# Deep Reinforcement Learning – Project 2: Continuous Control

NIKHIL TIWARI

The project demonstrates how policy-based methods can be used to learn the optimal policy in a model-free Reinforcement Learning setting using a Unity environment, in which 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 the 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 is a number between -1 and 1.

## Learning Algorithms

### Actor-critic method

We use Deep Deterministic Policy Gradients (DDPG) to solve the problem. which is a different kind of actor-critic method. Firstly, let us give ourselves a quick refresher of the Actor-critic method. One may recall that the agent can learn the policy either directly from the states using Policy-based methods or via the

action valued function as in the case of Value-based methods. The policy-based methods tend to have high variance and low bias and use Monte-Carlo estimates whereas the value-based methods have low variance but high bias as they use TD estimates. Now, the actor-critic methods were introduced to solve the bias-variance problem by combining the two methods. In Actor-Critic, the actor is a neural network which updates the policy and the critic is another neural network which evaluates the policy being learned which is, in turn, used to train the actor. In vanilla policy gradients, the rewards accumulated over the episode are used to compute the average reward and then, calculate the gradient to perform gradient ascent. Now, instead of the reward given by the environment, the actor uses the value provided by the critic to make the new policy update.

**Deep Deterministic Policy Gradients**

Deep Deterministic Policy Gradient (DDPG) lies under the class of Actor-Critic Methods but is a bit different than the vanilla Actor-Critic algorithm. The actor produces a deterministic policy instead of the usual stochastic policy and the critic evaluates the deterministic policy. The critic is updated using the TD-error and the actor is trained using the deterministic policy gradient algorithm. In fact, it could be seen as an approximate Deep-Q Network(DQN) method. The reason for this is that the critic in DDPG is used to approximate the maximizer over the

Q-values of the next state, and not as a learned baseline, as have seen so far. One of the limitations of the DQN agent is that it is not straightforward to use in continuous action spaces. For example, how do you get the value of continuous action with DQN architecture? This is the problem DDPG solves.

**Two sets of Target and Local Networks**

Recall that the concept of fixed targets was first introduced in the Deep Q Networks. We use the same concept in DDPG. In total, we use four deep neural networks –local and target networks for both actor and the critic. Now the actor here is used to approximate the optimal policy deterministically. That means we want to always output the best-believed action for any given state. This is unlike a stochastic policy in which we want to learn a probability distribution over the action. In DDPG we want the best-believed action every time we query the actor-network. That is a deterministic policy. The actor is basically learning the argmax of Q(s,a) which is the best action. The critic learns to evaluate the optimal action-value function by using the actors' best-believed action. Again we use this action which is an approximate maximizer to calculate a new target value for training the action-value function much in the way DQN does.

**Replay Buffer**

In reinforcement learning, the agent interacts with the environment and learns from the sequence of state(S), actions(A) and rewards(R) and next state(S') which is stored in the form of an experienced tuple. These sequences of experience tuples can be highly correlated, and the DDQN algorithm like the Q-learning algorithm that learns from each of these experienced tuples in sequential order can be swayed by the effects of this correlation. This can lead to oscillation or divergence in the action values, which can be prevented by having a replay buffer, from which experience replay tuples are used to randomly sample from the buffer. The replay buffer contains a collection of experience tuples (SS, AA, RR, S'S ′ ). The tuples are gradually added to the buffer as we are interacting with the environment. The act of sampling a small batch of tuples from the replay buffer in order to learn is known as experience replay. In addition to breaking harmful correlations, experience replay allows us to learn more from individual tuples multiple times, recall rare occurrences, and in general make better use of our experience.

**Ornstein-Uhlenbeck(O-U) Noise**

In Reinforcement learning for discrete action spaces, exploration is done via probabilistically selecting a random action such as epsilon-greedy or Boltzmann exploration. For continuous action

spaces, exploration is done via adding noise to the action itself. For details on the O-U noise please refer to the DDPG paper[1].

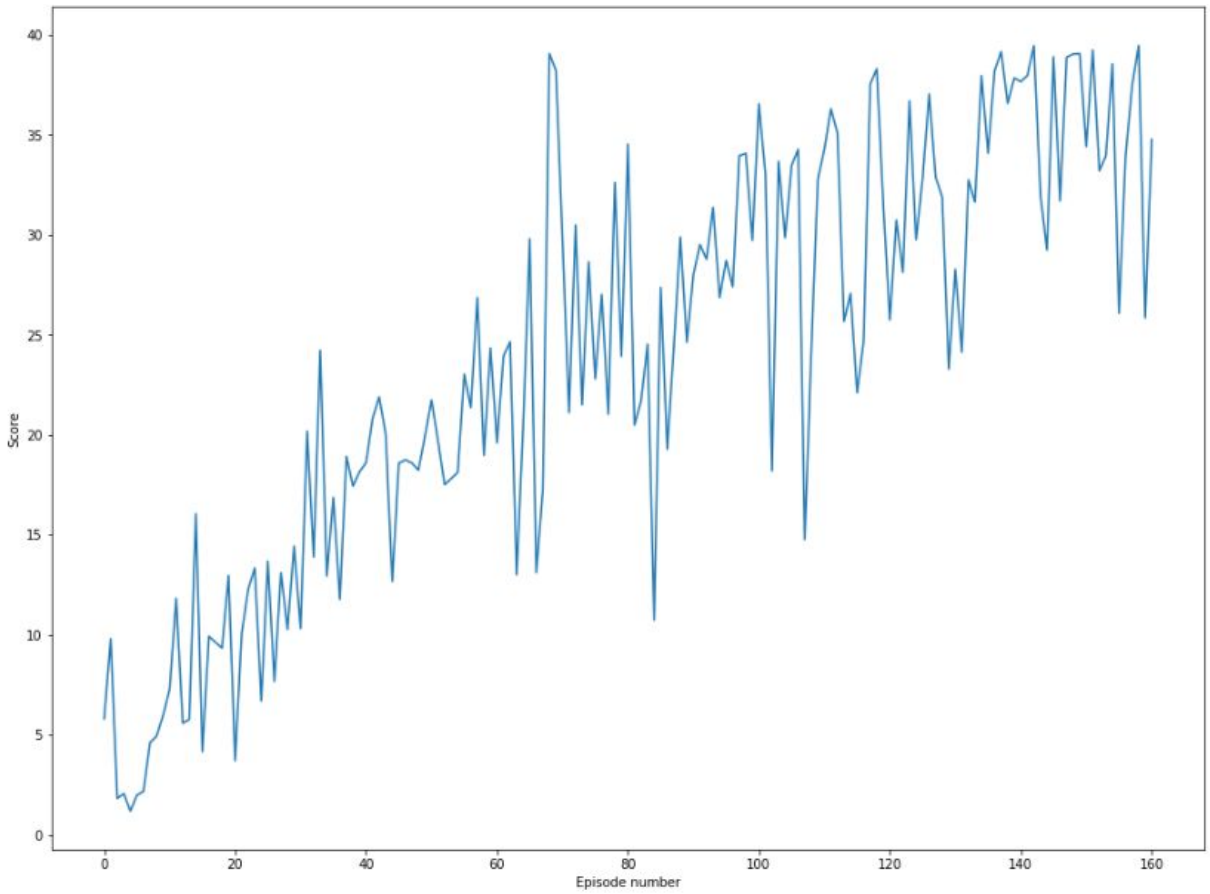
**Soft Updates**

In DDPG, target networks are updated using a soft update strategy. In DQN you have two copies of your network weights: the regular and the target network. In DDPG, you have two copies of your network weights for each network, a regular for the actor, a regular for the critic, and a target for the actor, and a target for the critic. A soft update strategy consists of slowly blending in your regular network weights with your target network weights. So, every timestep you make your target network be 99.99 percent of your target network weights and only 0.01 percent of your regular network weights. You are slowly mixing your regular network weights into your target network weights. Recall, the regular network is the most up to date network because it is the one we are training while the target network is the one we use for prediction to stabilize strain. In practice, you will get faster convergence by using the update strategy, and in fact, this way for updating the target network weights can be used with other algorithms that use target networks including DQN.

## Results

Because we have two models; the actor and critic that must be trained, it means that we have two sets of weights that must be optimized separately. Adam was used for the neural networks with a learning rate of $10^{-3}$ and $10^{-3}$ respectively for the actor and critic. For Q, I used a discount factor of $\gamma = 0.99$. For the soft target updates, the hyperparameter $\tau$ was set to 0.001. The neural networks have 2 hidden layers with 250 and 100 units respectively. For the critic Q, the actions did not include the 1st hidden layer of Q. The final layer weights and biases of both the actor and critic were initialized from a uniform distribution $[-3 \times 10^{-3}, 3 \times 10^{-3}]$ and $[3 \times 10^{-4}; 3 \times 10^{-4}]$ to ensure that the initial outputs for the policy and value estimates were near zero. As for the layers, they were initialized from uniform distribution $[-1/\sqrt{f}, 1/\sqrt{f}]$ where f is the fan-in of the layer. The training used mini batches of 128 and the replay buffer was of size $10^6$.For the exploration noise process, the parameters used for the Ornstein-Uhlenbeck process were $\theta = 0.15$ and $\sigma = 0.2$.

The agent is trained until it solves the environment, that is to say an average reward of at least +30 for the last 100 episodes. Because I have trained the agent for 1000 episodes and found it was not enough, I restarted from these weights.The environment is solved in 1060 episodes.




Plot of scores per episode (from episode 1000)

## Future Ideas:

I have only solved the single-agent problem. Future work should include solving the multi-agent continuous control problem with DDPG. Algorithms such as TRPO[2], PPO[3], A3C[4] might also be useful especially in working with the multi-agent problem particularly when parallelization may be very useful. The Q-prop algorithm, which combines both off-policy and on-policy learning may also be used. For DDPG which uses a

replay buffer, one can also try implementing it with prioritized replay which may yield better results.

**References:**

[1] DDPG Paper: https://arxiv.org/abs/1509.02971
[2] TRPO: https://arxiv.org/abs/1502.05477
[3] PPO: https://arxiv.org/pdf/1707.06347.pdf
[4] A3C: https://arxiv.org/pdf/1707.06347.pdf