Project Navigation

Provided code

First two cells provide the code to install packages for the environment and the environment itself. Environments contain ***brains*** which are responsible for deciding the actions of their associated agents. In the third cell we check for the first brain available, and set it as the default brain we will be controlling from Python. There are two environments to choose: with single agent and with 20 agents training and exchanging experiences at the same time.

In the fourth cell we access some information about the environment: how many agents, actions, states and what the state space looks like.

In the last cell of provided code we let the agent take random actions and check the score, which is always 0.

Model

Model is comprised of two networks: Actor and Critic. Actor makes predictions of the next agent's action and Critic evaluates that prediction based on generated experiences from Replay Buffer. Both networks learn simultaneously as the agent generates more experiences. Critic net tries to predict Q values from the state-action pairs and computes the loss comparing the prediction to the Q value from the experience. Actor net tries to predict the action based on the state and uses the Critic's loss when applying that action in state-action pair.

```
def learn(self, experiences, gamma):
    """Update value parameters using given batch of experience tuples.
    Params
    ======
        experiences (Tuple[torch.Variable]): tuple of (s, a, r, s', done) tuples
        gamma (float): discount factor
    """
    states, actions, rewards, next_states, dones = experiences

    #---------------------- update critic ----------------------#
    # Get predicted Q values (for next states) from target model
    actions_next = self.actor_target(next_states)
    Q_targets_next = self.critic_target(next_states, actions_next)

    # Compute Q targets for current states
    Q_targets = rewards + (gamma * Q_targets_next * (1 - dones))

    # Get expected Q values from local model
    Q_expected = self.critic_local(states, actions)

    # Compute loss
    critic_loss = F.mse_loss(Q_expected, Q_targets)
    # Minimize the loss
    self.critic_optimizer.zero_grad()
    critic_loss.backward()
    self.critic_optimizer.step()

    #---------------------- update actor ----------------------------#
    # Compute actor loss
    actions_pred = self.actor_local(states)
    actor_loss = -self.critic_local(states, actions_pred).mean()

    # Minimize the loss
    self.actor_optimizer.zero_grad()
    actor_loss.backward()
    self.actor_optimizer.step()
```

I noticed that the model is performing better when Actor's net is deeper and Critic's net is wider. So I used 5 fully connected layers for Actor with 128 nodes for the first and second layers and 64 nodes for the third and fourth. Critic has 3 fully connected layers with 256 nodes each. I also used batch

normalization as it was said in the paper that it helps with learning. The reset_parameters initializes the weights for both Actor and Critic so that the agent doesn't start from any random value.

Agent

Agent contains class ReplayBuffer which records every step (action taken) to its memory. When memory has enough of examples (BATCH_SIZE) they are passed to the model to learn. In learning step model computes targets and compares them to the states from the memory with Mean Squared Error, which is then used for backpropagation and optimizer step to update the weights. Increasing it to 128 seemed to improve the learning.
Every few time steps (UPDATE_EVERY), new batch of random samples is generated from the memory and is passed to the model to learn. The higher value generates more quality data, but makes the model learn less per episode (episode takes less time as well) so I chose the value of 20 which was in the upper range of suggested values.
When the agent acts it adds noise to the prediction and clips the output value between -1 and 1.
I didn't notice any change in performance from changing the value of GAMMA between 0.95 and 0.99 as was suggested.
Increasing BUFFER_SIZE to 1e6 seemed to improve performance.
ACTOR_LR I kept at 1e-4, CRITIC_LR at 1e-3, and TAU at 1e-3, as these were optimal values based on the lectures' material and research papers.

"ddpg" Method

This is the same runner function that was used in the previous projects. It gets the state from the environment, sends it to the agent to determine the action, sends it back to the environment and returns the reward. Then, it adds all the rewards and calculates the average score for the agent. Couple differences from the runner function in Bananas project are the absence of epsilon policy, because the action space is not limited (although between -1 and 1) and instead, the agent adds noise using Ornstein–Uhlenbeck formula when taking an action. Also, the environment in this project allows using 20 agents simultaneously that exchange experience among one another and state, action, reward, next_state and done have the shape of a tensor instead of a value and needing to be looped through.

When I started writing this report the agent was still training and hasn't reached the score of 30 yet. For several days I was trying all kinds of different parameters and hyperparameters. The agent would either not learn at all or would stuck with the score between 10 and 20. I also tried clipping the gradient at different values between 1 and 10, reorganizing the code looping through the values for each agent or passing them all at once as a tensor, but nothing worked. Finally, I found the problem – it was my ignorance! In all previous projects (Deep Learning and NLP nanodegrees) batch normalization increased the performance of any network, so I applied it to every layer right away and didn't even try to run the code once without it. As soon as I removed it, the agent reached the score of 30 in just 134 episodes (compared to running it over night for 3000 episodes and still not getting there).

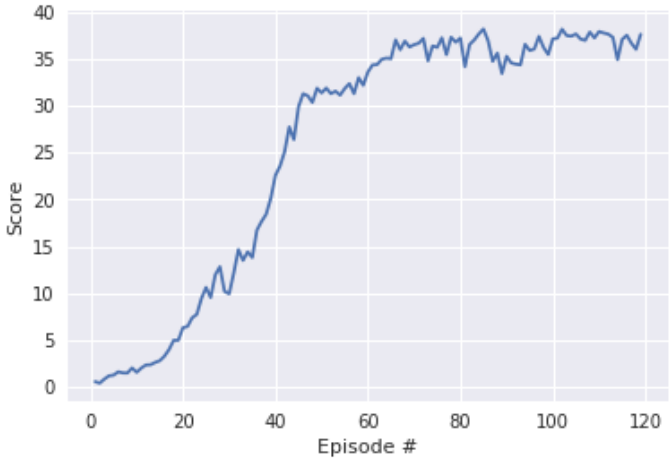
Interesting fact

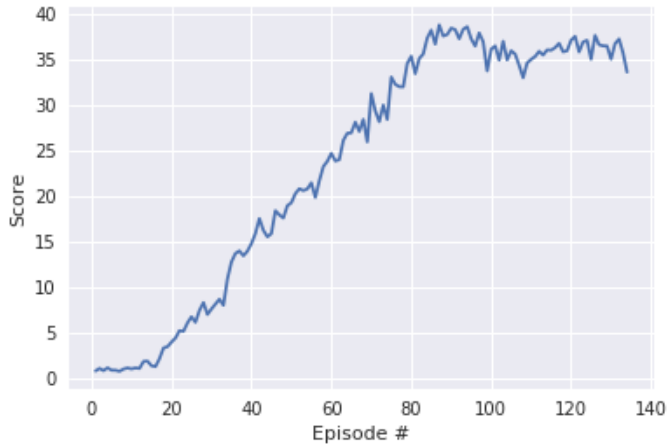
Actually, the first time I ran it, the environment was solved in 119 episodes, but I after I cleaned up displaying of some statistics and reran it, it was solved in 134 episodes. The only difference was that the first time I ran it on CPU and second time on GPU.

```
Episode 10      Average Score: 1.20
Episode 20      Average Score: 2.37
Episode 30      Average Score: 4.77
Episode 40      Average Score: 7.68
Episode 50      Average Score: 11.92
Episode 60      Average Score: 15.28
Episode 70      Average Score: 18.19
Episode 80      Average Score: 20.49
Episode 90      Average Score: 22.21
Episode 100     Average Score: 23.58
Episode 110     Average Score: 27.21
Episode 119     Average Score: 30.25
Environment solved in 19 episodes!      Average Score: 30.25
```

Compared the the second run on GPU:

```
Episode 10      Average Score: 0.92
Episode 20      Average Score: 1.52
Episode 30      Average Score: 3.14
Episode 40      Average Score: 5.30
Episode 50      Average Score: 7.70
Episode 60      Average Score: 10.02
Episode 70      Average Score: 12.42
Episode 80      Average Score: 14.80
Episode 90      Average Score: 17.25
Episode 100     Average Score: 19.22
Episode 110     Average Score: 22.64
Episode 120     Average Score: 26.03
Episode 130     Average Score: 29.04
Episode 134     Average Score: 30.11
Environment solved in 134 episodes!     Average Score: 30.11
```

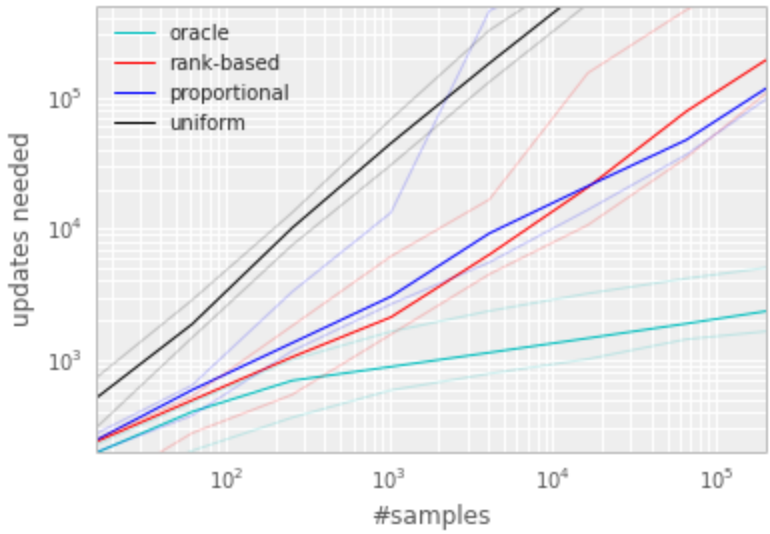




The model that succeed had 4 fully connected layers for Actor with 256, 128, 64 and 1 node; and 4 layers for Critic with 128, 128 + action size, 128 and 1 node. Buffer size was 500,000 and all other parameters the same as mentioned above. Those parameters were just the once I tried last before removing batch normalization from the model's layers and I think that parameters I mentioned before would also work, I am just too tired of this project to try them and happy to submit it with the great lesson learned – not to be ignorant to try everything.

Future Ideas to Improve the Algorithm

Experience replay (Lin, 1992) can improve learning, reduce the amount of experience and save computational resources. With experience stored in a replay memory, it becomes possible to break the temporal correlations by mixing more and less recent experience for the updates, and rare experience will be used for more than just a single update. Using a replay memory leads to design choices at two levels: which experiences to store and which to replay. This algorithm stores the last encountered TD error along with each transition in the replay memory. The transition with the largest absolute TD error is replayed from the memory. A Q-learning update is applied to this transition, which updates the weights in proportion to the TD error. New transitions arrive without a known TD-error, so we put them at maximal priority in order to guarantee that all experience is seen at least once.



Median number of updates required for Q-learning to learn the value function on the Blind Cliffwalk example, as a function of the total number of transitions (only a single one of which was successful and saw the non-zero reward). Faint lines are min/max values from 10 random initializations. Black is uniform random replay, cyan uses the hindsight-oracle to select transitions, red and blue use prioritized replay (rank-based and proportional respectively). The results differ by multiple orders of magnitude, thus the need for a log-log plot. it is evident that replaying experience in the right order makes an enormous difference to the number of updates required.