Creating New Artificial Life

by

Cara Reedy

A thesis submitted to the Faculty of the University of Delaware in partial fulfillment of the requirements for the degree of Bachelor of Science in Cognitive Science with Distinction

Spring 2015

Creating New Artificial Life

by

Cara Reedy

Approved: ____

Daniel Chester, Ph.D
Professor in charge of thesis on behalf of the Advisory Committee

Approved: ____

Kathleen McCoy, Ph.D
Committee member from the Department of Computer and Information Sciences

Approved: ____

Nancy Getchell, Ph.D
Committee member from the Board of Senior Thesis Readers

Approved: ____

Michelle Provost-Craig, Ph.D.
Chair of the University Committee on Student and Faculty Honors

ACKNOWLEDGMENTS

My advisor, Dr. Daniel Chester, was instrumental in helping me design and complete this project. He also advised me during independent study work that led to the development of this thesis project. I am grateful to the Summer Scholars Program for funding a summer of full-time research on this project. My father, Dr. Terry J. Reedy, a member of the Python Core Developers group, helped me with problems and issues related to Python.

TABLE OF CONTENTS

LIST OF FIGURES

ABSTRACT

Artificial life is the simulation of life or life processes. There are many kinds of artificial life programs available, but none that were exactly what I was looking for, so I decided to program my own. Using Python, I wrote a simulation that contains agents, which have neural networks as their brains and can learn, as well as genetics that define the neural networks. They also have needs such as hunger and can starve to death. The program shows a visualization of the agents and their environment, and it can also show their brain structure and live activity. It can save the network structure of any particular agent to a text file and later read it back in, making it suitable for running experiments. These agents are simple, but the program is an expandable platform for artificial life experiments, with little that is hardcoded. The code is freely provided so that anyone can both look at the code and modify it for their own interests.

Chapter 1

INTRODUCTION

I first grew interested in artificial life when I experimented with a series of computer programs called *Creatures*. The first version of *Creatures* was released in 1996; the latest version was released in 2001. In each version of the program, the user sees a simulated world full of different objects, such as plants and toys. This world is inhabited by the titular creatures, which age, breed with each other, and through disease, accident, old age, or even genetic weakness, eventually die. Although I had seen the artificial intelligences used in games and virtual pet programs before, the creatures of *Creatures* were different. Each creature has its own genetics that determine how it processes food, how its immune system works, and even how its brain is structured. Each creature's brain is an artificial neural network which allows it to learn over time, both through its experiences and through interaction with the human user. Over generations, new mutations can pop up in the genetics of new creatures, and the population will change. This was a much more complex simulation than anything I had ever seen before, and it was a sandbox that let me perform experiments easily. Tools that were used in developing the game were also made available for free to users. These tools let the user modify the genetics of any creature, to view the structure and state of a creature's brain and watch it respond to input, or to write small programs that could be run in the *Creatures* environment.

In the Spring 2014 semester, under Dr. Chester's advisement, I used these tools and what information I could find on the program's workings, and specifically on how the artificial neural networks in the program work, to try to create creatures that showed migratory behavior. I eventually had some success, and on the whole I was impressed with the work behind *Creatures* and some of the ideas it had. However, I also experienced some frustration with how the *Creatures* program worked and the lack of information available about certain aspects of the brain simulation.

There are many other artificial life simulations besides *Creatures*. However, none were suitable for the kinds of experiments I was interested in. The creatures of *AI.Planet*, which is a program meant to simulate ecology, are too simple for what I wanted to do; most of them do not learn, and those that do can only learn very narrow things (Kerr & Murczak). Some, like *Avida*, have different aims than I do in that they do not necessarily care about being biologically realistic. *Avida* is a simulation where computer programs themselves evolve ("Avida"). Many artificial life programs, such as *EcoSim*, are not available for download, and thus not accessible by the general public (Gras). Because of these reasons, and because there were some specific shortcomings with *Creatures* that I wanted to be able to address, I decided to write my own artificial life program.

I used Python to program my simulation, creating an environment with objects and creatures that interact with their environment. The creatures have simulated brains made of artificial neural networks and learn from their experiences. They also have genetics, and can thus evolve over time. This work is presented in the following three chapters. In Chapter 2, I present background information about artificial life and artificial neural networks that informed my design of the project. In Chapter 3, I describe the actual simulation itself in detail, including how I used the PyBrain neural network library as a basis for my project and how the genetics and neural networks work. Chapter 4 provides some conclusions and discusses future directions for this program. The link to the actual code itself as well as the instructions for running it can be found in the appendix.

Chapter 2

BACKGROUND

Artificial life simulations are those that seek to emulate life or life processes. Artificial life can be used to test hypotheses about life that might be difficult or impossible to replicate in the real world. For example, a program that is used to study the evolution of predator and prey dynamics can run tens of thousands of generations of simulated lifeforms in a short amount of time, each of which can have extensive data collected automatically – a task which is impossible in real life. Artificial life can also be used to observe and study new forms of life, some of which may not even be possible to see on Earth, or which could only be possible on computers (Adamatzky and Komosinski, 2005). Though these programs may not be alive by the usual meaning of the word, their dynamics and evolution can still be interesting to observe. An early form of artificial life, Thomas Ray's *Tierra*, consisted of 'organisms' that were just small computer programs that required memory space in which to 'live' and processing time to run and a function that killed off old or non-functional programs. As documented in chapter 15 of *Out of Control, Tierra* soon evolved not just programs that were smaller and more efficient than the initial program that was introduced to the memory space, but also parasitic programs that needed other, larger programs to replicate themselves, hyper-parasites that needed the parasitic programs, and even hyper-hyper-parasites (Kelly, 1994).

The term artificial life covers quite a variety of simulations, from small and simple, to extremely complex, to simulation of thousands of digital organisms. On the most simple level, cellular automata are often considered to be a form of artificial life (Grand, 2001). The most famous example of cellular automata is perhaps Conway's Game of Life. In the Game of Life, the world consists of many square cells that can be either 'alive' or 'dead', and simple rules govern whether a cell will be alive or dead in the next time step. From this simple basis, an astounding number of patterns can emerge, including ones that seem to move across the grid or which induce the creation of other moving patterns (Grand, 2001). On the other end of the scale, there is *OpenWorm*, a project to simulate the nematode *Caenorhabditis elegans* in extremely fine detail. *C. elegans* has one of the simplest nervous systems known, making it an important object of study for understanding the basics of how neurons work and connect together. It is also used as a model organism for a number of topics of study, as it is a simple multicellular organism that still needs to find food and mates while avoiding being eaten. The people working on *OpenWorm* hope that it will lead to a better understanding of the biology of *C. elegans*, which would lead to a better understanding of biology as a whole ("Getting Started").

Creatures is on a different level than many artificial life programs. Programs meant to study ecology might simulate a large number of relatively simple creatures or agents at a time. Programs like *OpenWorm* are highly detailed, but might only simulate one or perhaps a few agents at any one time. However, what I call mid-level simulations have anywhere from a dozen to perhaps a few hundred relatively complex (but not extremely fine-detailed – for example, not including details of biochemical interaction) agents in the world at any one time, each with its own 'brain' and details such as body layout or biochemistry. The creatures of the last version of *Creatures* have about 900 genes each and 1000 neurons in their artificial brains, and on a modern computer the program can run several hundred of these creatures at a time with no slowdown. Each creature's genome is haploid; that is, they only have one copy of each gene, unlike real-life animals, which generally are diploid, having two copies of each gene. These genes detail exactly how the creatures age, their instincts, their automatic reactions to stimuli, the chemical processes going on in their bodies, and so on, along with how the brain lobes and connections are structured and how they work (Grand, 2001). Although the workings of these do not need to be understood in order to enjoy the program, understanding them allows more advanced users to design new types of creatures, such as ones that can breathe underwater, which photosynthesize like plants, or which have even more realistic and detailed biochemistry than the creatures which come with the program.

Most forms of artificial life involve not just evolution of form or function, but some kind of learning as well. There are different ways to represent concepts and actions and thus allow learning, but a popular one is artificial neural networks. Artificial neural networks are a kind of machine learning algorithm based on biological neural networks, although highly simplified. In an artificial neural network, there are nodes, which have an input, an output, and a function that transforms the input in some way to produce an output. Connecting the nodes together are weights, the equivalent of biological synapses, which link together neurons. The weights might also transform the numbers that they pass along, and the numerical value of the weight determines how much the output of one node influences the input of the next node. Groups of nodes are called layers; a simple two-layer network would consist of a layer of input nodes, which receive and transform input to the network, and output nodes, which receive their input from the input nodes via the weights. A network with random weights will probably not produce a useful answer, so like in biological neural networks, there is a mechanism for the network to learn over time. When the network receives feedback from its output, it has an algorithm that uses the feedback to adjust the numerical values of the weights. Networks can be very simple or very complex, and can use different transformation functions in their nodes and weights. Because of their analog to biological brains, they are sometimes used to provide the brains of artificial life creatures, as they were in *Creatures*.

Creatures actually has its own variation on artificial neural networks which is more complex and in some ways perhaps more able to replicate real behavior. For example, in a standard network, weights do not change except when the feedback algorithm adjusts them. In real organisms, however, forgetting occurs. This is necessary, as sometimes connections between, for example, actions and consequences, are spurious, and forgetting allows an organism to focus more on connections that are more meaningful. The networks of *Creatures* included a forgetting mechanism: each weight had two values, a short-term and a long-term value. The short-term value falls back towards the long-term value, while the long-term value more slowly approaches the short-term value. For some layers, connections with weights that stay weak for long enough eventually disappear and new connections between different pairs of nodes are forged. If something turns out not to be relevant to a creature, it is eventually forgotten, letting it learn new things (Grand, 2001). Other artificial life programs have their own versions of neural networks, or use other structures altogether to give their creatures the ability to learn.

I decided to use neural networks for my project, as I was already somewhat familiar with them from working with *Creatures*, though I used a much simpler version. Like *OpenWorm*, I drew inspiration from real biology when possible, though my project is much more simplified from that biology than *OpenWorm*, and supports multiple creatures rather than just one. For example, I drew more on real aspects of genes and how two genomes create a child genome than the genetics of *Creatures* did. While the artificial ecology of a system like *Tierra* is fascinating, for this particular project I wanted to use concepts from the natural world when it was possible or practical to do so.

Chapter 3

DESCRIPTION OF SIMULATION

My program is written in Python, a language which is easy to use and understand and which has useful packages available for public download. In addition, I was already familiar with it before starting this project and I know an experienced Python user who could help me with language difficulties. Two packages were used in this project. One of these was PyGame, a package for making games which was used for displaying graphics, and the other was PyBrain, an artificial neural network package that was used to program the neural network brains of the creatures.

The neural networks for the creatures use reinforcement learning. That is to say, rather than learning from sets of problems where they are told whether or not they came up with the right answer or something similar, they take actions in an environment and receive a reward (positive or negative) based on the outcome of their actions. In the case of the creatures, this reward is based on changes in the creatures' drives. The neural network receives negative reward (that is, punishment) when the creature causes itself pain, for example, and positive reward when it fulfills its hunger and need for stimulation.

The way that reinforcement learning works in PyBrain is that one runs an 'experiment'. This experiment contains the agent, which consists of at least a neural network and a learning algorithm, and a task. This task contains the environment itself and acts as the go-between for the environment and agent. The task tells the agent what it can observe from the environment, tells the environment what actions the agent has taken, and calculates the reward from the agent's action and gives it to the agent. During a time step, the agent receives its observation from the task and runs it through its neural network. It then sends the result of this to the task and receives the reward that the task calculated. The learning algorithm uses this reward to decide how to change the weights in the neural network.

PyBrain has some documentation, but it is not complete, and it took me several days of looking through the code to understand how all the objects work together and how to use them for my simulation. Part of this is that the PyBrain documentation assumes that the reader is already familiar with not just the basic concept of neural networks, but with different learning methods and algorithms that beginner texts do not mention. I had difficulty either finding a text to take me up to the level PyBrain expected or understanding the definitions of the concepts that were mentioned in the tutorials. I ended up having to do a lot of experimentation rather than being able to rely on the documentation, and this led to further issues later on.

In my simulation, the environment is called a garden. It is a square grid-like environment that can be any size that is three-by-three or larger. Besides the creatures themselves, the garden contains grass, which the creatures need to eat to survive, a ball, which they can play with, a rock, which is not entertaining to play with and which is painful to attempt to eat, and everywhere else is covered with dirt, which does nothing but is the only surface the creatures can walk on. It loops around on each edge; that is, when a creature tries to go down when it is in the bottom row of the grid, it moves to the top row.

The experiment class of PyBrain is meant to run with only one agent at a time; it cannot be set up to run a multi-agent system. Therefore, I had to make a sub-class to allow for multiple agents in one environment. In my simulation, the agents are actually contained within another class, which is the class that defines the creatures. The creature class contains a creature's genome, position, direction, neural net, and so on, along with a PyBrain agent, which has the learning algorithm in addition to being connected to the creature's neural network.

The neural networks gave me the most difficulty in programming the simulation. On my first try, I attempted to follow the PyBrain tutorial and use a value-based learner with a state-action value table, but this only resulted in an error after the first time step until it was replaced with a policy-gradient learner and a network, which became more complex over time. Initially it only consisted of a vision layer, a group of neurons corresponding to the objects the creature can see, connected with an action layer, corresponding to the various actions the creature can do: go forward, turn left or right, eat what is in front of it, etc. When drives were added – pain, satiation, stimulation – so was another corresponding layer that connects to the actions layer. Initially this drives layer was set as a sigmoid layer, to squash the range of potential drive values to something more manageable. However, I found that the numbers were squashed too much and the creatures could not differentiate between being completely full and starving to death, for example. To overcome this problem, I wrote a new sigmoid function that squashed the drive values to a larger range.

Each creature has a simulated genome. This genome contains all of the information needed to make a creature's brain, and is diploid – that is, there are two 'strands' of genes, and so there are two copies of each gene. There are three kinds of genes: lobe genes, which define layers in the network; connection genes, which define connections in the network; and synapse genes, which define the range of potential starting numbers for each weight in the network. While lobe genes and connection genes do not mutate, synapse genes can. They have a special 'mutable field' parameter, which contains the maximum and minimum ends of the range, the mutation rate (how often that a gene will actually mutate when given the chance to do so), and numbers related to how the range mutates. The mutation rate itself can mutate as well, so over time, the population could change so that genes can mutate frequently or infrequently.

The main difficulty with working with a diploid genome is deciding on how to handle dominance. In real-life genomes, when there are two alleles, or copies of a gene on either chromosome, either the effects of the two can combine in some way, or only one of them is used – that is to say, one is dominant over the other. For example, if someone inherits from their mother a working allele for a certain protein, or copy of the gene, and inherits from their father a mutated allele that will not produce the correct protein that the body needs, the allele for the working protein will in most cases be dominant over the allele for the mutated protein. Other genes can co-dominate; for example, in some species of flower, a specific flower that receives an allele for white coloring and an allele for red coloring might end up pink. In biology, what determines which alleles are dominant over others and which can co-dominate and so on, are complex processes such as proteins folding correctly or not, which is beyond the level I am trying to simulate. Instead, I took something of a shortcut and use integers to compute dominance. If one allele has a higher dominance number than the other, then that allele is dominant. If both have the same dominance, then what happens depends on the gene type. In the genes that code for layers and connections, one copy is chosen randomly to be read. The genes that code for weights actually co-dominate; the minimum weights and maximum weights encoded in both alleles are averaged together. This range is then used to randomly choose the starting weight of that particular connection as usual.

The creatures that are created when the simulation is initialized are made from a default genome. The parameters for the synapse genes are pulled from a file in the directory; the values in this file can be altered, and were originally put together by looking at the range of network values in successful runs of creatures that started with completely random connection weights. However, as in real biology, genomes can also be created by recombination. The strands 'cross over' at a random point. This means that for both strands of genes, the strand up to the cross over point is inherited from the first parent, and the strand after that cross over point is from the other parent. This mixes the genes up between the strands, before all of the genes run their mutation function.

When the program is run, a window comes up showing the graphical display of the simulation. The PyGame library is used to run the graphics. The size of the window is determined by the size of the garden environment. On the left side of the screen, the garden is shown, displaying the creatures and objects. The right side of the screen is initially blank. However, when a creature is clicked on, the right side displays a live visualization of that creature's neural network (Figure 1). Each row or column of squares of the same color represents one layer of the network, and each square is an individual neuron in that layer. The vibrancy of the color represent the level of activation of that neuron; neurons with very low (less than -5) activation levels are grey, while ones that have higher activations have bolder colors. Each line represents a connection between two neurons, and while it is harder to see than the colors on the neurons, the connections with higher weights have lighter lines than the ones with lower weights.

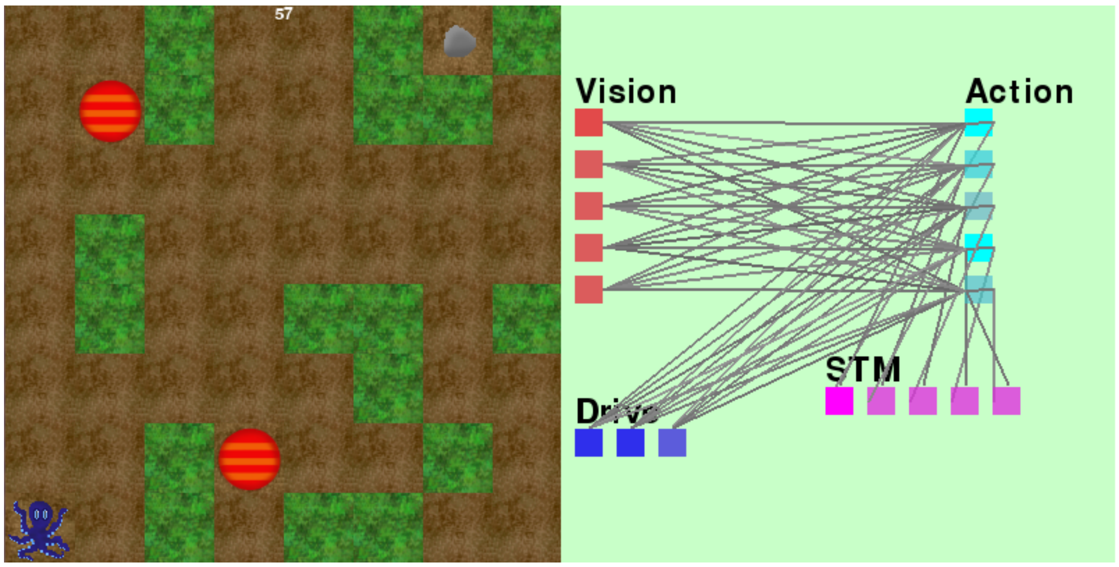

Figure 1 Screenshot of simulation window. Left: a creature in the garden. Right: visualization of its network.

The network has four layers, the Vision layer, the Drive layer, the Action layer, and the STM (short-term memory) layer. The Vision layer has five neurons, one for each object in the creatures' environment: dirt, grass, balls, rocks, and other creatures. The neurons of this layer are activated when the creature sees that particular object. The creature can only see images that are directly in front of it and to its front left and front right. It cannot see further away or directly to its side. The Drive layer has three neurons, one for each drive (hunger, boredom, pain), and the higher the activation level, the higher that drive is – higher hunger and boredom are more fulfilling for a creature, while pain should remain at zero. The action layer has five neurons, one for each action the creature can carry out: moving forward, turning left, turning right, attempting to eat what is in front of it, and attempting to play with what is in front of it. The neuron that has the highest activation level is the one whose corresponding action will be carried out. The STM layer also has five neurons, one for each neuron in the action lobe, and is explained in more detail further on.

Clicking on a particular neuron will change the view to only show the connections that originate in that particular neuron, along with the exact activation level of that neuron at any given moment (Figure 2). Viewing all of the connections and their weights in PyBrain is not straightforward; there is no one function or set of functions that does this. Instead, one has to nest several loops that go through each layer in the network, then use each layer to look up the connections that originate in that layer, and then use parameters that are not, by default, actually defined for all of the connection subclasses, to loop through all of the weights and find which neurons they connect. Recurrent connections are in another parameter entirely than all of the other connections, and have to be looped through separately. While this method works quite well, it is confusing to understand at first and is not a feature included with PyBrain or explained in its documentation, which made the visualization of the network more difficult to program.

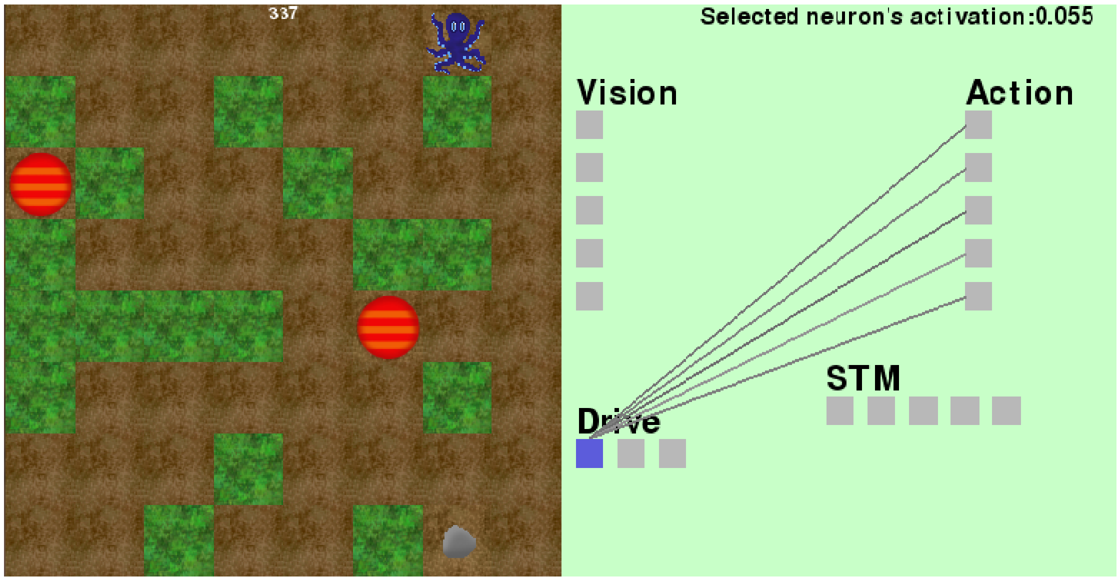

Figure 2 Left: creature about to eat grass. Right: selective view of neural net activity.

This method was also used to write functions that can print the state of a given network, save it to a file, or overwrite a creature's network from a file. When I started to use this to run experiments, I found that at some point the creatures' neural networks had actually stopped learning. I am still not entirely sure why. I was not able to figure out how to get the original learning algorithm working again, and I suspect that it might be meant for episodic learning rather than continuous or continual learning. After trying the other algorithms that come with PyBrain, the one called Episode Natural-Actor Critic worked – despite the name, it seems to function in a continuous environment. I did notice that the learning is quite slow for my purpose and that the creatures can take a long time (many repetitions) to learn things like 'don't eat rocks, it's very painful'. I experimented with the feedback to see if I could make the learning faster, but was unable to do so.

One consistent problem the creatures have had is that they often do the same action repeatedly – spinning in place, running forward, continuously attempting to eat something they cannot eat – in a way that feels unrealistic. The attempted solution for this involves a new layer, which is named the short-term memory (STM) layer. The action layer outputs to the STM layer via a custom connection class. The STM layer has the same number of neurons as the action layer, and each neuron of the action layer is connected only to the corresponding neuron in the STM layer. The connection only transmits the output of the neuron whose output is highest (which is what determines which action the creature will take). In the STM layer, the neuron which corresponds to that action neuron has its activation increased by one; for all the other neurons, their activations are decreased by half and rounded to zero if the new activation is under one. The STM layer is then connected back to the action layer. This solution did not work as well as hoped – there were some very strange patterns of activation in the action neurons, though this may be due at least in part to my not understanding how PyBrain handles neural networks that are recurrent (that is, which point back at themselves). However, it did visibly decrease the repetitive behavior somewhat, and in a small experiment it appeared to increase their lifespan. Their lifespans were limited to 7000 steps for the sake of time, and the group with the STM layer had a median lifespan that was 950 ticks higher than the group without the STM layer.

The main loop of the simulation goes like this: first, it checks for any events. If there is a quit event (i.e. the user has just clicked on the ‘quit’ button), then the simulation quits. If the user has clicked somewhere in the simulation window, the program checks to see if the click was in the left side (the garden side) or the right side (the network visualization side). If a creature was clicked on the left side, that creature becomes the selected creature and its network is displayed on the right. Otherwise, any selected creature is de-selected. If there is a visualization on the right side, if a neuron is clicked, that neuron becomes the selected neuron; otherwise any selected neuron is de-selected. Next, the experiment is run for a single time step, so that all of the creatures are given their input, carry out their actions, and learn. After this, all of the graphics are updated and drawn, including the network visualization if it is being shown, and the time step number. The simulation then checks to see if any creatures have starved to death, and if so, removes them from the environment's list of creatures. The display is sent to the screen to be shown to the user, and then the networks of all of the creatures are reset. This last step of resetting the networks is done because recursive networks in PyBrain save the entire history of the network, which is not very realistic, and which can take up a lot of memory and slow the simulation down over time. Resetting the network erases this history. The main loop continues to run until either the user quits the simulation or all of the creatures have died.

Chapter 4

CONCLUSION

The creatures of this simulation are simple in many ways. They can only see what is right in front of them, they have no real memory, and to them every other creature is the same, with no friends, enemies, or family. However, this simulation is not just the creatures, but also a platform for performing artificial life experiments. There is little that is hardcoded, so creating new brain structures, for example, is a matter of adding a few genes. New objects could very easily be added to the environment, as well. There are also built-in tools for recording useful data, such as the state of the network, as well as a graphical interface for easily understanding what is happening and the effects of changes to the network.

In the future, the program could be moved from a two-dimensional simulation to a three-dimensional one, in order to better replicate the experience of most animals. Along with this would come a better simulation of vision, and perhaps simulation of other senses such as hearing or smell. In addition, there are many things that could be done to make the neural network more complex – for example, adding object recognition rather than 'cheating' and letting the program tell the creature what it sees, or having the creature remember specific other creatures. Another thing that could be done would be perhaps moving the neural network away from PyBrain and to something more flexible and maybe more biologically-inspired.

Developing and experimenting with this simulation did meet my original objectives. The genetics system is biologically realistic in several ways and it is quite flexible and easy to expand, as is the object system. The creatures do learn, and over time as their neural networks and environment have improved, they have been able to survive more consistently and for longer amounts of time. Through this project I have also learned many things. In doing my background research I learned more about the variety of artificial life available. By working on it, not only did I improve my ability to code, both generally and in Python, but I also learned the uses and limitations of the PyBrain package and what features I might look for if I were to use another neural network library in the future. My knowledge of artificial neural networks, though not as complete as I would like, has improved considerably by working on this project. There are still more things I would like to do with this project, and even if I do not continue with it, I can apply the things I learned from it to other artificial life projects I take on in the future.

REFERENCES

Adamatzky, A., & Komosinski, M. (Eds.). (2005). *Artificial life models in software*. London: Springer.

Avida (2014, February 6). Retrieved from http://avida.devosoft.org/

Getting Started. (n.d.). Retrieved from http://www.openworm.org/getting_started.html

Grand, S. (2001*). Creation: Life and how to make it*. Cambridge, Mass: Harvard University Press.

Gras, R. (n.d.). *Ecosim: An ecosystem simulation*. Retrieved from https://sites.google.com/site/ecosimgroup/research/ecosystem-simulation

Kelly, K. (1994). *Out of control: The rise of neo-biological civilization*. Reading, Mass: Addison-Wesley. Available from http://kk.org/books/out-of-control/

Kerr, D. & Murczak, D. *The Artificial Planet User Manual*. Retrieved from http://aiplanet.sourceforge.net/manual/index.html

Appendix

NOTES ABOUT CODE

This simulation was written in Python 3.4. It is available for download at https://github.com/Lekyn/garden along with further instructions. It requires the PyBrain and PyGame libraries. PyBrain is currently only available in Python 2 and must be converted to Python 3 using the automatic 2to3 tool. PyGame is available in Python 3.