Reinforcement Learning in Robotics, COMP3211, Introduction to A.I. Project

Thomas Lew
HKUST
Student ID: 20414520
Exchange student from EPFL

***Abstract* - In this project report, we describe the COMP3211 course's project on Reinforcement Learning, which mostly consisted of implementing an A.I. to achieve real-time walking of a two-legged robot. A simulation of the robotic platform was built to train the learning agent, which was then implemented on a real-world robot. This paper also includes the difficulties encountered, assumptions and work to be done to improve the quality of the walking process. The most recent version of the code and instructions are released at: https://github.com/thomasjlew/RL_walking. A video of the walking robot can be also found at https://youtu.be/Hwkp4MxLccQ.**

I. Introduction

Robotics is a multi-disciplinary field and the design of a robotics platform focuses on two main aspects: control and sensing. Traditional methods have been developed to achieve these two aspects. In modern robotics, sensing and state estimation is usually dealt with Visual-Inertial Systems by combining different sensors and fusing their information in a Kalman Filter [1]. On the other hand, model-based closed-loop control can achieve optimal efficiency [2] in many cases but assumes a good knowledge of the system's state, of the model and of the robot's physical characteristics.

On the other hand, the advancements in Artificial Intelligence (A.I.) and Reinforcement Learning (RL) algorithms have proved to be efficient at dealing with many of these problems [3] with completely different approaches, implementation issues and advantages.

In this report, we use a Q-learning RL algorithm to solve the traditional control problem of a cart-pole. Also, we train an agent to walk on a simulation by using Evolution Strategies (ES) [4], before implementing the learnt model into a real-world robot to compare the result between the simulation and real-world application.

II. Related Works

Reinforcement Learning algorithms have been widely used in robotics applications. Many implementations managed to obtain working systems. For example, [5] trained an inverted helicopter using RL techniques and managed to successfully stabilize it. Also, good walking performances were achieved on a 2D simulation using a simple RL algorithm [6]. Other works [7] also implemented 3D walking abilities successfully.

Intelligent RL algorithms have successfully managed to replace traditional control systems and state estimation. Different approaches can be used, such as keeping a closed-loop control and learning the state representation using RL [8], or directly computing output commands from raw input data, such as [9] which used CNNs to derive the manipulator's motor torques directly from raw camera and joint angles data.

This project also tries to implement a working agent to achieve successful walking. The simulation and training environment which was used is OpenAI Gym, which offers many different environments with a standardized framework. This project goes a step further by testing the algorithm on a real-world platform. In this work, motor torques, represented as a change in the servomotor angles, are directly computed by the agent from the current robot's state, which is still derived from a simple model using real-time approximated data.

III. Overview of Simulation Environments

1. OpenAI Gym

For this project, we chose to use OpenAI Gym simulation environment in python. It is a simple toolkit which offers many simulation environments with a standardized framework to compare efficiently and easily different RL algorithms. Its main use is to return a reward described by the selected simulation, given an action set from the agent. This way, it is very simple to compare different RL algorithms since the testing environments are standardized.

2. Cartpole

The first simulation environment which was used in this project consists of the typical cart-pole control problem. A free box is moved linearly with an attached pole standing upwards with a revolute joint linking the two objects. Since the problem is very standardized and is explained in details in every control theory textbook, we chose it to test and implement a Q-learning algorithm.

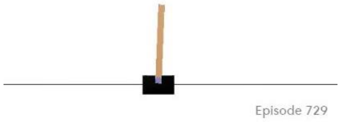


Fig. 1. Cart-pole simulation environment

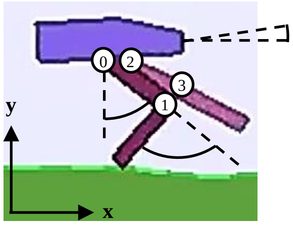
The state consists of four variables: the horizontal position of the box, the angle of the pole and their derivatives. In the implementation of the Q-learning algorithm, only the angle and its derivative are used and are enough to achieve a stable and immobile system. The action is either pushing the box to the left or the right.

In OpenAI Gym environment, the agent receives a reward for each time step remained upwards. The episode ends when the box is either too far from the middle, or when the pole is no longer vertical. Please refer to OpenAI Gym official website for more details.

3. Walking Robot

The walking robot simulation consists of a main body (the “hull”) on top of two legs consisting of four revolution joints. It also embeds a LIDAR returning ten distances measurements in front of the robot, which are useful in the hardcore version of the environment with obstacles. We will omit this information in this work.

The state consists of the hull’s angle, angular velocity, linear velocities, of the four joints angles and angular velocity, of two binary values depending on whether the bottom of the legs touch the ground or not and of ten LIDAR measurements (omitted). All angles are defined as positive in the counter-clockwise direction, as described in the Box2D framework.



$$joint[0] \approx {}^{\pi}/_{3}$$
$$joint[1] \approx -{}^{\pi}/_{2}$$

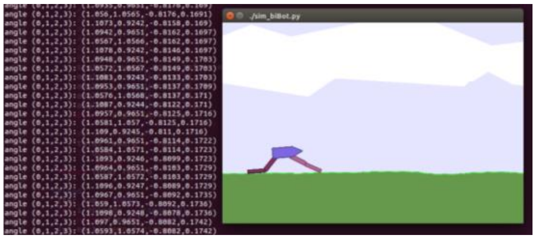
Fig. 2. Walking robot simulation: Coordinate system and conventions

A positive reward is given when the robot advances in the right direction and the episode ends with a negative reward when the hull touch the ground. Note that since an optimal agent should minimize the energy consumption, a negative reward is given each time that torque is applied on the joints.

It is interesting to note that for this simulation environment, different walking strategies can be obtained from RL algorithms. For example, an agent could constantly jump on one leg, or use one knee to stabilize itself while using the other leg to push itself forward or it can also run very fast with both legs.

In our project, we try to obtain an agent using a slow-paced walking pattern, in order to have a more stable walking process which is needed when implementing the A.I. on a real-world robot which input data and state estimation is corrupted by noise, which motor control is not optimal and constant and which variables and model are not generally very precise. Thus, it is easier to obtain a working prototype with an easier strategy than to adapt an optimal but almost unstable strategy such as a fast running pattern.

A modified simulation of our robot which takes into account the limited movements of the motors and the joints positions relatively to the main body was created for more accuracy. However, the learning agent was trained on the original simulation provided by OpenAI Gym with some minor state modifications, such as the LIDAR and ground-touching sensing removals, before being tested on the custom made simulation and then transferred to the robot.



```
angle (0,1,2,3): (1.056,1.0565,-0.8176,0.1691)
angle (0,1,2,3): (1.1073,0.9242,-0.8158,0.169)
angle (0,1,2,3): (1.0942,0.9651,-0.8162,0.1697)
angle (0,1,2,3): (1.0567,1.0566,-0.8162,0.1697)
angle (0,1,2,3): (1.1078,0.9242,-0.8146,0.1697)
angle (0,1,2,3): (1.0948,0.9651,-0.8149,0.1703)
angle (0,1,2,3): (1.0572,1.0567,-0.8149,0.1703)
angle (0,1,2,3): (1.1083,0.9243,-0.8133,0.1703)
angle (0,1,2,3): (1.0953,0.9651,-0.8137,0.1709)
angle (0,1,2,3): (1.0576,1.0568,-0.8137,0.171)
angle (0,1,2,3): (1.1087,0.9244,-0.8122,0.171)
angle (0,1,2,3): (1.0957,0.9651,-0.8125,0.1716)
angle (0,1,2,3): (1.0581,1.057,-0.8125,0.1716)
angle (0,1,2,3): (1.109,0.9245,-0.811,0.1716)
angle (0,1,2,3): (1.0961,0.9651,-0.8114,0.1722)
angle (0,1,2,3): (1.0584,1.0571,-0.8114,0.1723)
angle (0,1,2,3): (1.1093,0.9246,-0.8099,0.1723)
angle (0,1,2,3): (1.0964,0.9651,-0.8103,0.1729)
angle (0,1,2,3): (1.0587,1.0572,-0.8103,0.1729)
angle (0,1,2,3): (1.1096,0.9247,-0.8089,0.1729)
angle (0,1,2,3): (1.0967,0.9651,-0.8092,0.1735)
angle (0,1,2,3): (1.059,1.0573,-0.8092,0.1736)
angle (0,1,2,3): (1.1098,0.9248,-0.8078,0.1736)
angle (0,1,2,3): (1.097,0.9651,-0.8082,0.1742)
angle (0,1,2,3): (1.0593,1.0574,-0.8082,0.1742)
```

Fig. 3. Modified Walking robot simulation with angles output. It is currently in a blocked position, which often occurs when training the agent on this simulation.

The reason to use the original simulation for training is that the positions of the joints at the same point on the hull is intrinsically more unstable which favors innovative strategies and helps to a faster convergence of the algorithm. When training our agent on the modified simulation, one can observe that the robot adopts an immobile behavior and reaches a local maximum although it doesn’t move, which directly caused by the very stable nature of two legs separated by a distance.

We can assume that a working agent on the unstable original simulation would be able to walk on the modified simulation matching our real-world built robot. Hence, we trained the agent on the first simulation with less data in the state, such as with the lidar removed and without the knowledge of the legs’ contact to the ground. This last approximation is a major issue in the realization of an optimal agent, since the A.I. doesn’t know if it is currently touching the ground or not. In a future version of the robot, this information will be known by adding buttons at the end of each leg.

IV. A.I. Algorithms

1. Q-Learning Algorithm

In RL algorithms, the goal is to obtain an optimal agent which obtains the highest reward. The A.I. is trained on a simulation environment, which in our case is OpenAI Gym, and tries multiple possible actions in each episode. An episode is defined as the sequence of actions and states executed before reaching an end-state.

The Q-learning algorithm consists of computing the values of each possible states in order to later select the most optimal one, which is the one which leads to the highest expected reward at the end of the episode.

For each episode, a Q-value ***Q(s, a)*** is defined as the expected reward when choosing an action ***a*** at a state ***s***. It consists a sum of expected rewards of the next states pondered by the respective probabilities of reaching this state by choosing the action ***a***:

$$Q_{k+1}(s,a) \leftarrow \sum_{s'} T(s,a,s') \left[R(s,a,s') + \gamma \max_{a'} Q_k(s',a') \right]$$

In each iteration of an episode, a sample is computed as the sum of the immediate reward ***R(s, a, s’)*** from action ***s*** to ***s’*** by choosing action ***a*** and maximum ***Q***-value of next states amongst all actions ***a’***. The discount factor ***γ*** ensures that recent rewards have more value than later ones.

$$sample = R(s,a,s') + \gamma \max_{a'} Q(s',a')$$

Hence, with a factor alpha …, we can update our table of ***Q***-values from the returned sample:

$$Q(s,a) \leftarrow (1-\alpha)Q(s,a) + (\alpha)\,[sample]$$

It is important to note that as the number of possible actions a’ increases, the dimensions of the table of Q-values increases as well. This is problematic in many cases as we can shown in the results.

Also, it is necessary to discretize the actions-space in order to implement this RL algorithm. Hence, for the Cartpole, we only use 6 different angles linearly mapped from the model to construct the states s. Also, the number of actions is very low (apply torque to the left or right) which greatly simplifies the algorithm and number of states to explore.

2. *ES Algorithm [4]*

It is important to note that this algorithm doesn’t belong to the Reinforcement Learning category of intelligent learning agents.

The Evolution Strategies algorithm consists of optimizing a function (in our case, the reward) by successfully changing the parameters of our policy. For example, to minimize a quadratic function of three parameters,

$$f(x) = (x - solution)^2 \ \in \ R^3$$

we successfully change randomly the three parameters of *x* by adding Gaussian noise and compute the cost function for each sample. After this step, we update the parameters of *x* according to the most successful of the samples, by taking a pondered average of these items.

We repeat these steps (which could be computed in parallel if necessary) until necessary. It is interesting to note that it is very easy to decide of the noise involved in the parameters modification for each step. Also, it is not necessary to know exactly the function which can be treated as a black box.

For this reason, this algorithm can be very useful in intelligent agent policies determination, especially in a RL framework such as the environment setup by OpenAI Gym. In this walking robot simulation, we denote the parameters, or the model, ***W***, and use OpenAI Gym environment to return the reward given an action. Hence, the model ***W*** links the current state of the simulation to the actions computed, which are evaluated using the reward returned by the simulation.

In our implementation, a neural network was used with 100 neurons in the hidden layer. Hence, with 24 parameters between the state vector (many of them are set to zero) and the hidden layer and 4 possible actions output, the total number of parameters to determine is 2496.

The ES algorithm has many advantages [4] over traditional RL algorithms. For example, the simplicity of the algorithm makes the computation of an episode much faster, resulting in a faster converging algorithm. Also, each modification of the parameter vector is independent of the other ones. Hence, it is simple to parallelize the computation of the whole algorithm, leading to much easier scaling of the overall process and, once again, faster convergence of the algorithm.

Please note that once the model is obtained from the simulation, it is very simple to save it using *cPickle* and to later reload it in the robotics testing platform, before using it the same way as it was used during training. However, it is necessary to convert the state of the robot from raw data readings and actions returned by the agent into useful motor commands.

V. Robotic Testing Platform

“*Reinforcement learning provides a powerful and flexible framework for automated acquisition of robotic motion skills. However, applying reinforcement learning requires a sufficiently detailed representation of the state, including the configuration of task-relevant objects*” [8]. This quote denotes perfectly the biggest issue when implementing a trained agent in a simulation into a real-world system, which was the biggest focus of this project. Detailed coordinate systems can be found in the code available online, which is different from the Box2D toolkit.

The main program was written in python to be able to directly use the learnt model from the simulation. It is currently executed on the computer which communicates with the robot through a FT232RL using UART protocol. A simple Arduino Pro Mini microcontroller was programmed to follow the instructions sent from the computer. A MPU6050 was used as the Inertial Measurement Unit (IMU). Four servomotors are used to apply the torques to the joints and to keep the positions. The communication baud rates were adjusted to obtain a fast response of the robot to a change of commands from the main python code. Finally, tape was added to the legs extremities to mimic the simulation, which never slips.

A more advanced and later version will include a Ultrasound sensor fixed on a rotating frame to provide the LIDAR measurements needed for obstacle avoidance. Also, adding buttons to the legs is a necessity for obtaining a better A.I., which currently walks blindly.

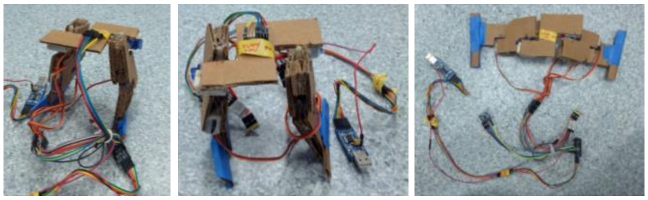


Fig. 4. Walking robotic platform. Note the difference with original simulation, with legs separated by a safe distance. The IMU can be fixed on top of the robot or removed to avoid additional noise and vibrations

An additional filter was used to remove the noise from the IMU measurements from the gyroscope and accelerometer, by updating the current values from raw data:

$$val_i = (1-\alpha) * val_{i-1} + \alpha * val_i, \quad \alpha = 0.6$$

This way, a sudden peak in the read value is slightly attenuated. A better version should embed a Kalman Filter, which was avoided for faster prototyping.

To implement the A.I. in our robot, we import the trained model from the simulation using *cPickle* and reuse the same functions used during the training process. Then, we remove the OpenAI Gym simulation part returning the reward and observations given an action and replace with actions executed directly on the real-world robot, with necessary units and scale conversions.

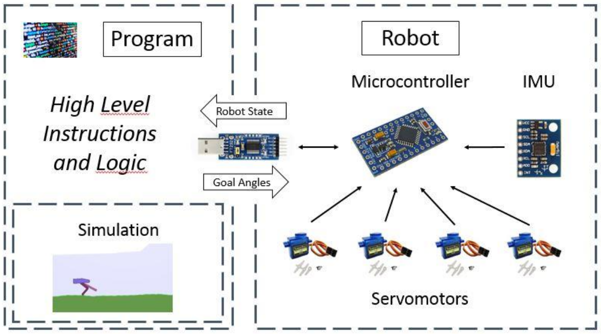


Fig. 5. Block diagram of the robot.
The simulation was run before implementing the model into the robot.

The main code also includes additional security instructions taking care of too large motor angles, which could possibly break the robot by crossing the two legs.

VI. Results

1. Simulation, Cartpole environment

The Cart-pole control problem is solved after 20 iterations and remains stable. In order to speed up the convergence of the Q-learning agent, different values for exploration rates and discount factors were used. Finally, a dynamically changing discount factor was used depending on the number of episodes and the exploration rate was set to $\alpha = 0.5$ which gives decent results. The obtained final reward is the maximum at 199 points average over the 100 last successful trials.

2. Simulation, Walking robot

First, our implementation of the Q-learning algorithm didn't give successful results and was thus quickly avoided for this problem.

Then, we successfully applied the ES algorithm on the simulation and obtained very good results with the full state including LIDAR measurements and ground touch sensing with a reward of 62 after 20 minutes of training on a single processor. Since it was impossible to obtain these values in the current version of the robot, we also removed the LIDAR measurements and touch sense which gave a simple walking strategy stagnating at a maximum reward of 22, which is the direct result of removing vital information.

After training the model on this latest variation of the ES algorithm, we tested the model on the modified simulation which gave similar results as in the original one. As stated in the environment section of this report, training the agent on the modified environment showed to be unsuccessful, since the initial state of the robot appears to be too stable to allow for innovative walking strategies and end up causing the A.I. to remain immobile.

3. Walking Robot Implementation

After training the model, the model was implemented in the final version of the robot. To limit the impact on the IMU readings, the forward direction velocity of the state was set to a positive constant value, in order to avoid too much impact of the accelerometer and of its noise. Also, the IMU remained on the ground to avoid major disturbances in the working agent.

An online available video shows the final result which shows that the final working strategy, although inefficient, achieves a walking pattern which leads to robot approximately 20cm from its original position. Please note that the robot was not pulled by the cables and that the IMU remains on the ground to limit the noise.

VII. Conclusion

In this project, a Q-Learning algorithm was implemented to stabilize a Cart-pole which proved once again that Reinforcement Learning is a viable strategy compared to standard Control theory approaches.

Also, we implemented an Artificial Intelligent trained on a simulation to control a walking robot and showed that it is possible to achieve viable results although if the state is not precise enough (noise, …) and doesn't contain enough information (legs touched the ground), which would be impossible to achieve with traditional approaches which need a mathematical justification behind each move. Here, complex and unpredictable movements and achieved thanks to RL.

VIII. References

[1] AI Mourikis, SI Roumeliotis, A multi-state constraint Kalman filter for vision-aided inertial navigation, IEEE International Conference, Robotics and Automation, 2007

[2] Wei Qingkai, Chen Bao, Niu Zhongguo, Huang Xun, Bang-bang optimal control method with plasma actuators in airfoil roll control, Control Conference (CCC), 2012

[3] L. Tai and M. Liu, "Deep-Learning in Mobile Robotics – from Perception to Control Systems: A Survey on Why and Why Not", arXiv preprint arXiv:1612.07139, 2016.

[4] Salimans, T., Ho, J., Chen, X., and Sutskever, I. (2017). Evolution strategies as a scalable alternative to reinforcement learning. arXiv preprint arXiv:1703.03864.

[5] Inverted autonomous helicopter flight via reinforcement learning, Andrew Y. Ng, Adam Coates, Mark Diel, Varun Ganapathi, Jamie Schulte, Ben Tse, Eric Berger and Eric Liang. In International Symposium on Experimental Robotics, 2004.

[6] Morimoto, J., Cheng, G., Atkeson, C. G., and Zeglin, G. 2004. A simple reinforcement learning algorithm for biped

walking. In Proc. IEEE Int’l Conf. on Robotics and Automation.

[7] KangKang Yin Kevin Loken Michiel van de Panne, SIMBICON: Simple Biped Locomotion Control, ACM Transactions on Graphics, 2007.

[8] Finn, Chelsea, Tan, Xin Yu, Duan, Yan, Darrell, Trevor, Levine, Sergey, and Abbeel, Pieter. Deep spatial autoencoders for visuomotor learning. International Conference on Robotics and Automation (ICRA), 2016

[9] Levine, Sergey, Finn, Chelsea, Darrell, Trevor, and Abbeel, Pieter. End-to-end training of deep visuomotor policies. arXiv preprint arXiv:1504.00702, 2015.

To do in the comments

Specify im MT student who knows a little of everything without CS background

Specify I did it all alone, so it was tough to optimize the agent and so on

Specify I though it would be more interesting to tackle a problem which wasn’t solved before, although it might not be possible to solve it, thus the weird final results available on *“youtube………..”*