total steps=330448   noise=1.0000)
Episode 5400    Average Score: 0.32 (nb of total steps=343197   noise=1.0000)
Episode 5500    Average Score: 0.35 (nb of total steps=357150   noise=1.0000)
Episode 5600    Average Score: 0.28 (nb of total steps=368493   noise=1.0000)
Episode 5700    Average Score: 0.36 (nb of total steps=382889   noise=1.0000)
Episode 5800    Average Score: 0.47 (nb of total steps=401371   noise=1.0000)
Environment solved in 5808 episodes with an Average Score of 0.50 0.29
```



At the episode 5808th, the system can get average score more than 0.5.

Section 3: Ideas for Future Work

Section 3.1: Improvement of Model Architecture

- Including a drop out layer to reduce the chance of overfitting.
- Use L2 regularization to reduce the chance of overfitting.
- More stable gradient flow, add batch normalisasion.

Section 3.2: Improvement of Algorithm (Agent itself)

1. twin-delayed DDPG (TD3)

- DOUBLE LEARNING IN DDPG

- SMOOTHING THE TARGETS USED FOR POLICY UPDATES

- DELAYING UPDATES

2. SAC: Maximizing the expected return and entropy

- ADDING THE ENTROPY TO THE BELLMAN EQUATIONS

- LEARNING THE ACTION-VALUE FUNCTION: Use two networks approximating the Q-function and take the minimum estimate for most calculations.

A few differences, however, are that, with SAC, independently optimizing each Q-function yields better results.

Second, add the entropy term to the target values. And last,don't use the target action smoothing directly

- LEARNING THE POLICY: Use a squashed Gaussian policy that, in the forward pass, outputs the mean and standard deviation.

Then we can use those to sample from that distribution, squash the values with a hyperbolic tangent function tanh, and then rescale the values to the range expected by the environment.

For training the policy, use the reparameterization trick. This "trick" consists of moving the stochasticity out of the network and into an input. This way, the network is deterministic, and we can train it without problems.

- AUTOMATICALLY TUNING THE ENTROPY COEFFICIENT

Reference: https://learning.oreilly.com/library/view/grokking-deep-reinforcement/9781617295454/

Pongsasit Thongpramoon