AUTOMATED MACHINE LEARNING SOLUTIONS TO GAMING APPLICATIONS

by

Fred Quentin Hickam Jr.

Honors Thesis

Appalachian State University

Submitted to the Department of Computer Science

in partial fulfillment of the requirements for the degree of

Bachelor of Science

May 2017

APPROVED BY:

_____

Dee Parks, Ph.D., Thesis Director

_____

Alice McRae, Ph.D., Second Reader

_____

Dee Parks, Ph.D., Departmental Honors Director

_____

James Wilkes, Ph.D., Department Chair, Computer Science

Table of Contents

Abstract

This thesis project details the creation of a simple platform game and an artificially intelligent agent capable of learning to play the game. An overview of relevant concepts, including genetic algorithms, artificial neural networks, and neuroevolution is presented. The algorithm Neuroevolution of Augmenting Topologies, utilized by the agent, is discussed in detail. Gameplay mechanics and game implementation details are presented. The agent's behavior and its development over time are explained.

Introduction

Video games have been a popular form of entertainment throughout the last half-century. Video games provide a challenge to human players by forcing them to provide proper game input in response to the obstacles represented in the game environment. From an artificial intelligence perspective, these games provide a different sort of challenge: creating an artificially intelligent agent capable of exploiting gameplay mechanics to yield automated game solutions. This process requires representing the gameplay environment with a computational model that inputs information from the game and outputs appropriate responses. This thesis project explores that challenge through the creation of an artificially intelligent agent that learns to play a simple platform video game.

Machine learning is a branch of artificial intelligence devoted to computational models capable of automatic adjustment without explicit programming [1]. One of the most important concepts within the field of machine learning is the artificial neural network, a self-learning computational model based on the same principles as the human brain [2]. In this thesis project, a machine learning algorithm known as Neuroevolution of Augmenting Topologies (NEAT) is used to develop an agent capable of playing a simple video game by using an artificial neural network to evaluate the game environment and generate appropriate responses [3].

A debt of inspiration for this project is owed to YouTube user SethBling. By loading a script written in the Lua programming language into the Nintendo Entertainment System emulator BizHawk, SethBling implemented *MarI/*O, an agent for automatically playing *Super Mario Bros.* levels [4]. The program controlled Mario through an artificial neural network developed by repeatedly attempting to solve the level by reading in data such as

Mario’s current position and the layout of the visible game area, then responding to the various forms of input data by outputting calls to controller commands on the emulator. The network’s fitness was evaluated based on Mario’s rightward progress through the level [4].

A core difference between this project and *MarI/O* is that *Super Mario Bros.* is a wholly static game wherein the landscape and gameplay mechanics of an individual level will not change on repeated attempts to complete the level. In *Swimmy Fish*, the game for which the agent described in the thesis was developed, the levels and obstacles therein are continually randomized throughout gameplay. This will test the agent’s ability to learn to recognize patterns within gameplay rather than memorize level architecture.

Chapter One

Conceptual Information

Genetic Algorithms

A genetic algorithm is a heuristic algorithm that operates with mechanisms designed to mimic those of biological evolution [5]. When attempting to solve a given problem with a genetic algorithm, the solution's domain is represented with a genetic encoding. The genetic algorithm then generates a *population* of potential solutions within that domain, known as *individuals*. After generation, individuals within a population are evaluated by a function known as a *fitness function*; this function, which is specific to the problem, returns the individual's *fitness*, a numerical value representing the individual's relative efficacy [5].

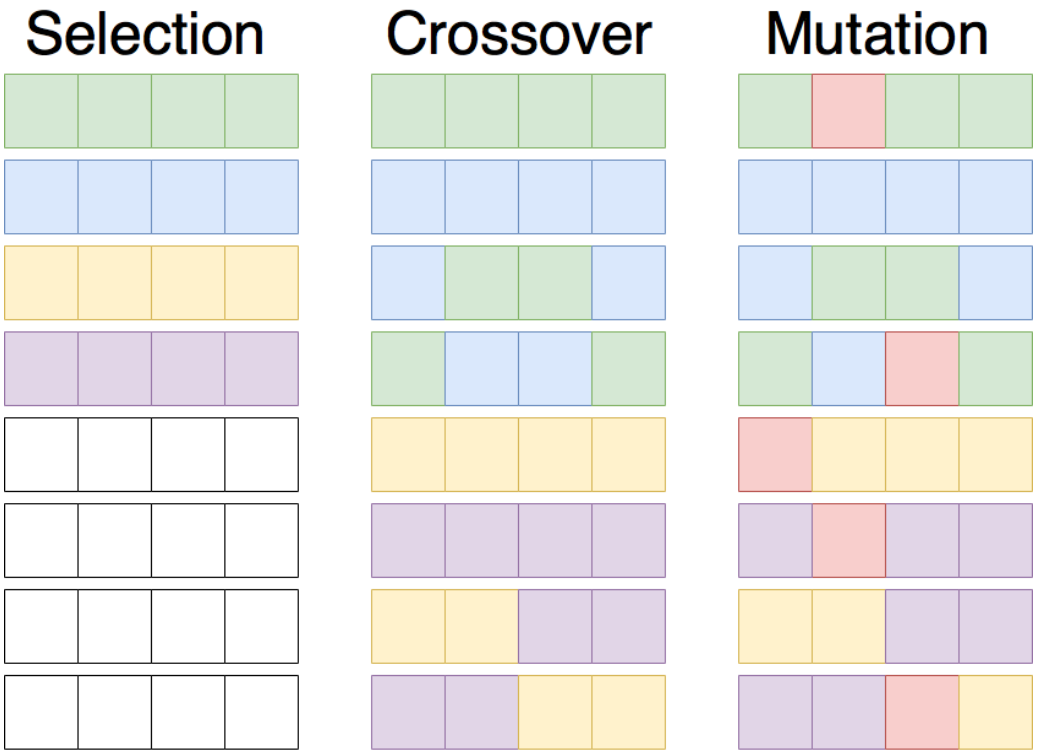

Figure 1: A visualization of the primary steps of a genetic algorithm.

During the breeding process for the new generation, the most effective members of the preceding generation are chosen as "parents" for future individuals; this is called *selection*. In a process known as *crossover*, portions of the genetic encodings of the parent individuals are swapped with one another to create a new individual containing characteristics of both parents. *Mutation*, in which the genetic encoding of an individual is randomly altered, subsequently occurs to preserve genetic diversity within the population by introducing new genotypes into the pool. The process of generating new individuals via natural selection, crossover, and mutation continues until a definitive solution has been found, or until a specified fitness score or number of generations is achieved [5].

Artificial Neural Networks

An *artificial neural network* is a computational model based on the behaviors of biological neural networks, the foundation of the nervous system [2]. An artificial neural network is composed of individual neurons and their weighted connections to one another, which are known as *synapses*. In a simple artificial neural network, neurons can be classified as one of three types, or *layers*: *input*, *output*, and *hidden*; the number of neurons in each layer is dependent upon the nature of the problem to which the network will be applied [2].

Input neurons take in data from an external source; for this reason, they are sometimes called *sensors*, as they mimic sensory input to a biological neural network. Between the inputs and outputs are hidden neurons, which serve as transitional stages between inputs and outputs. These neurons allow the network to "think" about the input data in its decision-making process by sending different ranges of input data along different pathways of the network. Output neurons represent the final decisions of the computational model [2].

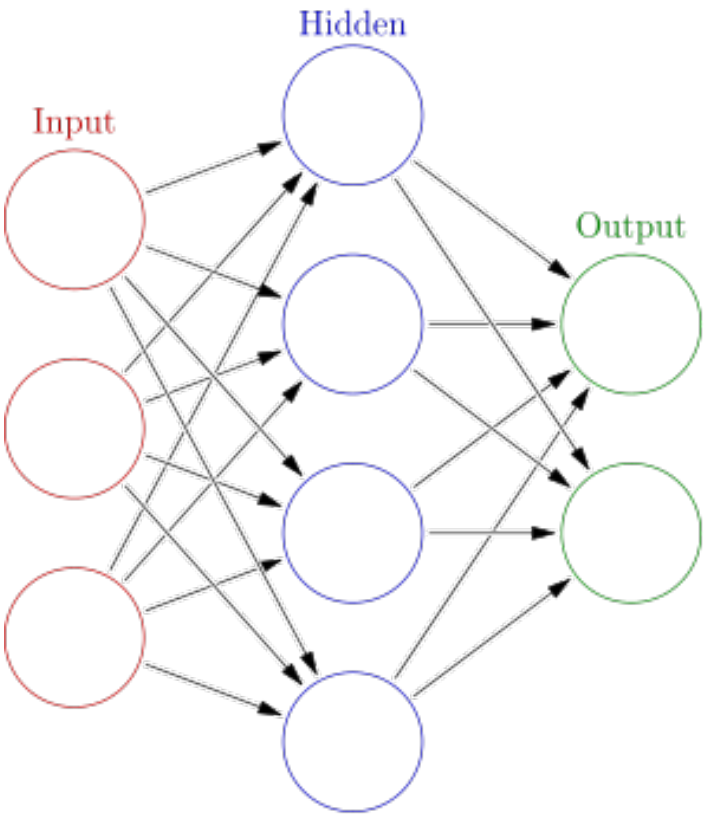

Figure 2: A simple artificial neural network.

Image provided by Glosser.ca under a Creative Commons BY-SA license.

Numerical values, typically normalized to a small range, enter the ANN through the input layer. If the values are great enough to meet a threshold value assigned to a neuron, they are fed forward to the next connected neuron. As the values travel along the connections between neurons, they are multiplied by the connections' weight, allowing neurons to *excite* (encourage) or *suppress* one another's influence on the final outputs through the positive or negative value of their weights. This eventually yields a signal from the output neurons. Traditionally, a series of example inputs and outputs is provided to train the ANN to produce correct output [2].

Neuroevolution

The term *neuroevolution* refers to the concept of evolving artificial neural networks with genetic algorithms [3]. Unlike conventional artificial neural networks, neuroevolution is not necessarily dependent upon human-created training data. A randomly-generated artificial neural network with user-defined inputs and outputs is applied to the given task, then evaluated through the genetic algorithm's fitness function. New, more adept networks are then developed through the reproductive functions of the genetic algorithm; potential mutations include altering the weight of a synapse, adding a new synapse between two neurons, adding a recursive synapse that causes a neuron to feed data back into itself, adding new neurons to the hidden layer, or removing an individual neuron or synapse from the hidden layer. The newly generated networks are then pitted against the same challenge as their parent networks. This process continues until user-defined goals have been met [3].

NEAT: Neuroevolution of Augmenting Topologies

Neuroevolution of Augmenting Topologies (NEAT) is a genetic algorithm for the evolution of artificial neural network structures [3]. This algorithm was first described by Kenneth Stanley and Risto Miikkulainen in a 2002 issue of the Massachusetts Institute of Technology's *Evolutionary Computing* journal. The NEAT algorithm begins by generating a population of simple feed-forward artificial neural networks consisting solely of a user-defined number of input and output neurons. Over time, NEAT structurally evolves these artificial neural networks to include a hidden layer of neurons and various connections amongst them.

Throughout iterations of NEAT, the structure of the artificial neural network may mutate in one of two different ways: it may either develop a new intermediary hidden neuron, or it may establish a connection between two previously unconnected neurons. NEAT keeps an historical record of unique mutations, which Stanley calls *innovations*. Each innovation is logged in this record and tagged with a specific identifier. In addition to these structural mutations, NEAT can also mutate the weights of the synapses between neurons [3].

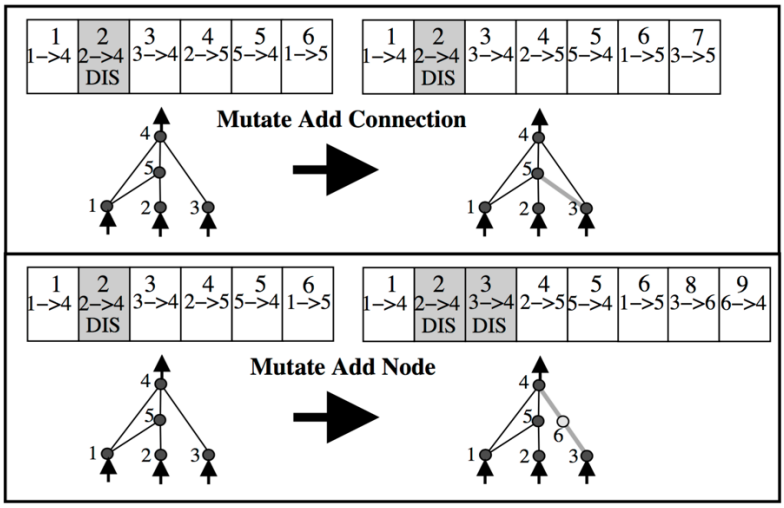

Figure 3: NEAT's structural mutation possibilities [3].

In comparing two separate network genomes produced by NEAT, the innovations possessed by only one genome are classified as either *disjoint* or *excess*, depending on whether the innovations are outside the other genome's range of innovation identifiers. During the crossover process of NEAT, two parent ANNs combine their genetic encodings to produce an individual offspring. Shared innovations are randomly inherited from either parent, but disjoint and excess innovations are inherited exclusively from the more fit parent. In the event that the parents are of equal fitness, disjoint and excess innovations are inherited from both parents [3].

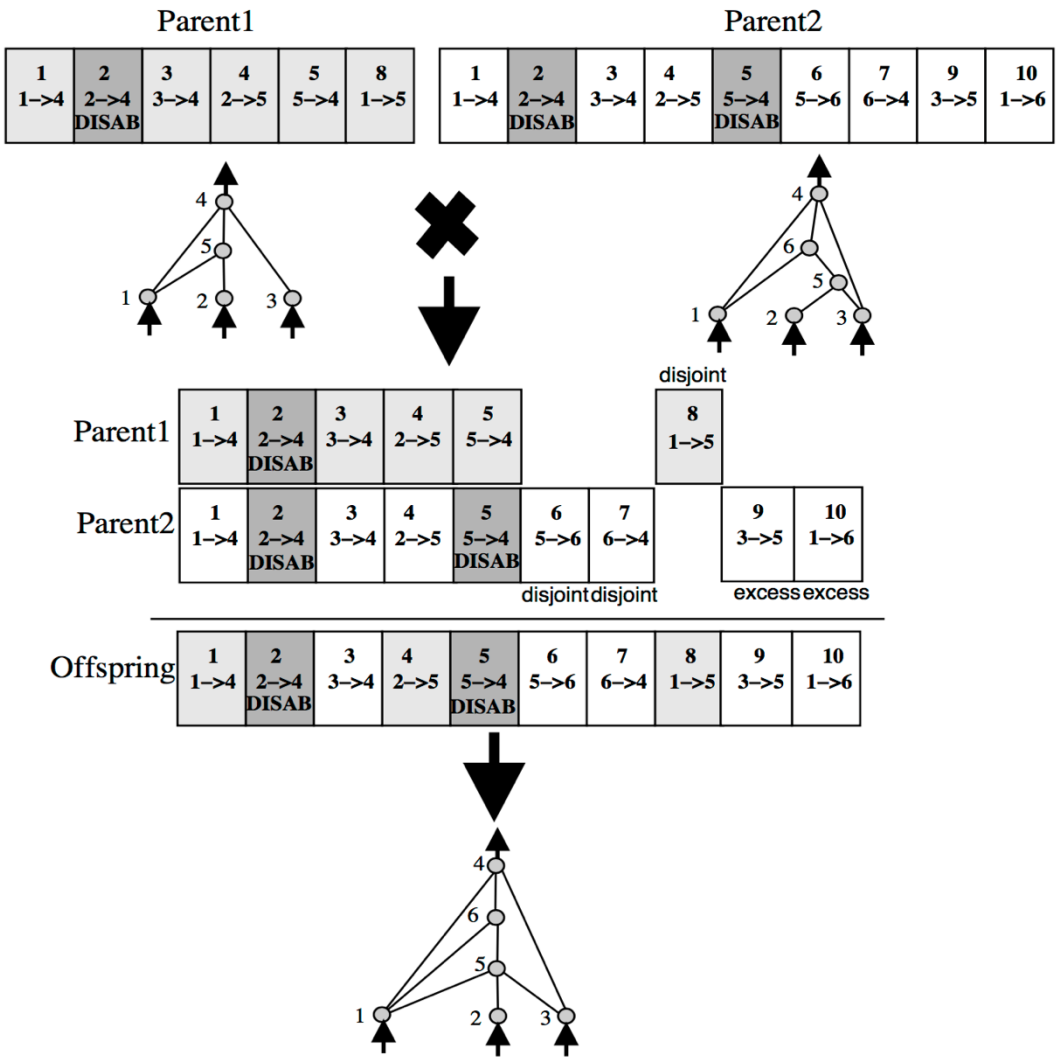

Figure 4: NEAT's reproductive process for two parents of equal fitness [3].

To protect diversity within the population of network structures, NEAT utilizes a concept known as *speciation*. The *compatibility distance* of two ANNs is calculated using the number of disjoint and excess innovations, as well as the average difference of the weights of the shared genes. ANNs within a specified threshold of compatibility distance are stated to be members of the same *species*. During the reproduction process of NEAT, individual networks have their fitness adjusted to reflect the size of their species. This reduction of fitness encourages species that would otherwise dominate the natural selection process to kill

off their weaker members, protecting structural diversity within the pool of individual network genotypes [3].

The implementation of NEAT utilized for this project is jNEAT by Dr. Kenneth O. Stanley and Ugo Vierucci. jNEAT is an open-source implementation of the NEAT algorithm in the Java programming language. It is the most faithful Java conversion of Dr. Stanley's original NEAT software package, which was written in C++. jNEAT has been endorsed by Dr. Stanley as part of the NEAT Software Catalog, an online listing of approved NEAT software packages [6].

Chapter Two

Swimmy Fish: A Simple Game

Overview of Game Concepts

Swimmy Fish is a simple platform video game developed for the purposes of this project. The player is tasked with guiding a fish named Swimmy through the process of navigating an infinite, procedurally generated underwater pipe system. Pipes scroll leftward from the rightmost side of the game screen. Swimmy must position herself to swim through the gaps in the pipes. Swimmy may maneuver through the game environment by swimming upward, surrendering to a downward current, or diving down at an even faster rate than the current. The game terminates upon passing through the 50th pipe.

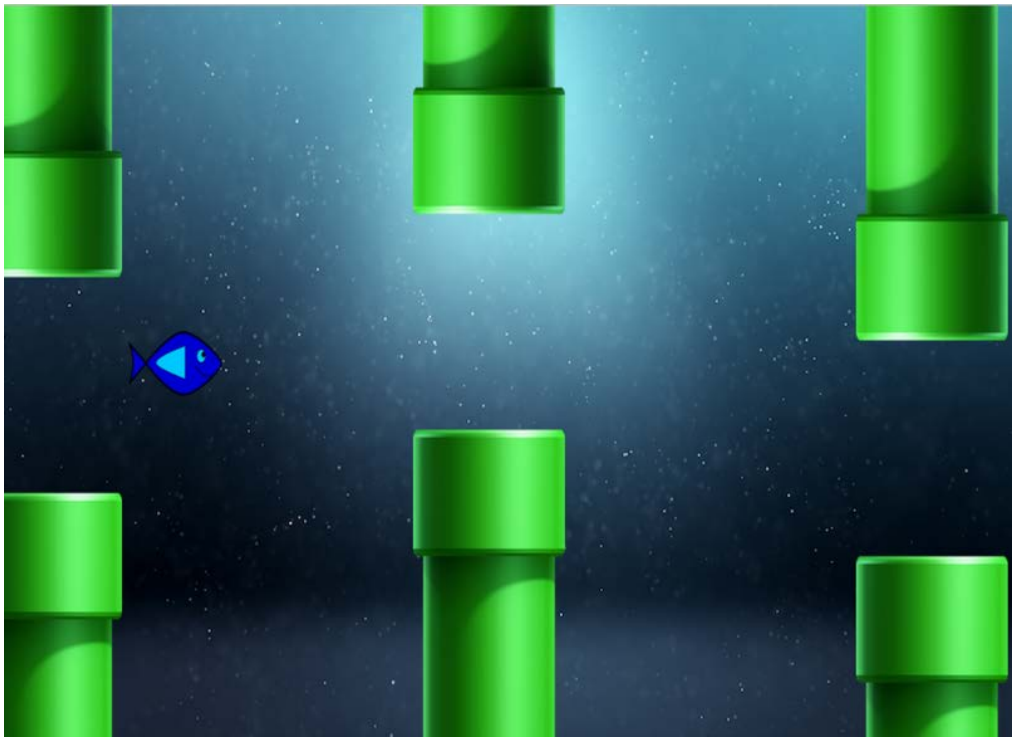

Figure 5: The Swimmy Fish *gameplay screen, showing a single Swimmy character and a series of pipes.*

Implementation Details

Swimmy Fish is written in the Java programming language using the Swing toolkit. An abstract class called *GameObject* defines basic behaviors of objects in the game world, including the object's coordinates, directions for drawing the object, and collision detection.

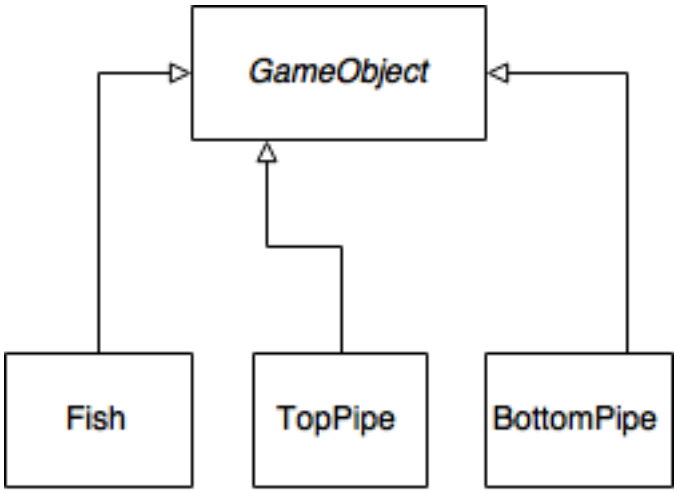

Figure 6: A very simple diagram showing inheritance of the game objects.

Another abstract class, *Board*, instantiates these three types of GameObject, draws them on the screen, and manipulates them according to designated game logic. This project includes two concrete subclasses of *Board*: *ManualBoard*, which allows a human player to control a single fish using keyboard inputs, and *NeuralBoard*, which allows the agent to automatically control fifty fish simultaneously.

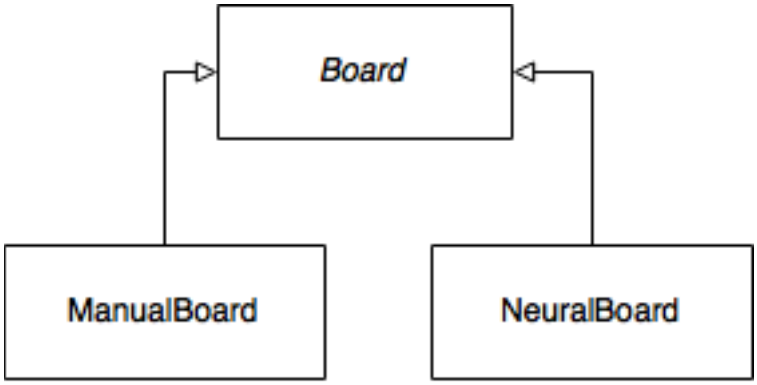

Figure 7: A simple diagram showing Board *inheritance.*

Detailed Game Logic

Swimmy Fish game logic is executed in a *Board* object. This object contains an ArrayList of *Fish*, allowing one or more Swimmy characters to share the game screen at one time. Additionally, two arrays with corresponding indices hold the *TopPipe* and *BottomPipe* pairs. On a frame-by-frame basis, these objects are manipulated according to the rules of the game in the gameLogic() method, which calls helper methods to operate the proper logic for each individual element of the game. After the game state has been properly calculated and adjusted, the game objects are drawn on the board in the appropriate positions.

The pipeLogic() method manipulates the pipes that Swimmy must overcome. On each frame, the pipes are advanced five x-positions to the left. Initially, the leftmost pipe pair is designated internally as the "current" pair, i.e. the pair that the player or agent should be focused upon avoiding. If Swimmy has advanced far enough to the right of this pair, the next pair to the right will then be designated as the "current" obstacle. If a pair of pipes advances completely off the screen, they will be respawned at the rightmost edge of the screen with a randomized height.

The fishLogic() method orders the Swimmy characters currently populating the game screen to update their position and dictates how they react to colliding with an obstacle. If a Swimmy character goes too far above or below the game area, or if she intersects with any of the pipes currently on screen, she is killed and ceases to score points. Otherwise, she receives points based on the game's scoring system.

The scoring system in *Swimmy Fish* is based upon Swimmy's position relative to the pipes. Swimmy begins with a score of 0 as the game begins. Within a certain x-coordinate distance of the next pair of pipes, Swimmy's score is penalized one point for each frame in

which her position is in a region parallel to the top or bottom pipe. Her score is rewarded by 15 points per frame if she is in the appropriate range to swim through the next pair of pipes. A visual representation of the regions evaluated by the scoring system is presented in the next figure.

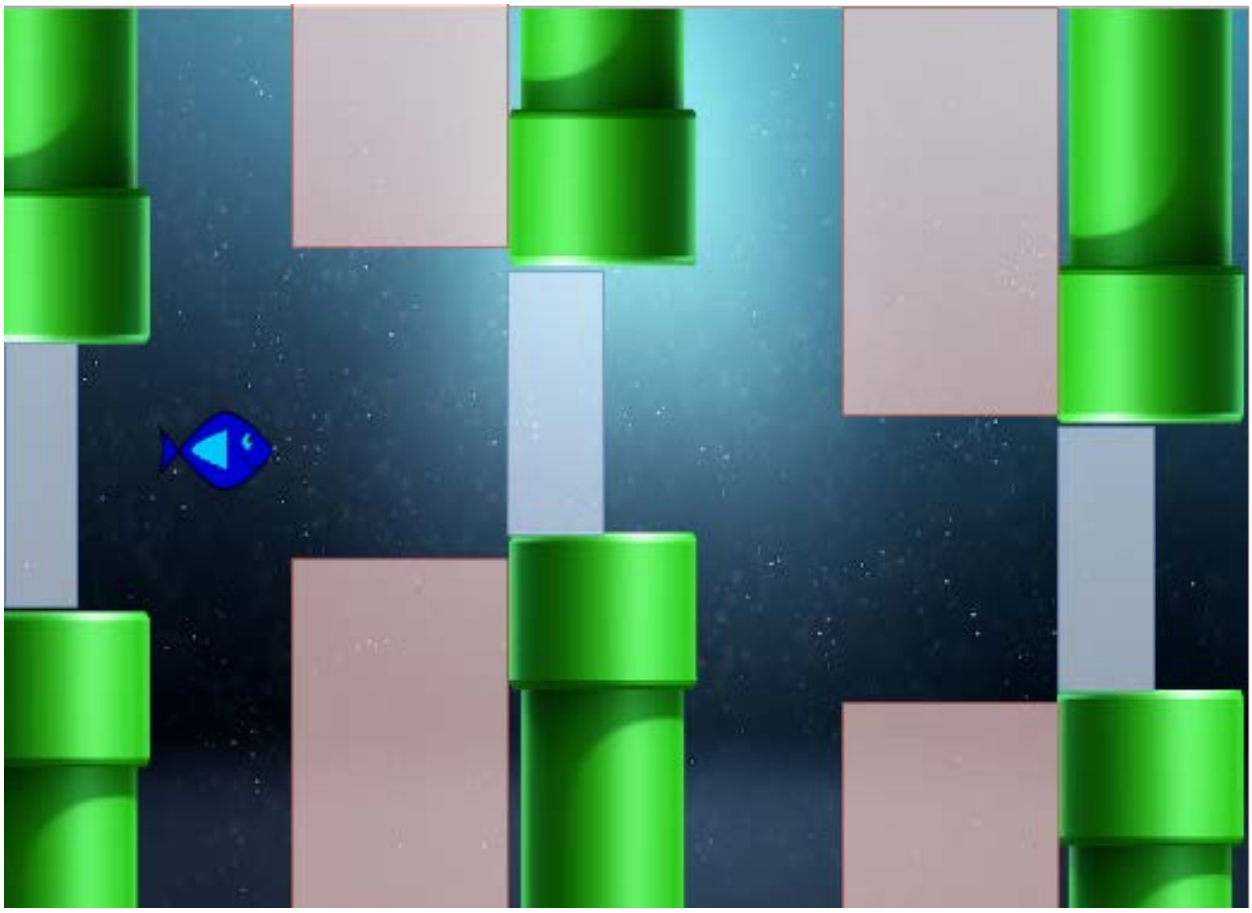

Figure 8: An approximate visual depiction of the scoring system.
Blue-shaded regions represent scoring zones while red-shaded regions represent penalty zones.

Chapter Three

The *Swimmy Fish* NEAT Agent

Overview

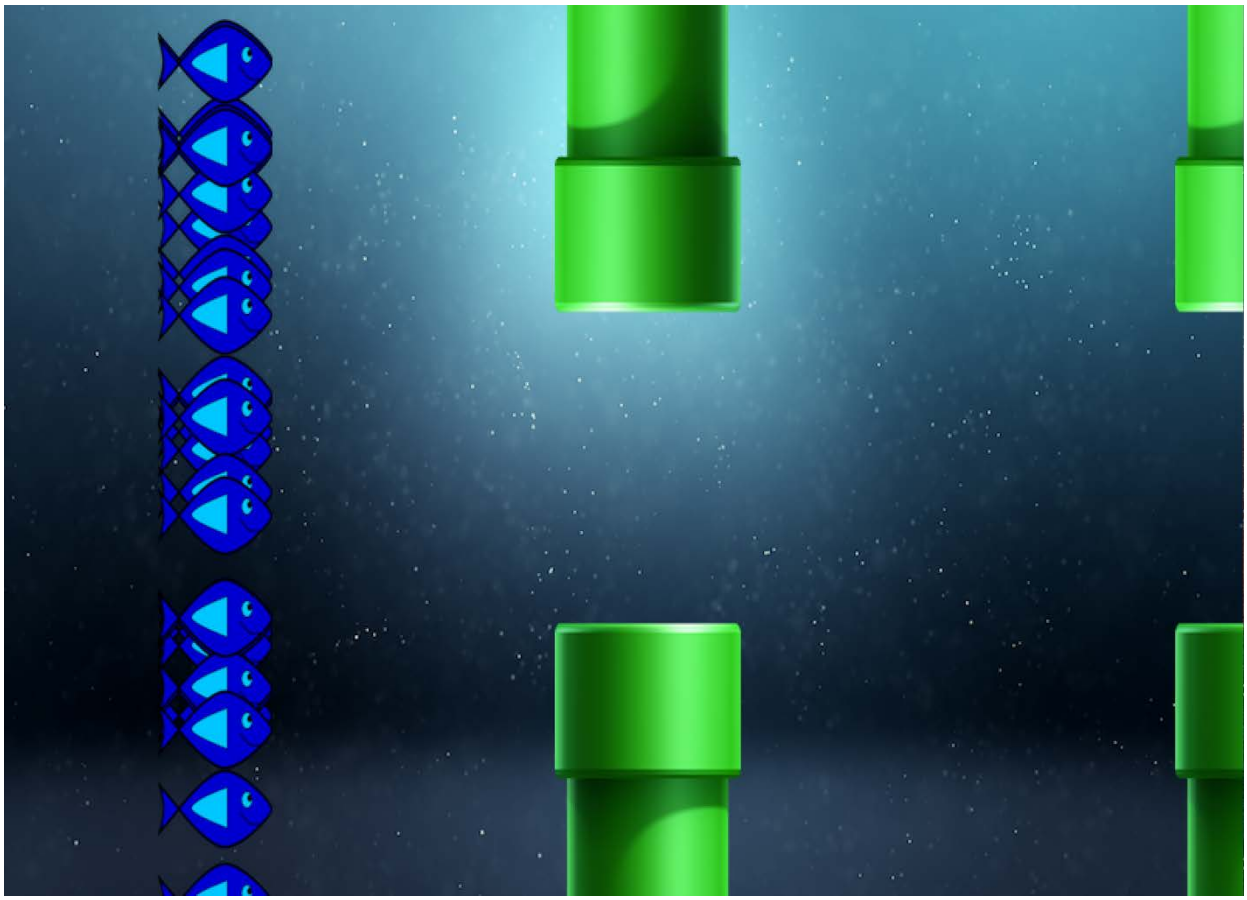

Figure 9: The first population of agent-controlled fish begins to play the game.

In the automated version of the game, dubbed *Neural Fish*, members of a population of fifty Swimmy characters compete against one another in navigating the maze of pipes. Each fish is controlled by a single artificial neural network generated using the jNEAT library. When the entire population has died, the game resets, the networks are evolved by the NEAT algorithm, and a new, improved population of fish is spawned to attempt the challenge.

Network Structure and Usage

This artificial neural network has three inputs: Swimmy's current position along the y-axis and the next pipe's position along the x-axis and y-axis; these values are normalized to the range [0, 1]. Members of the initial population consist of these four neurons and synapses connecting them, as depicted in the below diagram.

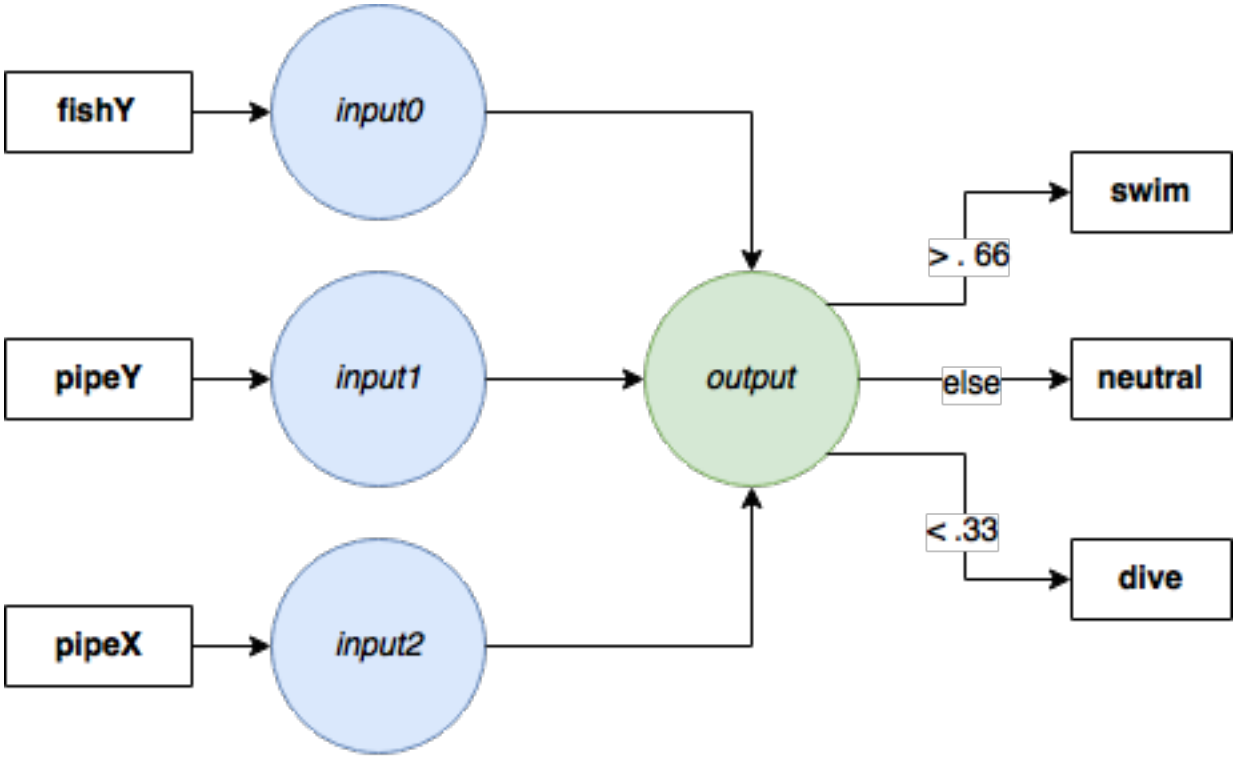

Figure 10: A primitive artificial neural network for controlling an individual Swimmy character.

Each frame, the inputs are updated and the network's output is generated. The network then outputs a value in the range [0,1]; this range is divided into thirds, with the upper third being interpreted as a command to swim, the lower third being interpreted as a command to dive, and the middle third being interpreted as a command to do neither.

Generational Growth

Each population of fifty Swimmy characters is known as a *generation*. When all fifty members of the population have died, the game loop enters an *evaluation* stage. In this stage, the fitness of the current population of networks is determined by assigning each network a fitness score equal to the gameplay score of its associated Swimmy character. The networks then enter NEAT's reproductive stage. Over time, the reproductions and mutations that occur between generations will yield more intricate neural networks capable of complex "thinking" about the game environment. The below figure, generated by jNEAT's built-in visualization utility, is a network structure evolved on the 20th generation of game attempts; this structure scored 18,054 points on its successful attempt to clear all 50 pipes. Neurons 1, 2, and 3 represent inputs while Neuron 14 represents output. The other neurons are mutations to the hidden layer.

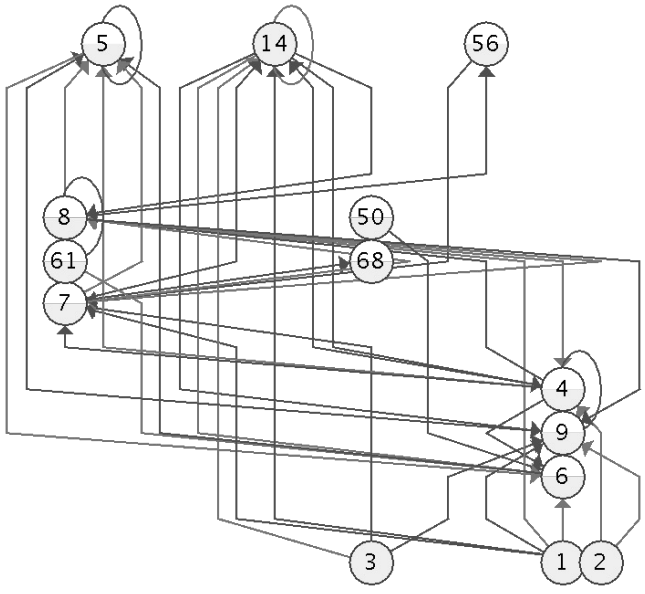

Figure 11: An evolved artificial neural network from the 20th generation.

Observations

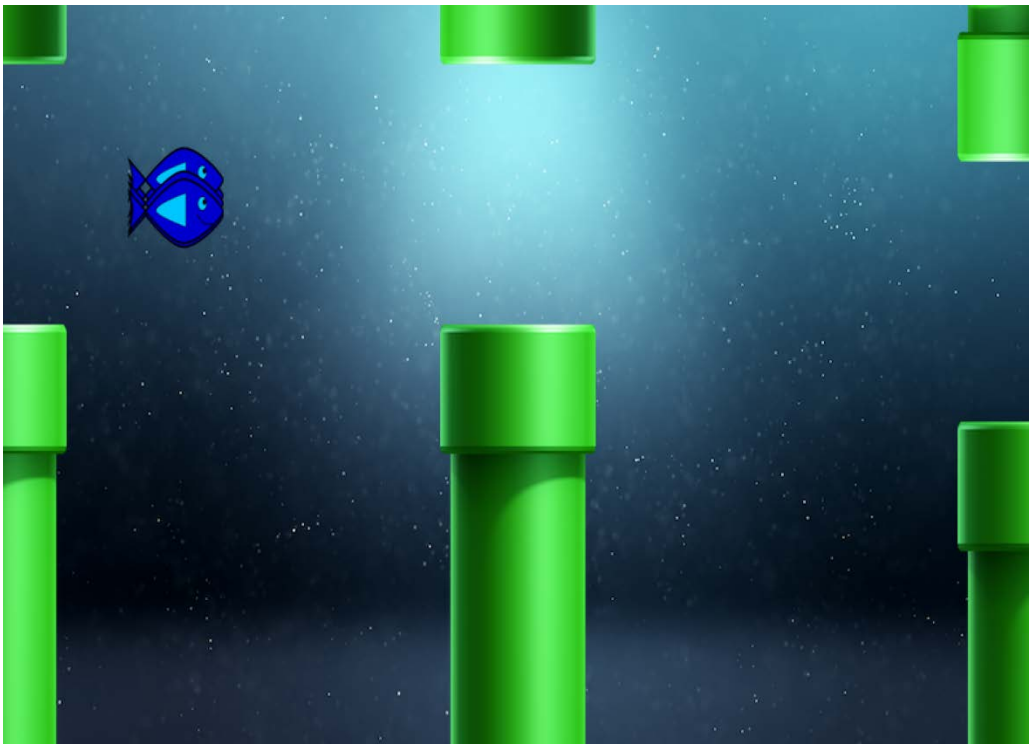

Figure 12: A group of Swimmy characters successfully navigating the game world.

When launching the *Neural Fish* game simulation, a generation of networks is spawned and each network is assigned to a single Swimmy character. What follows is a typical scenario that occurs upon starting the neuroevolutionary process from scratch: The first few generations are typically inept at navigating the game world and crash into the pipes. A few will successfully enter the gap but crash into the top or bottom pipe on their attempt to proceed forward. Eventually, enough of the population will make it through the first gap in the pipe for continual evolution to occur.

At the second pipe, the scoring system's negative reinforcement for swimming directly toward the pipe begins to take effect. The networks capable of navigating the first pipe gap will begin to "understand" that it is bad for the fish's y-coordinates to be within

range of the pipes; as such, they will correct the fish's trajectory toward the next gap. A few of these will then develop sufficient neural structure to pass through the second pipe.

Eventually, a handful of fish will be able to make it past the third pipe. It is here that the rate of evolution grows substantially. The networks that have developed sufficiently to compute the appropriate interaction for three of the twenty possible pipe heights are typically quite adept at handling the others in the pool. A few individuals begin to successfully navigate ten to twenty pipes, followed quickly by one or more individuals capable of completing the series of pipes.

At this point, these successful individuals' extremely high fitness scores begin to influence the remainder of the population. Gradually, the process changes from developing a successful solution to a refinement of that solution. Driven by the negative reinforcement from the scoring system, members of the population take divergent paths toward manipulating the fish through the pipes, improving the characters' reaction time.

Due to the randomized nature of the NEAT algorithm, the number of generations required for the artificial neural networks to evolve appropriate responses to the game environment can vary. One observation from the process is that the networks tend to evolve more quickly when faced with a series of pipes with relatively small differences in height. From this, perhaps, the network incorporates the subtleties of the necessary movement patterns into its computational model and creates a more accurate algorithmic process more quickly.

Conclusion

The project is presented in a functional state. The NEAT agent is capable of developing a population of neural networks that can play the game, sometimes with great success. Watching the agent play the game is an entertaining and informative visual demonstration of the process of neuroevolution within a gaming environment. This neuroevolutionary agent has effectively conquered the simple game developed for this project – no small feat, given there are twenty different potential pipe heights to navigate, and therefore 400 possibilities for the placement of two successive pairs of pipes.

Future work for the project includes further complications to gameplay. This may include enemy characters to dodge or fight, and more substantial variations in static obstacles' shape or relative distance. Development of NEAT-based agents for other games using similar approaches is also a future research goal.

Bibliography

[1] SAS Institute, "Machine Learning: What it is and why it matters," SAS Institute, [Online]. Available: https://www.sas.com/en_us/insights/analytics/machine-learning.html. [Accessed 29 March 2017].

[2] M. Buckland, "Neural Networks," in *AI Techniques for Game Programming*, Cincinatti, OH: Premier Press, 2002.

[3] K. O. Stanley and R. Miikkulainen, "Evolving Neural Networks through Augmenting Topologies," *Evolutionary Computation*, vol. 10, no. 2, pp. 99 - 127, 2002.

[4] SethBling, "MarI/O: Machine Learning for Video Games," 13 June 2015. [Online]. Available: https://www.youtube.com/watch?v=qv6UVOQ0F44. [Accessed 4 December 2016].

[5] M. Mitchell, *An Introduction to Genetic Algorithms*, Cambridge, MA: Massachusetts Institute of Technology Press, 1996.

[6] K. O. Stanley, "NEAT Software Catalog," Evolutionary Complexity Research Group at UCF, [Online]. Available: http://eplex.cs.ucf.edu/neat_software/. [Accessed 3 April 2017].

Appendix: Source Code

Config.java

```
package game;
import java.awt.Dimension;

public class Config
{
	public final static String GAME_TITLE = "Swimmy Fish";

	public final static int HORIZONTAL_RES = 800;
	public final static int VERTICAL_RES = 600;
	public final static Dimension DIMENSION = new Dimension(HORIZONTAL_RES,
VERTICAL_RES);

	public final static int TIMER_INT = 10;

	public final static int FISH_START_XLOC = 100;
	public final static int FISH_START_YLOC = 0;
	public final static int FISH_SWIM_RATE = -6;
	public final static int FISH_DIVE_RATE = 6;
	public final static int FISH_NEUT_RATE = 3;

	public final static int TOP_PIPE_BASE_HEIGHT = -390;
	public final static int BOTTOM_PIPE_BASE_HEIGHT = 390;

	public final static int FIRST_PIPE_XLOC = 650;
	public final static int SECOND_PIPE_XLOC = 1050;
	public final static int THIRD_PIPE_XLOC = 1450;

	public final static String RESOURCE_PATH = "src/images/";
	public final static String FISH_PATH = RESOURCE_PATH + "fish.png";
	public final static String BOARD_PATH = RESOURCE_PATH + "ocean.png";
	public final static String TOP_PIPE_PATH = RESOURCE_PATH + "topPipe.png";
	public final static String BOTTOM_PIPE_PATH = RESOURCE_PATH + "bottomPipe.png";

	public final static int MAX_SCORE = 17000;
	public final static int MAX_PIPES = 50;
}
```

Application.java

```
package game;

import java.util.Scanner;
import javax.swing.JFrame;
import agent.NeuralBoard;

public class Application extends JFrame
{
	public Board board;
	public boolean manual;

	public Application(boolean manual)
	{
		if (manual)
		{
			board = new ManualBoard();
		}
		else
		{
			board = new NeuralBoard();
		}

		initialize();
	}

	private void initialize()
	{
		add(board);
		setSize(Config.DIMENSION);
		setTitle(Config.GAME_TITLE);
		setDefaultCloseOperation(JFrame.EXIT_ON_CLOSE); //closes program when exit
is clicked
		setLocationRelativeTo(null); //centers window
	}

	public static void main(String[] args)
	{
		Application ex;
		Scanner scan = new Scanner(System.in);

		System.out.println("Would you like to play Swimmy Fish or Neural Fish?
Type 1 or 2");
		System.out.println("1. Swimmy Fish");
		System.out.println("2. Neural Fish");

		if (scan.nextLine().equals("1"))
		{
			ex = new Application(true);
		}
		else
		{
			ex = new Application(false);
		}

		ex.setVisible(true);
		ex.board.gameLogic();
	}
}
```

GameObject.java

```
package game;
import java.awt.Graphics;
import java.awt.Image;
import java.awt.Rectangle;

import javax.swing.ImageIcon;

public abstract class GameObject
{
	protected int x;
	protected int y;
	protected int dx;
	protected int dy;
	protected int height;
	protected int width;
	protected Image sprite;

	public GameObject()
	{
	}

	public GameObject(int x, int y)
	{
		this.x = x;
		this.y = y;
	}

	public void loadImage(String filename)
	{
		//load the resource
		ImageIcon ii = new ImageIcon(filename);
		sprite = ii.getImage();
		//set the object's height and width appropriately
		height = sprite.getHeight(null);
		width = sprite.getWidth(null);
	}

	public void draw(Graphics g)
	{
		g.drawImage(sprite, x, y, null);
	}

	//used for collision detection
	public Rectangle getBounds()
	{
		return new Rectangle(x, y, width, height);
	}

	public boolean collides(GameObject go)
	{
		return this.getBounds().intersects(go.getBounds());
	}

	public Image getSprite()
	{
		return sprite;
	}

	public int getX()
	{
		return x;
	}

	public int getRightX()
```

```
	{
		return x + width;
	}

	public int getY()
	{
		return y;
	}

	public int getBottomY()
	{
		return y + height;
	}

	public int getHeight()
	{
		return height;
	}

	public int getWidth()
	{
		return width;
	}

	public void setX(int x)
	{
		this.x = x;
	}

	public void setY(int y)
	{
		this.y = y;
	}

	public void move()
	{
		x += dx;
		y += dy;
	}

	public void moveLeft()
	{
		x--;
	}

	public void moveRight()
	{
		x++;
	}

	public void moveUp()
	{
		y--;
	}

	public void moveDown()
	{
		y++;
	}
}
```

Fish.java

```
package game;
import java.awt.event.ActionEvent;
import javax.swing.AbstractAction;

public class Fish extends GameObject
{
	protected static final String path = Config.RESOURCE_PATH + "fish.png";
	protected int score;
	protected boolean dead;

	public Fish()
	{
		this(Config.FISH_START_XLOC, Config.FISH_START_YLOC);
	}

	public Fish(int x, int y)
	{
		super(x, y);
		dy = 3;
		score = 0;
		dead = false;
		loadImage(path);
	}

	public void swim()
	{
		dy = Config.FISH_SWIM_RATE;
	}

	public void neutral()
	{
		dy = Config.FISH_NEUT_RATE;
	}

	public void dive()
	{
		dy = Config.FISH_DIVE_RATE;
	}

	public void kill()
	{
		dead = true;
		//used to weed out especially poor scorers
		if (score < -750)
		{
			score = -10000;
		}
	}

	public boolean isDead()
	{
		return dead;
	}

	public void setScore(int score)
	{
		this.score = score;
	}

	public int getScore()
	{
		return score;
	}
```

```
	public void score()
	{
		score += 15;
	}

	public void penalize()
	{
		score -= 1;
	}

	//for external invoking from an ActionListener

	private class SwimAction extends AbstractAction
	{
		public void actionPerformed(ActionEvent e)
		{
			{
				swim();
			}
		}
	}

	private class NeutralAction extends AbstractAction
	{
		public void actionPerformed(ActionEvent e)
		{
			{
				neutral();
			}
		}
	}

	private class DiveAction extends AbstractAction
	{
		public void actionPerformed(ActionEvent e)
		{
			dive();
		}
	}
}
```

TopPipe.java

```
package game;
public class TopPipe extends GameObject
{
	public TopPipe()
	{
		super();
		loadImage(Config.TOP_PIPE_PATH);
	}

	public TopPipe(int x, int y)
	{
		super(x, Config.TOP_PIPE_BASE_HEIGHT + y);
		loadImage(Config.TOP_PIPE_PATH);
	}

	public void setY(int y)
	{
		super.setY(Config.TOP_PIPE_BASE_HEIGHT + y);
	}
}
```

BottomPipe.java

```
package game;

public class BottomPipe extends GameObject
{
	public BottomPipe()
	{
		super();
		loadImage(Config.BOTTOM_PIPE_PATH);
	}

	public BottomPipe(int x, int y)
	{
		super(x, Config.BOTTOM_PIPE_BASE_HEIGHT + y);
		loadImage(Config.BOTTOM_PIPE_PATH);
	}

	public void setY(int y)
	{
		super.setY(Config.BOTTOM_PIPE_BASE_HEIGHT + y);
	}
}
```

Board.java

```
package game;
import java.awt.Graphics;
import java.awt.Graphics2D;
import java.awt.Image;
import java.awt.Toolkit;
import java.awt.event.ActionEvent;
import java.util.ArrayList;
import java.util.Random;
import java.util.Timer;
import java.util.TimerTask;

import javax.swing.AbstractAction;
import javax.swing.ImageIcon;
import javax.swing.JPanel;
import javax.swing.KeyStroke;

import game.Board.GameLoop;
import game.Board.PauseAction;

public abstract class Board extends JPanel
{
	protected static final int BOARD_X = 0;
	protected static final int BOARD_Y = 0;

	protected ArrayList<Fish> fish;
	protected TopPipe[] topPipe = new TopPipe[3];
	protected BottomPipe[] bottomPipe = new BottomPipe[3];

	protected Image background;

	protected Random rando;

	protected Timer timer;
	protected int gameSpeed = Config.TIMER_INT;

	protected boolean paused;

	protected int numPipes;
	protected int pipeSpeed;
	protected int closestPipeIndex;

	public Board()
	{
		initialize();
	}

	public BottomPipe getClosestBottomPipe()
	{
		return bottomPipe[closestPipeIndex];
	}

	public TopPipe getClosestTopPipe()
	{
		return topPipe[closestPipeIndex];
	}

	@Override
	public void paintComponent(Graphics g)
	{
		super.paintComponent(g);
		drawComponents((Graphics2D) g);
		Toolkit.getDefaultToolkit().sync();
	}
```

```
protected void drawComponents(Graphics2D g)
{
	g.drawImage(background, BOARD_X, BOARD_Y, null);

	for (int i = 0; i < bottomPipe.length; i++)
	{
		bottomPipe[i].draw(g);
		topPipe[i].draw(g);
	}

	for (int i = 0; i < fish.size(); i++)
	{
		if (!fish.get(i).isDead())
		{
			fish.get(i).draw(g);
		}
	}
}

protected void initialize()
{
	ImageIcon ii = new ImageIcon(Config.BOARD_PATH);
	background = ii.getImage();

	rando = new Random();

	initializeControls();
	newGame();
}

protected void initializeControls()
{
	//reset
	getInputMap().put(KeyStroke.getKeyStroke("R"), "resetButton");
	getActionMap().put("resetButton", new ResetAction());
	//pause
	getInputMap().put(KeyStroke.getKeyStroke("P"), "pauseButton");
	getActionMap().put("pauseButton", new PauseAction());
}

protected void newGame()
{
	initializePipes();

	pipeSpeed = 5;
	closestPipeIndex = 0;
	numPipes = 3;

	paused = false;

	populate();

	initializeTimer();
}

protected void initializePipes()
{
	int adj = pipeHeightAdjustment();

	topPipe[0] = new TopPipe(Config.FIRST_PIPE_XLOC, adj);
	bottomPipe[0] = new BottomPipe(Config.FIRST_PIPE_XLOC, adj);

	adj = pipeHeightAdjustment();

	topPipe[1] = new TopPipe(Config.SECOND_PIPE_XLOC, adj);
	bottomPipe[1] = new BottomPipe(Config.SECOND_PIPE_XLOC, adj);
```

```
                adj = pipeHeightAdjustment();

                topPipe[2] = new TopPipe(Config.THIRD_PIPE_XLOC, adj);
                bottomPipe[2] = new BottomPipe(Config.THIRD_PIPE_XLOC, adj);
        }

        protected int pipeHeightAdjustment()
        {
                return (rando.nextInt(20) * 15) - 150;
        }

        protected void populate()
        {
                fish = new ArrayList<Fish>();

                for (int i = 0; i < 10; i++)
                {
                        fish.add(new Fish());
                }
        }

        protected void initializeTimer()
        {
                timer = new Timer();
                timer.schedule(new GameLoop(), gameSpeed, gameSpeed);
        }

        protected void gameLogic()
        {
                fishLogic();
                pipeLogic();
                additionalLogic();

                if (fish.size() == 0)
                {
                        timer.cancel();
                }
        }

        protected void additionalLogic()
        {
        }

        private void fishLogic()
        {
                for (Fish f : fish)
                {

                        if (f.isDead()) //skip this iteration if the fish is dead
                        {
                                continue;
                        }

                        if      (   //kill the fish if it goes out of bounds, touches a
pipe
                                        f.collides(getClosestTopPipe())
                                        || f.collides(getClosestBottomPipe())
                                        || f.getY() > Config.VERTICAL_RES
                                        || f.getY() < -25
                                )
                        {
                                f.kill();
                        }

                        if (f.isDead()) //skip if the fish was just killed
                        {
                                continue;
```

```
                        }

                        f.move(); //move the fish according to its current action

                        if (f.getX() > getClosestTopPipe().getX())
                        {
                                f.score();    //grant the fish points for passing through a
pipe
                        }

                        if (    f.getX() > getClosestTopPipe().getX() - 225
                                        && (f.getY() < getClosestTopPipe().getBottomY()
                                        || (f.getBottomY()) >
getClosestBottomPipe().getY())) //
                        {
                                f.penalize(); //penalize the fish for being on a collision
course
                        }
                }
        }

        private void pipeLogic()
        {
                for (int i = 0; i < bottomPipe.length; i++)
                {
                        if (numPipes > Config.MAX_PIPES)
                        {
                                for (Fish f : fish)
                                {
                                        f.kill();
                                }
                        }

                        if (getClosestTopPipe().getX() < -25)
                        {
                                //change the "closest" pipe to the next pipe in the
sequence
                                closestPipeIndex = (closestPipeIndex + 1) % 3;
                                numPipes++;
                        }
                                // if the pipe has traveled far enough off-screen to the
left,
                                // reposition it at the right of the screen to begin anew

                        if (bottomPipe[i].getX() < -400)
                        {
                                int adj = pipeHeightAdjustment();

                                bottomPipe[i].setX(800);
                                bottomPipe[i].setY(adj);

                                topPipe[i].setX(800);
                                topPipe[i].setY(adj);
                        }
                        else
                        {
                                for (int j = 0; j < pipeSpeed; j++)
                                {
                                        bottomPipe[i].moveLeft();
                                        topPipe[i].moveLeft();
                                }
                        }
                }
        }

        protected class GameLoop extends TimerTask
        {
```

```
                public void run()
                {
                        gameLogic();
                        repaint();
                }
        }

        protected class PauseAction extends AbstractAction
        {
                public void actionPerformed(ActionEvent e)
                {
                        if (!paused)
                        {
                                timer.cancel();
                        }
                        else
                        {
                                timer = new Timer();
                                timer.schedule(new GameLoop(), gameSpeed, gameSpeed);
                        }
                        paused = !paused;
                }
        }

        protected class ResetAction extends AbstractAction
        {
                public void actionPerformed(ActionEvent e)
                {
                        timer.cancel();
                        newGame();
                }
        }

}
```

ManualBoard.java

```
package game;
import java.awt.event.ActionEvent;
import java.util.ArrayList;

import javax.swing.AbstractAction;
import javax.swing.KeyStroke;

public class ManualBoard extends Board
{

	public ManualBoard()
	{
		super();
		initializeManualControls();
	}

	private void initializeManualControls()
	{
		//swimming
		getInputMap().put(KeyStroke.getKeyStroke("SPACE"), "swimButton");
		getActionMap().put("swimButton", new SwimAction());
		getInputMap().put(KeyStroke.getKeyStroke("released SPACE"),
"swimRelease");
		getActionMap().put("swimRelease", new NeutralAction());
		//diving
		getInputMap().put(KeyStroke.getKeyStroke("DOWN"), "diveButton");
		getActionMap().put("diveButton", new DiveAction());
		getInputMap().put(KeyStroke.getKeyStroke("released DOWN"), "diveRelease");
		getActionMap().put("diveRelease", new NeutralAction());
	}

	protected void populate()
	{
		fish = new ArrayList<Fish>();
		fish.add(new Fish());
	}

	private class SwimAction extends AbstractAction
	{
		public void actionPerformed(ActionEvent e)
		{
			{
				for (Fish f : fish)
				{
					f.swim();
				}
			}
		}
	}

	private class NeutralAction extends AbstractAction
	{
		public void actionPerformed(ActionEvent e)
		{
			{
				for (Fish f : fish)
				{
					f.neutral();
				}
			}
		}
	}

	private class DiveAction extends AbstractAction
```

```
        {
                public void actionPerformed(ActionEvent e)
                {
                        for (Fish f : fish)
                        {
                                f.dive();
                        }
                }
        }
}
```

NeuralBoard.java

```
package agent;

import java.util.ArrayList;
import java.util.Scanner;
import java.util.Vector;

import jneat.*;

import game.Board;
import game.Config;
import game.Fish;

public class NeuralBoard extends Board
{
	protected int generation;
	protected Population neatPop;
	protected Vector orgs;

	public NeuralBoard()
	{
		Neat.initbase();

		neatPop = new Population
				(
					50,		/* population size */
					3,			/* network inputs */
					1,			/* network outputs */
					10,		/* max index of nodes */
					true,		/* recurrent */
					0.5		/* probability of connecting
two nodes */
				);

		orgs = neatPop.getOrganisms();
		generation = 0;
		gameSpeed = 6;

		loadFromFile();
		initialize();
	}

	protected void loadFromFile()
	{
		Scanner scan = new Scanner(System.in);
		System.out.println("Would you like to load from a file? Y/N");

		if (scan.nextLine().toUpperCase().equals("Y"))
		{
			System.out.print("Enter file name: ");
			neatPop = new Population(scan.nextLine());

			System.out.println();

			System.out.print("Enter generation number: ");
			generation = scan.nextInt();
		}

		Neat.readParam("parametri");
	}

	protected void populate()
	{
		fish = new ArrayList<Fish>(25);
```

```
		for (int i = 0; i < 50; i++)
		{
			fish.add(new Fish());
		}
	}

	private double normalizeY(int y)
	{
		return ((double) y / (double) Config.VERTICAL_RES);
	}

	private double normalizeX(int x)
	{
		return ((double) x / (double) Config.HORIZONTAL_RES);
	}

	public void simulate()
	{
		orgs = neatPop.getOrganisms();

		for (int i = 0; i < orgs.size(); i++)
		{
			if (fish.get(i).isDead())
			{
				continue;
			}

			Network brain = ((Organism)orgs.get(i)).getNet();

			double[] inputs =
				{
					normalizeX(getClosestBottomPipe().getX()),
					normalizeY(fish.get((i)).getY()),
					normalizeY(getClosestBottomPipe().getY())
					//normalizeY(getClosestTopPipe().getY()),
				};

			brain.load_sensors(inputs);

			int net_depth = brain.max_depth();

			brain.activate();

			for (int relax = 0; relax <= net_depth; relax++)
			{
				brain.activate();
			}

			double output = ((NNode)
brain.getOutputs().elementAt(0)).getActivation();

			if (output > 0.67)
			{
				fish.get(i).swim();
			}
			else if (output > 0.33)
			{
				fish.get(i).neutral();
			}
			else
			{
				fish.get(i).dive();
			}
		}
	}

	@SuppressWarnings("unchecked")
```

```
	protected void evaluate()
	{
		//Vector<Organism>
		orgs = neatPop.getOrganisms();

		for (int i = 0; i < orgs.size(); i++)
		{
			double score = (double) fish.get(i).getScore();

			((Organism)orgs.get(i)).setFitness(score);

			if (score > Config.MAX_SCORE)
			{
				((Organism)orgs.get(i)).setWinner(true);
			}

			if (score < -1000)
			{
				((Organism)orgs.get(i)).setEliminate(true);
			}
		}

		Vector species = neatPop.getSpecies();

		for (int i = 0; i < species.size(); i++)
		{
			Species s = (Species) species.elementAt(i);

			int index = 0;
			double max = 0;

			for (int j = 0; j < s.getOrganisms().size(); j++)
			{
				Organism o = (Organism)s.getOrganisms().elementAt(j);

				if (o.getFitness() > max)
				{
					max = o.getFitness();
					index = j;
				}

			}

			((Organism)s.getOrganisms().elementAt(index)).setChampion(true);

			s.compute_average_fitness();
			s.compute_max_fitness();

			if (s.getMax_fitness() > s.getMax_fitness_ever())
			{
				s.setMax_fitness_ever(s.getMax_fitness());
			}
		}

		generation++; //increase generation number

		neatPop.viewtext(); //print diagnostics

		neatPop.print_to_file_by_species("generation" + generation); //print to
file

		neatPop.epoch(generation); //begin new generation
	}

	protected void additionalLogic()
	{
		simulate();
```

```
                boolean eval = true;

                for (int i = 0; i < fish.size(); i++)
                {
                        if (!fish.get(i).isDead())
                        {
                                eval = false;
                        }
                }

                if (eval)
                {
                        System.out.println("GENERATION: " + generation);
                        timer.cancel();
                        evaluate();
                        populate();
                        newGame();
                }
        }
}
```