This report describes the implementation of a Deep Deterministic Policy Gradient (DPDG) algorithm developed to solve Unity ML-agents Reacher V02 environment.

Continuous_control.ipynb

1) Install dependencies by running the first cell (!pip -q install ./python)

2) To train the agent, run the code in cell # 2. This code will organize the interaction with the Unity environment, collect trajectories and run the ddpg algorithm form the ddpg_agent.py file

For each episode,

- Each agent collects a State
- Takes the action A using the policy, observes the reward and the next input frame
- Stores the experience tuple SARS' in replay memory.
- Select a small bunch of tuples from memory randomly and learn from it

ddpq_agent.py

The algorithm implemented in ddpg_agent.py has three classes: Agent, ReplayBuffer and OUNoise

- An agent is initialized with parameters state_size, action_size, and a seed for random number generation in PyTorch.
- ReplayBuffer is initialized with parameters action_size, BUFFER_SIZE, BATCH_SIZE, and seed
- OUNoise takes as inputs size, seed, mu, theta, sigma.

Four neural networks are initialized with the Agent. Basically, two networks with two instances: an Actor and a Critic network, with two versions (target and local). The critic network estimates the value function of policy pi, V(pi) using the TD estimate. The output of the critic will be used to train the actor network, which will take in a state, and output the distribution over possible actions.

The algorithm goes like this:
Input the current state into the actor and get the action to take in that state.
Observe next state and reward, to get your experience tuple s a r s'
Then, using the TD estimate, which is the reward R plus the critic's estimate for s', so r+yV(s';Thetav), the critic network is trained.
Next, to calculate the advantage r+yV(s') - V(s), we also use the critic network.
And finally, the actor is trained using the calculated advantage as a baseline.

This type of architecture is well suited for continuous action spaces, with the critic network learning the optimal action for every given state, and then passing that optimal action to the actor network which uses it to estimate the policy.

We use two networks for each of the two to control the update of the weights (if we were updating at every time step, our policy estimate would diverge). Only when the function learn2() is called after each episode the weights are transferred between the target and local versions.

Two functions are defined for selecting and performing an action:

step(self, state, action, reward, next_state, done):

This function saves the experiences of 20 agents in the replay memory buffer

learn2():

This function is called after each episode to sample from the replay buffer and learn from the experience accumulated from 20 agents.

act (self, state, eps):

This function selects an action for a given state following the policy encoded by the NN function approximator. The architecture of this network is defined in the model.py file (see below).

First, a transformation of the current state from numpy to torch is performed.
Then, a forward pass on the actor_local is performed. Data is moved to a cpu and to numpy.
Noise is added, and clipping between -1 and 1 is applied.

learn2(self, experiences, gamma):

This function will sample a bunch of experiences and learn from them by calling the learn function (see below).

learn(self, experiences, gamma):

- This function will update the Actor and Critic network's weights given a batch of experience tuples.
- In the update critic section of the function, it will first get the max predicted Q values (for next states) from the critic_target and actor_target models, and compute Q targets for current states. Then, it will get the expected Q values

from the critic_local model, compute the loss and minimize the loss using gradient descent. Note that we use gradient clipping.

- In the update actor section, we compute the loss of he acor, and get the predicted actions.
- The function soft_update is called in the end to update the target networks.

soft update (local_model, target_model, tau):
This function grabs all of the target_model and the local_model parameters (in the zip command), and copies a mixture of both (defined by Tau) into the target_param.
The target network receives updates that are a combination of the local (most up to date network) and the target (itself). In general, it will be a very large chunk of itself, and a very small chunk of the other network.

Replay Buffer

The replay buffer implemented as a class retains the end most recent experience tuples. If not enough experience is available to the agent (i.e., if self.memory < BATCH_SIZE), no learning takes place.
Note that we do not clear out the memory after each episode, which enables to recall and build batches of experience from across episodes.

The buffer is implemented with a Python deque. Given that maxlen is specified to BATCH_SIZE, our buffer is bounded. Once full, when new items are added, a corresponding number of items are discarded from the opposite end.

OUNoise

This class is implemented to create some random noise

Neural Network Architecture

Two classes are instantiated in the model.py file for Actor and Critic networks.

The actor network

The Actor network is estimating the policy. It was built using the PyTorch nn package. The network is defined by subclassing the torch.nn.Module class. The architecture includes an input layer (of size = state size), two fully connected hidden layers of 400 and 300 units and an output layer (size = action size).

RELU activation (Regularized linear units) is applied in the forward function for the first two layers. However, note that on the output side, given the continuous space we use the tahn activation function.

The critic network

The Critic network is approximating the Q function. It is defined by subclassing the torch.nn.Module class. The architecture includes an input layer (of size = state size), one fully connected hidden layer of 400 units, a second hidden layer of 300 + action size units and an output layer (size = action size).

RELU activation (Regularized linear units) is applied in the forward function. Note that in the forward pass, the action is being concatenated to get the value of the specific state-action pair.

Chosen hyperparameters

BUFFER_SIZE	= int(5e5)	# 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	= 3e-3	# learning rate of the critic
WEIGHT_DECAY	= 0	# L2 weight decay

The agent was trained until an average score of +30 for all 20 agents, over 100 consecutive episodes was reached. This was achieved after **442** episodes:

```
Episode 1, Mean last 100: 0.01, Mean current: 0.01, Max: 0.16, Min: 0.00
```

```
Episode 10, Mean last 100: 0.44, Mean current: 0.79, Max: 1.61, Min: 0.19
Episode 20, Mean last 100: 0.63, Mean current: 0.88, Max: 1.74, Min: 0.33
Episode 30, Mean last 100: 0.82, Mean current: 1.38, Max: 3.31, Min: 0.42
Episode 40, Mean last 100: 0.98, Mean current: 1.35, Max: 2.56, Min: 0.38
Episode 50, Mean last 100: 1.07, Mean current: 2.53, Max: 4.39, Min: 0.92
Episode 60, Mean last 100: 1.21, Mean current: 1.71, Max: 3.51, Min: 0.40
Episode 70, Mean last 100: 1.36, Mean current: 2.55, Max: 5.29, Min: 0.86
Episode 80, Mean last 100: 1.56, Mean current: 2.27, Max: 4.20, Min: 0.45
Episode 90, Mean last 100: 1.70, Mean current: 2.77, Max: 5.69, Min: 1.25
Episode 100, Mean last 100: 1.81, Mean current: 2.46, Max: 4.69, Min: 0.85
Episode 110, Mean last 100: 2.03, Mean current: 2.50, Max: 5.83, Min: 0.23
Episode 120, Mean last 100: 2.22, Mean current: 2.88, Max: 7.54, Min: 0.48
Episode 130, Mean last 100: 2.50, Mean current: 3.67, Max: 5.88, Min: 1.30
Episode 140, Mean last 100: 2.79, Mean current: 4.56, Max: 11.11, Min: 1.00
Episode 150, Mean last 100: 3.15, Mean current: 5.03, Max: 12.43, Min: 0.24
Episode 160, Mean last 100: 3.54, Mean current: 7.47, Max: 10.58, Min: 0.56
Episode 170, Mean last 100: 4.14, Mean current: 8.98, Max: 24.94, Min: 3.74
Episode 180, Mean last 100: 4.73, Mean current: 9.98, Max: 13.22, Min: 4.79
Episode 190, Mean last 100: 5.48, Mean current: 8.40, Max: 12.44, Min: 3.62
Episode 200, Mean last 100: 6.09, Mean current: 9.17, Max: 26.95, Min: 3.00
Episode 210, Mean last 100: 6.85, Mean current: 12.57, Max: 18.44, Min: 2.83
Episode 220, Mean last 100: 8.02, Mean current: 17.21, Max: 28.22, Min: 8.33
Episode 230, Mean last 100: 9.14, Mean current: 15.11, Max: 24.28, Min: 6.12
Episode 240, Mean last 100: 10.12, Mean current: 11.80, Max: 20.36, Min: 5.07
Episode 250, Mean last 100: 11.25, Mean current: 16.58, Max: 27.81, Min: 7.03
Episode 260, Mean last 100: 12.25, Mean current: 17.77, Max: 27.78, Min: 4.87
Episode 270, Mean last 100: 13.08, Mean current: 17.87, Max: 30.78, Min: 2.15
Episode 280, Mean last 100: 14.17, Mean current: 19.68, Max: 30.80, Min: 8.40
Episode 290, Mean last 100: 15.37, Mean current: 21.67, Max: 37.57, Min: 3.79
Episode 300, Mean last 100: 17.14, Mean current: 33.12, Max: 36.64, Min: 22.92
Episode 310, Mean last 100: 18.96, Mean current: 28.59, Max: 35.22, Min: 10.04
Episode 320, Mean last 100: 20.32, Mean current: 29.23, Max: 35.30, Min: 10.96
Episode 330, Mean last 100: 21.59, Mean current: 29.25, Max: 37.18, Min: 10.27
Episode 340, Mean last 100: 23.17, Mean current: 28.71, Max: 37.93, Min: 15.10
Episode 350, Mean last 100: 24.26, Mean current: 26.27, Max: 36.65, Min: 4.30
Episode 360, Mean last 100: 25.35, Mean current: 27.65, Max: 38.55, Min: 10.24
Episode 370, Mean last 100: 25.93, Mean current: 17.36, Max: 28.50, Min: 3.70
Episode 380, Mean last 100: 26.43, Mean current: 24.17, Max: 36.03, Min: 9.77
Episode 390, Mean last 100: 26.92, Mean current: 27.82, Max: 34.42, Min: 10.47
Episode 400, Mean last 100: 27.46, Mean current: 35.19, Max: 38.91, Min: 26.63
Episode 410, Mean last 100: 28.02, Mean current: 34.85, Max: 38.92, Min: 22.62
Episode 420, Mean last 100: 28.66, Mean current: 33.29, Max: 38.37, Min: 11.36
Episode 429, Mean last 100: 29.33, Mean current: 35.47, Max: 39.03, Min: 22.64
Episode 430, Mean last 100: 29.39, Mean current: 35.40, Max: 39.34, Min: 16.03
Episode 440, Mean last 100: 29.91, Mean current: 33.01, Max: 39.05, Min: 22.13
Episode 450, Mean last 100: 30.57, Mean current: 36.23, Max: 39.28, Min: 31.58
Episode 460, Mean last 100: 31.31, Mean current: 32.83, Max: 38.67, Min: 22.74
Episode 470, Mean last 100: 32.53, Mean current: 33.84, Max: 39.22, Min: 17.15
Episode 480, Mean last 100: 33.54, Mean current: 35.61, Max: 39.31, Min: 25.48
Episode 490, Mean last 100: 34.35, Mean current: 36.03, Max: 38.78, Min: 25.16
Episode 500, Mean last 100: 34.51, Mean current: 33.95, Max: 39.37, Min: 18.65
Episode 510, Mean last 100: 34.36, Mean current: 31.50, Max: 38.44, Min: 17.04
Episode 520, Mean last 100: 34.08, Mean current: 35.35, Max: 39.41, Min: 18.52
Episode 530, Mean last 100: 33.75, Mean current: 34.24, Max: 38.55, Min: 21.77
Episode 540, Mean last 100: 33.46, Mean current: 32.73, Max: 39.15, Min: 15.25
Episode 550, Mean last 100: 33.19, Mean current: 30.58, Max: 38.93, Min: 18.91
```

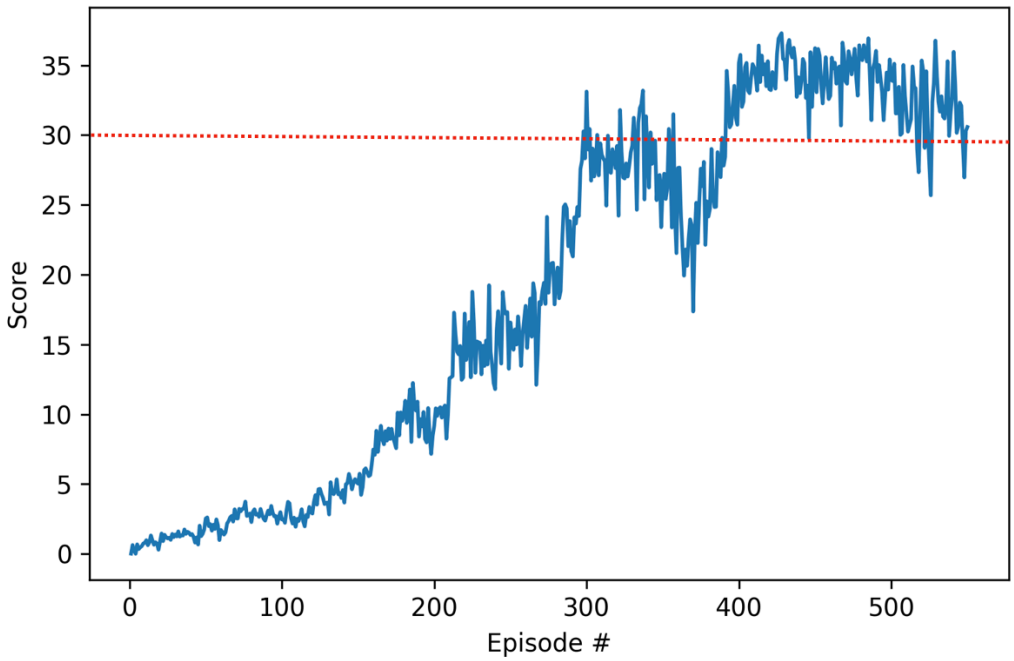

Figure 1. Figure shows that average reward over 100 episodes and over all 20 agents. Dotted line indicates an average of +30.

Instructions to train and test the agent's performance

To train the agent, run the ddpg function in the contionus_control.ipynb file (first cell)

The function will train the agent for 800 episodes

To observe a trained agent, run cell N3 which will load the policy stored in checkpoint_critic_V08_12.pth and checkpoint_actor_V08_12.pth

Ideas for future work

The agent is able to learn the task however, a number of improvements could be achieved.
For instance, it would be interesting to check other neural network architectures such as PPO or AC2. In addition, a challenging idea would be to train the agent using raw pixels.