Udacity Deep Reinforcement Learning Nano Degree

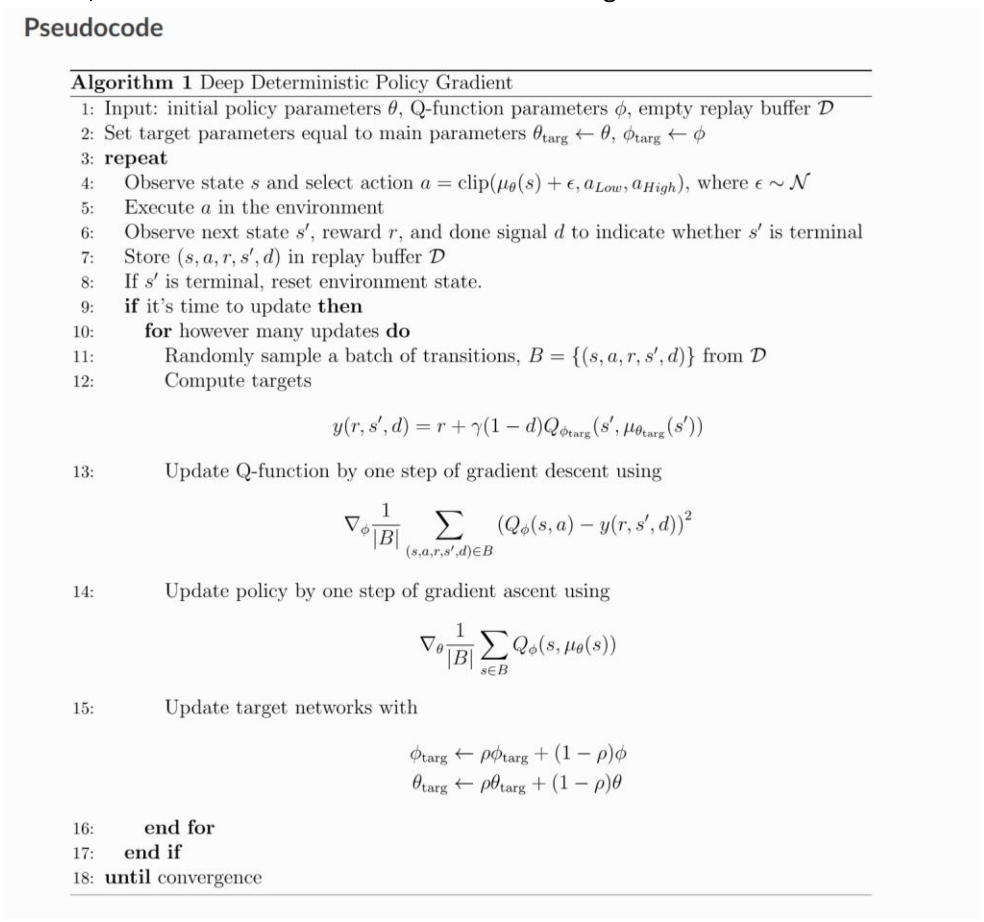
Report of Project 2: Reacher Continuous Control

Pongsasit Thongpramoon

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

```
1: Input: initial policy parameters $\theta$, Q-function parameters $\phi$, empty replay buffer $\mathcal{D}$
2: Set target parameters equal to main parameters $\theta_{\text{targ}} \leftarrow \theta$, $\phi_{\text{targ}} \leftarrow \phi$
3: repeat
4:   Observe state $s$ and select action $a = \text{clip}(\mu_\theta(s) + \epsilon, a_{Low}, a_{High})$, where $\epsilon \sim \mathcal{N}$
5:   Execute $a$ in the environment
6:   Observe next state $s'$, reward $r$, and done signal $d$ to indicate whether $s'$ is terminal
7:   Store $(s, a, r, s', d)$ in replay buffer $\mathcal{D}$
8:   If $s'$ is terminal, reset environment state.
9:   if it's time to update then
10:    for however many updates do
11:      Randomly sample a batch of transitions, $B = \{(s, a, r, s', d)\}$ from $\mathcal{D}$
12:      Compute targets
             $y(r, s', d) = r + \gamma(1-d)Q_{\phi_{\text{targ}}}(s', \mu_{\theta_{\text{targ}}}(s'))$
13:      Update Q-function by one step of gradient descent using
             $\nabla_\phi \frac{1}{|B|} \sum_{(s,a,r,s',d)\in B} (Q_\phi(s,a) - y(r,s',d))^2$
14:      Update policy by one step of gradient ascent using
             $\nabla_\theta \frac{1}{|B|} \sum_{s\in B} Q_\phi(s, \mu_\theta(s))$
15:      Update target networks with
             $\phi_{\text{targ}} \leftarrow \rho\phi_{\text{targ}} + (1-\rho)\phi$
             $\theta_{\text{targ}} \leftarrow \rho\theta_{\text{targ}} + (1-\rho)\theta$
16:    end for
17:  end if
18: 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: 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 =33 (state size of the environment), output size =128
- Second layer: input size =128, output size =128
- Third layer: input size =128, output size =4 (action size)

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 network from 1st to 3rd layers with relu activation function and feed output from last layer to Tanh function.

Class Critic

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

- First layer: input size =33 (state size of the environment), output size =128
- Second layer: input size =132, output size =128
- Third layer: input size =128, output size =1

Function: reset_parameters().

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

Function: forward()

- Input state (the state get from environment) and feed into the 1st fully connected layer then operate its output with relu fuction → got xs.
- Concatenate xs with action (get from environment) → got x
- Feed x into the 2nd fully connected layer then operate its output with relu fuction → x (replace an old x)
- Feed x to the 3rd fully connected layer get 1 dimension 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 memory buffer (Replay Buffer).

   Initialize the memory buffer (Ornstein-Uhlenbeck process).

   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.

   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.

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 agent.
    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).
```

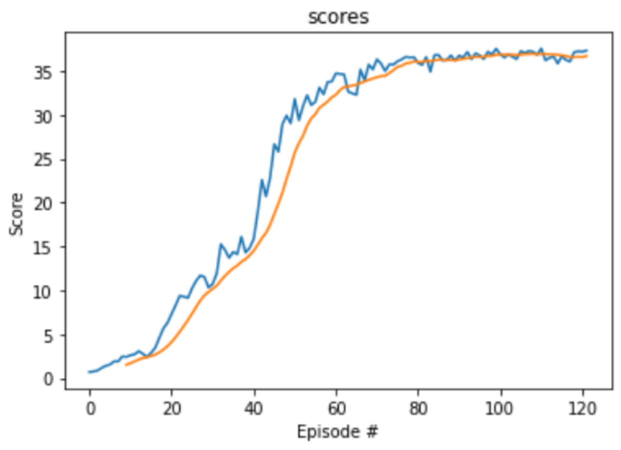
- Plot the scores

Section 2.2: Plot of Rewards

```
In [12]: scores = ddpg()
```

```
/home/pongsasit/anaconda3/envs/drlnd/lib/python3.6/site-packages/torch/nn/functio
nal.py:1698: UserWarning: nn.functional.tanh is deprecated. Use torch.tanh instea
d.
  warnings.warn("nn.functional.tanh is deprecated. Use torch.tanh instead.")
Episode 100	Average Score: 22.83
/home/pongsasit/anaconda3/envs/drlnd/lib/python3.6/site-packages/torch/nn/functio
nal.py:1698: UserWarning: nn.functional.tanh is deprecated. Use torch.tanh instea
d.
  warnings.warn("nn.functional.tanh is deprecated. Use torch.tanh instead.")
Episode 122	Average Score: 30.24
```

```
In [14]: plot_scores(scores)
```



At the episode 122th, the system can get average score more than 30.

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

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