Contents

DDPG Model implementation

Description of DDGP model use for collaboration game

The goal of this DDGP model is to train two Agents who bounce a tennis ball together. The game is collaborative.

Strategy used

The plan is to create two DDPG agents, each composed of a first DNN (the Actor) whose task is to approximate the action value function Q(s,a) by minimizing the squared distance of the Actor's predicted action/value vs it equivalent Bellman Equation while also training a second DNN (the Critic) whose task is to evaluate a given situation Q(s). Each agent is sharing a common memory with the other agent via a common replay buffer.

```
#create a shared memory for boths agents
buffer=ReplayBuffer(action_size=2, buffer_size=int(1e6), batch_size=512, seed=2)

agentA = Agentv03(state_size=24, action_size=2,repBuffer=buffer, random_seed=2)
agentB=Agentv03(state_size=24, action_size=2,repBuffer=buffer, random_seed=2)
```

```
    for t in range(max_t):
        stateA=states[0].reshape(-1,1).T # transpose the array to fit the model
        stateB=states[1].reshape(-1,1).T
        actions[0] = agentA.act(stateA) #take each of the 2 actions and assign each to an AI agent
        actions[1] = agentB.act(stateB) #take each of the 2 actions and assign each to an AI agent
```

The DDPG parameters of each agent are the same as in assignment 2. The only modification being a shared replay buffer among the agents as shown above

The unity environment provides 2 states at every step, one for each agent. The idea is to allocate each state to each DDPG agent since this collaborative game can be seen as simply 2 individual agents bouncing a ball against a wall. Each agent returns an action which is then sent to the unity brain that plays the environment.

Pseudo Code of the DDPG Algorithm:

1. Run a state input through each actor agent A and B
2. Collect experiences consisting of (actions, rewards, inputs) tuples from the actor
3. When enough experiences are memorized the update Actor & Critic Agent
4. Update critic agent
    a. Use actor predicted next action to obtain Q_next as Critic(action_next)
        i. Calculate target Q(S,A) as reward+ discount*Q_next
    b. Calculate the expected Q(S,A) as Critic(action)
    c. Calculate Cost function as "target – expected"
5. Update the Agent's parameters:
    a. For a given state sampled, predict the actions using the Actor

        i. Action_predicted=Actor(state)

   b. Loss is defined as the average expected Critic Agent value

        i. Average_over_actions(Critic(state,action predicted))

6. Perform as soft update of the Actor and critic target
   a. Target actor & critic are update as x*target + (1-x)*local_model
7. Iterate until Agent has discovery an adequate policy.

Agent Parameters:

BUFFER_SIZE = int(1e6) # replay buffer size

BATCH_SIZE = 512 # minibatch size

GAMMA = 0.99 # discount factor

TAU = 1e-3 # for soft update of target parameters

LR_ACTOR = 1e-3 # learning rate of the actor

LR_CRITIC = 1e-3 # learning rate of the critic

WEIGHT_DECAY = 0 # L2 weight decay

UPDATE_EVERY = 20 # how often to update the network

Parameters update

The parameters of the Agent are updated 10 times every 20 steps. The code that performs this is as follow:

```
# Learn, if enough samples are available in memory
        if len(self.memory) > BATCH_SIZE:
          for nbtimes in range(10):
            experiences = self.memory.sample()
            self.learn(experiences, GAMMA)
```

Agent's Actor and Critic network:

The model's architecture for approximating the Q(s,a) is the one described by the Google Deep Mind research paper.

The Actor's parameters:

The actor consist of one single hidden layer fully connected neural network of with one hidden layer of 128 nodes and an output layer of 4 nodes describing the action value function for each 4 possible actions. The output uses the tanH so that the model's output are between -1 and 1

The Critic's parameters:

The critic consist of a DNN fully connected with 3 hidden layers of 128,64,32 nodes reaching a final output of 1 that is between 0 and 1 to value the current state being passed to him.

The critic model also has a gradient clipping to improve performance

```
torch.nn.utils.clip_grad_norm(self.critic_local.parameters(), 1)
```

Batch normalization of actor and critic:

The input of both the actor and the critic use batch normalization.

```
    super(Critic, self).__init__()

    self.seed = torch.manual_seed(seed)

    self.bn0 = nn.BatchNorm1d(state_size)

def forward(self, state, action):

    """Build a critic (value) network that maps (state, action) pairs -> Q-values."""

    stateN=self.bn0(state)

    xs = F.relu(self.fcs1(stateN))
```

This has for effect to dampen the volatility that would be cause by an input with high low values. Example one dimension of the input is between -1000,1000 and the other one between 0-1, the first one will dominate the model and training will be poor.

Random Noise generator:

The Theta paremeter has been changed from 0.2 to 0.05. Also, The original code generated numbers between 0-1 but has been changed to the following:

```
#dx = self.theta * (self.mu - x) + self.sigma * np.array([random.random() for i in range(len(x))])

    dx =   self.theta * (self.mu - x)+self.sigma*np.random.uniform(-1, 1, len(x))
```

Results of Model Training:

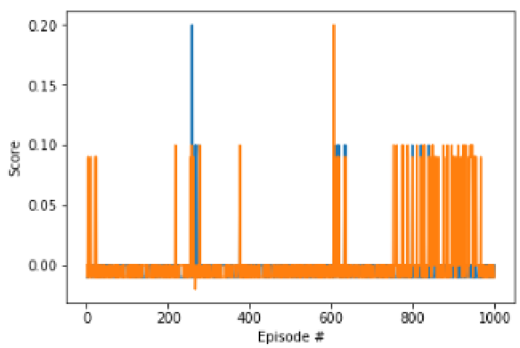
The graph starts with a model that was run for around 2500+ episodes for a total combined training time.

The model starts at a score of 0 and unfortunately doesn't increase at all even after a very large number of episodes

Running more episodes doesn't seem to work.

Graph of results

```
Episode 100	Average Score: -0.00
Episode 200	Average Score: -0.00
Episode 300	Average Score: 0.000
Episode 400	Average Score: -0.00
Episode 500	Average Score: -0.00
Episode 600	Average Score: -0.00
Episode 700	Average Score: 0.000
Episode 800	Average Score: -0.00
Episode 900	Average Score: 0.010
Episode 1000	Average Score: 0.01
```



Potential Improvements:

Using MADDPG could improve overall performance but it is a more complex algorithm to calibrate