COMP3211 Final Project Report

AI play QWOP-like game
Tong Wai (202713565)

Introduction
In this project, I would like to build an AI for the game QWOP. However, QWOP is a closed source game. I have used a QWOP simulator in this project instead of the QWOP game. The simulator provides a similar physical environment and a simplified character model as shown in Figure 1.

Figure 1: The QWOP game (left) and the QWOP simulator (right)

QWOP is a game requiring player to control an athlete’s movement using Q, W, O, P keys in a 100-meter event. To win the game, player need to control the movement of athlete’s thighs and calves to move forwards. The game is hard for most of the human players to win. It will be interesting to see how AI will play this game. This project can help simulating the motion of bipedal robots in 2d space.

To build the AI, I have tried 2 algorithms, Q-learning and Deep Deterministic Policy Gradient (DDPG).

Set-up
Before the implementation of the AI, I need to build the QWOP simulator. I have found a QWOP simulator written in Python in the Internet, which can be found at https://web.stanford.edu/class/cs168/qwop.py.

I have done some refactoring to fit the AI implementation later. I have created 2 class from the original code.

One is Qwop which is used to define QWOP environment with method step, reset, render and isDead.

Another class is Viewer which controls the animation. It has a render function to generate an animation using matplotlib and save it in MP4 format.
**Approach**

I have used 2 reinforcement learning algorithms, Q-learning and DDPG in this project. 

**Q-learning**

Firstly, I have decided to use Q-learning. I followed the pseudocode shown in Figure 2 to implement the learning algorithm of the AI.

```
Initialize Q(s, a) arbitrarily
Repeat (for each episode):
  Initialize s
  Repeat (for each step of episode):
    Choose a from s using policy derived from Q (e-greedy)
    Take action a, observe r, s'
    Q(s, a) = Q(s, a) + α[r + γ max_a' Q(s', a') - Q(s, a)]
    s ← s'
  until s is terminal
```

**Figure 2: The Q-learning algorithm**

There are 8 actions which are Q ([0.5,0]), W([-0.5,0]), O ([0,0.5]), P ([0,-0.5]), idle ([0,0]), Q+O ([0.5,0.5]), W+O([-0.5,0.5]), Q+P ([0.5,-0.5]), W+P([-0.5,-0.5]). For the state, I have used relative position of 4 joints and hip to head and velocity of 6 points (4 joints, head and hip) which are all 2-d vector. Therefore, the state is a 22-dimension vector and it is continuous. Since Q-learning is commonly used in discrete state, I have discretized the state by rounding the state to 2 digits after the decimal point.

However, the state is still large after discretization. The optimization is too slow with a large space. As a result, I tried to use a better algorithm which is good in continuous state and action -- DDPG.

**Deep Deterministic Policy Gradient**

DDPG is an algorithm based on Deep Q-learning (DQN), Actor Critic and Deterministic Policy Gradient (DPG). DDPG improve DPG, inspired by DQN, to allow the algorithm to perform well in large continuous state and action space [1]. The pseudocode is shown in Figure 3.

```
Algorithm 1 DDPG algorithm
Randomly initialize critic network Q(s, a;θQ) and actor µ(s;θµ) with weights θQ and θµ.
Initialize target network Q' and µ' with weights θQ' ← θQ, θµ' ← θµ
Initialize replay buffer R
for episode = 1, M do
  Initialize a random process N for action exploration
  Receive initial observation state s1
  for t = 1, T do
    Select action a_t = µ(s_t;θµ) + N, 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_t, a_t, r_t, s_{t+1}) from R
    Set y_t = r_t + γ Q'(s_{t+1}, µ'(s_{t+1};θµ'))
    Update critic by minimizing the loss: J = 1/2 Σ(y_t - Q(s_t, a_t;θQ))^2
    Update the actor policy using the sampled policy gradient:
    ∇θµ J = 1/2 Σ∇θQ(s_t, a_t;θQ)|_{µ(s_t;θµ)} ∇θµ µ(s_t;θµ)
  end for
  Update the target networks:
  θQ' ← r θQ' + (1 - r)θQ
  θµ' ← r θµ' + (1 - r)θµ
end for
```

**Figure 3: The DDPG algorithm**
To implement this algorithm, I have referred to the DDPG tutorial in a reinforcement learning tutorial online, which is available at


I have modified the code to fit the QWOP environment that is built above. I have also changed some hyper parameters so that the algorithm could perform well in the environment.

The state is a 22-dimension vector and it is continuous which is the same as in Q-learning. However, since DDPG support continuous action space, the action consists of 2 valves that is continuous with range -1 to 1.

**Results**

The baseline is to ensure the runner not to fall after the maximum steps/actions; The oracle is to reach the goal in the shortest time or least steps/actions

**Q-learning**

The AI is trained using Q-learning for 6000 iterations. It cannot learn to pass 5m with 200 maximum steps after 6000 iterations.

**Deep Deterministic Policy Gradient**

The AI is trained using DDPG for 6000 iterations. It aims at reaching 100m. The AI pass about 10m after 2000 iterations, 50m after 2800 iterations (shown in Figure 5).

![Figure 4: Rewards graph for target 100m (x-axis: iteration; y-axis: rewards)](image)

It finally reaches the goal after 3400 iterations. The best result in the project is to finish the 100m dash in 131 steps/actions.
Evaluation

Q-learning is not good at training agent in controlling continuous state and action and large space. The AI need to trained using a substantial number of iterations to learn all Q-value in a large state space which is not efficient. Also, discretized action is also a problem here because some states will be ignored.

On the other hand, DDPG performs well in continuous state and action because it gives approximation functions to the continuous state and action. The runner can be trained to run towards the target. With more iterations of training, the agent should be able to run forever without falling. However, DDPG did not produce the optimal solution/policy in this project i.e. run using minimum steps/actions.

To solve this problem, here are some approaches:
1. Redefine the state so that the agent could learn more efficient.
2. Modify the hyper parameters in the network.
3. Adjust the reward function

The agent can learn based on not only position and velocity. According to [2], Gustav and Ryan have used position, angle, angular velocities, time spent, center of mass, distance traveled as the state. This could be better because the state become more specific.

Tuning hyper parameters in the network also help increasing the performance of the network. For example, the learning rate for actor and critic, the discount rate can be modified to get a better result.

Lastly, the agent would act differently when the reward function is changed. For example, the agent will not fall much when the penalty of falling is high. One way to optimize the speed is to set more points on acceleration and less penalty when falling. This may lead the agent to learn a faster movement.

Conclusion

QWOP is an interesting game. I have used the simulator to simulate the game environment because it is hard to build an AI on a close source game. Q-learning and Deep Deterministic Gradient Policy are use applied and DDPG performed the best. However, the AI performance is not perfect and needs to be improved.

Reference