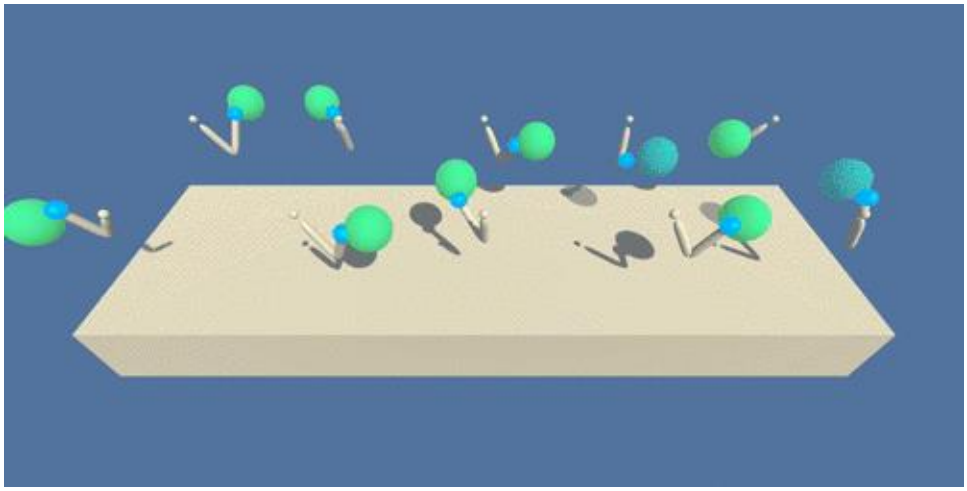
Deep Reinforcement Learning Nanodegree

Project 2 - Continuous Control

Vishal Anand S

1. Objective

For this project, we will work with the **Reacher** environment.



In this environment, a double-jointed arm can move to target locations. 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 position, rotation, velocity, and angular velocities of the arm. Each action is a vector with four numbers, corresponding to torque applicable to two joints. Every entry in the action vector should be a number between -1 and 1.

2. Concepts

The implementation contains various techniques to solve the environment which includes the use of **Experience Replay** and **DQN** from the last project. We will discuss the new and existing ones in this section.

2.1 Policy Based Methods

The value-based methods work towards finding the best Q value for the current state and thus optimizing policy using TD estimate which is an extension of Monte-Carlo method of estimating the expected return. The goal is to increase the expected return so that the agent would take better actions and reach the terminal state more quickly and efficiently.

DQN uses function nonlinear function approximator such as a neural network to approximate the expected return. The expected return is used to calculate the loss using TD equation:

$$\mathbf{Q(s) = R + \gamma \ * max(Q(s', a')) ,}$$

Two main reasons why policy-based methods are considered are: **1.** It directly optimizes the policy without the estimation of Q. **2.** It works well with continuous action spaces where the value-based methods fail since they can't tabulate every instance of the action.

This was considered as a significant improvement over value-based methods since it removes the value of state and used the probabilities and rewards to compute the loss function, thus improving the policy directly from the estimates. There are many policy-based methods such as hill climbing, cross entropy, evolution strategies, policy gradient (REINFORCE) etc.

2.2 REINFORCE

Reinforce is policy gradient method which uses gradient ascent to maximize the loss of the policy network. Basically, policy network is a neural network which takes the states as its input and return log probabilities of the actions. The loss function is as follows,

$$\mathbf{L(s) = \Sigma \log(P(a \mid s)) * R}$$

$$\Theta = \Theta + \alpha * L(s) ,$$

Where R is the cumulative discounted reward which is multiplied with the log probabilities of the action in the given state. The weights are updated according to the loss. The plus sign (+) in update is present since we are performing gradient ascent.

REINFORCE is a good method to start off since it provides the basics of policy gradient. This project uses a more advanced technique called Actor-Critic method which uses both value-based and policy-based to solve the problem.

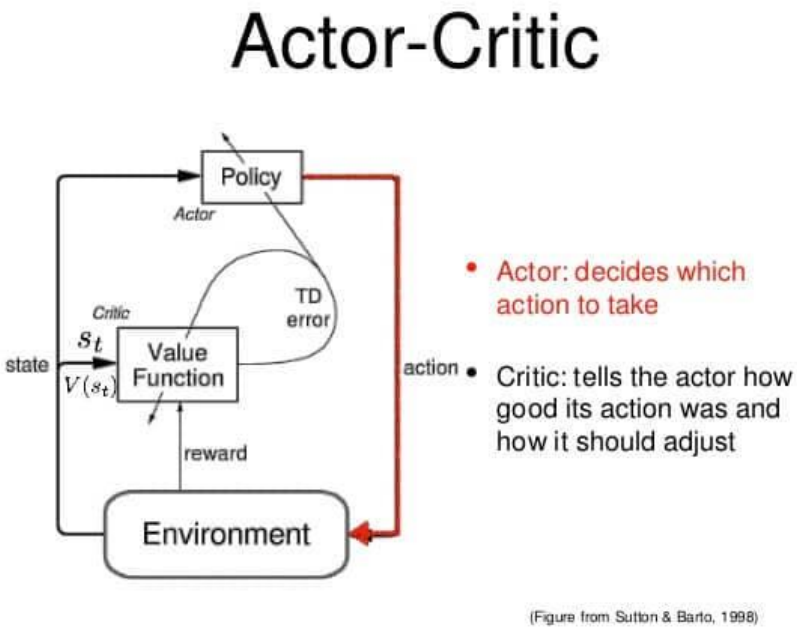
2.3 Actor Critic Method

Value based methods such as DQN produces the estimates of the future reward being in the current state, which means they are good at predicting the future. Monte Carlo methods do the same, but it uses the actual value of the future states to estimate the state value which in low bias but high variance. Temporal difference calculates the estimate of furure reward using the value of next state, thus resulting in low variance but high bias. Since DQN is a more evolved TD was of calculating the estimates, it has low variance.

Policy based methods has high variance since it directly estimates the policy. After every episode the probability of taking a particular action on a specific state increases or decreases based on the final reward. In some cases, the intermediate actions can be considered as good actions, but are given less probability due to the final reward. This can be reduced by experiencing more episodes but is less efficient due to the computational time. This is where the Actor-Critic method comes.

The actor critic method uses policy-based methods like PPO, TRPO as the actor and a value-based method such DQN as its critic. In a nutshell, the critic tells how bad or good the action taken by policy network is. The method consists of two neural networks (Actor network and Critic Network). The process is as follows:

- Actor takes an action based on the input state.
- The action is used to update the critic network.
- Critic produces the value of the state.
- The value is used to update the actor network.



This in-turn reduces the variance of the policy network and the amount of training required to solve the problems.

2.4 DDPG

DDPG stands for Deep Deterministic Police Gradient. It uses two separate networks, one for policy (actor) and other for value (critic) for solving environments. It's an off-policy method and works on continuous action spaces. This is considered as DQN for continuous actions spaces since its use of Experience Replay and update strategy. It uses Ornstein-Uhlenbeck Noise which is added to the predicted action for exploration. The pseudocode for the algorithm is as follows:

Algorithm 1 DDPG algorithm

Randomly initialize critic network $Q(s, a|\theta^Q)$ and actor $\mu(s|\theta^\mu)$ with weights θ^Q and θ^μ.
Initialize target network Q' and μ' 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:

$$\theta^{Q'} \leftarrow \tau\theta^Q + (1-\tau)\theta^{Q'}$$
$$\theta^{\mu'} \leftarrow \tau\theta^\mu + (1-\tau)\theta^{\mu'}$$

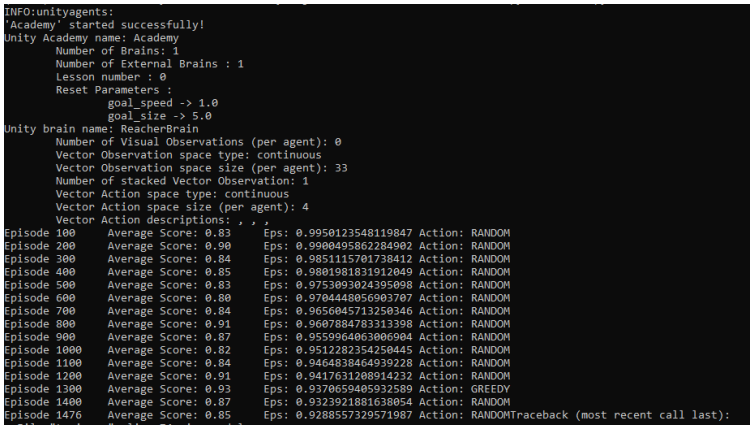
  end for
end for

3. Implementation

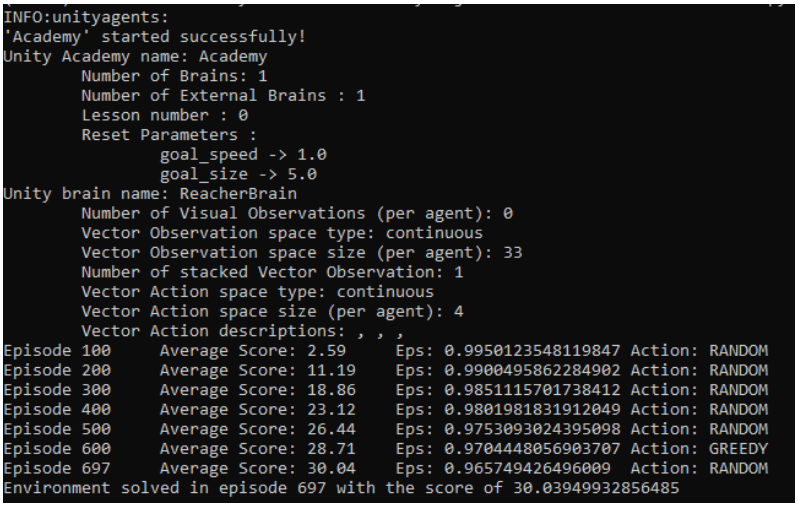
3.1 Experiments

In order to solve the Reacher environment, I used DDPG algorithm with Experience Replay. The learning rates for both actor and critic networks were 10^{-3} and discount rate (γ) of 0.99. Soft updates were done after each training step with TAU 0.001, with this the target network gradually converges with greater stability. Both networks contain two hidden layers with 256 and 128 neurons each with ReLU activations. These were the common settings used for multiple experiments.

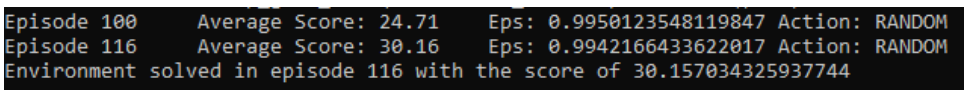
At first, I used the weight decay of 0.0001 for Critic network and a bigger network architecture for both Actor and Critic. The agent ran for more than 1000 episodes, but the average score didn't improve over 1.0. I spent lot of time changing the hyper parameters which didn't result in any change, the result is shown in the below image.



During the second run, I reduced the complexity of the network (mentioned on the first paragraph) and it started to show some improvements and set the weight decay to 0.0 and changed the learning steps. For every step the agent learns for 4 times. This significantly increased the performance but took around 697 steps to converge.

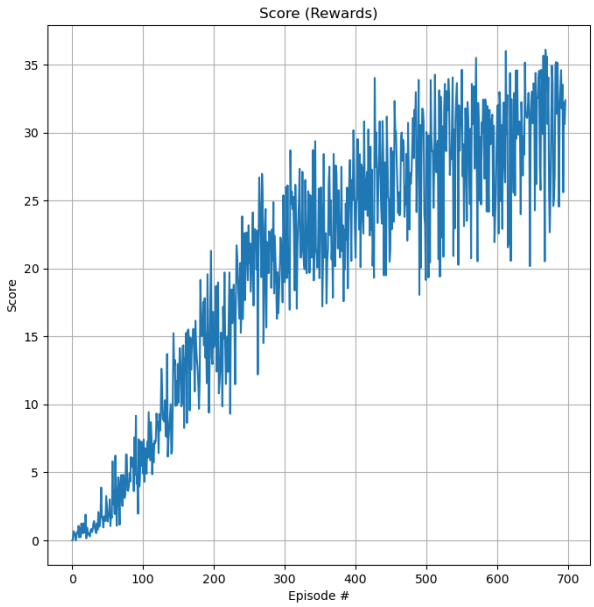


During the third run, I clipped the critic loss as shown in the benchmark implementation and changed the learning steps. For every 10 steps, the model trains 20 times. This made the learning faster by 1/6th of the previous run.

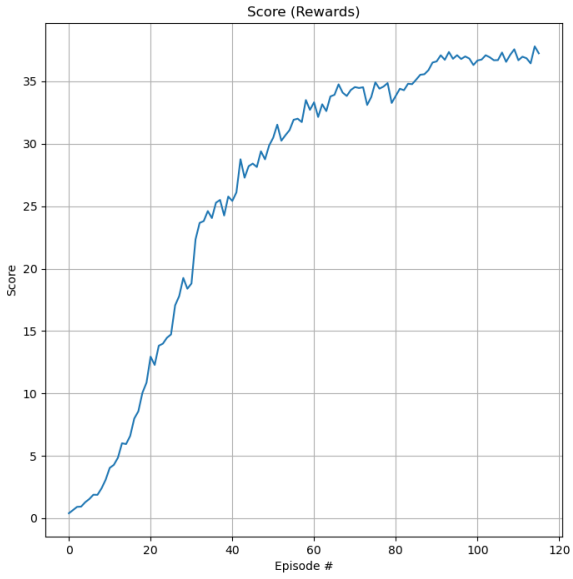


3.2 Results

The score graph for second and third runs are produced below,



As you can see the variance in producing the outputs are high, but this gets stabilized once we clip the critic loss and increase the step size as shown below:



What's Next

I was planning to implement TRPO after I experiment with DDPG, but it turns out that PPO performs better that TRPO since it is an advancement of the latter. One of the issues I must figure out is creating parallel environments with UnityEnvironments as done in REINFORCE notebooks in the nanodegree.