DEEP DETERMINISTIC POLICY GRADIENT FOR COLLABORATION AND COMPETITION

HUBERT DE LASSUS

Training Multiple Agents To Solve A Collaborative and Competitive Task Using Deep Deterministic Policy Gradient.

For this toy example, the problem is to solve in UNITY ML environment the task of letting 2 agents play tennis by bouncing a ball over a net with rackets. This implementation is largely borrowed from the bipedal. Improvements include using batch normalization to stabilize the learning process and gradient clipping. Most of the effort was allocated to tuning the models and finding the sweet spots in the hyper parameter spaces.

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 γ.

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 Tennis game is modeled as an episodic task during which the players (called "agents" in Reinforcement Learning terminology) have to maximize the expected cumulative rewards.

Date: December 13, 2018.

Because we choose to model the solution as a DDPG algorithm, let us introduce the Deep Deterministic Policy Gradient 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 Temporal Difference (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 the Deep Q Network (DQN) algorithm, 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: $a_t = \mu(s_t|\theta_\mu)$ instead. Since, deterministic policy gradient might not explore the full state and action space we have to introduce an Ornstein-Uhlenbeck noise process N to overcome this limitation. The process is a stationary Gauss–Markov process.Over time, the process tends to drift towards its long-term mean, this is why it is called mean-reverting.

The process is a modification of the random walk in continuous time, in which the properties of the process have been changed so that there is a tendency of the walk to move back towards a central location, with a greater attraction when the process is further away from the center. This gives the Noise process nice stabilizing properties that can be put to work in the off policy segment of the DDPG algorithm.

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

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 μ 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. (This analysis is 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 with three main steps:

1) A sample step
2) A learn step
3) A model change 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:

- Number of agents: 2
- State size: 24
- Action size: 2
- Replay buffer size: int(1e6)
- Minibatch size: 512
- Discount factor: 0.99
- Learning rate actor : 5e-4
- Learning rate critic : 1e-3
- L2 Weight decay : 0.0001
- τ soft update of target parameters: 1e-3
- Noise level $\mu = 0, \theta = 0.15, \sigma = 0.2$

The parameters pertaining to the actor network are:

- Number of layers and nodes: 1 input layer with 48 nodes, 2 hidden layers with 64 fully connected nodes, 1 output fully connected layer layer with 2 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 48 nodes, 2 hidden layers with 64 fully connected nodes,
  1 output fully connected layer layer with 2 nodes.
- 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: $StateSize$ is the input dimensions of the network, $ActionSize$ is the output dimensions of the network. Seed will initialize the weights of the network. $fc1units$

and $fc2units$ are the number of nodes in the hidden layers of the network. $fc3units$ is the output layer with two output node.

Finding the Right Number of Nodes. Following Occam razor principle, it is important to design the networks with the minimum number of nodes necessary to achieve the best performance. A network with fewer nodes will generalize better and will run faster. Figure 1 shows the training time as a function of number of nodes per layer for two hidden layer Multi Layer Perceptron.

Plot of Training Time. Figure1

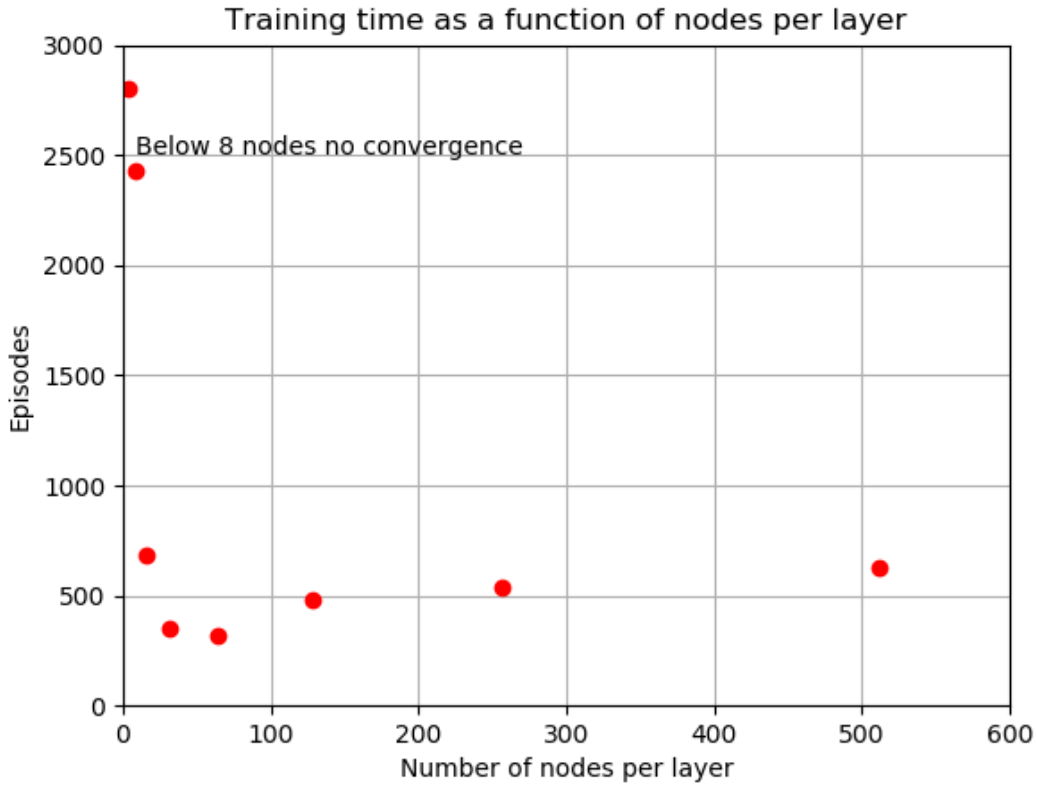

FIGURE 1. Training Time.

Figure 1 shows the training time depending on the number of nodes in the network hidden layers . We assume two hidden layers of equal size.

The input layer has 48 nodes corresponding to position, velocity of the rackets and ball. Two continuous actions are available, corresponding to movement toward (or away from) the net, and jumping. Specifically, each action is a vector with two numbers, corresponding to move – jump, value is between $[-1, 1]$.

The first hidden layer has 64 and the second 64 nodes. The output layer has 2 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 two 64 nodes hidden layers give better results than a larger network.

Critic Model Architecture. The critic model architecture is defined by 5 variables: $StateSize$ is the input dimensions of the network. $ActionSize$ is the output dimensions of the network . Seed will initialize the weights of the network. $fc1units$ and $fc2unit$ are the number of nodes in the hidden layers of the network. The first layer has 64 nodes, the second layer has 64 nodes. $fc3units$ is the output layer with two output nodes.

The input layer has 48 nodes corresponding to position, velocity of the rackets and ball. Two continuous actions are available, corresponding to movement toward (or away from) the net, and jumping. Each action is a vector with two numbers, corresponding to move – jump, value is between $[-1, 1]$.

Optimizing the number of nodes in the network to a minimum leads to faster learning and better generalization. I eventually found that 64-64 nodes on two hidden layers give similar or better results than a larger network.

Training. The training runs for a maximum of 5000 episodes with 100-time steps per episode. When the average of the 100 most recent scores reaches 0.5 the problem is considered solved and the algorithm stops.

A training episode is conducted as follows:

1) Get predicted next-state actions from actor target models

2) Get Q values from critic for this next-state

3) Compute Q targets for current states (y_i),

$$Q_{targets} = rewards + (\gamma * Q_{targetsnext} * (1 - dones))$$

4) Compute critic loss

5) Minimize the critic loss by backpropagation with adam optimizer

6) Minimize the actor loss by backpropagation with adam optimizer of mean critic loss

7) Update the target networks policy and value parameters using given batch of experience tuples.

$$Qtargets = r + \gamma * critictarget(nextstate, actortarget(nextstate))$$

where: $actortarget(state) \rightarrow action$ $critictarget(state, action) \rightarrow Qvalue$

Scores Dynamic. This 2 agents DDPG algorithm training time depends on Networks size, "Tau" copy coefficient, initial conditions, learning rates for actor and critic and Unity Environment response.

The example run in the notebook reaches the goal in 318 episodes :
The scores achieved during the learning process are:
Episode 100 Score: -0.00 Average Score: 0.00
Episode 200 Score: 0.10 Average Score: 0.03
Episode 300 Score: 0.09 Average Score: 0.24
Episode 318 Score: 2.60 Average Score: 0.50
Environment solved in 318 episodes! Average Score: 0.50

Typically if you run multiple time the code, changing the learning rates and target update rates, the algorithm will solve the problem at various speed depending on the target update coefficient Tau and the learning parameters. Getting faster training requires greater learning/update rates at the expense of stability: in some cases, the algorithm will diverge with aggressive rates.

Figure 2 shows the scores obtained in the notebook.

Plot of Rewards. Figure2
Figure 2 shows the scores during training.

Ideas For Future Work. Future work will include binary heap implementation of Prioritized Replay Memory and more refined tuning of hyper parameters.

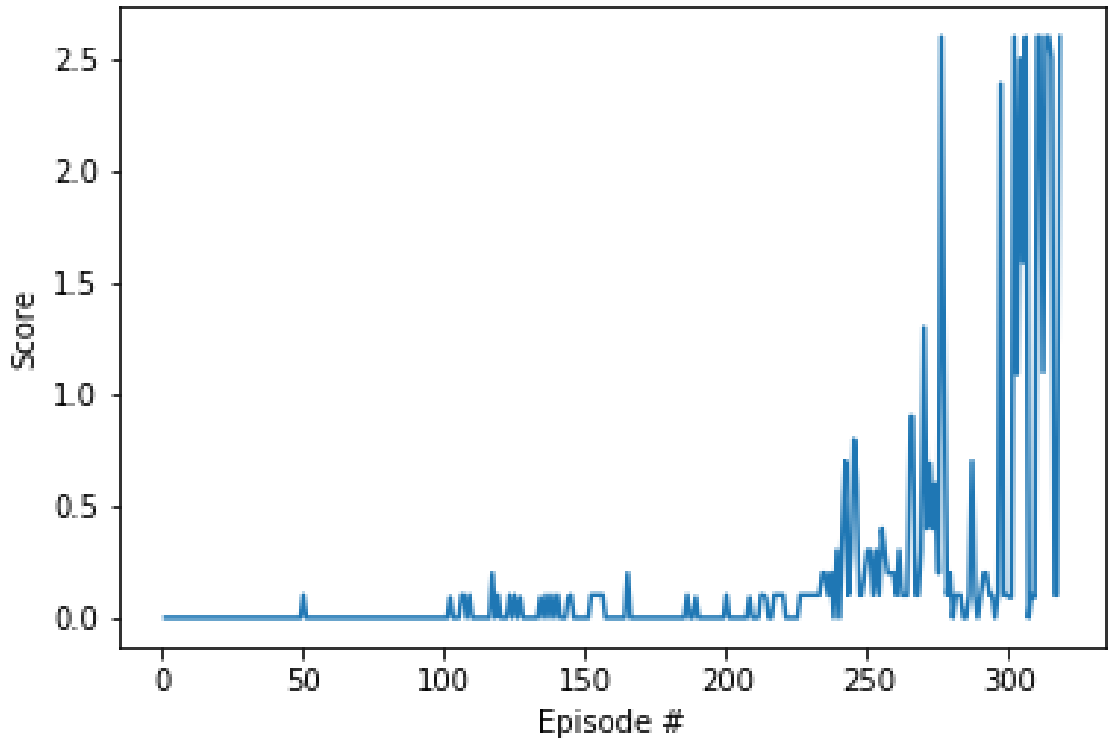

FIGURE 2. Scores.