DRLND - Continuous Control

Learning Algorithm

The learning algorithm used in this project is the Proximal Policy Optimization (PPO) algorithm. PPO is a policy gradient algorithm which uses the following improvements for better learning:

- For efficiency, reuses trajectories sampled from an older policy, and re-weights them based on how likely they are with respect to the current policy.
- Clips the surrogate objective function obtained as a result of the above re-weighting, to ensure that the surrogate does not continue to have a gradient once a point near the true optimum is reached.
- Uses multiple trajectories sampled from a single policy, to get a better Monte-Carlo estimate of the expected return (which is the objective we are trying to maximize).
- Attempts to assign credit to actions by using a return value which includes only the immediate reward obtained by executing the action as well as all future rewards henceforth (i.e. no rewards from the past).

For this project, I used the PPO implementation available in the Shangton Zhang DeepRL repository (https://github.com/ShangtongZhang/DeepRL). I made some minor modifications to the PPO agent class (*DeepRL/deep_rl/agent/PPO_agent.py*) by adding some extra fields (*rewards_deque* and *rewards_all*) which kept track of the averaged episodic reward (over all 20 agents) after every episode. I also created a custom extension of *BaseTask* (*ReacherMultiTask*) which suitably wrapped the provided Unity environment.

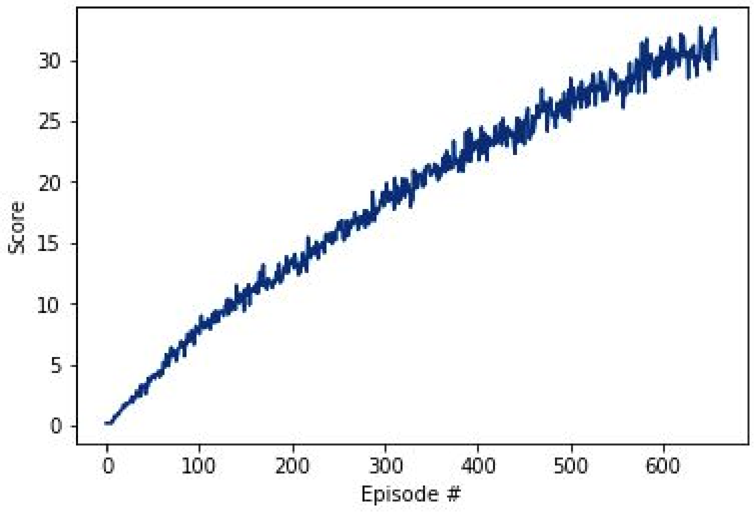
The following were the hyper-parameters chosen for this algorithm:

- Adam optimizer with a learning rate of 3e-4.
- Discount rate (gamma) of 0.99
- A tau value of 0.95 for soft-update of target networks.
- Gradient clipping using an absolute threshold of 5.
- An epsilon of 0.2 for clipping the surrogate objective.
- A maximum number of steps equal to 2e7.

The network architecture used was an actor-critic setup. The actor was a simple fully connected network with two hidden layers (each with 64 units) and a RELU activation. The output layer of the actor had as many units as the dimension of the action-space (4), followed by a tanh activation (keeping the action constraints in mind). Some Gaussian noise was further added to the predictions of the actor network. The critic was also a fully connected network with two hidden layers (each with 64 units) and a RELU activation. The output layer of the critic had a single unit (since it was emitting a scalar prediction for the value function), with a linear activation.

Results

The following is a plot of averaged episodic rewards over time. Note that each point here is an average taken over 20 agents. Also note that towards the end the curve consistently stays near 30 (i.e. the mean over the final 100 points is greater than 30), resulting in the environment being solved as per our definition.



A total of 658 episodes were needed to solve the environment. Note that in this case an episode is actually 20 parallel episodes being played concurrently by the 20 agents, and the episodic reward is the average episodic reward over all 20 agents.

Future Work

The training itself was very stable, and the mean reward steadily moved upwards towards the goal of 30 without any of the massive fluctuations which are typical in RL. However, the training was also rather slow, since the time taken to solve the environment was rather large compared to the benchmark implementation as well as what other students were managing to achieve (150 episodes or so). I would like to improve on the training time without sacrificing too much stability. I can think of a couple of approaches I could try:

- Tinker with the hyper-parameters which impact the convergence speed, e.g. the optimizer learning rates.
- Try a different algorithm which might converge faster. E.g. the benchmark and the other student both used DDPG.