PIPE TRAVERSAL BOT

Vivek Agarwal
20095126
Electronics Engineering
Indian Insitute of Technology (BHU)
Varanasi
vivek.agarwal.ece20@itbhu.ac.in

Gujulla Leel Srini Rohan
20095041
Electronics Engineering
Indian Insitute of Technology (BHU)
Varanasi
gujullal.srinirohan.ece20@itbhu.ac.in

Aryaman Gupta
20095015
Electronics Engineering
Indian Insitute of Technology (BHU)
Varanasi
aryaman.gupta.ece20@itbhu.ac.in

May 9, 2022

ABSTRACT

The challenges involved in cleaning and maintaining Pipes System in various industries that actually require high human workload, health risk and much time consuming. Hence there is much need for such autonomous robot, which can be deployed in various piplines, to ensure robustness and travel to the pipe where it is difficult for a human to reach. Hence, our work is aimed to train the robot to look for the defect and sanitising pipelines which saves money, time and labor effort.

***K*eywords** Quaruped · Reinforcement Learning · DDPG

1 INTRODUCTION

1.1 Problem Statement

The aim of this project is to get the Quaruped Bot to walk to a target in a specified direction (positive x-axis) in a plane world with no obstacles. We set the target distance to 1 metres.The robot has 4 legs, 4 revolute joints per leg, tabulating to a total of 16 joints.

The motivation behind this robot comes from the purpose of cleaning and navigating through pipes. Small robots like these are capable of reaching areas like small vents and holes and they can also navigate autonomously through various turns and size of pipes. The multiple legs and joints give the robot a lot agility and flexibility in navigating difficult terrains. In this example, we start simple with an infinite flat plane and no obstructions.

1.2 Existing Methods

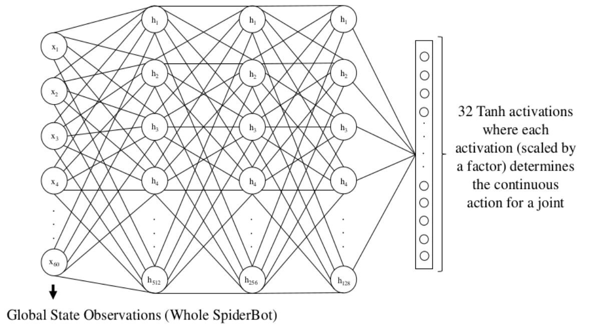
There are a few approaches one can conventionally take to make the Quadruped walk. The most conventional way would be the ”classical robotics” approach, which does not involve reinforcement learning. Unlike the end-to-end method that deep learning offers, a classical robotics approach would use low-level modularity in their implementation. This would make use of optimal control theory to control the actuation of the robot legs. Making a robot walk ’classically’ could also use dynamics calculations while requiring good understanding of the mechanical system. Such a classical approach would usually use sensor data and many steps of filtering and data processing to tell the robot what to do. Some processes even require complex inverse kinematics calculations. A cool example of this would be the Spot robot from Boston Dynamics that uses a wide range of smart sensors and modules to do specific tasks like climbing stairs or navigating an uneven terrain. To summarise, there are many modules and steps that work together to make a decision on actuation. The advantage of such a method is that they are generally reliable. As the designers ground the robot mechanics on theory, it would be very likely that they would be able to make our SpiderBot walk with enough work.

The issue with the classical approach however, is that many aspects of the system are essentially hardcoded. Such hard coded logic is extremely cumbersome, difficult to optimise and untenable in terms of scalability to more complex actions in complex environments. Moreover, if any link is damaged, the mechanics-dependent movement of the robot may not allow the robot to recover well unless a lot of tedious work is done to prepare for such scenarios. With this, we come to the next branch of algorithms, into the dark yet wonderful field of reinforcement learning (RL). To put it crudely, RL algorithms don't have to know any rules or strategy of the walking mechanism. There is no hardcoded or modular aspect that is commonly present in classical robotics methods. Being an end-to-end method, an RL agent simply needs to have an understanding of its state, an interface to actuate the joints and a feedback of reward at every time step. It does not need to know what happens with actuation or go through a series of complicated modules.

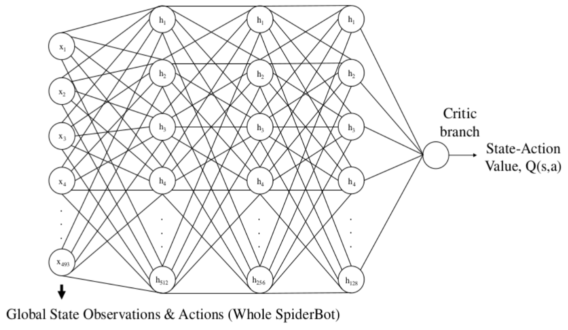
2 METHODOLOGY

2.1 Algorithm - DDPG

DDPG algorithm is founded on the deterministic policy gradient theorem and tries to learn a deterministic target policy. This approach uses both an actor and critic, but with a separate network for both. The actor network (Figure a) takes in a global input state of the spider bot and outputs an action in a continuous space for all 16 joints. The output comes from a hyperbolic tangent activation, which can be scaled a larger range. The critic (Figure b) outputs a Q value based on the same global input state and the action. It uses experience replay and target networks for the actor and critic. The loss of the critic is captured by taking the TD error, using both the target critic and critic. The actor network is updated using the deterministic policy gradient update and the target actor output. A soft update of = 0.005 is used to update the weights of both target networks. The motivation to use this algorithm came from the fact that we can implement actions in the continuous space, which allows for the SpiderBot to move in a smooth manner.


x_1
x_2
x_3
x_4
x_{60}
h_1
h_2
h_3
h_4
h_{512}
h_{256}
h_{128}
32 Tanh activations where each activation (scaled by a factor) determines the continuous action for a joint
Global State Observations (Whole SpiderBot)

(a) Actor Network



(b) Critic Network

2.2 RL Cast

It is extremely important to cast the RL framework onto the problem statement effectively. This refers to the design of the state space representation, action space and reward structure. In this project, the state space is represented by a vector of scalar values for the base link (centre square region of SpiderBot) and a joint as shown below. Many of the state values are taken with respect to the centre of mass (COM) in world coordinates.

State Representation for Base Link:

- Position of COM (x, y, z)
- Orientation in Quaternion $(a, b\mathbf{i}, c\mathbf{j}, d\mathbf{k})$
- Linear Velocity of COM (v_x, v_y, v_z)
- Angular Velocity $(\omega_x, \omega_y, \omega_z)$

State Representation for each Joint:

- Joint Position, θ
- Joint Angular Velocity, ω
- Position of local COM (x, y, z)
- Local Orientation in Quaternion $(a, b\mathbf{i}, c\mathbf{j}, d\mathbf{k})$
- Local Linear Velocity of COM (v_x, v_y, v_z)
- Local Angular Velocity $(\omega_x, \omega_y, \omega_z)$

The state of the base link can be represented by a vector of size 13 while each leg can be represented by a vector of size 15. In DDPG, where the critic, agent or actor takes in the global state observations, the input size of the neural network is 13 (base link) + 15*8 (8 legs) = 493. Note that as neural networks are used, the state space can be made continuous. Moving on to the action space, we have the ability to set actions for each one of the 16 joints. We initially considered setting a target angular displacement, , but our initial observations showed us that none of the joints attached to the base were moving at all. As DDPG algorithm allowed for a continuous action space, we scaled the final hyperbolic tangent activation for each joint by 10 to give a continuous action space from -10 rad/s to 10 rad/s.

The reward structure is a crucial part of the RL cast as its design can encourage or discourage the SpiderBot from taking certain actions. We took a multifaceted approach for the reward structure by including various components that affect the reward. There are 8 possible rewards the SpiderBot can get at every time step. The first is a reward proportional to the forward velocity of the spider with the second being a negative reward that gets smaller the closer SpiderBot gets to the target. The values are set to 500 250 respectively. The aim of these two rewards is to encourage the robot to move forward in the correct direction. The thirds and fourth are negative rewards that are proportional to the sideways velocity and sideways distance from the expected line of walk. They are set to -500 -250 respectively. Naturally, the negative rewards mean that we are trying to dissuade the SpiderBot from moving sideways. The fifth reward is the time step penalty (set to -1) that has an increasingly negative reward proportional to the time of the episode thus far with the aim of pressuring the SpiderBot to move. The sixth to eight rewards come from 3 specific termination conditions. It received a positive reward when the goal is reached (500) and a negative reward for falling down (-500) or moving out of a specified range (-500) i.e. 1m sideways and 1m backwards. We had initially imposed a punishment if the SpiderBot had any rotation. However we observed that the spider did not want to move at all in the end, therefore we went without it.

2.3 Environment Training Process

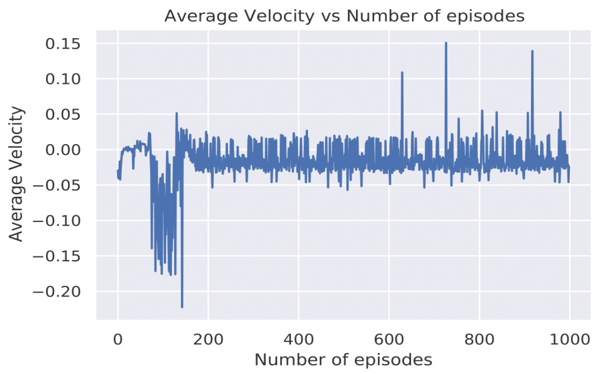
The creation of the environment involves a few steps, starting with the URDF of the SpiderBot, which contains information of the links, joints STL files. This URDF is then uploaded to a physics engine in OpenGL, where we fully designed a gym environment to interact with the virtual model. The PyBullet API is adopted to aid us in interfacing with the SpiderBot. We also use the API to limit the range of all joints from -60 $\circ$ to 60 $\circ$, where 0 $\circ$ is the starting position for each joint based on the URDF. The interfacing allows us to record state observations, set target velocities at joints and gather information to determine rewards. For instance, a robot is deemed to have fallen if any second link (where the first link is defined as the link normally in contact with the ground) has contact with the ground. Additional to the 3 termination conditions, an additional termination condition is added to limit an episode to 10 minutes in real time. The final aspect of the environment is the length of time given to allow for the SpiderBot to execute the action. If the time step is too low, the joints will barely be able to move due to the lack of time. A time step too large could mean that the joints have reached their limits, which would not be productive to movement and overall learning. Hence, we tuned the time step to about 0.1s when comparing across the algorithms.

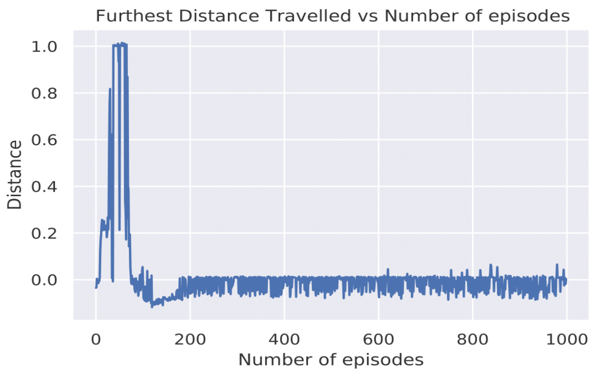
For the training itself, we ran ethe algorithm in TensorFlow 2, accelerated by an NVIDIA GeForce GTX 1060 GPU. Due to the lack of time and computational speed, we had to limit each run to 1000 episodes. A reasonable discount factor of 0.9 was used. A replay memory of 1 million with a batch size of 256 was used for DDPG algorithm. All neural networks used a total of 3 fully connected hidden layers with a decoder-like structure. An L2 regulariser and batch normalisation is used at every layer in all networks. All non-output neurons use a Rectified Linear Unit (ReLU) activation function in this project. To optimise the learning of the neural networks, an Adam optimiser is adopted. During training, all relevant information is logged and subsequently post-processed. The subsequent section will present the results and reward analysis.

3 RESULTS

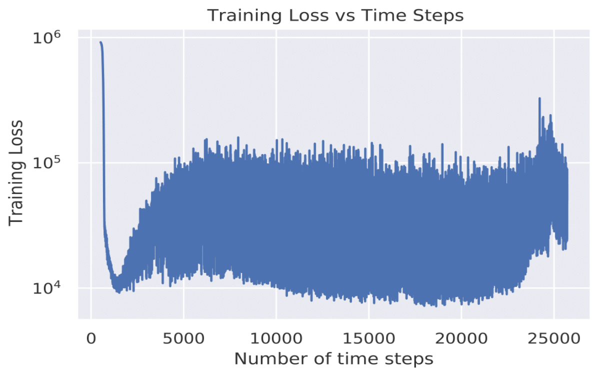
The plots in Figure shows that with DDPG approache, we actually met the target distance of 1m a few times. The DDPG approach achieved successes rather early on, however it was not able to maintain that for the remainder of the episodes.

Other than the best distance metric, another important measure is the average velocity of the SpiderBot during each episode. This is because the SpiderBot can theoretically achieve a positive furthest distance travelled but still be moving backwards most of the time. The plot in Figure for average velocity appear to be quite correlated with the furthest distance plots in Figure. A close look at Figure show that the SpiderBot has an impressively high forward velocity of 0.16 m/s in the cases where the DDPG approach reach the target. This shows that in those instances, the SpiderBot was able to walk forward to the goal without moving far sideways.





we take a look at the training loss of the approache.The loss dropped quickly in the early part of training it its minimum and then it started to increase afterwards. Interestingly, the region of lowest loss corresponds to the times the SpiderBot reached the goal. This is a sign that perhaps the SpiderBot is indeed learning to walk in the specified direction.



4 CONCLUSION

The most complex part of the training process is managing the hyperparameters. With the multiple approaches that use many features like experience replay, target networks, hybrid networks, multiple agents, the hyperparameter space exponentially blows up. We don't have nearly enough time or computational resources to run an extensive hyperparameter search. To manage this, we manually tuned hyperparameters we found more important and adjusted them based on how the SpiderBot was behaving in the short term.

One good consequence of going through a very challenging project is that I get to learn a lot. On the technical side, I learnt to use TensorFlow 2 for programming neural networks, especially to manually apply gradients using a gradient tape. I also learnt the fascinating theory and algorithms for deep RL. It is also my first time working with a physics engine, where it was satisfying to see my SpiderBot design come to life. However, more than the several technical implementation lessons of the project along with the findings from it, I learnt many other valuable lessons that I will take to my future projects. The first lesson being that simply throwing a fancy and complicated algorithm at a problem will not fix it. I must carefully analyse where the algorithm has worked before and hasn't and even consider using a simpler approach first. Second, it is important to manage expectations and resources for a project. Taking on a difficult project could mean that there would be many bouts of failure before getting it right. It also requires careful planning and it is important to consider the allocation of computational resources beforehand. Finally, it is extremely important to think and rethink the RL cast for RL problems. I can use a good model, but it is entirely dependent on the way I design my state actions space, along with the reward structure to achieve success. I am very satisfied to have learnt so much from this project and I am glad this report is evidence of that. No doubt that I will take these ideas to my next project in RL.

4.1 Future Work

A good idea for further work would be to implement a form of distributed learning by using multiprocessing, so that the agent(s) can learn faster in parallel. Additionally it would be wise to reduce the state representation. Not all the local angular velocity and location of COM information on the SpiderBot would be useful in the case of global observation as the base link angular velocity and COM would be enough to learn the location and orientation of the SpiderBot. I would also consider using less joints per leg to reduce the complexity of the action space for one leg. One should also consider simpler algorithms to first see if they can solve the problem statement. Finally, a robust hyperparameter search is necessary to tune the performance of the learning agents. All in all, I am very pleased with the work we put into this project and glad that we succeeded our goals with a fully-trained DDPG.

References

[1] https://www.mathworks.com/help/reinforcement-learning/ug/quadruped-robot-locomotion-using-ddpg-gent.html

[2] https://github.com/arijitnoobstar/SpiderBot$_{D}eepRL/$