Evolution of Artificial Creatures in Stair Climbing

Luna Li
lunaxli22@gmail.com

Kevin Nguyen
knguyen2525@gmail.com

Karthik Vemulapalli
gmkarthik64@gmail.com

Kiran Sivakumar
kiranssivakumar@gmail.com

ABSTRACT

There have previously been works in creating artificial models that evolve using information from the environment in order to become more fit[6]. In this paper, we present a snake-like model that learns to climb stairs over multiple generations using fitness of past generation agents. The paper details the specifications of our simulation, the implementation using a Python library called Panda3D[1], and an analysis of the model's evolution.

1. INTRODUCTION

In the past, many approaches have been made towards creating virtual creatures that evolve [3]. These approaches have used the idea of reinforcement learning to have agents self-learn actions and morphology based on the success of past generations. Reinforcement learning attempts to capture the most relevant and necessary components of an agent. Agents deemed most fit are then studied in hopes of retrieving information about how the agent's evolution can be applied to real life situations.

In this report, we target the popular scenario of stair climbing, which is an activity that robots can find difficult to achieve. By having our agent participate in reinforcement learning, we hope to learn more about the necessary actions required for an AI agent to successfully and efficiently climb stairs. In hopes of simplifying the learning process, We propose a new controlled random based algorithm to teach the agent the best stair climbing method over generations.

The rest of the report is organized as such: we will describe the details of the environment and agent setup. We will then provide details on the reinforcement learning algorithm used to have the agent self-learn. We will then present the results of the agent's learning process.

2. SYSTEM OVERVIEW

2.1 Panda3D

Panda3D is an open source 3D game engine for Python. The graphics api uses modern OpenGL/DirectX features and it supports ODE, Bullet and PhysX physics engines among other features. The main reason we chose Panda3D was because of its 3D physics capabilities as well as its Python support[1]. In addition, we wanted to attempt to build the simulation and learning process from the ground up in order to exercise greater control over parameters.

2.2 Environment

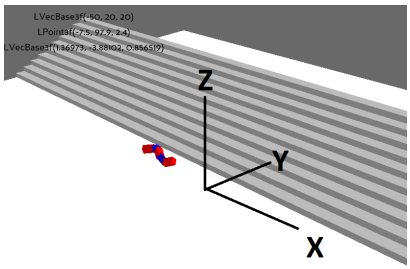

Figure 1: Simulation Environment

Figure 1 shows our simulation environment. The environment is made up of three main components: the 10 step staircase, the agent and the ground. The staircase has a mass that is arbitrarily large (much larger than the mass of the agent) to eliminate the chance for the agents to push the staircase over. Currently each step of the stairs is 0.5 units tall and 5 units wide. There are a total of two light sources, one ambient and one directional (both white light) added to make the environment seem natural.

Our environment is built using the Panda3D library and includes several typical features of a physical simulation environment. The first is the ground surface parameters which include: Coulomb friction coefficient, indicating the magnitude of friction forces of the surface; bounce, indicating the bounciness of the surface; bounce velocity, minimum velocity in order to bounce; soft-erp and soft-cfm, to simulate soft surfaces; slip and dampen, to simulate slipping and dampening effects.

Our environment also has collision detection between the agent and its surrounding environment. Each of the objects in our world: stairs, ground and agents have their own box geometry that is used to detect collisions, in addition to its own model.

2.3 Agent

Our agent is made up of five rectangular blocks with a width and height of 1 unit, and a length of 2 units arranged in a sequential manner. Figure 3 shows our agent in the Panda3D environment. Even blocks are colored blue and odd blocks are colored red in order to easily distinguish sequential blocks.

Each limb is connected by an ODE universal joint. This joint has 2 axes, which in our case are the x and y axes. This allows our agent's limbs to twist and bend effectively model-

Figure 2: Swivel Knuckle Joint [2]

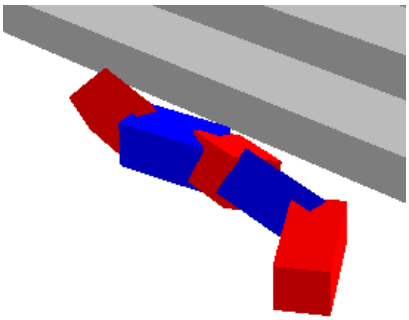

Figure 3: 5 Limbed Agent

ing a swivel knuckle joint shown in Figure 2. Each joint also has several parameters that can be modified such as torque applied on each axis and maximum angle of rotation allowed for each axis. Another parameter we considered is constraint force mixing, which is essentially how "soft" the agent is or how susceptible it is to gaining traction or slipping. Each of these parameters are accounted for in our learning process.

In addition the joint properties, the limbs are also assigned masses determined by the density of the material which we take into account during the learning process.

2.4 Fitness

Our fitness metric is calculated simply by measuring the distance the center of the agent traveled in the positive y direction (towards the stairs). We considered using the head of the agent as well as the center of mass but ultimately settled on the center as this would allow us to reward the agent for getting more of its body up the stairs. We also considered using the height (z displacement) to determine how many stairs were climbed but decided against it because it might negatively skew the agent to perform vertical jumps instead of moving up the stairs.

3. LEARNING IMPLEMENTATION

Our agent moves on the basis of pseudo-random generated waves and other randomly selected parameters. The waves represent the torques that are applied to each of the agent's joints per frame of the simulation. Agent evolution is formed on the basis of improving previous waves to converge towards the torque forces that generate the greatest fitness while

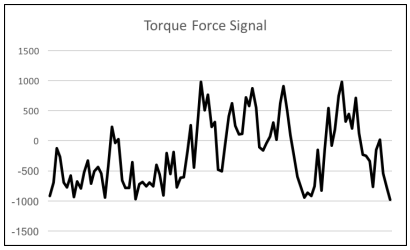

Figure 4: A torque force signal generated to control torques applied to the agent's limbs during a simulation.

also randomly selecting values for other parameters to get optimal use from the waves.

3.1 Initial Generation

To generate the initial waves, we use a point by point pseudo-random method. An initial starting torque x is chosen randomly from within a range. The next point or torque in the wave is generated by taking the previous point and picking a random value within a certain range of the previous point. The number of points generated is based on the length and fps of the simulation. For example, given an initial value of 5 and a range of 10, the next point in the wave can range from -5 to 15. For our simulations, the minimum and maximum torque values are defined as -1000 and 1000 and each next value may differ by at most 1000 from the previous. Figure 4 depicts a signal generated in this fashion.

The other parameters we worked with were specifying constraints on how much rotation specific joints could perform. For the twisting motion of a joint (similar to the motion made when wringing a sponge) we chose from four random values, specifically 0, 0.9, 1.2, and 1.5 in the code. This allowed us to have more types of agent actions with the same sets of signals. Similarly, we allowed for constraints on the bending angle (similar to the motion made when snapping a stick) which again chose from four values of 0.6, 0.8, 1.0, and 1.8.

3.2 Generational Learning

From the previous generation, the agent's signals with the greatest fitness are chosen. These best signals are then slightly modified for the next generation by taking each torque point in the signal and shifting the point within a certain range. Over each generation, the range in which these torque values can mutate becomes smaller, eventually leading to convergence. Simulations for the next generation are then run using these new signals. For our specific simulations, we used the best four agents from each generation to generate the signals for the next generation. Between each generation, we decreased how much the signals could differ from the previous. In the second generation, the value at each position on the signal could decrease by at most 100 from the first generation's. For each generation after, this value was halved, so for example the third generation values could differ by at most 50 from the second generation values.

4. RESULTS AND DISCUSSION

Using the parameters and approach from Section 3 and over 500-1000 simulations per generation, we produce the results shown in Table 1.

Generation	Fitness Value	Meaning
1	-7.23	Starting Position
2	-5.94	Slightly Forward
3	-2.05	Reaches Stairs
4	3.84	Climbs 1 Step
5	5.79	Climbs 2 Steps
6	8.05	Climbs 3 Steps

Table 1: Fitness values of agent across multiple generations.

The fitness values in the table refer to the average y-value reached by the agent in each generation across all simulations. The agent starts a few units from the staircase and a value of -2 approximately corresponds to reaching the stairs. From the results, it is clear that on average the agent improves across generations. This works as expected since each generation ignores the worst values from the previous ones and converges on the best climbing signals.

Figures 5 to 10 display a resulting simulation from each progressive generation. After generation 6, there was not much improvement in climbing ability. This is likely because the changes in signals between generations became too small and the resulting agent could not get much better than generation 6. This could possibly be fixed by allowing larger variation between generations.

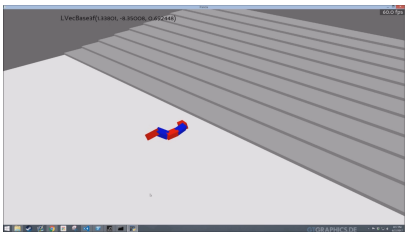

Figure 5: Result from generation 1.

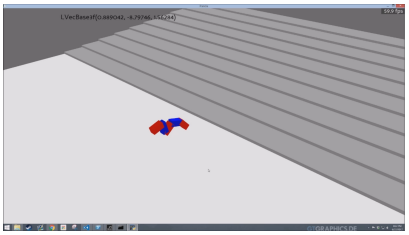

Figure 6: Result from generation 2.

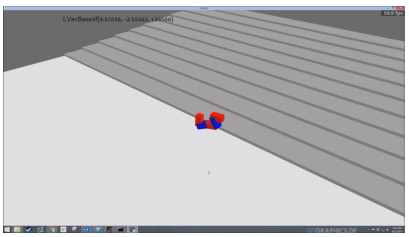

Figure 7: Result from generation 3.

5. RELATED WORK

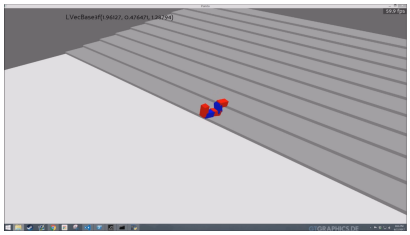

Figure 8: Result from generation 4.

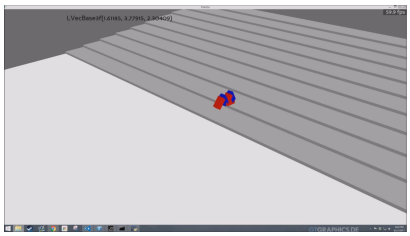

Figure 9: Result from generation 5.

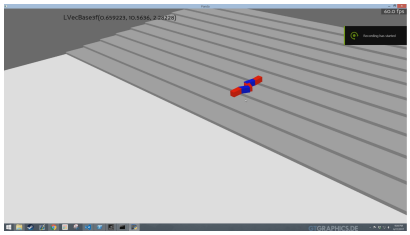

Figure 10: Result from generation 6.

Past works have focused on a variety of ways to accomplish reinforcement learning. Watkins and Dayan [7] propose Q-learning, a form of model-free reinforcement learning. Q-learning uses a table lookup approach to store the outcomes of actions during certain states. By trying all actions and states given the rewards and penalties of a scenario, the agent learns what action should be taken given a certain state to achieve the greatest fitness.

Ramos, et al [4] use a Learning Classifier System (LCS) that is another form of a Q-learning system. They focus on developing human like characters with autonomous social behavior in a video game scenario to better understand social interactions. These agents use a LCS to exhibit behavior that is classified as autonomous, reactive, proactive, or social based on environmental stimuli, resulting in certain actions being taken. The agents are given autonomy and adaptability via a genetic algorithm embedded in the LCS.

Lassabe, Luga, and Duthen [6] also use a LCS, but move away from social interactions between human-like agents towards simpler scenarios. Their agents consist of limbs and joints that are mutated per generation based on periodic wave inputs and fitness. Certain wave inputs are used based on environmental stimuli that pass through the LCS and activate articulations correctly. The best waves are appended together to activate next generation agents.

Miconi and Channon [5] propose a more complex approach that re-implements the work of Karl Sims[3]. Their model is similar to Sims's but with some substantial changes. These include the use of standard neural networks to control their

agents and tweaks to the genetic algorithm to allow independent wiring amongst newly created components upon mutation. With these works as background, we were able to understand the necessary components needed for a basic reinforcement learning system.

6. CONCLUSION

This project attempts to have an agent self-learn how to climb stairs. This is done by generating pseudo-random signals that activate the agent's joints by applying torque forces based on signal strength. The best signals are used to learn the next signals to use for the next generation. Our method attempted to stress simplicity in its design and resulted in moderate success as the agent was able to learn the best actions to climb stairs within a few generations. This project's simple methodology leaves room for much improvement in the future.

7. ISSUES

When working with Panda3D, there were a few problems that arose when running simulations. The first major issue was lag due to external factors affecting the simulation. The main reason lag was a problem is that our model would perform actions on a per frame basis, and by having frames take longer periods of time, the forces could affect the actors for a larger period of time during a frame and cause strange behavior such as gaining an extreme amount of momentum in a direction. One of the potential sources of lag were print statements to the UI screen which were used for diagnostics when first setting up the simulation. After removing these text printouts every frame, the lag severely diminished. Another cause of lag was moving the window or interacting with the simulation in other ways while running. This was easy to fix as it simply meant leaving the simulation alone during the runtime. Finally, external running processes could spontaneously cause lag. This problem could not actually be solved and the only real workaround was to run an iteration with the same parameter multiple times and ignore outliers.

Our second issue, which was also tied to the first of lag, was non-deterministic results of simulations when given the same parameters (specifically force values). This was a major issue because our proposed learning process would be ineffective as our simulations might not actually be useful. In our first attempts when we ignored the non-determinism issue. The future generations would often be just as bad as the first generation and actually climbing stairs. Once the efforts above were used to remove lag, simulations with the same parameters would run more similarly to one another, but not perfectly the same. Another possible reason for the non-deterministic behavior might be difference in hardware between machines or hidden optimizations performed on the back-end. This was seen when the same simulations run on separate machines would often give very different results even with a stable framerate. As a result, we had to run all our simulations on a single machine which slowed the overall iteration and learning process as we could not process signals on multiple computers.

Finally, we had many issues setting up the environment before eventually reaching one that made sense. For example, the actual surface had many parameters to set such as friction, bounciness, slip and dampening. Finding the best parameters were necessary to actually have working simulations; for example, heavy friction or extremely low friction would make it impossible for the agent to move. There was no easy way to determine the best values, so we mainly ran simple simulations many times while tuning parameters by eye. Another problem was the parameters we passed initially for the simulation were too simple to have proper learning. When we only used a list of forces, many of the agents would act similarly to one another and simply shake back and forth while some would properly climb by chance. We fixed this by using more parameters to learn on such as constraints on how much a joint could twist or turn and actually adding the number of forces that would take effect per second. Compared, to the method of using simpler parameters, we actually saw visual improvement between generations and the learning process led to agents that moved more specifically in the correct direction as opposed to unnecessarily moving to the side or even backwards before actually switching to the uphill direction.

8. FUTURE WORK

Our learning simulation leaves many avenues for future improvements and considerations. For one, our current work does not mutate the agent's physical properties. For future work we would like to include physical mutations to see how that might affect our results such as the addition and removal of limbs and the changing of limb dimensions. Along the same line, modifying the surfaces of the agent may also produce interesting results, for instance, by making the agent's limbs squishy, it might find some other interesting methods of climbing. Another feature we would like to try is to add an energy penalty to generate agents that climb stairs in an efficient manner. Currently we limit the torque by generating signals within a certain range, however, adding an energy penalty would be more realistic since real-world creatures must learn to accomplish tasks without expending too many resources. We are also interested in modifying our genetic algorithm in hopes of achieving faster convergence. Lastly, we hope to alter and account more for the environment whether this be by mutating the shape of the stairs or having the agent react to environmental stimuli.

9. REFERENCES

[1] Panda3D. https://www.panda3d.org/.

[2] Swivel Knuckle Joint. http://rakumba.com.au/lighting-components/assembly-parts/swivel-knuckle-joint-chrome/.

[3] Karl Sims. Evolving Virtual Creatures. *Computer Graphics (Siggraph '94 Proceedings)*, pages 15–22, 1994.

[4] Marco A. Ramos, et al. Evolutive Autonomous Behaviors for Agents System in Serious Games. 2015. IEEE.

[5] Miconi, Thomas and Channnon, Alastair. An Improved System for Artificial Creatures Evolution. 2014.

[6] Nicoloas Lasbe, Herve Luga and Yves Duthen. A New Step for Artificial Creatures. 2007. IEEE.

[7] Watkins, J.C.H. Christopher and Dayan, Peter. Q-Learning. *Machine Learning, 8*, pages 279–292, 1992.