Solving the Tennis Task

Marcel Kühn

Overview

The Tennis task is implemented in the Unity 3D framework and challenges an agent to train two players to maintain a tennis rally for as long as possible. A reward of +0.1 is granted to the player that manages to make the ball cross the net. If the player fails in the form of hitting the ball out of bounds or letting it drop to the ground, it receives a reward of -0.01. At the end of each episode, the maximum of the two rewards collected by the players is counted as the reward for the episode. Once the agent manages to collect an average of +0.5 per episode over the past 100 episodes, the task is considered to be solved.

The environment allows for two different actions, which are continuous variables and correspond to the movement of the tennis rack. These two possibilities define the action space of the agent and the output space of the actor network.

The player's decisions are based on their partially observed state. The state is a 24-dimensional vector for each player, that differ from each other since both experience the environment differently (partially). Since the MADDPG algorithm used here uses decentralized training for the actors but centralized training for the critic, the critic has access to both observed states and the actions taken, while the actor only has access to the state of the player it belongs to.

Learning Algorithm

The learning algorithm is based on the Multi-Agent Deep Deterministic Policy Gradient (MADDPG) algorithm. According to the the actor-critic paradigm, one network is responsible for picking the actions (actor) while the other one is needed to evaluate the current state (critic). Each player trains its own actor and uses its own observations of the environment as an input to these. However, both are using one centralized critic that is trained with both observations and both actions taken.

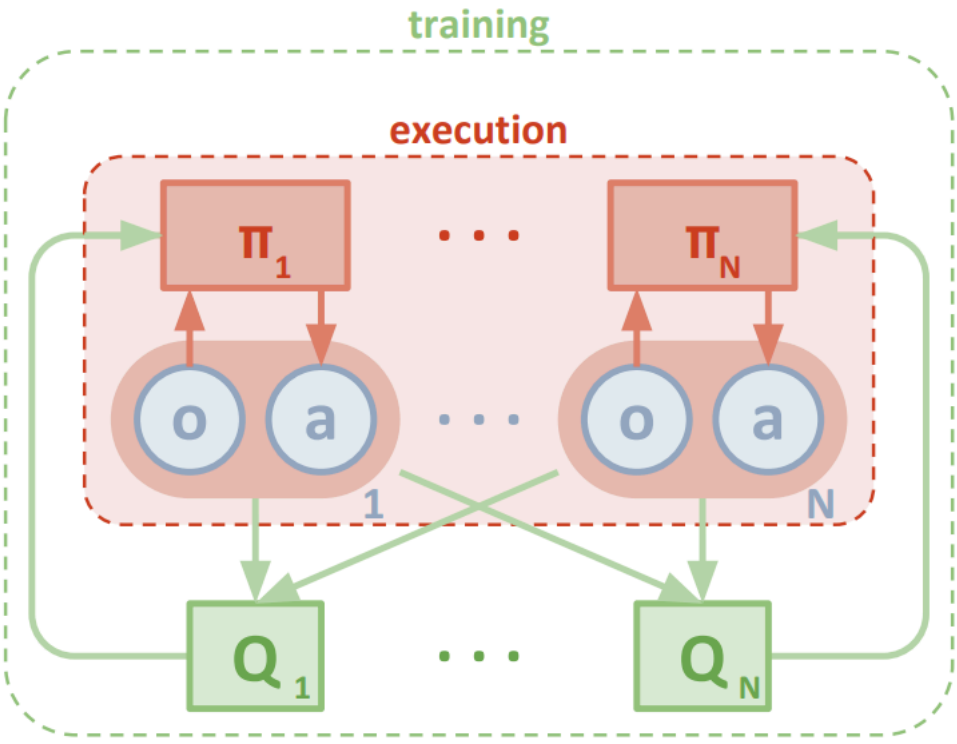
In order to optimize the networks, the algorithm works as follows:

In a first step, experience tuples that consist of the current state, the selected action, the following state and the resulting reward are collected and stored in a centralized replay buffer. The actions taken here can include some noise (sampled from a Ornstein Uhlenbeck Process in this implementation) to include some exploration. Then, periodically, samples are taken from the experience buffer and used for training.

The training then first updates the parameters of the critic network. This is done by calculating the target value (Q_target = reward + Q_expected_next_state * gamma) and comparing it to the current output of the network (Q_current). This resembles the training of a DQN network and utilizes the action, state, next state and reward from the given experience.

In the next step, the actor network is trained. To do this, the next actions are calculated by passing the next state from the experience to the actor network. Then, these next actions are passed into the critic network alongside the next state. Finally, the parameters of the actor network are maximized by backpropagating the output of the critic network and adjusting them accordingly.

The following figure outlines the algorithm schematically (from “Multi-Agent Actor-Critic for Mixed Cooperative-Competitive Environments“ by Ryan Lowe et al., 2017)



The architecture of the actor neural network consists of a 24-dimensional input layer (same dimension as the partial state space), two fully connected hidden layers of size 256 and a two-dimensional output layer (same dimension as the action space of each player). On the other hand, the critic network has an input layer of size 52 (both partial state spaces and both action spaces) and also two hidden layers of size 256. Furthermore, the output of the critic network is a scalar (the Q-value of the state). Both networks use batch normalization.

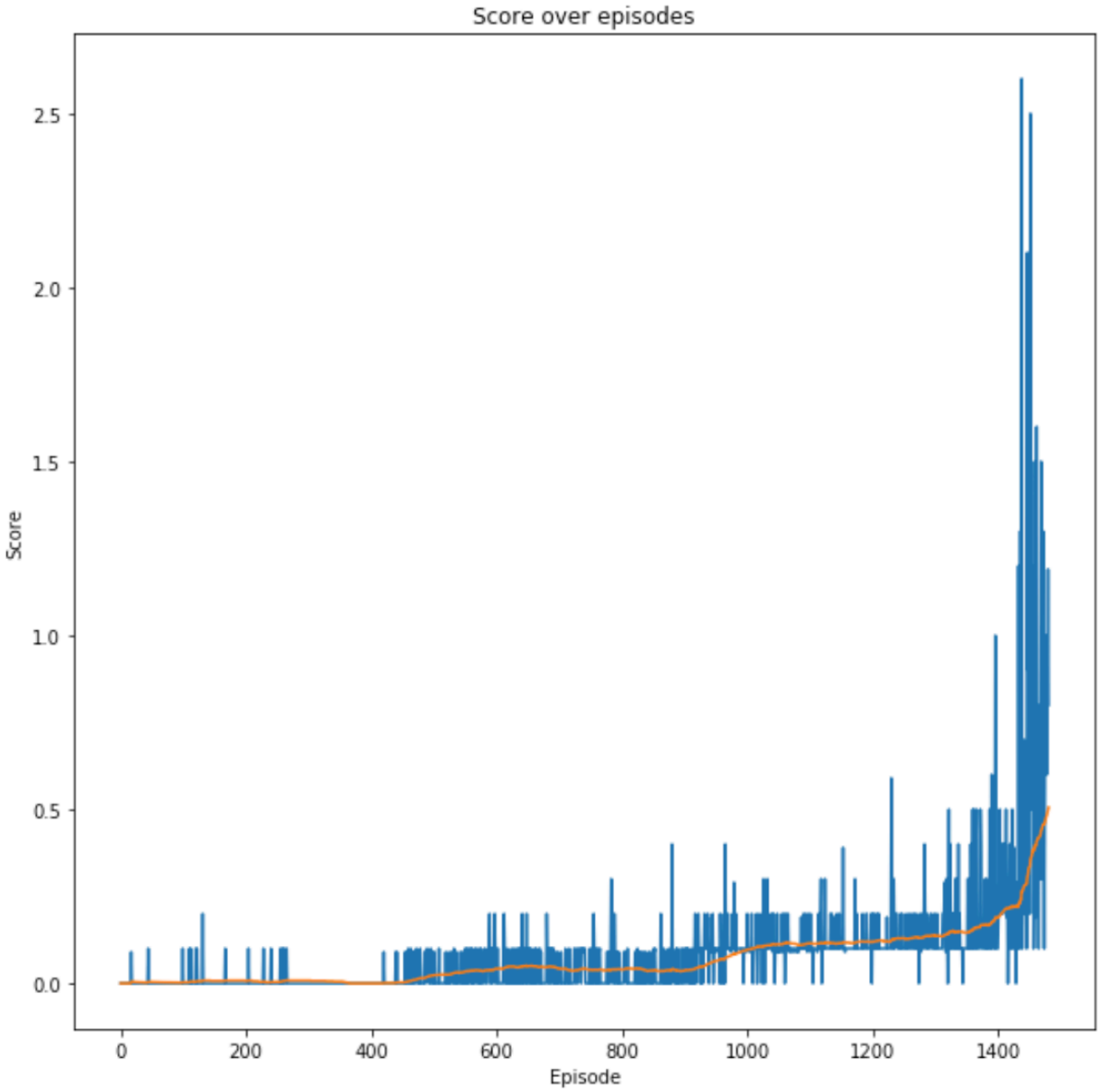
To stabilize the learning, two identical networks are initialized for both the actor and the critic, one on which the actual training happens (local network) and one which serves as a target network that is only updated once in a while by copying over the current parameter values from the local network (soft updating). Without this, the update rule for the parameters (weights) of the neural network are derived from the output of the network and thus its current weights. Having another neural network (target network) to calculate the output and then updating the weights of the local network according to this removes this correlation and improves the learning.

Overview of the Hyperparameters

LR_ACTOR	0.0005
LR_CRITIC	0.0005
TAU	0.001
WEIGHT_DECAY	0
BUFFER_SIZE	10000
BATCH_SIZE	256
GAMMA	0.99
UPDATE_EVERY	1
ADD_NOISE	True
NOISE_FACTOR_START	1.0
NOISE_REDUCTION	0.95
DIMENSION HIDDEN LAYERS	(256, 256)

Results

As it can be seen in the following plot, the algorithm proved to be able to solve the task in 1482 episodes.



Outlook

There are several different architectures for actor-critic deep reinforcement implementations such as A2C or A3C, which could improve the performance of the training. Furthermore, the hyperparameter tuning for this implementation was minimal and limited to the learning rate. A grid search on these could yield further performance gains. Also, a prioritized replay buffer has shown to improve the training performance.