Contents

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-3 # learning rate of the actor

LR_CRITIC = 1e-3 # 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 cause 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 paremeter 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:

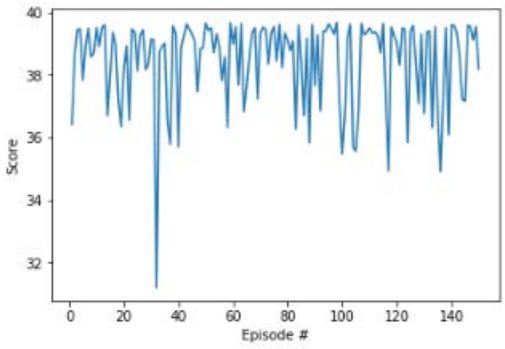
The graph starts with a model that was run for around 250 episodes for a total combined training time of 1,000 episodes.

The model starts at a score of +10 since it already had 250 iterations saved in the .pth files. Something Peculiar is happening. After 500 episodes, 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 ≈16.

Running more episodes and finally between 100 to 150 episodes, the model generates a quantum leap and double performance which gets him to 30+ score for 100+ episodes. I have no explanation as to why such a thing is possible. I can speculate that the network had a "Eureka!" moment.

Graph of results

```
Episode 50      Average Score: 12.72
Episode 100     Average Score: 18.76
Episode 150     Average Score: 29.32
Episode 200     Average Score: 33.51
Episode 250     Average Score: 34.55
Episode 300     Average Score: 34.31
```



Potential Improvements:

Result of 30+ over 100 episodes is achieved but as a very slow rate.

Using 20 independent agents could make a meaningful difference in training time. A PPO algorithm could also be an alternative. The current algorithm could also be improved via prioritized experience sampling.

As an example after 600 iterations, the model gets +30 in some starting conditions and between 0.5-2 in some other conditions. Since it makes no sense to get much better than +30 it might make more sense to prioritize starting positions with current low results.