Report on Results

Deep Reinforcement Learning

Project 1: Navigation

David Neuhäuser

1 Framework

In the current project, the goal was to train one **or** several agents in parallel where an agent corresponds to a double-jointed arm which can move to target locations. A reward of $R_t = 0.1$ is provided to an agent for each step staying with its hand within its green target location. Thus, in order to optimize the total score, the aim of the agent(s) is to stay in its/their position(s) at the target location for as many time steps as possible. The state space $\mathcal{S}$ has 33 dimensions and contains the agent's velocity, rotation, position as well as angular velocities of the arm. Based on this information, the agent has to learn how to select actions in the best way. Each action is a vector with four numbers, corresponding to torque applicable to the two joints. Every entry in the action vector is a number between -1 and 1. Therefore, the action space is given by $\mathcal{A} = [-1, 1] \times [-1, 1] \times [-1, 1] \times [-1, 1]$ and is thus continuous. The task is episodic, i.e. $t \in [0, 1, \ldots, T]$.

In the current project, focus was put on version two, i.e., considering 20 **identical agents, each with its own copy of the environment.**

In order to pass, the agents need an average score of 30 over 100 consecutive episodes, and over all 20 agents. To be more precise, after each episode j, add up the rewards $R_t^{(i,j)}$ that each agent i received to get a score $G_{i,j} = \sum_{t=0}^{T} R_t^{(i,j)}$ for each agent. This yields 20 scores $G_{1,j}, \ldots, G_{20,j}$. Then, take the average $M_j = \frac{1}{20} \sum_{i=1}^{20} G_{i,j}$ of these 20 scores leading to an average score M_j for each episode j. The environment is considered solved, when the average over 100 episodes of those average scores M_j is at least 30.

2 Detailed Description of the Algorithm

The algorithm which is used to train the agent for solving the task is the Deep Deterministic Policy Gradient algorithm. It is one of the so-called Actor-Critic Methods (others are for instance TRPO, PPO, A3C or A2C) and uses two neural networks for

learning: one for the actor and one for the critic. The actor is responsible for the "policy part" which chooses the next action based on the current state. The critic is responsible for the "value part" which evaluates the action chosen by the actor. The general idea of the algorithm is to use value-based techniques in order to reduce variance of policy-based models. As the name of the algorithm already points out, there is no stochastic policy in the model but instead,

$$\arg\max_a Q(s, a)$$

is used for the next action.

In addition, some stabilizing techniques helped to improve the performance in training the model.

- due to the presence of 20 agents, it turned out that updating the weights only after every 20 steps and for every such step, updating the weights 7 times improved results a lot. Of course, these numbers could be seen as hyperparameters in the algorithm and could be tuned. Moreover, some other techniques contributed significantly towards stabilizing the training, too.

- experience replay: in order to de-correlate the state-action-reward-next state tuples $(s_t, a_t, r_{t+1}, s_{t+1})$, $(s_{t+1}, a_{t+1}, r_{t+2}, s_{t+2}), \ldots$, it is useful to create a replay buffer of certain size. Then sample at random from this buffer leading to (at least close to) i.i.d. samples instead of learning from tuples as they come over time.

- fixed Q-targets: fix the weights of the neural net in the target $\widehat{Q}$. Of course, this leads to the consequence that we need two neural networks for the actor and two networks for the critic, one as the target network and one as the online network, respectively. However, we can get rid of instability issues when updating the model weights.

- soft updates: so far in the Deep Q-Networks algorithm, the target networks are updated by copying all the weights from the local networks after a certain number of epochs. Now, within the Deep Deterministic Policy Gradient framework, we have two copies of weights for each network: actor online, actor target, critic online, critic target. At every update step, we mix 0.01% of the online weights with the target weights to update the target weights of the actor and the critic.

- use of gradient clipping when training the critic network

An interesting topic in the provided solution is the underlying architecture of the neural networks for the actor and critic, respectively. More precisely, we investigated two different architectures which are as follows:

- Architecture 1:
  - Actor_2HL_BN: a neural net with two hidden layers. The first layer has $|\mathcal{S}| = 33$ nodes, the second layer (first hidden) 128 nodes, the third layer (second hidden) 256 nodes and finally the fourth layer $|\mathcal{A}| = 4$ nodes. Activation

function is ReLU, except for the last one which is tanh. Additionally, we apply batch-normalization to the first hidden layer.

  - Critic_2HL_BN: a neural net with two hidden layers. The first layer has $|\mathcal{S}| = 33$ nodes, the second layer (first hidden) 128 nodes, the third layer (second hidden) 256 nodes and finally the fourth layer 1 node. For details, we refer to the Python code. Activation function is ReLU, again. Additionally, we apply batch-normalization to both hidden layers.

- - Architecture 2:
  - Actor_1HL: a neural net with one hidden layer. The first layer has $|\mathcal{S}| = 33$ nodes, the second layer (first hidden) 256 nodes and finally the third layer $|\mathcal{A}| = 4$ nodes. Activation function is ReLU, except for the last one which is tanh.
  - Critic_3HL_leaky: a neural net with three hidden layers. The first layer has $|\mathcal{S}| = 33$ nodes, the second layer (first hidden) 256 nodes, the third layer (second hidden) 256 nodes, the fourth layer (third hidden) 128 nodes and finally the fifth layer 1 node. For details, we refer to the Python code. Activation function here is leaky ReLU.

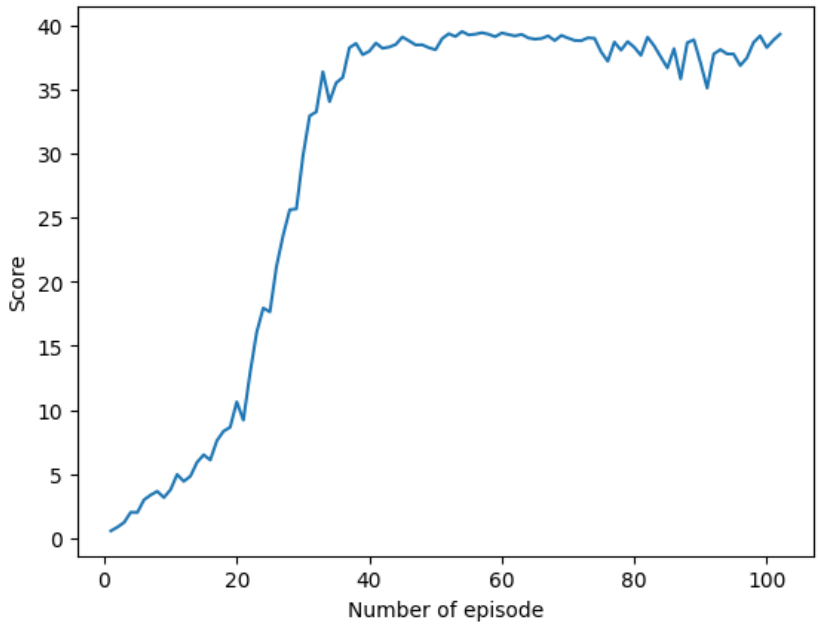
In the following table, all hyperparameters which were used for the Deep Deterministic Policy Gradient algorithm are listed:

Hyperparameter	Value
replay buffer size	1000000
batch size	128
discount factor γ	0.95
soft update parameter τ	0.001
learning rate actor	0.0001
learning rate critic	0.001
weight decay Adam optimizer	0.0
number of episodes	5000
maximum number of timesteps per episode	1000
negative slope leaky ReLU	0.01

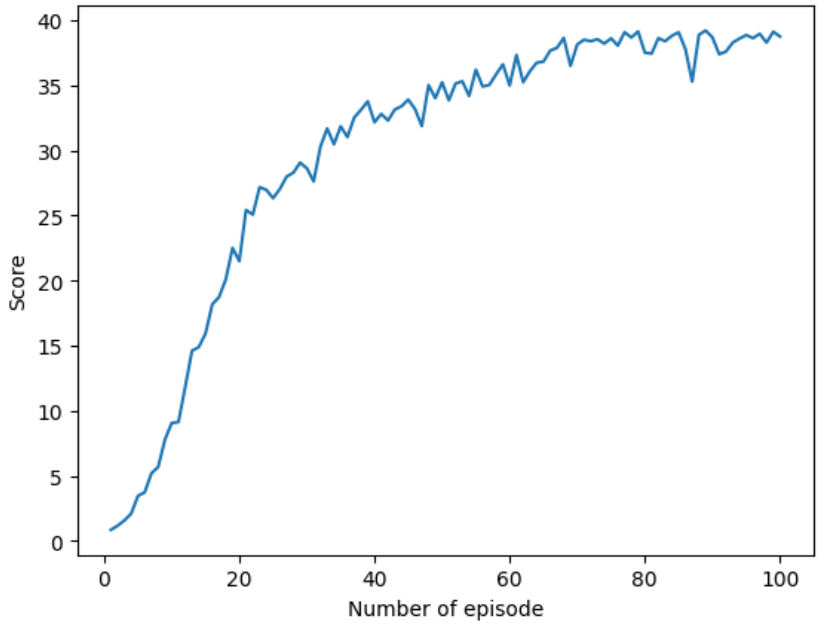
3 Results

The performance of the actor-critic-agent via architecture 1 was as follows: The environment was solved in 102 episodes with anverage score of 30.36 over the last 100 episodes,

also see the following figure.



The performance of the actor-critic-agent via architecture 2 was as follows: The environment was solved in 100 episodes with anverage score of 30.04 over the last 100 episodes, also see the following figure.



Obviously, architecture 1 has a very strong incline at the beginning and then plateaus from episode 40 onwards. Contrary, architecture 2 has a weaker increase of the score but

still seems to learn at episode 100. As a consequence, at least minor efforts of changing the fundamental structure of the neural net seem to eventually bring performance increase. Beyond, we can consider the ideas of the next section in future.

4 Ideas for Future Work

- Usage of other network architectures
- Prioritized experience replay: instead of uniformly sampling from the replay buffer, there should be some kind of importance sampling, taking more relevant tuples with higher probability than minor important ones.
- Hyperparameter tuning
- Testing other algorithms than DDGP: try A3C, A2C, TRPO, etc.