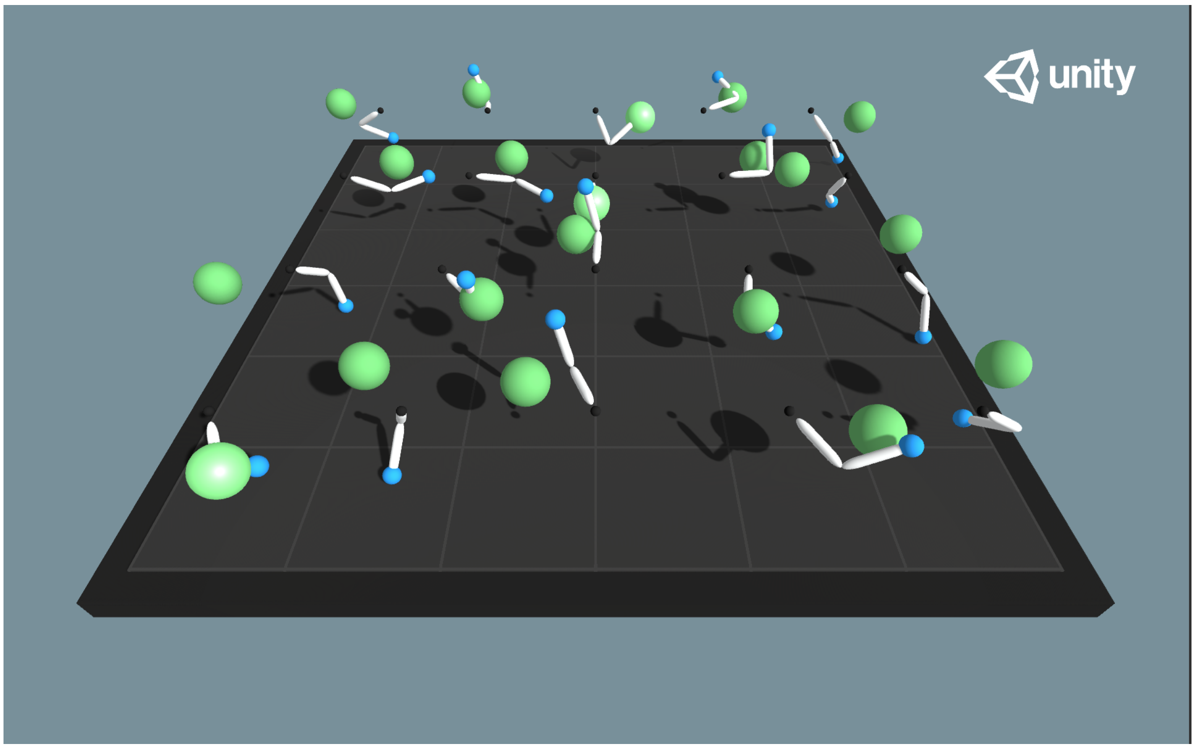
Project 2: Continous Control

Date: Ashraf Ibrahim

In my second Project for the Nanodegree *Deep Reinforcement Learning* (Deep Reinforcement Nanodegree) i tried to solve the *Unity Reacher Environment*. I managed to solve this Environment with a number of episodes. In the following I will first explain, which characteristics this environment has, secondly i am going to explain my solution approach (DDPG), 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 Reacher Environment



(Source: Unity)

The Environment consists of double-jointed arms (the agents), which can move to a target location (the green ball). The Goal is to reach the target location and to stay on target; for each step, where the Agent's hand is at the goal location, he receives an award of 0.1. In contrast to the Unity-Version (26 variables), the present Version has 33 Variables corresponding to position, rotation, velocity and angular velocity (of both arm-parts). 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 30 points must be achieved in 100 consecutive episodes. In this Analysis i have used the 20-Agent Environment.

2. DDPG

To Solve this Environment, i have used the a *Deep Deterministic Policy Gradient* (short: DDPG) - Approach. 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.

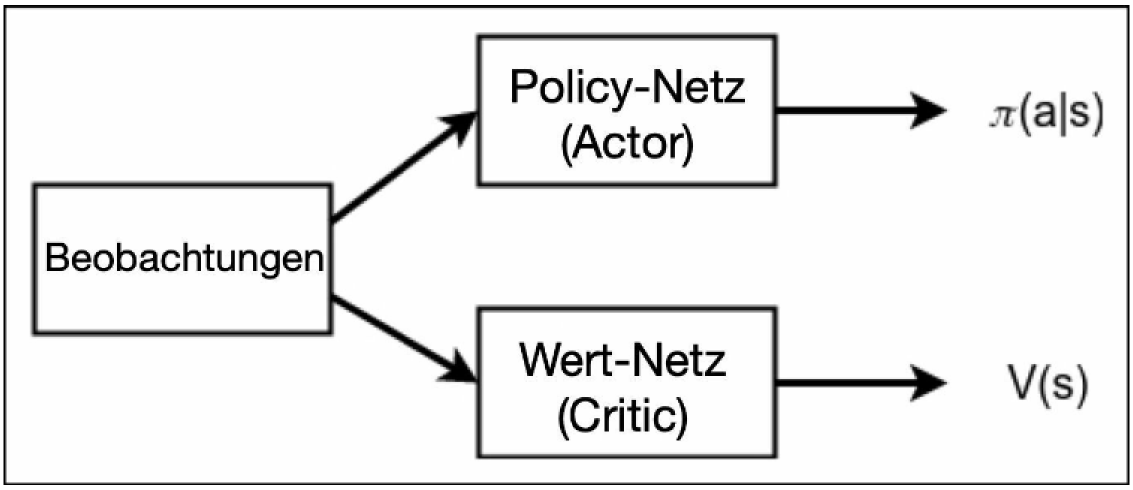


Abb. 12.6: Die A2C-Architektur

(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: $\nabla a Q(s,a) \nabla_{\theta_\mu} \mu(s)$.

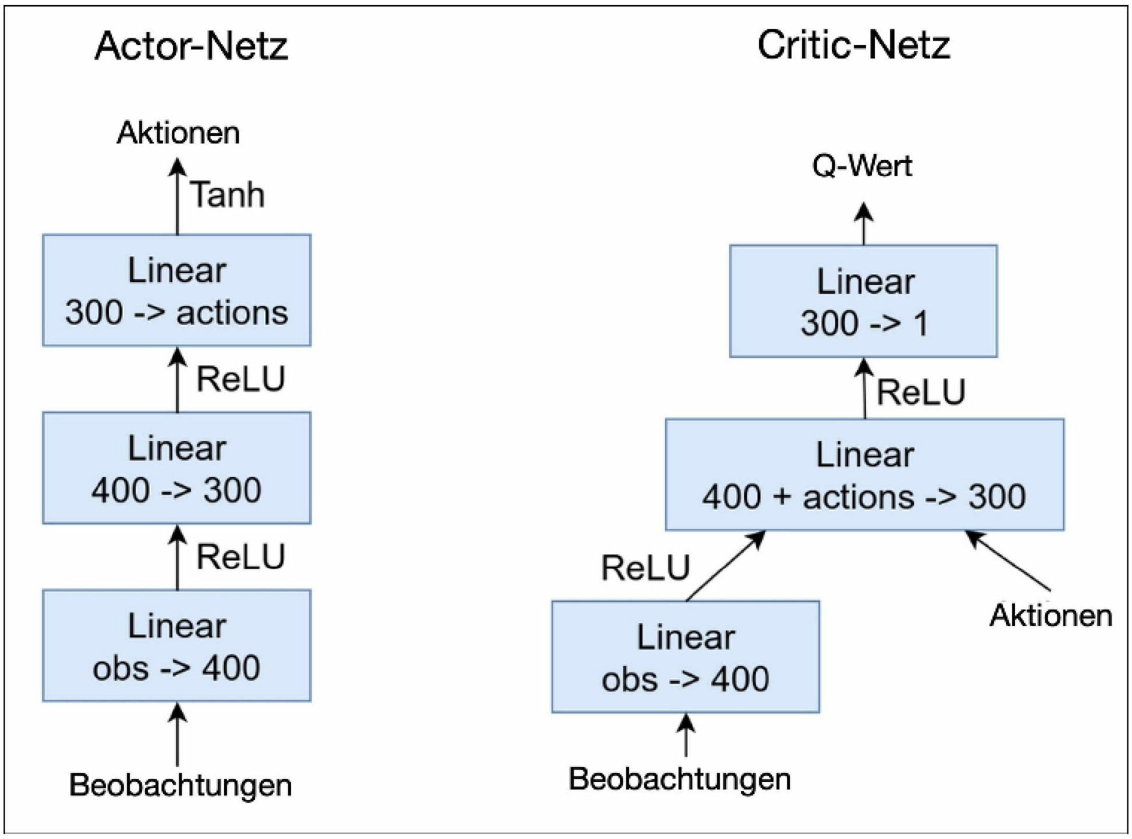


Abb. 17.4: Actor- und Critic-Netz beim DDPG-Verfahren

(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).

3 Implementation

My Implementation is based on three specific Python - Scripts:

```
├── Continous_Control.ipynb        # Major Script, runs the Agent
    ├── agent.py                   # Implementation of DDPG-Agent, with Actor and
Critic
        ├──actorcritic.py          # Defining Actor and Critic

```

Let's start with the *Neurel-Nets* in the Background.

```
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).

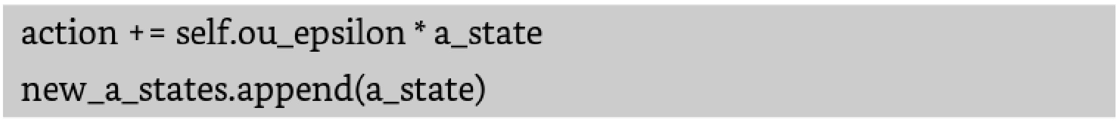
The *agent.py* is a definition of the DDPG Agent and includes a Class for the *Agent*, a Class for the *Noise* calculation and a Class for the *Replay Buffer*. The *Continous_Control.ipynb* initiates the Environment, starts the agents and evaluates its performance.

Now let's inspect the Hyperparameters:

```
BUFFER_SIZE = int(1e6)  # replay buffer size
BATCH_SIZE = 128        # minibatch size
GAMMA = 0.99            # discount factor
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

# Adding a decay and learning steps
LEARN_EVERY = 20        # Initialize Learning every 20 steps
LEARN_TIMES = 10        # update 10 times if larger than batch_size
EPSILON = 1.0           # noise for exploration
EPSILON_DECAY = 0.9999  # decay of exploration
```

At first i had *learning-rate* of 1e-4 - for *Actor* and for the *Critic* - but for my last approach i have chosen a more *aggressive* rate of 1e-3. I have added Hyperparameters to adjust when learning is initialized (*LEARN_EVERY*) and to for how many sampled Trjactory's a learning should take place (*LEARN_TIMES*) - accidentally i created a bug here, which did take some time to debug (stay tuned for the next Chapter :-). 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).

3. Experiences during Training and Results

As i mentioned above, i created a bug in my implementation, causing it to learn very slowly. After implementing the code - and basically seeing, that everything is running and functional - i increased the batch size from 256 to 512 - it seemed small to me on a second View (Note: i had learning rates of 1e-4 and nods of (300,200)). I than started to receive scores of 0. - something per episode for long period of time, without really seeing a steady increase.

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

I didn't relate this to the Batch size at first, but after some attentive code-reading, i realized, that my Agent only learns, after more than half of all timesteps per episodes are completed. The defined condition "len(self.memory = Trajectory's)> 512 and timesteps modulo 20 ==0", only got triggered in the late middle of an episode. Actually my intention was to find a good balance between learning and computation time, by starting after 256 Trajectory's, repeating learning after ervery 20 timesteps.

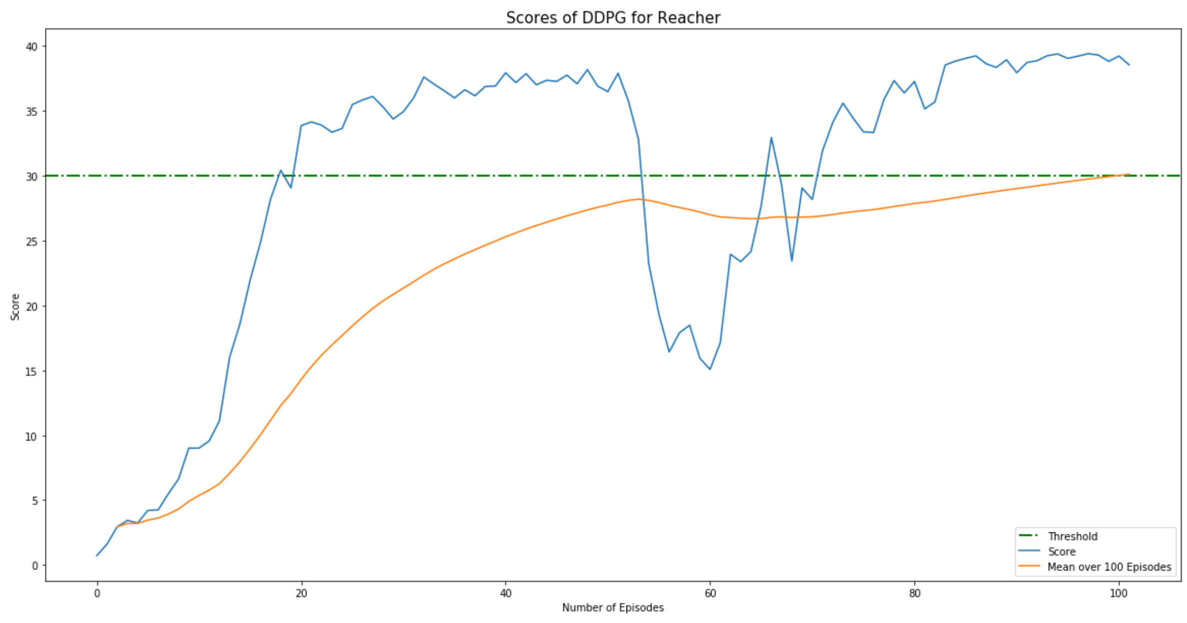
I reduced the Batch size to 128. which gave my a huge increase. Also i did increase the *learning-rate* to 1e-3 and increased the amount of Nodes to (500,400). With those Hyperparameters i could solve this Environment in 102 Episodes and a computation Time of about 5 Hours.

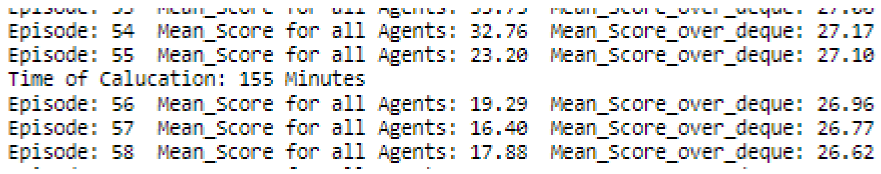
```
Episode: 91  Mean_Score for all Agents: 37.90  Mean_Score_over_deque: 28.37
Episode: 92  Mean_Score for all Agents: 38.68  Mean_Score_over_deque: 28.48
Episode: 93  Mean_Score for all Agents: 38.83  Mean_Score_over_deque: 28.59
Episode: 94  Mean_Score for all Agents: 39.21  Mean_Score_over_deque: 28.70
Episode: 95  Mean_Score for all Agents: 39.35  Mean_Score_over_deque: 28.82
Time of Calucation: 304 Minutes
Episode: 96  Mean_Score for all Agents: 39.00  Mean_Score_over_deque: 28.92
Episode: 97  Mean_Score for all Agents: 39.18  Mean_Score_over_deque: 29.03
Episode: 98  Mean_Score for all Agents: 39.37  Mean_Score_over_deque: 29.13
Episode: 99  Mean_Score for all Agents: 39.26  Mean_Score_over_deque: 29.24
Episode: 100  Mean_Score for all Agents: 38.77  Mean_Score_over_deque: 29.33
Time of Calucation: 322 Minutes

Episode: 101  Mean_Score for all Agents: 39.19  Mean_Score_over_deque: 29.72
Episode: 102  Mean_Score for all Agents: 38.50  Mean_Score_over_deque: 30.08

Environment solved in 102 episodes with a mean score of 30.084264327564277 - Finally :-) !
```

Looking into the Plot of the Scores, it shows an interesting development. After a steep increase in the first 10 episodes, the achieved scores became a bit unsteady but still increasing. This continued until episode 55, where the scores decreased intensively (Picture 2 shows the details).





```
Episode: 54  Mean_Score for all Agents: 32.76  Mean_Score_over_deque: 27.17
Episode: 55  Mean_Score for all Agents: 23.20  Mean_Score_over_deque: 27.10
Time of Calucation: 155 Minutes
Episode: 56  Mean_Score for all Agents: 19.29  Mean_Score_over_deque: 26.96
Episode: 57  Mean_Score for all Agents: 16.40  Mean_Score_over_deque: 26.77
Episode: 58  Mean_Score for all Agents: 17.88  Mean_Score_over_deque: 26.62
```

I am assuming, that this a consquence of my Epsilon-Value and my low Epsilon decay. The Epsilon-Value adds - as mentioned above - a gaussian noise to the *Actions*, whereas the Decay reduces this Noise over Time. Mayby with a higher Decay, the Agent wouldn't be so greedy. I assume, that the agent did identify a policy, which did function very well for the first 50 epsiodes and achieving around 30 score points really quick,but than - under unknown circumstances; unknown state-action pairs - the policy didnt function, so the Agent made *bad* decisions.

Nevertheless, the Agent did learn to accomplish those unknown Situations and did achieve good results again. Finally i did achieve a mean Score of 30.08 over 100 episodes at the 102 Episode.

4. Improvement Ideas

A general approach to improve the DDPG would be to use the *Distributed Distributional Deep Deterministic Policy Gradients* - short D4PG. The D4PG introduces several improvements: 1) changing one Q-Value of the *Critic* with a probability Distribution, 2) Reuse the Bellman-Equation for N-Following Steps and 3) the usage of a prioritized experience replay instead of a Replay Buffer with a steady sampling Probability (Lapan 2020:511).

Another idea would be, to let the DDPG compete against the PPO and the A3C to compare Scores in relation to computational Time.

And of course, i would like to do some Hyperparametertuning for the *Epsilon-Decay* to see, if i could get a more steady and robust growth!

Literature

Lapan, Maxim 2020: Deep Reinforcement Learning (German Edition).

Lillicrap et. al 2016 : Continuous Control with Deep Reinforcement Learning. Paper-link.