Report

November 19, 2019

1 Collaboration and Competition

Follow the instructions below to get started!

1.0.1 Setup the Environment

Run the next code cell to setup the environment. This line might take a few minutes to run!

The environments corresponding to both versions of the environment are already saved in the Workspace and can be accessed at the file paths provided below.

Please select one of the two options below for loading the environment.

```
In [1]: workspace = 'udacity'   # set to udacity if running in the workspace
        if workspace == 'udacity':
            !pip -q install ./python

        environments = {
            'local-linux-agent': './unity_environments/Tennis_Linux/Tennis.x86',
            'local-linux64-agent': './unity_environments/Tennis_Linux/Tennis.x86_64',
            'local-macos-agent': './unity_environments/Tennis.app',
            'local-windows-agent': './unity_environments/Tennis_Windows_x86/Tennis.exe',
            'local-windows64-agent': './unity_environments/single_agent/Tennis_Windows_x86_64/Te
            'udacity-agent': '/data/Tennis_Linux_NoVis/Tennis'
        }

        env_file_name = environments['udacity-agent']
```

```
tensorflow 1.7.1 has requirement numpy>=1.13.3, but you'll have numpy 1.12.1 which is incompatib
ipython 6.5.0 has requirement prompt-toolkit<2.0.0,>=1.0.15, but you'll have prompt-toolkit 2.0.
```

1.0.2 Runtime Initialization

Import all the packages required to run this notebook and setup the Unity Environment

```
In [2]: import numpy as np
        from collections import namedtuple, deque
        from datetime import datetime

        from unityagents import UnityEnvironment

        import random
        import torch
        from buffers.replaybuffer import ReplayBuffer
        from ddpg.agent import Agent as DDPG_Agent

        import matplotlib.pyplot as plt
        %matplotlib inline

        env = UnityEnvironment(file_name=env_file_name)
```

```
INFO:unityagents:
'Academy' started successfully!
Unity Academy name: Academy
        Number of Brains: 1
        Number of External Brains : 1
        Lesson number : 0
        Reset Parameters :

Unity brain name: TennisBrain
        Number of Visual Observations (per agent): 0
        Vector Observation space type: continuous
        Vector Observation space size (per agent): 8
        Number of stacked Vector Observation: 3
        Vector Action space type: continuous
        Vector Action space size (per agent): 2
        Vector Action descriptions: ,
```

1.0.3 Get required properties from the environment

Environments contain ***brains*** which are responsible for deciding the actions of their associated agents. Here we check for the first brain available, and set it as the default brain we will be controlling from Python.

```
In [3]: # get the default brain
        brain_name = env.brain_names[0]
        brain = env.brains[brain_name]
```

1.0.4 Examine the State and Action Spaces

Run the code cell below to print some information about the environment.

```
In [4]: # reset the environment
        env_info = env.reset(train_mode=True)[brain_name]

        # number of agents
        num_agents = len(env_info.agents)
        print('Number of agents}:', num_agents)

        # size of each action
        action_size = brain.vector_action_space_size
        print('Size of each action:', action_size)

        # examine the state space
        states = env_info.vector_observations
        state_size = states.shape[1]
        print('There are {} agents. Each observes a state with length: {}'.format(states.shape[0
        print('The state for the first agent looks like:', states[0])

        # examine the rewards space
        rewards = env_info.rewards
        print('The rewards for the agents looks like:', rewards)

        # examine the done space
        dones = env_info.local_done
        print('The dones for the agents looks like:', dones)
```

```
Number of agents}: 2
Size of each action: 2
There are 2 agents. Each observes a state with length: 24
The state for the first agent looks like: [ 0.          0.          0.          0.          0.
  0.          0.          0.          0.          0.          0.
  0.          0.          0.          0.          0.          0.
  0.          0.         -6.65278625 -1.5        -0.          0.
  6.83172083  6.         -0.          0.        ]
The rewards for the agents looks like: [0.0, 0.0]
The dones for the agents looks like: [False, False]
```

1.0.5 Train the agent in the Environment

This implementation using Deep Deterministic Policy Gradients (DDPG) algorithm to solve the problem.

1.0.6 Architecture

Agent The Agent uses the "Actor-Critic" that is a method which uses funcation approximation to learn a policy for any value function to solve the continuous action space. The Actor and Critic are 2 networks that do the following: 1. Actor - Takes in a state and outputs the distribution over actions. 2. Critic - Approximate the maximizer over the Q values of the next state that trains the action

The Agent also uses a ReplyBuffer to interact with the environment for training. It uses the Eplison Greedy method to select the action with a starting epsilon of 1.0. The epsilon is decayed over the episodes using a decay rate of 0.00000001 with an ending epsilon of 0.01. - Replay Buffer: It stores the experience that the agent observes, allowing us to reuse this data later. This allows the agent to sample the transistions randomly reducing the correlation in a batch. This has been shown to stablize and improve the DQN training. The buffer is a a cyclic buffer of bounded size that holds the experiences observed recently. It also implements a .sample() method for selecting a random batch of experiences for training. Each experience stored is actually a NamedTuple containing a single a named tuple representing a single interaction with the environment. It essentially maps (state, action) pairs to their (next_state, reward, done) result. - Target network: This network is compared to the local network minimize the error better the 2 to maximize the reward to solve the task. The agent randomly initialized it at a start and it is updated everytime the buffer hits the batch size. The update is performed by picking random samples from the Replay Buffer to get the bast action for a given state. - Local network: This network takes a state as input and outputs an action. It makes use of a Ornstein-Uhlenbeck process to perform exploration of actions during the training process

It takes in the following parameters during contruction: 1. device - Allows the training to switch between running on the CPU or GPU 2. config dictinarary that contains the key values - "num_agents": number of agents for environment - "state_size": dimension of each state - "action_size": dimension of each action - "buffer_size": replay buffer size - "batch_size": minibatch size - "random_seed": random seed - "gamma": discount factor - "tau": for soft update of target parameters - "weight_decay": L2 weight decay - "learn_every": learn from replay buffer every time step - "learn_batch_size": number of batches to learn from replay buffer every learn_every time step - "grad_clip": gradient value to clip at for critic - "eps_start": starting value of epsilon, for epsilon-greedy action selection - "eps_end": minimum value of epsilon - "eps_decay": multiplicative factor (per episode) for decreasing epsilon - "print_every": Print average every x episode, - "episode_steps": Maximum number of steps to run for each episode - "mu": mu for noise - "theta": theta for noise - "sigma": sigma for noise - "actor": actor specific config object - "fc": array of input sizes for hidden layers - "learning_rate": learning rate - "critic": actor specific config object - "fc": array of input sizes for hidden layers - "learning_rate": learning rate - "target_score": target average score that the agent will conside the episode as solving the problem

Contructor The constructor of the agent takes care of creating the Actor local and target network, Critic local and target network, an Adam optimizer, Ornstein-Uhlenbeck noise.

Methods

1. act This method is called with the current state which the agent uses with the Ornstein-Uhlenbeck noise process to explore the next action to perform.
2. step This method takes in the `state, action, reward, next_state, done, timestep` values that were obtained from performing the action obtained from the act method previously. Internally it stores the inputs into the ReplayBuffer. And during every `learn_every` step it will trigger the learn method on a random sample of responses if the memory capacity is greater than the `batch_size`. The step method will loop true all the experiences for all agents in the environment to be stored in the ReplayBuffer.
3. reset This method resets the noise process during learning
4. learn_episode This method is called by the runner to learn from all time steps in an episode. The runner will call it when it finds the best episode average. It makes the network converge

faster

5. learn This method will calculate the target rewards and calculate the loss against the Q values in the local network using the `MSELoss` function. It will then perform any required optimization using the optimizer. And it will trigger the `soft_update` method. It also decays the eplison value that is used to generate the noise
6. soft_update This method will perform an update on the parameters in the target network.

Network

Actor Forward pass Network that has 3 feed forward layers and a Batch Normalization Layer: 1. fc1 - in:state_size, out:fc1_units 2. bc1 - in:fc1_units, out:fc1_units 3. relu - activation for adding nonlinearity 4. fc2 - in:fc1_units, out:fc2_units 5. bc2 - in:fc2_units, out:fc2_units 6. relu - activation for adding nonlinearity 7. fc3 - in:fc2_units, out:1

Critic Forward pass Network mapping (Stae, action) pairs -> Q-values that has 3 feed forward layers and a Batch Normalization Layer: 1. fcs1 - in:state_size, out:fcs1_units 2. bc1 - in:fcs1_units, out:fcs1_units 3. relu - activation for adding nonlinearity 4. fc2 - in:fcs1_units+action_size, out:fc2_units 5. bc2 - in:fc2_units, out:fc2_units 6. relu: activation for adding nonlinearity 7. fc3: in:fc2_units, out:1

```
In [5]: config = {
            "num_agents": num_agents,     # number of agents for environment
            "state_size": state_size,     # dimension of each state
            "action_size": action_size,   # dimension of each action
            "buffer_size": int(1e10),     # replay buffer size
            "batch_size": 128,            # minibatch size
            "random_seed": 1,             # random seed
            "gamma": 0.99,                # discount factor
            "tau": 3e-3,                  # for soft update of target parameters
            "weight_decay": 0,            # L2 weight decay
            "learn_every": 1,             # learn from replay buffer every time step
            "learn_batch_size": 1,        # number of batches to learn from replay buffer every l
            "grad_clip": 1.0,             # gradient value to clip at for critic
            "eps_start": 1.0,             # starting value of epsilon, for epsilon-greedy action
            "eps_end": 0,                 # minimum value of epsilon
            "eps_decay": 1e-8,            # multiplicative factor (per episode) for decreasing ep
            "print_every": 20,            # Print average every x episode,
            "mu": 0,                      # mu for noise
            "theta": 0.15,                # theta for noise
            "sigma": 0.01,                # sigma for noise
            "actor": {                    # actor specific config object
                "fc": [ 512, 256 ],       # array of input sizes for hidden layers,
                "learning_rate": 1e-4     # learning rate
            },
            "critic": {                   # actor specific config object
                "fc": [ 512, 256 ],       # array of input sizes for hidden layers
                "learning_rate": 1e-3     # learning rate
```

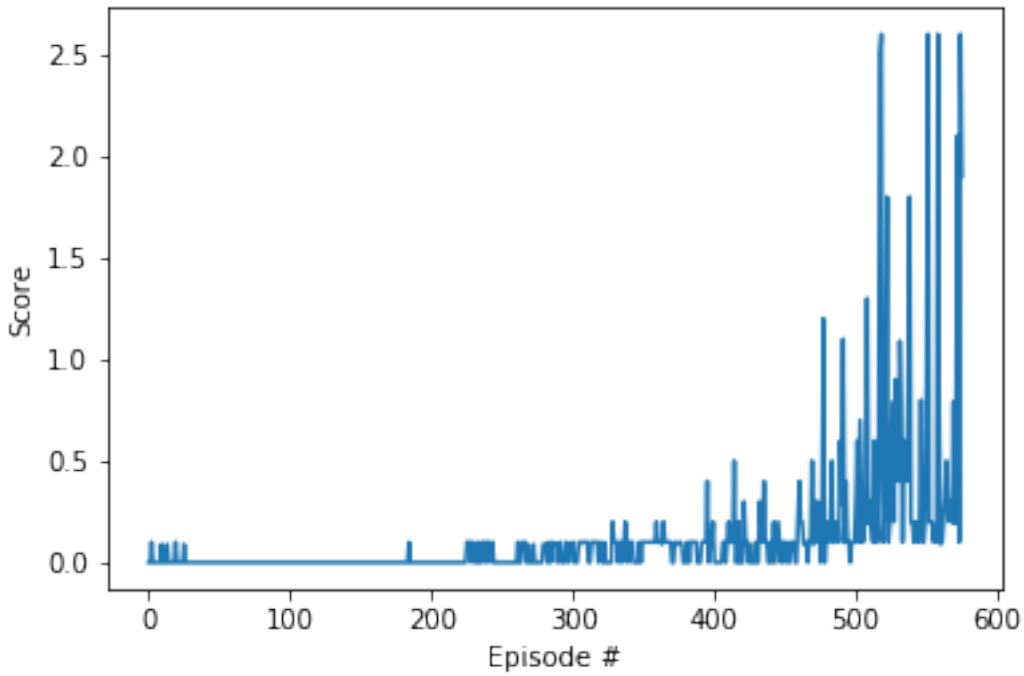
```
    },
    "target_score": 0.5          # target average score that the agent will conside the
}

def ddpg(agent, n_episodes=1000):
    best_score = 0
    max_scores = []                                    # Track best scores
    last_100_max_scores = deque(maxlen=100)            # Best scores from most recent
    last_100_average_scores = deque(maxlen=100)
    print(f'Starting training {datetime.now()}')
    for i_episode in range(1, n_episodes+1):
        start_datetime = datetime.now()
        agent.reset_episode()
        env_info = env.reset(train_mode=True)[brain_name]
        states = env_info.vector_observations
        scores = np.zeros(num_agents)
        timestep = 0
        while True:
            timestep += 1
            actions = agent.act(states, add_noise=True)
            env_info = env.step(actions)[brain_name]
            next_states = env_info.vector_observations         # get next state (for eac
            rewards = env_info.rewards                         # get reward (for each ag
            dones = env_info.local_done                        # see if episode finished
            agent.step(states, actions, rewards, next_states, dones, timestep)

            states = next_states
            scores += rewards
            if np.any(dones):
                break

        max_score = np.max(scores)
        max_scores.append(max_score)
        last_100_max_scores.append(max_score)
        if max_score >= best_score:
            best_score = max_score
            agent.learn_best_episode()

        if max_score >= config["target_score"]:
            agent.learn_best_episode()

        print(f'\rEpisode {i_episode}\tScore: {max_score:.2f}\tBest Score in all episode

        mean_last_100_max_scores = np.mean(last_100_max_scores)
        last_100_average_scores.append(mean_last_100_max_scores)
        if mean_last_100_max_scores >= config["target_score"]:
            print(f'\nEnvironment solved in {i_episode-100:d} episodes!\tAverage Score:
            torch.save(agent.actor_local.state_dict(), 'checkpoint_actor.pth')
```

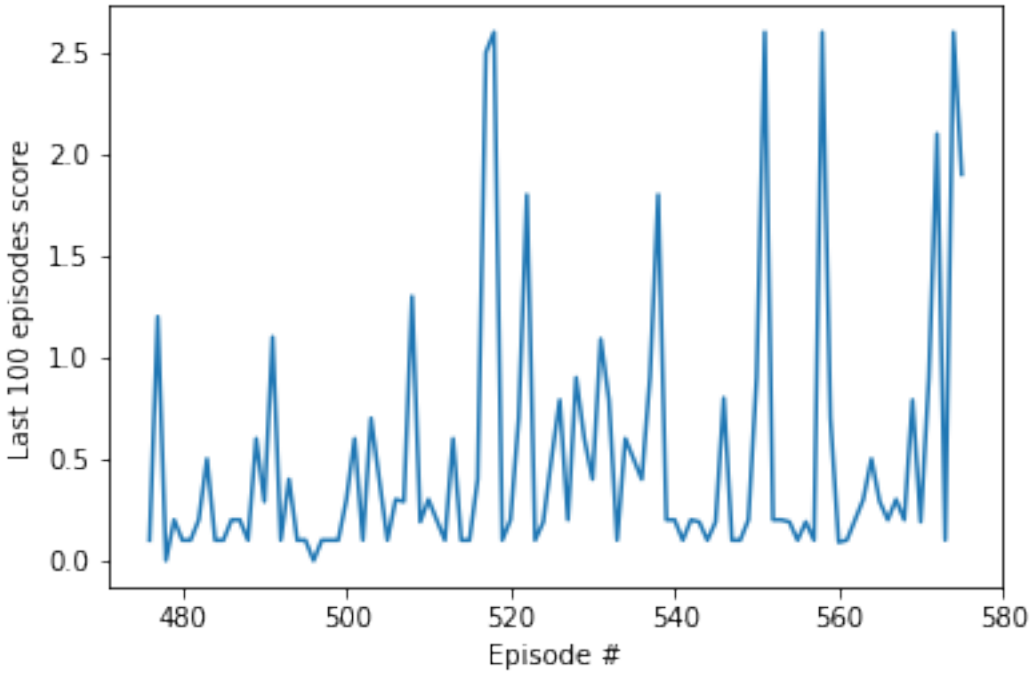
```
                    torch.save(agent.critic_local.state_dict(), 'checkpoint_critic.pth')
                    break

                if i_episode % config["print_every"] == 0:
                    print(f'\rEpisode {i_episode}\tLast 100 Episodes Average Scores: {mean_last_

            return max_scores, last_100_max_scores, last_100_average_scores, i_episode

        device = torch.device("cuda:0" if torch.cuda.is_available() else "cpu")
        agent = DDPG_Agent(memory=ReplayBuffer(device=device, config=config), device=device, con
        max_scores, last_100_max_scores, last_100_average_scores, episode = ddpg(agent, n_episod
```

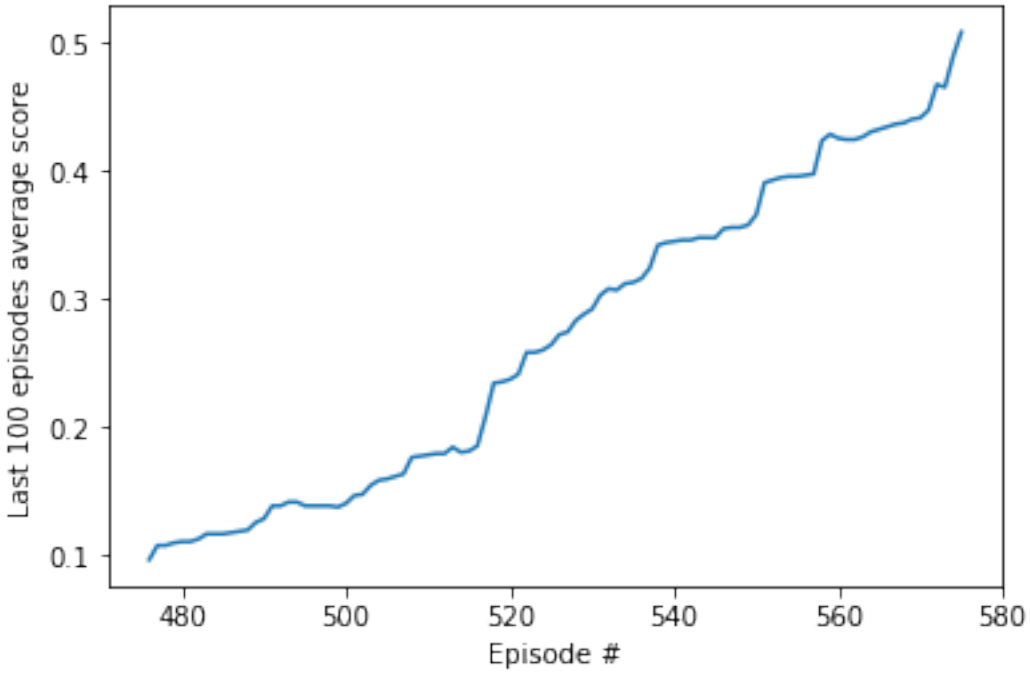
```
Starting training 2019-11-19 14:27:20.300440
Episode 20        Last 100 Episodes Average Scores: 0.02 took 0:00:00.505576
Episode 40        Last 100 Episodes Average Scores: 0.01 took 0:00:00.244517
Episode 60        Last 100 Episodes Average Scores: 0.01 took 0:00:00.251895
Episode 80        Last 100 Episodes Average Scores: 0.01 took 0:00:00.234612
Episode 100        Last 100 Episodes Average Scores: 0.00 took 0:00:00.241491
Episode 120        Last 100 Episodes Average Scores: 0.00 took 0:00:00.241524
Episode 140        Last 100 Episodes Average Scores: 0.00 took 0:00:00.231438
Episode 160        Last 100 Episodes Average Scores: 0.00 took 0:00:00.249882
Episode 180        Last 100 Episodes Average Scores: 0.00 took 0:00:00.223950
Episode 200        Last 100 Episodes Average Scores: 0.00 took 0:00:00.244351
Episode 220        Last 100 Episodes Average Scores: 0.00 took 0:00:00.227292
Episode 240        Last 100 Episodes Average Scores: 0.01 took 0:00:00.565536
Episode 260        Last 100 Episodes Average Scores: 0.01 took 0:00:00.223924
Episode 280        Last 100 Episodes Average Scores: 0.02 took 0:00:00.503914
Episode 300        Last 100 Episodes Average Scores: 0.03 took 0:00:00.494757
Episode 320        Last 100 Episodes Average Scores: 0.04 took 0:00:00.891465
Episode 340        Last 100 Episodes Average Scores: 0.05 took 0:00:00.236720
Episode 360        Last 100 Episodes Average Scores: 0.06 took 0:00:01.182335
Episode 380        Last 100 Episodes Average Scores: 0.07 took 0:00:00.241847
Episode 400        Last 100 Episodes Average Scores: 0.08 took 0:00:01.512084
Episode 420        Last 100 Episodes Average Scores: 0.08 took 0:00:00.237231
Episode 440        Last 100 Episodes Average Scores: 0.08 took 0:00:00.235224
Episode 460        Last 100 Episodes Average Scores: 0.08 took 0:00:01.475585
Episode 480        Last 100 Episodes Average Scores: 0.11 took 0:00:01.039863
Episode 500        Last 100 Episodes Average Scores: 0.14 took 0:00:00.805568
Episode 520        Last 100 Episodes Average Scores: 0.24 took 0:00:00.833335
Episode 540        Last 100 Episodes Average Scores: 0.34 took 0:00:00.994689
Episode 560        Last 100 Episodes Average Scores: 0.43 took 0:00:03.795777
Episode 576        Score: 1.90        Best Score in all episodes: 2.60
Environment solved in 476 episodes!        Average Score: 0.51
```

1.0.7 Results

```
In [6]: start_episode = episode - 100 if episode - 100 > 0 else 0
        fig = plt.figure()
        ax = fig.add_subplot(111)
        plt.plot(np.arange(len(max_scores)), max_scores)
        plt.ylabel('Score')
        plt.xlabel('Episode #')
        plt.show()
        ax = fig.add_subplot(211)
        plt.plot(np.arange(start_episode, episode), last_100_max_scores)
        plt.ylabel('Last 100 episodes score')
        plt.xlabel('Episode #')
        plt.show()
        ax = fig.add_subplot(311)
        plt.plot(np.arange(start_episode, episode), last_100_average_scores)
        plt.ylabel('Last 100 episodes average score')
        plt.xlabel('Episode #')
        plt.show()
```







When finished, you can close the environment.

```
In [ ]: env.close()
```

Summary Various hyperparameters were modified during the training sessions to find the sweet spot to make the agent perform better. The final run achieved the acheived the required running total of average score of 0.5 over 100 episodes. This run was solved from episode 476 to 576 with the highest score across all scenarios is 2.60 and the average hit was 5.1. The following was done in the implementation:

1. A best_scores value was tracked in the running loop, when an episode exceeds the currect best. The runner will make the agent learn from all the steps in that episode. This make the agent train faster form the testing
2. n_episodes was set to 2,000 this run was able to complete by 476, but from testing depending on the agents exploration during the run it might require to be a larger
3. eps_decay this is set to a low number that adds more noise at the start to introduce randomness that encourages exploration. As the episodes grow it slowly decays and reduces the noise to reduce action exploration
4. tau is set to a higher value and it appears to make the agent converged faster. When set to 1e-3 it takes > 900 episodes to solve the problem. Setting it to 3e-3 makes it solved around 400.

Ideas for Future Work

1. A multi agent algorithm with each agent having its own network and ReplayBuffer can be tested. The theory is that each agent in the environment should not move across the center of the tennis court pass the net. They should not require to know what is on the other side of the net.
2. Introducing Parameter noise that adds adaptive noise to the parameters of the neural network policy ranter than to its action space This injects randomness directly into the parameters of the agent, altering the types of decisions it makes such that they always fully depend on what the agent currently senses. It is suppose to help algorithms explore environments more effectively leading to higher scores and more elegant behaviours. (Reference -> https://openai.com/blog/better-exploration-with-parameter-noise/)
3. The implementation has a method that learns from the time steps every time it encounters an episode that beats the last best score found or is higher than the required average of 0.5. More exploration into the Prioritized Experience Replay described by (https://arxiv.org/abs/1511.05952) to replay important transitions more frequently, and therefore learn more efficiently might be explored.
4. Currently the learning occurs at every step for random sampling and the full scenario when the score is greater than the last highest or when the score is higher than the target 0.5. Learning on every step takes time and it might be worth looking into changing learn_every and learn_batch_size from 1 to higher numbers for the random learning ask the later scenarios start to have higher scores.

```
In [ ]:
```