```
//a connection between 2 nodes
class connectionGene {
  constructor(from, to, w, inno) {
    this.fromNode = from;
    this.toNode = to;
    this.weight = w;
    this.enabled = true;
    this.innovationNo = inno; //each connection is given a innovation number to compare genomes

  }

  //-----------------------------------------------------------------------------------------------------------------------------------------------------------------------------------
  //changes the this.weight
  mutateWeight() {
    var rand2 = random(1);
    if (rand2 < 0.1) { //10% of the time completely change the this.weight
      this.weight = random(-1, 1);
    } else { //otherwise slightly change it
      this.weight += (randomGaussian() / 50);
      //keep this.weight between bounds
      if (this.weight > 1) {
        this.weight = 1;
      }
      if (this.weight < -1) {
        this.weight = -1;

      }
    }
  }

  //---------------------------------------------------------------------------------------------------------
  //returns a copy of this connectionGene
  clone(from, to) {
    var clone = new connectionGene(from, to, this.weight, this.innovationNo);
    clone.enabled = this.enabled;

    return clone;
  }
}
```

```
class connectionHistory {
  constructor(from, to, inno, innovationNos) {
    this.fromNode = from;
    this.toNode = to;
    this.innovationNumber = inno;
    this.innovationNumbers = []; //the innovation Numbers from the connections of the genome which first had this mutation
    //this represents the genome and allows us to test if another genoeme is the same
    //this is before this connection was added
    arrayCopy(innovationNos, this.innovationNumbers); //copy (from, to)
  }

  //---------------------------------------------------------------------------------------------------------------
  //returns whether the genome matches the original genome and the connection is between the same nodes
  matches(genome, from, to) {
    if (genome.genes.length === this.innovationNumbers.length) { //if the number of connections are different then the genoemes aren't the same
      if (from.number === this.fromNode && to.number === this.toNode) {
        //next check if all the innovation numbers match from the genome
        for (var i = 0; i < genome.genes.length; i++) {
          if (!this.innovationNumbers.includes(genome.genes[i].innovationNo)) {
            return false;
          }
        }
        //if reached this far then the innovationNumbers match the genes innovation numbers and the connection is between the same nodes
        //so it does match
        return true;
      }
    }
    return false;
  }
}
```

```
class Genome {
  constructor(inputs, outputs, crossover) {
    this.genes = []; //a list of connections between this.nodes which represent the NN
    this.nodes = [];
    this.inputs = inputs;
    this.outputs = outputs;
    this.layers = 2;
    this.nextNode = 0;
    // this.biasNode;
    this.network = []; //a list of the this.nodes in the order that they need to be considered in the NN
    //create input this.nodes

    if (crossover) {
      return;
    }

    for (var i = 0; i < this.inputs; i++) {
      this.nodes.push(new Node(i));
      this.nextNode++;
      this.nodes[i].layer = 0;
    }

    //create output this.nodes
    for (var i = 0; i < this.outputs; i++) {
      this.nodes.push(new Node(i + this.inputs));
      this.nodes[i + this.inputs].layer = 1;
      this.nextNode++;
    }

    this.nodes.push(new Node(this.nextNode)); //bias node
    this.biasNode = this.nextNode;
    this.nextNode++;
    this.nodes[this.biasNode].layer = 0;



  }


  fullyConnect(innovationHistory) {

    //this will be a new number if no identical genome has mutated in the same

    for (var i = 0; i < this.inputs; i++) {
      for (var j = 0; j < this.outputs; j++) {
        var connectionInnovationNumber = this.getInnovationNumber(innovationHistory, this.nodes[i], this.nodes[this.nodes.length - j - 2]);
        this.genes.push(new connectionGene(this.nodes[i], this.nodes[this.nodes.length - j - 2], random(-1, 1), connectionInnovationNumber));
      }
    }

    var connectionInnovationNumber = this.getInnovationNumber(innovationHistory, this.nodes[this.biasNode], this.nodes[this.nodes.length - 2]);
    this.genes.push(new connectionGene(this.nodes[this.biasNode], this.nodes[this.nodes.length - 2], random(-1, 1), connectionInnovationNumber));

    connectionInnovationNumber = this.getInnovationNumber(innovationHistory, this.nodes[this.biasNode], this.nodes[this.nodes.length - 3]);
    this.genes.push(new connectionGene(this.nodes[this.biasNode], this.nodes[this.nodes.length - 3], random(-1, 1), connectionInnovationNumber));
    //add the connection with a random array


    //changed this so if error here
    this.connectNodes();
  }



  //-----------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------
  //returns the node with a matching number
  //sometimes the this.nodes will not be in order
  getNode(nodeNumber) {
    for (var i = 0; i < this.nodes.length; i++) {
      if (this.nodes[i].number == nodeNumber) {
        return this.nodes[i];
      }
    }
    return null;
  }


  //-----------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------
  //adds the conenctions going out of a node to that node so that it can acess the next node during feeding forward
  connectNodes() {

    for (var i = 0; i < this.nodes.length; i++) { //clear the connections
      this.nodes[i].outputConnections = [];
    }

    for (var i = 0; i < this.genes.length; i++) { //for each connectionGene
      this.genes[i].fromNode.outputConnections.push(this.genes[i]); //add it to node
    }
  }

  //-----------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------
  //feeding in input values varo the NN and returning output array
  feedForward(inputValues) {
    //set the outputs of the input this.nodes
    for (var i = 0; i < this.inputs; i++) {
      this.nodes[i].outputValue = inputValues[i];
    }
    this.nodes[this.biasNode].outputValue = 1; //output of bias is 1

    for (var i = 0; i < this.network.length; i++) { //for each node in the network engage it(see node class for what this does)
      this.network[i].engage();
    }

    //the outputs are this.nodes[inputs] to this.nodes [inputs+outputs-1]
    var outs = [];
    for (var i = 0; i < this.outputs; i++) {
      outs[i] = this.nodes[this.inputs + i].outputValue;
    }

    for (var i = 0; i < this.nodes.length; i++) { //reset all the this.nodes for the next feed forward
      this.nodes[i].inputSum = 0;
    }

    return outs;
  }

  //----------------------------------------------------------------------------------------------------------------------------
  //sets up the NN as a list of this.nodes in order to be engaged

  generateNetwork() {
      this.connectNodes();
      this.network = [];
      //for each layer add the node in that layer, since layers cannot connect to themselves there is no need to order the this.nodes within a layer

      for (var l = 0; l < this.layers; l++) { //for each layer
        for (var i = 0; i < this.nodes.length; i++) { //for each node
          if (this.nodes[i].layer == l) { //if that node is in that layer
            this.network.push(this.nodes[i]);
          }
        }
      }
    }
    //-----------------------------------------------------------------------------------------------------------------------------------------
    //mutate the NN by adding a new node
    //it does this by picking a random connection and disabling it then 2 new connections are added
    //1 between the input node of the disabled connection and the new node
    //and the other between the new node and the output of the disabled connection
  addNode(innovationHistory) {
    //pick a random connection to create a node between
    if (this.genes.length == 0) {
      this.addConnection(innovationHistory);
      return;
    }
    var randomConnection = floor(random(this.genes.length));

    while (this.genes[randomConnection].fromNode == this.nodes[this.biasNode] && this.genes.length != 1) { //dont disconnect bias
      randomConnection = floor(random(this.genes.length));
    }

    this.genes[randomConnection].enabled = false; //disable it

    var newNodeNo = this.nextNode;
    this.nodes.push(new Node(newNodeNo));
    this.nextNode++;
    //add a new connection to the new node with a weight of 1
    var connectionInnovationNumber = this.getInnovationNumber(innovationHistory, this.genes[randomConnection].fromNode, this.getNode(newNodeNo));
    this.genes.push(new connectionGene(this.genes[randomConnection].fromNode, this.getNode(newNodeNo), 1, connectionInnovationNumber));


    connectionInnovationNumber = this.getInnovationNumber(innovationHistory, this.getNode(newNodeNo), this.genes[randomConnection].toNode);
    //add a new connection from the new node with a weight the same as the disabled connection
    this.genes.push(new connectionGene(this.getNode(newNodeNo), this.genes[randomConnection].toNode, this.genes[randomConnection].weight, connectionInnovationNumber));
    this.getNode(newNodeNo).layer = this.genes[randomConnection].fromNode.layer + 1;


    connectionInnovationNumber = this.getInnovationNumber(innovationHistory, this.nodes[this.biasNode], this.getNode(newNodeNo));
    //connect the bias to the new node with a weight of 0
    this.genes.push(new connectionGene(this.nodes[this.biasNode], this.getNode(newNodeNo), 0, connectionInnovationNumber));

    //if the layer of the new node is equal to the layer of the output node of the old connection then a new layer needs to be created
    //more accurately the layer numbers of all layers equal to or greater than this new node need to be incrimented
    if (this.getNode(newNodeNo).layer == this.genes[randomConnection].toNode.layer) {
      for (var i = 0; i < this.nodes.length - 1; i++) { //dont include this newest node
        if (this.nodes[i].layer >= this.getNode(newNodeNo).layer) {
          this.nodes[i].layer++;
        }
      }
      this.layers++;
    }
    this.connectNodes();
  }

  //------------------------------------------------------------------------------------------------------------------
  //adds a connection between 2 this.nodes which aren't currently connected
  addConnection(innovationHistory) {
      //cannot add a connection to a fully connected network
      if (this.fullyConnected()) {
        console.log("connection failed");
        return;
      }


      //get random this.nodes
      var randomNode1 = floor(random(this.nodes.length));
      var randomNode2 = floor(random(this.nodes.length));
      while (this.randomConnectionNodesAreShit(randomNode1, randomNode2)) { //while the random this.nodes are no good
        //get new ones
        randomNode1 = floor(random(this.nodes.length));
        randomNode2 = floor(random(this.nodes.length));
      }
      var temp;
      if (this.nodes[randomNode1].layer > this.nodes[randomNode2].layer) { //if the first random node is after the second then switch
        temp = randomNode2;
        randomNode2 = randomNode1;
        randomNode1 = temp;
      }

      //get the innovation number of the connection
      //this will be a new number if no identical genome has mutated in the same way
      var connectionInnovationNumber = this.getInnovationNumber(innovationHistory, this.nodes[randomNode1], this.nodes[randomNode2]);
      //add the connection with a random array

      this.genes.push(new connectionGene(this.nodes[randomNode1], this.nodes[randomNode2], random(-1, 1), connectionInnovationNumber)); //changed this so if error here
      this.connectNodes();
    }
    //-------------------------------------------------------------------------------------------------------------------------------------------
  randomConnectionNodesAreShit(r1, r2) {
    if (this.nodes[r1].layer == this.nodes[r2].layer) return true; // if the this.nodes are in the same layer
    if (this.nodes[r1].isConnectedTo(this.nodes[r2])) return true; //if the this.nodes are already connected



    return false;
  }

  //-------------------------------------------------------------------------------------------------------------------------------------------
  //returns the innovation number for the new mutation
  //if this mutation has never been seen before then it will be given a new unique innovation number
  //if this mutation matches a previous mutation then it will be given the same innovation number as the previous one
  getInnovationNumber(innovationHistory, from, to) {
      var isNew = true;
      var connectionInnovationNumber = nextConnectionNo;
      for (var i = 0; i < innovationHistory.length; i++) { //for each previous mutation
        if (innovationHistory[i].matches(this, from, to)) { //if match found
          isNew = false; //its not a new mutation
          connectionInnovationNumber = innovationHistory[i].innovationNumber; //set the innovation number as the innovation number of the match
          break;
        }
      }
      if (isNew) { //if the mutation is new then create an arrayList of varegers representing the current state of the genome
        var innoNumbers = [];
        for (var i = 0; i < this.genes.length; i++) { //set the innovation numbers
          innoNumbers.push(this.genes[i].innovationNo);
        }

        //then add this mutation to the innovationHistory
        innovationHistory.push(new connectionHistory(from.number, to.number, connectionInnovationNumber, innoNumbers));
        nextConnectionNo++;
      }
      return connectionInnovationNumber;
    }
    //----------------------------------------------------------------------------------------------------------------------------------------

  //returns whether the network is fully connected or not
  fullyConnected() {

    var maxConnections = 0;
    var nodesInLayers = []; //array which stored the amount of this.nodes in each layer
    for (var i = 0; i < this.layers; i++) {
      nodesInLayers[i] = 0;
    }
    //populate array
    for (var i = 0; i < this.nodes.length; i++) {
      nodesInLayers[this.nodes[i].layer] += 1;
    }
    //for each layer the maximum amount of connections is the number in this layer * the number of this.nodes infront of it
    //so lets add the max for each layer together and then we will get the maximum amount of connections in the network
    for (var i = 0; i < this.layers - 1; i++) {
      var nodesInFront = 0;
      for (var j = i + 1; j < this.layers; j++) { //for each layer infront of this layer
        nodesInFront += nodesInLayers[j]; //add up this.nodes
      }

      maxConnections += nodesInLayers[i] * nodesInFront;
    }

    if (maxConnections <= this.genes.length) { //if the number of connections is equal to the max number of connections possible then it is full
      return true;
    }

    return false;
  }


  //-------------------------------------------------------------------------------------------------------------------------------
  //mutates the genome
  mutate(innovationHistory) {
    if (this.genes.length == 0) {
      this.addConnection(innovationHistory);
    }


    var rand1 = random(1);
    if (rand1 < 0.8) { // 80% of the time mutate weights

      for (var i = 0; i < this.genes.length; i++) {
        this.genes[i].mutateWeight();
      }
    }

    //5% of the time add a new connection
    var rand2 = random(1);
    if (rand2 < 0.05) {

      this.addConnection(innovationHistory);
    }

    //1% of the time add a node
    var rand3 = random(1);
    if (rand3 < 0.01) {

      this.addNode(innovationHistory);
    }
  }

  //---------------------------------------------------------------------------------------------------------------------------------
  //called when this Genome is better that the other parent
  crossover(parent2) {
    var child = new Genome(this.inputs, this.outputs, true);
    child.genes = [];
    child.nodes = [];
    child.layers = this.layers;
    child.nextNode = this.nextNode;
    child.biasNode = this.biasNode;
    var childGenes = []; // new ArrayList<connectionGene>();//list of genes to be inherrited form the parents
    var isEnabled = []; // new ArrayList<Boolean>();
    //all inherited genes
    for (var i = 0; i < this.genes.length; i++) {
      var setEnabled = true; //is this node in the chlid going to be enabled

      var parent2gene = this.matchingGene(parent2, this.genes[i].innovationNo);
      if (parent2gene != -1) { //if the genes match
        if (!this.genes[i].enabled || !parent2.genes[parent2gene].enabled) { //if either of the matching genes are disabled

          if (random(1) < 0.75) { //75% of the time disabel the childs gene
            setEnabled = false;
          }
        }
        var rand = random(1);
        if (rand < 0.5) {
          childGenes.push(this.genes[i]);

          //get gene from this fucker
        } else {
          //get gene from parent2
          childGenes.push(parent2.genes[parent2gene]);
        }
      } else { //disjoint or excess gene
        childGenes.push(this.genes[i]);
        setEnabled = this.genes[i].enabled;
      }
      isEnabled.push(setEnabled);
    }


    //since all excess and disjovar genes are inherrited from the more fit parent (this Genome) the childs structure is no different from this parent | with exception of dormant connec
    //so all the this.nodes can be inherrited from this parent
    for (var i = 0; i < this.nodes.length; i++) {
      child.nodes.push(this.nodes[i].clone());
    }

    //clone all the connections so that they connect the childs new this.nodes

    for (var i = 0; i < childGenes.length; i++) {
      child.genes.push(childGenes[i].clone(child.getNode(childGenes[i].fromNode.number), child.getNode(childGenes[i].toNode.number)));
      child.genes[i].enabled = isEnabled[i];
    }

    child.connectNodes();
    return child;
  }

  //-----------------------------------------------------------------------------------------------------------------------------------------
  //returns whether or not there is a gene matching the input innovation number  in the input genome
  matchingGene(parent2, innovationNumber) {
      for (var i = 0; i < parent2.genes.length; i++) {
        if (parent2.genes[i].innovationNo == innovationNumber) {
          return i;
        }
      }
      return -1; //no matching gene found
    }
    //-----------------------------------------------------------------------------------------------------------------------------------------
    //prints out info about the genome to the console
  printGenome() {
    console.log("Prvar genome  layers:" + this.layers);
    console.log("bias node: " + this.biasNode);
    console.log("this.nodes");
    for (var i = 0; i < this.nodes.length; i++) {
      console.log(this.nodes[i].number + ",");
    }
    console.log("Genes");
    for (var i = 0; i < this.genes.length; i++) { //for each connectionGene
      console.log("gene " + this.genes[i].innovationNo + "From node " + this.genes[i].fromNode.number + "To node " + this.genes[i].toNode.number +
        "is enabled " + this.genes[i].enabled + "from layer " + this.genes[i].fromNode.layer + "to layer " + this.genes[i].toNode.layer + "weight: " + this.genes[i].weight);
    }

    console.log();
  }

  //-----------------------------------------------------------------------------------------------------------------------------------------
  //returns a copy of this genome
  clone() {

      var clone = new Genome(this.inputs, this.outputs, true);

      for (var i = 0; i < this.nodes.length; i++) { //copy this.nodes
        clone.nodes.push(this.nodes[i].clone());
      }

      //copy all the connections so that they connect the clone new this.nodes

      for (var i = 0; i < this.genes.length; i++) { //copy genes
        clone.genes.push(this.genes[i].clone(clone.getNode(this.genes[i].fromNode.number), clone.getNode(this.genes[i].toNode.number)));
      }

      clone.layers = this.layers;
      clone.nextNode = this.nextNode;
      clone.biasNode = this.biasNode;
      clone.connectNodes();

      return clone;
    }
    //-----------------------------------------------------------------------------------------------------------------------------------------
    //draw the genome on the screen
  drawGenome(startX, startY, w, h) {
    //i know its ugly but it works (and is not that important) so I'm not going to mess with it
    var allNodes = []; //new ArrayList<ArrayList<Node>>();
    var nodePoses = []; // new ArrayList<PVector>();
    var nodeNumbers = []; // new ArrayList<Integer>();

    //get the positions on the screen that each node is supposed to be in


    //split the this.nodes varo layers
    for (var i = 0; i < this.layers; i++) {
      var temp = []; // new ArrayList<Node>();
      for (var j = 0; j < this.nodes.length; j++) { //for each node
        if (this.nodes[j].layer == i) { //check if it is in this layer
          temp.push(this.nodes[j]); //add it to this layer
        }
      }
      allNodes.push(temp); //add this layer to all this.nodes
    }

    //for each layer add the position of the node on the screen to the node posses arraylist
    for (var i = 0; i < this.layers; i++) {
      fill(255, 0, 0);
      var x = startX + float((i + 1.0) * w) / float(this.layers + 1.0);
      for (var j = 0; j < allNodes[i].length; j++) { //for the position in the layer
        var y = startY + float((j + 1.0) * h) / float(allNodes[i].length + 1.0);
        nodePoses.push(createVector(x, y));
        nodeNumbers.push(allNodes[i][j].number);
      }
    }

    //draw connections
    stroke(0);
    strokeWeight(2);
    for (var i = 0; i < this.genes.length; i++) {
      if (this.genes[i].enabled) {
        stroke(0);
      } else {
        stroke(100);
      }
      var from;
      var to;
      from = nodePoses[nodeNumbers.indexOf(this.genes[i].fromNode.number)];
      to = nodePoses[nodeNumbers.indexOf(this.genes[i].toNode.number)];
      if (this.genes[i].weight > 0) {
        stroke(255, 0, 0);
      } else {
        stroke(0, 0, 255);
      }
      strokeWeight(map(abs(this.genes[i].weight), 0, 1, 0, 3));
      line(from.x, from.y, to.x, to.y);
    }

    //draw this.nodes last so they appear ontop of the connection lines
    for (var i = 0; i < nodePoses.length; i++) {
      fill(255);
      stroke(0);
      strokeWeight(1);
      ellipse(nodePoses[i].x, nodePoses[i].y, 20, 20);
      textSize(10);
      fill(0);
      textAlign(CENTER, CENTER);
      text(nodeNumbers[i], nodePoses[i].x, nodePoses[i].y);

    }

    // print out neural network info text
    // textAlign(RIGHT);
    // fill(255);
    // textSize(15);
    // noStroke();
    // text("car angle", nodePoses[0].x - 20, nodePoses[0].y);
    // text("touching ground", nodePoses[1].x - 20, nodePoses[1].y);
    // text("angular velocity", nodePoses[2].x - 20, nodePoses[2].y);
    // text("Distance to ground", nodePoses[3].x - 20, nodePoses[3].y);
    // text("gradient", nodePoses[4].x - 20, nodePoses[4].y);
    // text("bias", nodePoses[5].x - 20, nodePoses[5].y);
    // textAlign(LEFT);
    // text("gas", nodePoses[nodePoses.length - 2].x + 20, nodePoses[nodePoses.length - 2].y);
    // text("break", nodePoses[nodePoses.length - 1].x + 20, nodePoses[nodePoses.length - 1].y);



  }
}
```

```
<html>
<head>
  <meta charset="UTF-8">
  <script language="javascript" type="text/javascript" src="libraries/p5.js"></script>
  <script language="javascript" src="libraries/p5.dom.js"></script>
  <script language="javascript" src="libraries/p5.sound.js"></script>
  <script language="javascript" src="libraries/Box2d.js"></script>

  <script language="javascript" type="text/javascript" src="sketch.js"></script>
  <script language="javascript" type="text/javascript" src="ConnectionGene.js"></script>
  <script language="javascript" type="text/javascript" src="ConnectionHistory.js"></script>
  <script language="javascript" type="text/javascript" src="Node.js"></script>
  <script language="javascript" type="text/javascript" src="Player.js"></script>
  <script language="javascript" type="text/javascript" src="Population.js"></script>
  <script language="javascript" type="text/javascript" src="Species.js"></script>
  <script language="javascript" type="text/javascript" src="Genome.js"></script>

</head>
<body>

    <div id = "main">
      <h2> Neat template </h2>
      <div id = "canvas">
      </div>
  </div>


  <script>
    //center the canvas
    var tempString = ""+((window.innerWidth*0.9- 1026 - window.innerWidth*0.9*0.08 )/2)+ "px";
    document.getElementById("main").style.marginLeft = tempString;


    window.onresize = function(event) {
      var tempString = ""+((window.innerWidth*0.9- 1026 - window.innerWidth*0.9*0.08 )/2)+ "px";
      document.getElementById("main").style.marginLeft = tempString;
    };

  </script>

</body>
</html>
```

```
class Node {

  constructor(no) {
    this.number = no;
    this.inputSum = 0; //current sum i.e. before activation
    this.outputValue = 0; //after activation function is applied
    this.outputConnections = []; //new ArrayList<connectionGene>();
    this.layer = 0;
    this.drawPos = createVector();
  }

  //------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------
  //the node sends its output to the inputs of the nodes its connected to
  engage() {
      if(this.layer != 0) { //no sigmoid for the inputs and bias
        this.outputValue = this.sigmoid(this.inputSum);
      }

      for(var i = 0; i < this.outputConnections.length; i++) { //for each connection
        if(this.outputConnections[i].enabled) { //dont do shit if not enabled
          this.outputConnections[i].toNode.inputSum += this.outputConnections[i].weight * this.outputValue; //add the weighted output to the sum of the inputs of whatever node this nc
        }
      }
    }
    //-----------------------------------------------------------------------------------------------------------------------------------------
    //not used
   stepFunction(x) {
      if(x < 0) {
        return 0;
      } else {
        return 1;
      }
    }
    //----------------------------------------------------------------------------------------------------------------------------------------------------------------------------------
    //sigmoid activation function
  sigmoid(x) {
      return 1.0 / (1.0 + pow(Math.E, -4.9 * x)); //todo check pow
    }
    //-----------------------------------------------------------------------------------------------------------------------------------------------------------
    //returns whether this node connected to the parameter node
    //used when adding a new connection
  isConnectedTo(node) {
      if(node.layer == this.layer) { //nodes in the same this.layer cannot be connected
        return false;
      }

      //you get it
      if(node.layer < this.layer) {
        for(var i = 0; i < node.outputConnections.length; i++) {
          if(node.outputConnections[i].toNode == this) {
            return true;
          }
        }
      } else {
        for(var i = 0; i < this.outputConnections.length; i++) {
          if(this.outputConnections[i].toNode == node) {
            return true;
          }
        }
      }

      return false;
    }
    //----------------------------------------------------------------------------------------------------------------------------------------------------------------------------------
    //returns a copy of this node
  clone() {
    var clone = new Node(this.number);
    clone.layer = this.layer;
    return clone;
  }
}
```

```
class Player {

  constructor() {

    this.fitness = 0;
    this.vision = []; //the input array fed into the neuralNet
    this.decision = []; //the out put of the NN
    this.unadjustedFitness;
    this.lifespan = 0; //how long the player lived for this.fitness
    this.bestScore = 0; //stores the this.score achieved used for replay
    this.dead = false;
    this.score = 0;
    this.gen = 0;

    this.genomeInputs = 5;
    this.genomeOutputs = 2;
    this.brain = new Genome(this.genomeInputs, this.genomeOutputs);
  }

  //----------------------------------------------------------------------------------------------------------------------------------------------------------
  show() {
      //<<<<<<<<<<<<<<<<<<<<<<<<<<<<<<<<<<<replace
    }
    //----------------------------------------------------------------------------------------------------------------------------------------------------------
  move() {
      //<<<<<<<<<<<<<<<<<<<<<<<<<<<<<<<<<<<replace
    }
    //----------------------------------------------------------------------------------------------------------------------------------------------------------
  update() {
      //<<<<<<<<<<<<<<<<<<<<<<<<<<<<<<<<<<<replace
    }
    //-----------------------------------------------------------------------------------------------------------------------------------------------------------

  look() {
    //<<<<<<<<<<<<<<<<<<<<<<<<<<<<<<<<<<<replace

  }


  //----------------------------------------------------------------------------------------------------------------------------------------------------------
  //gets the output of the this.brain then converts them to actions
  think() {

      var max = 0;
      var maxIndex = 0;
      //get the output of the neural network
      this.decision = this.brain.feedForward(this.vision);

      for (var i = 0; i < this.decision.length; i++) {
        if (this.decision[i] > max) {
          max = this.decision[i];
          maxIndex = i;
        }
      }

      //<<<<<<<<<<<<<<<<<<<<<<<<<<<<<<<<<<<replace
    }
    //----------------------------------------------------------------------------------------------------------------------------------------------------------
    //returns a clone of this player with the same brian
  clone() {
    var clone = new Player();
    clone.brain = this.brain.clone();
    clone.fitness = this.fitness;
    clone.brain.generateNetwork();
    clone.gen = this.gen;
    clone.bestScore = this.score;
    return clone;
  }

  //----------------------------------------------------------------------------------------------------------------------------------------------------------------------------------
  //since there is some randomness in games sometimes when we want to replay the game we need to remove that randomness
  //this fuction does that

  cloneForReplay() {
    var clone = new Player();
    clone.brain = this.brain.clone();
    clone.fitness = this.fitness;
    clone.brain.generateNetwork();
    clone.gen = this.gen;
    clone.bestScore = this.score;

    //<<<<<<<<<<<<<<<<<<<<<<<<<<<<<<<<<<<replace
    return clone;
  }

  //----------------------------------------------------------------------------------------------------------------------------------------------------------
  //fot Genetic algorithm
  calculateFitness() {
    this.fitness = random(10);
    //<<<<<<<<<<<<<<<<<<<<<<<<<<<<<<<<<<<replace
  }

  //----------------------------------------------------------------------------------------------------------------------------------------------------------
  crossover(parent2) {

    var child = new Player();
    child.brain = this.brain.crossover(parent2.brain);
    child.brain.generateNetwork();
    return child;
  }
}
```

```
//this is a template to add a NEAT ai to any game
//note //<<<<<<<<<<<<<<<<<<<<<<<<<<<<<<<<<<<<replace
//this means that there is some information specific to the game to input here


var nextConnectionNo = 1000;
var population;
var speed = 60;


var showBest = true; //true if only show the best of the previous generation
var runBest = false; //true if replaying the best ever game
var humanPlaying = false; //true if the user is playing

var humanPlayer;


var showBrain = false;
var showBestEachGen = false;
var upToGen = 0;
var genPlayerTemp; //player

var showNothing = false;


//----------------------------------------------------------------------------------------------------------------------------------------------------

function setup() {
  window.canvas = createCanvas(1280, 720);
  //<<<<<<<<<<<<<<<<<<<<<<<<<<<<<<<<<<<<replace
  population = new Population(500);
  humanPlayer = new Player();
}
//----------------------------------------------------------------------------------------------------------------------------------------------------------
function draw() {
  drawToScreen();
  if (showBestEachGen) { //show the best of each gen
    showBestPlayersForEachGeneration();
  } else if (humanPlaying) { //if the user is controling the ship[
    showHumanPlaying();
  } else if (runBest) { // if replaying the best ever game
    showBestEverPlayer();
  } else { //if just evolving normally
    if (!population.done()) { //if any players are alive then update them
      population.updateAlive();
    } else { //all dead
      //genetic algorithm
      population.naturalSelection();
    }
  }
}
//------------------------------------------------------------------------------------
function showBestPlayersForEachGeneration() {
  if (!genPlayerTemp.dead) { //if current gen player is not dead then update it

    genPlayerTemp.look();
    genPlayerTemp.think();
    genPlayerTemp.update();
    genPlayerTemp.show();
  } else { //if dead move on to the next generation
    upToGen++;
    if (upToGen >= population.genPlayers.length) { //if at the end then return to the start and stop doing it
      upToGen = 0;
      showBestEachGen = false;
    } else { //if not at the end then get the next generation
      genPlayerTemp = population.genPlayers[upToGen].cloneForReplay();
    }
  }
}
//------------------------------------------------------------------------------------
function showHumanPlaying() {
  if (!humanPlayer.dead) { //if the player isnt dead then move and show the player based on input
    humanPlayer.look();
    humanPlayer.update();
    humanPlayer.show();
  } else { //once done return to ai
    humanPlaying = false;
  }
}
//------------------------------------------------------------------------------------
function showBestEverPlayer() {
  if (!population.bestPlayer.dead) { //if best player is not dead
    population.bestPlayer.look();
    population.bestPlayer.think();
    population.bestPlayer.update();
    population.bestPlayer.show();
  } else { //once dead
    runBest = false; //stop replaying it
    population.bestPlayer = population.bestPlayer.cloneForReplay(); //reset the best player so it can play again
  }
}
//-----------------------------------------------------------------------------------------------------------------------------------------------------------
//draws the display screen
function drawToScreen() {
  if (!showNothing) {
    //pretty stuff
    //<<<<<<<<<<<<<<<<<<<<<<<<<<<<<<<<<<<<replace
    drawBrain();
    writeInfo();
  }
}
//----------------------------------------------------------------------------------------------------------------------------------------------------------------------------------
function drawBrain() { //show the brain of whatever genome is currently showing
  var startX = 0; //<<<<<<<<<<<<<<<<<<<<<<<<<<<<<<<<<<<<replace
  var startY = 0;
  var w = 0;
  var h = 0;

  if (runBest) {
    population.bestPlayer.brain.drawGenome(startX, startY, w, h);
  } else
  if (humanPlaying) {
    showBrain = false;
  } else if (showBestEachGen) {
    genPlayerTemp.brain.drawGenome(startX, startY, w, h);
  } else {
    population.players[0].brain.drawGenome(startX, startY, w, h);
  }
}
//--------------------------------------------------------------------------------------------------------------------------------------------------------------------------------
//writes info about the current player
function writeInfo() {
  fill(200);
  textAlign(LEFT);
  textSize(30);
  if (showBestEachGen) {
    text("Score: " + genPlayerTemp.score, 650, 50); //<<<<<<<<<<<<<<<<<<<<<<<<<<<<<<<<<<<replace
    text("Gen: " + (genPlayerTemp.gen + 1), 1150, 50);
  } else
  if (humanPlaying) {
    text("Score: " + humanPlayer.score, 650, 50); //<<<<<<<<<<<<<<<<<<<<<<<<<<<<<<<<<<<replace
  } else
  if (runBest) {
    text("Score: " + population.bestPlayer.score, 650, 50); //<<<<<<<<<<<<<<<<<<<<<<<<<<<<<<<<<<<replace
    text("Gen: " + population.gen, 1150, 50);
  } else {
    if (showBest) {
      text("Score: " + population.players[0].score, 650, 50); //<<<<<<<<<<<<<<<<<<<<<<<<<<<<<<<<<<<replace
      text("Gen: " + population.gen, 1150, 50);
      text("Species: " + population.species.length, 50, canvas.height / 2 + 300);
      text("Global Best Score: " + population.bestScore, 50, canvas.height / 2 + 200);
    }
  }
}

//---------------------------------------------------------------------------------------------------------------------------------------------------

function keyPressed() {
  switch (key) {
    case ' ':
      //toggle showBest
      showBest = !showBest;
      break;
      // case '+': //speed up frame rate
      //   speed += 10;
      //   frameRate(speed);
      //   prvarln(speed);
      //   break;
      // case '-': //slow down frame rate
      //   if(speed > 10) {
      //     speed -= 10;
      //     frameRate(speed);
      //     prvarln(speed);
      //   }
      //   break;
    case 'B': //run the best
      runBest = !runBest;
      break;
    case 'G': //show generations
      showBestEachGen = !showBestEachGen;
      upToGen = 0;
      genPlayerTemp = population.genPlayers[upToGen].clone();
      break;
    case 'N': //show absolutely nothing in order to speed up computation
      showNothing = !showNothing;
      break;
    case 'P': //play
      humanPlaying = !humanPlaying;
      humanPlayer = new Player();
      break;
  }
  //any of the arrow keys
  switch (keyCode) {
    case UP_ARROW: //the only time up/ down / left is used is to control the player
      //<<<<<<<<<<<<<<<<<<<<<<<<<<<<<<<<<<<replace
      break;
    case DOWN_ARROW:
      //<<<<<<<<<<<<<<<<<<<<<<<<<<<<<<<<<<<replace
      break;
    case LEFT_ARROW:
      //<<<<<<<<<<<<<<<<<<<<<<<<<<<<<<<<<<<replace
      break;
    case RIGHT_ARROW: //right is used to move through the generations

      if (showBestEachGen) { //if showing the best player each generation then move on to the next generation
        upToGen++;
        if (upToGen >= population.genPlayers.length) { //if reached the current generation then exit out of the showing generations mode
          showBestEachGen = false;
        } else {
          genPlayerTemp = population.genPlayers[upToGen].cloneForReplay();
        }
      } else if (humanPlaying) { //if the user is playing then move player right

        //<<<<<<<<<<<<<<<<<<<<<<<<<<<<<<<<<<<replace
      }
      break;
  }
}
```

```
# NEAT Template JavaScript
```

```
class Species {

  constructor(p) {
    this.players = [];
    this.bestFitness = 0;
    this.champ;
    this.averageFitness = 0;
    this.staleness = 0; //how many generations the species has gone without an improvement
    this.rep;

    //--------------------------------------------
    //coefficients for testing compatibility
    this.excessCoeff = 1;
    this.weightDiffCoeff = 0.5;
    this.compatibilityThreshold = 3;
    if (p) {
      this.players.push(p);
      //since it is the only one in the species it is by default the best
      this.bestFitness = p.fitness;
      this.rep = p.brain.clone();
      this.champ = p.cloneForReplay();
    }
  }

  //------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------
  //returns whether the parameter genome is in this species
  sameSpecies(g) {
    var compatibility;
    var excessAndDisjoint = this.getExcessDisjoint(g, this.rep); //get the number of excess and disjoint genes between this player and the current species this.rep
    var averageWeightDiff = this.averageWeightDiff(g, this.rep); //get the average weight difference between matching genes


    var largeGenomeNormaliser = g.genes.length - 20;
    if (largeGenomeNormaliser < 1) {
      largeGenomeNormaliser = 1;
    }

    compatibility = (this.excessCoeff * excessAndDisjoint / largeGenomeNormaliser) + (this.weightDiffCoeff * averageWeightDiff); //compatibility formula
    return (this.compatibilityThreshold > compatibility);
  }

  //------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------
  //add a player to the species
  addToSpecies(p) {
    this.players.push(p);
  }

  //------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------
  //returns the number of excess and disjoint genes between the 2 input genomes
  //i.e. returns the number of genes which dont match
  getExcessDisjoint(brain1, brain2) {
      var matching = 0.0;
      for (var i = 0; i < brain1.genes.length; i++) {
        for (var j = 0; j < brain2.genes.length; j++) {
          if (brain1.genes[i].innovationNo == brain2.genes[j].innovationNo) {
            matching++;
            break;
          }
        }
      }
      return (brain1.genes.length + brain2.genes.length - 2 * (matching)); //return no of excess and disjoint genes
    }
    //------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------
    //returns the avereage weight difference between matching genes in the input genomes
  averageWeightDiff(brain1, brain2) {
      if (brain1.genes.length == 0 || brain2.genes.length == 0) {
        return 0;
      }


      var matching = 0;
      var totalDiff = 0;
      for (var i = 0; i < brain1.genes.length; i++) {
        for (var j = 0; j < brain2.genes.length; j++) {
          if (brain1.genes[i].innovationNo == brain2.genes[j].innovationNo) {
            matching++;
            totalDiff += abs(brain1.genes[i].weight - brain2.genes[j].weight);
            break;
          }
        }
      }
      if (matching == 0) { //divide by 0 error
        return 100;
      }
      return totalDiff / matching;
    }
    //------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------
    //sorts the species by fitness
  sortSpecies() {

    var temp = []; // new ArrayList < Player > ();

    //selection short
    for (var i = 0; i < this.players.length; i++) {
      var max = 0;
      var maxIndex = 0;
      for (var j = 0; j < this.players.length; j++) {
        if (this.players[j].fitness > max) {
          max = this.players[j].fitness;
          maxIndex = j;
        }
      }
      temp.push(this.players[maxIndex]);

      this.players.splice(maxIndex, 1);
      // this.players.remove(maxIndex);
      i--;
    }

    // this.players = (ArrayList) temp.clone();
    arrayCopy(temp, this.players);
    if (this.players.length == 0) {
      this.staleness = 200;
      return;
    }
    //if new best player
    if (this.players[0].fitness > this.bestFitness) {
      this.staleness = 0;
      this.bestFitness = this.players[0].fitness;
      this.rep = this.players[0].brain.clone();
      this.champ = this.players[0].cloneForReplay();
    } else { //if no new best player
      this.staleness++;
    }
  }

  //------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------
  //simple stuff
  setAverage() {
      var sum = 0;
      for (var i = 0; i < this.players.length; i++) {
        sum += this.players[i].fitness;
      }
      this.averageFitness = sum / this.players.length;
    }
    //----------------------------------------------------------------------------------------------------------------------------------------------------------------------------------

  //gets baby from the this.players in this species
  giveMeBaby(innovationHistory) {
    var baby;
    if (random(1) < 0.25) { //25% of the time there is no crossover and the child is simply a clone of a random(ish) player
      baby = this.selectPlayer().clone();
    } else { //75% of the time do crossover

      //get 2 random(ish) parents
      var parent1 = this.selectPlayer();
      var parent2 = this.selectPlayer();

      //the crossover function expects the highest fitness parent to be the object and the lowest as the argument
      if (parent1.fitness < parent2.fitness) {
        baby = parent2.crossover(parent1);
      } else {
        baby = parent1.crossover(parent2);
      }
    }
    baby.brain.mutate(innovationHistory); //mutate that baby brain
    return baby;
  }

  //------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------
  //selects a player based on it fitness
  selectPlayer() {
      var fitnessSum = 0;
      for (var i = 0; i < this.players.length; i++) {
        fitnessSum += this.players[i].fitness;
      }
      var rand = random(fitnessSum);
      var runningSum = 0;

      for (var i = 0; i < this.players.length; i++) {
        runningSum += this.players[i].fitness;
        if (runningSum > rand) {
          return this.players[i];
        }
      }
      //unreachable code to make the parser happy
      return this.players[0];
    }
    //------------------------------------------------------------------------------------------------------------------------------------------
    //kills off bottom half of the species
  cull() {
      if (this.players.length > 2) {
        for (var i = this.players.length / 2; i < this.players.length; i++) {
          // this.players.remove(i);
          this.players.splice(i, 1);
          i--;
        }
      }
    }
    //------------------------------------------------------------------------------------------------------------------------------------------
    //in order to protect unique this.players, the fitnesses of each player is divided by the number of this.players in the species that that player belongs to
  fitnessSharing() {
    for (var i = 0; i < this.players.length; i++) {
      this.players[i].fitness /= this.players.length;
    }
  }
}
```

```
class Population {

  constructor(size) {
    this.players = []; //new ArrayList<Player>();
    this.bestPlayer; //the best ever player
    this.bestScore = 0; //the score of the best ever player
    this.globalBestScore = 0;
    this.gen = 1;
    this.innovationHistory = []; // new ArrayList<connectionHistory>();
    this.genPlayers = []; //new ArrayList<Player>();
    this.species = []; //new ArrayList<Species>();

    this.massExtinctionEvent = false;
    this.newStage = false;

    for (var i = 0; i < size; i++) {
      this.players.push(new Player());
      this.players[this.players.length - 1].brain.mutate(this.innovationHistory);
      this.players[this.players.length - 1].brain.generateNetwork();
    }
  }
  updateAlive() {
      for (var i = 0; i < this.players.length; i++) {
        if (!this.players[i].dead) {
          this.players[i].look(); //get inputs for brain
          this.players[i].think(); //use outputs from neural network
          this.players[i].update(); //move the player according to the outputs from the neural network
          if (!showNothing && (!showBest || i == 0)) {
            this.players[i].show();
          }
          if (this.players[i].score > this.globalBestScore) {
            this.globalBestScore = this.players[i].score;
          }
        }
      }

    }
    //-----------------------------------------------------------------------------------------------------------------------------------------
    //returns true if all the players are dead      sad
  done() {
      for (var i = 0; i < this.players.length; i++) {
        if (!this.players[i].dead) {
          return false;
        }
      }
      return true;
    }
    //-----------------------------------------------------------------------------------------------------------------------------------------
    //sets the best player globally and for thisthis.gen
  setBestPlayer() {
    var tempBest = this.species[0].players[0];
    tempBest.gen = this.gen;


    //if best thisthis.gen is better than the global best score then set the global best as the best thisthis.gen

    if (tempBest.score >= this.bestScore) {
      this.genPlayers.push(tempBest.cloneForReplay());
      console.log("old best: " + this.bestScore);
      console.log("new best: " + tempBest.score);
      this.bestScore = tempBest.score;
      this.bestPlayer = tempBest.cloneForReplay();
    }
  }

  //------------------------------------------------------------------------------------------------------------------------------------------
  //this function is called when all the players in the this.players are dead and a newthis.generation needs to be made
  naturalSelection() {

    // this.batchNo = 0;
    var previousBest = this.players[0];
    this.speciate(); //seperate the this.players varo this.species
    this.calculateFitness(); //calculate the fitness of each player
    this.sortSpecies(); //sort the this.species to be ranked in fitness order, best first
    if (this.massExtinctionEvent) {
      this.massExtinction();
      this.massExtinctionEvent = false;
    }
    this.cullSpecies(); //kill off the bottom half of each this.species
    this.setBestPlayer(); //save the best player of thisthis.gen
    this.killStaleSpecies(); //remove this.species which haven't improved in the last 15(ish)this.generations
    this.killBadSpecies(); //kill this.species which are so bad that they cant reproduce

    // if (this.gensSinceNewWorld >= 0 || this.bestScore > (grounds[0].distance - 350) / 10) {
    //   this.gensSinceNewWorld = 0;
    //   console.log(this.gensSinceNewWorld);
    //   console.log(this.bestScore);
    //   console.log(grounds[0].distance);
    //   newWorlds();
    // }

    console.log("generation  " + this.gen + "  Number of mutations  " + this.innovationHistory.length + "  species:   " + this.species.length + "  <<<<<<<<<<<<<<<<<<<<<<<<<<<<<<<<<<")


    var averageSum = this.getAvgFitnessSum();
    var children = [];
    for (var j = 0; j < this.species.length; j++) { //for each this.species
      children.push(this.species[j].champ.clone()); //add champion without any mutation
      var NoOfChildren = floor(this.species[j].averageFitness / averageSum * this.players.length) - 1; //the number of children this this.species is allowed, note -1 is because the cl
      for (var i = 0; i < NoOfChildren; i++) { //get the calculated amount of children from this this.species
        children.push(this.species[j].giveMeBaby(this.innovationHistory));
      }
    }
    if (children.length < this.players.length) {
      children.push(previousBest.clone());
    }
    while (children.length < this.players.length) { //if not enough babies (due to flooring the number of children to get a whole var)
      children.push(this.species[0].giveMeBaby(this.innovationHistory)); //get babies from the best this.species
    }

    this.players = [];
    arrayCopy(children, this.players); //set the children as the current this.playersulation
    this.gen += 1;
    for (var i = 0; i < this.players.length; i++) { //generate networks for each of the children
      this.players[i].brain.generateNetwork();
    }
  }

  //----------------------------------------------------------------------------------------------------------------------------------------
  //seperate this.players into this.species based on how similar they are to the leaders of each this.species in the previousthis.gen
```

```
speciate() {
    for (var s of this.species) { //empty this.species
      s.players = [];
    }
    for (var i = 0; i < this.players.length; i++) { //for each player
      var speciesFound = false;
      for (var s of this.species) { //for each this.species
        if (s.sameSpecies(this.players[i].brain)) { //if the player is similar enough to be considered in the same this.species
          s.addToSpecies(this.players[i]); //add it to the this.species
          speciesFound = true;
          break;
        }
      }
      if (!speciesFound) { //if no this.species was similar enough then add a new this.species with this as its champion
        this.species.push(new Species(this.players[i]));
      }
    }
  }
  //-------------------------------------------------------------------------------------------------------------------------------------------
  //calculates the fitness of all of the players
calculateFitness() {
    for (var i = 1; i < this.players.length; i++) {
      this.players[i].calculateFitness();
    }
  }
  //-------------------------------------------------------------------------------------------------------------------------------------------
  //sorts the players within a this.species and the this.species by their fitnesses
sortSpecies() {
    //sort the players within a this.species
    for (var s of this.species) {
      s.sortSpecies();
    }

    //sort the this.species by the fitness of its best player
    //using selection sort like a loser
    var temp = []; //new ArrayList<Species>();
    for (var i = 0; i < this.species.length; i++) {
      var max = 0;
      var maxIndex = 0;
      for (var j = 0; j < this.species.length; j++) {
        if (this.species[j].bestFitness > max) {
          max = this.species[j].bestFitness;
          maxIndex = j;
        }
      }
      temp.push(this.species[maxIndex]);
      this.species.splice(maxIndex, 1);
      // this.species.remove(maxIndex);
      i--;
    }
    this.species = [];
    arrayCopy(temp, this.species);

  }
  //-------------------------------------------------------------------------------------------------------------------------------------------
  //kills all this.species which haven't improved in 15this.generations
killStaleSpecies() {
    for (var i = 2; i < this.species.length; i++) {
      if (this.species[i].staleness >= 15) {
        // .remove(i);
        // splice(this.species, i)
        this.species.splice(i, 1);
        i--;
      }
    }
  }
  //-------------------------------------------------------------------------------------------------------------------------------------------
  //if a this.species sucks so much that it wont even be allocated 1 child for the nextthis.generation then kill it now
killBadSpecies() {
    var averageSum = this.getAvgFitnessSum();

    for (var i = 1; i < this.species.length; i++) {
      if (this.species[i].averageFitness / averageSum * this.players.length < 1) { //if wont be given a single child
        // this.species.remove(i); //sad
        this.species.splice(i, 1);

        i--;
      }
    }
  }
  //-------------------------------------------------------------------------------------------------------------------------------------------
  //returns the sum of each this.species average fitness
getAvgFitnessSum() {
  var averageSum = 0;
  for (var s of this.species) {
    averageSum += s.averageFitness;
  }
  return averageSum;
}

//-------------------------------------------------------------------------------------------------------------------------------------------
//kill the bottom half of each this.species
cullSpecies() {
  for (var s of this.species) {
    s.cull(); //kill bottom half
    s.fitnessSharing(); //also while we're at it lets do fitness sharing
    s.setAverage(); //reset averages because they will have changed
  }
}


massExtinction() {
    for (var i = 5; i < this.species.length; i++) {
      // this.species.remove(i); //sad
      this.species.splice(i, 1);

      i--;
    }
  }
  //-------------------------------------------------------------------------------------------------------------------------------------------
  //              BATCH LEARNING
  //-------------------------------------------------------------------------------------------------------------------------------------------
  //update all the players which are alive
updateAliveInBatches() {
  let aliveCount = 0;
  for (var i = 0; i < this.players.length; i++) {
    if (this.playerInBatch(this.players[i])) {

      if (!this.players[i].dead) {
        aliveCount++;
```

```
          this.players[i].look(); //get inputs for brain
          this.players[i].think(); //use outputs from neural network
          this.players[i].update(); //move the player according to the outputs from the neural network
          if (!showNothing && (!showBest || i == 0)) {
            this.players[i].show();
          }
          if (this.players[i].score > this.globalBestScore) {
            this.globalBestScore = this.players[i].score;
          }
        }
      }
    }


    if (aliveCount == 0) {
      this.batchNo++;
    }
  }


  playerInBatch(player) {
    for (var i = this.batchNo * this.worldsPerBatch; i < min((this.batchNo + 1) * this.worldsPerBatch, worlds.length); i++) {
      if (player.world == worlds[i]) {
        return true;
      }
    }

    return false;


  }

  stepWorldsInBatch() {
      for (var i = this.batchNo * this.worldsPerBatch; i < min((this.batchNo + 1) * this.worldsPerBatch, worlds.length); i++) {
        worlds[i].Step(1 / 30, 10, 10);
      }
    }
    //-------------------------------------------------------------------------------------------------------------------------------------------
    //returns true if all the players in a batch are dead      sad
  batchDead() {
    for (var i = this.batchNo * this.playersPerBatch; i < min((this.batchNo + 1) * this.playersPerBatch, this.players.length); i++) {
      if (!this.players[i].dead) {
        return false;
      }
    }
    return true;
  }

}
```

```
class Player {

  constructor() {
    this.fitness;
    this.vision = []; //the input array fed into the neuralNet
    this.decision = []; //the out put of the NN
    this.unadjustedFitness;
    this.lifespan = 0; //how long the player lived for this.fitness
    this.bestScore = 0; //stores the this.score achieved used for replay
    this.dead;
    this.score = 0;
    this.gen = 0;

    this.genomeInputs = 13;
    this.genomeOutputs = 4;
    this.brain = new Genome(this.genomeInputs, this.genomeOutputs);
  }

  //---------------------------------------------------------------------------------------------------------------------------------------------------------
  show() {
      //<<<<<<<<<<<<<<<<<<<<<<<<<<<<<<<<<<<replace
    }
    //---------------------------------------------------------------------------------------------------------------------------------------------------------
  move() {
      //<<<<<<<<<<<<<<<<<<<<<<<<<<<<<<<<<<<replace
    }
    //---------------------------------------------------------------------------------------------------------------------------------------------------------
  update() {
      //<<<<<<<<<<<<<<<<<<<<<<<<<<<<<<<<<<<replace
    }
    //----------------------------------------------------------------------------------------------------------------------------------------------------------

  look() {
    //<<<<<<<<<<<<<<<<<<<<<<<<<<<<<<<<<<<replace

  }



  //---------------------------------------------------------------------------------------------------------------------------------------------------------
  //gets the output of the this.brain then converts them to actions
  think() {

      var max = 0;
      var maxIndex = 0;
      //get the output of the neural network
      this.decision = this.brain.feedForward(this.vision);

      for(var i = 0; i < this.decision.length; i++) {
        if(this.decision[i] > max) {
          max = this.decision[i];
          maxIndex = i;
        }
      }

      //<<<<<<<<<<<<<<<<<<<<<<<<<<<<<<<<<<<replace


    }
    //---------------------------------------------------------------------------------------------------------------------------------------------------------
    //returns a clone of this player with the same brian
  clone() {
    var clone = new Player();
    clone.brain = this.brain.clone();
    clone.fitness = this.fitness;
    clone.brain.generateNetwork();
    clone.gen = this.gen;
    clone.bestScore = this.score;
    return clone;
  }

  //---------------------------------------------------------------------------------------------------------------------------------------------------------------------------------
  //since there is some randomness in games sometimes when we want to replay the game we need to remove that randomness
  //this fuction does that

  cloneForReplay() {
    var clone = new Player();
    clone.brain = this.brain.clone();
    clone.fitness = this.fitness;
    clone.brain.generateNetwork();
    clone.gen = this.gen;
    clone.bestScore = this.score;
    //<<<<<<<<<<<<<<<<<<<<<<<<<<<<<<<<<<<replace
    return clone;
  }

  //---------------------------------------------------------------------------------------------------------------------------------------------------------
  //fot Genetic algorithm
  calculateFitness() {
    //<<<<<<<<<<<<<<<<<<<<<<<<<<<<<<<<<<<replace

  }

  //---------------------------------------------------------------------------------------------------------------------------------------------------------
  crossover(parent2) {
    var child = new Player();
    child.brain = this.brain.crossover(parent2.brain);
    child.brain.generateNetwork();
    return child;
  }
}
```