CLASSICAL CONTROL PROBLEM
OPEN AI GYM: ACROBOT-V1

SELF-LEARNING SYSTEM PROJECT REPORT

ISA – GROUP MEMBERS
KAH GHI LIM (A0100172A)
CHNG YAN HAO (A0024023A)

Contents

1. Executive Summary

For robots and automation to ease our lives and allow us to enjoy a higher quality of life, the environment must be properly defined and described to the robots for optimal control. This is described by Control Theory, where state equations are often used to describe control problems. However, control of motion can be one of the toughest challenges in a physical environment.

i) Hard coding rules can be too tedious and the slightest change may cause to code from scratch again.

ii) Under stable conditioned environment, the physical science behind each interaction can be complex and formulating an equation may be impossible.

iii) Moreover, most environment have unknown noise or factors, where the physics and mathematics equations are changing and hard to derive.

To allow more efficient solutions to be developed, a robot must learn through interactions with the environment and governing goals or rewards to guide them towards the solution. For example, when young children learn to play a ball game, most of them do not remember the rules. They make mistakes, learn from playing and learn how to score while playing the games. With more practice, they violate less rules, become better and can score more. Self-learning systems aim to do the same, setting up code chunks ready to learn from the environment and the actions they take to perform better in getting the rewards.

In this project, we develop various algorithms from random, fixed to self-learning to solve a classical control problem, Acrobot-v1, from OpenAI Gym. We then compare the performance of the various algorithms to determine the best policy to use.

2. Classical Control Problem

2.1 Introduction

In Control Theory: History, Mathematical Achievements and Perspectives (Fernandez-Cara & Zuazua, 2003), the guiding goal of Control Theory is quoted as “the need of automatizing processes to let the human being gain in liberty, freedom, and quality of life”. State equations are often used to describe control problems in mathematical terms with key concepts such as feedback, room for fluctuations, and optimization.

The state equation basically describes what the controller can control and the effect it has on the system or environment’s state. Feedback then sends the state of the system back to the controller to determine the way the controller acts at any time, for real time control. Keeping room for fluctuations allow for a smoother control process and a harmonic dynamic that would drive the system to the desired state without forcing it too much. Optimization then guides the controller in making decisions with the goal of improving a variable to maximize a benefit or minimize a cost.

To solve control theory problems, the state equation must be derived and then solved to find the optimal point as inputs to the controller. However, hard coding the rules/equations can be tedious and any slight change in conditions may require the system to be coded from scratch again. Besides this, even in a stable conditioned environment, the physical science behind each interaction can be complex and formulating an equation to describe the problem may be impossible. Most environments also have unknown noise or factors, causing the physics and mathematical equations to change and making them difficult to derive.

2.2 Control Theory and Reinforcement Learning

In today’s world, automation and machines aiding humans to improve quality of life and gain liberty and freedom is no longer a distant dream. With control theory, one can now gain insights on how and why feedback control systems work and how to systematically deal with various design and analysis issues (Christiansen, Alexander & Jurgen, 2005). This learning can then in turn help to predict how the systems change over time, in response to different inputs, allowing a more efficient design that can make a system behave in a desirable manner. With automation and rapid advancement of machines and robots, control policies for robots can now be learned directly from camera inputs in the real world (Arulkumaran et al, 2017), with the help of reinforcement learning (RL), instead of hand-engineered controllers.

The same concepts described by Fernandez-Cara and Zuazua are utilized in RL as well. In the simplest context, an agent executes an action on the environment based on the environment's current state (state equation) and the corresponding reward (optimization) and the next state of the environment is then given to the agent as feedback. The agent then determines the next action to be made based on the new state and estimate of future rewards to maximize while keeping some room for fluctuations and the process continues. A simple diagram shown in Figure 1 depicts this process.

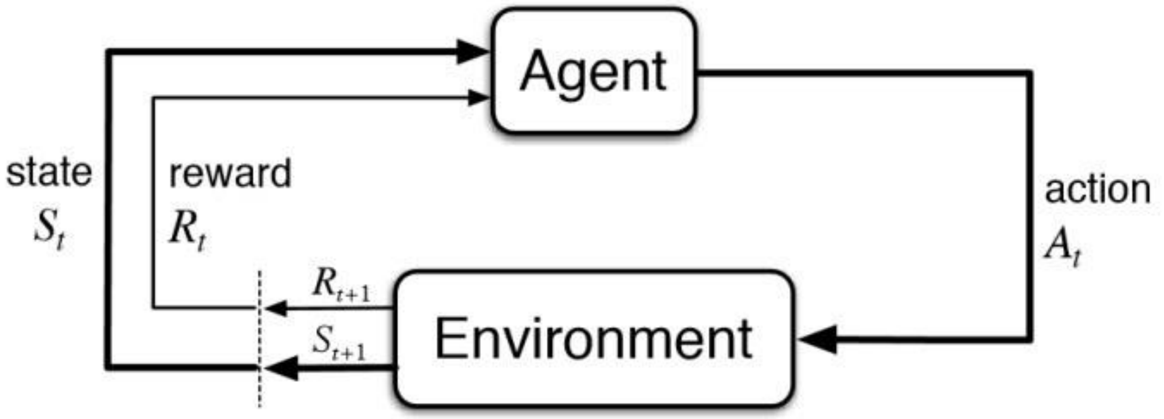


Figure 1: Reinforcement Learning

2.3 OpenAi Gym

Learning an agent's choice of action in a particular state, also known as its policy function, is not an easy task in real-world applications due to the many interactions required by the agent on its environment. RL is essentially driven by learning through interactions with the environment – by observing the consequences of its actions on the environment and how it can change its own behaviour to obtain a higher reward (Arulkumaran et al, 2017).

Having an agent carry out its actions in real-world applications are often not feasible due to the cost or dangers involved. As such, simulated or virtual environments are provided for the agents to learn from. Besides this, there is no standard environment for comparing different RL algorithms on. OpenAI Gym (OpenAI, 2021), which is a toolkit for developing and comparing reinforcement learning algorithm, solves both these issues. It supports a wide range of learning problems, from walking to playing games like Pong or Pinball. Gym also creates better benchmarks to give versatile numbers of environment with great ease of setting up (Rana, 2018).

2.3.1 OpenAI Gym – Control problems

Table 1 shows the various environment available in OpenAI Gym.

Table 1: Environments available in OpenAI Gym

Type	Description
Algorithms	Learn to imitate computations
Atari	Reach high scores in Atari 2600 games
Box2D	Continuous control tasks in the Box2D simulator
Classic control	Control theory problems from the classic RL literature
MuJoCo	Continuous control tasks, running in a fast physics simulator
Robotics	Simulated goal-based tasks for the Fetch and ShadowHand robots
Toy text	Simple text environments to get you started

Under Classic Control type environments, available problems are CartPole-v1, MoutainCar-v0, Pendulum-v0 & Acrobot-v1.

CartPole-v1 has a pole attached by an un-actuated joint to a cart, which moves along a frictionless track. The system is controlled by applying a force of +1 or -1 to the cart. The pendulum starts upright, and the goal is to prevent it from falling over. A reward of +1 is provided for every timestep that the pole remains upright. The episode ends when the pole is more than 15 degrees from vertical, or the cart moves more than 2.4 units from the centre.

MountainCar-v0 describes a car on a one-dimensional track, positioned between two "mountains". The goal is to drive up the mountain on the right; however, the car's engine is not strong enough to scale the mountain in a single pass. Therefore, the only way to succeed is to drive back and forth to build up momentum.

Pendulum-v0 is an inverted pendulum swing up problem, a classic problem in the control literature. In this version of the problem, the pendulum starts in a random position, and the goal is to swing it up so it stays upright.

Acrobot-v1is an acrobot system that includes two joints and two links, where the joint between the two links is actuated. Initially, the links are hanging downwards, and the goal is to swing the end of the lower link up to a given height.

As both CartPole-v1 and MountainCar-v0 has been covered in the course workshops while Pendulum-v0 is a relatively simple mathematical problem to solve with widely available solutions online, Acrobot-v1 is chosen and various techniques are used to learn the system.

3. Design of Acrobot-v1

3.1 Basic environment set-up

Acrobot-v1 aims to swing a two-link robot pass the line which is 1 unit above the primary support, and a schematic of the environment is depicted by the image on the left of Figure 2. Each link is of 1-unit length and 1-unit mass, with the centre of mass of each link at 0.5. The maximum of the first and second angular velocity is 4 π and 9 π respectively.

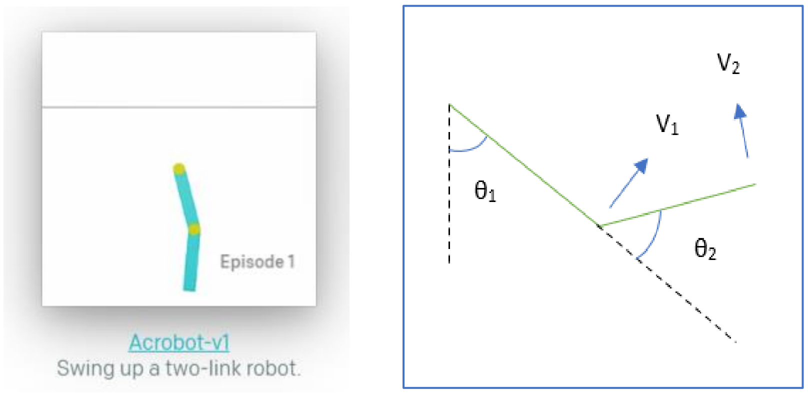


Figure 2: Schematic diagram of the Acrobot-v1 environment (left) and states (right)

The states of Acrobot-v1 consists of [θ_1, θ_2, V_1, V_2], where θ_1 represents the angle between the first link and imaginary vertical line while θ_2 represents the angle between the second link and link 1, both with values from $-\pi$ to π. V_1 and V_2 represents the first and second angular velocity respectively, from -4π to 4π and -9π to 9π.

The action space consists of three actions: [0, 1, 2], where action 0 applies -1 torque on the joint in a clockwise direction, action 1 does nothing and action 2 applies +1 torque on the joint in an anti-clockwise direction.

The observation space is described by 6 elements, [$\cos \theta_1$, $\sin \theta_1$, $\cos \theta_2$, $\sin \theta_2$, V_1, V_2], with terminal state described by the equation $-\cos \theta_1 - \cos(\theta_1 + \theta_2) > 1$.

A reward of -1 is given if the tip of 2nd link is below the line (i.e. $-\cos \theta_1 - \cos(\theta_1 + \theta_2) \leq 1$) and a reward of 0 is given if the tip of 2nd link is above the line (i.e. $-\cos \theta_1 - \cos(\theta_1 + \theta_2) > 1$).

The environment setup is based on Acrobot-v1 environment from OpenAI Gym as of May 2021 and the usual conventions. The environment self terminates after 250 swings within an episode. Any discrepancies made during the interpretation above should not affect the algorithms in Section 3.3 as the illustrations are purely for pictorial understanding of the problem.

3.2 Performance Measures

3.2.1 First Success

First success is defined as the primary aim of our agents on Acrobot-v1.

Example 1:

If our agent made 250 swings in an episode and none of the swings could go beyond the line, first success = -250.

Example 2:

If our agent made 250 swings in an episode and 61st, 71st & 81st swing had gone beyond the line, first success = -60 (cumulative reward till first success).

3.2.2 Cumulative Reward: Beyond First Success – Spinning and Acrobatics

A secondary objective for our agents is defined as the number of swings above the line over the total swings made within an episode. This is to explore whether any agent can learn some spinning or acrobatic techniques to allow the two-link robot to stay ‘afloat’.

Example 1: If our agent made 250 swings in an episode and none of the swings could go beyond the line, cumulative reward = -250

$\rightarrow$ [-250 – (-250)] / -250 = 0% of the swings

Example 2: If our agent made 250 swings in an episode and 25 of the swings could go beyond the line, cumulative reward = -225

$\rightarrow$ [-250 – (-225)] / -250 = 10% of the swings

For simplicity of calculations, only the swings that go beyond the line are counted, disregarding the prior swings that could have helped the Acrobot-v1 to swing up.

3.3 Algorithms

Various algorithms have been explored to solve Acrobot-v1, from random policy, fixed policy to learning ones such as genetic algorithm, Q-learning and SARSA. The different set-ups and their average performance, based on the total number of episodes ran, are covered here. More details and performance comparison between the algorithms are covered in Section 4. The graph depicting the performance of each algorithm can be found in Section 7.

3.3.1 Base Agent - Random

The base agent is present for all the subsequent agents. A random policy is loaded as its default.

Main functions

__init__:	loads the gym environment and variables.
choose_action:	randomly pick an action from the action space.
plot_graphs:	plots first success and mean cumulative reward against episodes.
save_results:	save the plots and lists.
run_episodes:	iterates episodes of swings based on the choices of action.

Main variables

self.episodes:	number of episodes to run (an integer set to 100).
self.swings:	number of swings in an episode (an integer set to 250).
self.track:	number of episodes to track at the end (an integer set to 100).

Performance of random agent

Best attempt:	-117 (Agent's first success using 117 swings).
Mean of first reward:	-239.77 (less than 20/100 episodes, agent able to swing up).
Mean of cumulative:	-248.92 (<1% could reach the line in an episode of 250 swings).
Time taken:	9.5s (To run 100 episodes).

3.3.2 Fixed policy - Theta1

The Theta1 agent is inspired from one of CartPole-v1's solutions (Xu, 2021). To achieve larger θ_1, we simply decide to push it further by following the direction it is pointing in. For example, when θ_1 is positive, a possible position is as depicted on the right of Figure 2. To push it up and achieve the goal, a +1 torque is applied, causing the link to turn in the anti-clockwise.

Main function

choose_action:	θ_1 is positive, apply +1 torque / θ_1 is negative, apply -1 torque.

Performance of Theta1 agent

Best attempt:	-64 (Agent's first success using 64 swings).
Mean of first reward:	-98.72 (all episodes, agent able to swing up).
Mean of cumulative:	-219.87 (~12% could reach the line in an episode of 250 swings).
Time taken:	10.4s (To run 100 episodes).

3.3.3 Fixed policy – V1

The angular velocity agent is a fixed policy agent discovered during exploration stage. The policy is also inspired from one of CartPole-v1's solution (Xu, 2021). To reach the line, we simply decide to push against V_1 to prevent it from going back to initial position.

Main function

choose_action:	V_1 is positive, apply -1 torque / V_1 is negative, apply +1 torque.

Performance of V1 agent

Best attempt:	-35 (Agent's first success using 35 swings).
Mean of first reward:	-43.08 (all episodes, agent able to swing up).
Mean of cumulative:	-204.21 (~18% could reach the line in an episode of 250 swings).
Time taken:	10.6s (To run 100 episodes).

3.3.4 Evolutionary Learning: Genetic Algorithm (GA)

Genetic Algorithm is an algorithm inspired by nature. A population of random chromosomes is initiated and validated against a fitness function. The chromosome is of length 250, each describing an action to be taken for the various swings. The initial population with solutions would survive the validation of the fitness function and get selected by the selection function. These chromosomes would produce offspring in the next generation through crossover and mutation. The offspring then go through the same process as their parents.

Main functions

__init__:	added new variables.
get_initial_pop:	get the initial population by the base agents' random approach
fitness:	returns a value, the lower the fitter E.g. for an episode with first success as -100, this value would be -250-(-100) = -150 and computed each episode.
selection:	selection is in tournament style, i.e. the chromosome with a better score with be selected among k-1 chromosomes. k is set to 3.
crossover:	one-point crossover, randomly choose a point and two chromosomes exchange their tails
mutation:	changes an action taken in a gene of the chromosome
genetic_algorithm:	as described in the initial paragraph

Main variables

self.n_iter:	number of iterations, set to 100 generations
self.n_bits:	number of bits, set to 250
	a chromosome is length 250, with [0,1,2] as each element
	E.g. [0, 0, 1, 2, 2, … <250th term>]
self.n_pop:	number of chromosomes in an iteration, set to 500
self.r_cross:	a chance of chromosome undergoing crossover
self.r_mut:	a chance of chromosome undergoing mutation

Performance of GA agent

Best attempt:	-44 (Agent's first success using 44 swings)
Mean of first reward:	-76.36 (most episodes, agent able to swing up)
Mean of cumulative:	-234.44 (~6% could reach the line in an episode of 250 swings)
Time taken:	2165s (To run 100 generations on 500 chromosomes)

3.3.5 Reinforcement Learning: Q-Learning

Q-Learning is a self-learning algorithm based on Bellman's equation by updating the q_table. The observations are discretized before forming the q_table. This off-policy approach of Q-Learning would update the q_table by estimating the next state based on best action for the state. An action with the maximum possible value would then be chosen for the current state.

Main functions

__init__:	changes number of episodes and added new variables.
discretize_state:	discretize the continuous numbers into bins.
get_epsilon:	ε is a small percentage that the action taken would be random. ε is set to decrease with time steps, so that exploitation is more than exploration towards the end.
get_learning_rate:	learning rate is set to decrease with time steps, so that learning is based on more recent episodes than episodes at the beginning.
choose_action:	takes action based on argument that return the max in q_table, with a small chance of action being random, based on epsilon-greedy action algorithm.
update_table:	Estimate the next state's value by choosing the best action for that state. Returns a value for current state taking the next state into consideration.

Main variables

self.episodes:	changed to 1000
self.buckets:	(1,1,1,1,2,5), setting the observation into bins
self.min_lr:	minimum learning rate = 0.05
self.min_epsilon:	minimum epsilon = 0 (allowing full exploitation at the end)
self.discount:	discount on next state = 0.99, used in updating q_table
self.decay:	an integer as denominator in learning rate. The larger the number the lower the learning rate
self.q_table:	a table of actions and states

Performance of Q-learning agent

Best attempt:	-35 (Agent's first success using 35 swings)
Best 100, first reward:	-36.82 (most episodes, agent able to swing up)
Last 100, first reward:	-50.49 (most episodes, agent able to swing up)
Mean of cumulative:	-209.19 (~16% could reach the line in an episode of 250 swings)
Time taken:	73.5s (To run 1000 episodes)

3.3.6 Reinforcement Learning: SARSA

SARSA is another self-learning algorithm based on Bellman's equation. The observation states are discretized to form the q_table. This on-policy approach of SARSA would estimate the next state by selecting the action based on the policy formulated previously. An action with the maximum possible value would then be chosen for the current state.

Main functions

__init__:	changes number of episodes and added new variables.
discretize_state:	discretize the continuous numbers into bins.
get_epsilon:	ε is a small percentage that the action taken would be random. ε is set to decrease with time steps, so that exploitation is more than exploration towards the end.
get_learning_rate:	learning rate is set to decrease with time steps, so that learning is based on more recent episodes than episodes at the beginning.
choose_action:	takes action based on argument that return the max in q_table, with a small chance of action being random, based on epsilon-greedy action algorithm.

update_table:	Estimate the next state's value by following the policy's previous update. Returns a value for current state taking the next state into consideration.

Main variables

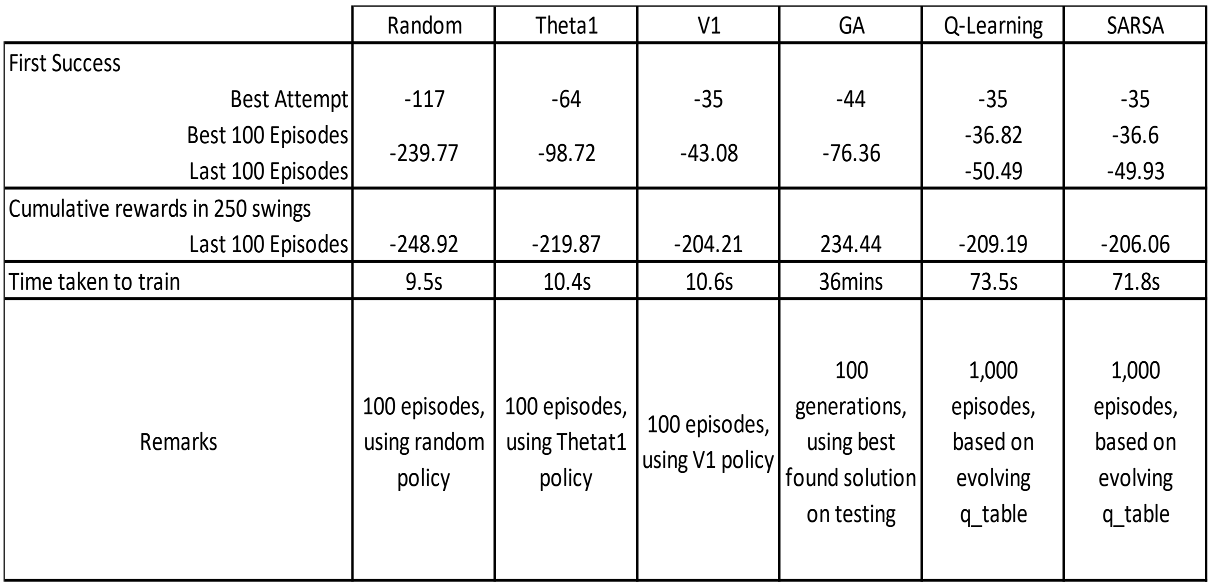
self.episodes:	changed to 1000
self.buckets:	(1,1,1,1,2,5), setting the observation into bins
self.min_lr:	minimum learning rate = 0.05
self.min_epsilon:	minimum epsilon = 0 (allowing full exploitation at the end)
self.discount:	discount on next state = 0.99, used in updating q_table
self.decay:	an integer as denominator in learning rate. The larger the number the lower the learning rate
self.q_table:	a table of actions and states

Performance of SARSA agent

Best attempt:	-35 (Agent's first success using 35 swings)
Best 100, first reward:	-36.6 (most episodes, agent able to swing up)
Last 100, first reward:	-49.93 (most episodes, agent able to swing up)
Mean of cumulative:	-206.06 (~18% could reach the line in an episode of 250 swings)
Time taken:	71.8s (To run 1000 episodes)

4. Results

4.1 Intra-comparison



	Random	Theta1	V1	GA	Q-Learning	SARSA
First Success						
Best Attempt	-117	-64	-35	-44	-35	-35
Best 100 Episodes	-239.77	-98.72	-43.08	-76.36	-36.82	-36.6
Last 100 Episodes					-50.49	-49.93
Cumulative rewards in 250 swings						
Last 100 Episodes	-248.92	-219.87	-204.21	234.44	-209.19	-206.06
Time taken to train	9.5s	10.4s	10.6s	36mins	73.5s	71.8s
Remarks	100 episodes, using random policy	100 episodes, using Thetat1 policy	100 episodes, using V1 policy	100 generations, using best found solution on testing	1,000 episodes, based on evolving q_table	1,000 episodes, based on evolving q_table

Figure 3: Performance of the various algorithms evaluated for Acrobot-v1

Figure 3 compares the performance of the various algorithms described in Section 3. Although the random policy took the least time to train compared to the rest, the performance of the first success and cumulative rewards are worse. This was because no learning is involved, and any action taken was random. A random policy has less than 20% of its episodes for the two-link robot to swing above the line even when given 250 swings in every episode. Most of the time, the two-link robot stayed below the target line. Policies guided by heuristics, such as Theta1 and V1, as well as learning policies performed better than then random policy.

Fixed policies such as Theta1 and V1 performed better than the random policy due to guided heuristics to aid the agent in choosing its action. Based on the results, controlling the first angular velocity, and hence preventing the first link from going back to its initial position, was a better heuristic than controlling Theta1. We can indirectly infer that the angular velocity, V_1, has a larger impact on the Acrobot-v1's dynamics than θ_1.

Using GA to do a guided search for a solution greatly increase the performance compared to random and Theta1 policies. The performance had outperformed the simple intuitive Theta1 policy in getting the first success. As the fitness function uses direct evaluation, the time taken is the longest. GA was able to provide good solutions although it was not fed with any instantaneous state information.

Q-Learning and SARSA were both able to perform on par with V1 Policy after some training episodes. SARSA seems to perform marginally better and this might suggest that on-policy algorithms are more suitable to solve Acrobot-v1. Both Q-Learning and SARSA have learnt from its interactions with the environment without knowing the physics behind each of the two-link robot's swings. The results show that RL methods are a feasible way to solve Classical Control problems without needing to understand the deeper fundamentals of mathematical equations and physics theory, allowing for automation and faster, efficient system designs.

4.2 Inter-comparison

Acrobot-v1 was also solved by others using Neural Network (NN) Agents (Pan, 2019) and other RL techniques (Upadhyaya, 2020) such as Advantage Actor Critic (A2C), Proximal Policy Optimization (PPO2), Actor-Critic with Experience Relay (ACER) and Actor-Critic using Kronecker-Factored Trust Region (ACKTR). Table 2 consolidates the results.

Table 2: Other solutions for Acrobot-v1 from other sources

First Success	Q-Learning	SARSA	NN	A2C	PPO2	ACER	ACKTR
Best 100 episodes	-36.82	-36.6	-42.37	-	-	-	-
Last 100 episodes	-50.49	-49.93	-74.9	-89.6	-74.9	-77.8	-72.3
Training time	73.5s	71.8s	404s	290.2s	273.3s	392.5s	253.5s
Remarks	1,000 episodes, based on evolving q_table	1,000 episodes, based on evolving q_table	1,000 episodes, based on simple 2-layer NN	1,000 episodes, based on stable baseline package	1,000 episodes, based on stable baseline package	1,000 episodes, based on stable baseline package	1,000 episodes, based on stable baseline package

The Neural Network Agent was built using two NN – one for the actor and one for the critic, based on an on-policy state-value temporal difference. Although conceptually this makes it most like SARSA, SARSA is still able to learn better with lesser training time and higher rewards.

Under this environment, our customized Q-Learning and SARSA agents are able to outperform the A2C, PPO2 ACER and ACKTR models loaded directly from stable baseline package (Stable Baselines, 2021) in terms of both training time and rewards obtained from the first success. This could be because the models were directly loaded with no tuning of hyperparameters and discretization of the observations.

4.3 Further Training & Beyond First Success

As SARSA appeared to yield the best results thus far, more episodes were used to train and update the q_table. SARSA was further trained with a total of 10,000 episodes and was set to have 5,000 swings per episode while keeping the same hyper parameters. The results are shown in Figure 4.

	SARSA	
First Success		
Best Attempt	-35	-35
Best 100 Episodes	-35	-36.6
Last 100 Episodes	50.23	-49.93
Cumulative rewards	-	
Last 100 Episodes	-3531.95	-206.06
Time taken to train	4 hrs	71.8s
Remarks	10,000 episodes, 5,000 swings	1,000 episodes, 250 swings

Figure 4: Results of SARSA trained with 10,000 episodes and 5,000 sings per episode

Despite training with more episodes and more swings per episode, the agent was unable to reach a solution below 35 moves. However, the reward for first success is comparable to that of 1000 episodes with an improvement in the average cumulative reward from ~18% (section 3.3.6) to ~29%.

In the gif storing the rendered frames attached with this file submission, the agent has learnt:

i. To swing itself up with few moves.
ii. Spinning techniques as shown in Figure 5. (to repeatedly being rewarded)
iii. Some acrobatic tries to explore further for developing new strategy above the line.

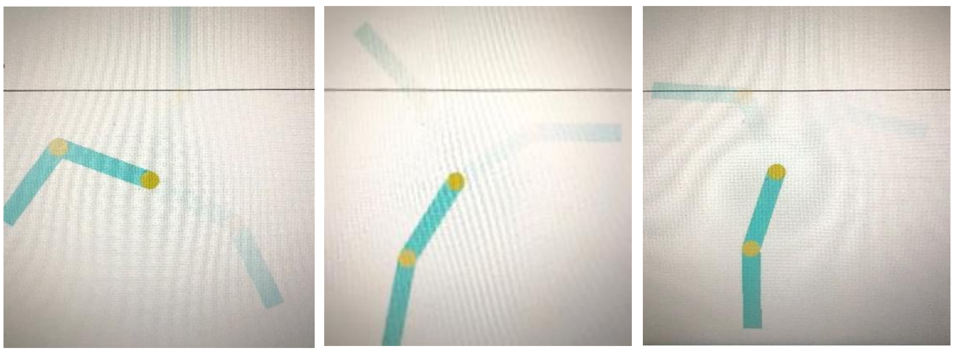


Figure 5: Agent learning how to spin

Although the agent was unable to reach a solution below 35 moves, the agent seems to have learnt some "techniques" to stay afloat 8% longer. Subtracting 35 moves in both numerator and denominator in section 3.3.6, the two-link robot guided by the SARSA agent trained for 1,000 episodes and 250 swings was estimated to have stayed afloat for -215-(-171)/-215 = ~21%.

5. Conclusion

In this project, we have developed solutions to solve OpenAI Gym's Acrobot-v1 using Genetic Algorithm, Q-Learning and SARSA. All methods were able to achieve above -100 reward for their first success - a measure used by other researchers.

Without knowledge of the environment, applying Genetic Algorithm had fared much better than the random agent and more efficient than any brute force evaluation means. It had even outperformed simple logic from angle theta policy.

Hand coded RL techniques, Q-Learning and SARSA with state discretization, were more suitable to this problem when they were compared with other baseline loaded RL packages.

Beyond the problem of first success, we hypothesized that the RL agents may even learn a technique beyond spinning to receive more rewards continuously. We might see the two-link robot balancing itself above the line similar to that in the CartPole-v1 problem. However, that would be another problem for another day. Through this simplified example on control problem, we are sure that RL techniques have the potential to solve real-world problems. They are feasible to improve our quality of life by aiding control systems design in a more efficient and safer manner.

6. References

Arulkumaran, K., Deisenroth, M. P., Brundage, M. & Bharath, A. A. (2017). A Brief Survey of Deep Reinforcement Learning. *IEEE Signal Processing Magazine, Special Issue on Deep Learning for Image Understanding (ARXIV Extended Version).* Retrieved from https://arxiv.org/pdf/1708.05866.pdf

Christiansen, D., Alexander, C. K. & Jurgen, R. K. (2005). Control Systems. In D., Christiansen, C. K., Alexander & R. K., Jurgen (Ed.) *Standard Handbook of Electronic Engineering* (5th ed., pp. 19.1 – 19.30) The McGraw-Hill Companies, Inc.

Fernandez-Cara, E. & Zuazua, E. (2003). Control Theory: History, Mathematical Achievements and Perspectives. *Bol. Soc. Esp. Mat. Apl. no0 (0000), 1–62*. Retrieved from https://www.sissa.it/fa/workshop_old/DCS2003/reading_mat/zuazuaDivSEMA.pdf

OpenAI. (2021). *Gym*. Retrieved from https://gym.openai.com/

Pan, H. (2019, Dec 3). OpenAI Gym - Acrobot-v1 [Blog Post]. Retrieved from http://www.henrypan.com/blog/reinforcement-learning/2019/12/03/acrobot.html

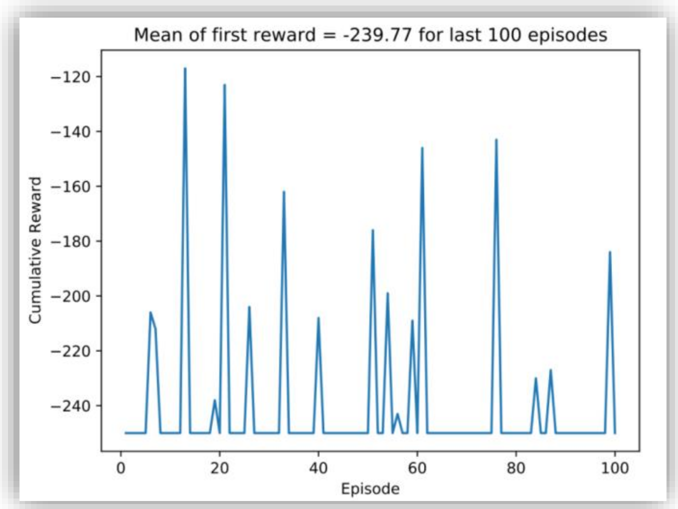
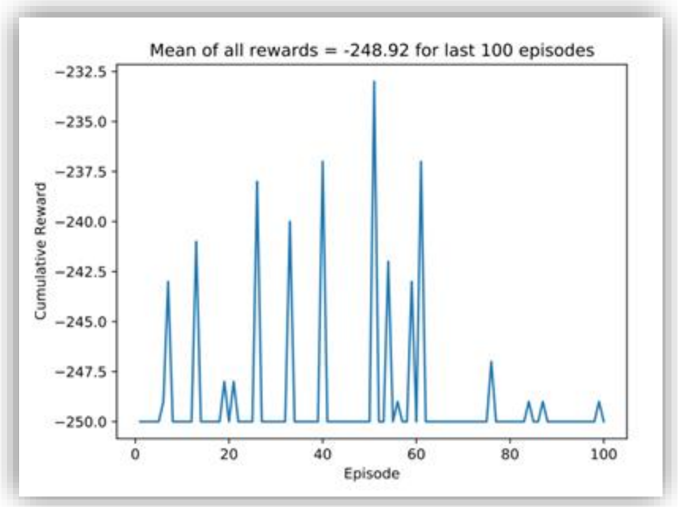
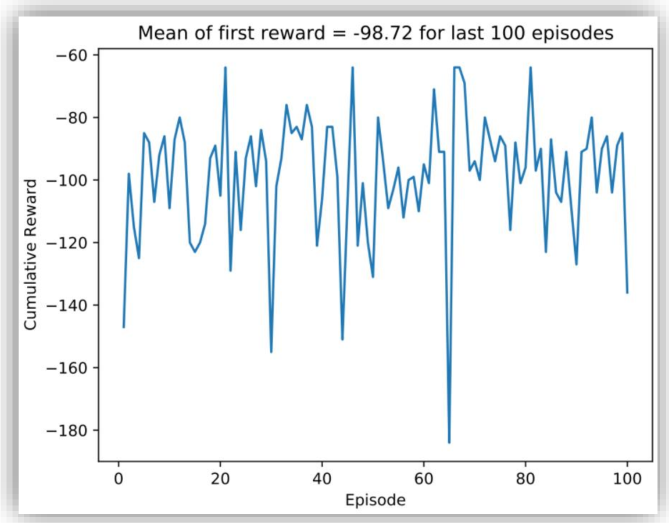
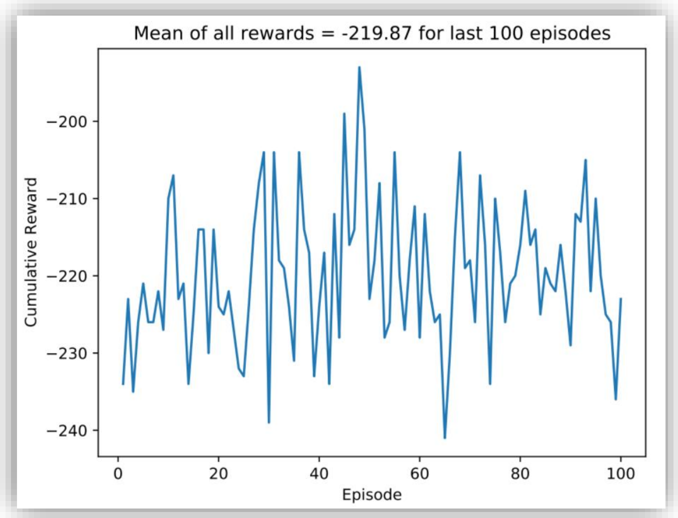
Rana, A. (2018, Sep 21). Introduction: Reinforcement Learning with OpenAI Gym [Blog Post]. Retrieved from https://towardsdatascience.com/reinforcement-learning-with-openai-d445c2c687d2

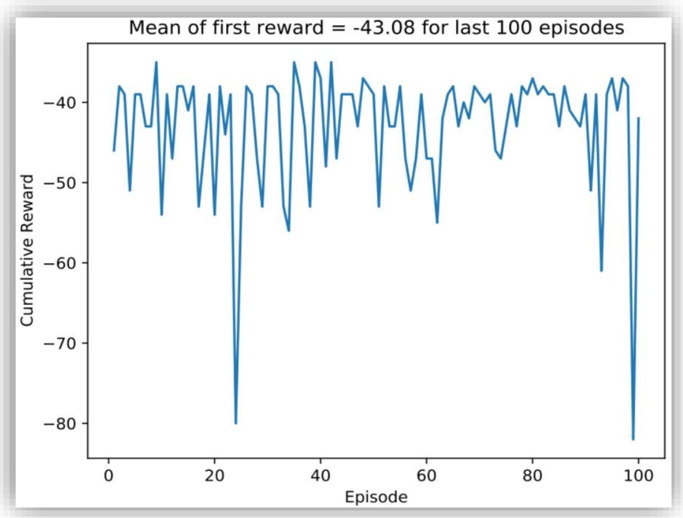
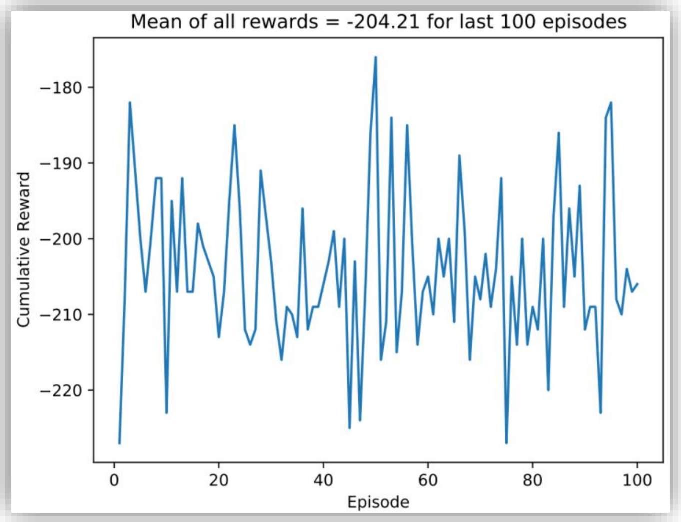
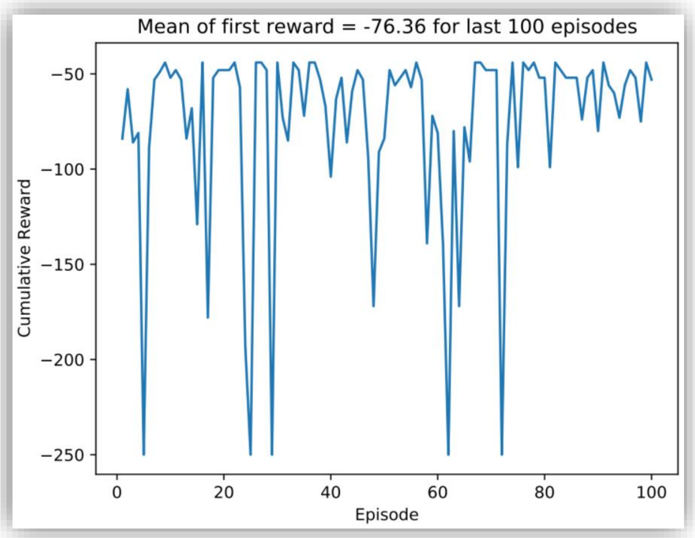
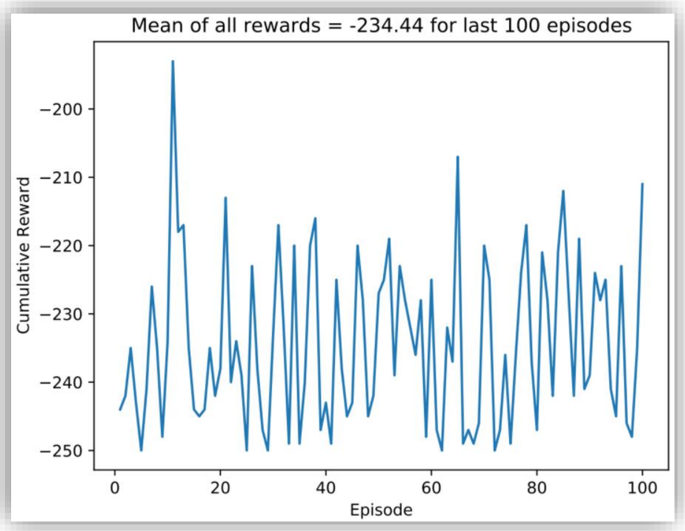
Stable Baselines. (2021). *Stable Baselines*. Retrieved from https://stable-baselines.readthedocs.io/en/master/modules/a2c.html

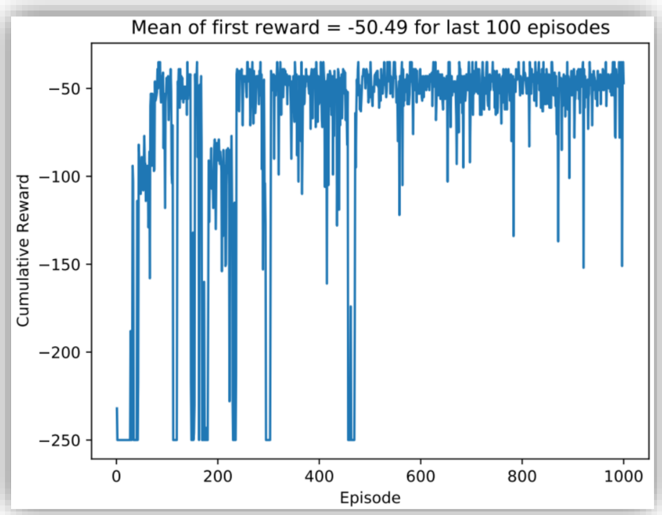
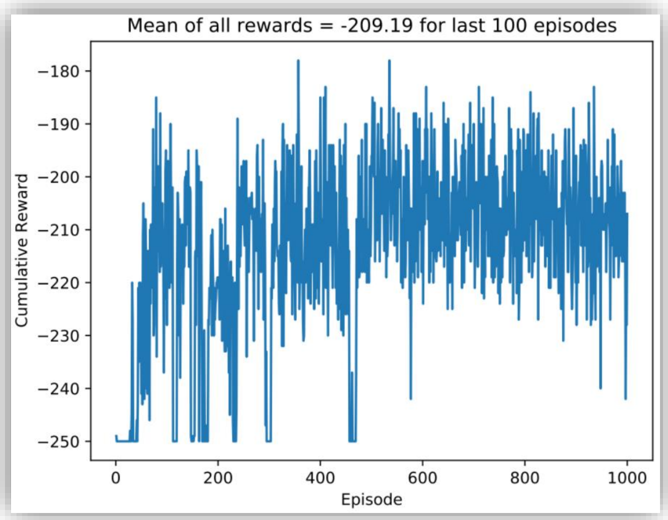
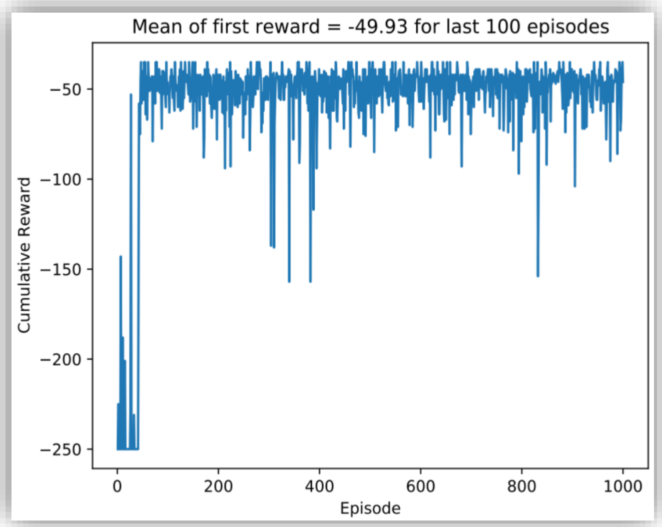
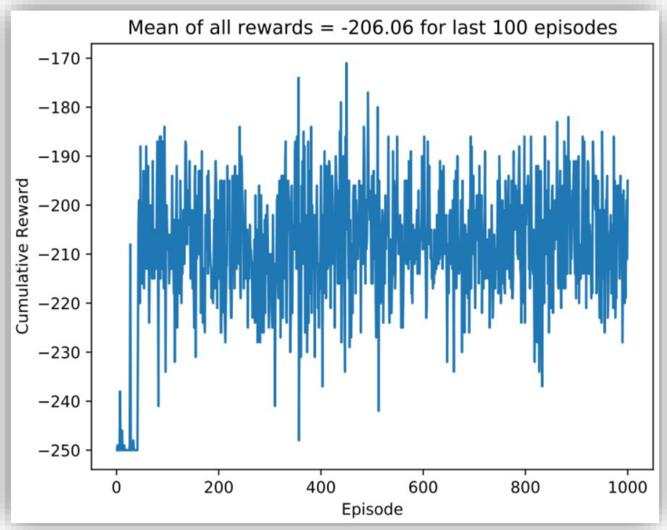
Upadhyaya, A. (2020, Mar 16). A Hands-On Guide on Training RL Agents on Classic Control Theory Problems [Blog Post]. Retrieved from https://analyticsindiamag.com/a-hands-on-guide-on-training-rl-agents-on-classic-control-theory-problems/

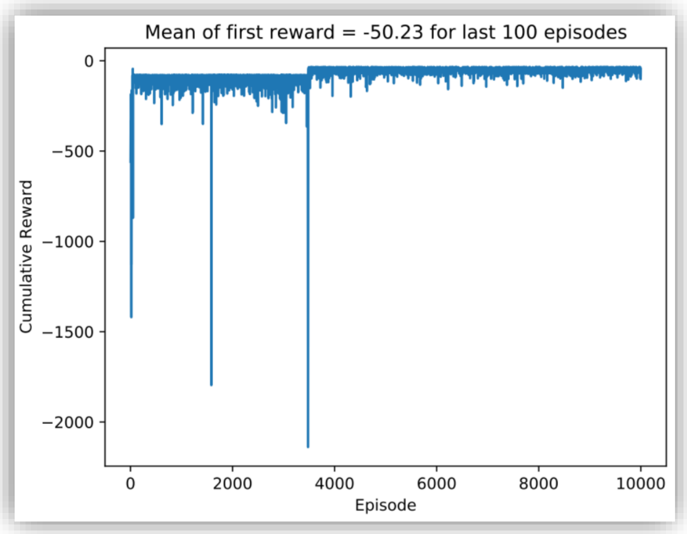
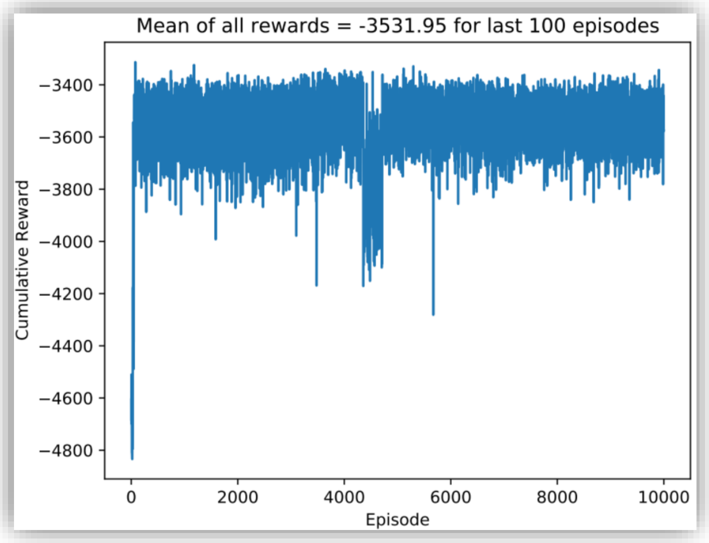
Xu, J. (2021, Feb). How to Beat the CartPole Game in 5 Lines [Blog Post]. Retrieved from https://towardsdatascience.com/how-to-beat-the-cartpole-game-in-5-lines-5ab4e738c93f

7. Appendices

Algorithm	First Reward	Cumulative Reward
Random Policy		
Theta1		

V1	Mean of first reward = -43.08 for last 100 episodes Cumulative Reward Episode	Mean of all rewards = -204.21 for last 100 episodes Cumulative Reward Episode
GA	Mean of first reward = -76.36 for last 100 episodes Cumulative Reward Episode	Mean of all rewards = -234.44 for last 100 episodes Cumulative Reward Episode

Q-Learning		
SARSA		

SARSA_10000		

System requirements and base codes are also available on GitHub:

https://github.com/RyanChngYanHao/ISA-PM-SLS-2021-01-09-IS02PT-GRP-Acrobot-v1