Project Report

Continuous Control

Tri. B. Minh

17th May, 2020

1. Introduction

Solving Problem by using DDPG Algorithms.

The DDPG can get good performance. But the result is very sensitive when tuning the hyperparameters. The common failure mode of the DDPG is the learner to overestimate Q-value then the policy breaking, the scores result get the good at the beginning but at the end, this will reduce dramatically. Because it exploits the errors in the Q-function. I have worked with many experiments to tune the Hyperparameters, but it did NOT reach the best result. All the experiments in chapter 6. Experiment with DDPG

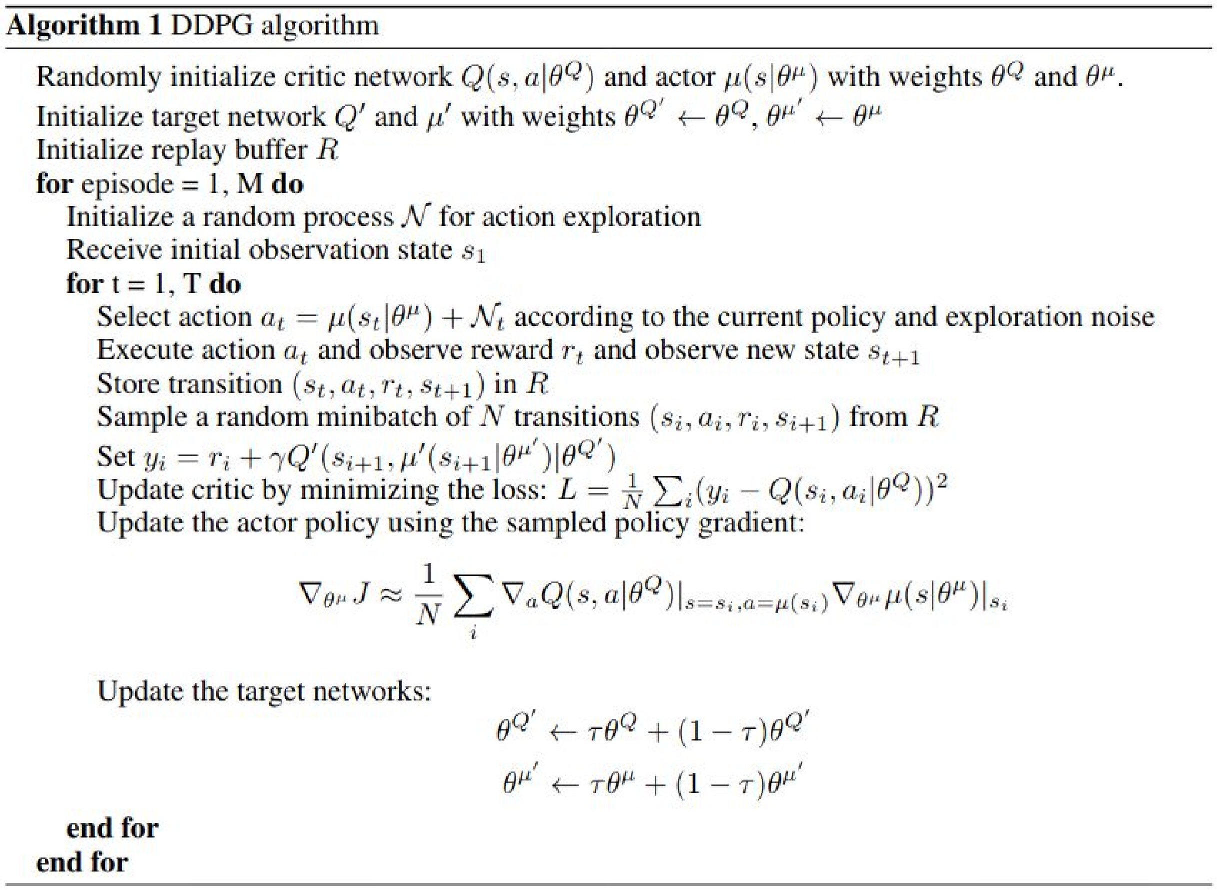
So, I use the **Target Policy Smoothing** in the **Twin Delayed DDPG** (Twin Delayed Deep Deterministic Policy Gradient **TD3)**. That means, I add noise decay to the target action and make the agent hard to exploit the error in the Q-function by smoothing out Q along changes action.

Q-learning decide the action based on the target policies, $\mu_{\theta_{\text{targ}}}$, but with clipped noise add on each dimension of the action. After add the clipped noise the target action, that action a is satisfy $a_{Low} \leq a \leq a_{High}$.

$$a'(s') = \text{clip}\left(\mu_{\theta_{\text{targ}}}(s') + \text{clip}(\epsilon, -c, c), a_{Low}, a_{High}\right), \qquad \epsilon \sim \mathcal{N}(0, \sigma)$$

2. Explain DDPG(Deep Deterministic Policy Gradient) Algorithms

The DDPG contains 2 parts: A **Learning Q function** and **Learning Policy**. In practice, this includes 2 Deep Neural networks, one Critics network and another one is Actor network.



Algorithm 1 DDPG algorithm

```
Randomly initialize critic network $Q(s, a|\theta^Q)$ and actor $\mu(s|\theta^\mu)$ with weights $\theta^Q$ and $\theta^\mu$.
Initialize target network $Q'$ and $\mu'$ with weights $\theta^{Q'} \leftarrow \theta^Q$, $\theta^{\mu'} \leftarrow \theta^\mu$
Initialize replay buffer $R$
for episode = 1, M do
    Initialize a random process $\mathcal{N}$ for action exploration
    Receive initial observation state $s_1$
    for t = 1, T do
        Select action $a_t = \mu(s_t|\theta^\mu) + \mathcal{N}_t$ according to the current policy and exploration noise
        Execute action $a_t$ and observe reward $r_t$ and observe new state $s_{t+1}$
        Store transition $(s_t, a_t, r_t, s_{t+1})$ in $R$
        Sample a random minibatch of $N$ transitions $(s_i, a_i, r_i, s_{i+1})$ from $R$
        Set $y_i = r_i + \gamma Q'(s_{i+1}, \mu'(s_{i+1}|\theta^{\mu'})|\theta^{Q'})$
        Update critic by minimizing the loss: $L = \frac{1}{N}\sum_i (y_i - Q(s_i, a_i|\theta^Q))^2$
        Update the actor policy using the sampled policy gradient:

            $$\nabla_{\theta^\mu} J \approx \frac{1}{N} \sum_i \nabla_a Q(s, a|\theta^Q)|_{s=s_i, a=\mu(s_i)} \nabla_{\theta^\mu} \mu(s|\theta^\mu)|_{s_i}$$

        Update the target networks:
            $$\theta^{Q'} \leftarrow \tau\theta^Q + (1-\tau)\theta^{Q'}$$
            $$\theta^{\mu'} \leftarrow \tau\theta^\mu + (1-\tau)\theta^{\mu'}$$
    end for
end for
```

1. **The Q-Learning Side of the DDPG:**

The Bellman Equation shows the optimal action-value function, $Q^*(s, a)$, in the Python code we call $Q^*(s, a)$ is **Q_target** . It is given by:

$$Q^*(s, a) = \underset{s' \sim P}{\mathrm{E}} \left[r(s, a) + \gamma \max_{a'} Q^*(s', a') \right]$$

Where $s' \sim P$ is the next state in the distribution $P(\cdot|s, a)$

This Bellman Equation is the starting point for the learning approximator to $Q^*(s, a)$. The approximator is a neural network $Q_\phi(s, a)$ with parameters ϕ , with a set $\mathcal{D}$ of transitions (s, a, r, s', d) (state, action, reward, next_state, done), with d indicates whether state s' is terminal, in the Python code we call $Q_{\phi}(s,a)$ is **Q_expected**. We can get the mean-squared Bellman error (MSBE) function, which tell us how closely Q_{ϕ} omes to satisfying the Bellman equation:

$$L(\phi, \mathcal{D}) = \underset{(s,a,r,s',d) \sim \mathcal{D}}{\mathrm{E}} \left[\left(Q_{\phi}(s,a) - \left(r + \gamma (1-d) \max_{a'} Q_{\phi}(s', a') \right) \right)^2 \right]$$

Here, in evaluating $(1-d)$, we've used a Python convention of evaluating True to 1 and False to zero. Thus, when `d==True--`which is to say, when s' is a terminal state—the Q-function should show that the agent gets no additional rewards after the current state. (This choice of notation corresponds to what we later implement in code.)

Q-learning algorithms for function approximators, such as DQN (and all its variants) and DDPG, are largely based on minimizing this MSBE loss function.

We can also call the **Q-Learning Side** is **Critic Network** , the behavior of the Critic Networks is High Variance but no Bias. In the Python code we can implement:

```
# Step 1. Initialize The Critic Network
# Critic Network (w/ Target Network)
        self.critic_local = Critic(state_size, action_size,
random_seed).to(device)
        self.critic_target = Critic(state_size, action_size,
random_seed).to(device)
        self.critic_optimizer = optim.Adam(self.critic_local.parameters(),
lr=lr_critic, weight_decay=weight_decay)

# Step 2. Update the Critic
 # ---------------------------- update critic ---------------------------- #
        # Get predicted next-state actions and Q values from target models
        actions_next = self.actor_target(next_states)
        Q_targets_next = self.critic_target(next_states, actions_next)
        # Compute Q targets for current states (y_i)
        Q_targets = rewards + (gamma * Q_targets_next * (1 - dones))
        # Compute critic loss
        Q_expected = self.critic_local(states, actions)
        critic_loss = F.mse_loss(Q_expected, Q_targets)
        # Minimize the loss
        self.critic_optimizer.zero_grad()
        critic_loss.backward()
        self.critic_optimizer.step()
```

Main tricks of the Critic Network:**Replay Buffers**

Replay Buffers. All standard algorithms for training a deep neural network to approximate $Q^*(s,a)$ make use of an experience replay buffer. This is the set of $\mathcal{D}$of previous experiences. The ***replay buffer should be large enough*** to contain a wide range of experiences, but if you use ***too much experience, the learning will slow down***.

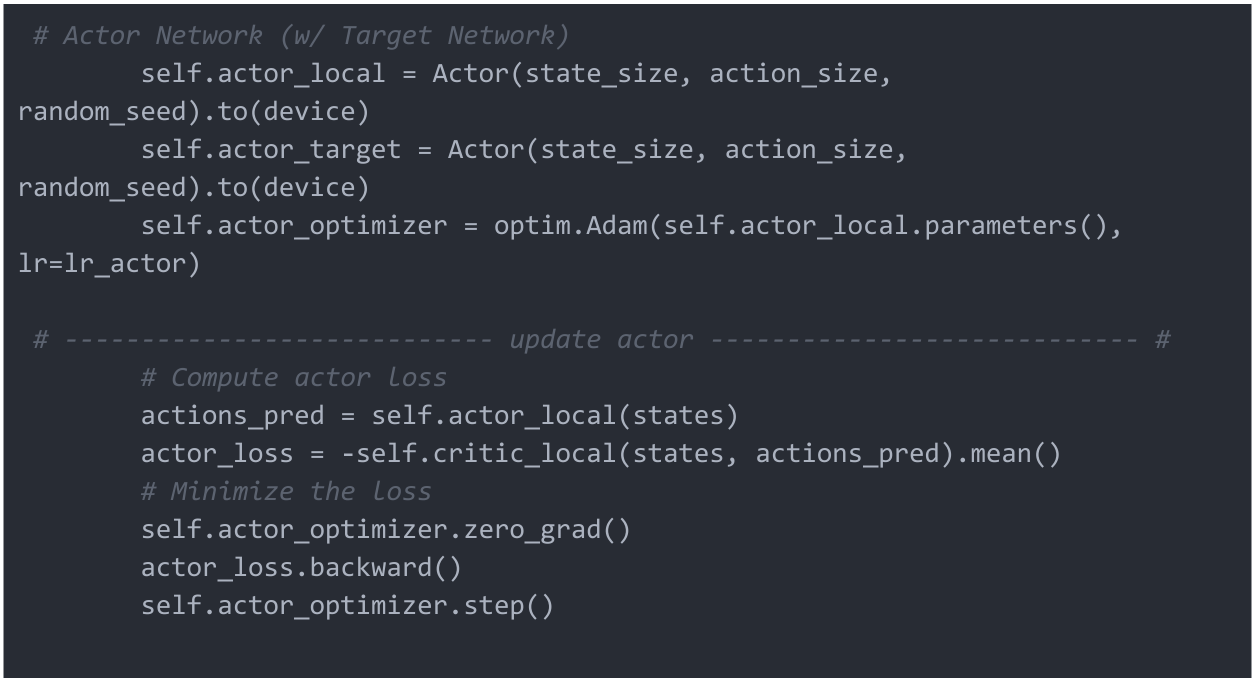
2. The Policy Learning Side of the DDPG

Policy Learning in the DDPG learn a deterministic policy $\mu_\theta(s)$ which gives the action that maximizes $Q_\phi(s,a)$. Because the action space is continuous, and we assume the Q-function is differentiable with respect to action, we can just perform gradient ascent (with respect to policy parameters only) to solve

$$\max_\theta \underset{s\sim\mathcal{D}}{\mathrm{E}}\left[Q_\phi(s,\mu_\theta(s))\right].$$

Note that the Q-function parameters are treated as constants here.

In the Python code we call **Policy Learning** Side is **Actor**.



```
 # Actor Network (w/ Target Network)
        self.actor_local = Actor(state_size, action_size,
random_seed).to(device)
        self.actor_target = Actor(state_size, action_size,
random_seed).to(device)
        self.actor_optimizer = optim.Adam(self.actor_local.parameters(),
lr=lr_actor)

 # ---------------------------- 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()
```

3. Adding noise to improve the policies explore

Improvement DDPG policies add noise on the actions at the training time. In the code we use the **OU noise (Ornstein–Uhlenbeck).** Reduce the **scale of the noise over the course of training to get higher-quality training data.**

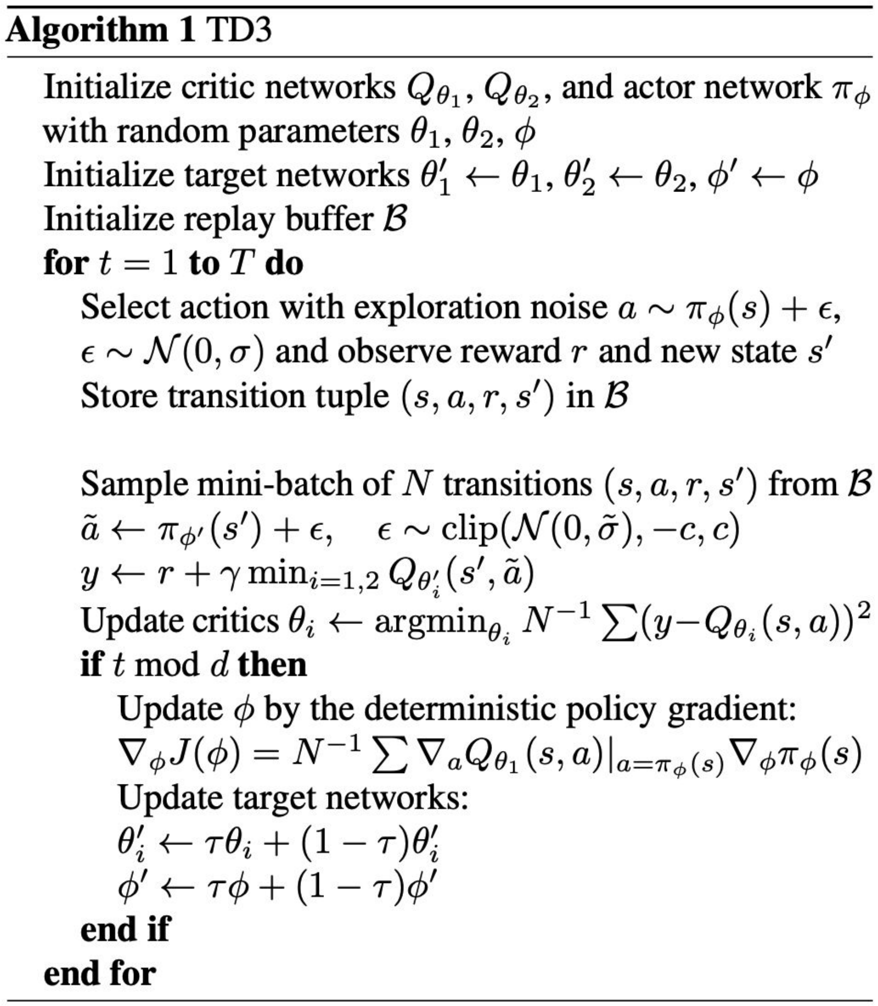
```
# Noise process
        self.noise = OUNoise(action_size, random_seed)
def act(self, state, add_noise=True):
        """Returns actions for given state as per current policy."""
        state = torch.from_numpy(state).float().to(device)
        self.actor_local.eval()
        with torch.no_grad():
            action = self.actor_local(state).cpu().data.numpy()
        self.actor_local.train()
        if add_noise:
            action += self.noise.sample()
        return np.clip(action, -1, 1)
class OUNoise:
    """Ornstein-Uhlenbeck process."""

    def __init__(self, size, seed, mu=0., theta=0.15, sigma=0.1):
        """Initialize parameters and noise process."""
        self.mu = mu * np.ones(size)
        self.theta = theta
        self.sigma = sigma
        self.seed = random.seed(seed)
        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))])
        self.state = x + dx
        return self.state
```

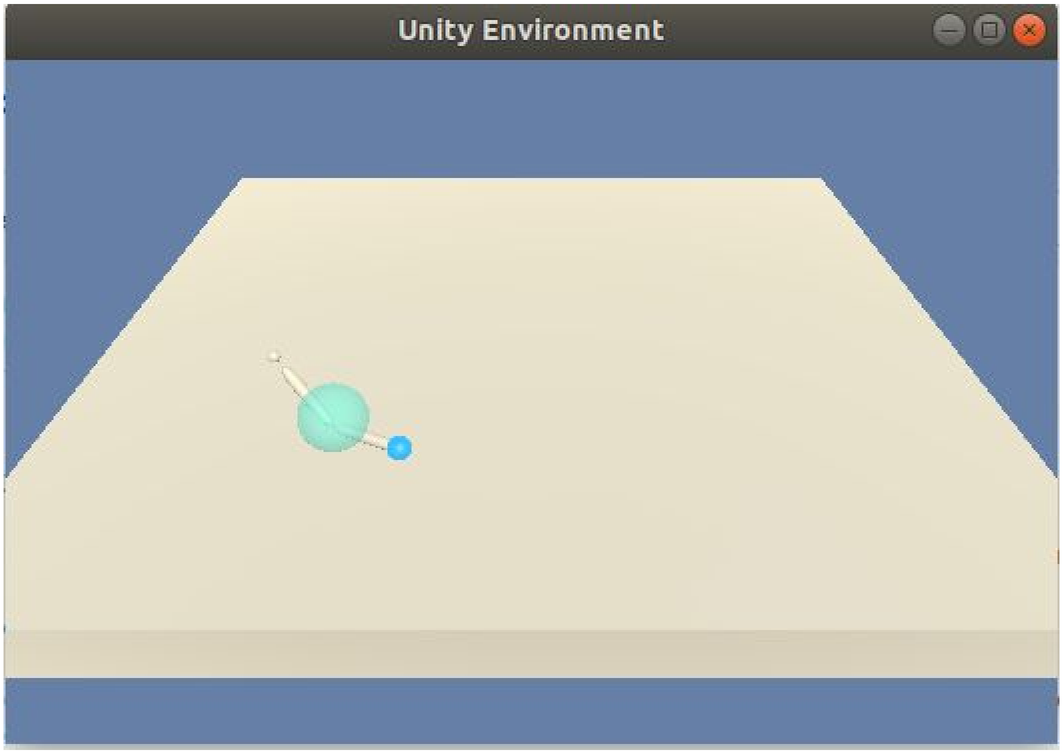
3. TD3 (Twin Delayed Deep Deterministic policy gradient algorithm).



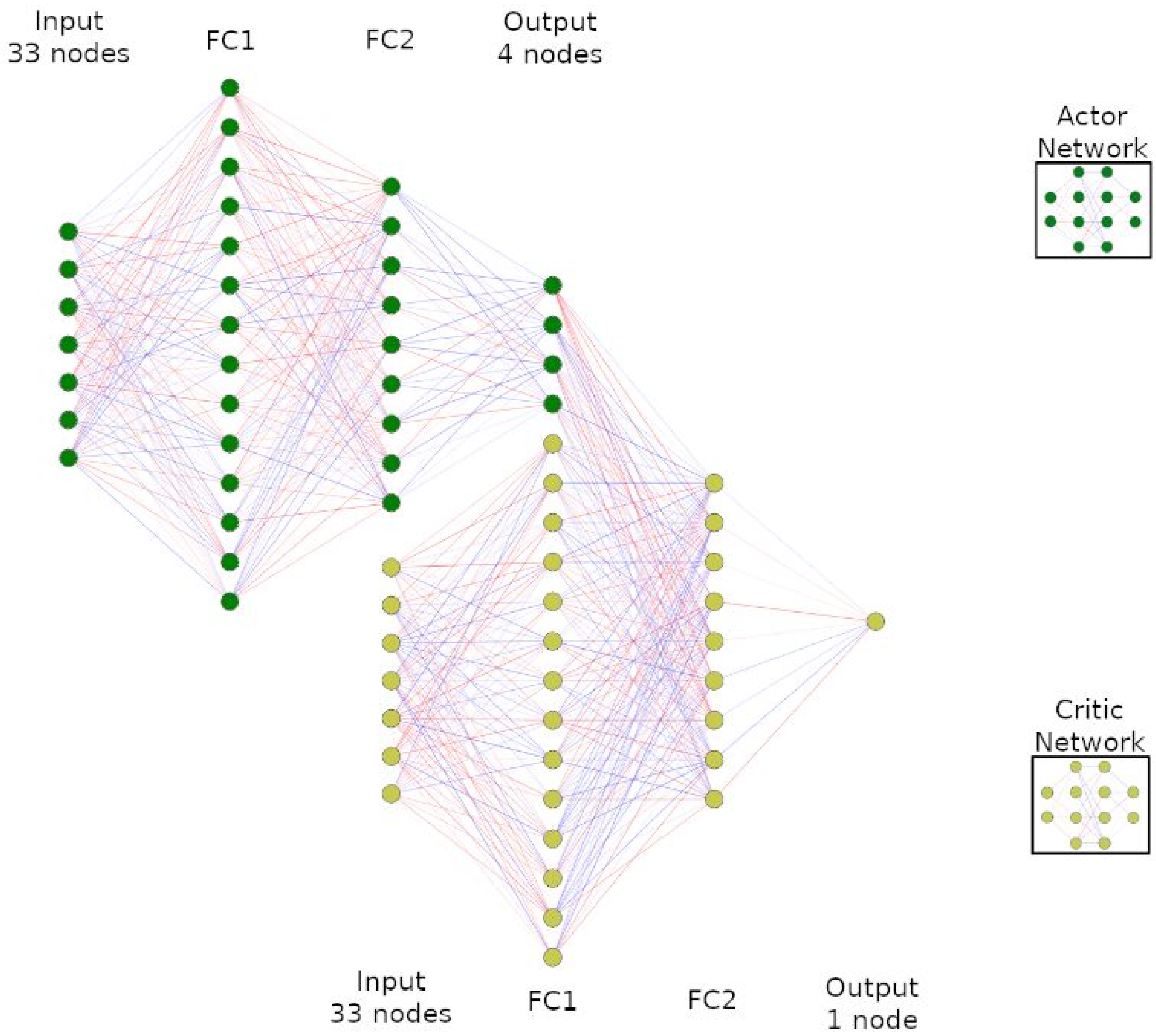
```
Algorithm 1 TD3
  Initialize critic networks $Q_{\theta_1}$, $Q_{\theta_2}$, and actor network $\pi_\phi$
  with random parameters $\theta_1, \theta_2, \phi$
  Initialize target networks $\theta'_1 \leftarrow \theta_1, \theta'_2 \leftarrow \theta_2, \phi' \leftarrow \phi$
  Initialize replay buffer $\mathcal{B}$
  for $t = 1$ to $T$ do
    Select action with exploration noise $a \sim \pi_\phi(s) + \epsilon$,
    $\epsilon \sim \mathcal{N}(0, \sigma)$ and observe reward $r$ and new state $s'$
    Store transition tuple $(s, a, r, s')$ in $\mathcal{B}$

    Sample mini-batch of $N$ transitions $(s, a, r, s')$ from $\mathcal{B}$
    $\tilde{a} \leftarrow \pi_{\phi'}(s') + \epsilon, \quad \epsilon \sim \text{clip}(\mathcal{N}(0, \tilde{\sigma}), -c, c)$
    $y \leftarrow r + \gamma \min_{i=1,2} Q_{\theta'_i}(s', \tilde{a})$
    Update critics $\theta_i \leftarrow \text{argmin}_{\theta_i} N^{-1} \sum (y - Q_{\theta_i}(s, a))^2$
    if $t$ mod $d$ then
      Update $\phi$ by the deterministic policy gradient:
      $\nabla_\phi J(\phi) = N^{-1} \sum \nabla_a Q_{\theta_1}(s, a)|_{a=\pi_\phi(s)} \nabla_\phi \pi_\phi(s)$
      Update target networks:
      $\theta'_i \leftarrow \tau \theta_i + (1 - \tau) \theta'_i$
      $\phi' \leftarrow \tau \phi + (1 - \tau) \phi'$
    end if
  end for
```

4. Solving the the One Arm Reacher



DDPG uses two network architectures, one for actor and one for critic. The actor network maps state to action and has the following structure:



1. State input (33 units)
2. Hidden layer (400 units) with ReLU activation
3. Hidden layer (300units) with ReLU activation
4. Action output (4 units) with tanh activation

The critic network maps state and action to Q value and has the following structure:

1. State input (33 units)
2. Hidden layer 1 (400 nodes) with ReLU activation + Action input (4 units)
3. Hidden layer 2 (300 nodes) with ReLU activation
4. Q-value output (1 node)

Discussion:

Some hyperparameters most effective to the DDPG (Deep Deterministic Policy Gradient) Algorithms are:

1. *Gamma (discount rate) if the discount rate =0.99 the agent does **NOT** get a good result and most depending all the future reward, so the score increases slowly, and not solving the environment. With all experiments below we can choose the **GAMMA =0.98** to get the best result.*
2. *The replay buffer_size will affect the experiments of the agent, when increasing the buffer_size =1e6 (experiment 3.1), the agent learns more faster than using the buffer_size =1e5 (experiment 3.4). The best choose of **BUFFER_SIZE = 1e6***
3. *The best option of the Learning Rate, **LR_ACTOR =4e-4** and **LR_CRITIC=4e-4** when increasing the learning rate to 5e-4 (Experiment 3.3) the Agent can solve problems but more episodes.*
4. *Max time step also affects timing to training the agent, max_t =5000 (Experiment 3.2) with 710 episodes has more episodes to solve the problem compared with the best option **max_t= 1000** (Experiment 3.1) with 194 episodes.*

5. Add the Noise decay in the OU noise. That means, OU noise will reduce over time. It has a good effect on the training model **NOISE_DECAY =0.999**

5. Solving experiments with Target Policy Smoothing in the Twin Delayed DDPG

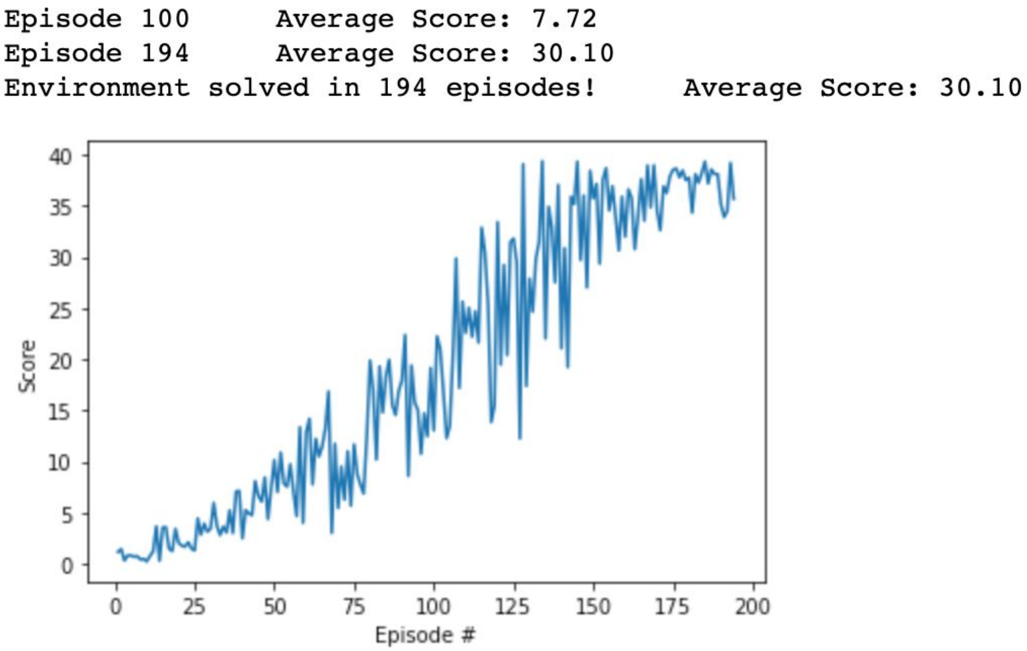
Add the Noise decay in the OU noise. That means, OU noise will reduce over time. It has a good effect on the training model.

```
if add_noise:
          action += self.noise_decay*self.noise.sample()
          self.noise_decay *= self.noise_decay # Decay the noise process along the time
```

5.1 The best result is solved Experiment with 194 episodes with parameters bellow:

BUFFER_SiZE	BATCH_SIZE	GAMMA	TAU	LR_ACTOR	LR_CRITIC
int(1e6)	128	0.98	1e-3	4e-4	4e-4

WEIGHT_DECAY	n_episode	max_t	random_seed	noise_decay
0	3000	1000	0	0.999



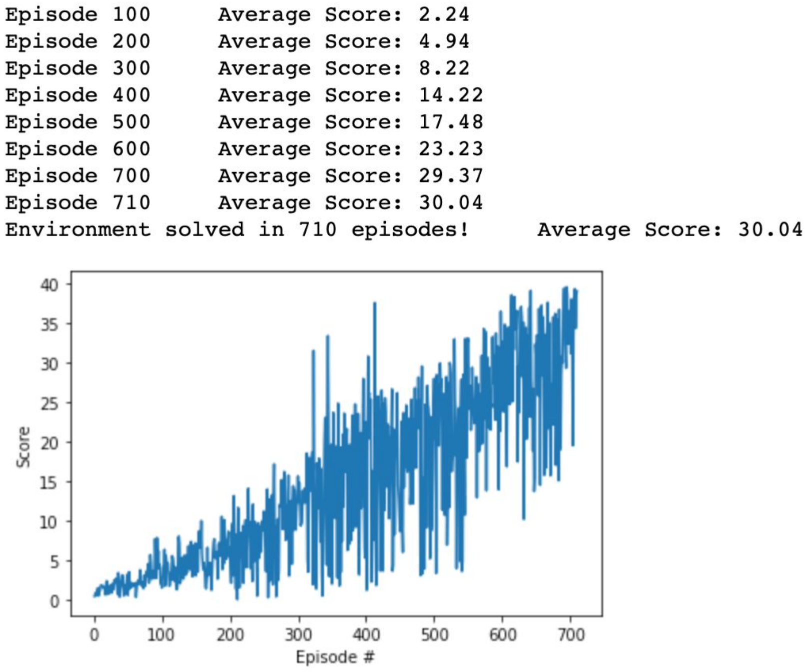
```
Episode 100     Average Score: 7.72
Episode 194     Average Score: 30.10
Environment solved in 194 episodes!     Average Score: 30.10
```

5.2 The another result is solved Experiment with 710 episodes with parameters bellow:

Before getting the best result in Experiment 3.1. I have turned some parameters, same with experiment 3.1, just increasing the max_t to 5000. The training agent can solve Environment in 710 episodes.

BUFFER_SiZE	BATCH_SIZE	GAMMA	TAU	LR_ACTOR	LR_CRITIC
int(1e6)	128	0.98	1e-3	4e-4	4e-4

WEIGHT_DECAY	n_episode	max_t	random_seed	noise_decay
0	3000	5000	0	0.999



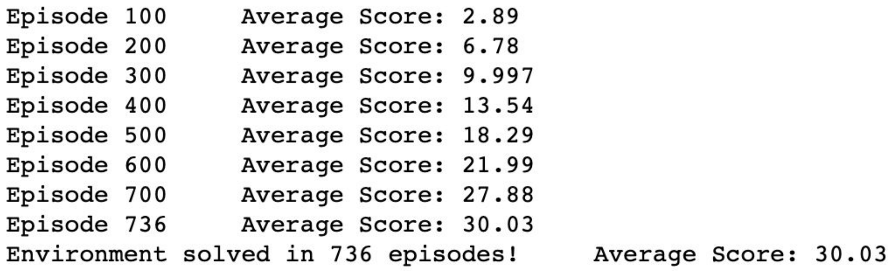
```
Episode 100     Average Score: 2.24
Episode 200     Average Score: 4.94
Episode 300     Average Score: 8.22
Episode 400     Average Score: 14.22
Episode 500     Average Score: 17.48
Episode 600     Average Score: 23.23
Episode 700     Average Score: 29.37
Episode 710     Average Score: 30.04
Environment solved in 710 episodes!     Average Score: 30.04
```

5.3 The another result is solved Experiment with 736 episodes with parameters bellow:

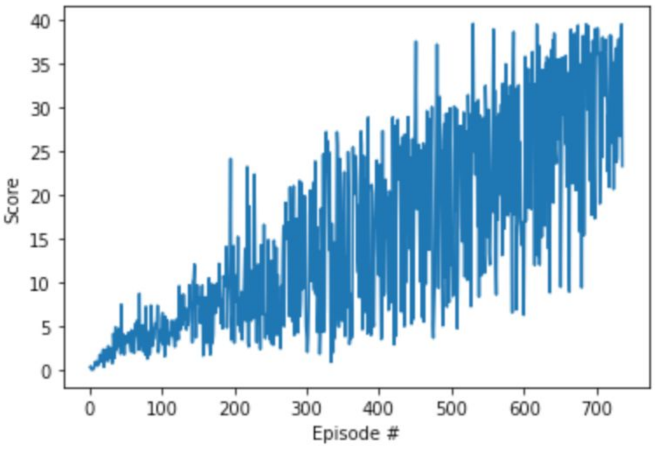
Before getting the best result in Experiment 3.1. I have turned some parameters, same with experiment 3.1, just increasing the learning rate LR_Actor =5e-4 and LR_Critic =5e-4. The training agent can solve Environment in 736 episodes.

BUFFER_SiZE	BATCH_SIZE	GAMMA	TAU	LR_ACTOR	LR_CRITIC
int(1e6)	128	0.98	1e-3	5e-4	5e-4

WEIGHT_DECAY	n_episode	max_t	random_seed	noise_decay
0	3000	1000	0	0.999



```
Episode 100     Average Score: 2.89
Episode 200     Average Score: 6.78
Episode 300     Average Score: 9.997
Episode 400     Average Score: 13.54
Episode 500     Average Score: 18.29
Episode 600     Average Score: 21.99
Episode 700     Average Score: 27.88
Episode 736     Average Score: 30.03
Environment solved in 736 episodes!     Average Score: 30.03
```

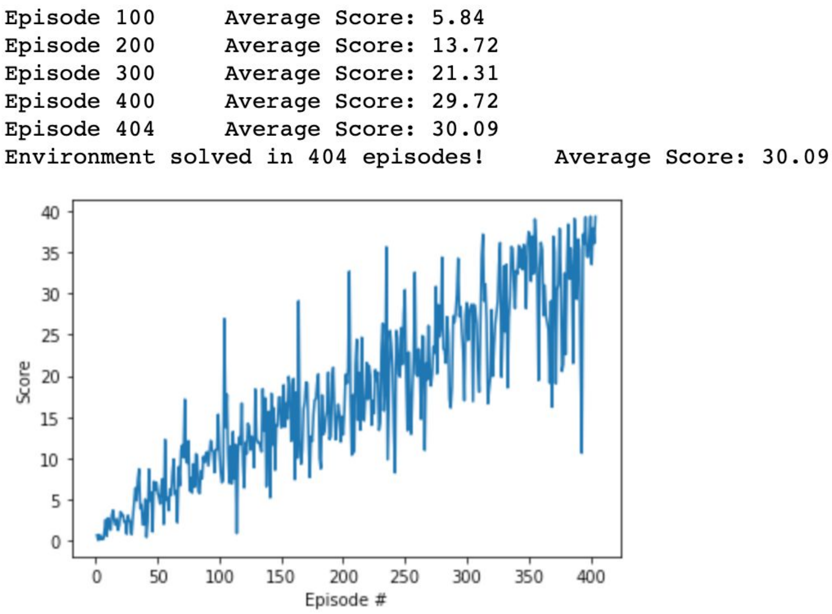


5.4 The another result is solved Experiment with 404 episodes with parameters bellow:

Before getting the best result in Experiment 3.1. I have turned some parameters, same with experiment 3.1, just decrease BUFFER_SIZE = 1e5. The training agent can solve Environment in 404 episodes.

BUFFER_SiZE	BATCH_SIZE	GAMMA	TAU	LR_ACTOR	LR_CRITIC
int(1e5)	128	0.98	1e-3	4e-4	4e-4

WEIGHT_DECAY	n_episode	max_t	random_seed	noise_decay
0	3000	1000	0	0.999



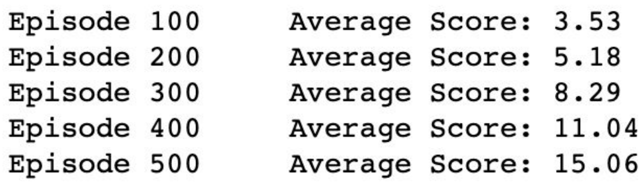
```
Episode 100     Average Score: 5.84
Episode 200     Average Score: 13.72
Episode 300     Average Score: 21.31
Episode 400     Average Score: 29.72
Episode 404     Average Score: 30.09
Environment solved in 404 episodes!     Average Score: 30.09
```

5.5 The another result is NOT solved Experiment with parameters bellow:

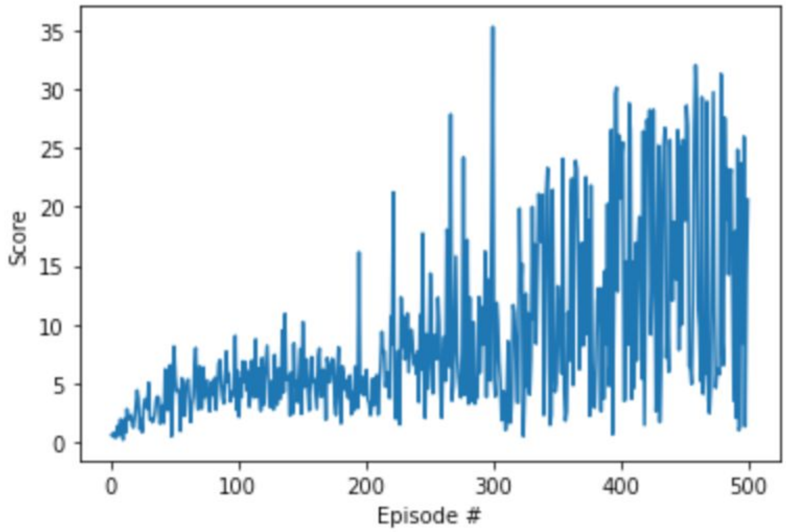
When **GAMMA** =**0.98** the environment is not solved.

BUFFER_SiZE	**BATCH_SIZE**	**GAMMA**	**TAU**	**LR_ACTOR**	**LR_CRITIC**
int(1e5)	128	0.98	1e-3	4e-4	4e-4

WEIGHT_DECAY	**n_episode**	**max_t**	**random_seed**	**noise_decay**
0	3000	1000	0	0.999



```
Episode 100     Average Score: 3.53
Episode 200     Average Score: 5.18
Episode 300     Average Score: 8.29
Episode 400     Average Score: 11.04
Episode 500     Average Score: 15.06
```



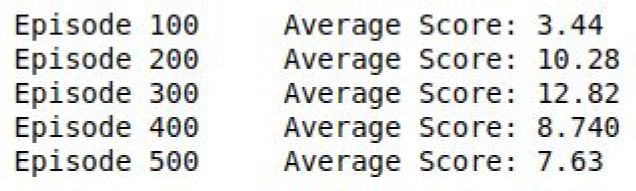
6. Experiment with DDPG

6.1 First Experiment

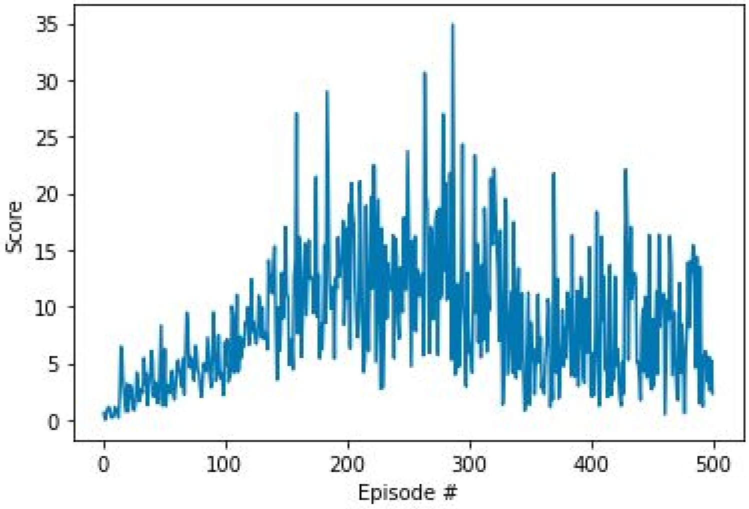
Parameters:

BUFFER_SiZE	**BATCH_SIZE**	**GAMMA**	**TAU**	**LR_ACTOR**	**LR_CRITIC**
int(1e5)	128	0.99	1e-3	3e-4	3e-4

WEIGHT_DECAY	**n_episode**	**max_t**	**random_seed**
0	500	1000	0



```
Episode 100     Average Score: 3.44
Episode 200     Average Score: 10.28
Episode 300     Average Score: 12.82
Episode 400     Average Score: 8.740
Episode 500     Average Score: 7.63
```



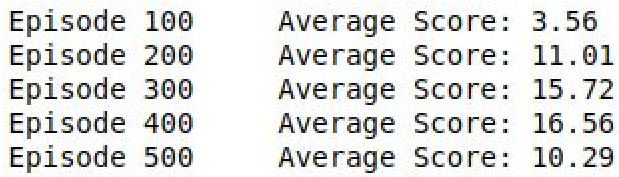
The maximum score is **12.82** at episode 300

6.2 Second Experiment

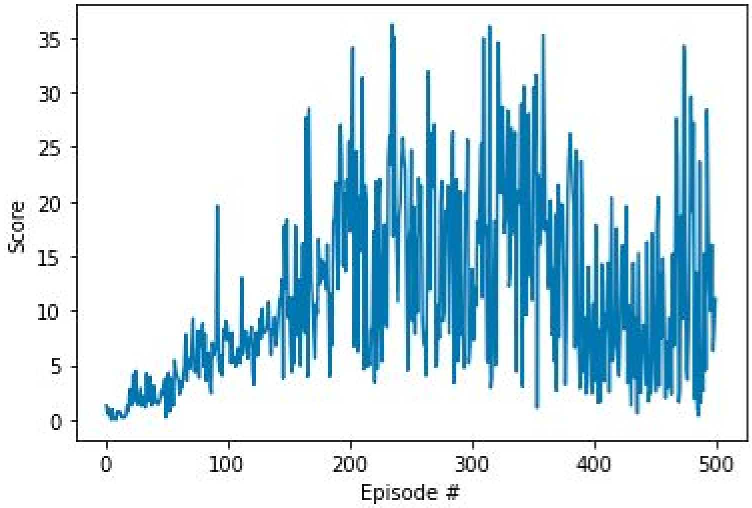
In this experiment we increase the max_t to 5000.

BUFFER_SiZE	**BATCH_SIZE**	**GAMMA**	**TAU**	**LR_ACTOR**	**LR_CRITIC**
int(1e5)	128	0.99	1e-3	3e-4	3e-4

WEIGHT_DECAY	**n_episode**	**max_t**	**random_seed**
0	500	5000	0



```
Episode 100	Average Score: 3.56
Episode 200	Average Score: 11.01
Episode 300	Average Score: 15.72
Episode 400	Average Score: 16.56
Episode 500	Average Score: 10.29
```

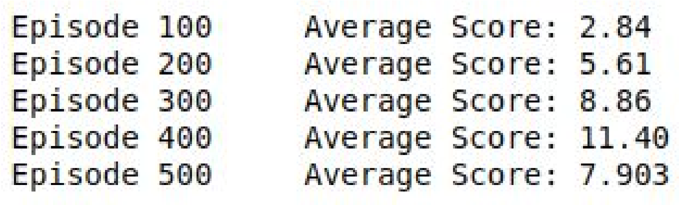


The maximum score is **16.56** at episode 400

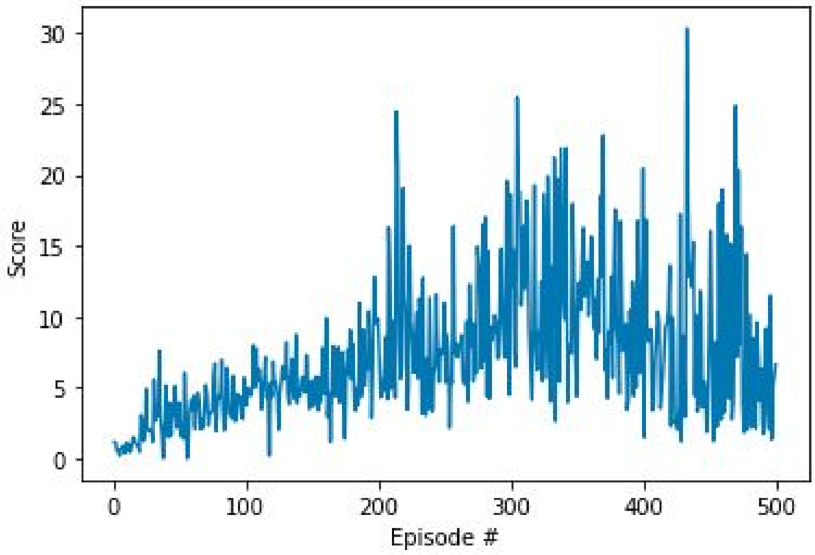
6.3 Third Experiment decrease Learning Rate

BUFFER_SiZE	BATCH_SIZE	GAMMA	TAU	LR_ACTOR	LR_CRITIC
int(1e5)	128	0.99	1e-3	2e-4	2e-4

WEIGHT_DECAY	n_episode	max_t	random_seed
0	500	5000	0



```
Episode 100     Average Score: 2.84
Episode 200     Average Score: 5.61
Episode 300     Average Score: 8.86
Episode 400     Average Score: 11.40
Episode 500     Average Score: 7.903
```

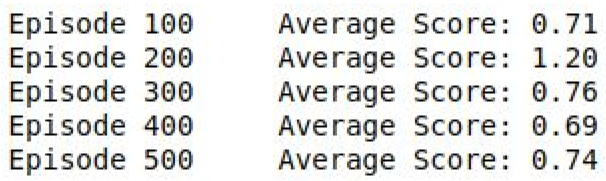


The maximum score is **11.04** at episode 400

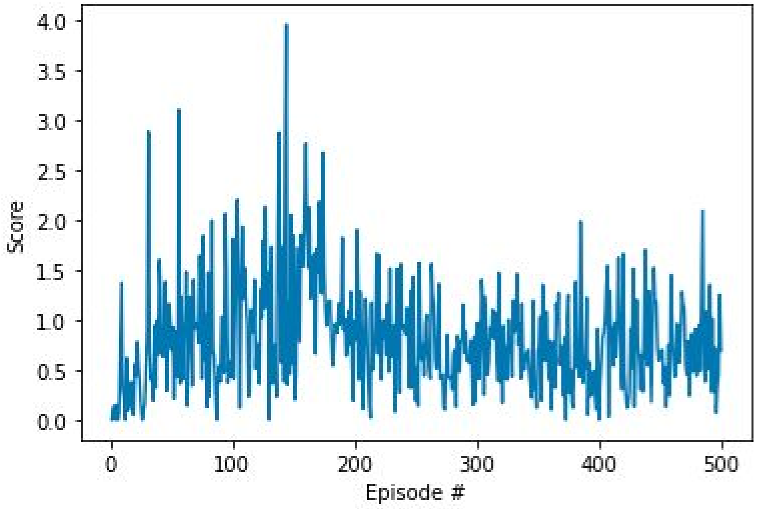
6.4 4nd try with 500 episode and time step 5000 & Decrease LR_Actor and Increase LR_Critic

BUFFER_SiZE	**BATCH_SIZE**	**GAMMA**	**TAU**	**LR_ACTOR**	**LR_CRITIC**
int(1e5)	128	0.99	1e-3	1e-4	1e-3

WEIGHT_DECAY	**n_episode**	**max_t**	**random_seed**
0	500	5000	0



```
Episode 100	Average Score: 0.71
Episode 200	Average Score: 1.20
Episode 300	Average Score: 0.76
Episode 400	Average Score: 0.69
Episode 500	Average Score: 0.74
```



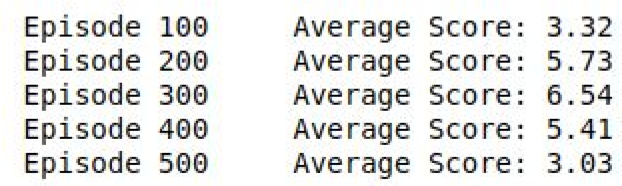
The maximum score is **1.2** at episode 200

6.5 5nd try with 500 episode and time step 1000 & Increase LR

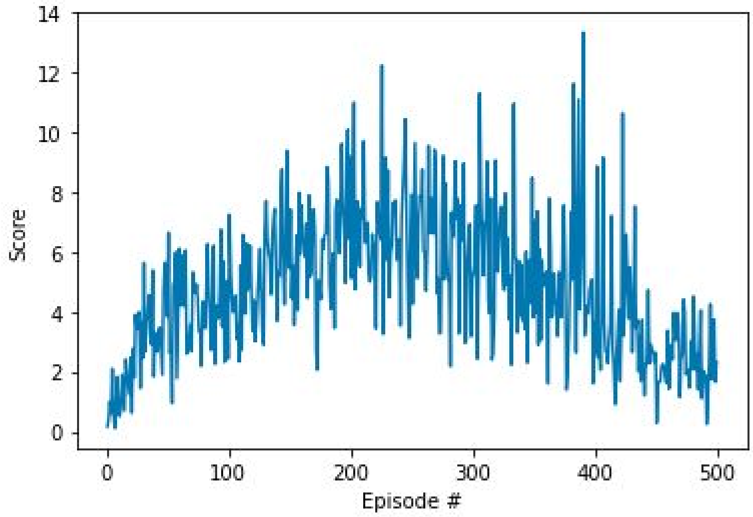
reduce max_t =1000 and Lr_actor =4e-4, lr_critic=4e-4

BUFFER_SiZE	**BATCH_SIZE**	**GAMMA**	**TAU**	**LR_ACTOR**	**LR_CRITIC**
int(1e5)	128	0.99	1e-3	4e-4	4e-4

WEIGHT_DECAY	**n_episode**	**max_t**	**random_seed**
0	500	1000	0



```
Episode 100     Average Score: 3.32
Episode 200     Average Score: 5.73
Episode 300     Average Score: 6.54
Episode 400     Average Score: 5.41
Episode 500     Average Score: 3.03
```

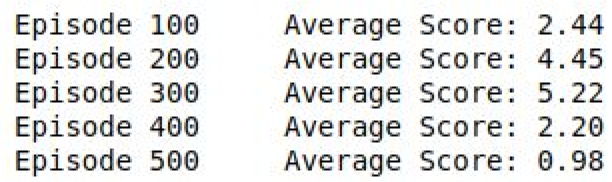


The maximum score is **6.54** at episode 300

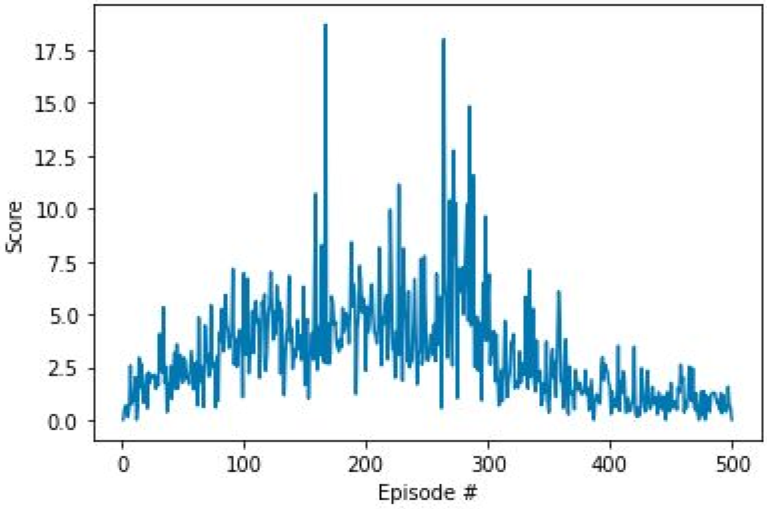
6.6 6nd try with 500 episode and increase time step to 5000 and Lr_actor =3e-4, lr_critic=3e-4, increase batch size =1024

BUFFER_SiZE	BATCH_SIZE	GAMMA	TAU	LR_ACTOR	LR_CRITIC
int(1e5)	1024	0.99	1e-3	3e-4	3e-4

WEIGHT_DECAY	n_episode	max_t	random_seed
0	500	5000	0



```
Episode 100     Average Score: 2.44
Episode 200     Average Score: 4.45
Episode 300     Average Score: 5.22
Episode 400     Average Score: 2.20
Episode 500     Average Score: 0.98
```



7. Future ideas for improving the agent's performance

- Can improve the agent model to get higher scores when twisting hyperparameter, some important value are: gamma, learning rate of the actor and critic, replay buffer size, and noise process

- Can try another noise process to verify their performance.

- Future works may implement them to verify their performance. Some of those algorithms are:

- TRPO - Trust Region Policy Optimization
- GAE - Generalized Advantage Estimation
- A3C - Asynchronous Advantage Actor-Critic
- ACER - Actor Critic with Experience Replay
- PPO - Proximal Policy Optimization
- D4PG - Distributed Distributional Deterministic Policy Gradients

References:

1. CONTROL WITH DEEP REINFORCEMENT LEARNING
2. Reproducibility of Benchmarked DRL Tasks for Continuous Control
3. Parameter Noise
4. Open AI DDPG Algorithms
5. Open AI Twin Delay DDPG
6. Addressing Function Approximation Error in Actor-Critic Methods