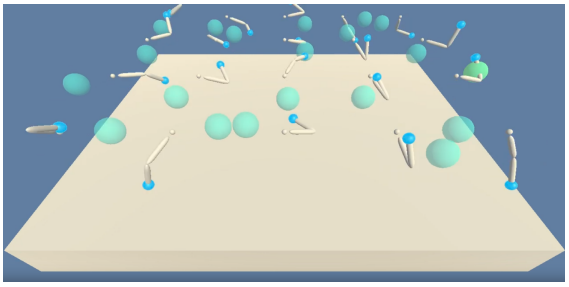
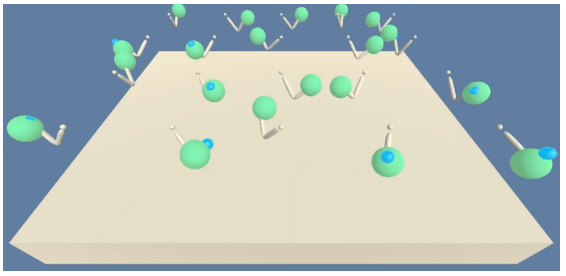
Continuous Control -- Ball Controlling Arm

Before Training (Click for video)	After Training (Click for video)
	

Overview

The primary motivation for this project was to implement a reinforcement learning algorithm in a continuous state-space and continuous action-space.

Environment

As mentioned in SETUP.md, the environment being worked in here is of Reacher.

In this environment, a double-jointed arm can move to target locations by adusting its joints. A reward of +0.1 is provided for each step that the agent's hand maintains contact with the moving ball (identified by a green colored ball). Every step the agent loses contact with the ball (identified by a light-blue colored ball) gets a reward of 0. The goal of the trained agent is therefore to adjust the movement of its arm so as to maintain contact with the ball for as many time steps as possible, which would mean that visually I should be able to see the ball(s) as being *green* most of the time.

Observation Space

The observation space is of 33 variables, corresponding to the position, rotation, velocity, and angular velocities of the arm and the ball. For example:

```
[ 0.00000000e+00 -4.00000000e+00  0.00000000e+00  1.00000000e+00
 -0.00000000e+00 -0.00000000e+00 -4.37113883e-08  0.00000000e+00
  0.00000000e+00  0.00000000e+00  0.00000000e+00  0.00000000e+00
  0.00000000e+00  0.00000000e+00 -1.00000000e+01  0.00000000e+00
  1.00000000e+00 -0.00000000e+00 -0.00000000e+00 -4.37113883e-08
  0.00000000e+00  0.00000000e+00  0.00000000e+00  0.00000000e+00
  0.00000000e+00  0.00000000e+00  5.75471878e+00 -1.00000000e+00
  5.55726624e+00  0.00000000e+00  1.00000000e+00  0.00000000e+00
 -1.68164849e-01]
```

Action Space

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. For example:

```
[ 0.00212 -0.2234 -0.98 0.00119]
```

Agent Count

The environment used here has 20 agents, each contributing an independent observation to the training process. The reason for having 20 agents was to gather more diverse experiences tuples, so that the agents can learn from each other's experiences. As you will see in the training section, the results were quite promising. The same algorithm would take far longer to train using a single agent, primarily because of the lack of exploratory experiences.

Goal

The average score of the learned policy (averaged over 100 episodes, over all 20 agents) should be 30+.

Design

I have structured the solution using the below components (classes).

Driver (driver.py)

This is the command-line entry-point to launch the training system. The goal was to mimic the IPynb Notebook environment without having to use the notebook, which greatly facilitated development and testing.

Trainer (trainer.py)

Encapsulates the train() and play() routines.

The train() routine runs the simulation in train mode, and coordinates the iteration through episodes, and the interplay between the Environment and the Agent.

The play() routine runs the simulation in play mode, for just one episode, querying the (presumably trained) agent for the next action to take for every environment state observed.

Tracker (tracker.py)

Encapsulates, collects and tracks various metrics through the training process, and can be used to generate a various types of graphs about the training.

Agent (agent.py)

Encapsulates the Deep Deterministic Policy Gradient algorithm in reinforcement learning, with the following key components:

- Experience replay buffer: A deque that stores experience tuples (state, action, reward, state', done)
- Noise generator: To add stochasticity to the actions selected by the agent
- Learning: This is the meat of the ddpg implementation, utilizing 2 neural networks -- one for the Actor and another for the Critic.

Actor Network (model.py)

A neural network used to directly learn the optimal policy (low variance, low bias) for the environment.

Critic Network (model.py)

A neural network that learns the value-based function (low variance, low bias) of the environment, and is used as a stabilizing supervisor for the policy learned by the Actor. The agent also uses the Actor's selected actions and corresponding rewards to update the Critic's network (to approximate an accurate value function for the environment).

Training

Single-Agent environment vs Multi-Agent environment

A single-agent environment trained very slowly (almost negligibly).

Additionally, the GPU utilization was rather low. The bottleneck was the CPU, since it was spending a lot of cycles between training runs simulating the actions in the environment, and copying training data back-and-forth to the GPU. The GPU, instead, was barely being used, at 10%.

Switching to the 20-agent environment improved the GPU utilization. Since each step generated 20x more data, the CPU's simulation of each step became less of a bottleneck per batch of training data. The GPU's utilization increased to ~25%, which was a significant improvement over the prior one. However, increasing this utilization requires more investigation and is listed as a future enhancement.

Using 20 agents also boosted the 'exploratory' aspect of the DDPG algorithm (without sacrificing exploitation of the learned policy at each step). This was because each agent would observe a different environment state and would therefore experience very varied experience tuples using the same policy model. And, since experiences from all agents were used to train that shared policy model, these agents learned from each other at every training run. This saw much faster convergence to an optimal policy.

Hyperparameters

The following hyperparameters were used:

```
BATCH_SIZE = 514        # minibatch size
BUFFER_SIZE = int(1e6)  # replay buffer size
GAMMA = 0.99            # discount factor
TAU = 0.2               # 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 = 0.00     # weight decay
```

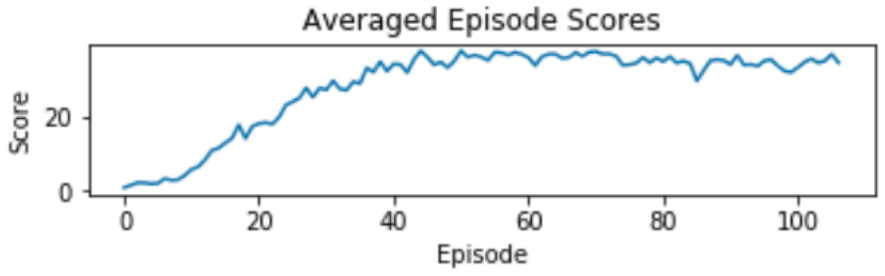
Here is a sample of the learning agent's improvement during training, which includes the episode-by-episode and 100-episode scores as well. As you can also see, the color of the balls gradually changes from light blue to green, as the training progresses.

Results

As the graphs below illustrate, the agent learned the problem space pretty well, achieving the goal score in a little over 100 episodes. See here for the training log.

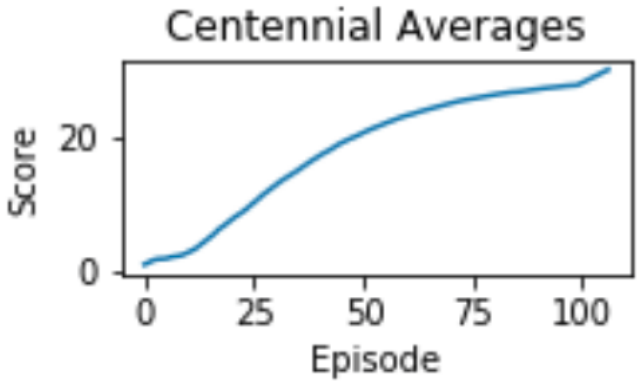
Average Agent Scores

The average scores for 20 agents, at the end of each episode, are plotted here. Not the steady increase until a score of ~38 is reach, after which the learning seems to plateau out. The maximum score that an agent can receive in an episode in this environment is 100 (1001 steps * 0.1 per step = 100). Clearly, 38 is not the highest score the agent should be able to hit, but for the sake of our goal above, it suffices.



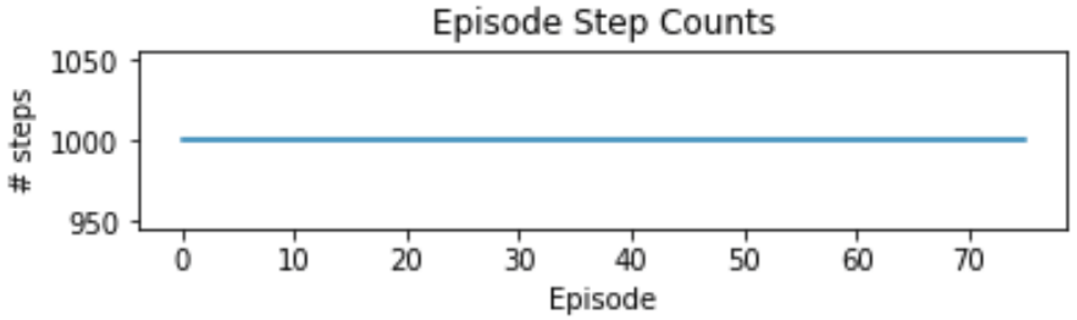
Running 100-episode averages

This is a clean and smooth curve that agrees with the Average scores graph above. Based on the above graph, the agents rapidly learn how to score points, and cross an average score of 30 around the 35th episode, after which point the mean rises as far as 37.665 around episode 50. So, using those numbers, it is not hard to see that the centennial (averaged over 100-episodes) score of 30+ in 106 episodes is indeed accurate.



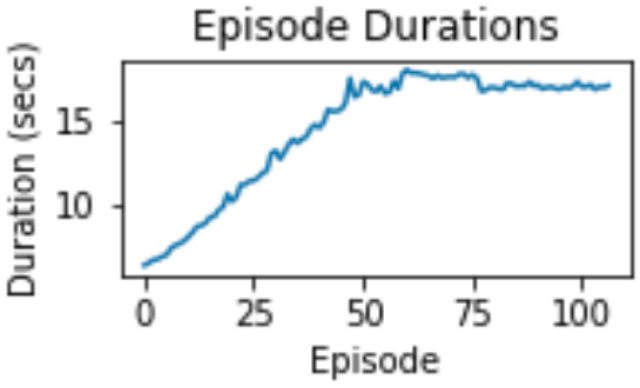
Episode step counts

This graph makes sense since all episodes in this environment run for exactly 1001 steps each, before the agent gets scored for that episode.



Episode duration

The time it took to complete each episode seems to have constantly risen through the training. The culprit here is most likely to be the sampling efficiency (or inefficiency) that (understandably) is linearly correlated with the length of the sample pool (O(n) runtime). In the case of the agent, its replay buffer is set to a size of 1e6 (=1,000,000 entries). As the agents gains experience, this replay buffer gets increasingly filled with experience tuples, until the ~50th episode, at which point the replay buffer would contain 50*1000*20 = exactly 1,000,000 entries! The 'Episode Duration' graph above supports this hypothesis, since the episode runtime hits a plateau at ~50th episode, after which it is maintained (since the buffer will not grow past 1,000,000 entries).



Future Enhancements

Representing Actions as Gaussian distributions

Instead of representing the action as a continuous-valued tensor, a potentially more generalized system might utilize each of the continuous action-values as the mean of a corresponding Gaussian distribution, by outputing a 2nd parallel vector to represent the variances. By penalizing the network for higher variance values, the system could learn more nuanced control of the ball, while also more closely mimicking learning of motor functions in humans.

Prioritized experience replay

The learning rate could be further improved using this, especially with the 20-agent environment where a lot of redundant data would be generated at the earlier parts of the learning process, and having better sampling from those experiences would produce an even faster learning curve.

Better Tooling

This is not a very user-friendly application at the moment, since the focus has been on exploring the learning process via iPython Notebooks. However, adding command-line parameters to launch the training and simulations from the shell would signficantly improve its utility as a *model-free* policy learner.

Better GPU utilization

The GPU capped at ~25% utilization with my present setup. A future enhancement would be to explore ways to increase that utilization while maintaining the CPU at its present level of utilization.

Deeper neural network models

The models used in this exercise had only 1 hidden layer (3 fully-connected layers in total). The first layer had 400 neurons, and the second had 300. The third was essentially constrained by the output size of 4 (for the action selected by the Actor) and 1 (for the state-action value estimated by the Critic). Adding additional depth to this network, could perceivably reduce the bias that is causing the existing models to saturate at an agent score between 30 - 40, even though the max score possible is 100.