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 a simple agents, which is 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).

The DDPG parameters of each agent had to be modified from assignment 2. After quite a few trial and error tests, the numbers of parameters in both actors & critic were increased to 512-256 for each of the first and second layer.A dropout rates was also added equivalent to 20%.Other dropout rates were tried but were not as effective.

The unity environment provides 2 states at every step, one for each agent. The agent provides a set of actions corresponding to the 2 states. He then loops through the actions to store them in memory for future training steps.

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

Parameters update

The parameters of the Agent are updated 10 times every 5 steps. The code that performs this is as follow. I had to extract some of the code of the Agent and put it into the DDPG function because otherwise the time step update was out of sync because of the multiple actions that are stored were previously counted as individual time step in my previous assignment. The "learnNiceThings() function is inside the Agent and the loop over the number of updates is done inside the DDPG. This breaks down the step function from my previous assignment into two functions which is better because in OOP each function should have only one job.

```
# Learn, if enough samples are available in memory

def learnNiceThings(self):

        if len(self.memory) > BATCH_SIZE:

            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 two hidden layer fully connected neural network of with one hidden layer of 512-256 nodes and an output layer of 2 nodes describing the action value function for each 2 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 2 hidden layers of 512-256 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)

Dropout Rate

A 20% dropout rate has been added to avoid overfitting. It has been tried in other cases to uses dropouts, it almost always have a positive effects. 20% is a standard starting point.

Batch normalization of actor and critic:

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

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:

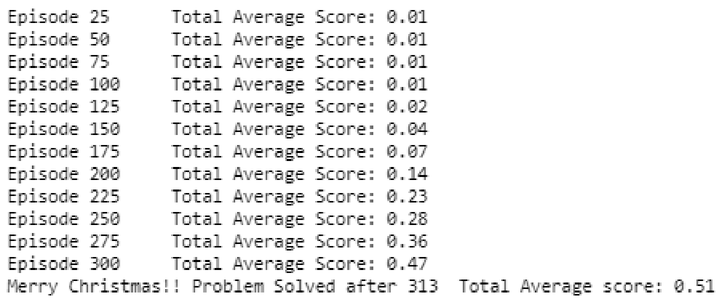
```
#dx = self.theta * (self.mu - x) + self.sigma * np.array([random.random() for i in range(len(x))])

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

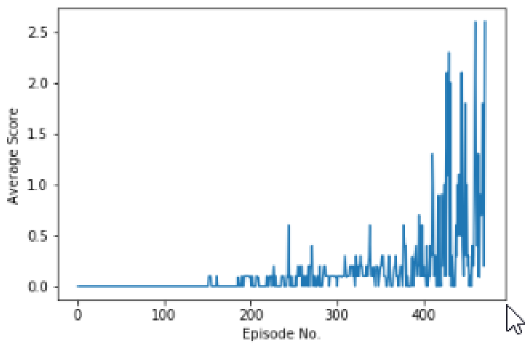
Results of Model Training:

The problem is solved with roughly 500 episodes with a DDPG with augmented number of nodes and dropout rates. This again surprises me given that with slightly different parameters it take at least twice as much time or maybe even not at all.

Graph of results



```
Episode 25      Total Average Score: 0.01
Episode 50      Total Average Score: 0.01
Episode 75      Total Average Score: 0.01
Episode 100     Total Average Score: 0.01
Episode 125     Total Average Score: 0.02
Episode 150     Total Average Score: 0.04
Episode 175     Total Average Score: 0.07
Episode 200     Total Average Score: 0.14
Episode 225     Total Average Score: 0.23
Episode 250     Total Average Score: 0.28
Episode 275     Total Average Score: 0.36
Episode 300     Total Average Score: 0.47
Merry Christmas!! Problem Solved after 313  Total Average score: 0.51
```



Potential Improvements:

Using MADDPG could improve overall performance but it is a more complex algorithm to calibrate. I tried but it didn’t work really well so that is why I reverted to simpler.

The hardest part of developing DRL algorithms is that you don’t know in advance what will work in terms of number of layers, calibration parameters and all other fine tuning. The only way to know is to let it run for a long time and see what happens. Maybe the best way to circumvent this problem is people sharing their work so that this “evolution trial and error” happens as fast as possible and the sames mistakes are not repeated too often by too many people. In my case I took a guess by increasing the number of nodes in layers because I remembered the Machine learning course with Andrew Ng and it happened to work but I didn’t know a priori.

Also, there is certainly a better way to code this whole thing than what I have done. Although it works, it doesn’t appear very scalable because of the hard coded parameters.