Project: Autonomous Softbody Locomotion

by Raphael Yamamoto

Abstract

My project aimed to teach locomotion to simulated soft robots equipped with neural networks. I used neuroevolution—specifically the NEAT algorithm—as my training method, with network fitness being determined by the distance traveled by a soft body. After my initial attempts failed due to the complexity of the problem, I decided to use RNNs, and I gave the network control over groups of springs instead of each one individually. This reduced the complexity of the networks, lowered the training time, and finally allowed the soft bodies to move themselves reliably. I then experimented with using various different body shapes and trying to introduce directional control to varying degrees of success. Overall, I found that neuroevolution and RNNs were an effective approach to learned softbody locomotion, although of course trying to exhibit more complex movement requires more complex networks and more rigorous training.

1. Introduction

1.2 Level of Autonomy

In most cases I would rank this level 4 or above, since learning is unsupervised and no human interaction occurs as the models are moving themselves. In the case where I attempted to implement directional control, however, I am not entirely sure, as a human dictates the direction the body moves, but all locomotion is performed by the network. This might place it at around level 1.

1.3 Description

My project consisted of training neural networks to control the locomotion of soft robots. To achieve this, I simulated soft bodies using the spring-mass model (simply a lattice of masses connected by springs) in PyBox2D and trained their networks using neuroevolution. I chose the spring-mass model mostly due to its simplicity and accessibility, which allowed me to implement and manipulate the soft bodies myself. Similarly, I chose neuroevolution because it is an accessible form of reinforcement learning, and the physics engine I am using is not differentiable—and thus any evaluation of a network's performance wouldn't be either. Specifically, the neuroevolution algorithm I chose was the NEAT (NeuroEvolution of Augmenting Topologies) algorithm, which evolves both network weights and topology (during each mutation phase, there is a chance a network will add a new node or a new connection). I chose this because I was interested in its approach, which promises to develop minimal network architectures for a given problem.

Initially, it difficult to get the physics engine to work properly with it's provided testbed. After some struggling, I ended up needing to control the specific version of each piece of software with a conda environment for it to work. Once working, I implemented soft bodies and a custom Environment class that would allow me to run the same setup with and without the visualization system. I then placed these into a Github repository which I could clone into Colab for training.

My first attempt at training consisted of using a feed forward neural network to move a robot with 33 masses and 160 springs. I gave the network individual control over each spring, feeding each spring "length" to the input nodes and applying the outputs to each spring on the next timestep. Fitness was determined by the horizontal displacement of the softbody. Naturally, this resulted in robots that only moved once and stayed still afterwards. Thus, my next attempt used RNNs to give the robots the ability to place themselves within time. This time, the robots exhibited continuous movement, but it was inconsistent and uncoordinated. The robots did not show any pattern in their actions, nor reliable lateral movement.

The breakthrough ocurred when I grouped springs into makeshift "muscles," allowing the network to control larger portions of the softbody at once. Alongside reducing the complexity of the problem, this also reduced the size of the networks, which I believe finally allowed them to learn the proper behavior in the training time I gave them. This time, the robots were able to consistently move themselves efficiently and elegantly.

I then tried to experiment with introducing directional control. To do this, I added an extra input node to the networks, which I would set to 1 if I wanted the network to move to the right and -1 if I wanted it to move to the left. This was notoriously difficult, which I assume was again due to the complexity of the problem and the limited training time and network size. My best model would switch directions if prompted to, but it moved very slowly and was unreliable.

Similarly, I tried experimenting with different body shapes. For the most part, they were able to learn to walk pretty effectively. However, one shape in particular—the "L" shape—didn't at all. In fact, it just curled up once and stopped moving, quite similarly to the initial feed forward model. I wasn't able to figure out why this was, and the behavior persisted despite multiple retrainings.

2. Related Work

Inspiration:

https://www.youtube.com/watch?v=CXTZHHQ7ZiQ

https://www.youtube.com/watch?v=z9ptOeByLA4

https://www.youtube.com/watch?v=HgWQ-gPIvt4

NEAT:

http://nn.cs.utexas.edu/downloads/papers/stanley.ec02.pdf

https://neat-python.readthedocs.io/en/latest/neat_overview.html

https://towardsdatascience.com/neat-an-awesome-approach-to-neuroevolution-3eca5cc7930f

3. Team Organization

3.1 Team Members and Roles

I worked on this project by myself.

4. Software and Developing tools

4.1 Software

Programming/Training environment

- Python 3.8.10.
- Anaconda, for running the testing/visualization.
- Google Colab, for training.

External Modules

- Pybox2D, a python port of Box2D, originally written in C++.
- NEAT-Python, a pure python implementation of the NEAT algorithm.
- Pygame, a graphics engine for visualizing the environment and running tests.

4.2 Laptop/Desktop setup

A MacBook Pro running macOS Big Sur 11.1 with a 2.5 GHz Dual-Core Intel Core i7 processor, 8 GB RAM, and an Intel Iris Plus Graphics 640 1536 MB graphics card.

Because all training was done on Colab, I didn't in into any issues with performance. Even without much of a GPU, running the pygame visualizations with the NEAT-Python neural networks wasn't an issue. (I had a consistent 60 fps when running the simulations.)

4.3 Hardware needed

No hardware was necessary.

4.4 Simulator needed

Pybox2D and Pygame.

5. List of Milestones

- Set up the PyBox2D TestBed with Pygame.
- Wrote a SoftBody class to simulate soft bodies in PyBox2D.
- Wrote a few modules to allow the same Box2D environment to be run with and without the testbed. This allowed me to train and test in the same environments.
- Tweaked the SoftBody class to accept a matrix as a body shape.
- Trained the first models with Feed Forward Networks in Colab.
- Trained models with Recurrent Networks.
- Added code to the SoftBody class to allow for muscle groupings.
- Trained models with Recurrent Networks and muscle groupings.

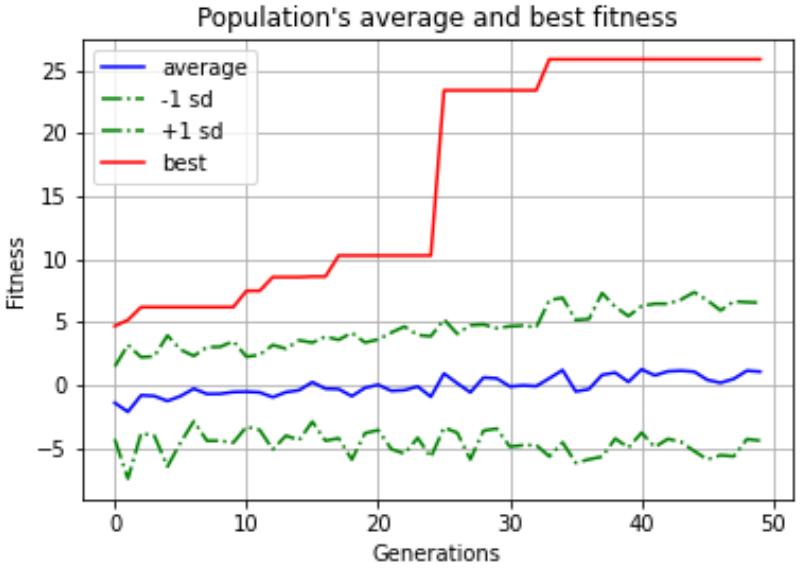
- Experimented further with different shapes and directional control.

6. Results and Discussion

As explained above, the robots were not able to learn to move themselves when given control and awareness of each spring in their body. Often, they did attempt to maximize their fitness by moving to some degree initially, but their motion wasn't continuous or reliable.

However, using both RNNs and muscle groupings, the soft robots were able to learn to reliably move themselves. The most performant models also exhibited effective running cycles that were quite fast (faster than I could have programmed them to run). I wouldn't be able to calculate an exact speed—as I don't think the units of the simulation correspond to real-life units—but the fastest robot could move at around 0.26 body lengths per second.

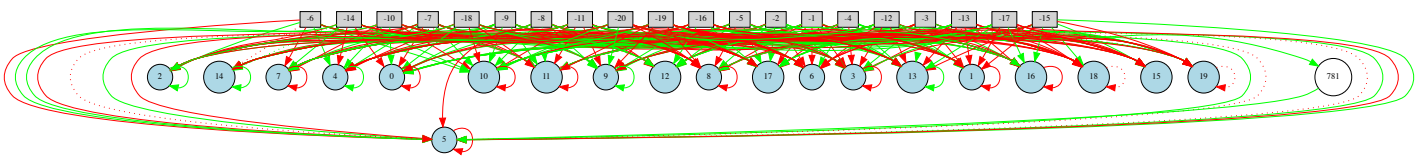
Due to the nature of neuroevolution, most graphs of the fitness looked like the one below (taken from one of my attempts to introduce directional control).



Because improvement relies on random mutation, better models lead to distinct jumps in best fitness, which stays the same until a better model appears. Similarly, the average fitness tends to get bogged down by underperforming mutations and not display much change despite the best models performing increasingly well.

With muscle grouping, training time also decreased immensely. Whereas two hours of training in the first models yielded poor results, the models with muscle groupings reliably learned to move in around four minutes. I assume this was because the networks were smaller.

Below is a representation of a recurrent neural network trained on a softbody with muscle groupings.



Interestingly, the network didn't need to add many hidden nodes to move effectively (even the one that it did add is weighted very lightly), and using the recurrent connections was sufficient.

7. Conclusions

A major issue throughout the project was the complexity of the problems, and thus the complexity of the networks needed. With too granular a control in the initial models (160 inputs and 160 outputs with no hidden nodes), the desired behavior would have taken much longer to train. However, simpler models allowed the networks to learn easily. This might also have been a side effect of using NEAT, as its prioritization of minimal architecture could have killed off any larger networks before they had a chance to develop properly.

These struggles with complexity highlight the unpredictability and power of neuroevolution. Often, when something wasn't working it was difficult to pinpoint where the problem was. (Had it not been trained for long enough? Were my hyperparameters off?) However, when the algorithm worked the results were quite effective.

8. Future Work

In order to combat the issues with complexity, I might experiment with other neuroevolution and reinforcement learning algorithms. For example, I might try to solve the more complex problem of directional control with an algorithm that allow for hidden nodes. I am also interested in trying to teach the robots to navigate through more difficult areas, such as through liquid or over rough terrain, as well as to perform more difficult actions like jumping. One might also take the project to its full extent and implement it in a three dimensional simulation, with some form of computer vision as input to the network.

9. References

http://nn.cs.utexas.edu/downloads/papers/stanley.ec02.pdf

https://github.com/pybox2d/pybox2d

https://neat-python.readthedocs.io/en/latest/