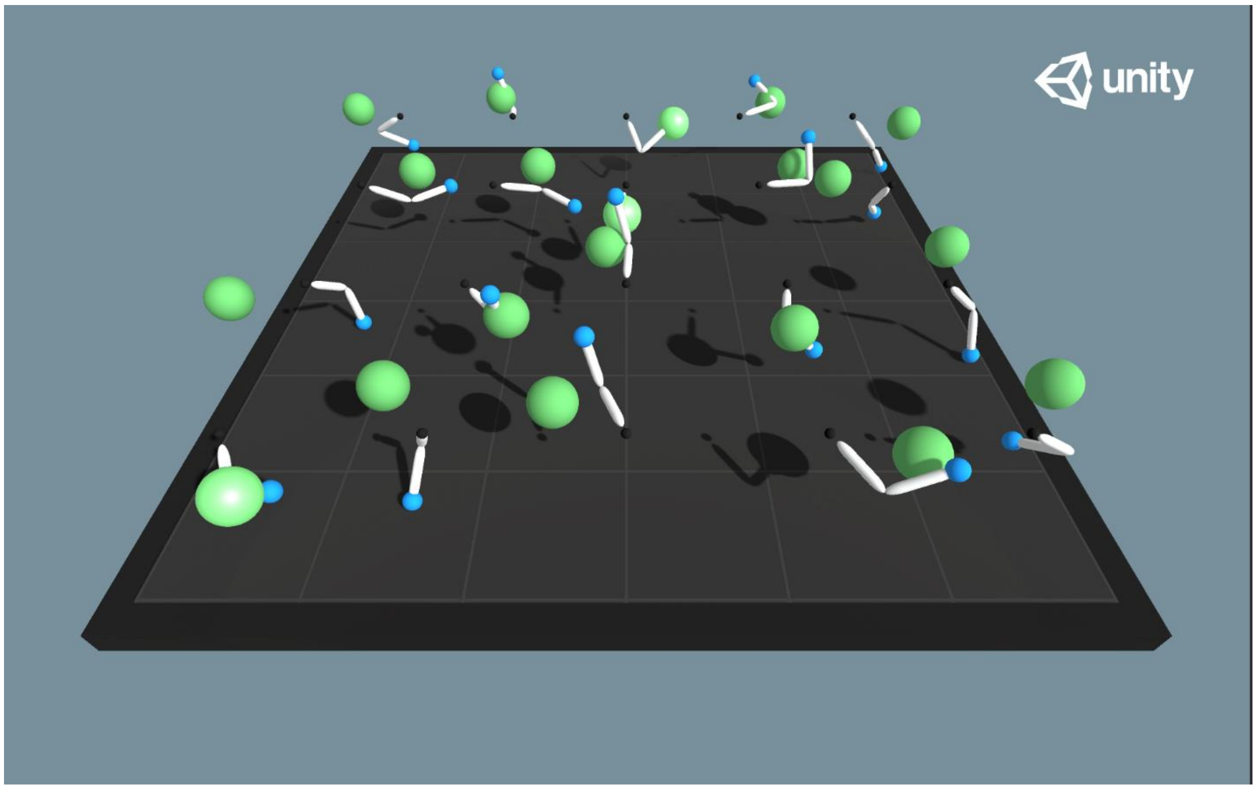
REACHER ENVIRONMENT

In the Reacher environment, the agent is a robot arm with two degrees of freedom. The goal of the agent is to maintain its position at the target location for as many time steps as possible as the target is moving. There are 20 agents in this environment and the task is episodic. The environment is considered solved, when the average score over 100 episodes is at least +30. Specifically,

- 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, where the average is over all 20 agents.

__**Behavior Parameters:**__

- Vector Observation space: 33 variables corresponding to position, rotation, velocity, and angular velocities of the two arms.
- Vector Action space: (**Continuous**) Size of 4, corresponding to torque applicable to two joints. Every entry in the action vector should be a number between -1 and 1.
- Visual Observations: None.
- A reward of +0.1 is provided for each step that the agent's hand is in the goal location.

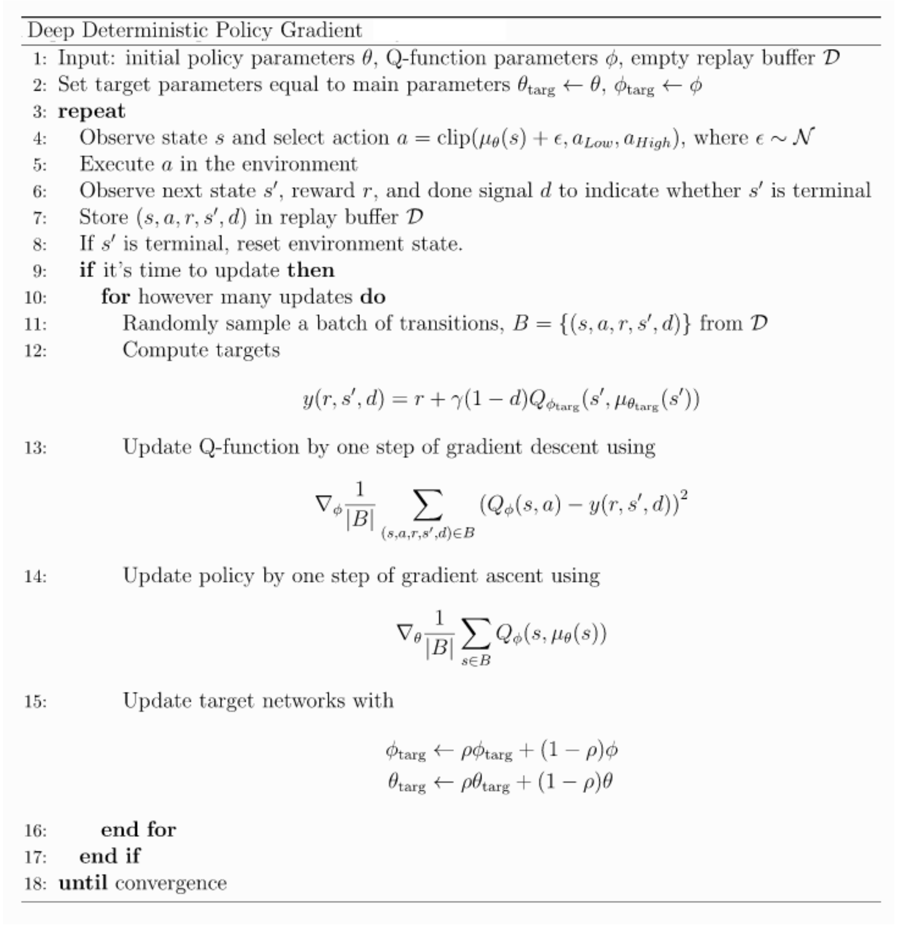


DEEP DETERMINISTIC POLICY GRADIENTS

Deep deterministic policy gradients (DDPG) was used as the learning algorithm as it is straightforward to build upon the existing deep-q-networks (DQN) implementation I've already had. DDPG is a combination of deterministic policy gradients with ideas from deep Q-learning. It is under the umbrella of actor-critic algorithms. DDPG uses a critic network that learns a Q-function and an actor network that learns a policy. The critic can be thought of generalization of the DQN to continuous action spaces. DQN estimates the Q function and computes the max over actions, $max_a = Q(s,a)$, to select a policy. Computing max over actions for a continuous action space, requires iterative optimization and is expensive. Therefore, critic approximates it by $max_a Q(s,a) \approx Q(s,\mu(s))$. The loss function for the critic is mean-squared Bellman error (MSBE), same as DQN. The actor network then performs policy gradient using $Q(s,\mu(s))$ as a loss function (step 14 in Figure 1).

Algorithm:

DDPG implementation is described here in detail. The pseudo code of the algorithm is below and follows the **OpenAI baseline**.



```
Deep Deterministic Policy Gradient
1: Input: initial policy parameters $\theta$, Q-function parameters $\phi$, empty replay buffer $\mathcal{D}$
2: Set target parameters equal to main parameters $\theta_{\text{targ}} \leftarrow \theta$, $\phi_{\text{targ}} \leftarrow \phi$
3: repeat
4:   Observe state $s$ and select action $a = \text{clip}(\mu_\theta(s) + \epsilon, a_{Low}, a_{High})$, where $\epsilon \sim \mathcal{N}$
5:   Execute $a$ in the environment
6:   Observe next state $s'$, reward $r$, and done signal $d$ to indicate whether $s'$ is terminal
7:   Store $(s, a, r, s', d)$ in replay buffer $\mathcal{D}$
8:   If $s'$ is terminal, reset environment state.
9:   if it's time to update then
10:    for however many updates do
11:      Randomly sample a batch of transitions, $B = \{(s, a, r, s', d)\}$ from $\mathcal{D}$
12:      Compute targets
             $y(r, s', d) = r + \gamma(1 - d)Q_{\phi_{\text{targ}}}(s', \mu_{\theta_{\text{targ}}}(s'))$
13:      Update Q-function by one step of gradient descent using
             $\nabla_\phi \frac{1}{|B|} \sum_{(s,a,r,s',d) \in B} (Q_\phi(s,a) - y(r,s',d))^2$
14:      Update policy by one step of gradient ascent using
             $\nabla_\theta \frac{1}{|B|} \sum_{s \in B} Q_\phi(s, \mu_\theta(s))$
15:      Update target networks with
             $\phi_{\text{targ}} \leftarrow \rho\phi_{\text{targ}} + (1 - \rho)\phi$
             $\theta_{\text{targ}} \leftarrow \rho\theta_{\text{targ}} + (1 - \rho)\theta$
16:    end for
17:  end if
18: until convergence
```

Figure 1: Pseudocode

To summarize the key components:

- ***Replay buffer:*** State transitions and corresponding rewards are saved in a buffer and sampled randomly in batches during training (step 11). Therefore, DDPG is an off-policy method. Since MSBE is used as a loss function and the off-policy samples can be used.
- ***Soft parameter update***: Just like DQN, DDPG uses target networks for both the actor and the critic to decorrelate target and predicted Q values, which in turn improves the stability of learning algorithm and convergence. The online network weights are slowly blended into the target network (step 15)
- ***Exploration:*** DDPG uses a deterministic policy which can hinder exploration. In discrete action spaces, exploration is encouraged by using an epsilon-greedy policy. In DDPG, Ornstein-Uhlenbeck noise is added during training to encourage exploration over a continuous action space (step 4). The state value is also clipped to the min and max allowable range. In the OpenAI implementation, the agent also takes uniformly sampled random actions for a fixed number of steps at the beginning before starting training.

Architecture*:*

Actor:

- (fc1) : Linear (in_features=33, out_features=256, bias)
- (bn1): BatchNorm1d(256, eps=1e-05, momentum=0.1)
- (fc2) : Linear(in_features=256, out_features=128)
- (fc3) : Linear(in_features=128, out_features=4)
- tanh activation

Critic:

- (fcs1): Linear(in_features=33, out_features=256)
- (bn1): BatchNorm1d(256, eps=1e-05, momentum=0.1)
- (fc2) : Linear(in_features=260, out_features=128)
- (fc3) : Linear(in_features=128, out_features=1)
- no activation

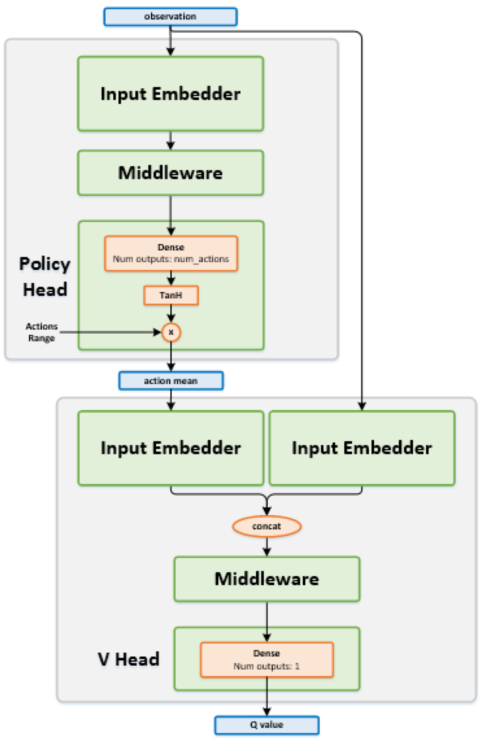


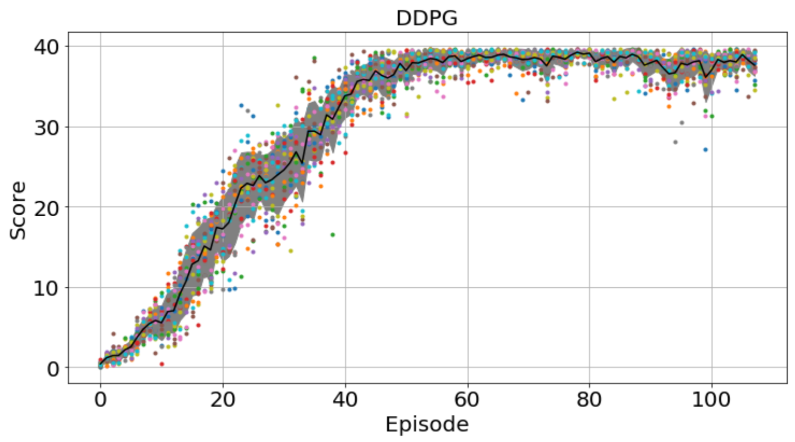
Figure 1: DDPG architecture (reference)

HyperParameters:

```
Parameters from OpenAI Baselines:
BUFFER_SIZE = int(1e6)    # replay buffer size
BATCH_SIZE = 128          # minibatch size
GAMMA = 0.99              # discount factor
TAU = 0.005               # for soft update of target parameters
LR_ACTOR = 0.001          # learning rate of the actor
LR_CRITIC = 0.001         # learning rate of the critic
WEIGHT_DECAY = 0          # L2 weight decay
UPDATE_EVERY=50           # Number of env interactions that should
                            elapse between gradient descent updates
UPDATE_AFTER=BATCH_SIZE   # Number of env interactions to collect before
                            starting to do gradient descent updates
WARM_UP=0                 # Number of steps for uniform-random action selection,
                            before running real policy. Helps exploration.
CLIP_NORM=True            # Apply gradient clipping
EPSILON = 1.0             # Exploration noise coefficient
EPSILON_DECAY = 1e-6      # Decay rate for exploration noise
```

RESULTS

I had trouble connecting and maintaining connection to the Udacity GPU server, so I had to train the network locally which took on average **4 hours on a CPU**. During training, I monitored moving average of all episodes and stopped training when the average of all episodes reached 100. Basically, I trained much longer than needed because I wanted to monitor the behavior and make sure performance does not degrade after reaching the target score. The figure below shows the ***score of each episode averaged over all agents (rubric)***. Individual agent scores for each episode are shown as colorful dots. The mean of all agents for each episode is shown as a black line and standard deviation is shown with a shaded gray region.



Parameter Selection: Performance varies greatly by the choice of hyper-parameters -- the most important being exploration noise and replay buffer size. I originally started with the **parameters used in the OpenAI baseline** which are different than the DDPG paper. DDPG paper does not provide enough information about the architecture and the training setup. I initially did not get good results with the parameters on the paper. Using batch normalization on all layers of the Critic Network and all layers prior to the action input in the Actor Network gave poor results. I ended up using **batch-normalization only after the first fully connected layer**. Using a weight decay in the optimizer also resulted in worse performance, so I set the **weight decay to zero**. It is crucial to **update the network less frequently** than advised in the paper. Additionally, **reducing the network size** helped with performance and speeded up training. Overall, DDPG is easy to implement and generalize to multiple agents, however the sensitivity to the choice of parameters, with no noticeable pattern in how and when performance degrades, is a big drawback. It would take me a fraction of the time to solve this problem efficiently and robustly using traditional robotics techniques.

IMPROVEMENTS:

- **Prioritized replay**: Sampling of the replay buffer can be prioritized based on each sample's TD error, thereby biasing sampling to samples with higher TD error or rare samples of the state space.
- **Hyperparameter selection**: More systematic way of selecting optimal hyperparameters using Bayesian optimization or starting with grid search. Not being able to connect to a GPU restricted this option for me.
- **Other algorithms:** DDPG required a lot of tweaking. The next step is to try the algorithms in OpenAI benchmarks such as Proximal Policy Optimization (PPO), Distributed Distributional Deep Deterministic Policy Gradient (D4PG), and twin-delayed DDPG (TD3).