Solving the Reacher Continuous Control Task

Marcel Kühn

Overview

The Reacher Continuous Control task is implemented in the Unity 3D framework and challenges an agent to make and stay in contact with an object for as long as possible with its hand. A reward of +0.1 is granted for every time step the agents hand touches the object. In the chosen version 1 of this task, the goal of a single agent is to maximize the overall reward in an episode. Once the agent manages to collect an average of +30 per episode over the past 100 episodes, the task is considered to be solved.

The environment allows for four different actions, which are continuous variables and correspond to the forces applied to the two joints of the arm. These four possibilities define the action space of the agent and the output space of the actor network.

The agent's decisions are based on its current state. The state is a 33-dimensional vector that contains information about its own arm as well as the object (goal). This 33-dimensional vector serves as the input to the critic network, which is then concatenated with the output of the critic network after the first layer.

Learning Algorithm

The learning algorithm is based on the Deep Deterministic Policy Gradient (DDPG) algorithm in which the parameters of two neural networks are tweaked according to their loss function. 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).

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 replay buffer. The actions taken here should 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.

For a pseudocode implementation of the algorithm, see below (taken from "Continuous Control with Deep Reinforcement Learning" by Lillicrap et al., 2019).

Algorithm 1 DDPG algorithm

Randomly initialize critic network $Q(s, a|\theta^Q)$ and actor $\mu(s|\theta^\mu)$ with weights θ^Q and θ^μ.
Initialize target network Q' and μ' with weights $\theta^{Q'} \leftarrow \theta^Q$, $\theta^{\mu'} \leftarrow \theta^\mu$
Initialize replay buffer R
for episode = 1, M **do**
  Initialize a random process $\mathcal{N}$ for action exploration
  Receive initial observation state s_1
  for t = 1, T **do**
    Select action $a_t = \mu(s_t|\theta^\mu) + \mathcal{N}_t$ according to the current policy and exploration noise
    Execute action a_t and observe reward r_t and observe new state s_{t+1}
    Store transition (s_t, a_t, r_t, s_{t+1}) in R
    Sample a random minibatch of N transitions (s_i, a_i, r_i, s_{i+1}) from R
    Set $y_i = r_i + \gamma Q'(s_{i+1}, \mu'(s_{i+1}|\theta^{\mu'})|\theta^{Q'})$
    Update critic by minimizing the loss: $L = \frac{1}{N}\sum_i (y_i - Q(s_i, a_i|\theta^Q))^2$
    Update the actor policy using the sampled policy gradient:

$$\nabla_{\theta^\mu} J \approx \frac{1}{N} \sum_i \nabla_a Q(s, a|\theta^Q)|_{s=s_i, a=\mu(s_i)} \nabla_{\theta^\mu} \mu(s|\theta^\mu)|_{s_i}$$

    Update the target networks:

$$\theta^{Q'} \leftarrow \tau\theta^Q + (1-\tau)\theta^{Q'}$$
$$\theta^{\mu'} \leftarrow \tau\theta^\mu + (1-\tau)\theta^{\mu'}$$

  end for
end for

The architecture of the actor neural network consists of a 33-dimensional input layer (same dimension as the state space), two fully connected hidden layers of size 128 and 64 and a four-dimensional output layer (same dimension as the

action space). On the other hand, the critic network has the same architecture, the only difference is that the first hidden layer is of size 132 to allow for the input of the action space. Furthermore, the output of the network is a scalar (the Q-value of the state).

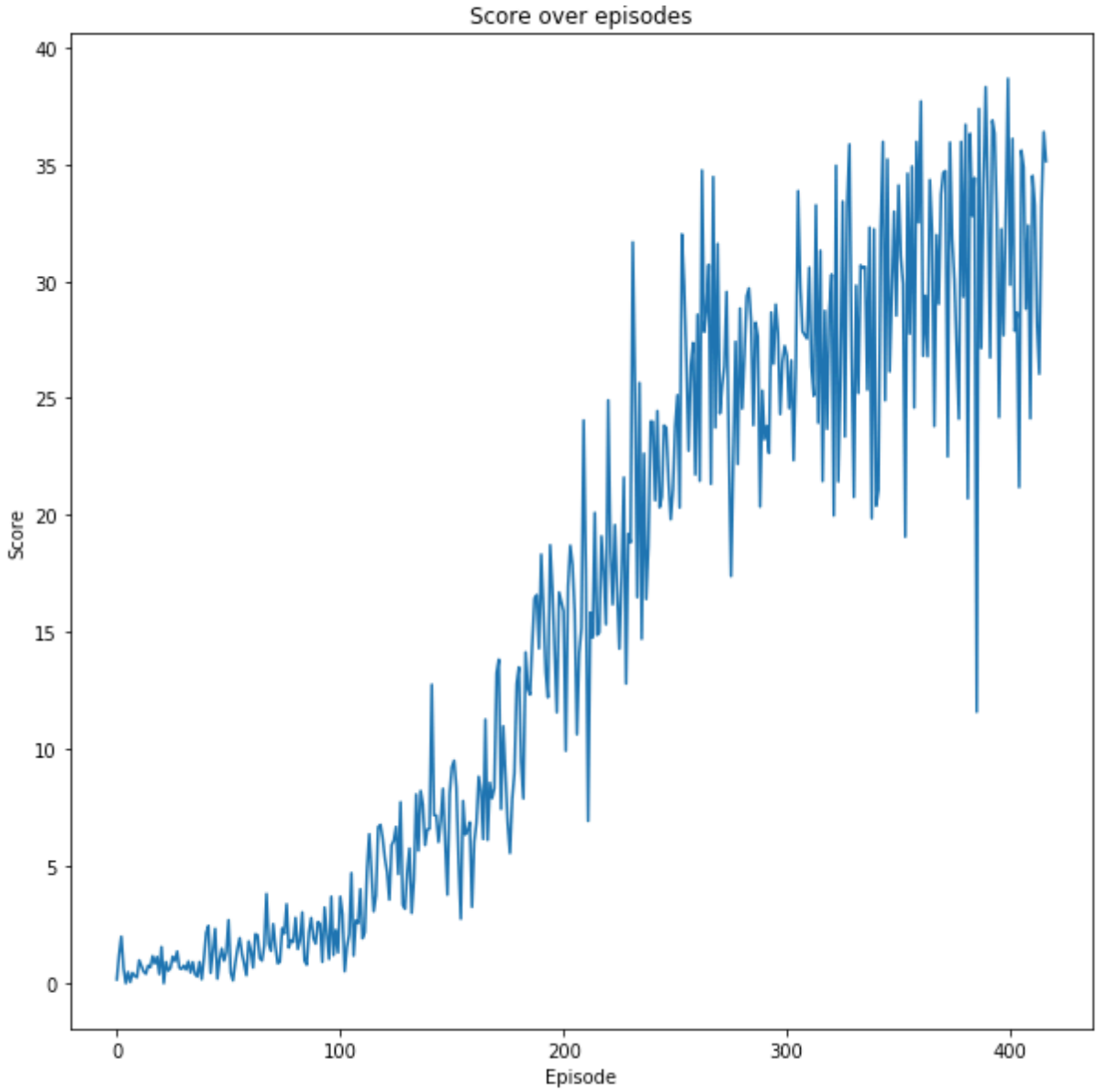
To stabilize the learning, two identical networks are initialized for both the actor and the critic, one on which the actual training happens (local network) and one which serves as a target network that is only updated once in a while by copying over the current parameter values from the local network (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

BUFFER_SIZE	10000
BATCH_SIZE	64
GAMMA	0.99
TAU	0.001
LR	0.0005
UPDATE_EVERY	4
WEIGHT_DECAY	0.00
DIMENSION HIDDEN LAYERS	Actor: (128, 64), Critic: (132, 64)

Results

As it can be seen in the following plot, the algorithm proved to be able to solve the task in 417 episodes. Tuning the learning rate proved to influence the speed of learning quite significantly. A learning rate of 0.0001 solved the task in almost twice the number of episodes as in the solution presented here.



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.