Project 2: Continuous Control Report

1. Introduction

The aim of this project was to train multiple agents to keep to a moving target area by using a double-jointed arm. There are total of 20 identical agents, where each one uses a copy of the environment and has its own double-jointed arm and moving target location. A reward of +0.1 is provided for each step that the agent's hand is in the goal location, where the goal of the agents is to maintain its position at the target location for as many timesteps as possible.

The state space consists of 33 variables corresponding to position, rotation, velocity, and angular velocities of the arm. Given this information, the agent learns how best to use an action vector containing four continuous actions that correspond to torque applicable to the two joints within the arm. Every entry in the action vector is a number between -1 and 1.

The approach taken uses a DDPG model with the Ornstein-Uhlenbeck process and Experience Replay, solving the environment in 118 episodes with an average score of +30 over 100 consecutive episodes, across all agents.

2. Model Implementation

The model implemented focuses on a Deep Deterministic Policy Gradient (DDPG), where the architecture used consisted of two neural networks, an actor and a critic. Both contained four layers: an input layer with 33 inputs representing the environment state space, two hidden layers both with 128 hidden nodes and an output layer with 4 output nodes, one output for each corresponding action. There is one addition to the critics network within the first hidden layer, the actions are added as inputs to create the effect of a Deep Q-Network (DQN). Both the hidden layers are passed through a ReLU activation function to provide non-linearity to the models training and are trained using Gradient Ascent with the Adam optimizer to update the models weights. Additionally, the batches of data passed into each network are normalized to help accelerate the agents training.

The actor is used to approximate the optimal policy deterministically, where it always outputs the best believed action for any given state, learning the best action to take. This is an approximate maximiser that is used to calculate a new target value for training the action-value function. The critic is then used to evaluate the optimal action-value function by using the actors best believed action.

When neural networks are used with an action-value function, they can become very unstable due to the non-linearity provided by the function approximator (neural network). To improve the models convergence, three modifications are applied: an experience replay buffer, fixed Q-targets and the Ornstein-Uhlenbeck process.

The experience replay buffer is used to store agent experiences (state, action, reward and next state) when the agent interacts with the environment, allowing the agent to learn additional information from each experience. Using this replay buffer, the agent can learn from individual experiences multiple times and recall rare occurrences after randomly sampling experiences from the buffer.

Fixed Q-targets are used to generate the TD target within the TD error and prevent correlation between the TD target and the parameters that are being changed. Without this, it would be like chasing a moving target, which can cause instability within the network. The fixed Q-targets (w^-) are a copy of the parameters (w) that are slowly updated after a set number of learning steps, called a soft update. The targets are a separate network, using an identical architecture as the original, that are compared against the parameters that are updated at every learning step. Both the actor and critic use their own separate sets of weights to account for this implementation by using a soft updates strategy. At every step, a minimum of 0.01% of the regular network weights blended with the target network weights.

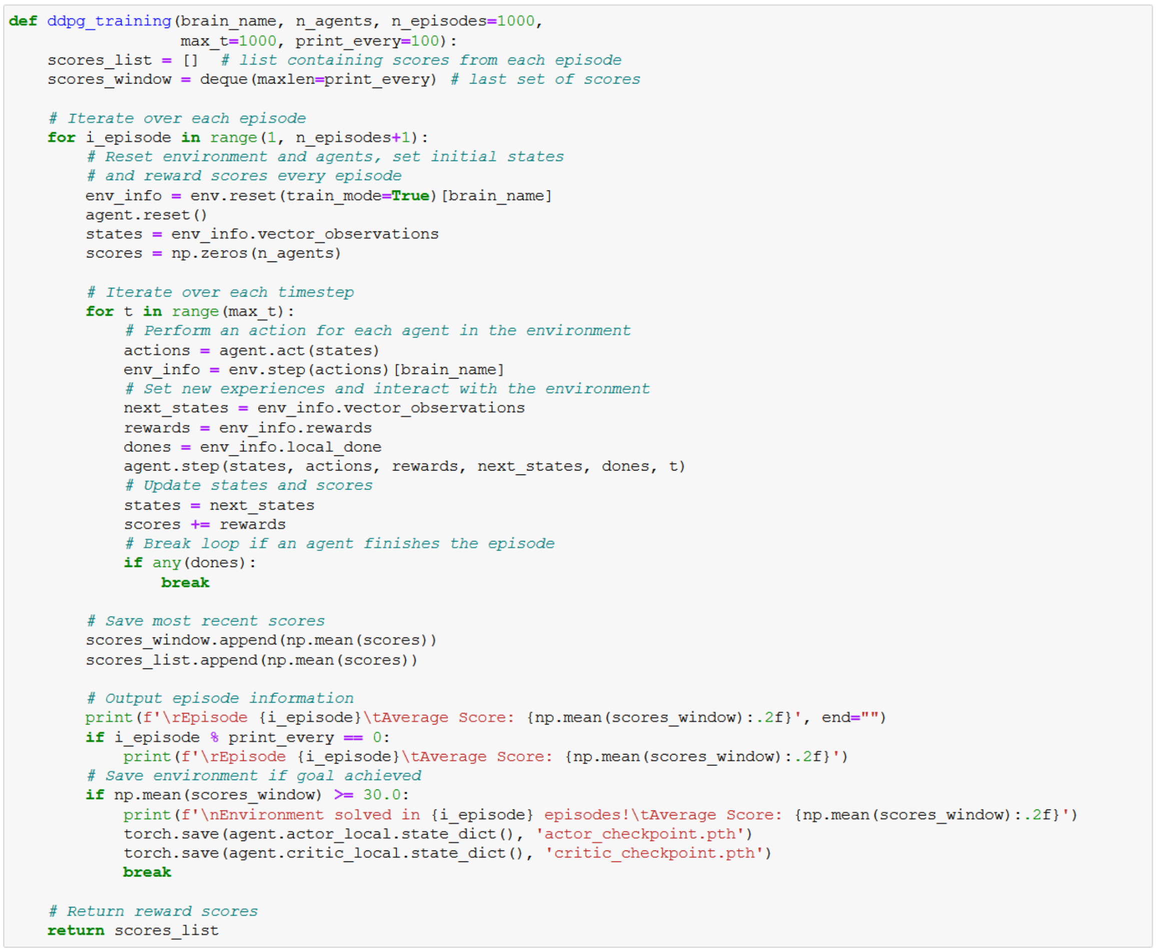
The Ornstein-Uhlenbeck process (OUP) is used to add noise to the action itself, providing the agent an opportunity to explore the environment more. As the actions used are continuous, an epsilon-greedy policy wouldn't be sufficient enough on its own for the agents to successfully explore the environment, hence the need for noisy actions as well.

3. DDPG Algorithm

The DDPG algorithm focuses on one core function: ddpg_training. This function returns a list of average scores for all episodes across all agents, once the model has achieved an average reward score of over 30.0. The hyperparameters used are as follows:

- BUFFER_SIZE = int(1e5): the experience replay buffer size stores a maximum of 100,000 experiences.
- BATCH_SIZE = 256: a total of 256 samples are created and stored within the replay buffer at a time.
- GAMMA = 0.99: a discount factor of 0.99 is used during the fixed Q-target calculation.
- TAU = 1e-3: a tau factor of 0.001 is used within the soft update of the target parameters.
- LR_ACTOR = 1e-3: a learning rate of 0.001 is used to create minor adjustments to the weight parameters within the Adam optimizer, during training, for the actor network.
- LR_CRITIC = 1e-3: a learning rate of 0.001 is used to create minor adjustments to the weight parameters within the Adam optimizer, during training, for the critic network.
- EPS_START = 1: the starting value for epsilon is set to 1, stating every action to be random in the beginning.
- EPS_END = 0.05: the epsilon value stops decaying at 0.05, causing agents to randomly select an action with a 5% chance, while always selecting the best action otherwise.
- EPS_DECAY = 1e-6: a noise decay rate of 0.000001 is used to reduce the noise within actions produced by OUP.
- UPDATE_EVERY = 20: at every 20 learning steps the Q-targets are updated to the latest parameters.
- NUM_UPDATE = 10: at every 20 learning steps the Q-targets are updated 10 times.

The hyperparameters are utilised within the agent.py file, which focuses on the Agent class that is a key component of the DDPG algorithm, which can be seen in Figure 1.



```python
def ddpg_training(brain_name, n_agents, n_episodes=1000,
                  max_t=1000, print_every=100):
    scores_list = []  # list containing scores from each episode
    scores_window = deque(maxlen=print_every) # last set of scores

    # Iterate over each episode
    for i_episode in range(1, n_episodes+1):
        # Reset environment and agents, set initial states
        # and reward scores every episode
        env_info = env.reset(train_mode=True)[brain_name]
        agent.reset()
        states = env_info.vector_observations
        scores = np.zeros(n_agents)

        # Iterate over each timestep
        for t in range(max_t):
            # Perform an action for each agent in the environment
            actions = agent.act(states)
            env_info = env.step(actions)[brain_name]
            # Set new experiences and interact with the environment
            next_states = env_info.vector_observations
            rewards = env_info.rewards
            dones = env_info.local_done
            agent.step(states, actions, rewards, next_states, dones, t)
            # Update states and scores
            states = next_states
            scores += rewards
            # Break loop if an agent finishes the episode
            if any(dones):
                break

        # Save most recent scores
        scores_window.append(np.mean(scores))
        scores_list.append(np.mean(scores))

        # Output episode information
        print(f'\rEpisode {i_episode}\tAverage Score: {np.mean(scores_window):.2f}', end="")
        if i_episode % print_every == 0:
            print(f'\rEpisode {i_episode}\tAverage Score: {np.mean(scores_window):.2f}')
        # Save environment if goal achieved
        if np.mean(scores_window) >= 30.0:
            print(f'\nEnvironment solved in {i_episode} episodes!\tAverage Score: {np.mean(scores_window):.2f}')
            torch.save(agent.actor_local.state_dict(), 'actor_checkpoint.pth')
            torch.save(agent.critic_local.state_dict(), 'critic_checkpoint.pth')
            break

    # Return reward scores
    return scores_list
```

Fig. 1. DDPG algorithm code

The algorithm accepts five parameters: the agent brain to use to solve the environment, the number of agents used, the number of episodes, the number of timesteps per episode and the quantity of episodes to print to the console, for algorithm progression updates. The algorithm starts by iterating over each episode, where it resets the environment, initializes the first state, sets all the scores to 0 and resets each agent.

Next, the model training begins by creating each agents action and uses it to interact with the environment, where a new experience is generated, stored and the Q-targets are updated after every 20 timesteps. Once the experience is generated and stored, the reward produced is added to the total and a new state is set. This process is repeated for the maximum number of timesteps specified, where the final timestep concludes the episode. Once the episode has finished, the scores are stored within a list and the average score is returned and checked for the average agent score goal of 30.0. If the score has been achieved, the models best parameters are saved, and the algorithm finishes with returning a full list of the average scores across all agents over each episode.

4. Results

The results achieved by the trained agents show an average score of +30.0 was achieved in 118 episodes over 100 consecutive episodes, combining all 20 agents scores, which is highlighted in Figure 2.

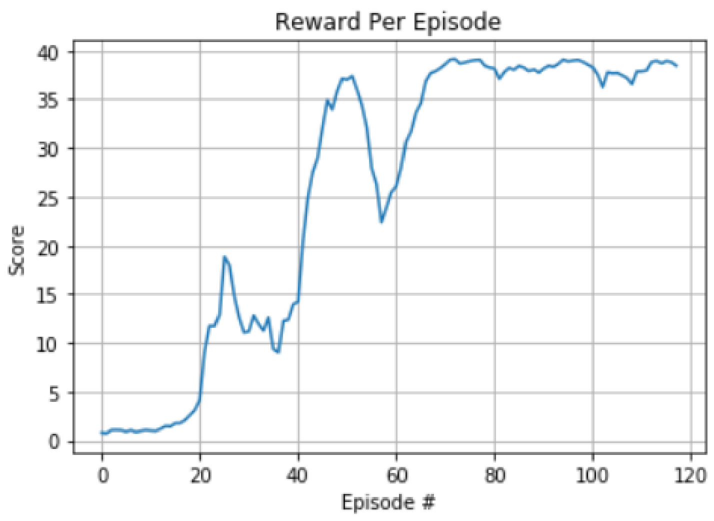


Fig. 2. Plot showing the average reward per episode, across all 20 agents

Looking at Figure 2, the models training is starts off low at first until the agents get a feel for utilising their double-jointed arm. Once familiar, the score continues to improve overtime before achieving peak performance by episode ~76.

5. Ideas for Future Work

To improve the results, the implementation of a Rainbow DQN could be implemented. Additionally, other algorithms could be considered such as: D4PG, A2C, A3C and PPO to determine if this algorithm is the most effective solution to this problem.