Tennis Collaboration – Project 3

Problem description

The Tennis environment to solve is a collaborative multi-agent problem where two agents must keep a ball in play by hitting it with their rackets to each other over a net. If an agent hits the ball it receives a reward of +0.1. However, if the agent misses the ball, therefore letting it fall to the ground or hits the ball out of play, it receives a negative reward of -0.01. The goal is for both agents to work together to keep the ball in play for as long as possible. The environment is considered solved when the average score over 100 episodes is at least +0.5.

Observation space

The observation space is a matrix of 2 x 24 variables. Each row is the observation for each agent, so each agent receives its own local observation of the environment. The observation consists of variables corresponding to position and velocity of the ball and racket.

Action space

The action space for the environment is a matrix of size [2 x 2], where the first row is the action taken by the first agent and the second row the action taken by the second agent. Each action corresponds to horizontal and vertical movement in a continuous space. Horizontal movement represents moving towards or away from the net and vertical corresponds to jumping. Every entry in the action vector should be a number between -1 and 1.

Proposed approach

Following the Udacity proposed approach I have elected to use a multi agent DDPG algorithm to solve the environment, due to the continuous nature of the observation and action space.

The rest of the report covers a brief overview of the DDPG method. Then an overview of the chosen network is presented followed by the results achieved. A conclusion with ideas about future work is presented at the end.

Deep Deterministic Policy Gradient: DDPG

Combining the idea of Actor-Critic and Deep Q Networks, Deep Deterministic Policy Gradient builds upon regular DQN learning. It learns a Q-function to estimate the action value for a given state and action and it learns a policy. In the same way as DQN, if the optimal action-value function is known, then for a given state the optimal action can be chosen.

The DQN algorithm cannot learn in an environment with a continuous action space as it would not be feasible to find the maximum Q-value for each action [1]. DDPG answers this by having four networks as opposed to DQN's two: two for the Actor (local and target networks) and two for the Critic (local and target networks). The Actor learns the continuous actions while the Critic learns the Q-function. When the Actor outputs a set of actions, the Critic network is used to select the actions as the role of the Critic is to output the expectation of the long-term reward [2].

The four networks that make up a DDPG agent are:

1) Actor local: takes an observation and outputs an action.
2) Actor target: Used in training to output actions. Helps with stability of learning as the weights are not quite the same as those of the Actor local network.

3) Critic local: takes and observation and an action and outputs the expected reward.
4) Critic target: Used in training to output expected reward. Helps with stability in the same way as the Actor target network.

Both Actor networks have the same architecture, the only difference are the weights. Prior to training the weights of both networks are identical but during training the target network is a blend of its original weights and the Actor local's weights. The Critic networks follow in the same way, both Critic networks are identical in architecture to each other and it is only the weights that differ over time that is the difference.

The DDPG algorithm also uses a replay buffer. This stores the previous experiences consisting of state, action chosen, reward received, next state and a Boolean indicating if the task is complete. This is sampled uniformly for a given batch size and this batch is used for learning. This makes DDPG an off-policy algorithm as it is learning from data generated from an older, outdated policy [1].

The learning algorithm for DDPG is as follows [2]:

1) Initialise the Actor local network with random weights.
2) Initialise the Actor target network and use the same weights as those of the Actor local.
3) Initialise the Critical local network with random weights.
4) Initialise the Critic target network and use the same weights as those of the Critic local.
5) For each training step:
    a. Pass the current environment observation to the Actor local to receive an action.
    b. Apply the action to the environment to receive a reward and the new state.
    c. Store the state, new state, action, reward, and Boolean task termination value in the experience replay buffer.
    d. Once there are enough experiences, sample from the buffer and use the sampled batch for training.
    e. Update the weights of both local networks by minimising the loss
    f. Use the tau interpolation parameter to update the weights of the target networks in the direction of the local networks.

To aid exploration of the environment at the beginning, and therefore avoid exploiting too early on, noise is added to the actions. Ornstein–Uhlenbeck process noise is added, but the amount added is reduced over time, therefore reducing exploration, and increasing exploitation as the algorithm learns [1].

Multi Agent DDPG Network

The multi DDPG agent used to solve this environment has two DDPG agents, where each DDPG agent has four networks with the following setup:

Actor local: (FC = fully connected layer)

- State size -> FC(size 64) -> Relu -> FC(size 128) -> Relu -> action size -> Tanh

Actor target has the same architecture.

Critic local: (CAT(x): concatenate x with input)

- State size -> FC(size 64) -> Relu -> CAT(action size) -> FC(size 128) -> Relu -> FC(size 64) -> Relu -> FC(size 1) -> output

Critic target has the same architecture.

To generate the actions the Actor accepts the state as input and uses the Tanh activation function squeeze its actions into the [-1, 1] range. The output of the Critic is a single value, signifying the Q-function value for taking the given action at the given state in its inputs.

To adapt the DDPG agent to the collaborative environment, a multi agent class was created that consisted of two separate DDPG agents, each with their own noise process and replay buffer. The state observed from the environment contains both observations from both agents, the agents received their own observations and output their own actions. These were concatenated and fed back to the environment to enact the action. The rewards received consisted of a reward per agent. The state, action and reward were all separated out for their respective agent and given back to each agent for them to store in their own replay buffer. The individual agents trained and learnt using their own replay buffer.

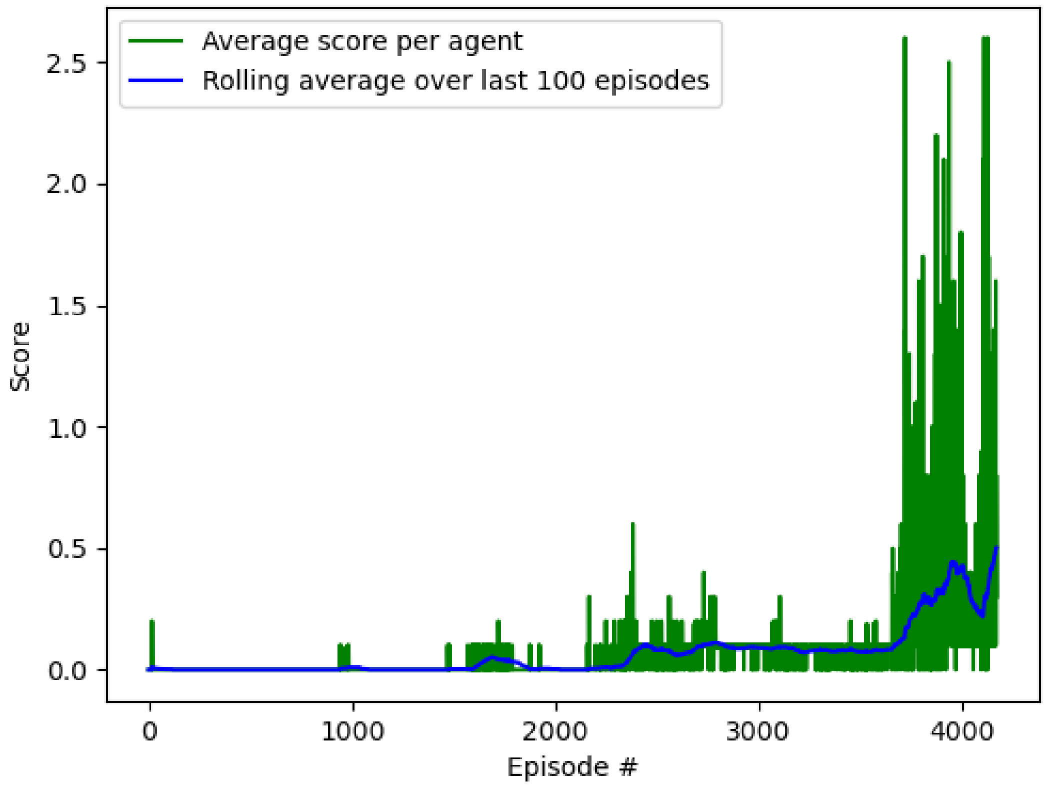
Hyperparameters

The following hyperparameters were used in the model architecture. Gamma, the reward discount was set to 0.95. The learning rate for the Actor networks was set to 1e-5 and the learning rate for the Critic networks was set to 1e-4. The replay buffer size was set to 1e6 and the sampling batch size was set to 64.

Results

Originally training using the mean squared error loss led to very slow training, which after 5000 episodes showed no signs of reaching the aim of +0.5. The Score had plateaued to around +0.25. Investigating the use of clipping the gradient showed some improvement but still did not get to the aim of +0.5. Reading further about loss functions [3] I chose to use the Smooth L1 loss as it seemed to be as well suited for the problem as well as being more robust to exploding gradients. Using this loss the multi DDPG agent was able to solve the environment.

After applying the multi agent DDPG algorithm the environment was solved in 4068 episodes with an average score of 0.502.



Conclusion and Future Work

The multi agent performed well but took longer to solve the environment than the DDPG example provided by Udacity. Future work to improve this would be to use look at a shared buffer and to tune the hyperparameters prior to looking to use any other method. Further on, Proximal Policy Optimisation (PPO) has been shown to be very effective in multi agent work as demonstrated by OpenAI5 and the work on Dota2 [4]. Investigation into MuZero [5], the latest Go playing agent, could be done to see if it can be adapted for collaborative multi agent learning instead of competitive learning, given its self-play aspect.

References

[1] Open AI Spinning Up, Deep Deterministic Policy Gradient - https://spinningup.openai.com/en/latest/algorithms/ddpg.html

[2] Mathworks, Deep Deterministic Policy Gradient Agents - https://uk.mathworks.com/help/reinforcement-learning/ug/ddpg-agents.html

[3] Pratyaksha Jha, A Brief Overview of Loss Functions in Pytorch – https://medium.com/udacity-pytorch-challengers/a-brief-overview-of-loss-functions-in-pytorch-c0ddb78068f7

[4] OpenAI, Dota 2 with Large Scale Deep Reinforcement Learning - https://arxiv.org/abs/1912.06680

[5] Deepmind, Mastering Atari, Go, Chess and Shogi by Planning with a Learned Model - https://arxiv.org/abs/1911.08265