

## Contents

DDPG Model implementation.....	2
Description of DDGP model .....	2
Agent Parameters: .....	2
Parameters update .....	3
Agent's Actor and Critic network:.....	3
The Actor's parameters: .....	3
The Critic's parameters: .....	3
Batch normalization of actor and critic:.....	3
Results of Model Training: .....	5
Graph of results .....	5
Potential Improvements: .....	5

## DDPG Model implementation

### Description of DDGP model

The goal of this DDGP model is train an Agent composed of an actor (the Actor) whose task is to approximate the action value function  $Q(s,a)$  by minimizing the squared distance of the Actor's predicted action/value vs it equivalent Bellman Equation while also training a second agent (the Critic) whose task is to evaluate a given situation  $Q(s)$ .

The model uses 2 networks(Critic and Actor) for solving the continuous control game.. The critic is the model that approximate the action value function  $Q(s,a)$  that guides the agent. The Actor is the one that chooses the actions based on the inputs.

### Pseudo Code of the DDPG Algorithm:

1. Run inputs through the actor agent
2. Collect experiences consisting of (actions, rewards, inputs) tuples from the actor
3. When enough experiences are memorized the update Actor & Critic Agent
4. Update critic agent
  - a. Use actor predicted next action to obtain  $Q_{next}$  as  $Critic(action_{next})$ 
    - i. Calculate target  $Q(S,A)$  as  $reward + discount * Q_{next}$
  - b. Calculate the expected  $Q(S,A)$  as  $Critic(action)$
  - c. Calculate Cost function as "target – expected"
5. Update the Agent's parameters:
  - a. For a given state sampled, predict the actions using the Actor
    - i.  $Action_{predicted} = Actor(state)$
  - b. Loss is defined as the average expected Critic Agent value
    - i.  $Average\_over\_actions(Critic(state, action_{predicted}))$
6. Perform as soft update of the Actor and critic target
  - a. Target actor & critic are update as  $x * target + (1-x) * local\_model$
7. Iterate until Agent has discovery an adequate policy.

### Agent Parameters:

***BUFFER\_SIZE = int(1e6) # replay buffer size***

***BATCH\_SIZE = 512 # minibatch size***

***GAMMA = 0.99 # discount factor***

***TAU = 1e-3 # for soft update of target parameters***

***LR\_ACTOR = 1e-4 # learning rate of the actor***

***LR\_CRITIC = 1e-4 # learning rate of the critic***

***WEIGHT\_DECAY = 0      # L2 weight decay***

***UPDATE\_EVERY = 20      # how often to update the network***

## Parameters update

The parameters of the Agent are updated 10 times every 20 steps. The code that performs this is as follow:

***# Learn, if enough samples are available in memory***

***if len(self.memory) > BATCH\_SIZE:***

***for nbtimes in range(10):***

***experiences = self.memory.sample()***

***self.learn(experiences, GAMMA)***

## Agent's Actor and Critic network:

The model's architecture for approximating the  $Q(s,a)$  is the one described by the Google Deep Mind research paper. In pseudo code it performs the following.

### The Actor's parameters:

The actor consist of one single hidden layer fully connected neural network of with one hidden layer of 128 nodes and an output layer of 4 nodes describing the action value function for each 4 possible actions. The output uses the tanH so that the model's output are between -1 and 1

### The Critic's parameters:

The critic consist of a DNN fully connected with 3 hidden layers of 128,64,32 nodes reaching a final output of 1 that is between 0 and 1 to value the current state being passed to him.

The critic model also has a gradient clipping to improve performance

***torch.nn.utils.clip\_grad\_norm(self.critic\_local.parameters(), 1)***

### Batch normalization of actor and critic:

The input of both the actor and the critic use batch normalization.

***super(Critic, self).\_\_init\_\_()***

***self.seed = torch.manual\_seed(seed)***

***self.bn0 = nn.BatchNorm1d(state\_size)***

***def forward(self, state, action):***

***"""Build a critic (value) network that maps (state, action) pairs -> Q-values."""***

```
stateN=self.bn0(state)
```

```
xs = F.relu(self.fcs1(stateN))
```

This has for effect to dampen the volatility that would be caused by an input with high low values. Example one dimension of the input is between -1000,1000 and the other one between 0-1, the first one will dominate the model and training will be poor.

#### **Random Noise generator:**

The Theta parameter has been changed from 0.2 to 0.05. Also, The original code generated numbers between 0-1 but has been changed to the following:

```
#dx = self.theta * (self.mu - x) + self.sigma * np.array([random.random() for i in range(len(x))])
```

```
dx = self.theta * (self.mu - x)+self.sigma*np.random.uniform(-1, 1, len(x))
```

## Results of Model Training:

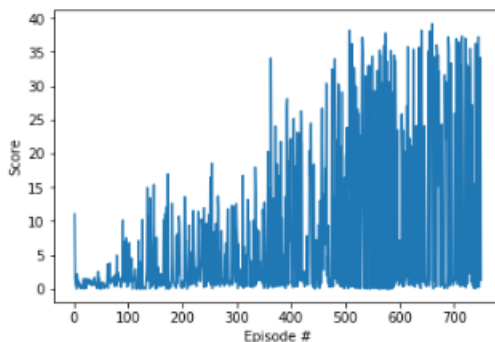
The graph starts with a model that was run for around 250 episodes for a total combined training time of 1,000 episodes.

The model starts at a score of +10 since it already had 250 iterations saved in the .pth files. Something Peculiar is happening. It appears that the model clearly can break above the 30+ score but the problem seems that 50% of the time model gets a +30 score and 50% he gets 0 for an average of  $\approx 16$ . That also explains why the +10 initial score vanishes. So it does appear the model can get 30 for 100 consecutive episode since it is impossible for the algorithm to produce +30 and 0 on next step since parameters are learned.

One brain is used and not 20 since benchmark implementation shows model can be trained with one brain.

## Graph of results

Episode 50	Average Score: 1.06
Episode 100	Average Score: 1.74
Episode 150	Average Score: 3.17
Episode 200	Average Score: 3.09
Episode 250	Average Score: 3.76
Episode 300	Average Score: 4.52
Episode 350	Average Score: 4.34
Episode 400	Average Score: 8.21
Episode 450	Average Score: 8.268
Episode 500	Average Score: 9.55
Episode 550	Average Score: 16.57
Episode 600	Average Score: 14.08
Episode 650	Average Score: 10.30
Episode 700	Average Score: 16.04
Episode 750	Average Score: 16.46



## Potential Improvements:

Results of the training are poor even though multiples improvements have been made, all of those listed in this document (batchNorm, hyperparameter tweaking, NN architecture changes). Using 20 agents cannot make a meaningful difference currently not until the training phase can break above a meaningful threshold of results (20+ for 100 episodes).

