Learning Algorithm

I adapted an implementation of the deep deterministic policy gradient algorithm (DDPG) with OpenAI Gym's BipedalWalker environment, which I found in the Udacity GitHub repository for the Deep Reinforcement Learning Nanodegree programme.

This learning algorithm utilises two neural networks (both are multilayer perceptrons) to teach an agent or 20 independent agents informed by the same neural networks to solve the Reacher environment.

The actor network takes a state (33 inputs), performs a linear transformation, passes the results through the ReLU activation function, and returns 128 outputs, which go into the second layer of neurons. The second layer transforms them into 128 outputs again in the same manner. Finally, there's a third layer which takes the second layer's outputs, performs a linear transformation, and passes the results through the hyperbolic tangent function (tanh) to produce four final outputs. They are the entries of the four-dimensional action vector corresponding to the state.

The critic network takes a state (33 inputs), performs a linear transformation, passes the results through the leaky ReLU activation function, and returns 128 outputs. The second layer transforms them linearly, passes the results through the leaky ReLU activation function, and concatenates the results with an action vector (four inputs), leading to 132 outputs. Finally, the third layer transforms them (132 inputs) linearly into a scalar value without applying an activation function. This scalar value is the action value of the state-action pair.

In this implementation, experience replay and fixed target neural networks are employed to boost training stability and efficiency. The former decorrelates the sequential observations made by the agent in an episode of training. The latter works by deploying two instances of each neural network (local and target) to decorrelate the target Q-value (calculated using the target actor and critic networks) and predicted Q-value (given by the local critic network) when the local critic network is being updated.

Also, because the actor network's predictions are deterministic, the implementation relies on something other than the epsilon-greedy policy to balance between exploitation and exploration. Noises generated by the Ornstein-Uhlenbeck process are added to the deterministic predictions to enable exploration.

Here are the major steps of this implementation.

1. Initialisation. Two instances of each multilayer perceptron are created and initialised with the same dimensions and seed. One is the target network and the other is the local network.
2. Training. The agent (or each agent when there are 20) is trained in an episodic task. During each episode, each transition (time step) is stored as a tuple, which

is also called an observation in this report, in a replay buffer. It comprises the agent's current state, the action it takes, the reward for this state-action pair, the next state, and whether the transition ends the episode. The action is selected based on the local actor network's outputs and the Ornstein-Uhlenbeck process.

3. When the replay buffer contains more than *BATCH_SIZE* tuples, all four neural networks are updated. This should be considered a part of training.
    - *BATCH_SIZE* tuples are sampled randomly from the replay buffer.
    - For each sampled tuple, both target networks are used to compute the target Q-value without considering the Ornstein-Uhlenbeck process. It means that DDPG is an off-policy algorithm because the behaviour and target policies are different.
    - For each target Q-value, the difference between it and the Q-value predicted by the local critic network is computed. The sum of these differences over the sampled tuples (loss function) is used to update the local critic network's weights by backpropagation.
    - After updating the local critic network, it is used to update the local actor network. For each sampled tuple, the state is fed to the local actor network, which predicts an action. The state-action pair is fed to the local critic network. The sum of the local critic network's predicted Q-values over the sampled tuples (objective function) is used to update the local actor network's weights by backpropagation.
    - After updating the local networks, the target networks are updated by blending with their updated local counterparts. A small fraction (*TAU*) of the updated local weights is added to a large fraction (1-*TAU*) of the target weights to update the latter.
4. Termination. When an episode ends, the cumulative reward for the episode is calculated for the agent (or each agent when there are 20). When the average cumulative reward over 100 consecutive episodes is at least 30 per agent, the algorithm terminates.

At first, I used the default hyperparameters in the provided solution to OpenAI Gym's BipedalWalker environment.

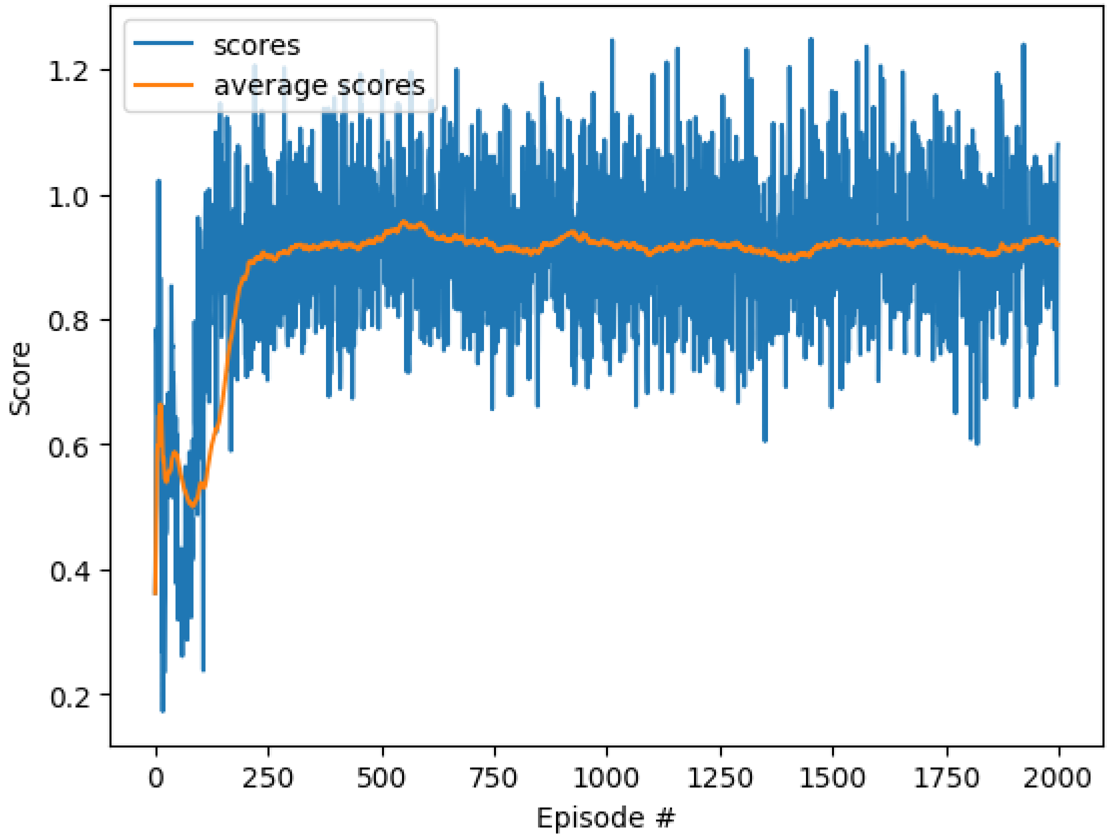
- Number of learning agents (*N_AGENTs*): 1.
- Maximum number of episodes (*n_episodes*): 2000.
- Maximum number of transitions in an episode (*max_t*): 700.
- Maximum number of tuples in the replay buffer (*BUFFER_SIZE*): 1e6.
- Number of tuples sampled from the replay buffer (*BATCH_SIZE*): 128.
- Discount rate in temporal difference learning (*GAMMA*): 0.99.
- Update rate of both target networks (*TAU*): 1e-3.
- Learning rate of the local actor network (*LR_ACTOR*): 1e-4.

- Learning rate of the local critic network (*LR_CRITIC*): 3e-4.
- L2 weight decay (regularisation coefficient) applied to the local critic network (*WEIGHT_DECAY*): 0.0001.
- Ornstein-Uhlenbeck process's long-term mean (*mu*), rate of mean reversion (*theta*), and magnitude of random fluctuations (*sigma*), as well as the multiplicative discount factor for *sigma* (*NOISE_DECAY*), applied at the start of every episode: 0, 0.15, 0.2, and 1.

At first, I trained just one agent using the default hyperparameters. When it was not learning efficiently, I decided to train 20 agents in parallel to explore more state-action pairs. Furthermore, because the environment only rewards and not penalises, I decided to increase the maximum number of transitions in one episode to give the agents more time to learn. To accelerate learning, I increased the learning rates by up to an order of magnitude. Finally, I decided to reduce *NOISE_DECAY* to encourage the agents to explore initially and exploit after a while.

- Number of learning agents (*N_AGENTs*): 20.
- Maximum number of episodes (*n_episodes*): 2000.
- Maximum number of transitions in an episode (*max_t*): 1000.
- Maximum number of tuples in the replay buffer (*BUFFER_SIZE*): 1e6.
- Number of tuples sampled from the replay buffer (*BATCH_SIZE*): 128.
- Discount rate in temporal difference learning (*GAMMA*): 0.99.
- Update rate of both target networks (*TAU*): 1e-3.
- Learning rate of the local actor network (*LR_ACTOR*): 1e-3.
- Learning rate of the local critic network (*LR_CRITIC*): 1e-3.
- L2 weight decay (regularisation coefficient) applied to the local critic network (*WEIGHT_DECAY*): 0.0001.
- Ornstein-Uhlenbeck process's long-term mean (*mu*), rate of mean reversion (*theta*), and magnitude of random fluctuations (*sigma*), as well as the multiplicative discount factor for *sigma* (*NOISE_DECAY*), applied at the start of every episode: 0, 0.15, 0.2, and 0.996.

Plot of Rewards



Using the learning algorithm, I trained 20 agents for 2000 episodes. In the last 100 episodes, the 20 agents achieved an average episodic cumulative reward (average score) of 0.92 per agent over the time window.

Ideas for Future Work

I would like to experiment with alternatives to the Ornstein-Uhlenbeck process to switch between exploration and exploitation adaptively, depending on how well the agent is learning. Also, if I had time, I would test different neural network architectures systematically.