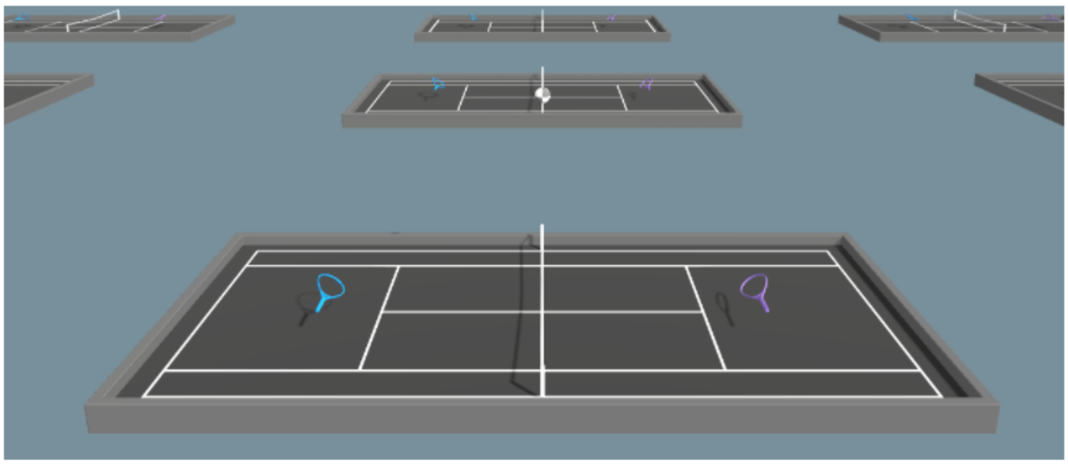
Project 3: Collaboration and Competition

Author: Ashraf Ibrahim

In my third Project for the Nanodegree *Deep Reinforcement Learning* (Deep Reinforcement Nanodegree) i tried to solve a *Tennis Environment*, based on the *Unity-Tennis-Environment*. I managed to solve this Environment with 736 episodes. In the following I will first explain, which characteristics this environment has, secondly i am going to explain my solution approach (MADDPG), thirdly i will talk about which experiences I made during all solution variants and finally which improvement possibilities and/or alternating solution options arise. The table of contents below provides an overview!

1. Introducing the Tennis Environment



(Source: Unity)

The Environment consists of two Agents (the Tennis-players),who have to play the ball to each other . which should be able to play Tennis with each other. The Goal is to play the ball over the net and for each successful attempt, the corresponding agent gets an reward of 0.1, but if an agent let's the ball hit the ground or hit's the ball out of bounds, he will receive an reward of -0.01. This is an episodic Task, where the rewards after an episode are going to be summed up for each agent. Obviously the action-space is of continuous nature, equivalent to the torque variables mentioned above.

To successfully complete this environment, an average of at least 0.5 points must be achieved in 100 consecutive episodes.

2. MADDPG

To Solve this Environment, i have used the a *Multi-Agent Deep Deterministic Policy Gradient* (short: MADDPG) -Approach. First i will briefly introduce the DDPG and then i will describe the MADDPG.

The DDPG is part of Actor-Critic-Family (A2C). Let's start with the basics of A2C and extend this by a description of the DDPG.

A basic A2C consists of two neurel Nets, whereas the first one - *the Policy-Net* or the *Actor* - calculates probabilities for Action a in State s and thus giving us Information on $\pi(a|s)$. The Second Net - *Value-Net* or *Critic* - approximates the Value Function $V(s)$ and thus gives us Information, on how *good* our Actions were.

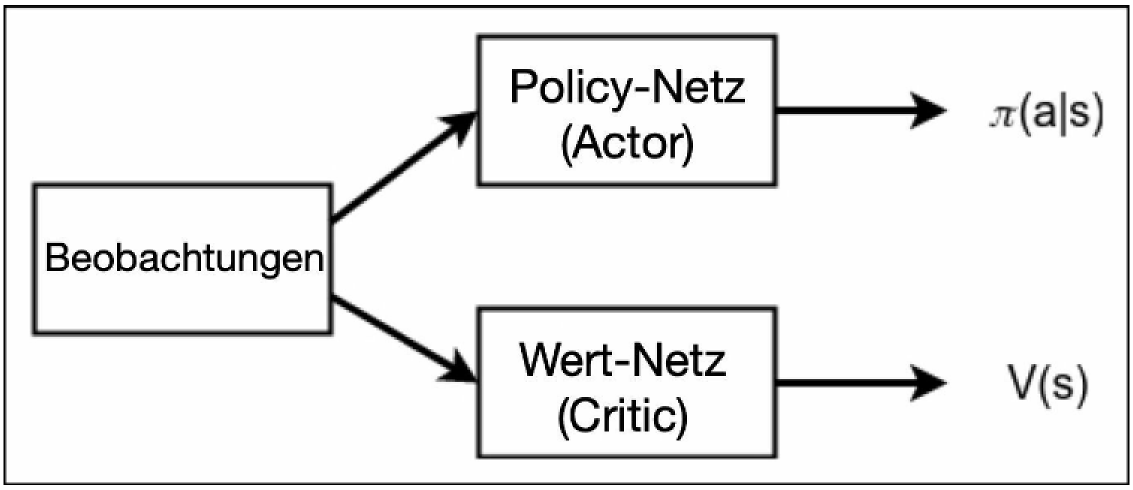


Abb. 12.6: Die A2C-Architektur

(Source: Lapan 2020:321).

In contrast to the above, the *Critic* in the *DDPG-Variant*, doesn't describe a Baseline for the current episode, instead it approximates the Q-value function based on the estimated actions of the *Actor*. Thus we are able to compute a Gradient. Another difference is, that - it is basically mentioned in the Name of the Algorithm - the DDPG handles Actions as deterministic and thus not learns the probability of an Action a given State s. The Algorithm learns the best Action for that State s.

In that manner, the *Actor* takes a State s as Input and outputs an Action s. This procedure is deterministic, so for every time a s_1 is given as Input, the *Actor* will always give the same a_1 as Output. The Critic, outputting the Q-Value, takes the calculated $s|a$ from the *Actor* as Input. So there are two functions available: The *Actor* function - $\mu(s;\theta_\mu)$, whereas $\mu(s)$ describes the *Actor* function and θ defines the weights for calculating the Gradient Ascent - and the *Critic* function - $Q(s,\mu(s;\theta_\mu);\theta_Q)$, which consists of the *Actor* function $\mu(s;\theta_\mu)$ under consideration of θ_Q given State s. Finally and taking into account the Chain-Rule, we are able to calculate the Gradient ascent: $\nabla aQ(s,a)\nabla_{\theta_\mu}\mu(s)$.

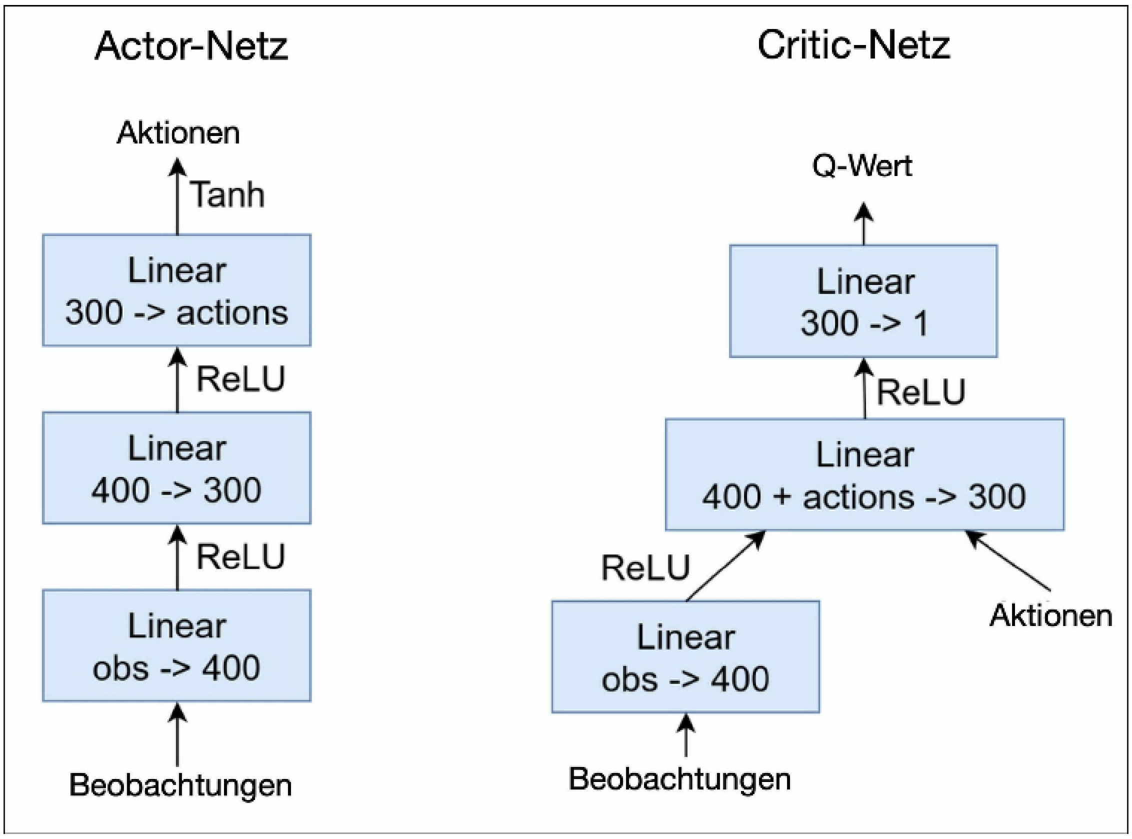


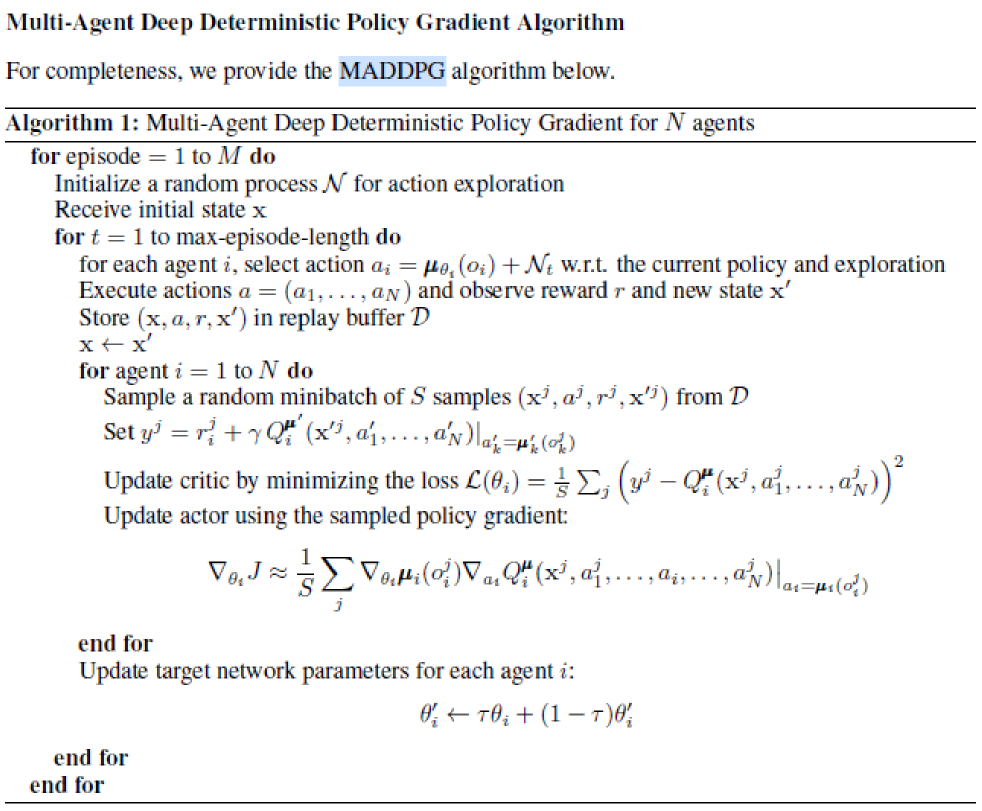
Abb. 17.4: Actor- und Critic-Netz beim DDPG-Verfahren

(Source: Lapan 2020:504).

Exploration is done via adding Noise to the actions of the *Actor*, before those are implemented in the Environment. This is done using the *Ornstein-Uhlenbeck-Process* which adds a random noise to each Action. DDPG also uses a *Replay-Buffer* and applies a *Soft-Update*. For a *Soft-Update* the *Actor* and the *Critic* are copied to calculate the target values. The weights of these *Target Networks* get a stepwise small update - like 0.01% - of the Weights from the *Learning Network* (Lillicrap et. al 2016:4).

A MADDPG is a multi-agent approach, where two or more Agents are interacting in the same Environment. A distinction here, can be made in the type of interaction between all agents: it can be *competitive*, where each agents tries to maximize its own reward or it can be *collaborative*, where all agents have to *work together* to get good results. Of Course mixed-types can occur as well (Lapan 2020: 757f.). The Tennis Environment can of course be played competitive or collaborative. This Variant is of collaborative Nature, as both Agent have to establish a stable game and playing the ball to each other to fulfill the rubric criteria. This is accomplished by learning a centralized critic based on observations of all Agents for all decentralized agents. This type of computation lets agents learn coordinative behaviors, to achieve their goals (Lowe et. al 2017: 7f.).

Thus the Algorithm is defined as:


Multi-Agent Deep Deterministic Policy Gradient Algorithm

For completeness, we provide the MADDPG algorithm below.

Algorithm 1: Multi-Agent Deep Deterministic Policy Gradient for N agents

```
for episode = 1 to $M$ do
  Initialize a random process $\mathcal{N}$ for action exploration
  Receive initial state $\mathbf{x}$
  for $t = 1$ to max-episode-length do
    for each agent $i$, select action $a_i = \boldsymbol{\mu}_{\theta_i}(o_i) + \mathcal{N}_t$ w.r.t. the current policy and exploration
    Execute actions $a = (a_1, \ldots, a_N)$ and observe reward $r$ and new state $\mathbf{x}'$
    Store $(\mathbf{x}, a, r, \mathbf{x}')$ in replay buffer $\mathcal{D}$
    $\mathbf{x} \leftarrow \mathbf{x}'$
    for agent $i = 1$ to $N$ do
      Sample a random minibatch of $S$ samples $(\mathbf{x}^j, a^j, r^j, \mathbf{x}'^j)$ from $\mathcal{D}$
      Set $y^j = r_i^j + \gamma Q_i^{\boldsymbol{\mu}'}(\mathbf{x}'^j, a'_1, \ldots, a'_N)|_{a'_k = \boldsymbol{\mu}'_k(o_k^j)}$
      Update critic by minimizing the loss $\mathcal{L}(\theta_i) = \frac{1}{S}\sum_j \left(y^j - Q_i^{\boldsymbol{\mu}}(\mathbf{x}^j, a_1^j, \ldots, a_N^j)\right)^2$
      Update actor using the sampled policy gradient:
          $\nabla_{\theta_i} J \approx \frac{1}{S}\sum_j \nabla_{\theta_i}\boldsymbol{\mu}_i(o_i^j)\nabla_{a_i} Q_i^{\boldsymbol{\mu}}(\mathbf{x}^j, a_1^j, \ldots, a_i, \ldots, a_N^j)|_{a_i = \boldsymbol{\mu}_i(o_i^j)}$
    end for
    Update target network parameters for each agent $i$:
          $\theta'_i \leftarrow \tau\theta_i + (1-\tau)\theta'_i$
  end for
end for
```

(Source: Lowe et. al 2017: 13.)

3 Implementation

My Implementation is based on four specific Python - Scripts:

```
├── Tennis.ipynb         # Major Script, runs the Agent
    ├── maddpg.py          # Implementation Multi-Agent DDPG
        ├── ddpg.py          # ddpg implementation
            ├──actorcritic.py  # Defining Actor and Critic
```

Let's start with the *Neurel-Nets* in the Background. As both performed really well in my second Project (See Report), i re-used them for this approach. Both Neural Nets consists of the following specifics:

```
class Actor(nn.Module):
    """Actor (Poicy) Model."""

    def __init__(self, state_size, action_size, seed, fc1_units=500, 
fc2_units=400):
        """Initialize parameters and build model.
        Params
        ======
            state_size (int): Dimension of each state
            action_size (int): Dimension of each action
            seed (int) : Random seed
            fc1_units (int) : Number of nodes in first hidden layer
            fc2_units (int) : Number of nodes in second hidden layer
        """
        super(Actor, self).__init__()
        self.seed = torch.manual_seed(seed)
        self.fc1 = nn.Linear(state_size, fc1_units)
        self.bn1 = nn.BatchNorm1d(fc1_units)
        self.fc2 = nn.Linear(fc1_units, fc2_units)
        self.fc3 =  nn.Linear(fc2_units, action_size)
        self.reset_parameters()

    def reset_parameters(self):
        self.fc1.weight.data.uniform_(*hidden_init(self.fc1))
        self.fc2.weight.data.uniform_(*hidden_init(self.fc2))
        self.fc3.weight.data.uniform_(-3e-3, 3e-3)

    def forward(self, state):
        """Build an actor(policy) network that maps states to actions"""
        x = F.relu(self.bn1(self.fc1(state)))
        x = F.relu(self.fc2(x))
        return torch.tanh(self.fc3(x))

class Critic(nn.Module):
    """Critic (Value) MOdel."""
    def __init__(self, state_size, action_size, seed, fc1_units=500,
fc2_units=400):
        """Initialize parameters and build model.
        Params
        ======
            state_size (int): Dimension of each state
            action_size (int): Dimension of each action
            seed (int): Random seed
            fcs1_units (int): Number of nodes in the first hidden layer
            fc2_units (int): Number of nodes in the second hidden layer
        """
        super(Critic, self).__init__()
        self.seed = torch.manual_seed(seed)
        self.fc1 = nn.Linear(state_size, fc1_units)
        self.bn1 = nn.BatchNorm1d(fc1_units)
        self.fc2 = nn.Linear(fc1_units+action_size, fc2_units)
        self.fc3 = nn.Linear(fc2_units, 1)
        self.reset_parameters()

    def reset_parameters(self):
        self.fc1.weight.data.uniform_(*hidden_init(self.fc1))
        self.fc2.weight.data.uniform_(*hidden_init(self.fc2))
        self.fc3.weight.data.uniform_(-3e-3, 3e-3)

    def forward(self, state, action):
        """Build a critic(value) network that maps (state,action) pairs to Q-
values"""
        x = F.relu(self.bn1(self.fc1(state)))
        x = torch.cat((x, action), dim=1)
        x = F.relu(self.fc2(x))
        return self.fc3(x)
```

Both Nets have the same architecture: An Inputlayer, followed by a batch Normalization, a fully connected hidden Layer with *relu*. The Nodes were set to 500 and 400 (this was done after experimenting a while; see next Chapter). The Second hidden Layer is different in both classes. The *Actor* takes the Output of 400 nodes and activates those with a *relu*, whereas the *Critic* takes the 400 Nodes and adds the possible Action size and activates those also with a *relu*. The *Output*

Layer for the *Actor* outputs the Action size, activated with a *tanh*, whereas the same Layer for the *Critic* outputs one Node, without an Activation (the Q-Value).

The *maddpg.py* functions as a wrapper for both Agents:

```
def __init__(self,state_size, action_size,random_seed):
        super(maddpg,self).__init__()

        self.state_size=state_size
        self.action_size=action_size
        self.seed = random.seed(random_seed)
        # As it was clear the the amount of Agent is limited to two, i
        # 'hardcoded' both into the class
        self.maddpg_agent=[Agent(state_size,action_size, random_seed),
                           Agent(state_size,action_size, random_seed)]
    ...
```

Since there are only two agents, I wrote it hardcoded. Of course this variant would not be recommended for multiple agents. I decided to choose an aggressive Learning and thus, initializing a Learning after every Game, based on 5 Samples.

```
# Hyperparams
BUFFER_SIZE = int(1e6)  # replay buffer size
BATCH_SIZE = 512        # minibatch size
LEARN_EVERY = 1        # Initialize Learning every 1. Game
LEARN_TIMES = 5        # update 5 times if larger than batch_size
GAMMA = 0.99            # discount factor
```

The DDPG, which is triggered by the *maddpg.py*, includes a Class for the *Agent*, a Class for the *Noise* calculation and a Class for the *Replay Buffer*.

Now let's inspect the Hyperparameters for the DDPG:

```
BUFFER_SIZE = int(1e6)  # replay buffer size
BATCH_SIZE = 256        # minibatch size
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        # weight decay

EPSILON = 1.0           # noise for exploration
EPSILON_DECAY = 0.99  # decay of exploration
```

I have chosen a more *aggressive* learning rate of 1e-3. I added an *Epsilon* value for the implementation of noise with a decay to slightly reduce the noise factor over time. I was inspired by Maxim Lapan's approach, which you can see underneath:

```
action += self.ou_epsilon * a_state
new_a_states.append(a_state)
```

(Source: Lapan 2020: 507).

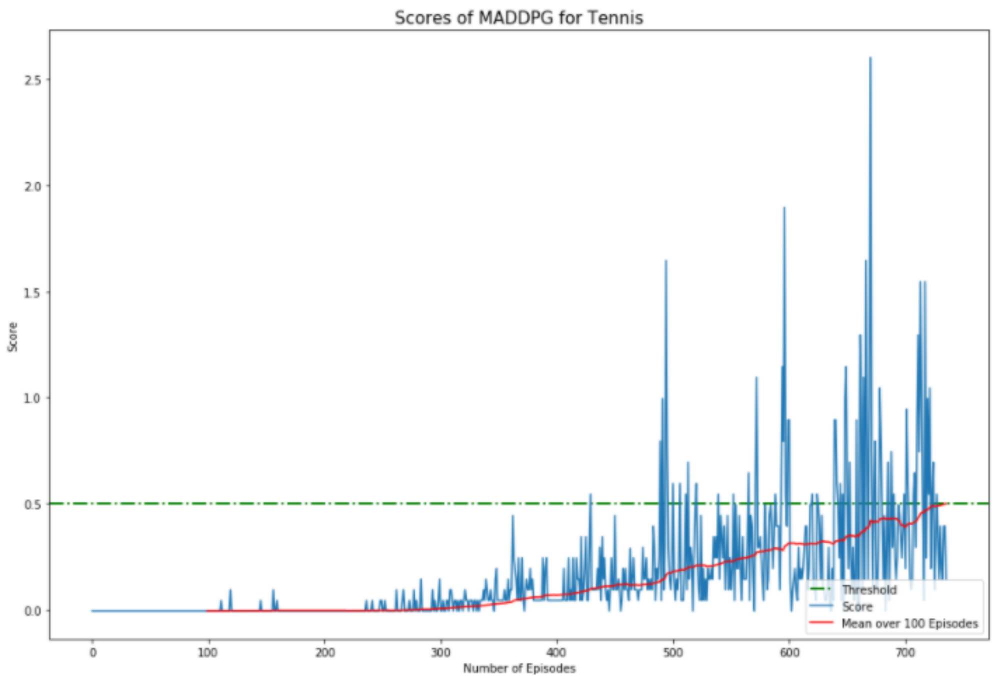
In Addition to my experience in my previous Project(See Report), i reduced the *epsilon decay* to be 0.99, which might me the reason for a more stable training!

3. Experiences during Training and Results

Fortunately, I already had some experience in dealing with the noise factor in the previous project. Nevertheless it took me some time to find the right hyperparameters.
A big difference was the frequency of learning: At first, I started learning every five episodes, but in the first 1000 episodes this did not show a constant improvement. Learning every two episodes made a more significant difference, but only if learning was done after each match, better rewards would be achieved faster. Of course this was affecting the calculation time of a single episode.

```
        # Learn, if enough samples are available in memory
        if len(self.memory) > BATCH_SIZE and times % LEARN_EVERY==0:
            # add agent iterator
            for agent in self.maddpg_agent:
                for _ in range(LEARN_TIMES):
                    experiences = self.memory.sample()
                    agent.learn(experiences, GAMMA)
```

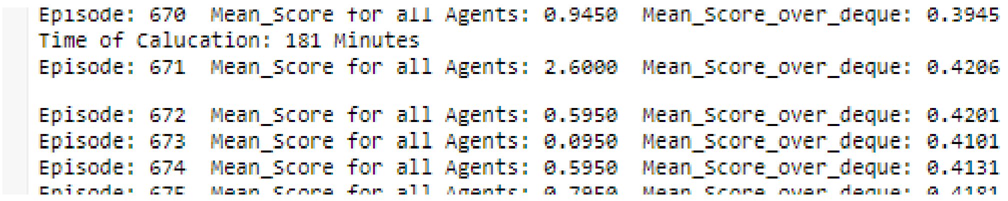
Both Agents did achieve good results really quick. Around the 300th Episode, the threshold of 0.5 was almost got crossed.



This did occur for the first time during Episode number 430, with 0.5450 Score-Points.

```
Time of Calucation: 43 Minutes
Episode: 426  Mean_Score for all Agents: 0.3450  Mean_Score_over_deque: 0.0915
Episode: 427  Mean_Score for all Agents: 0.0450  Mean_Score_over_deque: 0.0915
Episode: 428  Mean_Score for all Agents: 0.0450  Mean_Score_over_deque: 0.0915
Episode: 429  Mean_Score for all Agents: 0.2950  Mean_Score_over_deque: 0.0945
Episode: 430  Mean_Score for all Agents: 0.5450  Mean_Score_over_deque: 0.0995
Time of Calucation: 46 Minutes
```

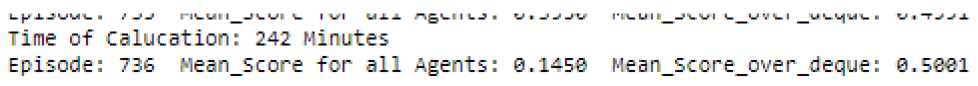
After that, there was a static development towards higher scores, with a frequent exceeding of the threshold value. The Highest Score achieved, was at Episode 671 with 2.6 Points.



```
Episode: 670  Mean_Score for all Agents: 0.9450  Mean_Score_over_deque: 0.3945
Time of Calucation: 181 Minutes
Episode: 671  Mean_Score for all Agents: 2.6000  Mean_Score_over_deque: 0.4206

Episode: 672  Mean_Score for all Agents: 0.5950  Mean_Score_over_deque: 0.4201
Episode: 673  Mean_Score for all Agents: 0.0950  Mean_Score_over_deque: 0.4101
Episode: 674  Mean_Score for all Agents: 0.5950  Mean_Score_over_deque: 0.4131
Episode: 675  Mean_Score for all Agents: 0.7950  Mean_Score_over_deque: 0.4181
```

Overall the Environment was solved at 736 Episodes, where the threshold over 100 Episodes of 0.5 was crossed and achieved a value of 0.5001.



```
Episode: 735  Mean_Score for all Agents: 0.5950  Mean_Score_over_deque: 0.4951
Time of Calucation: 242 Minutes
Episode: 736  Mean_Score for all Agents: 0.1450  Mean_Score_over_deque: 0.5001
```

4. Improvement Ideas

A Basic idea would be, to let the DDPG compete against the PPO and the A3C to compare Scores in relation to computational Time. But as i did achieve good Results with the *MADDPG*, i would tend to focus more on Hpyerparametertuning. There are several Hpyterparamteres, which can be tuned: The Parameters of the Neural Nets, and the Parameters for the Agents. For the Neural Nets, i would like to implement Tensorboard and HParams, to inspect several possibilities, like learning-rate, optimiziers etc.

Regarding the agents I would like to experiment again with the learning frequency. Maybe a similar good result can be achieved in less time.