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.

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()
```

Each agent uses the same network for training, which I simplified from the last project in the process of trying to get passed the average score of 0.01 per 100 episodes (the biggest challenge I had in this project). I used 3 layers for Actor with 256 and 128 nodes for the first and second layers. For Critic I used also 3 layers with 256 nodes for both first and second.

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 in the last project without slowing it, so I decided to keep it as is.
When the agent acts it adds noise using Ornstein_Uhlenbeck formula defined in class 'OUNoise' 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.
In last project I used BUFFER_SIZE of 1e6 which seemed to improve the performance, but in this project, 100,000 was enough.
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.

MadAgent multi agent

The main idea behind this class is the initiate 2 (for this project) agents that will use the same network and learning process but keep track of their weights separately. Also, the agents generate experiences at every step in 'ReplayBuffer' class and store in common memory, learning individually from this common set of experiences. At first, I located common_memory in the Agent class, then tried to locate it in the runner function 'maddpg' passing it to 'learn' function and returning it after each run, which slowed the learning process unbearably. Then, it made sense to locate it here in MadAgent class where I initiate the multi agent calling on each Agent from ddpg file. The final version of the file (lost track of how many of them I went through in last couple days and how many times I reset Workspace trying to debug) and the one that worked ended up being the simplest one: I only left the initiations of agents and their common memory, and ONLY the functions for agent commands from the runner code keeping them as simple as possible – calling the same existing functions from the Agent class but for both (multiple) agents.

```
def reset(self):
    [agent.reset() for agent in self.agents]

def step(self, states, actions, rewards, next_states, dones):
    [self.agents[i].step(states[i], actions[i], rewards[i], next_states[i], dones[i]) for i in range(self.num_agents)]

def act(self, states):
    actions = [self.agents[i].act(np.array([states[i]])) for i in range(self.num_agents)]
    return actions
```

Trying to define all the functions in MadAgent, especially 'step' and 'learn' only returned errors, or could never get the average score of 100 episodes passed 0.01 returning straight 0's most of the time.

```
Episode 8900    Average Score: 0.01
Episode 9000    Average Score: 0.00
Episode 9100    Average Score: 0.00
Episode 9200    Average Score: 0.00
Episode 9300    Average Score: 0.01
Episode 9400    Average Score: 0.01
Episode 9500    Average Score: 0.00
Episode 9600    Average Score: 0.00
Episode 9700    Average Score: 0.01
Episode 9800    Average Score: 0.00
Episode 9900    Average Score: 0.01
Episode 10000   Average Score: 0.00
```

“maddpg” 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. After each episode, it adds up the rewards that each agent received (without discounting), to get a score for each agent. This yields 2 (potentially different) scores. Then it takes the maximum of these 2 scores which yields a single score for each episode, and then the average of lest 100 scores is computed and shown in the learning statistics column.

Biggest challenge

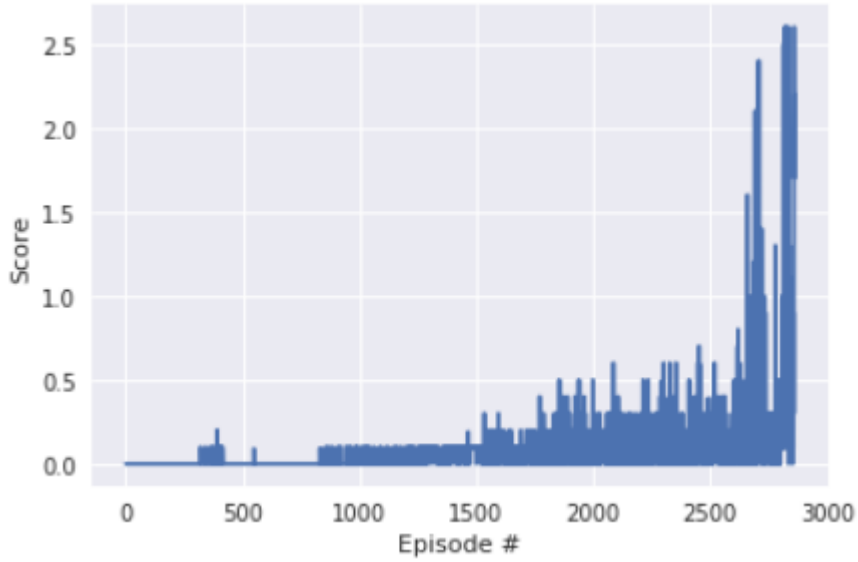
Every few time steps (UPDATE_EVERY), new batch of random samples is generated from the memory and is passed to the model to learn. The higher value generates more quality data, but makes the model learn less per episode (episode takes less time as well). In Bananas Project the value of 8 worked the best, in Continuous Control Project the value of 20. In this project I could not get an average score more than 0.01 for couple days of changing things around and tuning every parameter. First, I noticed that if I set UPDATE_EVERY to anything above 4, the agent doesn't even score 0.01 in the column of statistics, there are only 0.00's. So I tried setting it to 2 and I started seeing a lot more 0.01's than 0.00's during training. Then, I removed UPDATE_EVERY condition completely and made the agent learn at every step which immediately gave me 0.02 after couple hundred episodes.

I feel like the best step towards the solution was to simplify MadAgent class. Even though I was getting the same result as before, this gave me confidence that there is no problem in that code. Also, Agent class was proven to work in the previous project and I was confident that there was no problem in it as well. So I started trying different parameters and hyperparameters and observing any changes in agents’ behaviors.

Eventually, the environment was solved in 2861 episodes under 1 hour of GPU time.

```
Episode 100     Average Score: 0.00
Episode 200     Average Score: 0.00
Episode 300     Average Score: 0.00
Episode 400     Average Score: 0.02
Episode 500     Average Score: 0.01
Episode 600     Average Score: 0.00
Episode 700     Average Score: 0.00
Episode 800     Average Score: 0.00
Episode 900     Average Score: 0.01
Episode 1000    Average Score: 0.02
Episode 1100    Average Score: 0.04
Episode 1200    Average Score: 0.04
Episode 1300    Average Score: 0.04
Episode 1400    Average Score: 0.07
Episode 1500    Average Score: 0.09
Episode 1600    Average Score: 0.09
Episode 1700    Average Score: 0.08
Episode 1800    Average Score: 0.10
Episode 1900    Average Score: 0.11
Episode 2000    Average Score: 0.11
```

```
Episode 2100    Average Score: 0.11
Episode 2200    Average Score: 0.11
Episode 2300    Average Score: 0.10
Episode 2400    Average Score: 0.13
Episode 2500    Average Score: 0.13
Episode 2600    Average Score: 0.13
Episode 2700    Average Score: 0.23
Episode 2800    Average Score: 0.21
Episode 2861    Average Score: 0.51
Environment solved in 2861 episodes!    Average Score: 0.51
```



Notice that as soon as agents break out of the average score of ~ 0.13 or the best score of ~ 0.8, the algorithm converges almost immediately. We humans call that “Aha moment” when all of a sudden “oh we get it now!” Of course, I read the research papers saying that agents are not stable and soon after those breakthroughs they start losing their performance quickly and that’s why we save the most optimal model’s weights.

Future Ideas to Improve the Algorithm

Another very interesting approach to try is having a common Critic to compute the loss for both agents along with the shared memory, but separate Actors learning to predict an action. Or even the same Actor predicting “left” and “right” actions with negative sign for the second agent.

Also, it would be interesting to try something like ‘Importance Sampling’ concept but in simpler version, just for the beginning of learning to get over the hump (in my case was ~ 0.13 and ~ 0.8 for average and best scores thresholds) of figuring the game out. For example, out of the batch_size in ‘sample’ function of BufferReplay calss, subtract number 10, and then, add 5 experiences with the highest rewards and 5 with the lowest to each batch. Or better, add 5 random experiences from the 100 of those with the highest rewards and 5 from the 100 with the lowest. Once the agents “figured it out”, start sampling experiences randomly.