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.

Wasn't able to achieve the score of 30, given parameters made the agent get to the score of 15 in 100 episodes, but then stuck around 20 and started to decrease around episode 300