Project 3: Ludobots

James Gaskell, Jonathan Fischman

I.. Introduction

Josh Bongard's Ludobots tutorial provides a nice entry into the world of evolutionary robotics simulation. Since it is extremely expensive and time-consuming to work with real-world evolutionary robots, a common practice is to view behaviors in simulated environments then attempt to replicate this in the real world.

II.. Methods

In this project, we followed Josh Bongard's Ludobots tutorial [1] on Reddit, through Part K: Random Search.

A. Background and Installations

In parts A and B we were introduced to the physics engine pybullet, which is used to create a real world environment in which we can simulate our robots. Pybullet uses a step simulation to update the current state of objects within the simulation and to update the GUI (Graphical User Interface) accordingly. To maintain dependencies for other projects, we installed pybullet, and the many other packages this project required, in a virtual environment.

In Part C, we learned how to create a 3D object using pyrosim [2] - a package created by Bongard to write the objects and behaviors to URDF files which can be used by pybullet. Upon generating the URDFs and simulating the environment we were able to see a single cube which we had created and placed using pyrosim. The cube we created will act as one body segment (referred to as a link) of the robot.

B. Simulation and Setup

In Part D, we added gravity to our pybullet simulation. We also added a plane, to act as a floor and stop the cube from continuing to fall due to gravity. We also created more cubes, and learned how to manipulate the position the generated in. Interestingly, the more cubes we added to the environment, the slower our simulation was due to the number of calculations occurring within the simulation loop. To fix this we increased the number of iterations within the loop and decreased the time sleep that originally slowed the simulation down - it is worth noting that the simulation ran at different speeds on the different devices we used, so these values must be altered when recreating the results.

In Part E, we added joints to our Ludobot, creating connections between adjacent links. Whilst this seemed easy at first glance, the joint co-ordinates in pybullet are all relative to the first joint loaded, so it took a lot of effort to imagine and visualize the necessary co-ordinates. The joints allow the two leg links to rotate around their shared edge with the torso. The resulting robot is shown in Figure 1

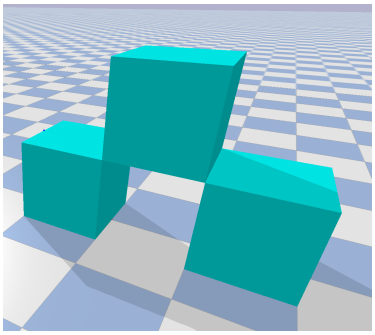

Fig. 1: Simple 3 link robot with joints

In Part F, we added touch sensors to the Ludobot's legs, allowing us to determine if the leg is touching the floor at a given moment. When we tested the robot by manually picking it up and placing it down the sensor array became populated with values 1 and -1. A value of 1 means the leg is toughing the floor, a value of -1 means the leg is in the air - this will make it easier to graph the robots gait later in the project.

C. Movement

In Part G, we added motors to the joints of the Ludobot, finally allowing it to move. The Ludobot's movement was based on a vectorized sine function created over the duration of the simulation loop with one value for every iteration. These values are the target angles of the motors in relation to the joints, and when the target angle is changed in the loop this provides the movement behavior. The sine functions use a set amplitude, frequency and phase offset to create the movement vector - to begin with these were both set equal for the two legs but we later investigated how changing these values altered the movement and gait.

D. Refactoring

In Part H, we refactored our code, following object-oriented programming practices. Whilst before, we were thinking about the robot as a collection of blocks, creating a robot object really helped to bring the project together. After the refactoring we have a simulation object, which has a robot, and the robot has motors and sensors - following a more real-world design paradigm. This helped massively with organization, and helped clean the simulation loop into just a few lines of code since the robot's methods were within classes.

E. Neural Networks

In Parts I and J, we added neurons - allowing our robot to ”think” and interpret the touch sensor data, and synapses - connecting the neurons to the motors and allowing the Ludobot to traverse the plane on its own. The neurons start out with a random seed - i.e. a random first target angle, and learn from the sensors to figure out how the object is moving. This learning process means the n^{th} motor value is informed by the n-1 values before it.

Finally, in Part K, we implemented a random search program, allowing us to generate and simulate robots with different sets of synaptic weights, resulting in differing behaviors. This program automates the generation process, which is responsible for creating the neural network with a random seed, and the simulation process which would lend itself well to an evolutionary process to work out the best motor values for movement. Running the automation program creates two simulations back-to-back to exemplify these different behaviors.

III.. Results

The discussion of results is separated into two sections. The first section discusses the Ludobot's behavior when it is controlled by a sine wave. The second section describes the Ludobot's behavior when it is controlled by the neural network.

A. Results after Part H

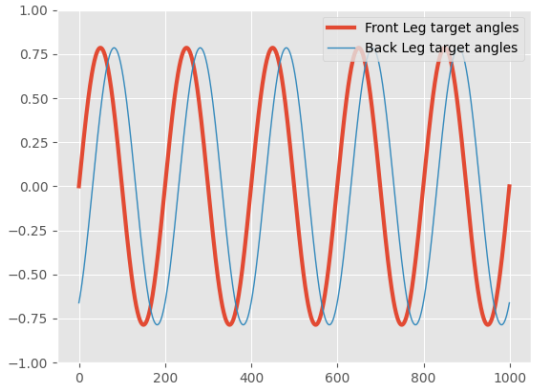

Fig. 2: Motor Target Angles for Sinusoidal Motion

After completing the refactoring in Part H, we ran the simulation, collected data about the Ludobot's motor target angles and sensor values, and graphed the results. At this point, the Ludobot's movement was being controlled by a sine function. This can be seen in Figure 2, where the motor target angle values for both the front and back legs follow vectorized sinusoidal curves and the sinusoidal curves are constructed from a set frequency, amplitude, and phase offset for each motor. The graph in Figure 2 shows the two motors with the same frequency and amplitude, but with different offsets hence the shift between the front and back leg motors. Whilst it is not possible to account for every combination of values manually, we found the robot behaved the best with the same amplitude and frequency for both legs, but introducing a phase offset propelled the robot forward quicker. The offset meant both legs were working in the same direction, but the delayed leg was being pulled forward somewhat by the movement of the first leg.

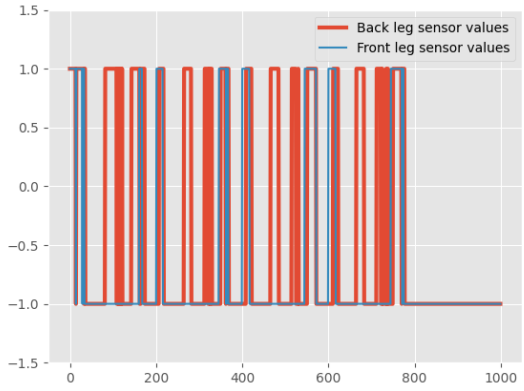

Fig. 3: Sensor Values for Sinusoidal Motion

In Figure 3, we see the values of the touch sensors on both the front and back legs, which seem to differ greatly. This indicates that the front and back legs were usually not touching the ground at the same time while the Ludobot was walking. Again, the behavior shown is for both motors with the same amplitude and frequency but a slight offset. Notably, the blue line showing the front leg's sensor values is less prolific than the red - this exemplifies the same dragging behavior we witnessed in the simulation as the front leg was leaving the ground less often.

B. Results after Part K

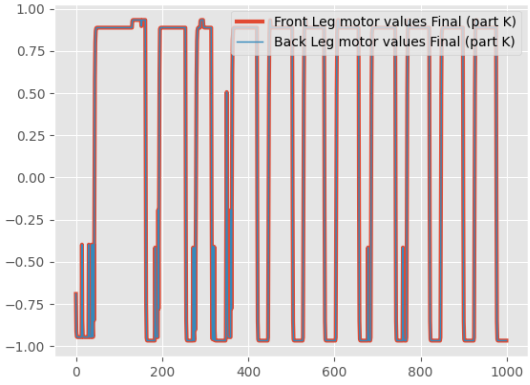

Fig. 4: Motor Target Angles for Part K

After completing Part K, we ran the simulation, collected data, and graphed the results. The Ludobot's movement

was no longer controlled by the sine function. Instead, the movement was controlled entirely by the robot's neurons and synapses. This resulted in very different behavior compared to the trial in Part H. The resulting graph does not resemble the sine wave in Figure 2.

In Figure 4, the motor target angle values match almost perfectly for the front and back legs. This indicates that the neurons and synapses controlling were giving the same instructions to both legs.

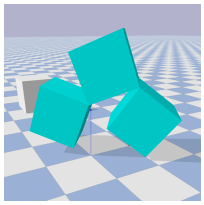

Fig. 5: Left Leaning Ludobot

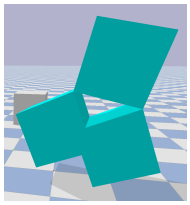

Fig. 6: Right Leaning Tiptoes Ludobot

We also noticed that the starting position of the legs seemed to matter greatly. Since the neurons gained their starting values from a random distribution between 1 and -1, on any run of the simulation the legs did not start in the same position. That said, even though the neurons changed the movement of the legs over time, they did not vary greatly from their starting positions. The robot shown in Figure 5 started with a left lean which was propagated throughout the whole simulation, whilst the robot shown in Figure 6 developed a right lean and stood on its tip-toes and the motor updates through the neural network were not able to fix this.

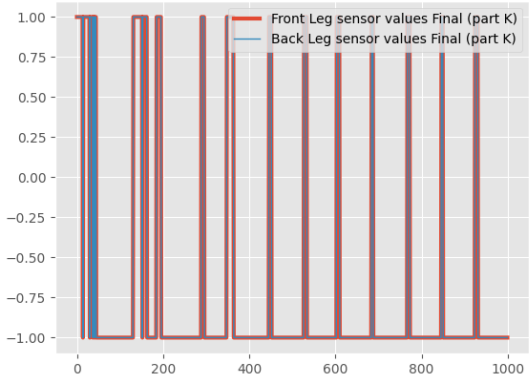

Fig. 7: Sensor Values for Part K

The data graphed in Figure 7 shows that the touch sensor values for the front and back legs were the. This indicates that the legs were moving in unison, as they both touched the ground at the same time, and stopped touching the ground simultaneously. The behavior also changes most at the start and then seems to settle into a periodic signal which makes sense as the neural network learns over time.

IV.. Conclusion

We completed the Ludobots project up to part K. The robot's behavior followed the expected outcomes and would work very well with a genetic algorithm of random mutation hillclimber going forward. Since we can judge the fitness of a robot by its ability to move, and throughout the project up to part K this was what we hoped to achieve, with a big enough population and enough trials we could definitely find out the best motor values vector across the simulation loop to maximize movement in any direction.

References

[1] J. Bongard, "Ludobots Tutorial," https://www.reddit.com/r/ludobots/, 2013, [Accessed 20-11-2024].

[2] ——, "Pyrosim respository," https://github.com/jbongard/pyrosim, 2013, [Accessed 20-11-2024].