Learning algorithm

The learning algorithm applied goes by the name 'Deep Deterministic Policy Gradient'.

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

(from https://arxiv.org/abs/1509.02971)

The deep deterministic policy gradient method offers a solution to the fact that deep Q networks (DQN) don't scale to high dimensional action spaces.

Like DQN the DDPG algorithm makes use of a replay buffer from which steps are taken in randomized minibatches for learning to remove variance and a local and target network to improve learning stability. The big difference is however that DDPG uses two separate local/target networks. One that learns the action policy and another that learns the action-state value Q.

To solve the reacher environment initially the ddpg agent from https://github.com/udacity/deep-reinforcement-learning/tree/master/ddpg-pendulum was taken. This however proved a frustrating endeavour because changing hyperparameters didn't seem to have any effect. It would never learn anything.

However, after replacing the critic with the critic from https://github.com/udacity/deep-reinforcement-learning/tree/master/ddpg-bipedal (which adds an extra layer) the learning started to work.

So these are the networks used:

Actor:

3 linear layers (400, 300, action_size), 2 relu activations with tanh output.

Critic:

5 linear layers (256, 256 +action_size, 256,128, 1), leaky relu for first 4 layers. Last layer has no activation function.

Actor and Critic can be found in models.py

The agent was modified to take in packed hyper parameters in order to be able to do a grid search. In addition priority replay buffer was added, which can be turned on with setting the Boolean PR to True. This however turned out to run way to slow on a cpu so it's not part of the solution offered in this report.

```
class Agent():
    """Interacts with and learns from the environment."""

    def __init__(self, state_size, action_size, hyper_params, PR=False, random_seed=0):
        """Initialize an Agent object.
```

A grid search was done on the hyperparameters using optima. With optima the agent was run for 40 steps and the mean of the last hundred scores was taken as an optimality objective.

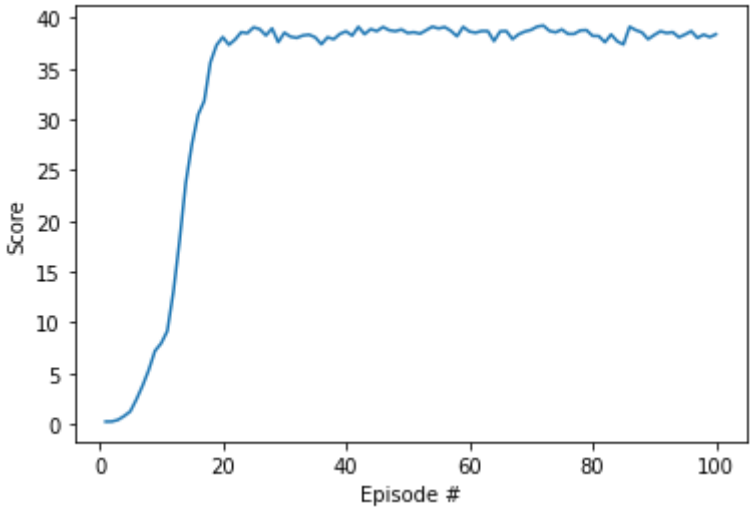
Optima ran for 9 trials and was then terminated because trial 5 already gave a good result and the search is very time consuming.

Solution

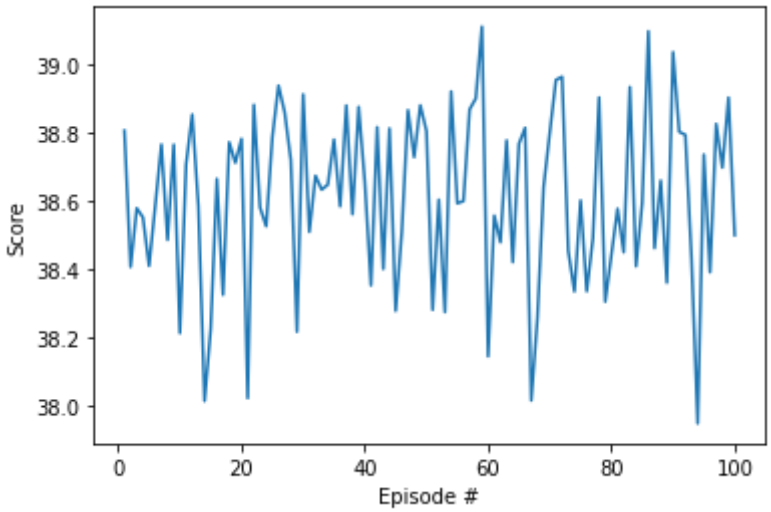
The hyperparameters found using optima and used in the the final solution are:

Buffer size	909973
Batch size	384
gamma	0.9901736020389454
tau	0.0018092674864975023
Learning rate actor	0.00017377360292847218
Learning rate critic	0.00019323989991911093
Weights decay	0.0007434075695379568
theta	0.20215485858989818
sigma	0.2053184766109179
mu	-0.0036382192060112045

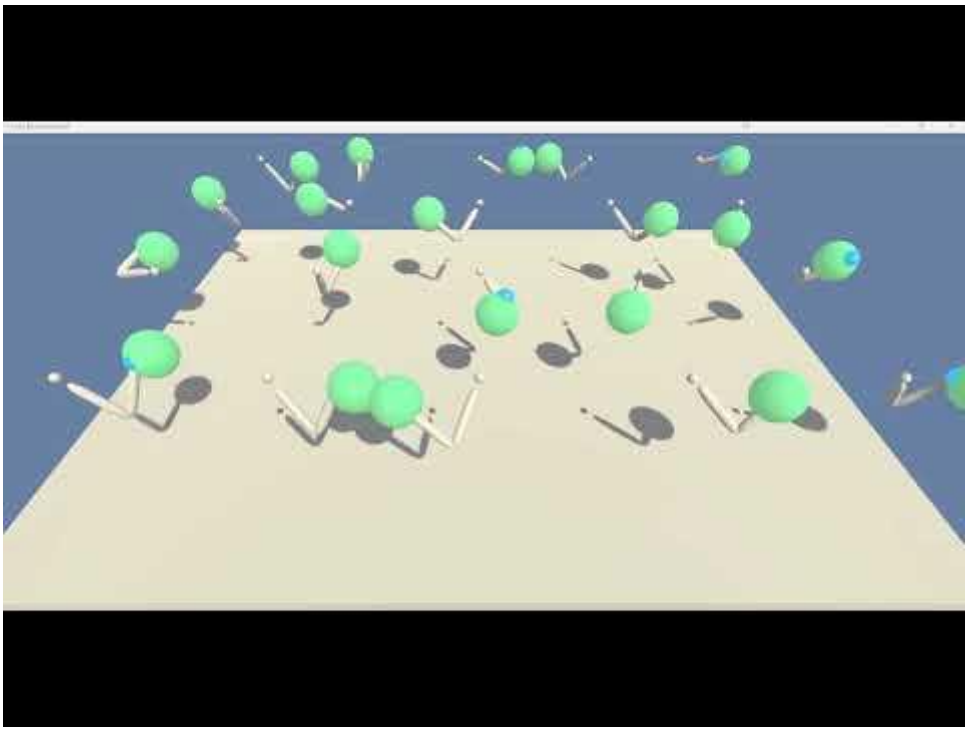
Learning is done every 16 steps with 16 minibatches.



The actor first scores a score higher than 30 at episode 16. Running the final agent for 100 episodes show the score averaging somewhere around 38.6.



Short clip of the trained agent working.



Ideas for future work.

Given the fact that most of the space around the arm is empty, without reward. It seems that prioritized experience replay with a relative high priority on underestimation of reward relative to overestimation of reward could improve performance.

Experiments with PER however proved to computationally expensive to explore this within a reasonable time.

Another idea would be to train a separate network for ‘first contact’. In the trained network most points are missed in the initial stage, when it is unknown where the ball is. There likely is an optimal movement for on average hitting the ball the first time in the shortest amount of time. If is trained the agent could start with this network and transfer control to the original network after first contact.