# Continuous Control Project

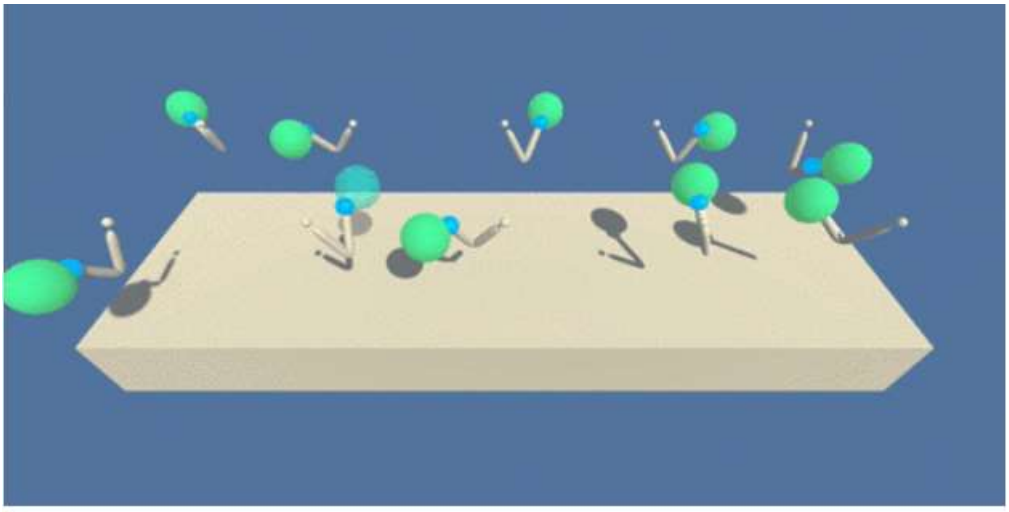
Udacity Deep Reinforcement Learning

Dhar Rawal (dharrawal@gmail.com)

2/6/2021

## Introduction

The goal of this project is to train a deep RL agent to track a ball using a robot arm. Two environments are provided. One with a single robot arm and another with 20 robot arms as shown below. I have chosen to work with the 20 agent environment given that I want to use the D4PG algorithm for training the agents.



Unity ML-Agents Reacher Environment

The agent gets a reward of 0.1 if the tip of the robot arm is inside the ball. The state space has 33 dimensions corresponding to the position, rotation, velocity, and angular velocities of the arm. The agent has to learn to select an action consisting of 4 dimensions. Each dimension of the action is a number between -1 and 1. ***In other words, the action space is continuous***.

The task is NOT inherently episodic, so we will arbitrarily stop the episode after 1000 time steps. Success will be judged by whether the agent can score 30+ averaged over last 100 consecutive time steps averaged over all 20 agents.

## Learning Algorithm

We will use the D4PG algorithm, which is a modification of the DQN algorithm (Human-level control through deep reinforcement learning) that takes advantage of learning in parallel using multiple agents. The idea is to collect training data from all agents at every timestep to train the agent AI. ***Because the***

***environment is continuous, D4PG uses an Actor-Critic RL architecture***. Refer to the DDPG paper for details on this algorithm.

## Actor Network Architecture

Pytorch is used to implement the below deep neural network in a class called *Actor*:

1. nn.Linear (33 => 128)
2. nn.Linear (128 => 128)
3. nn.Linear (128 => 4)

The first two layers have their weights initialized using a uniform distribution. The last layer weights are initialized as follows based on the reference Udacity implementation of DDPG. ***This step is critical to getting good results (Even Xavier initialization does not work in this case)***.

*self.dense3.weight.data.uniform_(-3e-3, 3e-3)*

The Adam optimizer is used for its momentum and automatic learning rate adjustment capabilities. Relu is used as the activation function for the 1st two layers, and tanh is used for the output layer because the output must be in the range [-1, 1].

## Critic Network Architecture

Pytorch is used to implement the below deep neural network in a class called *Critic*:

1. nn.Linear (33 => 128)
2. nn.Linear (128+4 => 128)
    a. Note: For this layer, the previous layer output (128) is concatenated with the actions (4)
3. nn.Linear (128 => 1)

Layer weights as initialized same as the Actor network. Note that the output here consists of a single Q (action) value.

The Adam optimizer is used for its momentum and automatic learning rate adjustment capabilities. Relu is used as the activation function for the 1st two layers, but the output layer is linear and no activation function is applied.

## The RL Agent

The RL agent is implemented in a class called *Agent*. It maintains a couple of instances of the *Actor* class and a couple of instances of the *Critic* class. The "local" actor and critic classes are actively used for selecting action values, and the "target" actor and critic classes are used to maintain a copy of weights from the immediately prior batch training call.

```
# Actor-Network

self.actorNetwork_local = Actor(state_size, action_size, seed).to(device)

self.actorNetwork_target = Actor(state_size, action_size, seed).to(device)

self.actorNetwork_optimizer = optim.Adam(self.actorNetwork_local.parameters(),
lr=LR_ACTOR)
```

```
# Critic-Network
self.criticNetwork_local = Critic(state_size, action_size, seed).to(device)
self.criticNetwork_target = Critic(state_size, action_size, seed).to(device)
self.criticNetwork_optimizer = optim.Adam(self.criticNetwork_local.parameters(),
lr=LR_CRITIC, weight_decay=WEIGHT_DECAY)
```

## ReplayBuffer

The agent also maintains the replay buffer which is as follows:

```
self.memory = ReplayBuffer(action_size, BUFFER_SIZE, BATCH_SIZE, seed)
```

Here we use a BUFFER_SIZE of 10^6 to give us enough space to store lots of episodes worth of data and a BATCH_SIZE of 256, so the data from prior 256 steps would be used for training the agent.

***This replay buffer is populated with data from all 20 agents at every time step.***

When BATCH_SIZE worth of data is gathered, training begins. Training is then done every UPDATE_EVERY steps (set at 4)

```
if len(self.memory) > BATCH_SIZE and timestep % UPDATE_EVERY == 0:
  # If enough samples are available in memory, get random subset and learn
  for _ in range(UPDATE_TIMES):
    experiences = self.memory.sample()
    self.learn(experiences, GAMMA)
```

The ReplayBuffer class provides a sample function that returns a batch of experience data sampled according to a uniform random distribution as follows:

```
def sample(self):
    """Randomly sample a batch of experiences from memory."""
    experiences = random.sample(self.memory, k=self.batch_size)

    states = torch.from_numpy(np.vstack([e.state for e in experiences if e is not
None])).float().to(device)
    actions = torch.from_numpy(np.vstack([e.action for e in experiences if e is not
None])).float().to(device)
    rewards = torch.from_numpy(np.vstack([e.reward for e in experiences if e is not
None])).float().to(device)
```

*next_states = torch.from_numpy(np.vstack([e.next_state for e in experiences if e is not None])).float().to(device)*

*dones = torch.from_numpy(np.vstack([e.done for e in experiences if e is not None]).astype(np.uint8)).float().to(device)*

*return (states, actions, rewards, next_states, dones)*

## The learn function

This is the guts of the Agent and how it learns! It basically uses the Actor Critic RL algorithm to determine a TD loss that is then used to train the *Critic*. The guts of this algorithm calculates a delayed ("fixed") target using a prior snapshot of the *Critic*.

*Qs = self.criticNetwork_local(states, actions)*

*next_actions = self.actorNetwork_target(next_states)*

*Qns = self.criticNetwork_target(next_states, next_actions)*

*TD_Target = rewards + gamma*Qns*(1-dones)  # multiply by (1-dones): Qmax_ns = 0 for terminal state*

*critic_loss = F.mse_loss(Qs, TD_Target)*

*self.criticNetwork_optimizer.zero_grad()*

*critic_loss.backward()*

*#torch.nn.utils.clip_grad_norm_(self.criticNetwork_local.parameters(), 1)*

*self.criticNetwork_optimizer.step()*

After updating the critic weights, the algorithm then updates the *Actor* network. First, the *Actor* network is used to predict the actions using the current state. The *Critic* is then used to predict the Q-value of the actions predicted by the actor. This Q-Value is directly used as the loss and this is what the actor tries to maximize.

pred_actions = self.actorNetwork_local(states)

actor_loss = -self.criticNetwork_local(states, pred_actions).mean()  ***#negative sign because we want to maximize***

self.actorNetwork_optimizer.zero_grad()

actor_loss.backward()

#torch.nn.utils.clip_grad_norm_(self.actorNetwork_local.parameters(), 1)

self.actorNetwork_optimizer.step()

As the last step, it calls the *soft_update* function to update the prior snapshots of the *Actor/Critic* using the latest iteration of the networks

```
def soft_update(self, local_model, target_model, tau):
    for target_param, local_param in zip(target_model.parameters(), local_model.parameters()):
        target_param.data.copy_(tau*local_param.data + (1.0-tau)*target_param.data)
```

**A value of 0.001 is used for TAU. This value assigns very little importance to the latest iteration of the QNetwork and maximum importance to the prior snapshot of the QNetwork, thus ensuring that the weights update very gradually and the learning improvement is stable.**

## The act function

The act function uses the Actor (local) network to predict the actions. BUT, given that the environment is NOT episodic, setting up an exponentially decaying exploration vs. exploitation parameter is not straightforward. SO…, we inject noise to the action prediction to simulate exploration.

```
self.actorNetwork_local.eval()    # switch to eval mode
states = torch.from_numpy(states).float().to(device)
with torch.no_grad():
    actions = self.actorNetwork_local(states).cpu().data.numpy()
    actions += epsilon*self.noise.sample()   # Noise injected to simulate exploration
actions = np.clip(actions, -1, 1)
self.actorNetwork_local.train()    # switch back to training mode
return actions
```

The noise function is as follows (Borrowed from the reference Udacity implementation of DDPG):

```
class OUNoise:
    """Ornstein-Uhlenbeck process."""

    #def __init__(self, size, seed, mu=0., theta=0.15, sigma=0.2):
    def __init__(self, size_tuple, seed, mu=0., theta=0.15, sigma=0.2):
        """Initialize parameters and noise process."""
        #self.mu = mu * np.ones(size)
        self.mu = mu * np.ones(size_tuple)
        self.theta = theta
        self.sigma = sigma
        self.seed = random.seed(seed)
```

```
        self.size_tuple = size_tuple
        self.reset()

    def reset(self):
        """Reset the internal state (= noise) to mean (mu)."""
        self.state = copy.copy(self.mu)

    def sample(self):
        """Update internal state and return it as a noise sample."""
        x = self.state
        #dx = self.theta * (self.mu - x) + self.sigma * np.array([random.random() for i in range(len(x))])
        #dx = self.theta * (self.mu - x) + self.sigma * np.random.random(self.size_tuple)
        dx = self.theta * (self.mu - x) + self.sigma * np.random.standard_normal(self.size_tuple)
        self.state = x + dx
        return self.state
```

## Calibrating Hyperparameters

**This was by far the most difficult and time-consuming part of the project!** I spent almost 1 week tuning the hyperparameters to obtain satisfactory performance. Below is the list of hyperparameters that were tuned and a brief summary of the effect of varying them on performance.

**Number of neurons for the Actor/Critic (1st layer, 2nd layer):**

- Variations tried: (16,16), (32,16), (32, 32), (64,32), (64, 64), (128, 128), (256, 256), (512, 256)
- Observation: The (512, 256) learned the fastest but it was show and demonstrated variability. (32, 32) and lower learnt very slowly. In the end, I settled on (128,128)

**Batch size:**

- Variations tried: 64, 128, 256, 512
- Observation: A larger batch size seems to speed up learning somewhat

**Learning rate (Actor and Critic):**

- Variation tried: 0.01, 1e-3, 1e-4
- Observation: 1e-3 exhibited a lot of variability, 1e-4 was the most stable

**Weight Initialization:**

- Variations tried: Xavier initialization, uniform random initialization 1st 2 layers with custom initialization last layer
- Observation: **Xavier initialization DOES NOT WORK! Custom initialization is required. This part seems the most magical! How did the reference implementation writers stumble upon this??**

**Noise generation algorithm:**

- Variations tried: random.random vs np.standard_normal
- Observation: np.standard_normal generates noise in [-1, 1] and it is NOTICEABLY better than random.random which generates noise in [0, 1]

**Add same vs different noise value to all agent actions:**

- Variations tried: Calculate a noise and add it to actions of all agents vs calculate 20 noise values and use a different noise value for each agent
- Observations: Adding a different noise value to each agent gives NOTICEABLY better learning performance

**"Update every/Update times":**

- Variations tried: 20/10, 10/20, 1/1, 1/5, 4/1
- Observations: 1/1 and 4/1 seem to work best although this is not a decisive hyperparameter

## HUGE GOTCHA!

I overlooked a tiny mistake when I used the DQN code to start my D4PG implementation. In my agent training code, looping over the episodes and time-steps, I was initializing the actions as follows:

*actions = agent.act(states, epsilon)**.astype(int)***

Notice the astype(int) which converts the actions returned by the agent into an integer. **HUGE MISTAKE!! This environment has a continuous action space, NOT a discrete one. NOTHING WORKED, until I fixed this issue, for obvious reasons.**

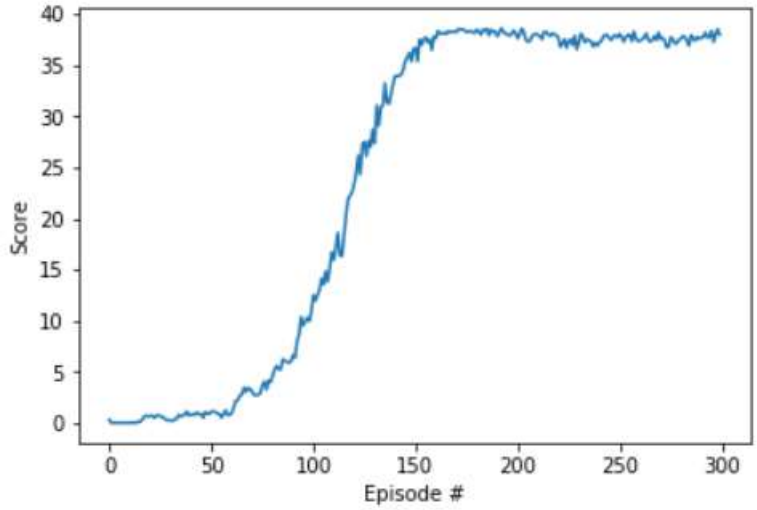
The correct implementation is as below:

*actions = agent.act(states, epsilon)*

## Plot of Rewards (Performance)

Once I fixed the gotcha above and with the final selection of hyperparameters, the agent performs well, learning to score 30+ per episode, averaged over 100 consecutive time steps within 200 episodes. Below is a snapshot of learning progress from one such run.

```
Episode 100     Average Score: 2.38
Episode 195     Average Score: 30.12
Environment solved in 195 episodes!     Average Score: 30.12
Episode 200     Average Score: 31.51
Episode 300     Average Score: 37.62
```




Here is the complete agent training code:

```
def train_d4pgAgent(env, brain_name, agent, n_episodes=100, max_t=1000):
    """D4PG for continuous control.

    Params
    ======
        n_episodes (int): maximum number of training episodes
        max_t (int): maximum number of timesteps per episode
        eps_start (float): starting value of epsilon, for epsilon-greedy action selection
        eps_end (float): minimum value of epsilon
        eps_decrement (float): subtractive factor (per episode) for decreasing epsilon
    """
    avgScores = []                  # list containing average scores over all agents from each episode
    avg_scores_window = deque(maxlen=100)  # last 100 average scores over all agents
    epsilon = 1.0
    epsilon_dec = 0
```

```
envSolved = False

for i_episode in range(1, n_episodes+1):
    agent.reset_noise()
    scores = np.zeros(num_agents)                    # initialize the score (for each agent)
    env_info = env.reset(train_mode=True)[brain_name]     # reset the environment

    states = env_info.vector_observations              # get the current state (for each agent)
    for t in range(max_t):
        actions = agent.act(states, epsilon)    # num_agents x action_size

        env_info = env.step(actions)[brain_name]      # send the action to the environment
        next_states = env_info.vector_observations     # get the next states
        rewards = env_info.rewards                 # get the rewards
        done = env_info.local_done                 # see if episodes have finished

        agent.step(states, actions, rewards, next_states, done, t)

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

    epsilon -= epsilon_dec

    avgScore = np.mean(scores)
    avg_scores_window.append(avgScore)      # save most recent average score over all agents
    avgScores.append(avgScore)           # save most recent average score over all agents
```

```
        avgScoreAllAgentsOver100Episodes = np.mean(avg_scores_window)


        print('\rEpisode {}\tAverage Score: {:.2f}'.format(i_episode,
avgScoreAllAgentsOver100Episodes), end="")


        if i_episode % 100 == 0:

            print('\rEpisode {}\tAverage Score: {:.2f}'.format(i_episode,
avgScoreAllAgentsOver100Episodes))

            torch.save(agent.actorNetwork_local.state_dict(), 'actorNetwork.pth')

            torch.save(agent.criticNetwork_local.state_dict(), 'criticNetwork.pth')


        if not envSolved and avgScoreAllAgentsOver100Episodes >= 30.0:

            print('\nEnvironment solved in {:d} episodes!\tAverage Score: {:.2f}'.format(i_episode,
avgScoreAllAgentsOver100Episodes))

            envSolved = True

            #break


    return avgScores
```

## Ideas for Future Work (Recommendations)

The next logical challenges in the project are to implement:

1. Implement n-step learning. Currently this D4PG algorithm uses 1-step learning
2. Tackle the Crawl optional project
3. Implement TRPO and TNPG for this project
4. Implement PPO for this project