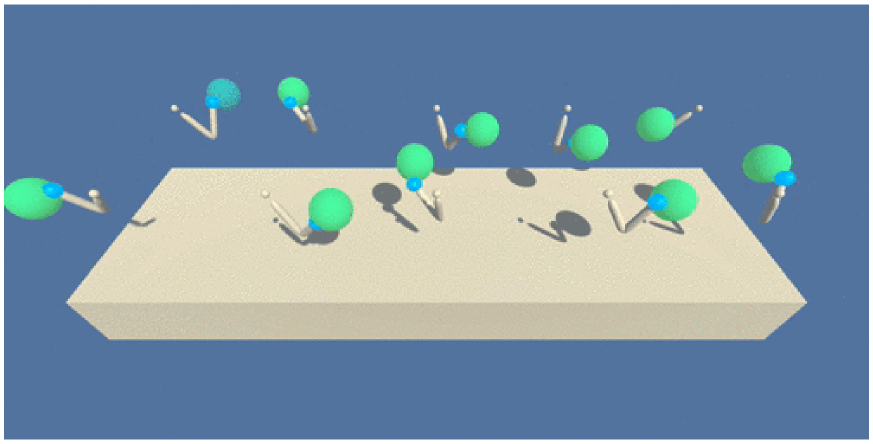


Project 2: Continuous Control

04.4.2021

Project Goals:

The goal of this project is to train an agent (or agents) for controlling a double jointed arm to reach a goal position in the Unity ML-Agents Reacher Environment with proficiency defined as: a score of +30 after 100 episodes for a single agent; or the average score of all 20 agents, averaged over 100 consecutive episodes, reaches +30. An agent receives a score of +0.1 each episode the arm reaches a goal position.

Description of the implementation:

To solve this problem, I use the Deep Deterministic Policy Gradient (DDPG) algorithm since it operates on the continuous action spaces found in this environment. DDPG has an actor model to map the environment states to actions for the current policy, and a critic Q model. The DDPG algorithm, as published by Timothy P. Lillicrap, et Al., in “CONTINUOUS CONTROL WITH DEEP REINFORCEMENT LEARNING”, is given below.

Algorithm 1 DDPG algorithm

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

The implementation consists for four parts:

1. A function called ddpg defined in the Continuous_Control_20_wEPS.ipynb notebook that controls the agent training process.
2. The ddpg_agent_pen_20_eps.py module that contains the agent class, replay buffer class and OUNoise class definitions and methods.
3. The model_ddpg.py module that contains the Actor class and Critic class. The Actor class defines a network, represented by μ in Algorithm 1 above, is a fully connected multilayer perceptron model used to select an action based on the state of the environment. The Critic class defines the Critic network, represented by Q in Algorithm 1 above, is also a fully connected multilayer perceptron model, but it is used to generate the Q targets based on the states and actions, as well as Q expected values.
4. The code in the Continuous_Control_20_wEPS.ipynb notebook calls the ddpg function and then prints out the results once the function has completed training.

<u>Two Implementations, two successful runs:</u>

The Repository consists of two implementations: one with epsilon decay of added noise for exploration to the actions and one without epsilon decay. This is not the epsilon of epsilon greedy policy, but it is used in a similar way as in the DQN method to reduce exploration as the networks train. The Jupyter Notebook with epsilon decay is called

Continuous_Control_20_wEPS.ipynb and the one without is called Continuous_Control_multi.ipynb. Both networks solve the 20 agent environment. However, since the Continuous_Control_multi.ipynb notebook yielded results showing declining average scores beyond around 80 episodes, I created Continuous_Control_20_wEPS.ipynb to see if reducing the probability of exploring with added noise as the networks train might reverse this trend. As can be seen in the commentary below, this was the case and the environment was solved faster.

In addition to the two different Jupyter notebooks, there are two corresponding python modules defining the Agent class: *ddpg_agent_pen_20_eps.py* and *ddpg_agent_pen_20.py*. These two modules are presently the same, but the ddpg_agent_pen_20_eps.py was created in the event hyperparameters need adjustment to run with the epsilon decay defined in the jupyter notebook Continuous_Control_20_wEPS.ipynb.

Hyperparameters:

The following hyperparameters were used:

Hyperparameters used in the Jupyter Notebooks for the two runs

Continuous_Control_20_wEPS.ipynb

Continuous_Control_multi.ipynb

Epsilon (eps): Not the epsilon of the epsilon greedy policy in DQN, but used in a similar manner in the algorithm to reduce the probability of exploration as the networks train (the probability of adding the noise *N* as indicated in Algorithm 1 using a uniform distribution). Epsilon was not used in the Continuous_Control_multi.ipynb notebook.

eps_start: starting value of epsilon

eps_end: ending value of epsilon

eps_decay: factor to multiply epsilon by after each episode

The ddpg() function tests if a random number between 0 and 1 is greater than eps. This result is passed to the Act() method as a flag where if it is True, then noise is not added to the action and if False, then noise is added to the action. Since starting eps at 1 guarantees that the result is False, the first action of the Actor will have added Ornstein–Uhlenbeck process noise, where each successive episode will reduce epsilon by a factor of eps_decay increasing the likelihood of not applying noise. Thus exploration declines as the networks train, but does not go to zero because a lower bound of eps_end maintains a small probability of exploration.

PRINT_EVERY: Sets the length of the moving average used to determine whether the environment has been solved. It is also used to indicate how often to print the moving average: prints every 1 in PRINT_EVERY episodes.

max_t: During an episode, if no Done has been received from the environment after max_t timesteps, then the episode is terminated.

n_episodes: The number of episodes run by DDPG() function.

Hyperparameters for the Agent Class in the modules:

ddpg_agent_pen_20_eps.py

ddpg_agent_pen_20.py

BUFFER_SIZE: the depth of the replay buffer

The replay buffer is defined as a python deque (double ended queue)

BATCH_SIZE: The number of experiences in a sample - A.K.A the mini-batch size

GAMMA: The discount factor as shown in Algorithm 1 (DDPG)

TAU: Defines the proportions of the local and target QNetwork parameters to apply to the target QNetwork's parameter update. This is performed by the soft_update() method of the Agent class.

LR_ACTOR: Learning Rate used by the Optimizer for the Actor

LR_CRITIC: Learning Rate used by the Optimizer for the Critic

WEIGHT_DECAY: Adams optimizer weight decay for the Critic network - not used, set to 0

UPDATE_FREQ: Update the models every one in UPDATE_FREQ calls of the Agent step() method (one in every UPDATE_FREQ trial of every episode).

UPDATE_NUM: Number of updates to perform to the models for every call of the Agent step() method.

Hyperparameters for the Actor and Critic Classes in model_ddpg_.py

fc1_units: The size of the first hidden layer in the Actor network model.

fcs1_units: The size of the first hidden layer in the Critic network model

fc2_units: The size of the second hidden layer in both the Actor and Critic network models

<u>Description of functions in the Jupyter Notebooks</u>:

ddpg() function

ddpg() is defined in Continuous_Control_20_wEPS.ipynb and Continuous_Control_multi.ipynb. The latter takes additional arguments: n_episodes, max_t, eps_start, eps_end and eps_decay. For both notebooks, ddpg() has the arguments:

agent: Is the instantiated Agent object.

env: Is the Unity-ML Reacher environment.

The remaining arguments: n_episodes,, max_t and print_every are defined above.

ddpg() steps through each of n_episodes for a maximum of max_t timesteps. It keeps a 100 episode moving average of reward totals that includes the average score across all the agents, prints those totals to the screen, and breaks when the moving average reaches +30.

ddpg_agent_pen_20_eps.py and ddpg_agent_pen_multi.py

These python modules contain the class definitions for Agent, ReplayBuffer and OUNoise. The Agent Class has the step(), act(), learn(), reset() and soft_update() methods. It also instantiates ReplayBuffer and OUNoise objects. The term “pen” in the filenames indicates they were derived from the ddpg_agent() file in the ddpg pendulum exercise found in the DRLND repository.

The step() method does two actions:

- Adds an experience to the replay buffer for all 20 agents.
- Calls the learn() method **UPDATE_NUM** times every one in **UPDATE_FREQ** time steps.

It is the only method of the Agent class modified to accommodate multiple agents. Following the Udacity Benchmark implementation, this method provides the means to use hyperparameters to determine how often updates are performed to the Actor and Critic models as well as many updates are performed.

The learn() method performs a training step for the actor and critic networks. It also calls soft_update() to copy a portion of the local network parameters to the target networks parameters.

The act() method gets an action from the local actor network with the current environment state as the input. If the add_noise argument is True, then a noise sample from the OUNoise class object noise is added to the actions returned from the local actor network. The noise shifts the actions to explore the environment and potentially actions yielding more consistent rewards.

The OUNoise class provides samples of Ornstein-Uhlenbeck noise. These noise samples differ from random noise samples in that they are a running sum of noise with mean 0. I modified the sample() method of this class from the DDPG exercise to change random.random(), a uniform distribution between 0 and 1, with random.randn(), a gaussian distribution with mean 0.0 and sigma 1. The random.randn() method yielded significantly better results. I got this idea from the code in this github repository:

The ReplayBuffer Class has the replay buffer memory as well as the sample() and add() methods. The sample() method returns a mini-batch of experiences from the replay buffer choosing them at random. The add() method adds an experience to the replay buffer.

model_ddpg.py

Contains the network class, reset() and forward() methods for the Actor and Critic networks. The Actor class takes the state size, action size and hidden size (**fc1** and **fc2**) to create a fully connected network. The Critic class also takes the state size and action size, and hidden size (**fcs1** and **fc2**), but the Critic network also takes in the action as input to generate the q-values. For the Actor class, the forward() method takes the state as input and moves it through the network, returning the action tensor. The Critic class concatenates in the action with the state to generate q-values. Both networks use two batch normalization layers.

Chosen hyperparameters:

Used only in the Continuous_Control_20_wEPS.ipynb notebook

eps_start: 1.0

eps_end: 0.01

eps_decay: 0.999

Used in both Continuous_Control_20_wEPS.ipynb and Continuous_Control_multi.ipynb notebook

PRINT_EVERY: 100

max_t: 1000

n_episodes: 200

Hyperparameters for the Agent Class in the modules:

ddpg_agent_pen_20_eps.py

ddpg_agent_pen_20.py

BUFFER_SIZE: 1000000

BATCH_SIZE: 128

GAMMA: 0.99

WEIGHT_DECAY: 0

TAU: 0.001

LR_ACTOR: 0.001

LR_CRITIC: 0.001

UPDATE_FREQ: 20

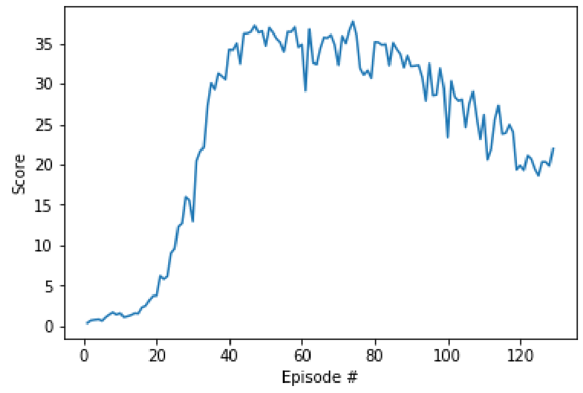
UPDATE_NUM: 10

Hyperparameters for the Actor and Critic Classes in model_ddpg_.py

fc1_units: 264, **fcs1_units**: 264. **fc2_units**: 198

Plot of Rewards:

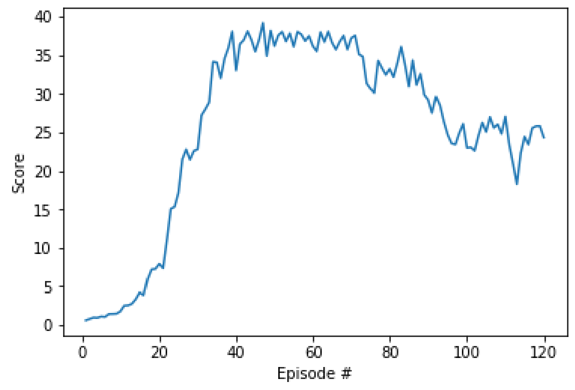
The plot of rewards versus episodes for the chosen hyperparameters using the notebook Continuous_Control_multi.ipynb that does not reduce the probability of adding noise as the number of episodes increases:



+30 average rewards were reached after 129 episodes.

Considering how the average rewards began to decline after around 80 episodes, I added epsilon to reduce the probability of adding noise with successive episodes. Typically, declining accuracy indicates overtraining, but in the case of DDPG, this might indicate the agent has learned the environment and the added noise just destabilizes the training process after a number of episodes. To find out, I created a version with gradual reduced exploration using an approach similar to what was done in Project 1.

The plot of rewards versus episodes for the chosen hyperparameters using the notebook *Continuous_Control_20_wEPS.ipynb* that reduces the probability of adding noise as the number of episodes increases:



+30 average rewards were reached after 120 episodes.

Discoveries from trying different sets of hyperparameters:

I started out with the single agent environment and following the hyperparameters given in the DDPG foundational paper: "CONTINUOUS CONTROL WITH DEEP REINFORCEMENT LEARNING". I also started out with the actor and critic models from the DRLND repository exercise for ddpg-pendulum as well as the Agent python code. During this training, I made the following observations:

- Smaller hidden layer models yielded better performance. My best run used an fc1, fc2, fcs1 size of 32 units.
- A single batch normalization layer, after the first fully connected layer, yielded higher scores than no batch normalization layers, or more than one batch normalization layers.
- A batch size of 128 yielded better results than 64

However, I was never able to get over an average score of 2.46 after 400 episodes. The run took over an hour just to get to 400 episodes, so in the interest of time, I changed to the 20 agent environment.

In the 20 agent environment, I found that none of my discoveries about hyperparameter effects from training the one agent environment carried over to the 20 agent environment, other than batch size. Namely,

- Larger hidden layer models yielded better results.
- Two batch normalization layers was better than one or more than two.

I could not get an average score higher than 6.31 regardless of how I set the hyperparameters. The breakthrough moment came when I discovered a different Ornstein-Uhlenbeck noise algorithm in the https://github.com/ShangtongZhang/DeepRL repository than the DRLND repository. The use of the gaussian noise function random.randn() was used instead of the uniformly distributed random.random() from 0 to 1. This gaussian approach is actually what is found in the standard implementation. I also got slightly better performance by changing sigma from 0.2 to 0.19, but going to 0.18 made it worse. After implementing these changes, I got average scores up to 29.19 - not good enough. Changing the learning rate for the Actor model from 0.0001 to 0.001 was enough to reach the goal of an average of +30 over all agents and over 100 episodes.

Observing the decline in average score as the number of episodes increased over 80, I decided to add an epsilon similar to Project 1, but in this case to reduce the probability of adding the Ornstein-Uhlenbeck noise as the number of episodes increased. This improved results, but not significantly.

Ideas for Future Work:

In the interest of time, I did not try a lot of hyperparameter changes to observe the impact. Runs could take over an hour just to get to 400 episodes. It is quite possible that many of my runs would have reached an average score of +30 if I let it run longer. However, that would have taken an enormous amount of time, so I concentrated on only those hyperparameters that yielded the fastest results. These runs tended to result in fading performance, similar to the Udacity Benchmark implementation, so the objective became getting the highest scores as fast as possible and sustaining them as long as possible to reach the goal average score of +30. For future work, I would try hyperparameters I did not have time to try out, such as:

- Changing the frequency of model updates and number of updates per learning step. I never changed these hyperparameters from the Udacity Benchmark implementation (**UPDATE_FREQ**: 20, **UPDATE_NUM**: 10) but they could have a profound impact on performance.
- Changing the GAMMA from 0.99. I kept this the same for all my runs.
- Going back to the single agent environment and applying what I learned about the Ornstein-Uhlenbeck noise, let it train longer and see if it gets to a score of +30 over 100 episodes.
- Using a different algorithm to reduce the Ornstein-Uhlenbeck noise such as not reducing it unless the average scores begin to decline significantly. This would verify the impact of exploration on a well trained agent.