Tennis project report

Part 1 Algorithm for solving the Tennis environment

The tennis environment involves 2 agents playing against each other. To solve this environment, I used two methods: self-play method and maddpg method. The pseudocodes for the two methods are listed in the following blocks:

Algorithm: self-play with DDPG

0: Hyperparameters: Train_every N_learn_updates

1: Input: initial policy network (Actor) with parameters θ, Q-function network (Critic) with parameters ϕ, empty replay buffer D, agent size AS

2: Set the target networks' parameters equal to main parameters $\theta_{\text{target}} \leftarrow \theta, \phi_{target} \leftarrow \phi$

3: episode_count = 0

4: Repeat

5: Observe state s and select action a = $\text{clip}(\mu_\theta(s) + \epsilon, a_{Low}, a_{High})$, a includes action for both agents

6: Execute *a* in the environment

7: Observe next state s', reward r, and done signal d to indicate whether s' is terminal

8: **for** both agents **do**

9: Store (s,a, r,s',d) in replay buffer D, a is the action taken by this agent, r is the reward received by this agent

10. **end for**

11. episode_count += 1

12: If s' is terminal, reset environment state.

13: **If** episode_count % Train_every_steps == 0 **then**

14: **for** N_learn_updates **do**

15: Randomly sample a batch of transitions, B = {(s,a,r ,s',d)} from D

16: Compute targets

$$y(r, s', d) = r + \gamma(1 - d) Q_{\phi_{targ}(s', \mu_{\theta_{target}}(s'))}$$

Algorithm: adapted MADDPG

0: Hyperparameters: Train_every N_learn_updates

1: Input: agent size AS, AS initial policy networks (Actors) with parameters $\theta = \theta_1, \ldots, \theta_{AS}$, centralized Q-function network (Critic) with parameters ϕ, empty replay buffer D,

2: Set the target networks' parameters equal to main parameters $\theta_{\text{target}} \leftarrow \theta, \phi_{target} \leftarrow \phi$

3: episode_count = 0

4: Repeat

5: Observe local state s_{local}, global observation state s_{global} and select action a, *where* $a_i = \text{clip}(\mu_{\theta_i}(s_i) + \epsilon, a_{Low}, a_{High})$, s_i is the local observation for each agent.

6: Execute *a* in the environment

7: Observe next local state $s_{local}{}'$, next global observation state $s_{global}{}'$, reward r, and done signal d to indicate whether $s_{local}{}'$ is terminal

8: Store (s_{local}, s_{global},a,r, $s_{local}{}'$,$s_{global}{}'$) in replay buffer D

9: episode_count += 1

10: if $s_{local}{}'$ is terminal, reset environment state

11: **If** episode_count % Train_every_steps == 0 **then**

12: **for** N_learn_updates **do**

13: Randomly sample a batch of transitions, B = {(s_{local}, s_{global},a,r, $s_{local}{}'$,$s_{global}{}'$))} from D

14: Compute targets

$$y(r, s_{\text{local}}{}', s_{\text{global}}{}', d) = r + \gamma(1 - d) Q_{\phi_{targ}(s'_{global}, \mu_{\theta_{target}}(s_{local}{}'))}$$

Update Q-function by one step of gradient descent using

$$\nabla_{\phi} \frac{1}{|B|} \sum_{(s,a,r_{avg},s',d)\in B} \left(Q_{\phi}(s,a) - y(r,s',d)\right)^2$$

Update policy by one step of gradient ascent using

$$\nabla_{\theta} \frac{1}{|B|} \sum_{s\in B} Q_{\phi}(s, \mu_{\theta}(s))$$

Update target networks with

$$\phi_{\text{target}} \leftarrow \tau\phi_{target} + (1-\tau)\phi$$

$$\theta_{\text{target}} \leftarrow \tau\theta_{target} + (1-\tau)\theta$$

end for

end if

until environment solved

Update Q-function by one step of gradient descent using

$$\nabla_{\phi} \frac{1}{|B|} \sum_{(s,a,r,s'_{avg},d)\in B} \left(Q_{\phi}(s_{global}, a) - y(r, s_{\text{local}}', s_{\text{global}}', \text{d})\right)^2$$

Update policy by one step of gradient ascent using

$$\nabla_{\theta} \frac{1}{|B|} \sum_{s\in B} Q_{\phi}(s_{global}, \mu_{\theta}(s_{local}))$$

Update target networks with

$$\phi_{\text{target}} \leftarrow \tau\phi_{target} + (1-\tau)\phi$$

$$\theta_{\text{target}} \leftarrow \tau\theta_{target} + (1-\tau)\theta$$

end for

end if

until environment solved

The implementations of self-play method are package self_play and self-play-test. They are implemented based on the same algorithm described above, with a tiny but critical difference which will be discussed in part 3. While implementing the self-play method, I referenced nunesma's reinforcement learning repo.

The implementation of maddpg method is presented in package maddpg_agent. While implementing the maddpg method, I referenced the code in udacity's maddpg-lab.

Part 2 Basic neural network design and Hyperparameter selection

Part 2.1: Neural Network architecture

The neural networks for both self-play method and maddpg algorithm are with same hidden layer size.

Actor network:

Layer	Layer type	Input size	Output size	Activation function	Parameter Initialization
1	Fully connected	State size	fc1_units = 128	Relu	Uniform distribution $[-\frac{1}{\sqrt{state\ size}}, \frac{1}{\sqrt{state\ size}}]$
2	Fully connected	128	fc2_units = 256	Relu	Uniform distribution $[-\frac{1}{\sqrt{128}}, \frac{1}{\sqrt{128}}]$
3	Fully connected	256	Action size	tanh	Uniformly sampled within $[-3 \times 10^{-3}, 3 \times 10^{-3}]$

Critic network:

Layer	Layer type	Input size	Output size	Activation function	Parameter Initialization
1	Fully connected	State size (local state size for self-play; Global state for maddpg)	fc1_units =128	Relu	Uniform distribution $[-\frac{1}{\sqrt{state\ size}}, \frac{1}{\sqrt{state\ size}}]$
2	Batch normalize	128	128	/	/
3	Fully connected	128+action size (action for 1 agent if self-play, action for all agents if maddpg)	fc2_units 256	Relu	Uniform distribution $[-\frac{1}{\sqrt{128+\text{action size}}}, \frac{1}{\sqrt{128+\text{action size}}}]$
4	Fully connected	256	Action size	/	Uniformly sampled within $[-3 \times 10^{-3}, 3 \times 10^{-3}]$

The model is almost the same as the one I have for my Reacher project. The only difference is that no batchnorm layer is added for the actor model here.

Part 2.2 General Hyperparameters

Hyperparameter	Value	Usage	Reason for choosing
Buffer size	10^5	Size of the replay buffer	Referenced nunesma's github.
Batch size	128	Size of each sample from the replay buffer	Default value of the DDPG implementation of Udacity drlnd
Gamma	0.95	Discount factor	Referenced gcolmen's GitHub
Tau	10^{-3}	Coefficient for soft target updates	Value suggested by section 7, DDPG Paper
LR_ACTOR	10^{-4}	Learning rate for the actor network	Referenced gcolmen's GitHub
LR_CRITIC	10^{-3}	Learning rate for the critic network	Referenced gcolmen's GitHub
WEIGHT_DECAY	0	Weight decay for critic network	Suggested by this discussion (require Udacity account to access)
TRAIN_EVERY	20	The agent(s) will learn every TRAIN_EVERY timesteps	Suggested by Udacity benchmark implementation
N_LEARN_UPDATES	10	The agent(s) will update N_LEARN_UPDATES each learning process	Suggested by Udacity benchmark implementation
Theta	0.15	Parameter for Ounoise	Referenced nunesma's github.
Sigma	0.2	Parameter for Ounoise	Referenced nunesma's github.

These Hyperparameters are shared between the self-play method and maddpg method. The hyperparameters almost remain unchanged with respect to the one I have for my Reacher project.

Part 3 Algorithm comparison and important things to note

Part 3.1 Basic performance of self-play and maddpg

The self-play method solves the environment much faster than the maddpg method.

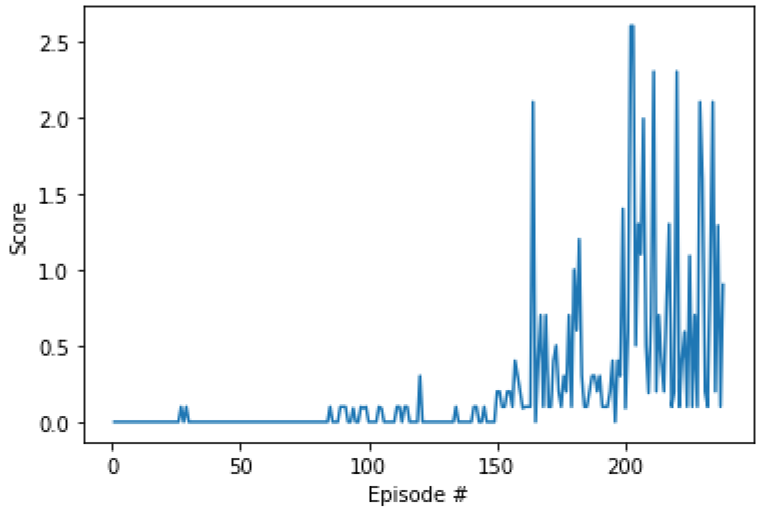

Figure 3.1-1: the score plot of the self-play method. The environment is solved within 238 episodes with self-play.

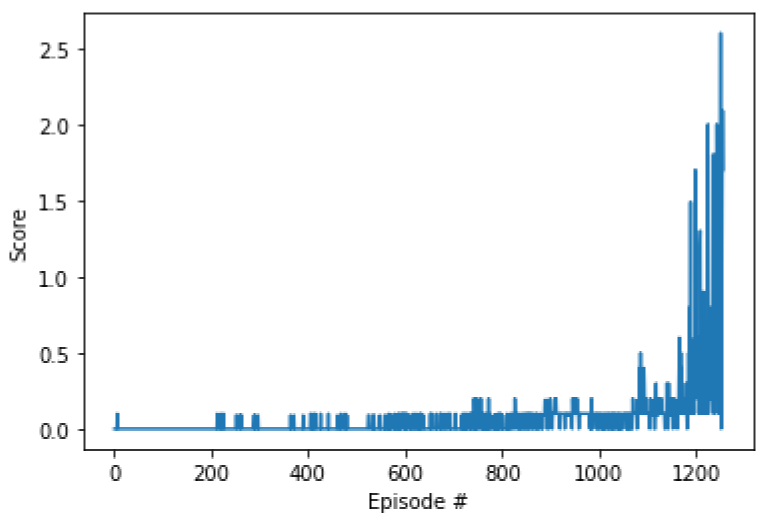

Figure 3.1-2: the score plot of the maddpg method. The environment is solved within 1225 episodes with maddpg.

Part 3.2 Performance of OUNoise and gaussian noise for self-play exploration

The OUNoise performs much better for the agents to explore the Tennis environment.

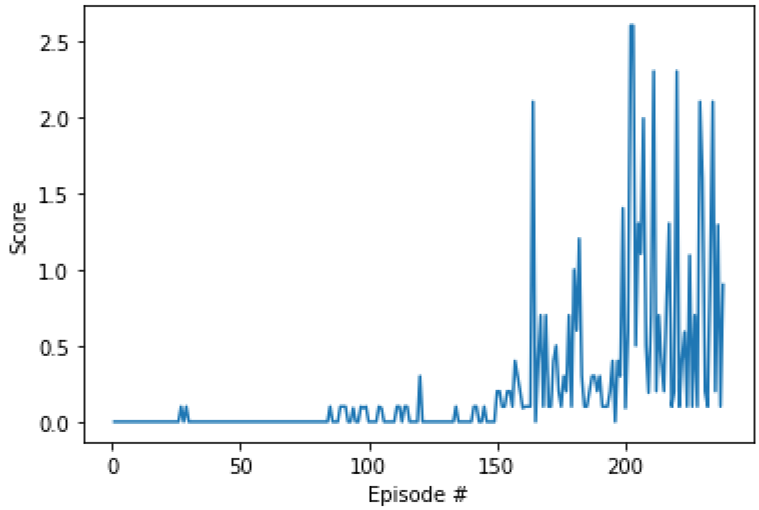

Figure 3.2-1: the score plot of the self-play method with OUNoise. The environment is solved within 238 episodes.

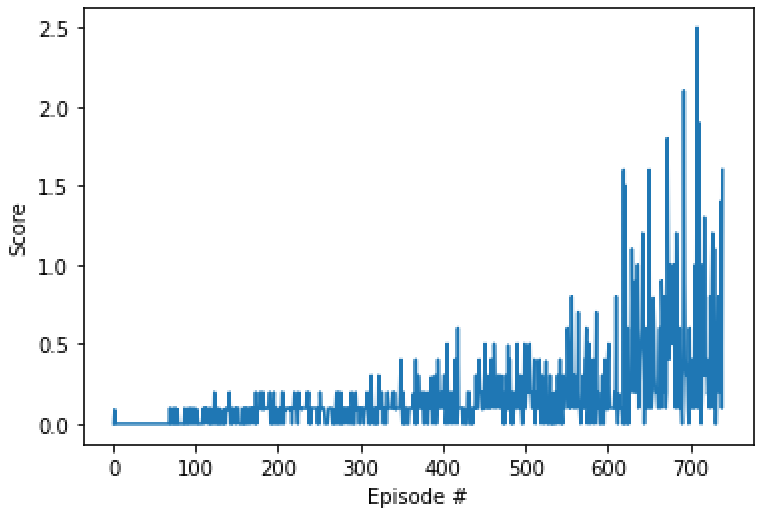

Figure 3.2-2: the score plot of the self-play method with gaussian noise. The environment is solved within more than 700 episodes.

Part 3.3 difference between “self_play” and “self-play-test” and the batch normalization problem

The only difference between self-play and self-play-test is that their ddpg agents use different ways to take actions.

Self-play:

```
def act(self, state, add_noise=True):
    """Returns actions for given state as per current policy."""
    state = torch.from_numpy(state).float().to(device)
    self.actor_local.eval()
    with torch.no_grad():
        action = self.actor_local(state).cpu().data.numpy()
    self.actor_local.train()
    if add_noise:
        action += self.noise.sample() * self.noise_coef#self.sigma * np.random.randn(self.action_size)
    return np.clip(action, -1, 1)
```

Self-play-test:

```
def act(self, state, add_noise=True):
    """Returns actions for given state as per current policy."""
    state = torch.from_numpy(state).float().to(device)
    self.actor_local.eval()

    '''
    with torch.no_grad():
        action = self.actor_local(state).cpu().data.numpy()
    self.actor_local.train()
    if add_noise:
        action += self.noise.sample() * self.noise_coef#self.sigma * np.random.randn(self.action_size)
    '''
    actions = []
    with torch.no_grad():
        for local_state in state:
            actions.append(self.actor_local(local_state).cpu().data.numpy())
    if add_noise:
        actions += self.noise.sample() * self.noise_coef
    return np.clip(actions, -1, 1)
```

The input state here is a pytorch tensor with size 2×24 (i.e. the local observation for both agents are stacked together). The “self-play” package directly use the state as input to the actor network and get action for both agents together. The “self-play-test” package uses one local observation (i.e. a 1×24 tensor) at a time.

This does not seem to be a big difference, but the “self-play-test” agent will fail in solving the Tennis environment if a batch-normalization layer is included in the actor network. The reason I believe that causes this phenomenon is discussed in “batchnorm_problem.ipynb” I wrote.

The same issue occurs when using my maddpg implementation. If I use the batch-normalization layer in my actor model, the agent will fail in solving the environment.

```
def forward(self, state):
    """Build an actor (policy) network that maps states -> actions."""
    #x = F.relu(self.batchnorm_1(self.fc1(state)))
    if state.dim() != 1:
        x = F.relu(self.batchnorm_1(self.fc1(state)))
        #x = F.relu(self.fc1(state))
    else:
        x = F.relu(self.fc1(state))


    x = F.relu(self.fc2(x))
    return F.tanh(self.fc3(x))
```

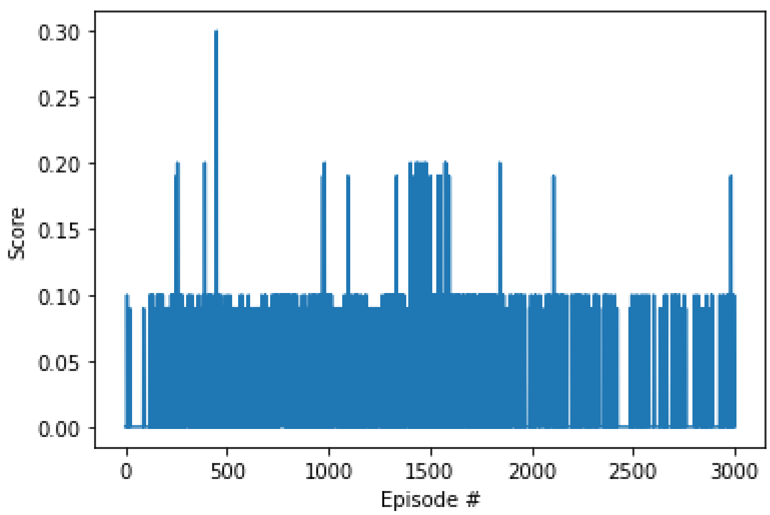

Figure 3.3-1: the result for adding batch normalization layer in actor network for maddpg. The maddpg agent cannot solve the environment.

Part 4 Future improvement discussion

For this project, I paid much emphasis on finding out the problem with respect to the batch normalization layer. For future improvements, it may be a good choice for me to try the param-noise method, which adds noise in neural-network parameters. Also, implementing other actor-critic algorithms may also be a good idea.