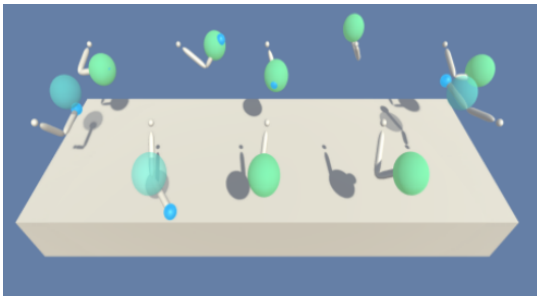
Report: Continuous Control

Environment

For this project, you will work with the **Reacher** environment. In this environment, a double-jointed arm can move to target locations.
A reward of +0.1 is provided for each step that the agent's hand is in the goal location. Thus, the goal of your agent is to maintain its position at the target location for as many time steps as possible.
The observation space consists of 33 variables corresponding to position, rotation, velocity, and angular velocities of the arm. Each action is a vector with four numbers, corresponding to torque applicable to two joints. Every entry in the action vector should be a number between -1 and 1.



DDPG Algorithm

The DDP algorithm was proposed and is described in reference [1]. Deep Deterministic Policy Gradient (DDPG) is an algorithm which concurrently learns a Q-function and a policy. It uses off-policy data and the Bellman equation to learn the Q-function, and uses the Q-function to learn the policy. Policy gradients methods directly operates in the policy space. They can learn stochastic policies, can model continuous action spaces and usually show faster convergence than DQN. However they can have high variance in estimating the gradient of the estimated reward. Using a Q-function as a baseline can be used to reduce variance. In this context, the Policy gradient network is called an **ACTOR** and the Q-funcion baseline is called a **CRITIC**.
To calculate losses two networks are initialized for both the Actor and the Critic. One is a target network and the other one is a local or training network that we are constantly updating. The target network is updated more slowly. In this case we are using a soft update approach that updates the target constantly but by a little amount in the direction of the local network. The little amount is managed by the parameter *TAU.*

Approach

For this project, I essentially used the same code as in the **ddpg-pendulum example**. The code can be found in Udacity's deep-reinforcement-learning repository: https://github.com/udacity/deep-reinforcement-learning/tree/master/ddpg-pendulum
Only minor changes were made to solve the project. The main changes were:

- The **ddpg** method was adapted for the Reacher Unity environment. The environment is able to handle 20 agents, which improves training times. To account for this, the **agent.step** method must be able to save the experience for all agents. The "score" returned will be the **mean score** for all the agents. The Benchmark Implementation suggest to learn 10 times for every 20 time steps. In the code this can be controlled with the *LEARN_EVERY* parameter. If *LEARN_EVERY*=0, the agent trains once every timestep. For the final implementation, I chose the following behavior: save the experiences of the 20 agents **two times** in the buffer and train once every time step. I saved the experiences two times because I noticed it trained a little faster, probably because the buffer fills-up faster.
- As suggested in the Benchmark Implementation for this project, we added **gradient clipping** when training the critic network. This helps us avoid an exploding gradient as it seemed to be causing sudden drops in score. This is managed by the parameter *GRAD_CLIPPING.*
- A normalization, using pytorch's **Batchnorm1d**, was added between layers for the actor and critic networks. This helps with training giving a nicer distribution of data for the activation functions.
- From the pendulum example, we are returning a noisy value for our agents' actions, As we get better with time, we would like to reduce the noise as we learn. For this reason, a noise decay rate factor EPSILON was added. For every pass of **agent.learn** the factor is reduced in EPSILON_DECAY.

Model

The model is essentially the same as the one in the ddpg-pendulum example, but with Batchnorm1d added. The Agent is a 3-layered network with **Relu** activation for the hidden dimensions and a **Tanh** activation for the output. The Agent receives a state as an input and returns the probabilities for each action. The critic has a similar architecture but actions are being concatenated after the first layer. It also receives a state but outputs a single number (with no activation), which represents the predicted Q-value for that action-state.

Hyperparameters

The relevant hyperparameters are:

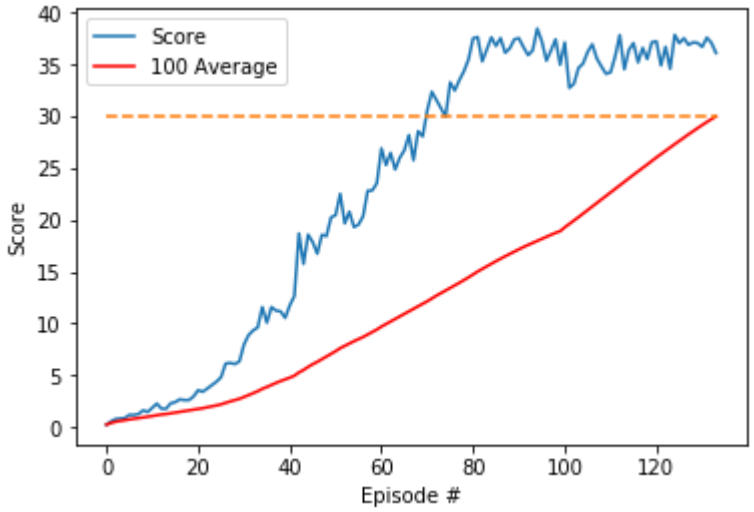
BUFFER_SIZE = int(1e6)	*Replay buffer size*
BATCH_SIZE = 256	*Minibatch size*
GAMMA = 0.99	*Discount factor*
TAU = 1e-3	*For soft update of target parameters*
LR_ACTOR = 1e-3	*Learning rate of the actor*
LR_CRITIC = 1e-3	*Learning rate of the critic*
WEIGHT_DECAY = 0	*L2 weight decay*
EPSILON = 1.0	*Noise decay start value*
EPSILON_DECAY = 1e-6	*Noise decay factor*
LEARN_EVERY = 0	*Learning timestep interval*

LEARN_NUM = 10	*Number of learning passes*
GRAD_CLIPPING = 1.0	*Gradient Clipping*

From the **ddpg-pendulum example** only few parametesr where modifed. The *BUFFER_SIZE* was increase from 1e5 to 1e6. *BATCH_SIZE* was increased from 128 to 256. All other parameters in the example were left the same.
The params *EPSILON, EPSILON_DECAY, LEARN_EVERY, GRAD_CLIPPING* were added. They are explained in the "Approach" section.

Result

The environment was solved in 134 episodes. An average of 30.01 was achieved for a running mean of scores between episode 34 to episode 134.



Ideas for Future Work

The first thing that should be interesting to try will be to implement different algorithms for this same environment. Particularly the ones mentioned in the Actor-Critic lesson:

• A3C: Asynchronous Advantage Actor-Critic
• A2C: Advantage Actor-Critic
• GAE: Generalized Advantage Estimation

Also plain REINFORCE should be able to solve the environment.

Apart from that, playing a bit more with the hyper parameters might make a difference. For example, I didn't' touch the noise parameters, which might result in something interesting. I also chose to learn once every time step, because it worked ok, but definitely other behaviors might make an important

difference. Other hyper parameter that I didn't change was the learning rate. It could be that a smaller number might find a better minimum.

References

[1] https://arxiv.org/abs/1509.02971
[2] https://github.com/Unity-Technologies/ml-agents/blob/master/docs/Learning-Environment-Examples.md
[3]https://flyyufelix.github.io/2017/10/12/dqn-vs-pg.html
[4] https://spinningup.openai.com/en/latest/algorithms/ddpg.html