Project 2: Robot arm control for deep soil mixing

Ding Man

November 18, 2020

1 Introduction

My friend is a geotechnical engineer specialised in deep soil mixing. I was curious about how a typical machine performs soil mixing on a particular region without touching on pipes, ground foundations and other possible obstacles. After having a conversation with her, the idea of designing an intelligent machine that uses deep reinforcement learning has occupied my mind.

1.1 Problem identification

Weak soil strength has always been a complicated challenge when it comes to executing excavation tasks. Deep soil mixing is an effective way to strengthen the soil at a particular region to ensure the soil stability during excavation, especially in Singapore where most of the ground formations are consist of marine clay with relatively weak strength. However, the condition underground is very intricate as there are ground foundations, pipes, other types of soil that the soil mixing machine must not touch on. These constraints might be on top, below, on the left, on the right of the targeted soil mixing region.

Soil mixing machines do not always perform well on the tasks as many of the times, they end up violating the constraints to some degrees, for example, damaging the water pipes (very common).

In this project, the objective is to train a robot to perform the task without violating the constraints. The problem is modelled as robot arm control.

2 Environment

2.1 General setting

As a typical machine uses a hollow auger to mix the soil with concrete, controlling the auger can be modelled as a robot arm with auger on the tip. The robot arm is made of two links and two joints. Each join is free to rotate. The robot arm is operating in a 2 dimensional space that can be modelled by the coordinate system (i.e. x, y).

The length of each segment of the arm is set as 100. As it is practical only to consider the area that is reachable by the robot arm, the limits of the coordinate system are set as 400 with the upper joint of the robot arm positioned in the centre (i.e. $x = 200, y = 200$).

The state of the arm is defined by the two angles θ_1, θ_2 as shown in figure 1. The action space is continuous. Each action made will result in changing of the two angles.

2.1.1 Target area

The target area is modelled as a square with location and width as parameters. However, constraints can be on top of the target area such as building foundation, or below the target such as pipes, or on the left/right such as other types of soil. It is useful to restrain the target area inside a bigger

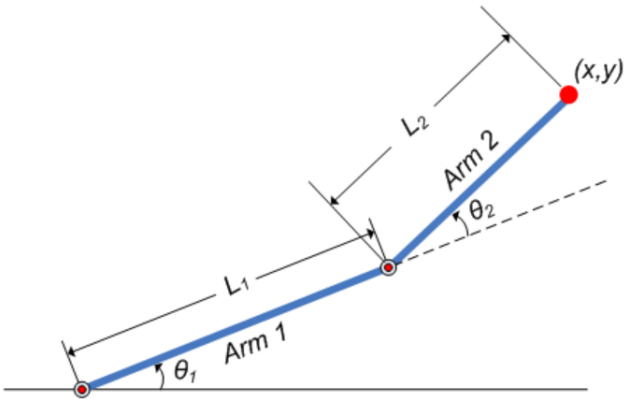


Figure 1: State of robot arm illustrate
Source: `https://www.mathworks.com`

area which takes into account all the constraints. As shown in figure 2, the green square represents the target area, while the yellow square that is slightly bigger than the green target is the area with constraints.

In this project, a worst-case scenario is considered: the target is surrounded by constraints from three sides, only one side is free to access. As the example in figure 2, this target has constraints on top, below and on the right of it (shown in yellow). As a result, the robot arm is only allowed to reach the target from the left side. Entering from any of the other three sides will result in a severe penalty. Even if the robot arm manages to find the correct side to access the target, it might accidentally end up in the yellow zone above or below the target area. It will also be penalized as this small yellow region serves as a safety margin between constraint and target.

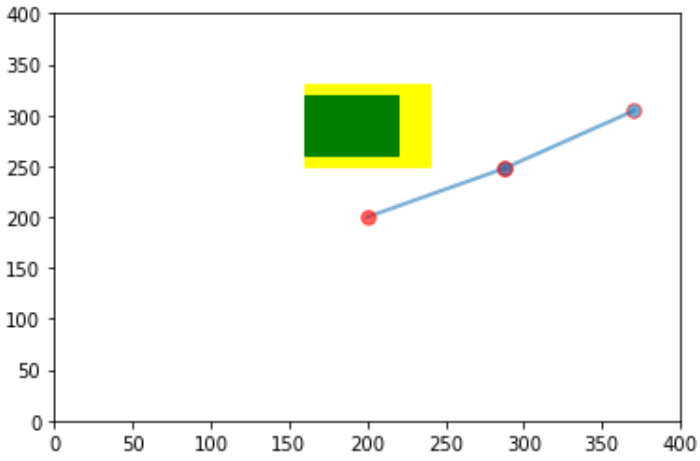


Figure 2: target area

2.2 Implementation of gym environment

To implement the environment, it is necessary to define three methods: `reset, step, render`. In this session, I will discuss how these methods were implemented.

2.2.1 Reset

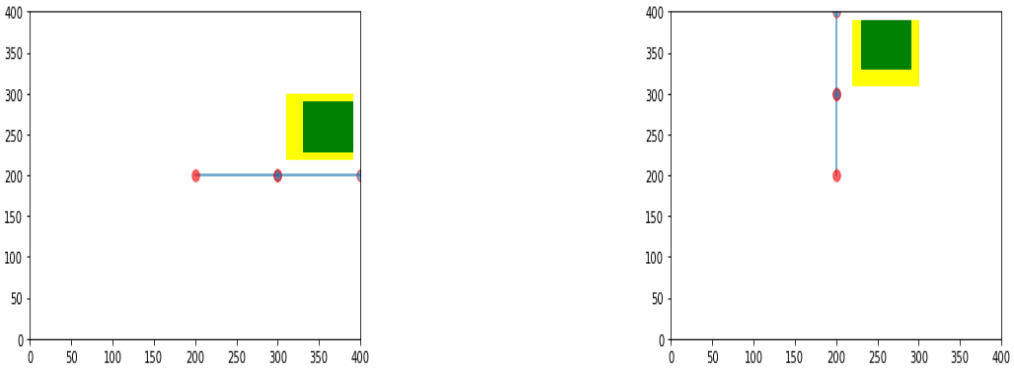
Reset method, as the name suggested, requires to generate a new map, set the global variables to their initial condition.

The variables that need to be reset are:

- set episode finish status to False
- set graph object to None
- generate a random initial state

Resetting the above variables is relatively straightforward while generating a random map requires more detailed examination.

The basic logic behind map generation is that the constrained area will be randomly generated first (if it is not pre-defined), followed by randomly choosing a side to place the target on. It is important to make sure that the randomly chosen side is indeed reachable by the robot arm. For example, when the constrained area is placed beyond the length of upper arm, the furthest side of the area is not reachable as shown in figure 3.



(a) right side not reachable (b) top side not reachable

Figure 3: Non-reachable sides illustration

2.2.2 Step

The `step` method takes in current action as input and returns new state, reward and episode status.

First of all, a `valid_action` method is implemented to ensure that the current action is within the action limit, i.e. $[-1, 1]$. The new state is updated by adding a fraction of the action value to the existing state. As in this project, the change of state should be approximately continuous; the fraction should be set to a small number such as 0.15.

Next, the reward calculation is the biggest challenge. Before going into the reward, it is necessary to be clear about the project objective and implement the method accordingly. In this case, the objective is to access the target area from the correct side without touching the constrained area. But how do we ensure that the robot arm is indeed entering the correct side? By checking whether the upper arm is intersecting with the correct side.

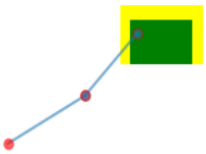
To check whether two line segments intersect, a `intersect` is implemented by simplifying an existing algorithm. `check_intersection` is implemented to check whether the upper arm is intersecting with each side of the constrained area. Based on the boolean output of the method, it is possible to define the different conditions that an action can lead to.

The conditions are listed below:

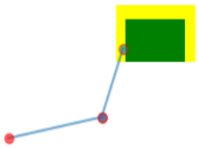
(1) Reach the constrained area: This condition consists of two possible scenarios as shown in figure 4. First scenario, the arm intersects with the wrong side; second scenario, the arm intersects with the correct side, but the tip of the arm is inside the constrained area instead of the target area.

(2) Penetrate through the constrained area: The tip of the arm does not land inside the constrained area nor the target area, but the upper arm goes through the area.

(3) Reach the target: to fulfil this condition, the robot arm must intersect with the correct side of the constrained area, and the tip of the arm should be inside the target area.



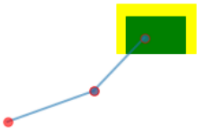
(a) scenario1: wrong side



(b) scenario2: inside constrained area

Figure 4: First condition illustrate

(a) Penetrate through the area



(b) Reach the target

Figure 5: Second and third conditions illustrate

(4) No contact with the constrained area: no intersection between the upper arm and any side of the constrained area.

Based on the above conditions, four types of reward are defined as below:

- Area reward: reward for action that leads to condition (1), i.e. violate the constraints.
- Goal reward: reward for action that leads to condition (3), i.e. reaching the target area.
- Wrong intercept reward: reward for action that leads to condition (2), i.e. penetrating through the constrained area.
- Distant reward: reward for action that leads to condition (4), i.e. far away from the constrained area.

The calculation for the first three types of rewards is straightforward. For the Distant reward, the idea of setting this reward is to motivate the agent to go closer to the goal even if the possibility of getting a negative reward is higher. So I decided to give a little reward if the agent is closer to the target area than it was before and vice versa.

2.2.3 Render

To implement the `render` method, it is practical to implement a `viewer` class to draw everything out.

The drawing is configured using `Matplotlib` package. `get_location` method is implemented to get the coordinates of the lower joint and tip of the arm. With the above information, two line segments can be drawn, which represent the lower and upper arm, respectively. The constrained area and target area are drawn using the Rectangle patch of yellow and green colors, respectively.

Lastly, in order to save multiple `Matplotlib` plots as gif, it is necessary to first convert the plot into 1-dimensional `numpy array` and then to reshape the `numpy array` into a 3-dimensional array that represents an RBG image.

The `render` method initiates a `viewer` class object if the object is None and calls the `render` method in the `viewer` class to render the drawing.

2.2.4 Unit tests

Unit tests are implemented to test the above methods. Two tests are conducted to check if the constrained area, target area are valid, i.e. reachable by the robot arm. The other tests are conducted

to test the `step` method so as to ensure that the conditions are corresponded to the correct reward and episode status.

3 Deep Deterministic Policy Gradient (DDPG)

In this project , deep deterministic policy gradient is chosen to train the agent as DDPG is a popular model-free off-policy algorithm for learning continuous actions. DDPG is a combination of DPG and DQN which uses two target networks and experience replay.

This algorithm is implemented using keras since keras is a high level API that facilitates the process. I adopted and made some modifications to an existing implementation from na Youtube video. The existing code that I referenced on is an direct implementation of the algorithm below:

Algorithm 1 DDPG algorithm

Randomly initialize critic network $Q(s, a|\theta^Q)$ and actor $\mu(s|\theta^\mu)$ with weights θ^Q and θ^μ.
Initialize target network Q' and μ' with weights $\theta^{Q'} \leftarrow \theta^Q$, $\theta^{\mu'} \leftarrow \theta^\mu$
Initialize replay buffer R
for episode = 1, M **do**
  Initialize a random process $\mathcal{N}$ for action exploration
  Receive initial observation state s_1
  for t = 1, T **do**
    Select action $a_t = \mu(s_t|\theta^\mu) + \mathcal{N}_t$ 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_i, a_i, r_i, s_{i+1}) from R
    Set $y_i = r_i + \gamma Q'(s_{i+1}, \mu'(s_{i+1}|\theta^{\mu'})|\theta^{Q'})$
    Update critic by minimizing the loss: $L = \frac{1}{N}\sum_i (y_i - Q(s_i, a_i|\theta^Q))^2$
    Update the actor policy using the sampled policy gradient:

$$\nabla_{\theta^\mu} J \approx \frac{1}{N} \sum_i \nabla_a Q(s, a|\theta^Q)|_{s=s_i, a=\mu(s_i)} \nabla_{\theta^\mu} \mu(s|\theta^\mu)|_{s_i}$$

    Update the target networks:

$$\theta^{Q'} \leftarrow \tau\theta^Q + (1-\tau)\theta^{Q'}$$
$$\theta^{\mu'} \leftarrow \tau\theta^\mu + (1-\tau)\theta^{\mu'}$$

  end for
end for

Figure 6: DDPG algorithm
Source: `https://arxiv.org/abs/1509.02971`

3.1 Implementation of basic structure

Three classes are implemented as the structure of DDPG: `replay buffer`, `actor network`, `critic network`. The structure of `actor network` and `critic network` are similar: two fully connected layers with ReLU activation. The only difference of the structure is the last layer: `actor network` uses `tanh` while `critic network` does not require activation. `tanh` is used to ensure the output of the network does not blow up. `replay buffer` class stores state, action, reward, next state and episode status inside numpy arrays.

3.2 Implementation of Agent class

Initializing the parameters/hyperparameters such as learning rate, environment, batch size in the agent class. Initiazing four networks: actor, target actor, critic, target critic and compile the above models using `Adam` optimizer.

Random noise is added when generating the action to encourage exploration.

The critic loss is calculated using mean squared error of $y - Q(s, a)$ where y is the expected return as seen by the target network, and $Q(s, a)$ is action value predicted by the critic network. The actor loss is calculated using the mean of the value given by the critic network for the actions taken by the actor network.

During training, we want to maximize actor loss and minimize critic loss. As a result, the actor network is updated in a way to generate actions that obtain the maximum predicted value as seen by the critic network, for a given state.

4 Train the model

The training process is monitored using `Tensorboard` by capturing the moving average score, moving average success rate, moving average steps, critic loss and actor loss.

4.1 Experiment 1: small constrained and target area

For the first experiment, I set the width of constrained and target area to be 40, 20 respectively. The reward for reaching the goal was 1.0 while the reward for violating the constraints was -0.5. The model failed to converge mainly because the agent did not succeed once in a few hundred episodes. Hence, the agent could not learn any useful experience on how to reach the goal.

To increase the random success rate of the task, I reset the negative reward to zero hoping by relaxing the constraints, the agent would be able to get to the goal easily. Although the success rate indeed increases, the model does not converge as well as the actor loss decreased and critic loss increased as training goes on.

As I realized setting the negative reward is not going to fulfil the objective, I decided to increase the width of the two areas to increase the success rate through random actions.

4.2 Experiment 2: bigger areas

In this experiment, the widths of constrained and target area are set to be 80, 60 respectively. There were still two types of reward: reaching the target with 1.0 while reaching the constrained area -0.5.

However, when I checked the replay of success episodes, the agent did violate the constrains many times before reaching the target which did not fulfil the objective of this project.

Hence, I decided to reduce the maximum steps that the agent can have for each episode and increase the negative reward from -0.5 to -1.0. In the meanwhile, adjust the batch size from 64 to 32.

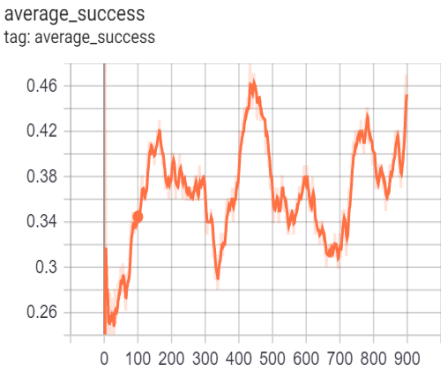
4.3 Experiment 3: reduce maximum steps, more punishment

This time, the performance has increased a lot compared to the previous experiment as shown in figure 7. The average success rate has reached 0.46 between 400-500 training episodes. The actor loss has indeed increased during training.

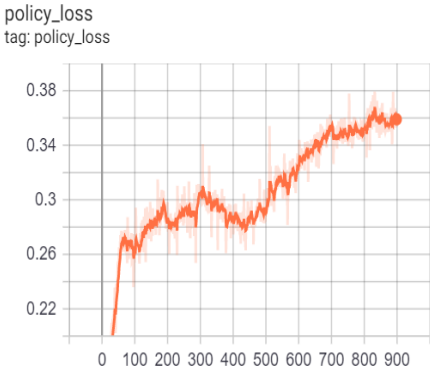
However, when analyzing the success episodes, the agent sometimes went through the constrained area without being penalized. Although the success rate has increased significantly during training, the objective has yet to be met. This was when I decided to add another reward to address the condition when the agent's upper arm goes through the constrained area.

4.4 Experiment 4: add another penalty reward

This is the setting I have described in session 2.2.2. I set the goal reward, constrained area reward and wrong intercept reward to be 1.0, -1.0, -0.8. The average success rate has decreased compared to the previous experiment as shown in figure 8.

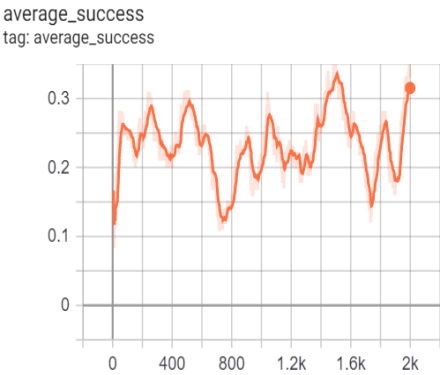


(a) average success rate for latest 100 episodes

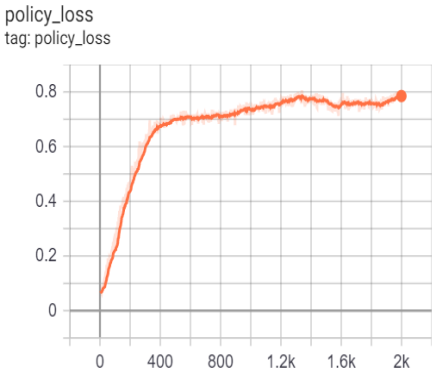


(b) policy loss

Figure 7: Experiment 3 training



(a) average success rate for latest 100 episodes



(b) policy loss

Figure 8: Experiment 4 training

4.5 Experiment 5: accumulate goal rewards

Next, I was curious to see if accumulate goal rewards would make any difference to the current performance. Hence, instead of ending the episode once reaching the goal, the agent has to reach the goal 20 times before ending one episode. I was expecting the agent to establish a memory of how to access the goal from one side of the constrained area.

The result did confirm my expectation, but quite in the opposite way, the agent has stuck in a sub-optimal policy by performing the same actions during rest of the training episodes.

5 Discussion

The model does not achieve the project objective as the maximum success rate during training is around 0.35 and the success rate during test is only 0.2. I have summarized the possible reasons as below:

5.1 Reward setting

Based on the previous experiments, the reward setting is directly related to agent's performance. Not only the value of the reward but also the condition of ending an episode decide how agent learn during training. I have tried various values as well as the ending condition, but the agent still performed badly during training.

However, I think reset the reward criteria may be a way to improve the model.

5.2 DDPG

I have read several papers regarding the issues with DDPG. The agent might end up in sub-optimal policy if the agent fails to succeed during the first few episodes. However, I have yet to figure a way to approach this problem due to limited understanding of DDPG or deep learning in general.

5.3 Neural network

Another challenge I faced was to decide when to abort a training process. Although I have configured to save the checkpoint when the current score is better than the average score, the performance usually goes up and down while the actor loss increases and critic loss decreases as desired. Hence, I had struggled to identify a suitable point to abort training process.

One possible way to improve is adding the gradients to Tensorboard. By observing the changes in gradients, it might be possible to identify the point.

5.4 Discretize action space

A possible way to improve the current model is to discretize the continuous action space and use the other algorithms such as DPG or SAC.

In my case, after trying out DDPG on some classic problems such as inverted pendulum, I was confident that DDPG would render a good result for this project as well. However, it did not work out in the end.

6 Self evaluation

I have learnt a lot from this project. First of all, I simplified a real-world problem into a simple environment. Secondly, I am able to implement the environment on my own from scratch using open AI gym and tested the environment using unit tests(I was scared of implementing my own environment a month ago while working on project 1). Thirdly, I have gained a deeper understanding of deep reinforcement learning and deep learning and learnt to use keras to implement neural networks (although with online reference). Lastly, the hands-on experience with Tensorboard was very exciting as well as it is such a powerful tool for monitoring training process.

However, there are still plenty room for improvements. Due to my master research, I did not have enough time to dive deeper into DDPG to figure out a way to improve performance. Also, I failed to explore other algorithms which might render good performance in my case.

Thanks to this project, I am more confident to embark on reinforcement learning as I have acquired with the basic skills to work on a problem using reinforcement learning. I will continue to explore the opportunities to apply the knowledge I have learnt in this project.

7 References

1 DDPG code implementation reference
`https://www.youtube.com/watch?v=4jh32CvwKYw&t=2117s`

2 DDPG paper
`https://arxiv.org/abs/1509.02971`