Report

Project 2: Continuous Control

Benji Alwis

Introduction

In this project, I use a model-free policy-based reinforcement method to control a double-jointed arm that can move in carrying out a defined task. Hence the set goal for the agent is that it should maintain its position at the target location at the measured time steps during the episodes as much as possible. As the incentive for following the expected behaviour, a reward of +0.1 is given at each time step if the arm is in the target location. The goal is to get an average score of +30 over 100 consecutive episodes.

This learning of optimal policy was done using a Unity ML-agents based environment, in which movements of a double-jointed arm was simulated. We used the project environment provided by Udacity. One version had a single agent while the other version was provided with 20 agents.

The observation space consisted of **33 variables**. They included variables such as position, rotation, velocity, and angular velocities of the arm. Each action was represented using a vector of four values, representing torque applicable to two joints.

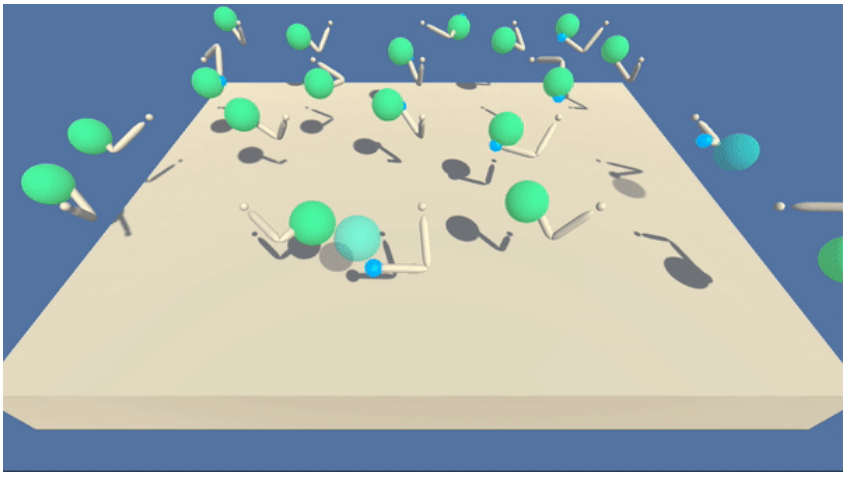

Figure 1 – An example view of a set trained agents.

Method

The value based methods used in the previous project calculate the policy based on state or state-action values. Policy based methods can directly parameterise policy functions. As model free methods, they calculate the optimum policy directly through experience. Some of the advantages of this method include better convergence, effectiveness in high dimensional or continuous action spaces and the ability to learn stochastic policies. However the disadvantages include its convergence to local rather than a global optimum and its inefficiency and high variance. In particular, REINFORCE policy based method which uses a Monte-Carlo estimate suffer due to high variance. On the other hand, TD estimates used in value-based methods give some bias but low

variance. Actor-Critic methods combine policy based methods together with value based methods to reduce variance at the expense of adding some bias. In this approach, a neural network is used to learn the policy function (actor) while another method is used to learn the state action values (critic).

Deep Deterministic Policy Gradient (DDPG) method is an actor-critic off-policy gradient method that concurrently learns Q value and Policy together. It learns the optimum state action value (Q*(s,a)) by learning to optimise the action value (a*(s)). In this framework, the actor is trained to produce a deterministic policy in difference to stochastic policies generated by other actors. The critic's role is to assess this deterministic policy.

This is a method specifically adapted for environments with continuous action spaces. Neural networks are used to learn the Q value as well as the policy function. The following tricks are used to minimise some of the keys issues in this approach.

Experience Replay Buffer-The key objective of having a replay buffer is to minimise the impact of correlated inputs. If inputs from a sequential process is directly fed into a neural network, they may adhere to the assumption that they are iid samples. Instead, they are stored inside of a temporary buffer and some of it is randomly chosen in mini batches to train the network. However, care must be given to the size of the buffer since a too small buffer would not decorrelate data as a result, it would over-fit. On the other hand, a too large buffer would make the process too slow.

Fixed Targets-During the training phase, we aim to minimise the loss function between the Q values and the network output. If we use the same data to update both of them at the same time, it would make training process unstable. To avoid this, the two networks can be updated (target and local) with a lag in a similar manner done in the DQN project. We follow the soft update approach for both actor and the critic. In this approach, 0.01% of the local network weights are mixed with 0.99% target networks weights.

Implementation

My code used in this project was adapted from the "DDPG bipedal" tutorial from the Deep Reinforcement Learning Nanodegree. It twas written using Python 3.6 and PyTorch 0.4.0.

The following are the main components of the implementation.

- `model.py` : This file contains the implementation of the **Actor** and the **Critic** classes. Both of these classes contain target and local networks.

  The following functions perform similar tasks in both classes.

  reset_parameters() - initialise the network weights

  forward(state) – neural network forward propagation with different options for batch normalisation.

  Actor network consists of one input layer, hidden layer and an output layer. The forward function provides options for batch normalisation of the input vector, the output of the input

layer before or after the activation function and the output of the hidden layer. Either relu or leaky relu is used as the activation function.

Critic network consists of an input layer, two hidden layers and an output layer. The best normalisation found with the actor network, normalisation of the input vector together with leaky relu as the activation function were used.

- `ddpg_agent.py` : This file contains the implementation of the DDPG_agent, Replay Buffer and OUNoise.

  Agent Class implements Actor's *Local* and *Target* neural networks, and the Critic's *Local* and *Target* neural networks.

  The main methods of the Agent class are

  - act() - Given a list of states, it returns the actions to be taken by each agent based on the current policy.
  - step() - Given a set of S,A,R,S' experiences, it saves them into the experience buffer, and samples from the experience buffer to perform training steps.
  - Learn() - Updates the policy and value parameters using given batch of experience tuples.
  - soft_update() - Soft update model parameters based on the rule
    $\theta\_target = \tau*\theta\_local + (1 - \tau)*\theta\_target$

    where θ_target - parameters of the target network and
    θ_local - parameters of the local network.

- `Continuous_Control_DDPG.ipynb` : This is the implementation of the actual training process and the experimental procedure. It calls the Unity environment, trains a DDPG agent and plots the experimental results.

Hyper Parameters

Replay buffer size (BUFFER_SIZE) = 1e6
Mini-batch size (BATCH_SIZE) = 1024
Discount factor (GAMMA) = 0.99
Soft update of target parameter (TAU) = 1e-3
Learning rate of the actor (LR_ACTOR) = 1e-4
Learning rate of the critic (LR_CRITIC) = 3e-4
L2 weight decay (WEIGHT_DECAY) = 0.0001
Update interval = 20
Number of episodes = 500
Maximum number of timesteps per episode = 1000
Leak for LeakyReLU =0.01

Results

The experimental setup contained 20 agents making decisions in parallel. I tried different hyper-parameters and also different batch normalisation methods for the actor network. I observed that normalisation of only the input vector and the use of leaky relu as the activation function delivers the best results. The following graph shows how the average score per episode increased against training episodes.

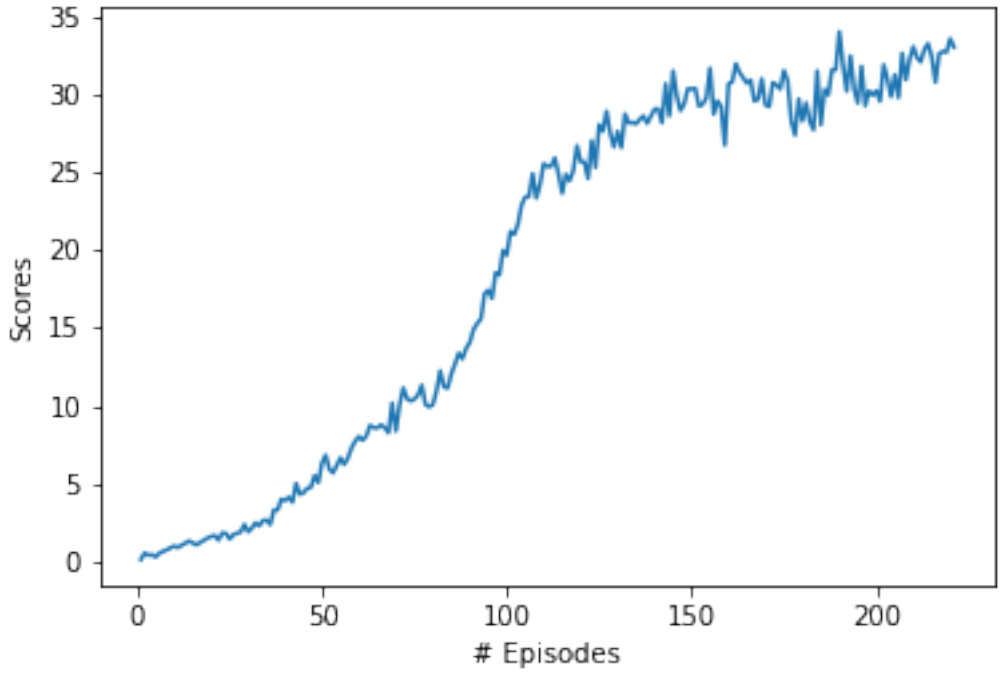

Figure – Averages score against the episodes.

Future Work

We have used a replay buffer in the setup. However, it has been shown that prioritised replay improves the results. We have used the DDPG algorithm. However, the other algorithms discussed in the course such as Asynchronous Advantage Actor-Critic (A3C) and Advantage Actor-Critic (A2C), Generalised Advantage Estimation (GAE) and Q-Prop algorithm which combines online and offline learning will be good ones to try out as well. Distributed Distributional DDPG (D4PG) which has shown improvements over the DDPG algorithm will also be another algorithm to investigate in the future.