Trabajo Práctico Especial

Programación Orientada a Objetos

Docente: Berdún, Luis

Alumnos: Carbajo, Ezequiel - Castilla, Tomas

Fecha: 26/4/2019

Prólogo

El siguiente informe pretende ilustrar de manera funcional y objetiva el trabajo realizado por Carbajo Ezequiel y Castilla Tomas.

El proyecto consistió en el diseño y programación orientada a objetos en Java de un sistema capaz de entrenar a varias redes neuronales, de manera que estas aprendan a conducir y controlar un vehículo en base a un análisis del ambiente en el que se encuentran.

La red neuronal recibirá como datos iniciales los brindados por sensores capaces de detectar obstáculos. Se brindará la distancia a la que está cada sensor al obstáculo y se obtendrá por resultado la cantidad de aceleración y giro que deberá aplicarse al vehículo.

Dicho de otro modo, los sensores de cada vehículo serán los ojos de cada red neuronal correspondiente. En base a esos datos, se moverán los autos.

Para hacer esto posible de manera que los vehículos aprendan a evadir obstáculos y avanzar cuán rápido puedan, se deberá profundizar en el diseño y los comportamientos definidos para el sistema final.

Diseño de clases e implementación

En el diseño, podemos encontrar 3 grandes grupos de clases:

- Las clases que se utilizan para modelar todo lo que es el **entorno y escenario**, tales como las calles, los autos, las físicas y demás.
- Las clases utilizadas para implementar la **inteligencia** en sí, y su procesamiento de los datos del entorno para tomar decisiones en cuanto al control del auto.
- Las clases que unen estos dos anteriores grupos para el **entrenamiento** de la inteligencia, a través del algoritmo genético.

Entorno y escenario

En este grupo de clases, se buscó modelar todo aquello que tenga que ver con el entorno sobre el cual se van a mover los autos, y su visualización. Para esto, se crearon dos interfaces clave, y una clase "Cámara" que se encargará de mostrar debidamente cada uno de los objetos en el plano.

Dibujable:

Esta interfaz define solamente un simple método que debe tener todo aquel objeto que pueda ser visible a través de la interfaz gráfica.

→`void draw(PVector desplazamiento) :` Este método, permite mostrar en la pantalla aquel objeto que sea dibujable. Como parámetro, tenemos un PVector desplazamiento, el cual me indica el desplazamiento del objeto en el espacio relativo sobre el que estoy dibujando. Este, será de utilidad para la clase ***Cámara*** a la hora de seguir un auto, la cual será descrita más adelante.

Colisionable:

Esta interfaz, que hereda de ***Dibujable*** (lo que implica que se debe poder dibujar), contiene un conjunto de métodos que deben implementar aquellos objetos que puedan ser colisionados. Representa los obstáculos, encargados de informar si fueron chocados.

→`boolean colision(ArrayList<PVector> puntos):` Este método devuelve verdadero si alguno de los puntos pasados por parámetro está colisionando al objeto.

→`ArrayList<PVector> calcularIntersecciones(PVector p1, PVector p2) :` Este método devuelve todos los puntos de intersección del segmento p1-p2 con la instancia del ***colisionable***.

Cámara:

Esta clase es la encargada de que se dibujen todos los ***Dibujables***, y en qué posición de la pantalla. Tras definir el vehículo que debe seguir, posicionará visualmente todos los ***dibujables*** en su lugar en relación con el ***auto*** enfocado, de manera que este último siempre quede centrado. En el diseño de este sistema se optó por seguir siempre al vehículo que más fitness posea en ese instante.

→`PVector getDespl():` Este método retorna el desplazamiento actual que los ***Dibujables*** deben adquirir para centrar en ventana al vehículo enfocado.

→**`void setEnfoque(Auto d):`** Este método permite setear el enfoque de la cámara.

→**`void draw(ArrayList<Dibujable> visibles):`** Este método llama al draw de cada uno de los ***dibujables***, pasandoles el desplazamiento actual como parámetro, para poder dibujarlos de modo que el enfoque se mantenga al centro.

Estas interfaces ***Dibujable*** y ***Colisionable*** permiten implementar fácilmente nuevas clases que puedan ser "dibujadas" en el plano y "colisionadas" por otros objetos. Si necesitaramos implementar, por ejemplo, algún otro tipo de obstáculo colisionable que los autos debieran esquivar, simplemente bastaría con implementar las funcionalidades que definen cómo se dibuja el obstáculo, y como se colisiona (estos son, los correspondientes métodos de dibujable y colisionable). El resto del sistema no se vería afectado por los cambios y creaciones sobre estas clases.

Para este problema, solo se implementan aquellas clases dibujables y coleccionables para definir un escenario simple de una ruta y un vehículo que la recorra. Estas clases y sus métodos son descritos a continuación:

Auto:

Implementando ***Dibujable***, es la clase que posee todo lo que necesito saber sobre el auto y su visualización. Posee atributos tales como la posición en el plano, su velocidad, su angulo de rotacion, sus dimensiones y sus ***sensores***.

→**`void draw(PVector desplazamiento):`** Este método implementa la función abstracta de ***Dibujable***, permitiendo dibujar en la pantalla al objeto. En este caso, el método dibuja un rectángulo dado por el ancho y alto del vehículo rotado en el ángulo que su variable "ángulo" indique.

→**`ArrayList<PVector> getPuntosColision():`** Este método retorna aquellos puntos del vehículo que pueden colisionar con el entorno. En este caso, devuelve los 4 vértices que definen el rectángulo, rotados según el ángulo al que mira el vehículo.

→**`void setGirar(float giro):`** Este método permite setear la intensidad de giro que el vehículo realizará en el próximo instante, siendo positiva para rotar a la derecha, negativa para rotar a la izquierda, y nula para mantener el auto derecho. (Se debe aclarar que en esta implementación, el auto rotará en base a este valor y la magnitud de la velocidad, evitando que realice un giro estando quieto; siendo esto equivalente a girar el volante sin velocidad).

→**`void setAcelerar(float aceleracion):`** Este método permite definir al vehículo que tanta aceleración deberá adquirir en el próximo instante (positiva o negativamente).

→**`void mover():`** Este método permite mover al vehículo según sus condiciones de velocidad y giro actuales.

→**`PVector getPos():`** Este método retorna la posición actual del vehículo en el plano.

→**`ArrayList<Double> getValoresSensor(Colisionable ruta):`** Este método retorna el conjunto de datos obtenidos por los ***sensores*** del vehículo.

→**`void reset():`** Resetea la posición, la velocidad y el ángulo del vehículo a cero.

→**`void actualizarSensores():`** Este método privado es el encargado de mover y rotar los ***sensores*** del vehículo.

Sensor:

Esta clase que implementa ***Dibujable*** (haciendo sus instancias visibles), contiene los datos relativos al sensor del vehículo, encargado de detectar obstáculos en su cercanía. Esto es, los dos puntos que me definen el segmento del sensor, su ángulo, y los puntos en los que ese segmento interseca con algún ***colisionable***.

→**`void draw(PVector desplazamiento):`** Este método implementa la función abstracta de ***Dibujable***, permitiendo dibujar en la pantalla al objeto. En este caso, simplemente dibujara una línea con los dos puntos que definen el segmento del sensor, y una circunferencia por cada punto que este segmento intersecta con los ***colisionables***.

→**`void setPos(PVector punto, PVector direccion)`**: Este método permite setear los dos puntos del segmento del sensor.

→**`double calcularValor(Colisionable r) :`** Este método retorna la distancia al ***colisionable*** más cercano, en la dirección que apunta el sensor. Si no hay intersección, retorna el tamaño del sensor.

→**`void mover(Auto anclaje, PVector punto, PVector direccion):`** Permite actualizar la posicion y angulo del sensor, de manera de mantenerse “anclado” a su vehículo correspondiente.

Checkpoint:

Esta clase que implementa ***Dibujable***, se basa en un simple punto que se encuentra en el centro de una ***calle***. Tiene como función principal colaborar en el cálculo del fitness score de cada vehículo, de manera de saber que tanto progreso realizó sobre

la pista marcada. Hay una relación lineal directa entre la cantidad de checkpoints pasados por un auto y su fitness score.

→**`void setPos(PVector pos):`** Este método setea la posición del checkpoint en el plano.

→**`PVector getPos():`** Este método permite ver la posición actual del checkpoint.

→**`void draw(PVector desplazamiento):`** Este método implementa la función abstracta de ***Dibujable***, permitiendo dibujar en la pantalla al objeto. En este caso dibuja una circunferencia cuyo centro es la posición del checkpoint.

Calle:

Esta clase, que implementa ***Colisionable***, es un camino simple definido por dos segmentos, que representará un fragmento del camino que el vehículo deberá tomar.

→**`void draw(PVector desplazamiento):`** Este método implementa la función abstracta de ***Dibujable***, permitiendo dibujar en la pantalla al objeto. En este caso, se dibujan los dos segmentos que me definen a la calle.

→**`boolean colision(ArrayList<PVector> puntos):`** Este método indica si uno de los puntos está a una cierta distancia de alguno de los segmentos de la calle. En caso de positivo, se interpreta como colisión, retornando verdadero.

→**`ArrayList<PVector> calcularIntersecciones(PVector p1, PVector p2):`** Este método calcula los puntos de intersección entre el segmento p1-p2 con los dos segmentos de la calle.

Ruta:

Esta clase implementando ***Colisionable*** posee un conjunto de objetos ***Colisionables*** que la componen, junto con los ***checkpoints*** del recorrido, generados automáticamente en el medio de cada ***calle***. De forma abstracta, esta clase se encarga de armar el recorrido completo que todos los vehículos deberán tomar.

→**`void draw(PVector desplazamiento):`** Este método implementa la función abstracta de ***Dibujable***, permitiendo dibujar en la pantalla al objeto. En este caso, la ruta pide a todos los objetos ***colisionables*** que contiene que se dibujen llamando a sus métodos draw.

→**`boolean colision(ArrayList<PVector> puntos):`** Este método llamará a todos al método “colisión” de todos los ***colisionables*** de la ruta. Devolverá verdadero, indicando colisión con la ruta, si hay colisión con alguno de sus elementos

colisionables, delegandoles la responsabilidad de que cada uno de ellos chequee la colisión de los puntos consigo mismos.

→**`ArrayList<PVector> calcularIntersecciones(PVector p1, PVector p2):`** Este método calcula y devuelve los puntos de intersección del segmento p1-p2 con todos los objetos ***colisionables*** de la ruta. Al igual que el método anterior, le delega la responsabilidad de que cada uno de sus ***colisionables*** calcule las intersecciones de p1-p2 consigo mismos.

→**`void addCalle(Calle c):`** Este método permite agregar una ***calle*** a la ruta, calculando el ***checkpoint*** que se encuentra en ella y agregandolo a la lista de ***checkpoints***.

→**`void cargarCalles(String rutaArchivo):`** Este método permite cargar un conjunto de ***calles*** desde un archivo.

→**`Checkpoint getCP(int i):`** Este método retorna el i-esimo ***checkpoint*** de la lista de ***checkpoints***, ordenada según el orden en que fueron agregadas las calles. Es de utilidad a la hora de chequear el ***checkpoint*** al que debe llegar cada ***auto***, de manera de saber si lo pasó, y en caso tal, cual es el siguiente.

→**`void addElemento(Colisionable e):`** Este método permite agregar un ***colisionable*** a la calle. Podría ser cualquier obstáculo que los vehículos puedan colisionar

Inteligencia

Este grupo de clases compone todo lo necesario para la implementación de la red neuronal. Se diseñó de manera abstracta, de modo de poder usarse en más de un contexto. Para este sistema, tomará las decisiones de acelerar, desacelerar y girar el auto teniendo en cuenta los valores de entrada definidos por los ***sensores*** del mismo

ElemRedNeuronal:

Esta interfaz define el conjunto de métodos que un elemento de una red neuronal debe implementar. Permite agregar nuevas implementaciones de elementos de redes neuronales, para quizás, crear distintos tipos de estas. En este caso, las clases que la implementan son solamente aquellas necesarias para crear una red feed forward.

→**`ArrayList<Double>getActivacion(ArrayList<Double>entradas):`**
Este método genera las salidas correspondientes, según los valores de entrada.

→**`ArrayList<Double> getParametros():`** Este método retorna todos los parámetros del elemento que son usados para calcular las salidas.

→**`void modificarParametro(int nroP, double valorP):`** Este método modifica el parámetro nroP por el valor valorP.

→**`int getCantParametros():`** Este método retorna la cantidad de parámetros que posee el elemento.

→**`void setCantEntradas(int c):`** Este método setea la cantidad de entradas que tendrá el elemento. Esto me sirve para saber cuantos parametros voy a necesitar a la hora de calcular las salidas.

→**`ElemRedNeuronal clone():`** Este método retorna un elemento con la misma estructura que el actual. Esto es, clonar en base a estructura, y no en base a los datos (parámetros).

Neurona:

Implementando ***ElemRedNeuronal***, este es el elemento más simple que puede tener una red neuronal. Calcula un una salida en base a una suma ponderada de las entradas, mas un bias. Estos valores son pasados por la funcion de activacion que la neurona tenga, para generar el valor de salida final.

→**`ArrayList<Double>getActivacion(ArrayList<Double>entradas):`**
Este método retorna un ArrayList con un solo elemento, el cual será el valor de activación de la neurona.

→**`ArrayList<Double> getParametros():`** Este método retorna los valores de todas las ponderaciones y del bias, en orden.

→**`void modificarParametro(int nroP, double valorP):`** Este método modifica un parámetro de la neurona por el valor dado. Si el nroP es invalido, no se modifica nada.

→**`int getCantParametros():`** Este método retorna la cantidad de ponderaciones que la neurona tenga +1 (el bias).

→**`void setCantEntradas(int c):`** Permite setear la cantidad de entradas de la neurona.

→**`ElemRedNeuronal clone():`** Este método retorna una neurona que tenga la misma cantidad de parámetros que la actual, pero no necesariamente con los mismos valores.

Capa:

Implementando ***ElemRedNeuronal***, agrupa un conjunto de ***neuronas*** de iguales características estructurales. Cada una de ellas, generará su activación tomando en cuenta los valores que llegan de la capa anterior. En caso de no existir una capa anterior, se toman como entrada los valores de entrada de la red.

→**`ArrayList<Double>getActivacion(ArrayList<Double>entradas):`** : Este método retorna los valores de activación de cada ***neurona*** de la capa, teniendo en cuenta los valores de la capa anterior o de la entrada de la red, según corresponda.

→**`ArrayList<Double> getParametros():`** Este método retorna todos los parámetros de la capa. Esto es, todos los parámetros de todas las ***neuronas***, en orden.

→**`void modificarParametro(int nroP, double valorP):`** Este método permite modificar un valor de un parámetro de la capa. Primero, según el numero de parametro, se decide en qué ***neurona*** va, y luego se le pasan los datos a la ***neurona*** para que modifique su parámetro.

→**`int getCantParametros():`** Este método retorna la cantidad total de parámetros de la capa, teniendo en cuenta la cantidad de parámetros de cada ***neurona***.

→**`void setCantEntradas(int c):`** Este método permite pre setear la cantidad de entradas que tendrá la capa, para poder ajustar la cantidad de parámetros de cada ***neurona***.

→**`ElemRedNeuronal clone():`** Este método clona la capa, y las ***neuronas*** que contenga, pero solo teniendo en cuenta aspectos estructurales (cantidad de parámetros).

Red:

Implementando ***ElemRedNeuronal***, este elemento se compone de todos los anteriores. Posee un conjunto de ***capas*** con ***neuronas*** en cada una de ellas, que le permite generar sus valores de salida a partir de sus entradas teniendo en cuenta cada una de sus activaciones.

→**`ArrayList<Double>getActivacion(ArrayList<Double>entradas):`** Este método retorna las salidas correspondientes, pidiendo las activaciones a la última ***capa***, quien pedirá las activaciones de la ***capa*** anterior, y así sucesivamente hasta llegar a la primera, la cual usa como entrada los valores de 'entradas'.

→**`ArrayList<Double> getParametros():`** Este método retorna todos los parámetros de la red en orden, pidiendose a cada una de las ***capas*** que contiene.

→**`void modificarParametro(int nroP, double valorP):`** Este método permite modificar un parámetro de la red, por un valor. Primero, la red buscará en que ***capa*** debe modificarse el parámetro y luego le delega la responsabilidad a dicha ***capa*** para que lo modifique.

→**`int getCantParametros():`** Este método retorna la cantidad total de parámetros de la red, teniendo en cuenta la cantidad de parámetros de cada ***capa***.

→**`void setCantEntradas(int c):`** Este método setea la cantidad de entradas de la red para ajustar los parámetros de cada **capa**.

→**`ElemRedNeuronal clone():`** Este método retorna una red con las mismas características estructurales. Esto se logra pidiendo el clone a cada ***capa***, y cada una de ellas pedirá el clone a cada ***neurona***.

→**`void addCapa(Capa c):`** Este método añade una ***capa*** al final de la red, y la conecta con su anterior ***capa*** (la que antes era la última).

FuncionActivacion:

La función de activación es la encargada de trasladar el resultado de la suma ponderada a otro conjunto de valores, variando según la función de la que se trate. Esta interfaz define un método que toda función de activación debe tener:

→**`double calcularValor(double x):`** Este método retorna el valor recalculado de x, según como la instancia de la función de activación lo defina.

Inicialmente, se crearon dos funciones de activación, pero a la hora de crear la red que se utiliza en el algoritmo solo se utilizó una.

FuncionSigmoide:

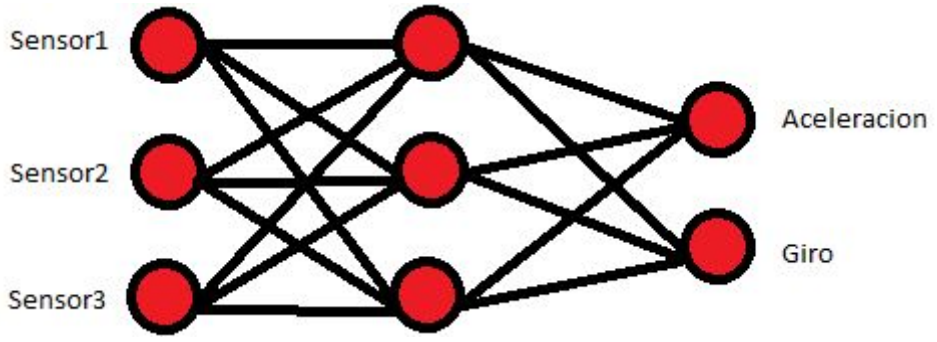
Esta clase representa la función cuyo valor es calculado a partir de la ecuación f(x) = 1/(1+e^-x). Define el comportamiento del método `calcularValor(...)` y posibilita que el resultado final se encuentre entre los valores 0 y 1.

TangenteHiperbolica:

Esta clase (la única utilizada por ser más adecuada para la resolución del problema) implementa el método `calcularValor(...)` retornando el valor correspondiente según la función tanh(x), permitiendo solo valores entre -1 y 1.

Además, esta interfaz permite la creación de nuevas formas de calcular el valor de la funcion de activacion, agregando clases que implementen el método `calcularValor(...)`.

Para este problema, se creó una red de 3 entradas (3 sensores del auto), una capa oculta de 3 neuronas, y finalmente una capa de salida de 2 neuronas (giro y aceleración a aplicar). Esto resulta en una red neuronal como la que se ve en la figura. Todas las neuronas tendrán la función de activación ***TangenteHiperbolica***, la cual, al tener un rango de [-1,1] permitirá setear un giro y una aceleración adecuadas.



Esta red contará con un total de 20 parámetros, contando todas las ponderaciones y biases.

Geom:

Esta clase fue utilizada en múltiples ocasiones para cálculos de intersecciones de segmentos, distancias de un punto a una recta, etc. Esto fue de gran utilidad en momentos de chequear si un punto estaba sobre un segmento, colisiones, o calcular los valores del sensor teniendo en cuenta las colisiones del mismo con la pared. Se decidió utilizar una clase estática, dado que esto permite poder hacer uso de sus métodos sin tener que tener una instancia de la misma dentro del módulo en el que se utiliza.

→**double distanciaEuclidea(PVector a, PVector b) :** Este método retorna la distancia entre el punto a, y el punto b.

→**double distanciaSegmento(PVector pc, PVector pa, PVector pb) :** Este método retorna la distancia del punto pc al segmento definido por pa-pb.

→**double distanciaRecta(PVector punto, PVector puntoLinea1, PVector puntoLinea2) :** Este método retorna la distancia de 'punto' a la recta que pasa por 'puntoLinea1' y 'puntoLinea2' . Es utilizado en el cálculo de distanciaSegmento(...).

→**PVector puntoInterseccion(float x1,y1,x2,y2,x3,y3,x4,y4) :** Este método retorna el punto de intersección entre los segmentos definidos por (x1,y1) - (x2,y2) y (x3,y3) - (x4,y4).

→**PVector puntoInterseccion(PVector p1, p2,p3,p4) :** Este método funciona igual que el método anterior, solo que en este los parámetros se especifican como PVectors.

Entrenamiento

Este conjunto de clases se encarga de ser la conexión entre el entorno y la inteligencia. Su función en conjunto involucra la creación de tantos ***autos*** como inteligencias y su debida administración durante el transcurso del programa. Eso incluye:

- Sincronizar y setear los datos que son enviados entre la inteligencia y su respectivo ***auto*** a lo largo de la ejecución, de manera que el auto sepa que hacer un vez enviado el panorama a la inteligencia.
- Actualizar el fitness score que en cada instante el auto adquiere. En este caso, según cantidad de ***Checkpoints*** pasados, cercanía al actual y tiempo transcurrido.
- Mutar las inteligencias una vez chocados todos los ***autos***, y resetear los autos para volver a iniciar.

Entrenamiento:

Esta clase administra la conexión que cada ***auto*** poseerá con su respectiva ***red*** neuronal, de manera que el pasaje de parámetros sea posible entre ellos, se actualice el puntaje de cada ***auto*** y, una vez todos los autos chocados, se muten las ***redes*** y se reinicien los autos.

→ `void update():` Este método administra de forma general la ejecución de cada generación, actualizando datos si aún quedan ***autos*** andando y llamando a la mutación y reinicio de ***autos*** si todos chocaron.

→ `void terminarEntrenamiento():` Este método llamado por el update una vez todos los ***autos*** choquen, se encarga de administrar las cruzas y mutaciones entre ***redes*** para generar una nueva generación completa.

→ `void reset():` Este método resetea los datos necesarios para que la nueva generación inicie en el mismo contexto en que iniciaron sus generaciones pasadas (sin puntos, en la posición (0,0), etc).

→ `Dupla getMejor():` Este método busca y devuelve la ***dupla*** Auto/Individuo con la mayor cantidad de puntos.

→ `void calcularPuntos(Individuo ind, Auto auto):` Este método setea la cantidad de puntos actuales que adquirió el ***individuo*** pasado por parámetro. Es necesario pasar por parámetro el ***individuo*** y su respectivo ***auto***.

→ `ArrayList<Dibujable> getAutos():` Este método devuelve un listado de todos los ***autos*** que posee la instancia.

FormaMutacion:

Esta interfaz agrupa las posibles implementaciones del método mutar(). Fue creada con el objetivo de poder aplicar más de un tipo de mutación si así se desea, e implementar cuantas se consideren.

→ `void mutar(ArrayList<Individuo> inds):` Este método modifica parámetros de ***redes*** neuronales que se pasen por parámetro según el criterio con el que se implementa la mutación.

MutacionRandom:

Implementando ***FormaMutacion***, esta clase recibe por parámetro el rango o tamaño de modificación aleatoria que pueden tener los parámetros de una ***red*** ingresada. Consiste en una mutación aleatoria de los parámetros de la ***red*** con cierto rango de modificación.

MutacionDual:

Implementando ***FormaMutacion***, esta clase recibe por parámetro ***individuos*** seleccionados para cruzarse y generar uno nuevo a partir de esta cruza, con una ligera mutación aleatoria en cada una que les brinde identidad.

MutacionCompuesta:

Implementando ***FormaMutacion***, esta clase recibe por parámetro dos ***FormaMutacion*** y el porcentaje de ***redes*** que se mutará con la primer ***FormaMutacion*** ingresada (de manera que el porcentaje restante sea mutado con la segunda ***FormaMutacion***).

MutacionTotalRandom:

Implementando ***FormaMutacion***, esta clase muta a las ***redes*** que reciba por parámetro brindándoles nuevos parámetros completamente aleatorios.

Individuo:

Esta clase representa la combinación entre la ***red*** neuronal, sus puntos adquiridos y ciertos datos necesarios para el cálculo de estos últimos. Al implementar el método “compareTo()” permite el ordenamiento de las ***redes*** según la cantidad de puntos que sus individuos hayan adquirido, y de este modo encontrar con mayor facilidad las más prometedoras.

Dupla:

Esta clase contiene, como bien dice el nombre, una dupla ***Auto***/***Individuo***, la cual representa la necesaria conexión directa entre la ***red*** neuronal con el vehículo que controla. Implementa junto con ***Individuo*** el método “compareTo()” de manera de permitir el ordenamiento de las redes neuronales según la cantidad de puntos de sus ***Individuos***.

Comportamiento general

Una buena alternativa a la hora de diseñar y describir el comportamiento de un programa es el uso de abstracción y enumeración de comportamientos generales, de manera de mantener una coherencia en el diseño y detectar clases necesarias para cumplir cada funcionalidad.

A continuación se ilustrara el paso a paso del programa de manera funcional por frame una vez inicializado lo básico:

1. La cámara enfoca al auto con el mejor puntaje.
2. Se plasma en pantalla todos los objetos visibles, centrando el mejor auto.
3. Se extraen los valores brindados por los sensores de cada auto.
4. Se ingresan los valores de cada sensor en su correspondiente red neuronal linkeada.
5. Se actualizan el giro y aceleración para el próximo frame de cada auto según la salida de su correspondiente red neuronal.
6. Se actualiza la posición de los autos según las instrucciones recibidas por su red.
7. Se calculan los puntos de cada auto en base a su posición actual en el mapa.
8. Se chequea si alguno de los autos vivos chocó con un colisionable, y en caso tal se lo tilda como muerto.
9. Si quedan autos vivos, se regresa al paso 1.
10. Si no hay más autos vivos, se realiza la cruza de genes y mutación entre las redes de la forma descrita en el apartado "**Entrenamiento elegido**" para generar las nuevas redes neuronales.
11. Se asigna a un auto cada nueva red creada para la nueva generación.
12. Se resetean los datos de cada auto.

De esta forma, los parámetros de cada red se irán ajustando en cada iteración, teniendo en cuenta los parámetros de los individuos que más fitness score lograron, añadiendo también una cierta mutación para tener la chance de mejorar ese fitness score en próximas iteraciones.

Entrenamiento elegido

El proceso de aprendizaje de las redes neuronales se realiza una vez tras cada fin que se le da a una generación, ya sea por el choque de todos los vehículos, por finalizar el tiempo definido por generación, o por simplemente terminar su ejecución anticipadamente en caso de que el usuario solicite un reinicio rápido.

Para este sistema, se optó por mutar en la siguiente serie de pasos:

- Las primeras 2 redes neuronales que mayor puntaje obtuvieron permanecen intactas en la siguiente generación, de manera de asegurar que las siguientes generaciones no empeorarán.
- De entre las redes neuronales restantes, se toma el 70% por ciento y se resetean con nuevas ponderaciones en cada una de sus neuronas. Estas ponderaciones serán creadas a partir de distintas cruzas aleatorias entre las mejores redes neuronales, con una ligera variación en sus valores de manera de darle mayor identidad a la nueva red.
- Por último, para asegurarnos de que alguna red logre encontrar la cota mínima global y no simplemente una mínima local, las redes neuronales restantes se resetean con ponderaciones en cada una de sus neuronas completamente aleatorias.