Introduction

This report describes a Python with Pytorch implementation of a Deep Deterministic Policy Gradients (DDPG) algorithm to solve the Reacher environment (20 agents). This was completed as part of my 2nd project from the Udacity Deep Reinforcement Learning Nanodegree, Continuous Control. The agent solves the environment in 258 episodes.

The Environment

The environment used for this project is the Reacher environment created by Unity. The environment contains 20 double jointed arms. The objective of the environment is to keep the hands of these arms in a target location for as many time steps as possible. The environment is considered solved when an average reward of +30 across 100 episodes is achieved for all 20 arms. The agents receive a reward of +0.1 if the arm stays in the target location.

The observation space of the state contains 33 variables which contain information about the position, velocity, angular velocities and rotation of the arm. Each action is a vector of 4 numbers that are values between -1 and 1.

Learning Algorithm

The algorithm used is a Deep Deterministic Policy Gradient (DDPG). DDPG is a reinforcement learning algorithm is an off-policy method that uses the actor-critic technique. The actor-critic technique combines both Q-learning and policy gradients. The actor is a policy network that takes an input state and outputs an exact continuous action instead of a probability distribution over the actions. The critic is a value-based network that takes a state, action pair as input and outputs a Q-value.

The DDPG also uses an Experience Replay buffer, replay.py, so the agent can randomly sample experiences without them being correlated. Similar to fixed Q-targets the actor and critic uses online and target models, this is why DDPG is an off-policy method as the online model learns and the other is fixed until after a defined number of time steps then the weights of the online model are copied to the target model using a soft update as shown below,

```
def soft_update(self, local_model, target_model, tau):

    for target_param, local_param in zip(target_model.parameters(), local_model.parameters()):
        target_param.data.copy_(tau*local_param.data + (1.0-tau)*target_param.data)
```

The target models' weights are fixed so a loss can be calculated between the 2 models therefore making the agent converge quicker.

Another feature added to the DDPG is a noise is added to the action to allow the agent to explore the action space, this is similar to epsilon-greedy in Q-learning. The noise is decayed over time so the agent will exploit more and explore less. The noise is generated by the Ornstein-Uhlenbeck process. The process is a stochastic process that is Gaussian and Markov.

In the implenementation, firstly the agents choose an action using act function in the Agent class found in ddpg_agent.py. The local actor chooses actions for all 20 agents based on the states and the current weights of the model, some noise is then added to the action and then the action is returned as a value between -1 and 1.

The actions for the agents are then passed through the environment to get the rewards and next states of the agents. With this information the agents take a step to add an experience to the replay buffer, in the implementation a vanilla experience replay and a prioritized experience replay buffer were used. If the number of experiences in the buffer is greater than the batch size, 256, and the agent has reached the defined number of time steps then the agent will learn from the experiences.

In the learn step the target actor gets the predicted next action from the next state and the target critic gets the predicted next Q value from the next state. The Q target for the current state is calculated using the bellman equation and the Q values from the current states are calculated using the local critic model. Using the target and local values the loss for the critic is found and the critic model is then updated. A similar process happens for the actor model. The predicted next actions are found from the state using the local actor, those are then fed into the local critic and the loss is calculated which then means we can update the actor model.

Model Architecture

The DDPG uses 2 architectures, one for the actor model and one for the critic model. The target and local models in the actor and critic have the same architectures. The architecture for actor model is as follows:

1. Input layer 33 nodes to map the state
2. Hidden layer 400 nodes with ReLU activation
3. Hidden layer 300 nodes with ReLU activation
4. Output layer 4 nodes with tanh activation

Tanh is used as we need to output a continuous value between -1 and 1 and output size of 4 for the 4 values in the action vector. The Pytorch actor model is shown below,

```
class Actor(nn.Module):
    # Actor (Policy) Model

    def __init__(self, state_size, action_size, seed, fc1_units=400, fc2_units=300):

        super(Actor, self).__init__()
        self.seed = torch.manual_seed(seed)
        self.fc1 = nn.Linear(state_size, fc1_units)
        self.fc2 = nn.Linear(fc1_units, fc2_units)
        self.fc3 = nn.Linear(fc2_units, action_size)
        self.reset_parameters()

    def reset_parameters(self):
        self.fc1.weight.data.uniform_(*hidden_init(self.fc1))
        self.fc2.weight.data.uniform_(*hidden_init(self.fc2))
        self.fc3.weight.data.uniform_(-3e-3, 3e-3)

    def forward(self, state):
        # Build an actor network that maps input states to actions.
        x = F.relu(self.fc1(state))
        x = F.relu(self.fc2(x))
        return torch.tanh(self.fc3(x))
```

The architecture for critic model is as follows:

1. Input layer 33 nodes to map the state
2. Hidden layer 400 nodes + the action input with ReLU activation
3. Hidden layer 300 nodes with ReLU activation
4. Output layer 1 node with ReLU activation

The Pytorch critic model is shown below,

```
class Critic(nn.Module):
    # Critic (Value) Model

    def __init__(self, state_size, action_size, seed, fc1_units=400, fc2_units=300):

        super(Critic, self).__init__()
        self.seed = torch.manual_seed(seed)
        self.fc1 = nn.Linear(state_size, fc1_units)
        self.fc2 = nn.Linear(fc1_units+action_size, fc2_units)
        self.fc3 = nn.Linear(fc2_units, 1)
        self.reset_parameters()

    def reset_parameters(self):
        self.fc1.weight.data.uniform_(*hidden_init(self.fc1))
        self.fc2.weight.data.uniform_(*hidden_init(self.fc2))
        self.fc3.weight.data.uniform_(-3e-3, 3e-3)

    def forward(self, state, action):
        # Build a critic network that maps state, action pairs to Q-values.
        x = F.relu(self.fc1(state))
        x = torch.cat((x, action), dim=1)
        x = F.relu(self.fc2(x))
        return self.fc3(x)
```

Results

Vanilla DDPG

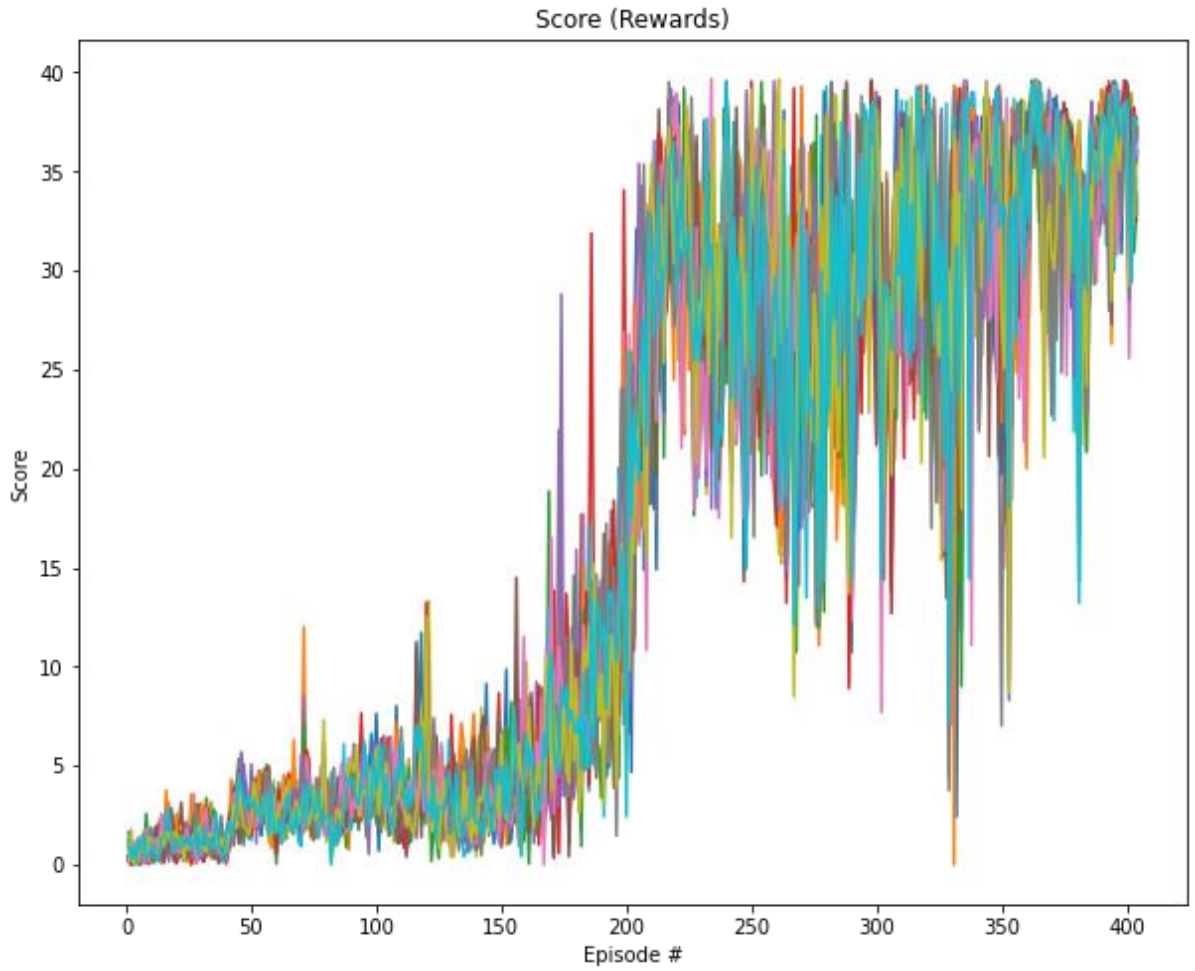
With the following hyperparameters,

{buffer_size: 20000,
max time steps: 3000,
batch_size: 256,
noise decay: 0.99,
gamma: 0.9,
update_every: 20,
Ornstein-Uhlenbeck, mu: 0,
Ornstein-Uhlenbeck, theta: 0.15,
Ornstein-Uhlenbeck, sigma: 0.2,
actor learning rate: 1e-3,
critic learning rate: 1e-3,
soft update tau: 1e-3}

```
Episode 100    Average Score: 1.35
Episode 200    Average Score: 3.48
Episode 300    Average Score: 17.57
Episode 400    Average Score: 29.84
Episode 404    Average Score: 30.01
```

```
Environment solved in 404 episodes! Average Score: 30.01
Wall time: 58min 49s
```

The vanilla ddpg managed to solve the environment in 404 episodes with a time of 58 minutes 49 seconds. The graph below shows the scores of all 20 agents over time.



DDPG with Prioritized Experience Replay

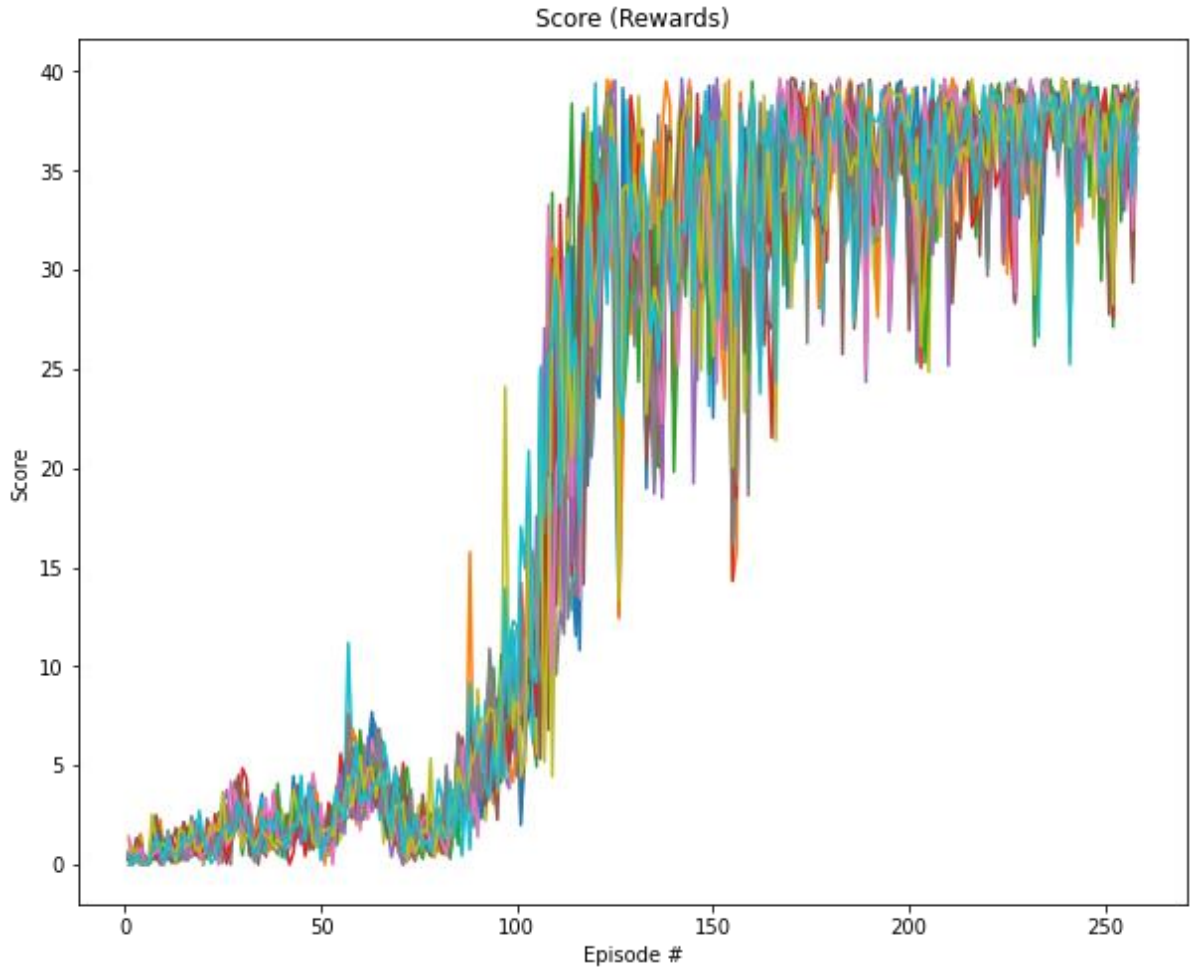
With the following hyperparameters,

{buffer_size: 20000,
max time steps: 3000,
batch_size: 256,
noise decay: 0.99,
gamma: 0.9,
update_every: 20,
Ornstein-Uhlenbeck, mu: 0,
Ornstein-Uhlenbeck, theta: 0.15,
Ornstein-Uhlenbeck, sigma: 0.2,
actor learning rate: 1e-3,
critic learning rate: 1e-3,
soft update tau: 1e-3,
a: 0.5

e: 1e-5,
beta: 0.4}

```
Episode 100    Average Score: 1.42
Episode 200    Average Score: 15.75
Episode 258    Average Score: 30.09
Environment solved in 258 episodes!  Average Score: 30.09
Wall time: 1h 20min 42s
```

The DDPG with prioritized experience replay managed to solve the environment in 258 episodes wit h a time of 1 hour 20 minutes 42 seconds. The graph below shows the scores of all 20 agents over ti me



Future Work

In the future more work could be done on trying to optimize the hyperparameters and work on the models. Using batch normalization, changing the architecture and experimenting with the number of nodes in the hidden layers was tried but more work could be done on this to improve the agent's performance. Moving away from DDPG other actor-critic methods could be tried such as PPO and A3C or A2C.

References

To help me with the implementation of a DDPG for this project I looked into and studied the following material:

- Continuous control with deep reinforcement learning paper by Timothy P. Lillicrap, Jonathan J. Hunt, Alexander Pritzel, Nicolas Heess, Tom Erez, Yuval Tassa, David Silver, Daan Wierstra
- Udacity deep reinforcement learning repository
- DeepRL repository by ShangtongZhang