High-Dimensional Continuous Control Using Linear Genetic Programming

Jingxuan Liu*
liu1819@mcmaster.ca
McMaster University
Hamilton Ontario, Canada

ABSTRACT

Currently, Linear Genetic Programming has not been widely applied to the training of MuJoCo environment. This article will explore and attempt to use Linear Genetic Programming on Ant tasks in this environment. Research on the performance of Linear Genetic Programming in Reinforcement Learning for High Dimensional Continuous Control Tasks.

KEYWORDS

gym environment, reinforcement learning, linear genetic programming

ACM Reference Format:
Jingxuan Liu. 2023. High-Dimensional Continuous Control Using Linear Genetic Programming. In *Proceedings of ACM Conference (Conference'17).* ACM, New York, NY, USA, 4 pages. https://doi.org/10.1145/nnnnnnn.nnnnnnn

1 INTRODUCTION

Linear Genetic Programming (LGP) is an evolutionary algorithm used to generate computer programs to solve problems. It optimizes the program through evolution and automatic search, enabling the program to perform the required tasks. It has been successfully applied on several domains, like regression and control problems.

For control problems, the Gym environment is widely used for reinforcement learning. The Gym environment is an open-source Python library for building, testing, and comparing reinforcement learning algorithms. It provides a standardized interface for defining and creating reinforcement learning environments, making it easy for researchers to develop and evaluate reinforcement learning algorithms.

The characteristics of genetic programming (GP) make it a powerful tool for implementing reinforcement learning. We will use LGP to generate the programs to interact with the Gym environment.

2 BACKGROUND DISCUSSION

We will attempt to use the existing linear genetic programming Python implementation [1] to the gym environment. The platform we are using was created to perform the regression task, which means we need to modify it to fit our task.

2.1 Related Work

The original program consists of several programs. $calc\_fit.cpp$ is responsible for calculating the output of an individual, and both the reading and output processes interact with the memory. This code is written in C++and needs to be generated through Swig and Python interfaces; $fitness.py$ is responsible for calculating the fitness of individuals and populations; $ind\_creation.py$ is responsible for generating individuals and populations; $LGP.py$ is the main running program of the algorithm, which also includes the writing of logs and the use of EA.

Since the source code is running on the regression task, we need to make changes to adapt it to our task. Firstly, the source code uses 2d-array as the register, and we have changed it to one-dimensional. Due to the changes in the memory shape, we need to modify all the functions that reference registers, for instance, the function inside the $calc\_fit.cpp$ that calculates the result of the instructions, we need to make sure they interact with the register appropriately. At the same time, in $ind\_creation.py$, we ensure that the generation of the register index conforms to our format. The modification of $fitness.py$ is the most important, and we need to create an evaluation function to enable the environment to interact with individuals to calculate the individual's fitness through the environment's reward.

2.2 Gym Environment

We consider two tasks in the Mujoco environment; one is the most difficult, and the other one is the opposite.

The 'Ant' [1] is a 3D robot consisting of one torso (free rotational body) with four legs attached to it with each leg having two links. The goal is to coordinate the four legs to move in the forward (right) direction by applying torques on the eight hinges connecting the two links of each leg and the torso (nine parts and eight hinges).

Table 1: Space for Ant

Action Space	Box(-1.0, 1.0, (8,), float32)
Observation Space	(27,)

The 'Swimmers' [5] consist of three or more segments 'links' and one less articulation joints 'rotors', the goal is to move as fast as possible towards the right by applying torque on the rotors and using the fluids friction.

Conference'17, July 2017, Washington, DC, USA

ACM ISBN 978-x-xxxx-xxxx-x/YY/MM…$15.00
https://doi.org/10.1145/nnnnnnn.nnnnnnn

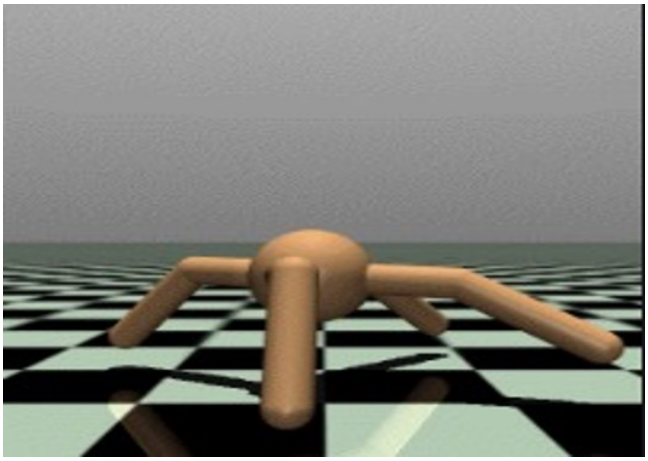

Figure 1: Ant Task.

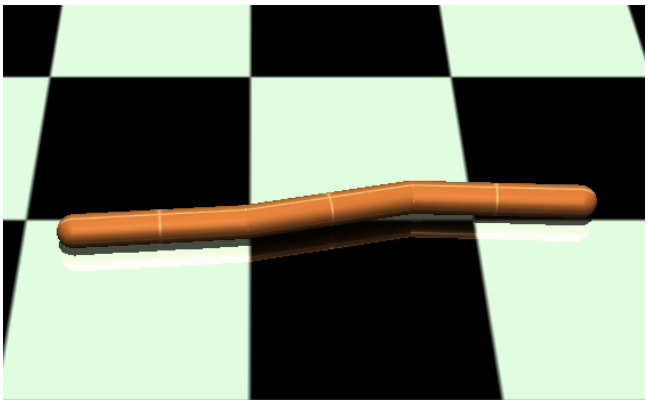

Figure 2: Swimmer Task.

Table 2: Space for Swimmers

Action Space	Box(-1.0, 1.0, (2,), float32)
Observation Space	(8,)

3 METHOD

In this report, we used Linear Genetic Programming (LGP), an evolutionary algorithm used to generate computer programs to solve problems. The working principle of LGP can be summarized as follows: Firstly, we randomly generate an initial population of computer programs. Next, use problem specific fitness functions to evaluate the performance of each program. The probability of selecting programs with high fitness is higher. Then, the selected program generates the next generation program through crossover and mutation operations, and repeats this process for multiple generations until the stop condition is met. Ultimately, we choose the program that best suits the problem as the result. The core idea of LGP is to generate optimized computer programs through evolutionary search to solve various problems, such as function fitting, symbolic regression, and control tasks.

3.1 Individual

LGP will initialize and generate individuals on its own, and then generate offspring through mutation. The individual is represented by a linear sequence of instructions, each instruction has read and output arguments and operators. The read and output argument is associated with registers. And many individuals made a population.

1: r[3] = r[0] XOR r[1]
2: r[4] = r[0] AND r[1]
3: r[5] = r[2] AND r[3]
4: r[6] = r[3] AND 1.0
5: r[0] = r[5] OR r[4]
6: r[1] = r[2] XOR r[3]

Figure 3: Individual made by a sequence of instructions.

In our genetic program, two functions are responsible for optimizing an individual's instruction set and generating new instructions, respectively. The remove introns function simplifies an individual's genome by identifying and removing instructions that have no impact on their final output. This process starts from the end of the individual and checks in reverse to see if each instruction affects the output register, effectively filtering out the set of instructions that directly contribute to the output. This method improves individual execution efficiency and helps reduce the consumption of computing resources.

On the other hand, the generate instruction function randomly generates a new instruction, involving randomly selecting operators, destination registers, and one or two operands. The operand can be a register value or a constant, depending on predefined probabilities.

Table 3: List of Computational Symbols Used

Symbol Type	Symbol
Logical Operators	AND, AND_INV1, XOR, OR
Arithmetic Operators	+, -, *, /
Trigonometric Functions	sin, cos
Exponential Functions	e, ln

We only used high school mathematical operators to perform the basic physics formula calculations, which will used for 'survival' in the environment.

3.2 Register

The length of a register is the number of observations plus the number of actions. The length of a register is the number of observations plus the number of actions. For instance, in the case of an ant task, the register is 27 plus 8, and the argument reads the value

from the first 27 elements of the register. The output value is stored in the last 8 elements of the register, and the environment's action is also read from the last 8 elements of the register.

3.3 Mutation

There are two kinds of mutation strategies, the micro and the macro mutation. For micro mutation, there are three change actions. first is change target register, it changes the target register of an instruction at a specified position, randomly selects a new target register and ensures that it's different from the current target register. second is change the parameter, modify the parameter of instructions at a certain position in a mutated individual, focus on mutation register parameters. And third is the change operator, which changes the operator of an instruction at a certain position.

For the macro mutation, it also have three actions. The first is insertion, insert a parameter for instructions at a certain position in the mutated individual. The second is removal, which removes an instruction from the specified position of the individual. The third is substitution, which replaces the instruction at the specified position in the individual with a newly generated instruction. The mutation also has a learning rate to set.

3.4 Evaluation

The purpose of the evaluation function is to evaluate the fitness of individuals in a predetermined environment. It evaluates individual performance through multiple trials (20 in this case) to ensure that the fitness score obtained reflects the individual's average performance well. In each experiment, the environment is first reset to its initial state and initial observation data is obtained from it. Next, the function enters a loop, continuously calculating the actions that individuals should take and executing these actions. Every time an action is executed, the individual receives feedback, including new observation data, rewards received, whether the goal has been achieved or whether the interruption condition has been met.

If an exception is encountered during the execution of an action, such as an action exceeding the allowable range of the environment, the function will immediately stop the current experiment and set the individual's fitness score to zero. This protection mechanism ensures that the program will not continue to execute when encountering illegal operations, thereby avoiding possible errors or uncertain behaviour. The rewards obtained for each action will be accumulated into the total reward for this experiment. After the end of an experimental cycle, this accumulated reward will be added to the total fitness. Finally, the function calculates the average reward for all experiments and returns it as an individual's fitness score. This score can determine whether an individual is retained, replicated, or eliminated in the iteration of genetic algorithms.

3.5 Linear Genetic Programming

The core of evolutionary algorithms lies in constantly selecting and updating individuals to find the optimal solution. Firstly, the tournament function implements the tournament selection mechanism. In this process, a group of individuals is randomly selected and their fitness is compared to determine which individuals are the "winners" (those with the highest fitness) and "losers" (those with the lowest fitness).

Next, the population will be updated by the function. After the tournament selection, the selected winner is used to replace the loser, thus preserving excellent genetic characteristics. At the same time, the function introduces new individuals to replace the original winner's position, which are usually generated by mutating the winner's genes. This update mechanism aims to introduce new genetic diversity while retaining existing excellent characteristics.

We also use the Steaty State genetic algorithm. In this process, the algorithm only replaces a portion of the population in each generation. The algorithm effectively balances the relationship between exploration (introducing new individuals) and utilization (retaining excellent individuals) by repeatedly executing tournament selection and updating the population.

Ultimately, we will utilize the functions mentioned above to perform linear genetic programming. It first initializes the population and calculates their fitness, then enters the main loop and continuously updates the population according to the predetermined evolutionary strategy. At the end of each generation, the algorithm evaluates the current optimal individual and updates log data to track the performance of the algorithm. This loop continues until the stopping condition is reached, and the most common stopping condition is when the number of iterations reaches the preset upper limit or a satisfactory solution has been found.

4 RESULT

Firstly, we attempt to apply the model to Ant tasks. In Table 4 [4], Through 10 generations, the best fitness increased from 702 to 1005, while the mean fitness also increased from -156 to 956, indicating that the population is evolving. But even with the best individual we can obtain, it can only make the robot stand, not walk or run.

Table 4: Ant Task Fitness

Gen	Best Fitness	Mean Fitness	Worst Fitness
1	702.442	-156.045	-1367.138
2	703.168	648.448	206.737
3	991.376	762.377	530.253
4	995.968	895.901	704.8236
5	998.596	946.7583	232.644
6	999.827	995.5291	989.7627
7	1001.344	977.505	423.842
8	1004.952	989.4449	713.940
9	1003.523	970.7129	707.7175
10	1005.0024	956.181	209.1741

Figure 4 [4] shows the fitness change in 20 generations. There is not much difference between the previous one. The fitness stays around 1000. The 20 generations took 2 hours to run, and the computational cost for this task is extremely high.

Now, we perform our model in the Swimmer task to see if we can get different results. In Figure 5 [5], our fitness has not improved during 20 generations. But inside the environment simulates, the swimmer moves slightly, most of the time forward (right). Considering it's taken only 20 generations, maybe it could achieve a decent

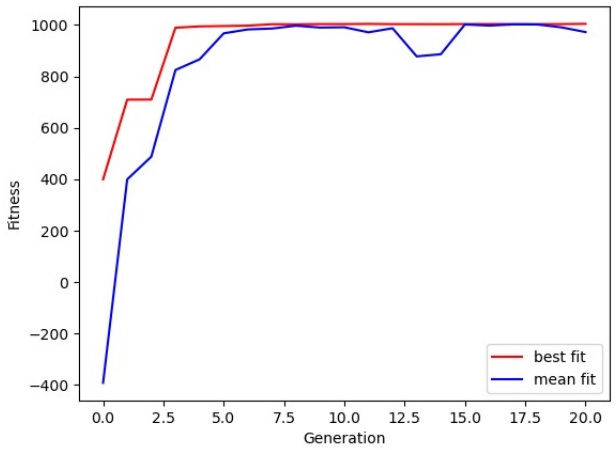

Figure 4: Ant fitness in 20 generations.

result with more generations. The 20 generations took around 20 minutes, slightly better than the ant task.

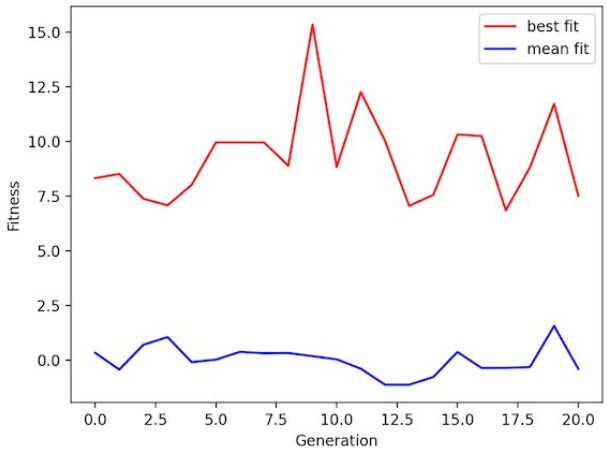

Figure 5: Ant fitness in 20 generations.

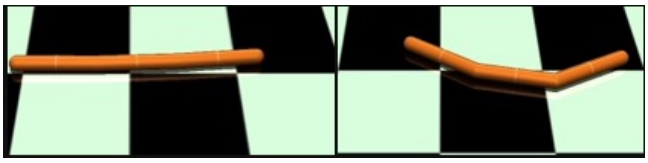

Figure 6: Swimmer Before and Moved.

5 DISCUSSION

After experiencing two environments in Mujoco, we found that the high-dimensional continuous control tasks provided by this environment have a high computational cost, which makes it difficult for us to complete related tasks on personal laptops. However, we are still optimistic about the results obtained, believing that increasing the generation and population sizes will lead to a more accurate result.

Through some literature reviews, we found a small number of studies on ant tasks, with some models achieving a fitness of around 4000, but without providing clear computational costs. But there is a training log [[2]] about Swimmer that states that it tooks 70000 episodes and about 3 days of runtime to train to observe an effective result. Considering that Swimmer is already a relatively task in Mujoco, we can attribute the problem to the high computational cost.

In future work, we gonna consider running tasks on more powerful servers or try implementing pool package in our program to accelerate it.

REFERENCES

[1] Léo Françoso Dal Piccol Sotto, Paul Kaufmann, Timothy Atkinson, Roman Kalkreuth, and Marcio Porto Basgalupp. 2021. Graph representations in genetic programming. *Genetic Programming and Evolvable Machines* 22, 4 (2021), 607–636. https://doi.org/10.1007/s10710-021-09413-9

[2] Matthew Gerber. [n. d.]. RLAI: Reinforcement Learning and Artificial Intelligence. https://matthewgerber.github.io/rlai/. Accessed: yyyy-mm-dd.