Collaboration Competition Report

This report includes the description of the implementation of the Udacity Reinforcement Learning Collaboration Competition project.

The problem

1. Environment: This project contains two agents in the Unity ML-Agents Tennis environment.
2. Agents: two agents control rackets to bounce a ball over a net. The goal of each agent is to keep the ball in play.
3. Actions: Each agent can move horizontally and vertically. Each action has 2 numbers in the range between -1 and 1 corresponding to the 2 dimensions.
4. States: States include observations at 3 consecutive time steps for each of the 2 agents. Each agent receives its observation of positions and velocities of itself and the ball represented by 8 components. Each episode has multiple time steps, and a state vector of 2*24 is generated at each step. Each row of the state vector is for an agent, and represents positions and velocities of the agent and the ball in a rolling window of 3 steps: two previous time steps and the current time step. The observation at each step includes 8 components: first 2 components are the 2-d coordinates of the position of the racket where the coordinate center is the net, the components 3 and 4 are the velocity coordinates for the ball, the components 5 and 6 are the coordinates of the ball, the components 7 and 8 are the same as components 3 and 4. When components 1 and 2 are close to components 5 and 6, the racket hits the ball and the ball starts to change its direction.
5. Rewards: The task is episodic. After the ball is hit back and moves over the net, if it's expected to hit within bounds, a reward 0.1 is given to the agent. If the ball hits the ground or is expected to be out of bounds, a reward if -0.01 is given to the agent. There is no limit of rounds for each episode if agents can keep the ball in play.
6. Done: the ball stops.
7. Goal: After each episode, we add up the rewards that each agent received (without discounting), to get a score for each agent. This yields 2 (potentially different) scores. We then take the maximum of these 2 scores. This yields a single score for each episode. In order to solve the environment, agents must get an average score of +0.5 (over 100 consecutive episodes, after taking the maximum over both agents).

Learning Algorithm, hyperparameters, model architectures

1. This project uses Deep Deterministic Policy Gradients (DDPG) algorithm, an example of Actor-Critic methods. The Actor-Critic method combines the advantages of actor-only (Policy-based) and critic-only (Value-based) methods. In this method, the critic learns the value function and uses it to determine how the actor's policy parameters should be changed. The actor brings the advantage of computing continuous actions without the need for optimization procedures on a value function, while the critic supplies the actor with knowledge of the performance. Actor-critic methods usually have good convergence properties, in contrast to critic-only methods.
2. An agent is created with both Actor and Critic networks built respectively with deep learning (DL) models.

3. The learning algorithm is to train the weights for all layers of the Actor network and the Critic network to find a policy to optimize the action taken at each step to maximize the score of an episode. The model is trained with a mini batch containing a given number of samples.
4. For each step in an episode, from the current states, the agents decide the actions. After actions are taken, the environment generates the next states, rewards and whether the episode is done or not.
5. Each of the agent will store the data in memory and will be trained if update is turned on.
6. The Critic loss function is the mean squared error (MSE) of two Q-value (action-value function) results. After the next actions are generated from the Actor model with next states, one of Q-value results is the predicted Q-value generated by the target Critic DL model using next states and next actions as the inputs and then adjusted by discounting factor Gamma and added by rewards. The other one of the Q-value results is the expected Q-value generated by the local Critic DL model using the current states as the inputs. The local Critic model is trained and updated by using back propagation on the loss function.
7. For the Actor model here, the predicted actions are generated with the Actor model and the current states. Then the Actor loss function is the negative mean of the value generated by the Critic model with the current states and predicted actions. Actor model parameters are trained by using this Actor loss function.
8. DDPG algorithm formulae:
   Loss function for the Critic Model for $Q(\mathrm{s}_i, \mathrm{a}_i; \theta)$ that generates action value Q:

$$L(\theta) = E\{(\mathrm{r}_i + \gamma * Q'(\mathrm{s}_{i+1}, \mu'(\mathrm{s}_{i+1}; \bar{w}); \bar{\theta}) - Q(\mathrm{s}_i, \mathrm{a}_i; \theta))^2\}$$

$$\theta = \theta - \alpha * \nabla_\theta L$$

   Update the Actor policy using the sampled policy gradient for $\mu(\mathrm{s}_i; w)$ that generates action. Critic model $Q(\mathrm{s}_i, \mathrm{a}_i; \theta)$ is used to evaluate the action:

$$\nabla_w J \approx \frac{1}{N} \sum_i \nabla_a Q(s, a; \theta) \mid_{\mathrm{s}=\mathrm{s}_i, a=\mu(\mathrm{s}_i)} \nabla_w \mu(\mathrm{s}_i; w)$$

$$w = w + \alpha * \nabla_w J$$

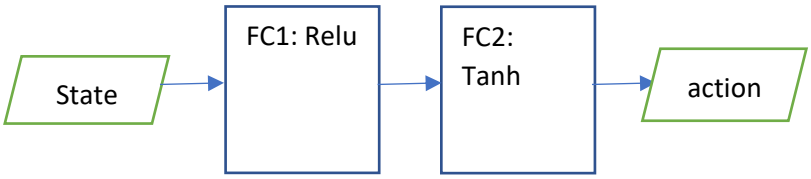
   Update the target networks:

$$\bar{\theta} = \tau * \theta + (1 - \tau) * \bar{\theta}$$
$$\bar{w} = \tau * w + (1 - \tau) * \bar{w}$$

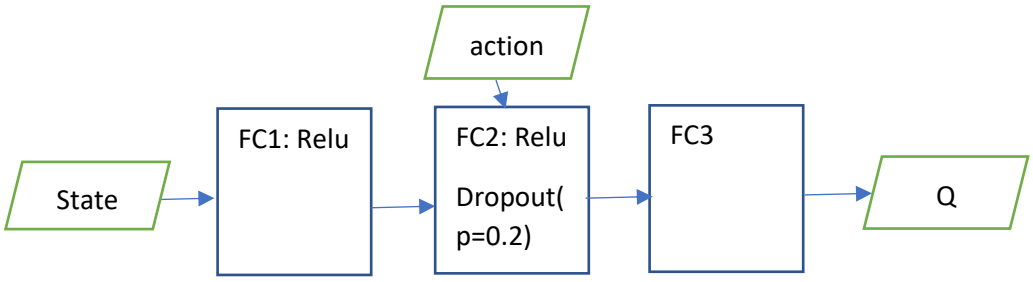
9. In order to avoid the situation with local maximum, a noise generated by the autoregressive stochastic Ornstein-Uhlenbeck process with normal distribution is used to add randomness to the action results generated by Actor model. Each agent gets its own noise for its action. The weight of the noise is controlled by Epsilon with a decay. The impact of noise will decay gradually. Because of the noise is from a normal distribution with mean 0, lots of noise samples have values close to 0.
10. Actions need also to be clipped to be in the range between -1 and 1. The clipping is used after the noise is added.
11. When training for Critic model, it needs to use the in place function clip_grad_norm_ to prevent the gradients from growing too large.
12. The hyperparameters are:

BUFFER_SIZE = int(1e5)	replay buffer size
BATCH_SIZE = 256	minibatch size
GAMMA = 0.99	discount factor
TAU = 1e-2	for soft update of target parameters
LR_ACTOR = 1e-4	learning rate of the actor
LR_CRITIC = 1e-3	learning rate of the critic
WEIGHT_DECAY = 0	L2 weight decay
UPDATE_EVERY = 2	Learning frequency
UPDATE_TIMES = 1	Times to repeat when learning
Actor fc1_units=256	Actor network hidden layer dimension
Critic fc1_units=256, fc2_units=128	Critic network hidden layer dimension
EPSILON_START = 1	Coefficient for noise
EPSILON_DECAY = 1e-5	Coefficient decay for noise
Sigma = 0.5	Standard deviation of noise with normal distribution: mean=0

13. The DL model architectures for the Actor (Policy-based) neural networks are 2 fully connected (FC) layers shown in the chart below. BatchNorm1d for the first layer is not used. The weights are stored in the checkpoint_actor.pth file.
    The number of units in the hidden layer is 256.
    FC1: nn.Linear(24, 256)
    FC2: nn.Linear(256, 2)

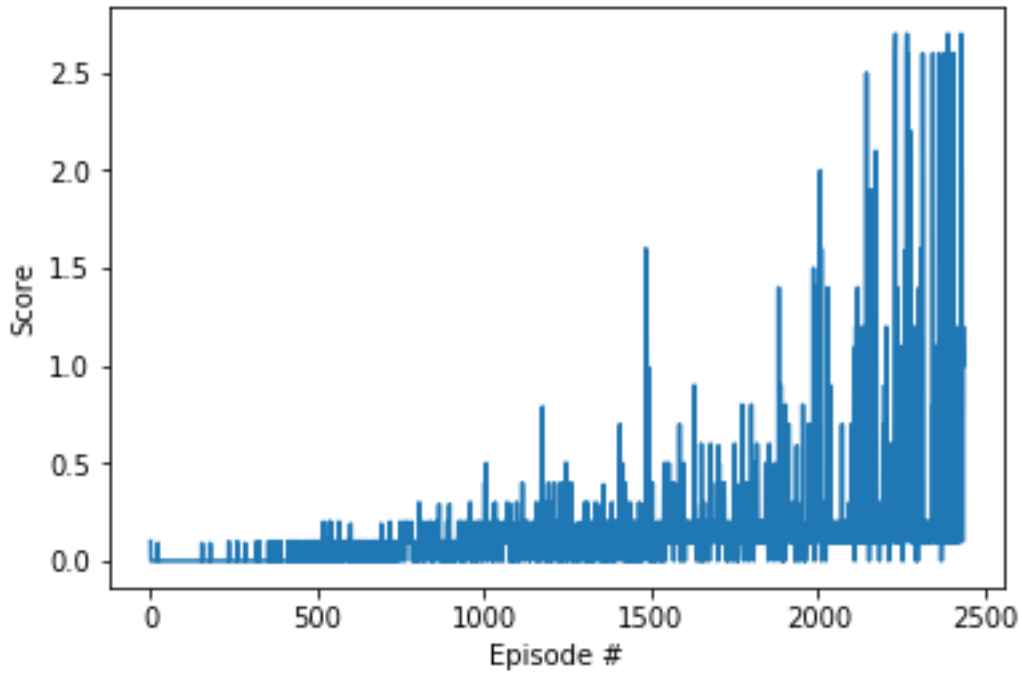


14. The DL model architectures for the Critic (Value-based) neural networks are 3 fully connected (FC) layers shown in the chart below. BatchNorm1d for the first layer is not used. The weights are stored in the checkpoint_critic.pth file.
    The numbers of units in the hidden layers are 256 and 128.
    FC1: nn.Linear(24, 256)
    FC2: nn.Linear(256+2=258, 128)
    FC3: nn.Linear(128, 1)



Plot of Rewards

The plot of rewards per episode illustrates that the agents are able to receive an average reward (over 100 episodes) of at least 0.5. The number of episodes needed to solve the environment is 2435 with average reward 0.51.



Ideas for Future Work for improving the agent's performance

1. Use Multi-Agent Deep Deterministic Policy Gradients (MADDPG) algorithm.
2. Use Trust Region Policy Optimization (TRPO) and Truncated Natural Policy Gradient (TNPG).
3. Use Proximal Policy Optimization (PPO).
4. Use Distributed Distributional Deterministic Policy Gradients (D4PG) algorithm.
5. Use hyperparameter optimization packages such as Optuna to find the optimal sets of hyperparameters.

References

1. Udacity Deep Reinforcement Learning Nanodegree Program: 3. Policy-based Methods and 4. Multi-Agent Reinforcement Learning https://www.udacity.com/
2. https://github.com/udacity/deep-reinforcement-learning/tree/master/ddpg-pendulum
3. CONTINUOUS CONTROL WITH DEEP REINFORCEMENT LEARNING https://arxiv.org/abs/1509.02971
4. https://github.com/ShangtongZhang/DeepRL
5. Unity-Technologies/ml-agents: Unity Machine Learning Agents Toolkit (github.com)