Research Report on the Topic of

Artificial Intelligence and Games

For

Advanced Technical Elective I (MECH74100) Course – 2018

Submitted By

Thomas Abdallah 7141518

David Eelman 6365316

Stanislav Rashevskyi 7028178

Submitted in

August *2018*

ELECTRONIC SYSTEMS ENGINEERING (ESE) PROGRAM
CONESTOGA COLLEGE

Table of Contents

Introduction **3**
Background to the Topic 3
Goal of the Research Project 3
Methodology 3
Outline of the Remaining Parts of the Document 3
Literature Review **5**
Overview and History of AI Research 5
Examples of Videos Games Played by ANNs 5
Artificial Neural Networks (ANNs) **7**
History 7
Structure 8
Input Layer 9
Hidden Layer 9
Output Layer 9
Mathematical Theory 9
Artificial Neural Networks & Games **12**
History 12
Monte Carlo Tree Search and Reinforcement Learning 13
Genetic Algorithms 15
Experimentation & Results **17**
Conclusion **20**
Appendix - Air Hockey Simulation Code **21**
References **22**

Introduction

Background to the Topic

Programming computers to play games has been a popular area of research for decades. By using Artificial Intelligence techniques that can mimic human behavior, more complex games can be played.

Goal of the Research Project

This research project investigates machine learning methods used to allow computers to play games.

Methodology

Our research was conducted in three phases. First, we researched Artificial Neural Networks and the mathematics and statistics supporting neural network theory in order to understand the underlying concepts. Second, we investigated recent examples of machine learned game applications in order to understand what techniques have been successful for this type of work. We documented the pros and cons of various methods and make an informed decision about which method is most appropriate for our application (simulated air hockey game). Finally, if time allows we created a simulated air hockey game and prototype an application using our selected machine learning method.

Outline of the Remaining Parts of the Document

In the sections that follow, our group describes:

- A literature review of the overview and history of AI research as well as examples of video games played by Artificial Neural Networks
- The researched theory behind modern neural networks including the mathematical and statistical theory supporting them.
- The researched methods for structuring and training neural networks to play simple games.

- The applied method used to prototype a computer program that can play a simulated air hockey game.

Literature Review

Overview and History of AI Research

The article Artificial Intelligence [1] provides a thorough overview of the history of Artificial Intelligence (AI) and some of the differences in approach the industry has developed. The article discusses two fundamentally different approaches to AI development, the top-down (symbolic) approach and the bottom-up (connectionist) approach. Artificial Neural Networks (ANNs) are an example of the bottom up approach which imitate the structure of the brains neurons and their connections. The bottom-up approach relies on training the network by strengthening certain neural connections to improve the performance of the network in completing the intended task. Both bottom-up and top-down approaches have been the focus of much research in the past, but this project will focus on techniques for implementing the bottom-up approach as this has been widely used in recent years throughout various industries.

This paper also discusses the first successful AI program which was developed at the University of Oxford and could play a game of checkers at a reasonable speed by the summer of 1952. A researcher in the United States further developed this checkers program and notably implemented one of the first efforts of evolutionary computing whereby a modified copy of the program would play against the current best version and the winner would become the new standard. Through this method a highly optimized solution will evolve over multiple generations. This method is likely to be of interest for our project as playing a game such as air hockey is not a simple task that can be trained using a pre-annotated data set.

Examples of Videos Games Played by ANNs

The article Write an AI to win at Pong from scratch with Reinforcement Learning [2] shows how to implement an ANN for Reinforcement Learning (RL) that's able to beat a computer in a game of Pong. RL is a learning method based on not telling the system what to do, but only what is right and what is wrong. The author utilizes OpenAI Gym [3], which is a "toolkit for developing and comparing RL algorithms. It supports teaching agents everything from walking to playing games like Pong or Pinball”. The article shows how to create ANN from the scratch, implement a Policy Gradient with RL, build an AI for the game in Python in less than 250 lines, and use OpenAI Gym. The code for the Pong’s ANN consists of Initialization, How to Move, and Learning blocks. The author states that it takes ~3 days on a Macbook for the ANN to start beating computer in Pong. Overall, this article seems to be very relevant to our investigation, and explains well a common subfield of machine learning for video games (RL), together with OpenAI Gym toolkit.

Using Machine Learning Agents in a real game: a beginner’s guide [4] is a blog article that demonstrates how Unity Machine Learning Agents use Reinforcement Learning (RL) model to train an ANN to play 2D and 3D games. Their system also uses an additional library “TensorFlow machine learning library” to allow their agents to communicate with an external Python environment. This article shows how powerful and easy it is to work with Unity’s Machine Learning API Agents.

Artificial Neural Networks (ANNs)

History

In 1943 neurophysiologist Warren McCulloch and mathematician Walter Pitts decided to simulate intelligent brain behaviour. This concept termed “connectionism”, and utilized electrical circuits to model brain neurons. Their model is still used today in Artificial Neural Networks (ANN). Two key parts of this model: summation over weighted inputs and an output function of the sum.

In 1949 Donald Hebb continued this idea in a book “The Organization of Behaviour” outlining a law for synaptic neuron learning. Later this law was called “Hebbian Learning” saying that “Cells that fire together, wire together.” In 1954 at MIT, Farley and Clark were first to use computational machines (calculators), to simulate a Hebbian network.

The Mark I Perceptron was designed in 1958 by Frank Rosenblatt. The Perceptron was created to use neural network for character recognition. However, the Perceptron only learnt to separate linearly separable classes, when the simple exclusive-or (XOR) circuit was an irresistible barrier.

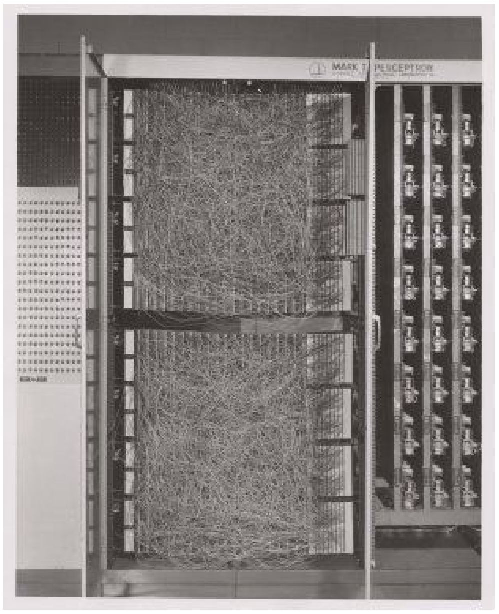

Figure 1 - First known implementation of a Mark I Perceptron

Bernard Widrow and Marcian Hoff in 1959 made a huge step in ANN, they created the first network successfully applied to a real world problem. The system could eliminate noise in phone lines and still remains in use today. Moreover, the artificial neurons differed from the Perceptron's in what they returned as output, which in this case was the weighted input.

In 1974 Werbos originally discovered the Backpropagation algorithm, but it wasn't publicly recognized. Later in 1986 Rumelhart, Hinton and Williams rediscovered the concept in their book "Learning Internal Representation by Error Propagation". Backpropagation is a form of the Gradient Descent algorithm used with ANN for minimization and curve-fitting. This algorithm remains the backbone and powerhouse of neural networks today.

In 1987 the IEEE international ANN conference was launched for the AI enthusiasts. In 1987 the International Neural Network Society (INNS) was formed, along with the INNS Neural Networking journal in 1988. By 1990's, things for neural networks started moving quickly, and the ANN started being applied in various fields.

Structure

Artificial Neural Networks are composed of three essential layers: the input layer, the hidden layer, and the output layer (See Figure 2).

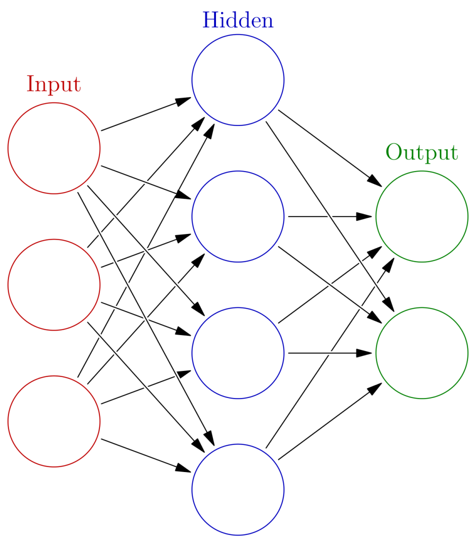

Figure 2 - Basic Artificial Neural Network Structure

Input Layer

The input layer consists of neurons collecting real world information that we are attempting to classify and passing it through to the hidden layer. The sources of data can range from electronic sensors to data files.

Hidden Layer

The hidden layer consists of neurons which process the input information and act upon it based on weightings determined by training the neural network. Each of the 9 circles shown in Figure 2 represents a neuron and they can be explained as functions that take the output of all previous layers neurons as an input and output a single value based on an activation function explained in the mathematical theory. The hidden layer often consists of numerous sub-layers of neurons with each performing a different aspect of the data processing. Once the data has passed through the hidden layer, it is finally forwarded on to the output layer.

Output Layer

The output layer consists of neurons which hold the finished computations of the artificial neural network. The output represents some function of the hidden layer(s) and it’s form is directly related to the network type and user modifiable parameters of the network.

Mathematical Theory

The basis of ANNs is the neuron, which performs calculations on multiple inputs to produce a single output. Each input is multiplied by a weighting factor and the resulting values are added together along with a bias value. In some cases it may be desirable to restrict the neurons output value by applying what is referred to as an “Activation function”. For example, it may be desirable to limit the output of a neuron to a value between 0 and 1. Figure 3 graphically depicts the function of a Neuron.

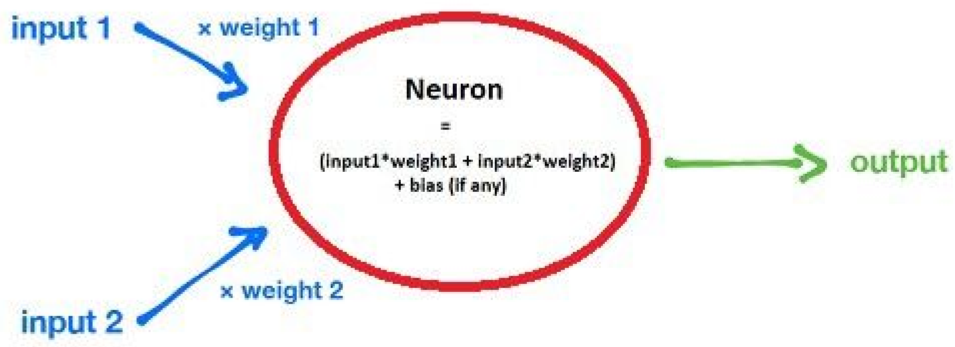

Figure 3 - Graphical Depiction of a Neuron

The input layer will contain as many neurons as there are features in the dataset, plus an additional 'bias' neuron. The hidden layer(s) should contain at least as many neurons as the input layer, plus an additional 'bias' neuron. The output layer will contain one or more neurons based on the structure of the problem. For example, an ANN attempting to reproduce the functionality of an AND gate would have two inputs and one output. Therefore the input layer would have three neurons (two inputs + bias), the hidden layer(s) would have at least four neurons, and the output layer would have one output - the binary output of the circuit.

Training an ANN involves four distinct steps.

1. Select a network architecture and initialize all connections between neurons with random weights. The network architecture is simply the definition of how many neurons are in each layer, and how many hidden layers are in the network.

2. Pass some input data through the layers of the ANN, with the outputs of each layer being fed into the inputs of the next layer. This is referred to as "forward propagation" and will produce outputs that have some amount of error.

3. Calculate the total error of the outputs, that is, how far the ANNs outputs are from the expected output. For an ANN with multiple outputs the total error is some function of the individual output errors. Most ANNs must be able to correctly calculate the output with varying inputs, so the total error is actually the sum of the errors produced by each possible set of input values. This value is referred to as the

‘cost function’ and the goal of training the ANN is to minimize the cost function so that the output for all possible inputs has minimal error.

4. Update each of the weights in the ANN by “back propagation” to reduce the total error of the ANNs output. Back propagation uses the “gradient descent” method to minimize the cost function by making small changes to the weights of each neuron that will result in the cost function decreasing.

5. Repeat steps 2-4 until the cost function reaches an acceptably low value.

Artificial Neural Networks & Games

History

Artificial intelligence has been used in games to create realistic opponents for human players to play against. Many techniques have been employed throughout the years to achieve human-like behavior for video games' virtual opponents. Among the first computer programs ever written was a program to play checkers and another to play chess. Early video games such as Pong were implemented using discrete logic for the virtual opponent, not using any academic AI techniques. As games got more complex the actions in-game opponents became more sophisticated, but still relied on traditional rules-based programming techniques rather than modern academic AI techniques such as Artificial Neural Networks (ANNs).

Recent research in the video game industry has focused on applying modern AI developments to various areas of video game development, including generating new content and player-experience modelling to better tailor the in-game experience to a specific user.

Many researchers outside of the video game industry are working on adapting AI techniques to allow computer programs to play video games intelligently rather than with a rules-based approach. As discussed in the background research paper, ANNs are trained on large datasets in order to minimize some measurable cost function. In playing a video game there is often not an obvious source of training data or a clear-cut cost function that can be minimized. This means that traditional techniques for training ANNs to perform a task cannot be applied to most video games. Current research has focused on implementing unsupervised training techniques where the ANN plays the video game against itself or some other virtual opponent. The ANNs cost function is based on the success of its in-game play, and in this way a large set of pre-annotated training data is not needed. Several techniques for unsupervised learning will be covered in this report.

Monte Carlo Tree Search and Reinforcement Learning

A Monte Carlo Tree Search is a heuristic search algorithm used to make optimal decisions. It's often utilized in move planning for games, implementing a four step process to make decisions. Figure 4 depicts the four steps: Selection, Expansion, Simulation, and Backpropagation.

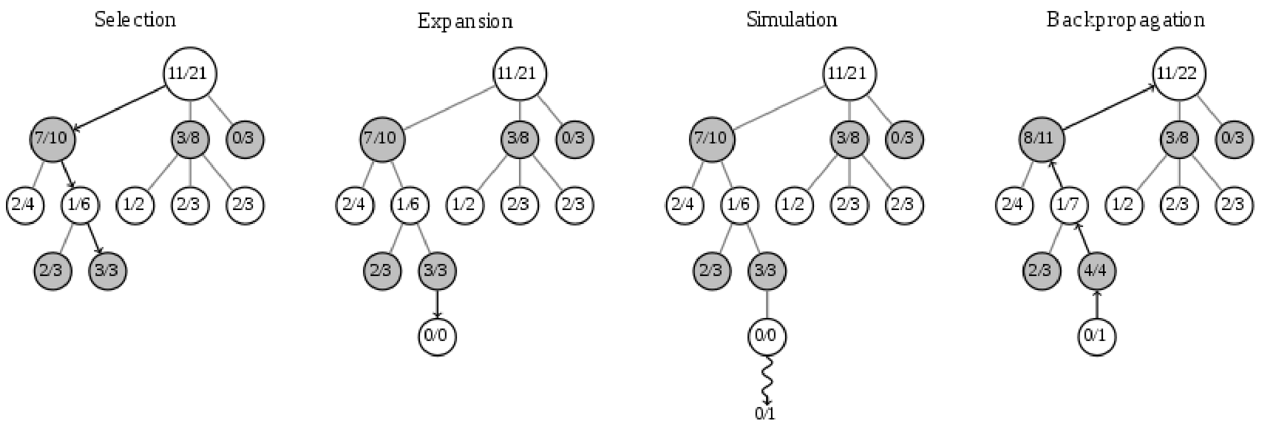

Figure 4 - Monte Carlo Tree Search Algorithm

1) **Selection** - Start at the root and traverse one path of the tree down to a leaf node. The path selection procedure is a core element of the algorithm.

2) **Expansion** - As long as this leaf node isn't the end of the game, add one or more child nodes and select one of these child nodes

3) **Simulation** - Begin a simulation starting from the selected child node all the way through to the end of the game. The playout path in this case is random.

4) **Backpropagation** - Update the traversed path from the selected child node all the way to the root node with the results of selecting this path.

The above process is repeated until the time allotted for a move expires, and then the move with the most simulations made is selected.

Reinforcement Learning is a method of machine learning that produces feedback on past decisions through trial and error to adjust decisions to be made in the future. Figure 5 shows the basic structure of reinforcement learning, where each action made by the Agent is assessed a reward/punishment based on the outcome in the Environment and is then fed back to the Agent. This feedback loop allows the Agent to adjust it’s action selection in order to maximize the reward produced by the Environment.

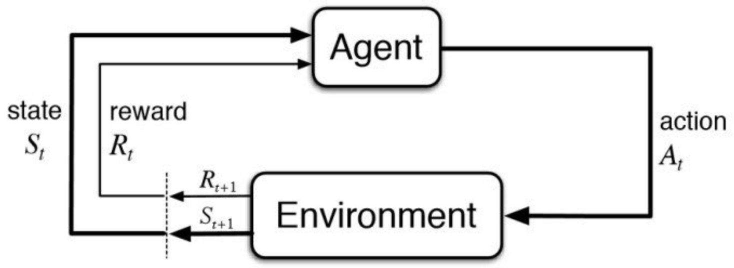

Figure 5 - Reinforcement Learning Loop

The selection process mentioned in the Monte Carlo Tree Search Algorithm is based on Reinforcement Learning to make the optimal path traversal selection for winning the game.

AlphaGo is a computer program designed by Google DeepMind to play the board game Go. Go is a two player strategy board game where the goal is to surround more territory than the opposing player. AlphaGo uses a combination of machine learning and tree search algorithms to make playing decisions and has received significant publicity in the AI community for successfully defeating a professional Go player in 2016. Fun fact: AlphaGo runs on 1202 CPUs and 176 GPUs demonstrating the advantage of parallel computing. AlphaGo utilized the Monte Carlo Tree Search algorithm to run game play simulations and Reinforcement Learning methods to train the Artificial Neural Network to be capable of playing Go within the rules and time constraints.

Let’s consider how these methods might be applied to develop a system that can play an air hockey game. The objective of air hockey is to score more goals in the opponents net than they score in your net. There are boundless ways that this goal can be achieved so in order to implement a Monte Carlo Tree Search algorithm to simulate game play, it’d likely be beneficial to break up the playing surface into very small sections - let’s say 5mm by 5mm squares - in order to reduce granularity and simplify the tree structure. From there, we’d select a starting point for the MCTS algorithm and begin simulating different paddle positions used to hit the puck, recording which positions were optimal for goal scoring and defending. Using this collected data we could implement Reinforcement Learning to train a neural network how to best play air hockey.

Genetic Algorithms

A genetic algorithm is a technique inspired by the process of natural selection that can be used to solve optimization and search problems [18]. Genetic algorithms have been applied to many different types of problems, including training ANNs to accomplish complex tasks such as playing video games. Training an ANN with a genetic algorithm is a “trial and error” approach to learning where many different versions of the ANN are tested and the most successful are used to spawn future “generations” of the ANN until an acceptable version is created.

A basic application of a genetic algorithm in training an ANN is as follows:

1. Define the structure of the ANN to be trained.
2. Generate a random “population” of ANNs by initializing neural connections with random weights.
3. Evaluate the performance of each ANN.
4. Select the ANN with the highest performance and use it to generate a new population of ANNs by applying random mutations to each member of the new population.
5. Repeat steps 3-4 until an ANN with acceptable performance is generated.

The Youtube channel “Code Bullet” [14] has documented several successful applications of this technique in training ANNs to play video games of varying difficulty including PACMAN, Asteroids, Minesweeper, Snake, and more.

To apply this technique to train an ANN to play an air hockey game the main challenge would be identifying the structure of the ANN to be trained, specifically the inputs and outputs of the ANN. One such structure is depicted in Figure 6. Once the ANN structure is defined the genetic algorithm could be applied to evaluate the performance of each ANN in a simulated air hockey game, where the ANN with the highest goals for/against ratio would survive to spawn the next population of ANNs.

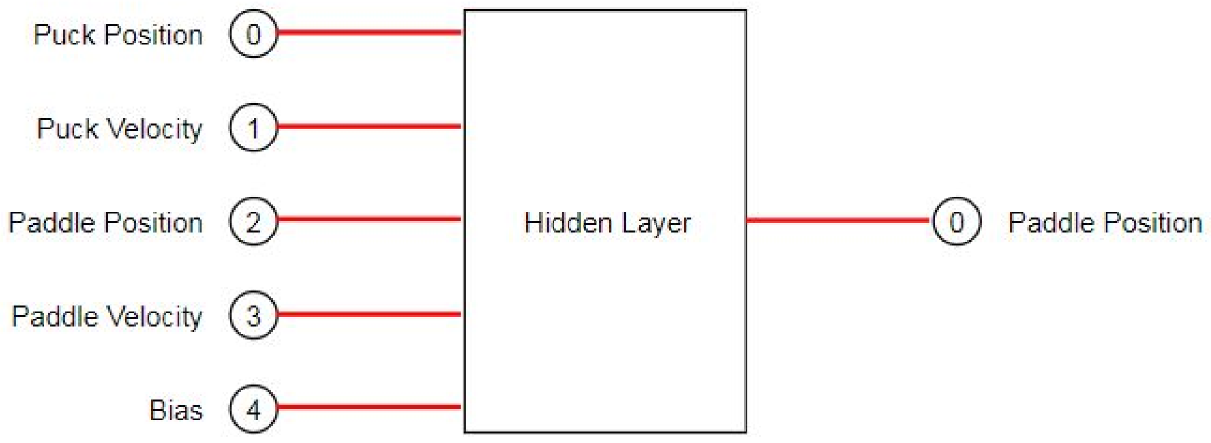

Figure 6 - ANN Structure for an Air Hockey Game

Experimentation & Results

An artificial neural network was designed that outputs a position command for the AI air hockey paddle and takes as inputs AI paddle position, AI paddle velocity, puck position, and puck velocity (Figure 7).

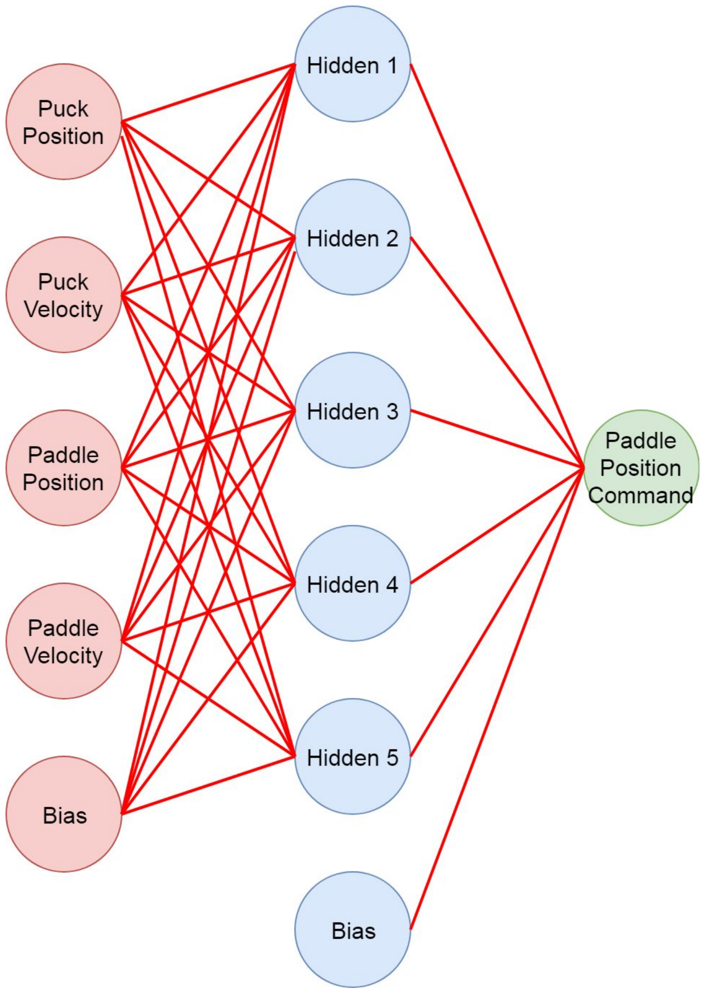

Figure 7 - Artificial Neural Network Structure

The neural network was prototyped from scratch using the “Processing” programming language. “Processing” is based on Java and provides an Integrated Development Environment (IDE) with user-friendly visualization tools to allow for quick prototyping of visual applications. A simulated air-hockey game was developed to allow the neural network to play against an opponent and evolve over time (Figure 8).

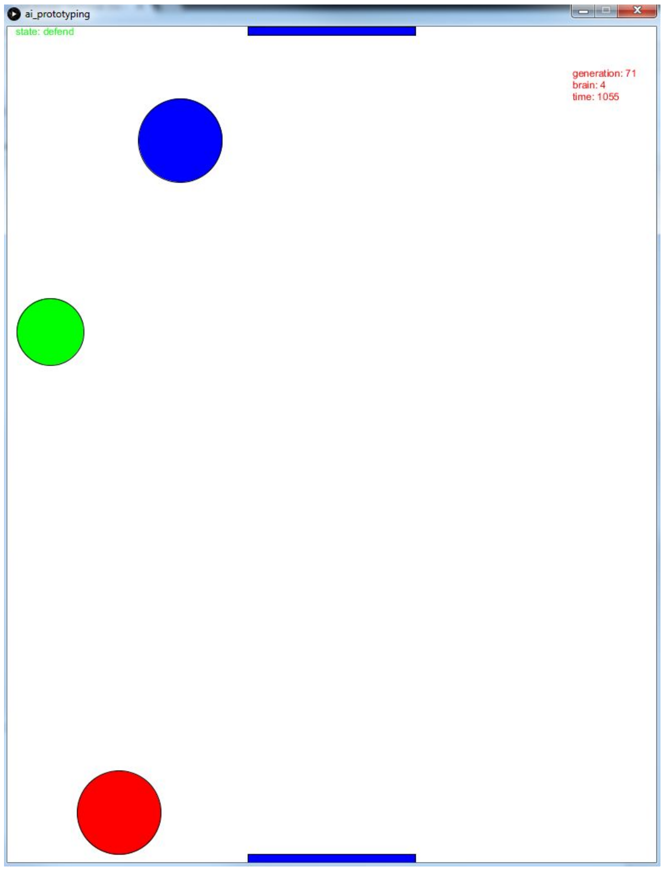

Figure 8 - Visualization of Simulated Air Hockey Game

The AI was trained using a genetic algorithm where each “generation” had five copies of the neural network with different connection weights hereby referred to as “brains”. Each brain played a game of air hockey against a virtual opponent which was based off of a state-based control strategy developed as a part of the group members capstone project. Each game lasted until a goal was scored or some predefined time limit was reached. A fitness function for each brain was calculated based on whether the AI won or lost the game, how long it took for the AI to score a goal (or be scored on), how fast on average the AI moved its paddle, and how often the AI redirected the virtual air hockey puck. The brain with the highest fitness score was declared to be the “best brain” and was used to spawn a new generation of brains. In each generation one unmodified copy of the prior generations “best brain” was kept and a number of copies of that brain were made and then slightly mutated to create a new generation of unique brains. The mutations are performed by randomly changing one or more neural connection weights in each copy of the “best brain”.

Initial attempts resulted in an AI that would simply sit stationary in front of its net to prevent the virtual opponent from scoring a goal. The fitness function was modified to encourage the AI to move its' paddle more by increasing the weight of the AI paddle average speed in the fitness evaluation. The AI evolved over time to successfully play defense against the virtual opponent, but in our limited prototyping time we were unable to witness the AI evolve competitive "offense" strategies to try and score goals.

This early prototyping is very promising as with a relatively small amount of work (<30 hours) we were able to develop an AI capable of learning simple air hockey strategies by itself with no predefined training data. Further research is required to create an AI capable of scoring goals. Specifically, further development of the fitness function and strategies for mutating the brains in each generation could lead to a more targeted evolution process rather then the basic approach we have currently implemented.

Conclusion

In conclusion our research and prototyping has demonstrated that it is feasible to develop an AI capable of playing air hockey at a basic level. The unsupervised learning techniques that were researched are an active area of development in academia and in industry and will likely see numerous advances in the coming years. Our prototyping with a simple neural network trained with a genetic algorithm was successful in proving that an “analog” game such as air hockey could conceivably be played by an AI. Further refinement of our neural network and our genetic algorithm should lead to improved performance of the AI at playing air hockey.

Appendix - Air Hockey Simulation Code

Code is written in the "Processing" language. The IDE to run this code can be downloaded here: https://processing.org/download/

ai_prototyping.pde (Main function for the program)

```
// All dimensions in millimetres to match real table
Puck puck;
Paddle human_paddle;
Paddle robot_paddle;
Human human_player;
Brain[] brains;
int number_brains = 5;
int current_brain = 0;
int generation = 1;
int max_game_time_ms = 10000;
int goal_width_x = 200;
int goal_height_y = 10;
PVector pos_old;

//-----------------------------------------------------------------------------------------------------------------
// Gets called at the very beginning of the program
void setup()
{
  size(775, 1000);  // size of the window
  frameRate(10000);  // Run as fast as the CPU will handle
  puck = new Puck();
  puck.vel.x = 1;
  puck.vel.y = 1;
  human_paddle = new Paddle(false);
  human_paddle.robot = false;
  robot_paddle = new Paddle(true);
  robot_paddle.robot = true;
  human_player = new Human();
  brains = new Brain[number_brains];
  pos_old = new PVector(0, 0);
  for (int i=0; i < number_brains; i++) {
      brains[i] = new Brain();
  }
  println("Game started");
}

//---------------------------------------------------------------------------------------------------------------
// Gets called at whatever the framerate is
void draw()
{
  int best_brain = 0;
  int number_of_mutations = 1;
  int similar_brains = 0;
  Brain best_brain_copy = new Brain();

  // Let the brains play
  play(current_brain);
  if (puck.goal_robot) {
    brains[current_brain].robot_victory = true;
    reset();
  } else if (puck.goal_human) {
    brains[current_brain].robot_victory = false;
    reset();
  }

  // Check if the generation is complete
  if (current_brain >= number_brains) {
    print("generation ");
    print(generation);
    println(" complete");

    // Pick the best brain
    for (int i=0; i < number_brains; i++) {
      // Calculate the "loser fitness" i.e. how long did it last and how high was its average
speed
      brains[i].loser_fitness = (brains[i].time_count_ms * 10) + brains[i].average_dist_to_puck;

      // Calculate the "winner fitness" i.e. how long did it take to win
      brains[i].winner_fitness = ((max_game_time_ms - brains[i].time_count_ms) * 10);

      // First priority: brains that win (higher fitness score is better)
      if (brains[best_brain].robot_victory) {
        if (brains[i].robot_victory) {
          if (brains[i].winner_fitness > brains[best_brain].winner_fitness) {
            best_brain = i;
          }
        }
      } else {
        // Second priority: Brains that last the longest and have highest average speed (higher
fitness score is better)
        if (brains[i].loser_fitness > brains[best_brain].loser_fitness) {
          best_brain = i;
        }
      }
    }

    // Make sure all the brains aren't the same
    if (!brains[best_brain].robot_victory) {
      for (int i=0; i < (number_brains-1); i++) {
        if (brains[i].loser_fitness == brains[i+1].loser_fitness) {
          similar_brains++;
        }
      }
    }
    if (similar_brains >= (number_brains-1)) {
      println("All brains are the same, big mutation");
      best_brain = int(random(0, number_brains-1));
      number_of_mutations = 2;
    }

    // Copy the best brain and mutate each child
    brains[best_brain].redirections = 0;
    brains[best_brain].average_speed = 0;
    brains[best_brain].average_dist_to_puck = 0;
    best_brain_copy = brains[best_brain].copy();
    for (int i=1; i < number_brains; i++) {
      brains[i].log(i);
      brains[i] = best_brain_copy.copy();
      brains[i].mutate(number_of_mutations);
    }
    brains[0] = best_brain_copy;  // Keep one unmutated copy
    print("best brain: ");
    println(best_brain);
    generation ++;
    reset();
    current_brain = 0;
  }
}

//-----------------------------------------------------------------------------------------------------------
// Logic to play a game of air hockey with visualization
void play(int brain)
{
  PVector dist_to_puck = new PVector(0, 0);
  String info_text;
  info_text = "generation: " + generation + "\nbrain: " + brain + "\ntime: " +
(brains[brain].time_count_ms);

  background(255);
  text(info_text, width - 100, 60);

  //draw goals
  fill(0, 0, 255);
  rect((width/2) - (goal_width_x/2), (height - goal_height_y), goal_width_x, goal_height_y);  //
Robot goal
  rect((width/2) - (goal_width_x/2), 0, goal_width_x, goal_height_y);  // Human goal

  // draw puck and paddles
  brains[brain].update();
  puck.update(brains[brain]);
  puck.show();
  human_player.update(human_paddle.pos, puck.pos, puck.vel);
  human_paddle.update(human_player.pos_command);
  human_paddle.show();
  robot_paddle.update(brains[brain].output_command);
  robot_paddle.show();

  // End the game if a goal is scored
  if (puck.goal_robot) {
    print("Brain: ");
    println(brain);
    println("Robot wins");
    println(brains[brain].time_count_ms);
    println(brains[brain].average_speed);
    println(brains[brain].redirections);
    println(brains[brain].average_dist_to_puck);
    return;
  }
  if (puck.goal_human) {
    print("Brain: ");
    println(brain);
    println("Human wins");
    println(brains[brain].time_count_ms);
    println(brains[brain].average_speed);
    println(brains[brain].redirections);
    println(brains[brain].average_dist_to_puck);
    return;
  }

  if (brains[brain].time_count_ms >= max_game_time_ms) {
    // Robot can't take forever to win or lose
    print("Brain: ");
    println(brain);
    println("Robot took too long");
    println(brains[brain].average_speed);
    println(brains[brain].redirections);
    println(brains[brain].average_dist_to_puck);
    puck.goal_human = true;
    return;
  }

  brains[brain].average_speed += abs(robot_paddle.pos.x - pos_old.x) +
abs(robot_paddle.pos.y - pos_old.y);
  if (puck.pos.y > height/2) {
    // Determine how close the paddle is to the puck (reverse logic, higher is better)
    dist_to_puck.x = robot_paddle.pos.x - puck.pos.x;
    dist_to_puck.y = robot_paddle.pos.y - puck.pos.y;
    brains[brain].average_dist_to_puck += (height/2) - (dist_to_puck.mag());
  }

  pos_old.x = robot_paddle.pos.x;
  pos_old.y = robot_paddle.pos.y;
}

void reset()
{
  current_brain ++;
  puck = new Puck();
  puck.vel.x = 1;
  puck.vel.y = 1;
  robot_paddle.pos.x = width/2;
  robot_paddle.pos.y = height - (paddle_diameter * 2);
  robot_paddle.vel.x = 0;
  robot_paddle.vel.y = 0;
  pos_old.x = 0;
  pos_old.y = 0;
}
```

brain.pde (Data structure for each instance of the neural network)

```
int brain_input_neurons = 9;
int brain_hidden_neurons = 10;
int brain_output_neurons = 2;

class Brain {
  PVector output_command;
  int time_count_ms;      // Keeps track of how long the game went
  int redirections;
  boolean robot_victory;  // True if the robot won the game
  float average_speed;
  float average_dist_to_puck;
  float loser_fitness;
  float winner_fitness;
  // Input order: puck_pos_x, puck_pos_y, puck_vel_x, puck_vel_y, paddle_pos_x,
paddle_pos_y, paddle_vel_x, paddle_vel_y, bias
  Neuron[] input;

  // Hidden layer neurons 1-9 are fed from input layer, 10 is bias
  Neuron[] hidden;

  // Output order: pos_cmd_x, pos_cmd_y
  Neuron[] output;

  Brain() {
    time_count_ms = 0;
    redirections = 0;
    robot_victory = false;
    output_command = new PVector(0, 0);
    average_speed = 0;
    average_dist_to_puck = 0;

    // All input layer neurons only have 1 input with no weight
    input = new Neuron[brain_input_neurons];
    for (int i=0; i < brain_input_neurons; i++) {
      input[i] = new Neuron(1);
      input[i].weights[0] = 1;
    }

    // Hidden layer neurons except for bias neuron take input from every input layer neuron
    hidden = new Neuron[brain_hidden_neurons];
    for (int i=0; i < brain_input_neurons; i++) {
      hidden[i] = new Neuron(brain_input_neurons);
    }
    hidden[brain_hidden_neurons-1] = new Neuron(1);  // Bias neuron in hidden layer

    // Output layer neurons take input from every hidden layer neuron
    output = new Neuron[brain_output_neurons];
    for (int i=0; i < brain_output_neurons; i++) {
      output[i] = new Neuron(brain_hidden_neurons);
    }
    output[0].upper_limit = width;
    output[0].lower_limit = 0;
    output[1].lower_limit = height/2;
    output[1].upper_limit = height;
  }

  //----------------------------------------------------------------------------------------------------------------
  // Calculates the output of the neural network
  void update() {
    // Set inputs for input layer
    input[0].input_data[0] = puck.pos.x;
    input[1].input_data[0] = puck.pos.y;
    input[2].input_data[0] = puck.vel.x;
    input[3].input_data[0] = puck.vel.y;
    input[4].input_data[0] = robot_paddle.pos.x;
    input[5].input_data[0] = robot_paddle.pos.y;
    input[6].input_data[0] = robot_paddle.vel.x;
    input[7].input_data[0] = robot_paddle.vel.y;
    input[8].input_data[0] = 1;  // Bias

    // Process input layer
    for (int i=0; i < brain_input_neurons; i++) {
      input[i].update();
    }

    // Set inputs for hidden layer
    for (int i=0; i < brain_input_neurons; i++) {
      for (int j=0; j < brain_input_neurons; j++) {
        hidden[i].input_data[j] = input[j].output;
      }
    }
    hidden[brain_hidden_neurons-1].input_data[0] = 1;

    // Process hidden layer
    for (int i=0; i < brain_hidden_neurons; i++) {
      hidden[i].update();
    }

    // Set inputs for output layer
    for (int i=0; i < brain_output_neurons; i++) {
      for (int j=0; j < brain_hidden_neurons; j++) {
        output[i].input_data[j] = hidden[j].output;
      }
    }

    // Process output layer
    for (int i=0; i < brain_output_neurons; i++) {
      output[i].update();
    }

    // Set output data
    output_command.x = output[0].output;
    output_command.y = output[1].output;

    time_count_ms ++;
  }

  //-----------------------------------------------------------------------------------------------------------------
  // Randomly selects an input weight to mutate (one in the hidden layer and one in the
output layer)
  void mutate(int number_of_mutations) {
    for (int i=0; i < number_of_mutations; i++) {
      int hidden_neuron_to_mutate = int(random(0, brain_input_neurons-1));  // Don't mutate
the bias neuron
      int output_neuron_to_mutate = int(random(0, brain_output_neurons-1));
      int hidden_weight_to_mutate = int(random(0, brain_input_neurons-1));
      int output_weight_to_mutate = int(random(0, brain_hidden_neurons-1));

      hidden[hidden_neuron_to_mutate].weights[hidden_weight_to_mutate] = random(-10,
10);
      output[output_neuron_to_mutate].weights[output_weight_to_mutate] = random(-10,
10);
    }
    redirections = 0;
  }

  //---------------------------------------------------------------------------------------------------------
  // Returns an exact duplicates of the brain
  Brain copy() {
    Brain foo = new Brain();
    for (int i=0; i < brain_input_neurons; i++) {
      foo.input[i] = input[i].copy();
    }
    for (int i=0; i < brain_hidden_neurons; i++) {
      foo.hidden[i] = hidden[i].copy();
    }
    for (int i=0; i < brain_output_neurons; i++) {
      foo.output[i] = output[i].copy();
    }
    return foo;
  }

  //---------------------------------------------------------------------------------------------------------
  // Records a copy of the brain in the brain_log.csv file
  void log(int brain_number) {
```

```
Table table = new Table();
table.addColumn("generation");
table.addColumn("brain_number");
table.addColumn("time_count_ms");
table.addColumn("average_speed");
table.addColumn("average_dist_to_puck");
for (int i=0; i < brain_hidden_neurons; i++) {
  for (int j=0; j < brain_input_neurons; j++) {
   table.addColumn("h" + str(i) + "w" + str(j));
  }
}
for (int i=0; i < brain_output_neurons; i++) {
  for (int j=0; j < brain_hidden_neurons; j++) {
   table.addColumn("o" + str(i) + "w" + str(j));
  }
}

TableRow newRow = table.addRow();
newRow.setInt("generation", generation);
newRow.setInt("brain_number", brain_number);
newRow.setInt("time_count_ms", time_count_ms);
newRow.setFloat("average_speed", average_speed);
newRow.setFloat("average_dist_to_puck", average_dist_to_puck);
for (int i=0; i < brain_hidden_neurons; i++) {
  for (int j=0; j < hidden[i].num_inputs; j++) {
   newRow.setFloat("h" + str(i) + "w" + str(j), hidden[i].weights[j]);
  }
}
for (int i=0; i < brain_output_neurons; i++) {
  for (int j=0; j < output[i].num_inputs; j++) {
   newRow.setFloat("o" + str(i) + "w" + str(j), output[i].weights[j]);
  }
}
```

```
    saveTable(table, "log_data/generation_" + str(generation) + ".csv");
  }
}
```

human_player.pde (Data structure and functions to create a virtual opponent)

```
class Human {
  final static int state_Home = 1;
  final static int state_Attack = 2;
  final static int state_Defend = 3;
  final static int state_Return = 4;
  final static float attack_line_y = 400;
  int state;
  PVector pos_command;
  boolean attack_move_active = false;

  Human() {
    state = state_Home;
    pos_command = new PVector(width/2, paddle_diameter * 2);
  }

  //------------------------------------------------------------------------------------------------------------
  // Logic for "human player" control
  void update(PVector paddle_pos, PVector puck_pos, PVector puck_vel) {
    float paddle_vel_y_calc = 0;
    float paddle_distance_to_attack_line_y = 0;
    float puck_distance_to_attack_line_y = 0;
    float paddle_time_to_attack_line_y = 0;
    float puck_time_to_attack_line_y = 0;
    PVector pos_command_calc = new PVector(0, 0);
    String debug_text;
    // Logic for 'home' state
    if (state == state_Home) {
      text("state: home", 10, 10);
      pos_command.x = width/2;
      pos_command.y = paddle_diameter/2;

      // Check conditions for state transitions
      if ((puck_pos.y > attack_line_y) && (puck_vel.y < 0)) {
        state = state_Attack;
        return;
      }
      if ((puck_pos.y < attack_line_y) && (puck_vel.y < 0)) {
         state = state_Defend;
         return;
      }
      if ((puck_pos.y < height/2) && (abs(puck_vel.y) < 1)) {
        state = state_Return;
        return;
      }
    }

    // Logic for 'attack' state
    if (state == state_Attack) {
      text("state: attack", 10, 10);

      if (attack_move_active) {
        // X-Axis position command should be a straight line between the puck, paddle and net
        pos_command_calc.x = puck_pos.x - (width/2);
        pos_command_calc.y = puck_pos.y - height;
        pos_command_calc.normalize();
        pos_command_calc = pos_command_calc.mult((paddle_diameter/2) +
(puck_diameter/2));
```

```
        pos_command.x = puck_pos.x + pos_command_calc.x;

        //pos_command.x = puck_pos.x;
        pos_command.y = attack_line_y;
      } else {
        // Calculate distance and time to attack line for paddle and puck
        paddle_distance_to_attack_line_y = abs(attack_line_y - paddle_pos.y);
        puck_distance_to_attack_line_y = abs(attack_line_y - puck_pos.y);
        debug_text = "puck d: " + puck_distance_to_attack_line_y + "\npaddle d: " +
paddle_distance_to_attack_line_y;
        text(debug_text, 10, 30);

        // Calculate time to travel to attack line for paddle and puck
        while(paddle_distance_to_attack_line_y > 0) {
          paddle_time_to_attack_line_y ++;
          if (paddle_vel_y_calc < 4) {
            paddle_vel_y_calc ++;
          }
          paddle_distance_to_attack_line_y -= paddle_vel_y_calc;
        }
        while(puck_distance_to_attack_line_y > 0) {
          puck_time_to_attack_line_y ++;
          puck_distance_to_attack_line_y -= abs(puck_vel.y);
        }
        debug_text = "puck t: " + puck_time_to_attack_line_y + "\npaddle t: " +
paddle_time_to_attack_line_y;
        text(debug_text, 10, 60);

        // Start moving paddle when timing for collision is good
        if (abs(puck_time_to_attack_line_y - paddle_time_to_attack_line_y) <= 0.1) {
          pos_command.x = puck_pos.x + random(-30, 30);
          pos_command.y = attack_line_y;
          attack_move_active = true;
        }
      }

      if ((puck_vel.y > 0) || (puck_pos.y < attack_line_y)) {
        attack_move_active = false;
        state = state_Home;
        return;
      }
    }

    // Logic for 'defend' state
    if (state == state_Defend) {
      text("state: defend", 10, 10);
      pos_command.x = puck_pos.x + random(-30, 30);
      pos_command.y = paddle_diameter/2;

      if (puck_vel.y > 0) {
        state = state_Home;
        return;
      }
    }

    // Logic for 'return' state
    if (state == state_Return) {
      text("state: return", 10, 10);
      pos_command.x = puck_pos.x + random(-30, 30);
      pos_command.y = puck_pos.y - 10;

      if (puck_pos.y > height/2) {
        state = state_Home;
        return;
      }
    }
    }
}
```

neuron.pde (Data structure and functions for simulated neurons)

```
class Neuron {
  float upper_limit;
  float lower_limit;
  float[] weights;
  float[] input_data;
  float output;
  int num_inputs;

  Neuron(int inputs) {
    upper_limit = 1000;
    lower_limit = -1000;
    num_inputs = inputs;
    weights = new float[num_inputs];
    input_data = new float[num_inputs];
    output = 0;

    // Randomize all weights
    for (int i = 0; i < num_inputs; i++) {
      weights[i] = random(-10, 10);
    }
  }

  //---------------------------------------------------------------------------------------------------------------
  // Calculates the output of the neuron
  void update() {
    output = 0;
    for (int i = 0; i < num_inputs; i++) {
      output += (input_data[i] * weights[i]);
    }
    output = output * upper_limit / sqrt(1 + pow(output, 2));  // Apply sigmoid function x /
sqrt(1 + x^2)

    if (output > upper_limit) {
      output = upper_limit;
    } else if (output < lower_limit) {
      output = lower_limit;
    }
  }

  //-----------------------------------------------------------------------------------------------------------
  // Returns an exact duplicates of the brain
  Neuron copy() {
    Neuron foo = new Neuron(num_inputs);
    foo.upper_limit = upper_limit;
    foo.lower_limit = lower_limit;
    foo.weights = weights;
    foo.num_inputs = num_inputs;
    return foo;
  }
}
```

paddle.pde (Data structure and functions for simulated air hockey paddles)

```
int paddle_diameter = 100;

class Paddle {
  PVector pos;
  PVector vel;
  PVector acc;
  boolean robot;  // If true this paddle is the robot side
```

```
Paddle(boolean robot_option) {
  pos = new PVector(width/2, height/2);
  vel = new PVector(0, 0);
  acc = new PVector(0, 0);

  robot = robot_option;
  if (robot) {
    pos.y = height - (paddle_diameter * 2);
  } else {
    pos.y = paddle_diameter * 2;
  }
}

//-----------------------------------------------------------------------------------------------------------
// Draws the paddle on the screen
void show() {
  if (robot) {
    fill(255, 0, 0);
  } else {
    fill(0, 0, 255);
  }
  ellipse(pos.x, pos.y, paddle_diameter, paddle_diameter);
}

//-----------------------------------------------------------------------------------------------------------
// Moves the paddle
void move() {
  if (robot) {
    acc.limit(0.5);
  } else {
    acc.limit(0.8);  // Artificially make the human faster than the robot
  }
  vel.add(acc);
  vel.limit(2);  // limit to 4m/s
  pos.add(vel);
}

//-----------------------------------------------------------------------------------------------------------
// Calls the move function and check for collisions and stuff
void update(PVector pos_command) {
  // Handle position commands
  if (abs(pos_command.x - pos.x) <= 4) {
    vel.x = 0;
  }
  if (abs(pos_command.y - pos.y) <= 4) {
    vel.y = 0;
  }

  if (pos_command.x > pos.x) {
    acc.x = 1;
  } else if (pos_command.x < pos.x) {
    acc.x = -1;
  } else {
    acc.x = 0;
    vel.x = 0;
  }
  if (pos_command.y > pos.y) {
    acc.y = 1;
  } else if (pos_command.y < pos.y) {
    acc.y = -1;
  } else {
    acc.y = 0;
    vel.y = 0;
  }
```

```
    // Move paddle to new position
    move();

    // Limit to edges of table
    if (pos.x < (paddle_diameter/2 + puck_diameter)) {
      pos.x = paddle_diameter/2 + puck_diameter;
    }
    if (pos.x > width - (paddle_diameter/2) - puck_diameter) {
      pos.x = width - (paddle_diameter/2) - puck_diameter;
    }
    if (robot) {
      // Robot is on bottom half of table
      if (pos.y > height - (paddle_diameter/2)) {
        pos.y = height - (paddle_diameter/2);
      }
      if (pos.y < (height/2) + (paddle_diameter/2)) {
        pos.y = (height/2) + (paddle_diameter/2);
      }
    } else {
      // Human is on top half
      if (pos.y < (paddle_diameter/2)) {
        pos.y = paddle_diameter/2;
      }
      if (pos.y > (height/2) - (paddle_diameter/2)) {
        pos.y = (height/2) - (paddle_diameter/2);
      }
    }
  }
}
```

puck.pde (Data structure and functions for simulated puck)

```
int puck_diameter = 80;

class Puck {
  PVector pos;
  PVector vel;
  PVector acc;
  boolean goal_robot;  // True when the robot scores a goal
  boolean goal_human;  // True when the human scores a goal

  Puck() {
    // Start the puck at centre ice
    pos = new PVector(width/2, height/2);
    vel = new PVector(0, 0);
    acc = new PVector(0, 0);
  }

  //-----------------------------------------------------------------------------------------------------------
  // Draws the puck on the screen
  void show() {
    fill(0, 255, 0);
    ellipse(pos.x, pos.y, puck_diameter, puck_diameter);
  }

  //-----------------------------------------------------------------------------------------------------------
  // Moves the puck
  void move() {
    // Apply the acceleration and move the paddle
    vel.add(acc);
    vel.limit(10);  // Limit to 4m/s
    pos.add(vel);
```

```
 }

 //-------------------------------------------------------------------------------------------------------------
 // Calls the move function and check for collisions and stuff
 void update(Brain brain) {
   float distance_to_paddle = 0;
   PVector new_direction = new PVector(0, 0);
   float vel_magnitude = 0;

   // Check for collision with human paddle
   distance_to_paddle = sqrt(pow(pos.x - human_paddle.pos.x, 2) + pow(pos.y -
human_paddle.pos.y, 2));
   if (distance_to_paddle <= ((puck_diameter/2) + (paddle_diameter/2))) {
     // determine new direction vector based on line drawn between centre of paddle and
puck
     new_direction.x = pos.x - human_paddle.pos.x;
     new_direction.y = pos.y - human_paddle.pos.y;
     new_direction.normalize();

     // determine magnitude of new velocity
     vel_magnitude = max(sqrt(pow(human_paddle.vel.x, 2) + pow(human_paddle.vel.y, 2)),
sqrt(pow(vel.x, 2) + pow(vel.y, 2)));

     // scale new direction to create new puck velocity
     vel.x = new_direction.x * vel_magnitude;
     vel.y = new_direction.y * vel_magnitude;
   }

   // Check for collision with robot paddle
   distance_to_paddle = sqrt( pow(pos.x - robot_paddle.pos.x, 2) + pow(pos.y -
robot_paddle.pos.y, 2));
   if (distance_to_paddle <= ((puck_diameter/2) + (paddle_diameter/2))) {
     // determine new direction vector based on line drawn between centre of paddle and
puck
     new_direction.x = pos.x - robot_paddle.pos.x;
     new_direction.y = pos.y - robot_paddle.pos.y;
     new_direction.normalize();

     // determine magnitude of new velocity
     vel_magnitude = max(sqrt(pow(robot_paddle.vel.x, 2) + pow(robot_paddle.vel.y, 2)),
sqrt(pow(vel.x, 2) + pow(vel.y, 2)));

     // scale new direction to create new puck velocity
     vel.x = new_direction.x * vel_magnitude;
     vel.y = new_direction.y * vel_magnitude;
     brain.redirections ++;
   }

   // Move puck to new position and handle bounces
   move();
   if (pos.x < (puck_diameter/2)) {
     pos.x = puck_diameter/2;
     vel.x = -vel.x;
   }
   if (pos.x > width - (puck_diameter/2)) {
     pos.x = width - (puck_diameter/2);
     vel.x = -vel.x;
   }
   if (pos.y < (puck_diameter/2)) {
     pos.y = (puck_diameter/2);
     vel.y = -vel.y;
   }
   if (pos.y > height - (puck_diameter/2)) {
     pos.y = height - (puck_diameter/2);
     vel.y = -vel.y;
   }
```

```
  // Check for goals scored
  if ((((width/2) - (goal_width_x/2)) < pos.x) && (pos.x < ((width/2) - (goal_width_x/2) +
goal_width_x))) {
    // X position is close to goal
    if (pos.y == (height - (puck_diameter/2))) {
      goal_human = true;
    } else if (pos.y == (goal_height_y - (puck_diameter/2))) {
      goal_robot = true;
    } else {
      goal_human = false;
      goal_robot = false;
    }
  }
 }
}
```

References

[1] B.J. Copeland, "Artificial Intelligence" Encyclopedia Britannica, [Online access on May 28, 2018]. URL: https://www.britannica.com/technology/artificial-intelligence

[2] Parthasarathy, Dhruv. “Write an AI to win at Pong from scratch with Reinforcement Learning.” Medium, [Online access on May 29, 2018]. URL: https://medium.com/@dhruvp/how-to-write-a-neural-network-to-play-pong-from-scratch-956b57d4f6e0

[3] OpenAI Gym, [Online access on May 29, 2018]. URL: https://gym.openai.com/

[4] Nigretti, Alessia. “Using Machine Learning Agents in a real game: a beginner’s guide” Unity Blog, [Online access on May 29, 2018]. URL: https://blogs.unity3d.com/2017/12/11/using-machine-learning-agents-in-a-real-game-a-beginners-guide/

[5] Valkov, Venelin, "Creating a Neural Network from Scratch" Medium. [Online access on July 10, 2018]. URL: https://tinyurl.com/yc6ed6kw

[6] Jiaconda, "A Concise History of Neural Networks" Medium. [Online access on July 2018] URL: https://medium.com/@Jaconda/a-concise-history-of-neural-networks-2070655d3fec

[7] Kriesel, David, "A Brief Introduction to Neural Networks" [Online access on July 2018] URL: http://www.dkriesel.com/_media/science/neuronalenetze-en-zeta2-2col-dkrieselcom.pdf

[8] Wikipedia, "Artificial Neural Networks/History" [Online access on July 2018] URL: https://en.wikibooks.org/wiki/Artificial_Neural_Networks/History

[9] Reingol, D., "Artificial Neural Networks Technology" [Online access on July 2018] URL: http://www.psych.utoronto.ca/users/reingold/courses/ai/cache/neural2.html

[10] Logan, Savannah, "Understanding the Structure of Neural Networks" Medium. [Online access on July 2018] URL: https://becominghuman.ai/understanding-the-structure-of-neural-networks-1fa5bd17fef0

[11] Sanjeevi, Madhu, "Artificial Neural Networks with Math" Medium. [Online access on July 2018] URL: https://medium.com/deep-math-machine-learning-ai/chapter-7-artificial-neural-networks-with-math-bb711169481b

[12] Sanjeevi, Madhu, "Gradient Descent with Math" Medium. [Online access on July 2018] URL: https://medium.com/deep-math-machine-learning-ai/chapter-1-2-gradient-descent-with-math-d4f2871af402

[13] Wikipedia, “Artificial intelligence in video games“. [Online access on July 24, 2018]. URL: https://en.wikipedia.org/wiki/Artificial_intelligence_in_video_games

[14] Youtube, “Code Bullet”. [Online access on July 24, 2018]. URL: https://www.youtube.com/channel/UC0e3QhIYukixgh5VVpKHH9Q/featured

[15] Wikipedia, “AlphaGo”. [Online access on July 24, 2018]. URL: https://en.wikipedia.org/wiki/AlphaGo

[16] Wikipedia, “Monte Carlo tree search”. [Online access on July 24, 2018]. URL: https://en.wikipedia.org/wiki/Monte_Carlo_tree_search

[17] Wikipedia, “Reinforcement learning”. [Online access on July 24, 2018]. URL: https://en.wikipedia.org/wiki/Reinforcement_learning

[18] Wikipedia, “Genetic algorithm”. [Online access on July 24, 2018]. URL: https://en.wikipedia.org/wiki/Genetic_algorithm