Weights: 0.17364325887120052 , 0.4367939745313998

Weights: 0.13431871877094592 , 0.25313809845637203

Weights: 0.16590967634090653 , 0.14983448838095526

Weights: 0.1625598452571151 , 0.5042918591736545

Weights: 0.17928080892932455 , 0.3715588988670978

Weights: 0.1772128150050226 , 0.3715016822913039

NEW:

Weights: 0.21652219987655558 , 0.24580249537466092

if (ke.getKeyCode() == KeyEvent.VK\_LEFT) {

int counter = 0;

while (!board.getSelectedDino().getDino().isVisible()) {

board.setSelectedDino(dinos.getPopulation().get(counter));

counter++;

}

}

Useful links:

<https://en.wikipedia.org/wiki/Artificial_neural_network>

<https://medium.com/technologymadeeasy/for-dummies-the-introduction-to-neural-networks-we-all-need-c50f6012d5eb>

Terms:

Perceptron

Heaviside step function

Back propagation

<https://machinelearningmastery.com/implement-backpropagation-algorithm-scratch-python/>

<https://www.youtube.com/watch?v=aeWmdojEJf0&feature=youtu.be>

<https://www.youtube.com/watch?v=9zfeTw-uFCw&feature=youtu.be>

<http://zetcode.com/tutorials/javagamestutorial/movingsprites/>

Java Neural Nets

<https://www.youtube.com/watch?v=ZJNklhq1zvg>

<https://github.com/Flood2d/ocr-neuralnet/blob/master/java/src/net/flood/ocrnn/CharacterTester.java>

Really simple game -> Dinosaur jumping over an obstacle

2 input nodes: distance to closest wall and distance to closest bird

Fitness = (positive) total distance - (negative) distance to next obstacle

Dinosaur/Character sprite. Obstacle sprite. Background

Randomly generate obstacles

Frame by frame moving the obstacles

Gravity for the dinosaur

Make sure dinosaur can’t jump while in the air

Dinosaur has to die when it hits the obstacle

Delay between consecutive jumps

Extra things:

* More obstacles
* Difficulties
* More available inputs

Dinosaur Class extends Sprite

* Important stuff to keep in mind
  + Xcoord probably doesn’t change
* Private data
  + Private final int ground;
  + Private int jumpDelay = 0; (milliseconds between finish jumping and next jump)
  + Private int jumpVelocity = 0;
  + Private int duckHeight = 0;
  + Private boolean isDucking = false;
* Dinosaur(int x, int y, int w, int h, int jumpDelay, int duckHeight)
* All appropriate setters and getters
* Void jump()
  + Precondition: Y = ground and jumpDelay = 0
  + Changes jumpVelocity to 20
* Void startDuck() - On key press
  + Precondition: isDucking is false
  + Set height to height - duckHeight
  + Set yCoord to height + duckHeight
  + Set width to width + duckHeight
  + Set isDucking to true
* Void endDuck() - On key lift
  + Precondition: isDucking is true
  + Set height to height + duckHeight
  + Set yCoord to height + duckHeight
  + Set width to width - duckHeight
  + Set isDucking to false
* Boolean isTouchingGround

Wall Class extends Sprite

* X and Y values (y doesn’t change)
* Wall(int x, int y, int w, int h)

Bird Class extends Sprite

* Private data int velocity;
* Bird(int x, int y, int w, int h, int velocity)
* X and Y values (y doesn’t change)

Game class extends JFrame implements ActionListener

* Creates Dinosaur object, ArrayList of walls, ArrayList of birds
* Timer (every X milliseconds)
  + Decreases the x values of the walls and birds (If not isInBounds(screen), delete it from the list)
  + Decreases the y value of the dinosaur if it’s not touching the ground. Make sure it doesn’t go past the ground.
  + If dinosaur’s jumpVelocity > 0, add the jumpVelocity to the y value of the dinosaur, and decrease velocity
  + For every wall and every bird, calls the isInBounds() method for the dinosaur and passes x,y pair and width and height for the parameters. If true, stop timer and call resetGame()
* loadObstacles() - randomly creates bird and wall objects and adds them to the Arraylist
  + Generate a random number between 2000 and 5000. This will be the time until the next obstacle.
  + Generate a random number between 0 and 1, if >50 use bird. If <= 50 use wall
  + Add obstacle to appropriate list and initialize so it is in bounds of the screen
* Start Button - starts timer
* Pause Button
* Settings/other controls
* resetGame()

Board Class extends JPanel

* Board(Dinosaur, ArrayList of walls, ArrayList of birds)
* paintComponent
  + Draw the dinosaur if visible and isInBounds(screen)
  + Draw each wall if visible and isInBounds(screen)
  + Draw each bird if visible and isInBounds(screen)

Abstract Sprite class:

* private data
  + private boolean visibility;
  + private int xCoord;
  + private int yCoord;
  + private int width;
  + private int height;
* Public Sprite()
  + Constructor that sets everything to default
* Public Sprite(int x, int y, int w, int h)
* Public String toString()
  + Returns a string representation of the Sprite: "location: (" + xCoord + ", " + yCoord + "); visibility: " + visibility;
* Getters and Setters for xCoord, yCoord, height, width
* Public boolean isVisible()
  + Returns the visibility
* Public void setVisibility
* isInBounds(x,y,width,height)
  + Returns true if it is IN the rectangle

Genetic Algorithm Part

2 input nodes: distance to closest wall and distance to closest bird

Fitness = (positive) total distance

Class DinoGene

* Double wallWeight
* Double birdWeight
* Dinosaur dino
* Double evaluation
* Int fitness
* DinoGene()
  + Sets wallWeight and birdWeight to random stuff
* public DinoGene() {
* wallWeight = Math.random();
* birdWeight = Math.random();
* dino = new Dinosaur();
* }
* public DinoGene(Dinosaur d) {
* wallWeight = Math.random();
* birdWeight = Math.random();
* dino =d;
* }
* public DinoGene(double wallWeight, double birdWeight, Dinosaur d) {
* this.wallWeight = wallWeight;
* this.birdWeight = birdWeight;
* dino =d;
* }
* public int calculateFitness(int distTraveled) {
* return distTraveled;
* }
* public double evaluate(int distToWall, int distToBird) {
* return (distToWall \* wallWeight + distToBird \* birdWeight)/(distToBird + distToWall);
* }

Class Population

* ArrayList<DinoGene> population
* Private double mutationFreq
  + the frequency at which mutations occur
* Private int maxPop, the maximum population
* Population(double mutationFreq, int maxPop)
* Void CalcFitness()
* Void evaluate()
* void naturalSelection()
* void breed()

**Example javascript implementation of flappy bird genetic algorithm:**

/\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*  
/\* Genetic Algorithm implementation  
/\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*/  
  
var GeneticAlgorithm = function(max\_units, top\_units){  
 this.max\_units = max\_units; // max number of units in population  
 this.top\_units = top\_units; // number of top units (winners) used for evolving population  
   
 if (this.max\_units < this.top\_units) this.top\_units = this.max\_units;  
   
 this.Population = []; // array of all units in current population  
   
 this.SCALE\_FACTOR = 200; // the factor used to scale normalized input values  
}  
  
GeneticAlgorithm.prototype = {  
 // resets genetic algorithm parameters  
 reset : function(){  
 this.iteration = 1; // current iteration number (it is equal to the current population number)  
 this.mutateRate = 1; // initial mutation rate  
   
 this.best\_population = 0; // the population number of the best unit  
 this.best\_fitness = 0; // the fitness of the best unit  
 this.best\_score = 0; // the score of the best unit ever  
 },  
   
 // creates a new population  
 createPopulation : function(){  
 // clear any existing population  
 this.Population.splice(0, this.Population.length);  
   
 for (var i=0; i<this.max\_units; i++){  
 // create a new unit by generating a random Synaptic neural network  
 // with 2 neurons in the input layer, 6 neurons in the hidden layer and 1 neuron in the output layer  
 var newUnit = new synaptic.Architect.Perceptron(2, 6, 1);  
   
 // set additional parameters for the new unit  
 newUnit.index = i;  
 newUnit.fitness = 0;  
 newUnit.score = 0;  
 newUnit.isWinner = false;  
   
 // add the new unit to the population   
 this.Population.push(newUnit);  
 }  
 },  
   
 // activates the neural network of an unit from the population   
 // to calculate an output action according to the inputs  
 activateBrain : function(bird, target){   
 // input 1: the horizontal distance between the bird and the target  
 var targetDeltaX = this.normalize(target.x, 700) \* this.SCALE\_FACTOR;  
   
 // input 2: the height difference between the bird and the target  
 var targetDeltaY = this.normalize(bird.y - target.y, 800) \* this.SCALE\_FACTOR;  
   
 // create an array of all inputs  
 var inputs = [targetDeltaX, targetDeltaY];  
   
 // calculate outputs by activating synaptic neural network of this bird  
 var outputs = this.Population[bird.index].activate(inputs);  
   
 // perform flap if output is greater than 0.5  
 if (outputs[0] > 0.5) bird.flap();  
 },  
   
 // evolves the population by performing selection, crossover and mutations on the units  
 evolvePopulation : function(){  
 // select the top units of the current population to get an array of winners  
 // (they will be copied to the next population)  
 var Winners = this.selection();  
  
 if (this.mutateRate == 1 && Winners[0].fitness < 0){   
 // If the best unit from the initial population has a negative fitness   
 // then it means there is no any bird which reached the first barrier!  
 // Playing as the God, we can destroy this bad population and try with another one.  
 this.createPopulation();  
 } else {  
 this.mutateRate = 0.2; // else set the mutatation rate to the real value  
 }  
   
 // fill the rest of the next population with new units using crossover and mutation  
 for (var i=this.top\_units; i<this.max\_units; i++){  
 var parentA, parentB, offspring;  
   
 if (i == this.top\_units){  
 // offspring is made by a crossover of two best winners  
 parentA = Winners[0].toJSON();  
 parentB = Winners[1].toJSON();  
 offspring = this.crossOver(parentA, parentB);  
  
 } else if (i < this.max\_units-2){  
 // offspring is made by a crossover of two random winners  
 parentA = this.getRandomUnit(Winners).toJSON();  
 parentB = this.getRandomUnit(Winners).toJSON();  
 offspring = this.crossOver(parentA, parentB);  
   
 } else {  
 // offspring is a random winner  
 offspring = this.getRandomUnit(Winners).toJSON();  
 }  
  
 // mutate the offspring  
 offspring = this.mutation(offspring);  
   
 // create a new unit using the neural network from the offspring  
 var newUnit = synaptic.Network.fromJSON(offspring);  
 newUnit.index = this.Population[i].index;  
 newUnit.fitness = 0;  
 newUnit.score = 0;  
 newUnit.isWinner = false;  
   
 // update population by changing the old unit with the new one  
 this.Population[i] = newUnit;  
 }  
   
 // if the top winner has the best fitness in the history, store its achievement!  
 if (Winners[0].fitness > this.best\_fitness){  
 this.best\_population = this.iteration;  
 this.best\_fitness = Winners[0].fitness;  
 this.best\_score = Winners[0].score;  
 }  
   
 // sort the units of the new population in ascending order by their index  
 this.Population.sort(function(unitA, unitB){  
 return unitA.index - unitB.index;  
 });  
 },  
  
 // selects the best units from the current population  
 selection : function(){  
 // sort the units of the current population in descending order by their fitness  
 var sortedPopulation = this.Population.sort(  
 function(unitA, unitB){  
 return unitB.fitness - unitA.fitness;  
 }  
 );  
   
 // mark the top units as the winners!  
 for (var i=0; i<this.top\_units; i++) this.Population[i].isWinner = true;  
   
 // return an array of the top units from the current population  
 return sortedPopulation.slice(0, this.top\_units);  
 },  
   
 // performs a single point crossover between two parents  
 crossOver : function(parentA, parentB) {  
 // get a cross over cutting point  
 var cutPoint = this.random(0, parentA.neurons.length-1);  
   
 // swap 'bias' information between both parents:  
 // 1. left side to the crossover point is copied from one parent  
 // 2. right side after the crossover point is copied from the second parent  
 for (var i = cutPoint; i < parentA.neurons.length; i++){  
 var biasFromParentA = parentA.neurons[i]['bias'];  
 parentA.neurons[i]['bias'] = parentB.neurons[i]['bias'];  
 parentB.neurons[i]['bias'] = biasFromParentA;  
 }  
  
 return this.random(0, 1) == 1 ? parentA : parentB;  
 },  
   
 // performs random mutations on the offspring  
 mutation : function (offspring){  
 // mutate some 'bias' information of the offspring neurons  
 for (var i = 0; i < offspring.neurons.length; i++){  
 offspring.neurons[i]['bias'] = this.mutate(offspring.neurons[i]['bias']);  
 }  
   
 // mutate some 'weights' information of the offspring connections  
 for (var i = 0; i < offspring.connections.length; i++){  
 offspring.connections[i]['weight'] = this.mutate(offspring.connections[i]['weight']);  
 }  
   
 return offspring;  
 },  
   
 // mutates a gene  
 mutate : function (gene){  
 if (Math.random() < this.mutateRate) {  
 var mutateFactor = 1 + ((Math.random() - 0.5) \* 3 + (Math.random() - 0.5));  
 gene \*= mutateFactor;  
 }  
   
 return gene;  
 },  
   
 random : function(min, max){  
 return Math.floor(Math.random()\*(max-min+1) + min);  
 },  
   
 getRandomUnit : function(array){  
 return array[this.random(0, array.length-1)];  
 },  
   
 normalize : function(value, max){  
 // clamp the value between its min/max limits  
 if (value < -max) value = -max;  
 else if (value > max) value = max;  
   
 // normalize the clamped value  
 return (value/max);  
 }  
}