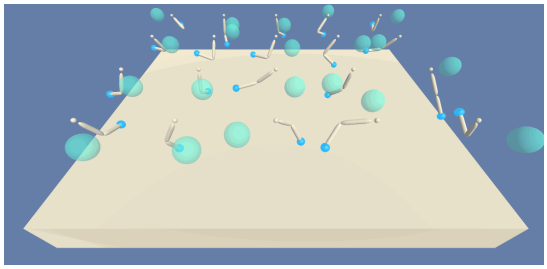
Project Details

The project uses the unity agent environment Reacher.

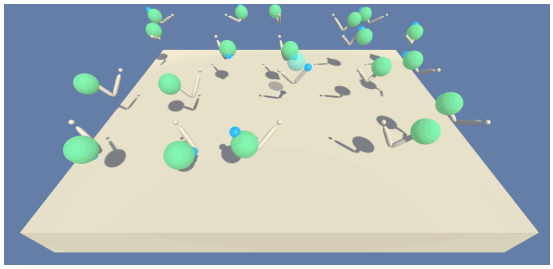
In this environment, a double-jointed arm can move to target locations. A reward of +0.1 is provided for each step that the agent's hand is in the goal location. Thus, the goal of the agent is to maintain its position at the target location for as many time steps as possible.

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.

Example of untrained agents.



Trained Agents



For the implementation I used DDPG as the base algorithm.

DDPG

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.

Algorithm 1 Deep Deterministic Policy Gradient

Input: initial policy parameters θ, Q-function parameters ϕ, empty replay buffer $\mathcal{D}$
Set target parameters equal to main parameters $\theta_{\text{targ}} \leftarrow \theta$, $\phi_{\text{targ}} \leftarrow \phi$
repeat
  Observe state s and select action $a = \text{clip}(\mu_\theta(s) + \epsilon, a_{Low}, a_{High})$, where $\epsilon \sim \mathcal{N}$
  Execute a in the environment
  Observe next state s', reward r, and done signal d to indicate whether s' is terminal
  Store (s, a, r, s', d) in replay buffer $\mathcal{D}$
  If s' is terminal, reset environment state.
  if it's time to update **then**
    for however many updates **do**
      Randomly sample a batch of transitions, $B = \{(s, a, r, s', d)\}$ from $\mathcal{D}$
      Compute targets

$$y(r, s', d) = r + \gamma(1 - d)Q_{\phi_{\text{targ}}}(s', \mu_{\theta_{\text{targ}}}(s'))$$

      Update Q-function by one step of gradient descent using

$$\nabla_\phi \frac{1}{|B|} \sum_{(s,a,r,s',d) \in B} \left(Q_\phi(s, a) - y(r, s', d)\right)^2$$

      Update policy by one step of gradient ascent using

$$\nabla_\theta \frac{1}{|B|} \sum_{s \in B} Q_\phi(s, \mu_\theta(s))$$

      Update target networks with

$$\phi_{\text{targ}} \leftarrow \rho\phi_{\text{targ}} + (1 - \rho)\phi$$
$$\theta_{\text{targ}} \leftarrow \rho\theta_{\text{targ}} + (1 - \rho)\theta$$

    end for
  end if
until convergence

DDPG is an off-policy algorithm.

DDPG can only be used for environments with continuous action spaces.

DDPG can be thought of as being deep Q-learning for continuous action spaces.

The Spinning Up implementation of DDPG does not support parallelization.

DDPG use an Actor-Critic method, reason why we implemented it as two models

- Actor - It proposes an action given a state.
- Critic - It predicts if the action is good (positive value) or bad (negative value) given a state and an action.

Why two networks? Because it adds stability to training. In short, we are learning from estimated targets and Target networks are updated slowly, hence keeping our estimated targets stable.

Conceptually, this is like saying, "I have an idea of how to play this well, I'm going to try it out for a bit until I find something better", as opposed to saying "I'm going to re-learn how to play this entire game after every move".

DDPG uses experience relay : We store a list of tuples `(state, action, reward, next_state)`, and instead of learning only from recent experience, we learn from sampling all of our experience accumulated so far.

Case one agent:

The task is episodic, and in order to solve the environment, the agent must get an average score up to 30 over 100 consecutive episodes.

Reacher environment used: Reacher_One_Linux_NoVis.x86_64

Case Multi Agent case:

The agents must get an average score up to 30 (over 100 consecutive episodes, and over all agents).

After each episode, we add up the rewards that each agent received (without discounting), to get a score for each agent. This yields 20 (potentially different) scores. We then take the average of these 20 scores. This yields an average score for each episode (avg of the 20 agents).

Reacher environment used: Reacher.x86_64

Getting Started:

What you need to install

Instructions:

How to Run, simple agent:

Github Link:
https://github.com/ChristianET-DS/ContinuesControl/tree/main/SingleAgent

Reset environment:
There is 1 agent. Each observes a state with length: 33 and act within an action space of length: 4

Simple Agent Logic

Actor Model Architecture
3 Linear layers and a Batch Normalization
Dense layers
state_size 33 fc1_units 128 fc2_units 128 bn_mode 2

```
        self.fc1 = nn.Linear(33, 128)
        self.fc2 = nn.Linear(128, 128)
        self.fc3 = nn.Linear(128, 4)

# Normalization layers ##############################################
         nn.BatchNorm1d(128)
 #  Forward ###################################################################
         # Batch Normalization after Activation
           x = F.relu(self.fc1(state))
           x = nn.BatchNorm1d(128 )
           x = F.relu(self.fc2(x))
           return F.tanh(self.fc3(x))
```

state_size 33 fc1_units 128 fc2_units 128 bn_mode 2

Critic Model Architecture

```
 # Dense layers ##########################################################
        self.fcs1 = nn.Linear(33, 128)
        self.fc2 = nn.Linear(128+33,128)
        self.fc3 = nn.Linear(128, 1)

 # Normalization layers ##################################################
        self.bn1 = nn.BatchNorm1d(128)
        self.bn2 = nn.BatchNorm1d(128)
        self.reset_parameters()
# Forward ##############################################################
        xs = F.relu(self.fcs1(state))
        xs = nn.BatchNorm1d(128 )
        x = torch.cat((xs, action), dim=1)
        x = F.relu(self.fc2(x))
        return self.fc3(x)
```

Agent:
DDPG requires two networks one actor and one critic.

The logic starts with the creation of Actor Network instances one local and one target. The Target Network has the same weight values as the Local Network.

Followed by the creation of Critic Network one local and one target..The Target Network has the same weight values as the Local Network.

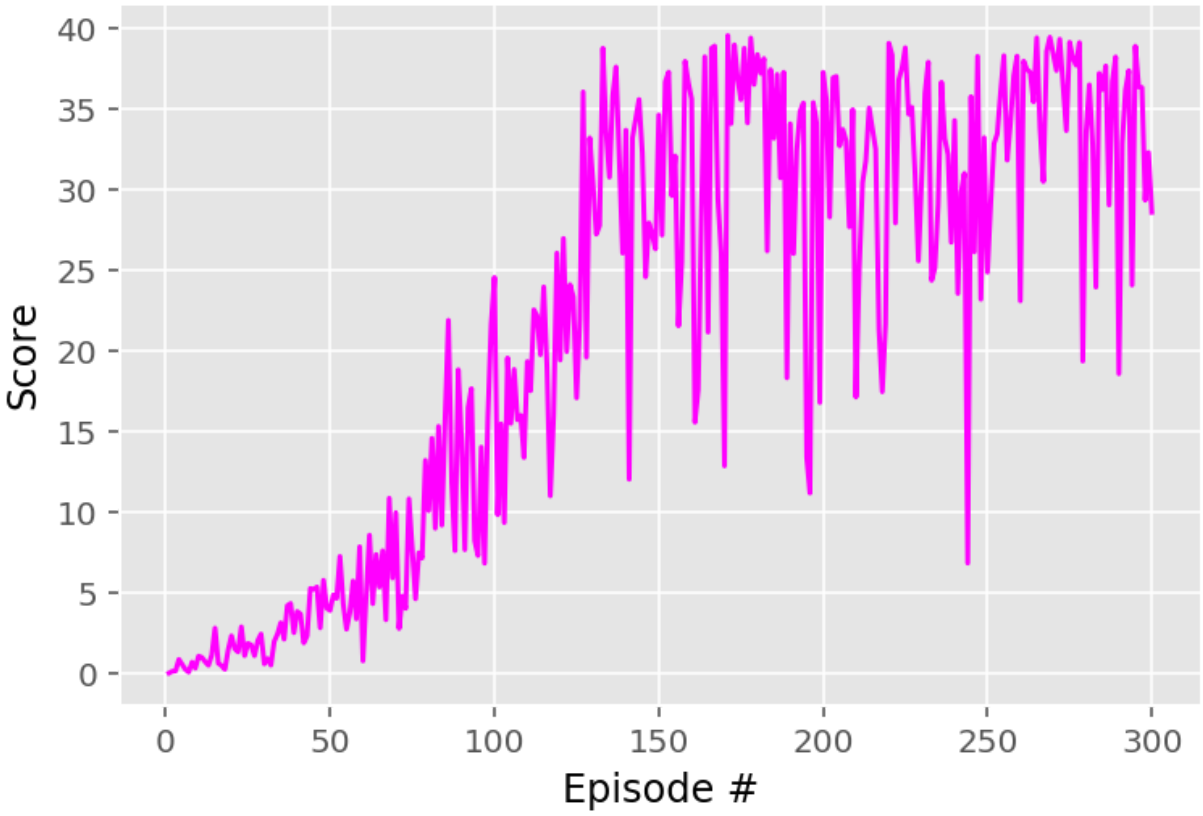
In each step it saves the experience in replay memory, then it uses a random sample from replay memory to learn.

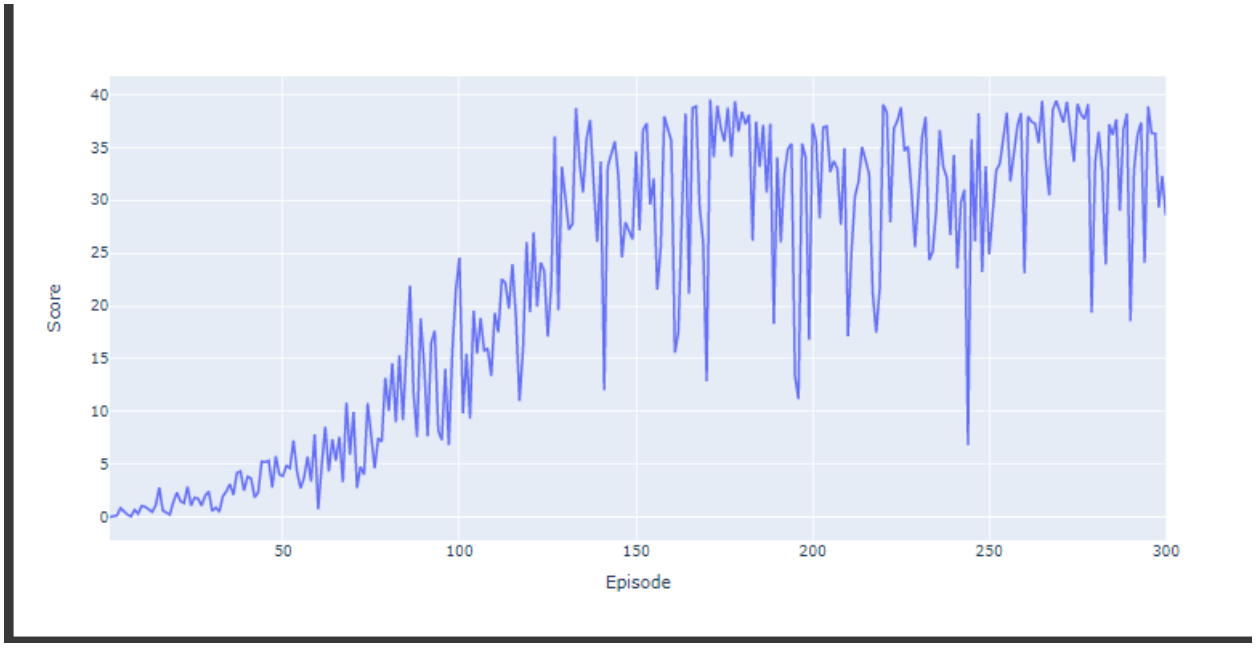
Each time that we return an action for a given state as per current policy (act function), we also add the Noise. In theory the noise makes the learning better.

In the learn function.
We predict the next-state actions and Q values from target models, compute Q targets for current states and minimize the loss. Then update the local and target networks.

Results:

Episode 100 Average Score: 5.62 Score: 24.55
Episode 200 Average Score: 27.80 Score: 37.28
Environment solved in 300 episodes with an Average Score of 32.45


Score
Episode #
0
5
10
15
20
25
30
35
40
0
50
100
150
200
250
300


Score
Episode
0
5
10
15
20
25
30
35
40
50
100
150
200
250
300

How to Run, multiple agents (20):

Run notebook :
https://github.com/ChristianET-DS/ContinuesControl/tree/main/MultiAgent

Multi Agent Logic

Actor model

```
        self.fc1 = nn.Linear(state_size, fc1_units)
        self.fc2 = nn.Linear(fc1_units, fc2_units)
        self.fc3 = nn.Linear(fc2_units, action_size)

 def forward(self, state, action=None):
        """ Perform forward pass and map state to action """
        state = F.relu(self.fc1(state))
        state = F.relu(self.fc2(state))
        return torch.tanh(self.fc3(state))
```

Critic model

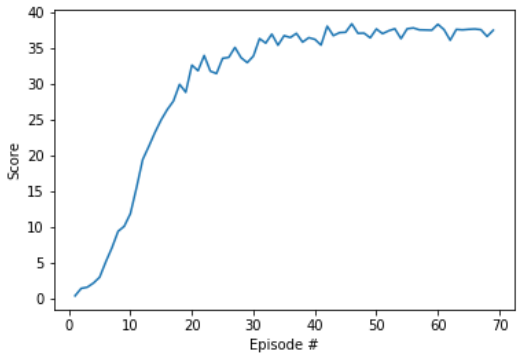
```
        self.fc1 = nn.Linear(state_size, fc1_units)
        self.fc2 = nn.Linear(fc1_units + action_size, fc2_units)
        self.fc3 = nn.Linear(fc2_units, 1)
    def forward(self, state, action=None):
        xs = F.leaky_relu(self.fc1(state))
        x = torch.cat((xs, action), dim=1)
        x = F.leaky_relu(self.fc2(x))
        return self.fc3(x)
```

Differents with multiple agents
The score is not a simple value , we have an array of scores
score = 0 if single_agent else np.zeros(num_agents) # initialize the score
Also an array of observations and rewards.
env_info.vector_observations, env_info.rewards, env_info.local_done

Results:

The multi agent shows better learning with a simple network. Not batchnormalization.

```
Initialised 'Actor: Local' model
Initialised 'Actor: Target' model
Initialised 'Critic: Local' model
Initialised 'Critic: Target' model
2022-11-19 13:51:34.212906 - Training 20 agents for max 3000 episodes. Target score to reach is 30.0
Agent experiences collected 32, agent requires  128 experiences
Agent experiences collected 64, agent requires  128 experiences
Agent experiences collected 96, agent requires  128 experiences
Agent experiences collected 128, agent requires  128 experiences
---2022-11-19 13:51:34.212906---Episode 69      Average Score: 30.07
---2022-11-19 13:51:34.212906---Environment solved in 69 episodes!      Average Score: 30.07
Target score of 2022-11-19 13:51:34.212906 has been reached. Saving model to 30.0
2022-11-19 15:27:15.418694 - Finished training successfully!
```



Ideas for Future Work:

The next could be train the agent with another kind of algorithm like **D4PG**

Try keras implementation https://keras.io/examples/rl/ddpg_pendulum/

Reference :https://spinningup.openai.com/en/latest/algorithms/ddpg.html