Name: Oluwaseyi Awoga

Program: Udacity Deep Reinforcement Learning Nanodegree Program [First Draft - 2021]

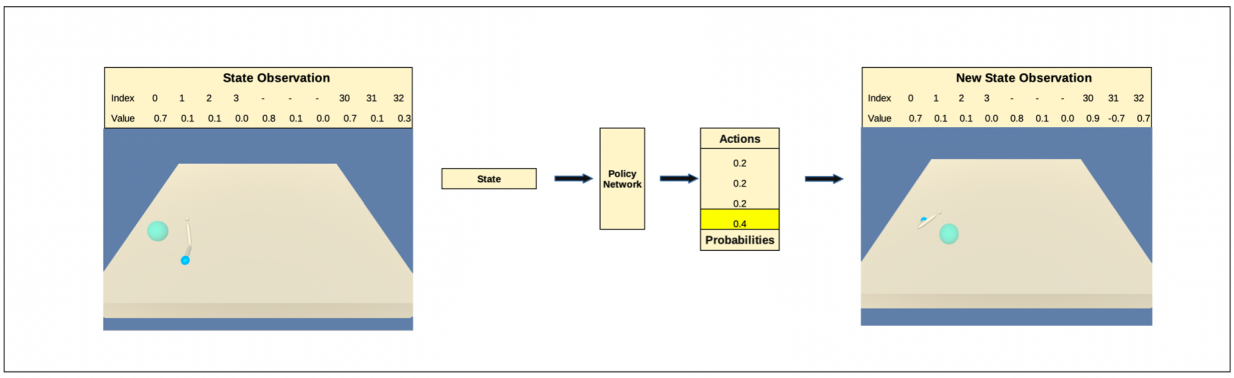
Topic: Using Deep Deterministic Policy Gradient (DDPG) to Train a Double-Jointed Arm to Reach Target Locations in the Unity ML-Agents Reacher Environment.

Introduction & Literature Review:

The Q-learning process including many of its variations such as Deep Q-Network (DQN) uses an indirect approach to train reinforcement learning agents. Generally, we calculate the Q-values for state-action pairs and then take the maximum or highest value from the list of actions possible in that environment. This is an indirect approach, perhaps inefficient or intractable for training agents with continuous action space. Put another way, Q-learning based methods are only ideal for problems with discrete action spaces.

(Google DeepMind/Lillicrap et al., 2016) in their paper entitled "Continous Control with Deep Reinforcement Learning" in fact argued that *"while DQN solves problems with high-dimensional observation spaces, it can only handle discrete and low-dimensional action spaces"*. They stated further that *"DQN cannot be straightforwardly applied to continuous domains since it relies on finding the action that maximizes the action-value function, which in the continuous-valued case requires an iterative optimization process at every step"*.

But what if we could bypass calculating Q-values and directly learn a policy from the environment? That is, instead of training our system to generate action values we instead train it to generate the probability of selecting the right actions? (Zai & Brown 2020) stated that *"this class of algorithms is called policy gradient methods"*.



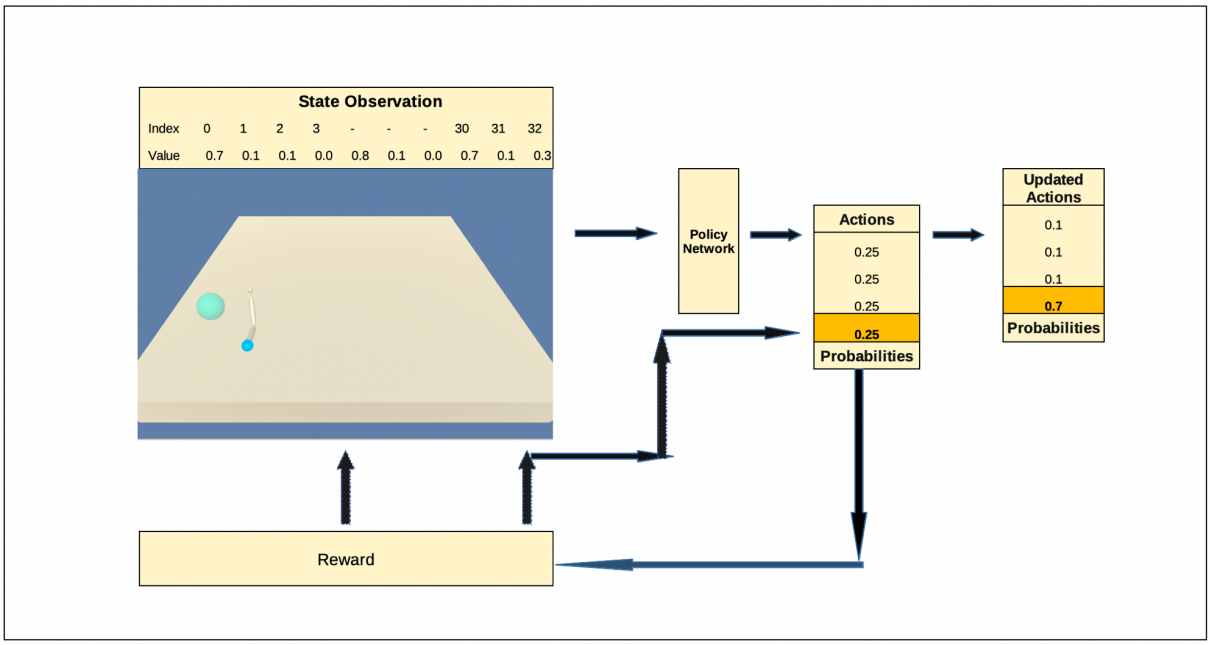
As shown above, in a simple policy gradient method, the system is trained to output probabilities of taking each action and the agent selects the action with the maximum probability and acts on it which leads the agent to a new state.

REINFORCE is the simplest form of policy gradient and it is designed to minimize the logarithm of the probability of an action a given state S multiplied by the return R which is the sum of the discounted rewards in an episode as shown in the formulae below:

$$R = \Sigma\gamma_t * r_t$$

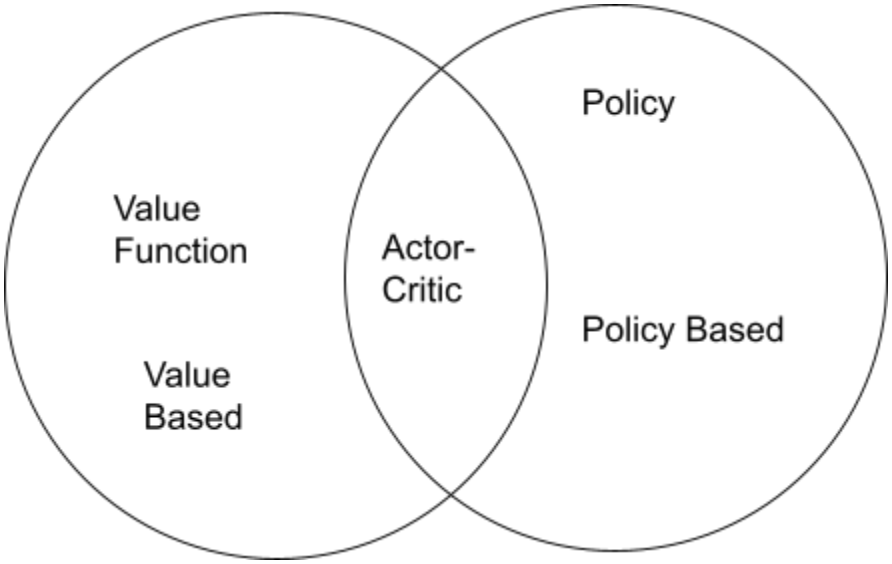
$$Loss = -log(P(a|S)) * R$$

After the discounted rewards are collected after each episode, if it is positive it will reinforce that action and nudge its probability higher. Conversely, if the reward was negative it will nudge that probability lower hence the name “reinforce”.



Once an action is sampled from the policy network’s probability distribution, it produces a new state and reward. The reward signal is used to reinforce the action that was taken, that is, it increases the probability of that action given the state if the reward is positive, or it decreases the probability if the reward is negative. Adapted from (Zai & Brown 2020).

However, the reinforce method suffers from a problem of variance of returns between episodes. That is, large differences in the returns between episodes could make the training process unstable with volatility in the losses between episodes.



The relationship between value and policy-based techniques (actor-critic methods)

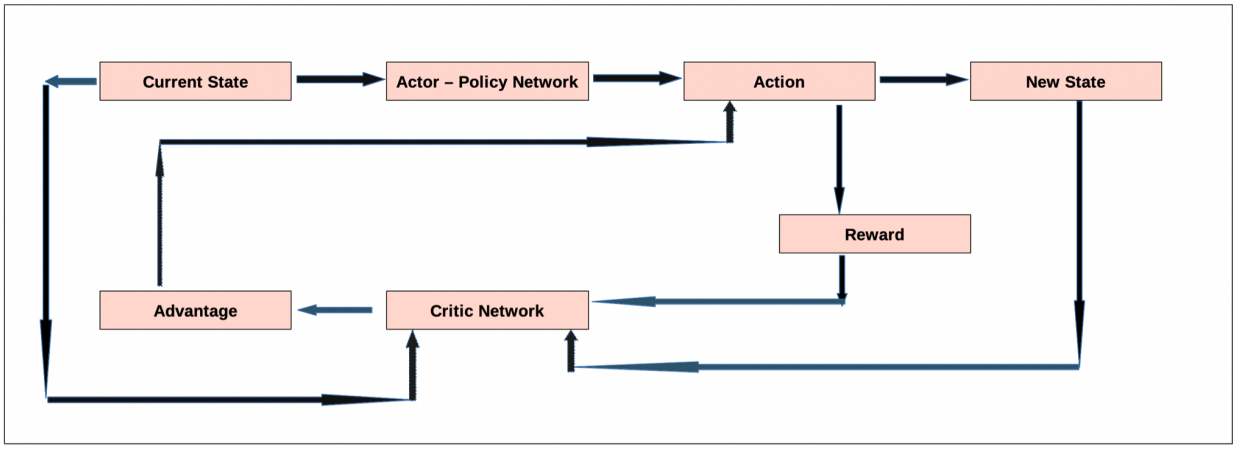
The problem with the REINFORCE method could be addressed by combining the advantages of a purely DQN method with that of a method that uses policy gradients. This combined value-policy learning algorithm attempts to overcome two challenges associated with using either a purely policy gradient approach or a DQN approach. The two main challenges are (Zai & Brown 2020):

- *Improve the sample efficiency by updating more frequently*
- *Decrease the variance of the rewards used to update the model*

Therefore, instead of minimizing the REINFORCE loss that included a direct reference to the observed return R, from an episode, we instead add a baseline value such that the loss is now (Zai & Brown 2020):

$$Loss = -log(P(a|S)) * (R - V\pi(S))$$

The V(S) is the value of state S, which is the value-value function (a function of the state) rather than an action-value function (a function of both state and action), although an action-value function bemused as well. This quantity, $V(S) - R$, is termed the *advantage*.

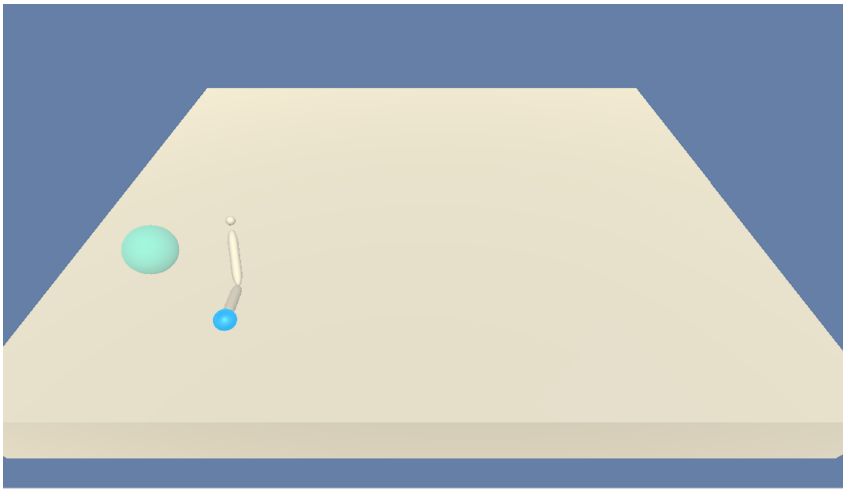


The actor predicts the best action and chooses the action to take, which generates a new state. The critic network computes the value of the old state and the new state. The relative value of the next state is called the advantage, and this is the signal used to reinforce the action that was taken by the action. Adapted from (Zai & Brown 2020).

The Unity ML-Agents Reacher Environment

For this project, we will be training a double-jointed arm to move to target locations.
Each action is a vector with four numbers, corresponding to torque applicable to two joints.
Every entry in the action vector is a number between -1 and 1.

A reward of +0.1 is provided for each step that the agent's hand is in the goal location. Thus, the goal of the agent is to maintain its position at the target location for as many time steps as possible. The observation space consists of 33 variables corresponding to the position, rotation, velocity, and angular velocities of the arm.



The abstract reinforcement learning canonical representation of the environment is shown below:

Abstract RL Concept	Realization in the Unity ML-Agents Reacher Environment
Environment	A double-jointed arm moving to target locations.
Agent	Continous Action: Each action is a vector with four numbers, corresponding to torque applicable to two joints. Every entry in the action vector is a number between -1 and 1.
Reward	A reward of +0.1 is provided for each step that the agent's hand is in the goal location. Thus, the goal of the agent is to maintain its position at the target location for as many time steps as possible.
Observation	The observation space consists of 33 variables corresponding to position, rotation, velocity, and angular velocities of the arm.

Having established some foundational background in policy gradient methods, we now implement a variation of the methodology known as Deep Deterministic Policy Gradient (DDPG).

Implementation of the DDPG Process in PyTorch

The DDPG methodology for the Reacher was implemented using the approach discussed in (Palmas et al., 2020), as discussed below:

1. Import required libraries and packages:

```
import os
import gym
import torch as T
import matplotlib.pyplot as plt
%matplotlib inline
import numpy as np
from unityagents import UnityEnvironment
```

2. Implement the Ornstein-Uhlenbeck Noise: According to (Palmas et al., 2020), “the Ornstein-Uhlenbeck process, in physics, is used to model the velocity of a Brownian particle under the influence of friction. Brownian motion is the random motion of particles when suspended in a fluid (liquid or gas) resulting from their collisions with other particles in the same fluid. Ornstein–Uhlenbeck noise gives you a type of noise that is temporally correlated and is centered on a mean of 0. Since the agent has zero knowledge of the model, it becomes difficult to train it. Here, Ornstein–Uhlenbeck noise can be used as a sample to generate that knowledge. Let's look at the code implementation of this class”:

```
class OUActionNoise(object):
    def __init__(self, mu, sigma=0.15, theta=.2,dt=1e-2, x0=None):
        self.theta = theta
        self.mu = mu
        self.sigma = sigma
        self.dt = dt
        self.x0 = x0
        self.reset()

    def __call__(self):
        x = self.x_previous
        dx = self.theta * (self.mu - x) * self.dt + self.sigma *
np.sqrt(self.dt) * np.random.normal(size=self.mu.shape)
        self.x_previous = x + dx
        return x

    def reset(self):
        self.x_previous = self.x0 if self.x0 is not None else
np.zeros_like(self.mu)
```

3. Implement the ReplayBuffer class to the agent's learning experiences:

```
class ReplayBuffer(object):
    def __init__(self, max_size, inp_shape, nb_actions):
        self.memory_size = max_size
        self.memory_counter = 0
        self.memory_state = np.zeros((self.memory_size, *inp_shape))
        self.new_memory_state = np.zeros((self.memory_size, *inp_shape))
        self.memory_action = np.zeros((self.memory_size, nb_actions))
        self.memory_reward = np.zeros(self.memory_size)
        self.memory_terminal = np.zeros(self.memory_size, dtype=np.float32)

    def store_transition(self, state, action, reward, state_, done):
        index = self.memory_counter % self.memory_size
        self.memory_state[index] = state
        self.new_memory_state[index] = state_
        self.memory_action[index] = action
        self.memory_reward[index] = reward
        self.memory_terminal[index] = 1 - done
        self.memory_counter += 1

    def sample_buffer(self, bs):
```

```
        max_memory = min(self.memory_counter, self.memory_size)

        batch = np.random.choice(max_memory, bs)

        states = self.memory_state[batch]
        actions = self.memory_action[batch]
        rewards = self.memory_reward[batch]
        states_ = self.new_memory_state[batch]
        terminal = self.memory_terminal[batch]

        return states, actions, rewards, states_, terminal
```

4. Implement the Critic Network

```
class CriticNetwork(T.nn.Module):
    def __init__(
        self, beta, inp_dimensions,
        fc1_dimensions, fc2_dimensions,
        nb_actions):
        super(CriticNetwork, self).__init__()
        self.inp_dimensions = inp_dimensions
        self.fc1_dimensions = fc1_dimensions
        self.fc2_dimensions = fc2_dimensions
        self.nb_actions = nb_actions

        self.fc1 = T.nn.Linear(*self.inp_dimensions, self.fc1_dimensions)
        f1 = 1./np.sqrt(self.fc1.weight.data.size()[0])
        T.nn.init.uniform_(self.fc1.weight.data, -f1, f1)
        T.nn.init.uniform_(self.fc1.bias.data, -f1, f1)

        self.bn1 = T.nn.LayerNorm(self.fc1_dimensions)

        self.fc2 = T.nn.Linear(self.fc1_dimensions, self.fc2_dimensions)
        f2 = 1./np.sqrt(self.fc2.weight.data.size()[0])

        T.nn.init.uniform_(self.fc2.weight.data, -f2, f2)
        T.nn.init.uniform_(self.fc2.bias.data, -f2, f2)

        self.bn2 = T.nn.LayerNorm(self.fc2_dimensions)

        self.action_value = T.nn.Linear(self.nb_actions, self.fc2_dimensions)
        f3 = 0.003
        self.q = T.nn.Linear(self.fc2_dimensions, 1)
```

```
        T.nn.init.uniform_(self.q.weight.data, -f3, f3)
        T.nn.init.uniform_(self.q.bias.data, -f3, f3)

        self.optimizer = T.optim.Adam(self.parameters(), lr=beta)

        self.device = T.device("gpu" if T.cuda.is_available() else "cpu")
        self.to(self.device)

    def forward(self, state, action):
        state_value = self.fc1(state)
        state_value = self.bn1(state_value)
        state_value = T.nn.functional.relu(state_value)
        state_value = self.fc2(state_value)
        state_value = self.bn2(state_value)

        action_value = T.nn.functional.relu(self.action_value(action))
        state_action_value = T.nn.functional.relu(
            T.add(state_value, action_value))
        state_action_value = self.q(state_action_value)

        return state_action_value
```

5. Implement the actor-network:

```
class ActorNetwork(T.nn.Module):
    def __init__(
        self, alpha, inp_dimensions,
        fc1_dimensions, fc2_dimensions,
        nb_actions):
        super(ActorNetwork, self).__init__()
        self.inp_dimensions = inp_dimensions
        self.fc1_dimensions = fc1_dimensions
        self.fc2_dimensions = fc2_dimensions
        self.nb_actions = nb_actions

        self.fc1 = T.nn.Linear(*self.inp_dimensions, self.fc1_dimensions)
        f1 = 1./np.sqrt(self.fc1.weight.data.size()[0])
        T.nn.init.uniform_(self.fc1.weight.data, -f1, f1)
        T.nn.init.uniform_(self.fc1.bias.data, -f1, f1)

        self.bn1 = T.nn.LayerNorm(self.fc1_dimensions)
```

```
        self.fc2 = T.nn.Linear(self.fc1_dimensions, self.fc2_dimensions)
        f2 = 1./np.sqrt(self.fc2.weight.data.size()[0])

        T.nn.init.uniform_(self.fc2.weight.data, -f2, f2)
        T.nn.init.uniform_(self.fc2.bias.data, -f2, f2)

        self.bn2 = T.nn.LayerNorm(self.fc2_dimensions)

        f3 = 0.003
        self.mu = T.nn.Linear(self.fc2_dimensions, self.nb_actions)
        T.nn.init.uniform_(self.mu.weight.data, -f3, f3)
        T.nn.init.uniform_(self.mu.bias.data, -f3, f3)

        self.optimizer = T.optim.Adam(self.parameters(), lr=alpha)

        self.device = T.device("gpu" if T.cuda.is_available() else "cpu")
        self.to(self.device)

    def forward(self, state):
        x = self.fc1(state)
        x = self.bn1(x)
        x = T.nn.functional.relu(x)
        x = self.fc2(x)
        x = self.bn2(x)
        x = T.nn.functional.relu(x)
        x = T.tanh(self.mu(x))

        return x
```

6. Create the agent class:

```
class Agent(object):
    def __init__(
        self, alpha, beta, inp_dimensions, tau, env,
        gamma=0.99, nb_actions=2, max_size=1000000,
        l1_size=400, l2_size=300, bs=64):
        self.gamma = gamma
        self.tau = tau
        self.memory = ReplayBuffer(max_size, inp_dimensions, nb_actions)
        self.bs = bs

        self.actor = ActorNetwork(
```

```
            alpha, inp_dimensions, l1_size, l2_size, nb_actions=nb_actions)
        self.critic = CriticNetwork(
            beta, inp_dimensions, l1_size, l2_size, nb_actions=nb_actions)
        self.target_actor = ActorNetwork(
            alpha, inp_dimensions, l1_size, l2_size, nb_actions=nb_actions)
        self.target_critic = CriticNetwork(
            beta, inp_dimensions, l1_size, l2_size, nb_actions=nb_actions)

        self.noise = OUActionNoise(mu=np.zeros(nb_actions))

        self.update_params(tau=1)

    def select_action(self, observation):
        self.actor.eval()
        observation = T.tensor(
            observation, dtype=T.float).to(self.actor.device)
        mu = self.actor.forward(observation).to(self.actor.device)
        mu_prime = mu + T.tensor(
            self.noise(),
            dtype=T.float).to(self.actor.device)
        self.actor.train()
        return mu_prime.cpu().detach().numpy()

    def remember(self, state, action, reward, new_state, done):
        self.memory.store_transition(state, action, reward, new_state, done)

    def learn(self):
        if self.memory.memory_counter < self.bs:
            return
        state, action, reward, new_state, done = \
                                        self.memory.sample_buffer(self.bs)

        reward = T.tensor(reward, dtype=T.float).to(self.critic.device)
        done = T.tensor(done).to(self.critic.device)
        new_state = T.tensor(new_state, dtype=T.float).to(self.critic.device)
        action = T.tensor(action, dtype=T.float).to(self.critic.device)
        state = T.tensor(state, dtype=T.float).to(self.critic.device)

        self.target_actor.eval()
        self.target_critic.eval()
        self.critic.eval()

        target_actions = self.target_actor.forward(new_state)
        critic_value_new = self.target_critic.forward(
            new_state, target_actions)
```

```
        critic_value = self.critic.forward(state, action)

        target = []
        for j in range(self.bs):
            target.append(reward[j] + self.gamma*critic_value_new[j]*done[j])
        target = T.tensor(target).to(self.critic.device)
        target = target.view(self.bs, 1)

        self.critic.train()
        self.critic.optimizer.zero_grad()
        critic_loss = T.nn.functional.mse_loss(target, critic_value)
        critic_loss.backward()
        self.critic.optimizer.step()

        self.critic.eval()
        self.actor.optimizer.zero_grad()
        mu = self.actor.forward(state)
        self.actor.train()
        actor_loss = -self.critic.forward(state, mu)
        actor_loss = T.mean(actor_loss)
        actor_loss.backward()
        self.actor.optimizer.step()

        self.update_params()

    def update_params(self, tau=None):
        if tau is None:
            tau = self.tau # tau is 1

        actor_params = self.actor.named_parameters()
        critic_params = self.critic.named_parameters()
        target_actor_params = self.target_actor.named_parameters()
        target_critic_params = self.target_critic.named_parameters()

        critic_state_dict = dict(critic_params)
        actor_state_dict = dict(actor_params)
        target_critic_dict = dict(target_critic_params)
        target_actor_dict = dict(target_actor_params)

        for name in critic_state_dict:
            critic_state_dict[name] = tau*critic_state_dict[name].clone() + \
                (1-tau)*target_critic_dict[name].clone()

        self.target_critic.load_state_dict(critic_state_dict)
```

```
        for name in actor_state_dict:
            actor_state_dict[name] = tau*actor_state_dict[name].clone() + \
                (1-tau)*target_actor_dict[name].clone()
        self.target_actor.load_state_dict(actor_state_dict)
```

7. Import the Unity ML-Agents Reacher Environment:

```
env = UnityEnvironment(file_name='Reacher.app')

#Output:
INFO:unityagents:
'Academy' started successfully!
Unity Academy name: Academy
        Number of Brains: 1
        Number of External Brains : 1
        Lesson number : 0
        Reset Parameters :
            goal_speed -> 1.0
            goal_size -> 5.0
Unity brain name: ReacherBrain
        Number of Visual Observations (per agent): 0
        Vector Observation space type: continuous
        Vector Observation space size (per agent): 33
        Number of stacked Vector Observation: 1
        Vector Action space type: continuous
        Vector Action space size (per agent): 4
        Vector Action descriptions: , , ,
```

8. Query some environment information about the agent to familiarize with the agent.

```
brain_name = env.brain_names[0]
brain = env.brains[brain_name]
env_info = env.reset(train_mode=True)[brain_name]

#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], state_size))
print('The State for the First Agent Looks Like:', states[0])

#Output:

Size of Each Action: 4
There are 1 Agents. Each Observes a State with Length: 33
The State for the First Agent Looks Like: [ 0.00000000e+00 -4.00000000e+00
0.00000000e+00  1.00000000e+00
 -0.00000000e+00 -0.00000000e+00 -4.37113883e-08  0.00000000e+00
  0.00000000e+00  0.00000000e+00  0.00000000e+00  0.00000000e+00
  0.00000000e+00  0.00000000e+00 -1.00000000e+01  0.00000000e+00
  1.00000000e+00 -0.00000000e+00 -0.00000000e+00 -4.37113883e-08
  0.00000000e+00  0.00000000e+00  0.00000000e+00  0.00000000e+00
  0.00000000e+00  0.00000000e+00 -6.30408478e+00 -1.00000000e+00
 -4.92529202e+00  0.00000000e+00  1.00000000e+00  0.00000000e+00
 -5.33014059e-01]
```

9. Train the agent:

```
agent = Agent(alpha=0.000025, beta=0.00025, \
              inp_dimensions=[33], tau=0.001, \
              env=env, bs=64, l1_size=400, \
              l2_size=300, nb_actions=4)

num_agents = 1
brain_name = env.brain_names[0]
env_info = env.reset(train_mode=True)[brain_name]
states = env_info.vector_observations
scores = np.zeros(num_agents)
scoresToUse = []
actions = np.random.randn(1, 4)
```

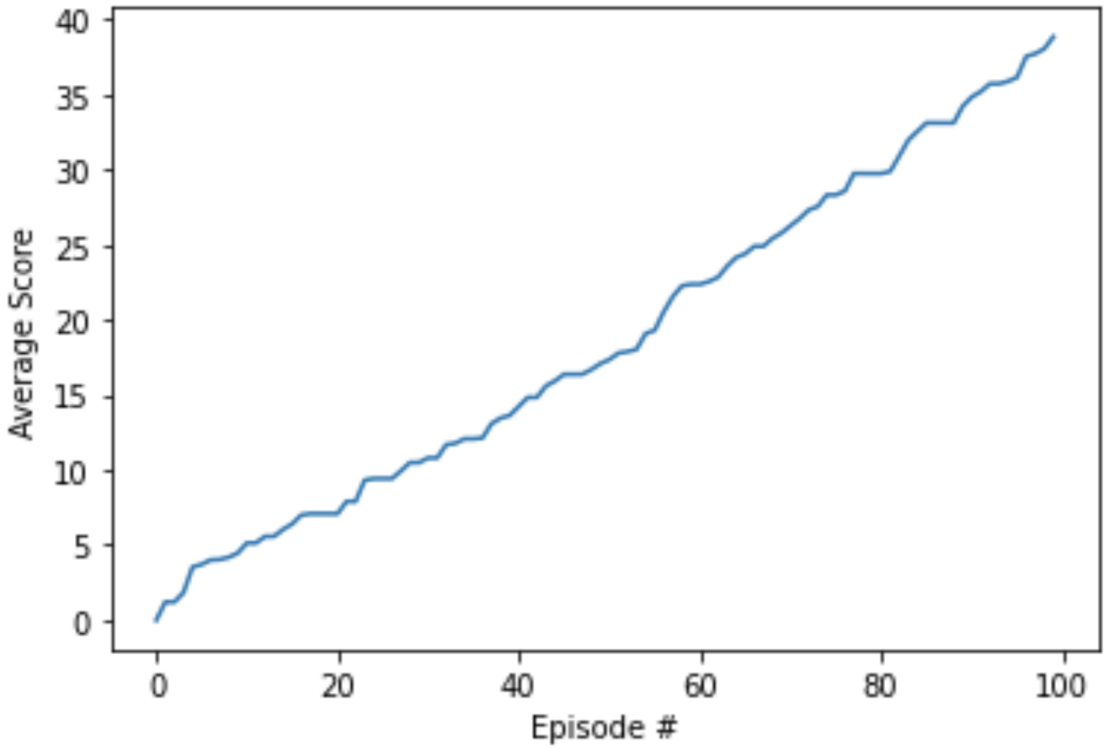
```
for x in range(0,100):
    env_info = env.reset(train_mode=True)[brain_name]
    while True:
        observation=env_info.vector_observations
        actions = agent.select_action(observation)
        actions = np.clip(actions, -1, 1)
        env_info = env.step(actions)[brain_name]
        next_states = env_info.vector_observations
        rewards = env_info.rewards
        dones = env_info.local_done
        scores += env_info.rewards
        states = next_states
        if np.any(dones):
            scoresToUse.append(np.mean(scores))
            break
    print('Total Average Score After {} episodes: {}'.format(x+1,
np.mean(scores)))

#Output:

Total Average Score After 1 episodes: 0.0
Total Average Score After 2 episodes: 1.2099999729543924
Total Average Score After 3 episodes: 1.2099999729543924
Total Average Score After 4 episodes: 1.8299999590963125
Total Average Score After 5 episodes: 3.539999920874834
Total Average Score After 6 episodes: 3.709999917075038
.....
.....
.....
.....
Total Average Score After 88 episodes: 33.129999259486794
Total Average Score After 89 episodes: 33.129999259486794
Total Average Score After 90 episodes: 34.23999923467636
Total Average Score After 91 episodes: 34.85999922081828
Total Average Score After 92 episodes: 35.219999212771654
Total Average Score After 93 episodes: 35.73999920114875
Total Average Score After 94 episodes: 35.73999920114875
Total Average Score After 95 episodes: 35.89999919757247
Total Average Score After 96 episodes: 36.1699991915375
Total Average Score After 97 episodes: 37.55999916046858
Total Average Score After 98 episodes: 37.739999156445265
Total Average Score After 99 episodes: 38.06999914906919
Total Average Score After 100 episodes: 38.8699991311878
```

10. Plot and preview the average score

```
# plot the scores
fig = plt.figure()
ax = fig.add_subplot(111)
plt.plot(np.arange(len(scoresToUse)), scoresToUse)
plt.ylabel('Average Score')
plt.xlabel('Episode #')
plt.show()
```



11. Close the Unity ML-Agent environment.

```
env.close()
```

Conclusion

In the paper, we examined policy gradient and the DDPG methods and discussed how they could be implemented (in Python/PyTorch) to solve reinforcement learning problems with continuous action space. Our agent was trained using a real-life implementation of the technique and the

agent scored an average score of 38.8 over the one hundred (100) episodes which is more than the benchmark score of thirteen (35).

Other variations of the policy gradient method such as Trust Region Policy Optimization (TRPO) and Proximal Policy Optimation (PPO) could be explored to benchmark the performance of the agent. In addition, Distributed Actor-Critic (DA2C) could be explored to train a version of the Reacher environment with multiple agents.

References

1. Alessandro Palmas, Emanuele Ghelfi et al (2020). "*The Reinforcement Learning Workshop"*. Packt Publication, United Kingdom.
2. Miguel Morales (2020). *"Grokking Deep Reinforcement Learning"*. Manning Publications, Shelter Island, New York, United States.
3. Mnih et al., (2015). *"Human-level control through deep reinforcement learning"*. Google DeepMind, London, United Kingdom.
4. Nimish Sanghi (2021). *"Deep Reinforcement Learning with Python"*. Apress Publications, New York, United States.
5. Shanqing Cai, Stanley Bileschi, Eric Nielsen & Francois Chollet (2020). *"Deep Learning with Javascript"*. Manning Publications, Shelter Island, New York, United States.
6. Sudharsan Ravichandiran (2020). *"Deep Reinforcement Learning with Python"*. Packt Publication, United Kingdom.
7. Udacity (2020). *"Deep Reinforcement Learning Nanodegree Program - Class Resources - Videos & Lectures"*. Mountain View, California, United States of America.
8. Zai & Brown (2020). *"Deep Reinforcement Learning in Action"*. Manning Publications, Shelter Island, New York, United States.