DEEP DETERMINISTIC POLICY GRADIENT FOR CONTINUOUS CONTROL

HUBERT DE LASSUS

Training Multiple Agents To Solve A Continuous Control Task Using Deep Deterministic Policy Gradient.

For this toy example, the problem is to solve in UNITY ML environment the task of letting 20 robotic arms each keep control of its ball as it pushes the ball around. First, let's define some terminology. The world we address is defined as a Markovian Decision Process MDP with states s, action a, reward r and the conditional Probability P of future state s' given current state s, action a, reward r and discount factor gamma."

Environment: The conditional distribution of the state transitions and the reward function constitute the model of the environment.

Action: set A of available move from a given state s to a state s'. Only the elements of the set A where P(s'—s,a) ¿ 0 are considered.

Episode: An episode is a complete sequence of events from an initial state to a final state.

Terminal states: The states that have no available actions are called terminal states.

Cumulative reward: The cumulative reward is the discounted sum of rewards accumulated throughout an episode.

Policy: A Policy is the agent's strategy to choose an action at each state.

Optimal policy: the policy that maximizes the expectation of cumulative reward.

The problem is modeled as an episodic task during which the agents have to maximize the expected cumulative rewards. Because we choose to model the solution as a DDPG algorithm, let us introduce its concept.

DDPG algorithm concept. This is an algorithm that lies in between Value based methods and Policy based methods. While actor function specifies action a given the current state of the environment s, critic value function specifies a TD Error to criticize the actions made by the actor.

• Stochastic methods are of the form: $\pi_\theta(a|s) = \mathbb{P}[a|s;\theta]$

• Deterministic methods are of the form: $a = \mu_\theta(s)$

• Computing stochastic gradient requires more samples, as it integrates over both state and action space. Deterministic gradient is preferable as it integrates over state space only.

• In DQN, action was selected as: $a_t = max_a Q^*(\phi(s_t), a; \theta)$

• Above equation is not practical for continuous action space. Using deterministic policy allows us to use: $a_t = \mu(s_t|\theta_\mu)$ But,deterministic policy gradient might not explore the full state and action space. To overcome this, we introduce a noise process N:

$$a_t = \mu((s_t|\theta_\mu) + N_t)$$

This replaces the epsilon greedy algorithm of DQN for state and action space exploration. We want to maximize the rewards (Q-values) received over the sampled mini-batch. The gradient is given as:

$$\nabla_{\theta_\mu} J \approx \mathbb{E}_{s_t} \sim \rho^\beta [\nabla_{\theta^\mu} Q(s, a|\theta^Q)|_{s=s_t, a=\mu(s_t|\theta^\mu)}]$$

Applying the chain rule using mu instead of Q we get

$$\nabla_{\theta_\mu} J = \mathbb{E}_{s_t} \sim \rho^\beta [\nabla_a Q(s, a|\theta^Q)|_{s=s_t, a=\mu(s_t)} \nabla_{\theta_\mu} \mu(s|\theta^\mu)|_{s=s_t}]$$

This equation yields the maximum expected reward as we update the parameters using gradient ascent.

DDPG Algorithm steps. (These steps are borrowed from San Jose University CS DDPG course hand out.)

Randomly initialize critic network $Q(s, a|\theta^Q)$ and actor $\mu(s|\theta^\mu)$ with weights θ^Q and θ^μ
Initialize replay buffer R
Initialize target networks Q' and μ' with weights

$$\theta^{Q'} \leftarrow \theta^Q, \theta^{\mu'} \leftarrow \theta^\mu$$

This step use target network to do off-policy updates
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 using deterministic actor $a_t = \mu(s_t|\theta^\mu) + \mathcal{N}_t$
do it according to the current policy and exploration noise.
Execute action a_t and observe reward r_t, then 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.
This is the experience replay sequence.
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

Learning Algorithm. The actor implements a Policy gradient algorithm

The critic implements a Q-Learning algorithm takes two steps:

1) A sample step

2) A learn step

To sample the environment a Multi Layer Perceptron is used to estimate the value actions based on environment observations.

Experience replay is used to reduce the oscillations of the output function of the network and accelerate the learning process by emphasizing the most meaningful samples. During the Learning step, a batch of past experiences is randomly sampled to train the agent. The randomness of this experiences selection, helps also the learning process by reducing correlation between input samples.

Hyperparameters. The parameters pertaining to the algorithms are

- Replay buffer size: int(1e6)
- Minibatch size: 64
- Discount factor: 0.99
- Learning rate actor : 1e-4
- Learning rate critic : 3e-4
- L2 Weight decay : 0.0001
- Tau soft update of target parameters: 1e-3
- epsilon init value, final value, update value

The parameters pertaining to the actor network are

- Number of layers and nodes: 1 input layer with 33 nodes, 2 hidden layers with 256 fully connected nodes, 1 output fully connected layer layer with 4 nodes.
- Activation function: hyperbolic tangent of ReLu
- The neural network optimizer: Adam optimizer and its parameters.

The parameters pertaining to the critic network are

- Number of layers and nodes: 1 input layer with 33 nodes, 2 hidden layers with 256 fully connected nodes, 1 hidden layer with 128 fully connected nodes, 1 output fully connected layer layer with 1 node.
- Activation function: hyperbolic tangent of ReLu
- The neural network optimizer: Adam optimizer and its parameters.

Actor Model Architecture. The actor model architecture is defined by 5 variables: $State_size$ is the input dimensions of the network $Action_size$ is the output dimensions of the network Seed will initialize the weights of the network " $fc1_units$ and $fc2_units$ are the number of nodes in the hidden layers of the network

The input layer has 33 nodes 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 action vector value is between $[-1, 1]$.

The first hidden layer has 256 and the second 256 nodes. The output layer has four nodes one for each action. Optimizing the number of nodes in the network leads to faster learning and better generalization. By setting the number of hidden layers to two and using dichotomy node sampling, I eventually found that 40 hidden nodes on two hidden layers give better results than a larger network.

Training. The training runs for 300 episodes with 500-time steps per episode. When the average of the 100 most recent scores reaches 30 the problem is considered solved and the algorithm stops.

A training episode is conducted as follows: 1) Select the most likely agent action given the current state and epsilon. 2) Retrieve the environment response and update next state, reward and done flag 3) Update the agent structure with present state, present action, reward and next state 4) Update the state 5) Update the reward

Every so episodes decrease the value of epsilon towards zero.

The scores achieved during the learning process are:

Episode 100 Average Score: 6.28

Episode 200 Average Score: 10.24

Episode 300 Average Score: 12.94

Episode 302 Average Score: 13.04

Environment solved in 202 episodes! Average Score: 13.04

End Score: 19.0

Ideas For Future Work. There are may different ways this project can be improved. However, the problem we have to solve is simple and it is not warranted that resource hungry refinements are necessary in this case. The list of classical algorithms that could be tried includes double DQN, dueling DQN and prioritized experience replay.

Plot of Rewards. Figure 1 shows the scores during training.

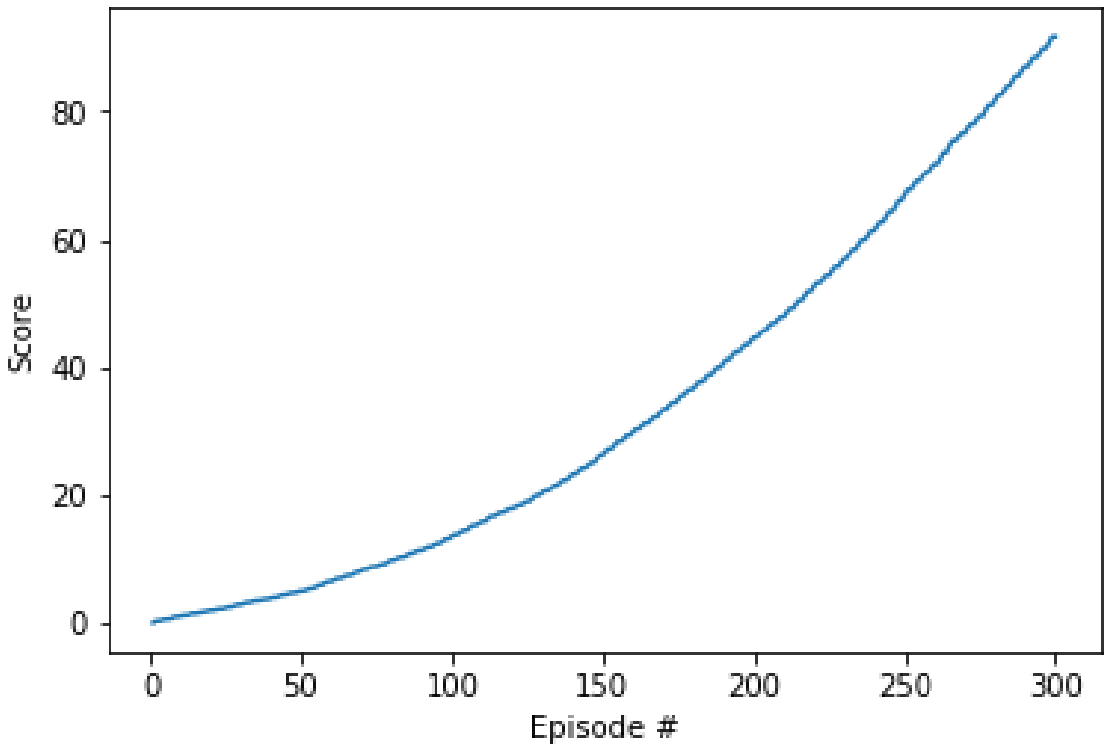

FIGURE 1. Scores.