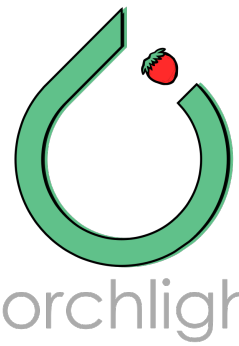
Ovidiu Marian Aricsan
Dragos Constantin Butnariu
Carlos Pérez Mateos

Torchlight

Juego Snake guiado por una Red Neuronal

Grado Superior
Desarrollo de Aplicaciones Multiplataforma



Colegio Salesiano Santo Domingo Savio

Junio, 2021

Agradecimientos

En primer lugar queremos agradacer a todos los profesores que nos habeis guíado en estos dos años para conseguir llegar hasta aqui.

Tras estos dos extraños años estamos muy ilusinados de llegar al final de ésta bella etapa en nuestras vidas como estudiantes, pero tenemos claro que el camino hasta aqui ha sido mejor que la meta.

En nuestros corazones y mentes siempre quedarán presentes los sentimientos generados hacia todo el equipo de profesionales que nos habeis enseñado todo lo tecnico y lo no tan tecnico.

Ha habido momentos muy duros, de mucho estrés pero los agradecemos ya que de esta forma nos convertís en grandes profesionales

Índice general

Capítulo 1

Memoria

1.1. Presentación

Resumen

Torchlight es un proyecto que unifica diferentes técnicas de desarrollo de software para conseguir que una red neuronal controle el juego de la serpiente.

Se hace referencia al uso de diferentes técnicas de software ya que el juego y la red neuronal se han programado en distintos lenguajes y es imprescindible que ambos se comuniquen.

El juego se ha realizado en el lenguaje C++, mientras que la red neuronal se ha realizado en Python.

Como mecanismo de comunicación entre los dos procesos, hemos optado por el uso de D-Bus

Repositorio en Gitlab `gitlab.com/txemagon/torchlight`

Motivación

La parte más interesante del proyecto ha sido la red neuronal, además de la comunicación entre los procesos mediante D-Bus y el desarrollo gráfico mediante el framework Qt.

Llama la atención como funciona el mundo de las redes neuronales y queríamos iniciarnos en este campo, que a dia de hoy, es muy demandado y cada vez más accesible para todo el mundo, gracias a la gran cantidad de herramientas que nos ayudan a desarrollar distintos tipos de redes neuronales.

Previamente a la realización del proyecto desconocíamos por completo todo acerca de este campo, por lo tanto, no supimos si la elección sería un proyecto muy complejo o muy básico.

Lo que más nos ha motivado para elegir este proyecto ha sido aprender algo nuevo y complejo. Siempre hemos intentado aprender lo mismo todos los integrantes del proyecto y compartir conocimiento entre todos, para que al final de éste terminemos con la misma base.

Alcance

Objetivo

El objetivo principal del proyecto fue programar el juego del Snake con un algoritmo desarrollado por nosotros y una Red Neuronal que aprenda a moverse por el plano, que aprenda a no chocarse con los límites del plano o con su propio cuerpo, que aprenda a acercarse a la fruta y que consiga aumentar su tamaño.

1.2. Exposición

Juego Snake

Motivo de elección

Para este proyecto, hemos querido utilizar diferentes lenguajes de programación, de forma que pudiéramos implementar una comunicación entre procesos.
Para la elección del lenguaje del juego, hemos elegido *C++* ya que es un lenguaje orientado a objetos, potente, y con el cuál, los conocimientos que se adquieren se pueden implementar en otros muchos lenguajes de programación.

Por otra parte, hemos hecho uso del framework *Qt*. Con este framework, también orientado a objetos, hemos desarrollado desde la interfaz de la aplicación hasta la conexión con la red neuronal.
Nos hemos decantado por Qt por ser multiplataforma, software libre, código abierto y por la cantidad y calidad de su documentación en línea.

Diseño de la Aplicación

Se ha querido mantener una estructura organizada, creando las siguientes clases:

UML Juego

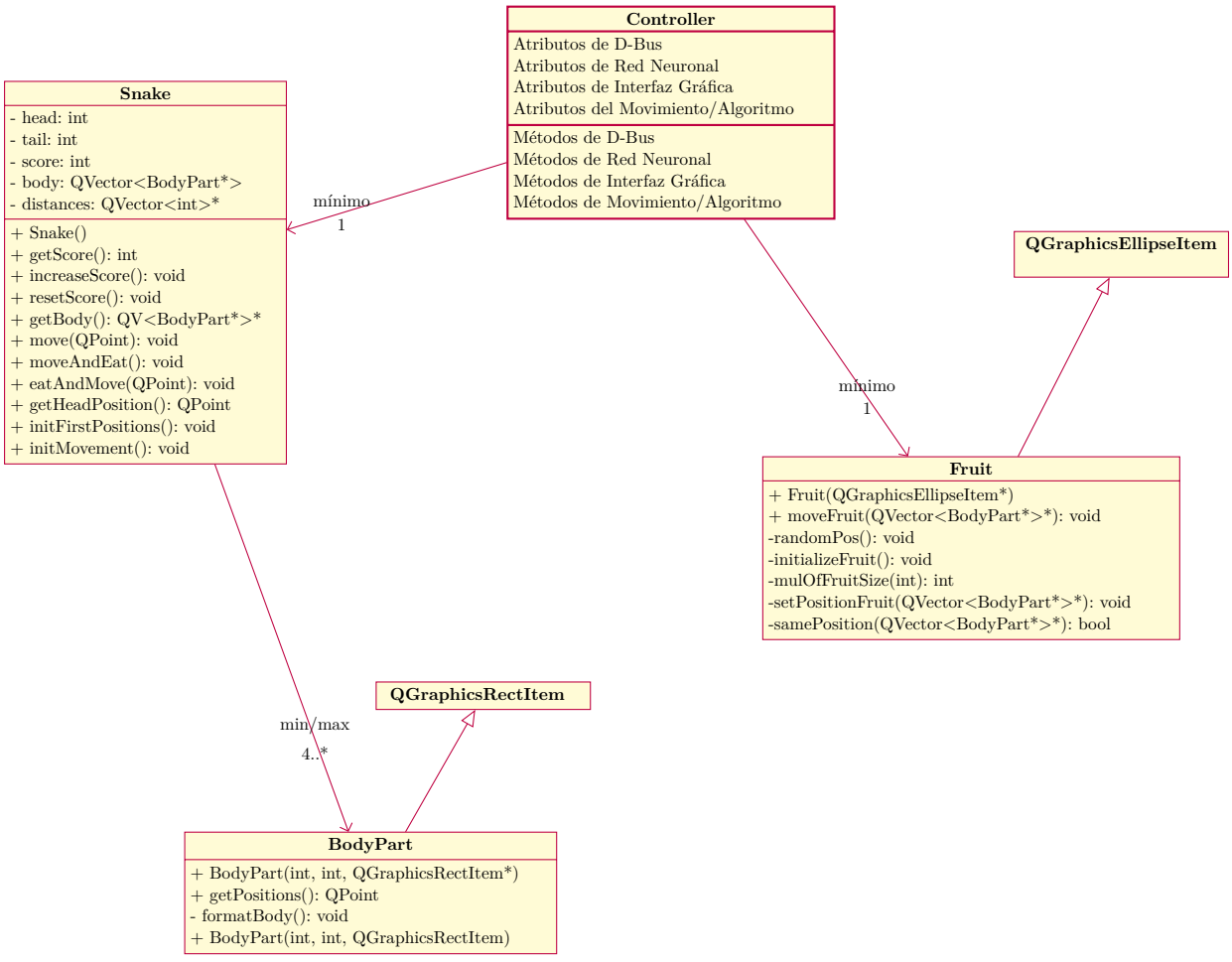


Figura 1.1: UML de la arquitectura del juego

- *Snake*: Esta clase contiene el propio cuerpo de la serpiente, formado por un vector de objetos de la clase *BodyPart*, y la puntuación de la misma en el juego. Por otra parte, se encarga del movimiento de ésta, del que más adelante en este documento se explicará su desarrollo, y de la posibilidad de cumplir su objetivo; de ahora en adelante, comer la fruta.

- *BodyPart*: Esta clase, que hereda de la clase de QT `QGraphicsRectItem`, contiene unas coordenadas x e y, puesto que cada posición del cuerpo de la serpiente contiene unas coordenadas del plano.

- *Fruit*: Esta clase, que hereda de la clase de QT `QGraphicsEllipseItem`, contiene las coordenadas x e y en el plano, y se encarga de generar, cada vez que las posiciones de la cabeza de la serpiente coincidan con las de la fruta, unos nuevos valores para estas coordenadas, totalmente aleatorias entre las posiciones del plano, respetando el cuerpo de la propia serpiente, puesto que no deben generarse en una misma celda dos elementos diferentes.

- *Controller*: Esta clase se encarga de comunicar la serpiente con la fruta, puesto que desde aquí se inicializan objetos de ambas clases. A su vez, las reglas del juego se reflejan en esta clase, impidiendo que la serpiente se choque con los límites del plano y con su propio cuerpo. Si esto ocurriera, la partida termina. Por último, comunicará mediante *D-Bus*, los datos que necesite la red neuronal, como son las posiciones de la serpiente en cada momento.

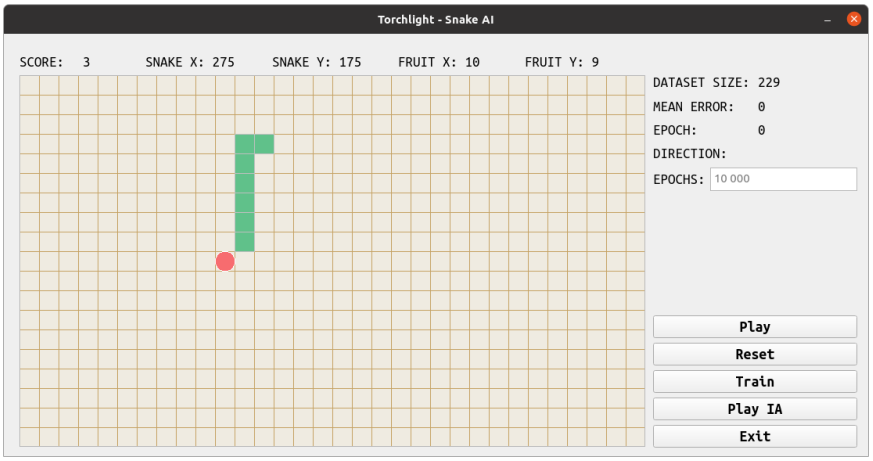


Figura 1.2: Interfaz de la aplicación

Desarrollo del Algoritmo de Movimiento

Uno de los puntos principales de este juego, es el algoritmo utilizado para el movimiento de la serpiente. Para lograr entenderlo, se comenzará explicando cómo está creado el plano.

El plano El plano consta de 5 filas y 5 columnas, de forma que, para hacer referencia a una celda en concreto, utilizamos unas coordenadas de orden (x, y). La primera posición del plano, entendiendo primera como el origen $(0, 0)$, se ubica en la parte superior izquierda, de forma que cuanto más se avance a la derecha y hacia abajo, aumentarán la x y la y, respectivamente, como se puede ver en la Figura : 1.3

(0,0)					(0,5)
(5,0)					(5,5)

Figura 1.3: El plano del juego

La serpiente

La forma en la que se almacena la serpiente, o lo que es lo mismo, las posiciones en el plano de cada parte de ésta, es en un vector formado de instancias de la clase BodyPart, de forma que, para cuando la serpiente tenga una longitud de 4, el vector tendría la siguiente estructura:

$$\Big((x,y),(x,y),(x,y),(x,y)\Big)$$

$$\Downarrow$$

$$\text{Ej:}\Big((2,0),(2,1),(0,0),(1,0)\Big)$$

Al contrario de lo que pueda parecer, la posición con índice 0 del vector, no tiene por qué corresponder a la cola de la serpiente. Para conocer en qué posición se encuentran las coordenadas de cabeza y cola, se almacena dos variables que contienen, cada una de ellas, el índice de dichas posiciones.
Para el ejemplo del vector anterior, las variables podrían ser las siguientes:

```
1 head = 1   // (2,1)
2 tail = 2   // (0,0)
```

Fragmento de Código 1.1: Índices de cabeza y cola del vector

La fruta

La fruta se genera en una posición aleatoria del plano, de la que se conoce su posición en x y su posición en y, para comprobar cuándo la serpiente cumple el objetivo. Cuando esto ocurre, se vuelve a generar una nueva posición.

El objetivo

El objetivo del juego es que la serpiente se coma la fruta, esto es que tanto la posición en x como en y de la cabeza de la serpiente coincida con las posiciones de la fruta. Para ello, la serpiente ha de moverse, o lo que es lo mismo, editar las coordenadas de cada una de sus posiciones.
Por ejemplo, para la serpiente en la siguiente posición:

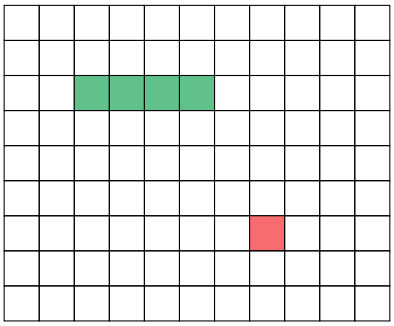


Figura 1.4: Tablero

Para que la serpiente siga de frente (derecha para el usuario), debe aumentar en uno cada una de las equis de la serpiente. Para que gire a su izquierda (arriba para el usuario), debe aumentar la y de la cabeza de la serpiente en uno, mientras que el resto de posiciones deben adquirir las coordenadas de su posición superior.

En el caso de este proyecto, el algoritmo de movimiento es el siguiente:

Movimiento de la serpiente sin comer la fruta

Para el movimiento normal de la serpiente, edita únicamente el valor de la posición de la serpiente equivalente a la cola, el cual adquiere las nuevas coordenadas de la serpiente, es decir, la nueva posición se almacena en la posición que guardaba anteriormente la cola, de forma que las coordenadas de la serpiente se almacenan en el vector como una cola circular.

A continuación se muestra un ejemplo:

Si la serpiente continua en su misma dirección, la siguiente posición sería $(2, 8)$, y para evitar editar todas las coordenadas del vector, el valor de la cola (en este caso índice 0), adquirirá el valor de la siguiente posición $(2, 8)$.
Para calcular el valor de la siguiente posición, se le suma a la posición de la cabeza las siguientes coordenadas dependiendo de la dirección a la que gire la serpiente:

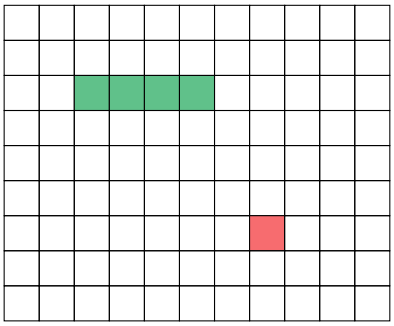


Figura 1.5: Tablero

Dirección	Coordenadas
Arriba	(0,-1)
Abajo	(0,1)
Izquierda	(-1,0)
Derecha	(1,0)

Figura 1.6: Tabla de direcciones y coordenadas.

```
1  // tail = índice del vector con coordenadas de la cola
2  // head = índice del vector con coordenadas de la cabeza
3  // ADVANCE = Tamaño de cada celda del plano
4
5  body.at(tail)->setX( headPosition.x() + direction.x()*ADVANCE );
6  body.at(tail)->setY( headPosition.y() + direction.y()*ADVANCE );
```

Fragmento de Código 1.2: Movimiento estándar

Movimiento de la serpiente comiendo la fruta

En el momento que la serpiente come la fruta, la longitud del vector debe aumentar en uno, para que al usuario le dé la sensación de crecimiento.

Para conseguir esto, en primer lugar se debe conocer la siguiente posición de la cabeza, para que la serpiente continúe moviéndose, pero a diferencia del algoritmo anterior, *la cola permanecerá en el vector*. De esta forma, la serpiente tendrá una longitud mayor.

Para conseguir la siguiente posición a la cabeza, se hace de la misma manera que en el moviento normal:

```
1  // body: new QVector<BodyPart>
2  int x = body.at(head)->x() + direction.x() * BODY_SIZE;
3  int y = body.at(head)->y() + direction.y() * BODY_SIZE;
```

Fragmento de Código 1.3: Movimiento mientras come

Y para que la longitud aumente, se inserta una posición con los valores conseguidos en la Figura 1.4 en la siguiente posición a la cabeza:

```
// body: new QVector<BodyPart>
body.insert(body.begin() + (head+1), new BodyPart(x, y));

// Se actualizan los índices de cabeza y cola:
head++;
tail = head+1;
```

Fragmento de Código 1.4: Movimiento mientras come

La matriz resultante, para un movimiento hacia al frente, sería la siguiente:

$$\Big((2,0),(2,1),(0,0),(1,0)\Big) \qquad \text{Fruta: } (3,1) \quad \text{Sig. Posición: } (3,1)$$

$$\downarrow$$

$$\Big((2,0),(2,1),\underbrace{(3,1)}_{Nueva},(0,0),(1,0)\Big)$$

Implementación

A continuación se explicará la implementación práctica del código del juego de la serpiente:

En el constructor de la clase `Controller` podemos encontrarnos con la inicialización de la interfaz, configuraciones de la ventana de la aplicación y la creación del D-Bus.

```
Controller::Controller(QWidget *parent)
    : QMainWindow(parent) , ui(new Ui::Controller)
{
    ui->setupUi(this);  // Configura la interfaz para el widget indicado

    /* Centrar ventana */
    move(QGuiApplication::screens()
         .at(0)->geometry().center() - frameGeometry().center());

    initializeGraphics();
    setDirections();

    // Vectores para entrenar la red
    inputs    = new QVector<int>();
    targetKey = new QVector<char>();

    createDBus();
}
```

Fragmento de Código 1.5: Constructor de la clase `Controller`

Se hace la llamada a los métodos `initializeGraphics` y `setDirections`, los cuales tienen las siguientes funciones:

- `initializeGraphics`
  Crea la escena de la aplicación, establece los límites de ésta y dibuja el fondo del plano.
- `setDirections`
  Inicializa el atributo de clase `directions`, con las 4 direcciones posibles que puede elegir la serpiente en el juego.

```
directions[0].setX(1),  directions[0].setY(0);  // Derecha
directions[1].setX(-1), directions[1].setY(0);  // Izquierda
directions[2].setX(0),  directions[2].setY(-1); // Arriba
directions[3].setX(0),  directions[3].setY(1);  // Abajo
```

Fragmento de Código 1.6

A la hora de añadir un nuevo objeto a la escena, por ejemplo, al inicio del juego tanto cada posición de la serpiente como la de la fruta, se hace uso del siguiente método:

```
void
Controller::addObject(QGraphicsItem *object)
{
    graphicScene->addItem(object);
}
```

Fragmento de Código 1.7

El tiempo que tarda la serpiente en ejecutar cada movimiento está establecido por un objeto de tipo `QTimer` como el siguiente:

```
// Controller.h
QTimer *timer;

// Controller.cpp
timer = new QTimer(this);
timer->start(TIMER_SPEED);
```

Fragmento de Código 1.8

Cuando el objeto `timer` mande la señal timeout, se conecta con el método de esta misma clase, llamado `updateSnake`:

```
connect(timer, SIGNAL(timeout()), this, SLOT(updateSnake()));
```

Fragmento de Código 1.9: Conexión de un método al objeto de tipo QTimer

Este método ejecutado periódicamente, cumple las siguientes funciones:
ELEMENTO 1 Calcula las distancias de la serpiente a los bordes del tablero:

```
// BODY_SIZE: Tamaño en píxeles de cada posición de la serpiente

// Distancia hacia la derecha
distances->append((SCENE_WIDTH - snakeHead.x()) / BODY_SIZE);
// Distancia hacia la izquierda
distances->append((snakeHead.x() + BODY_SIZE) / BODY_SIZE);

// Distancia hacia abajo
distances->append((SCENE_HEIGHT - snakeHead.y()) / BODY_SIZE);
// Distancia hacia arriba
distances->append((snakeHead.y() + BODY_SIZE) / BODY_SIZE);
```

Fragmento de Código 1.10

ELEMENTO 2 Genera el *target* (2.2) para enviar a la red neuronal, añadiendo al `QVector targetKey` los siguientes valores dependiendo de cual fuera el movimiento correcto en cada situación del juego:

Izquierda	Derecha	Frente
1,0,0	0,1,0	0,0,1

Figura 1.7: Tabla de direcciones y coordenadas.

ELEMENTO 3 Por otra parte, comprueba que la serpiente no se choca con su propio cuerpo, con el metodo `cannibalism`, y evitando que se choque con los límites del plano, con el método `hintWall`:

```
bool
Controller::cannibalism()
{
  QPoint headPosition = snake->getBody()
                                          ->at(snake->getHead())
                                          ->getPositions();

  for (int i=0; i<snake->getSize(); i++)
      if (i != snake->getHead())
          if (headPosition.x() == snake->getBody()->at(i)->x() &&
              headPosition.y() == snake->getBody()->at(i)->y() )
              return true;
  return false;
}
```

Fragmento de Código 1.11

```
bool
Controller::hintWall()
{
    int headX = snake->getHeadPosition().x();
    int headY = snake->getHeadPosition().y();

    if( (headX >= SCENE_WIDTH) || (headX < 0) )
        return true;

    else if ((headY >= SCENE_HEIGHT) || (headY < 0 ))
        return true;

    else return false;
}
```

Fragmento de Código 1.12

D-Bus

Motivo de elección

Al tener el juego creado en C++ y la red neuronal en Python, hemos tenido que elegir un método de comunicación entre procesos *(IPC)* para que la red neuronal tenga la posibilidad de recibir datos desde el juego y enviar datos al mismo.

Nos hemos decantado por utilizar D-Bus como mecanismo de comunicación entre los dos procesos, ya que permite una transmisión de datos fiable y se consigue tener los dos programas muy desacoplados, es decir, que podremos tener procesos que se comunican entre sí, pero que no dependen unos de otros para funcionar de forma correcta.

Funcionamiento

La implementación oficial de D-Bus se gestiona mediante tres capas.

- La biblioteca `libdbus`, que contiene toda la implementación de D-Bus.
- El demonio en el sistema, que hace uso de la biblioteca y permite tener un bus de mensajes, al cual se pueden conectar distintos procesos para que exista comunicación entre ellos.
- Adaptaciones de terceros para utilizar en distintos lenguajes o frameworks con un nivel superior de abstracción.

En la Figura 1.8 se muestra como sería la comunicación entre procesos sin utilizar D-Bus.

Y en la Figura 1.9 se puede ver la implementación de D-Bus para comunicar los mismos procesos.

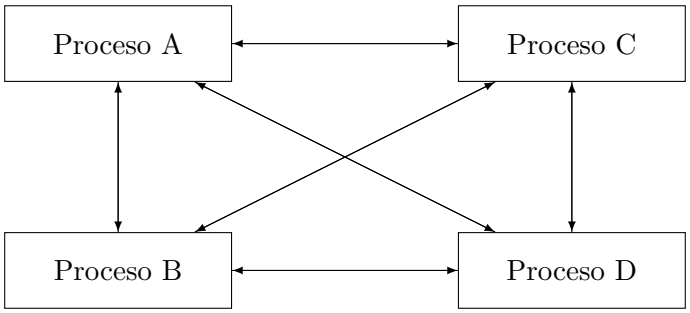


Figura 1.8: Implementación IPC mediante D-Bus

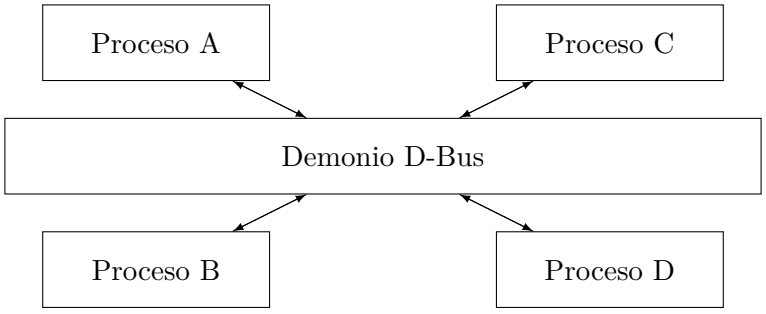


Figura 1.9: Implementación IPC mediante D-Bus

Como se puede observar en la Figura 1.8, todos los procesos están conectados entre sí, uno por uno y de forma muy acoplada, es decir, que dependen unos de otros para su correcto funcionamiento.

En cambio, en la Figura 1.9 vemos que los procesos se conectan a un demonio del sistema y se comunican con este mismo. En este caso los procesos dependen del demonio D-Bus, pero no de los demás procesos.

Tipos de D-Bus

D-Bus cuenta con dos especificaciones en su implementación, y esto depende del caso de uso que se le vaya a dar.

- *Bus de Sistema*, disponible para cualquier usuario y cualquier proceso, se utiliza para la comunicación del sistema operativo con la sesión de escritorio de los usuarios.
- *Bus de Sesión*, disponible en cada sesión de usuario y facilita la comunicación entre diferentes procesos en el mismo entorno de escritorio.

En nuestro caso, hemos utilizado el tipo D-Bus de Sesión, ya que era el que mejor se adaptaba a nuestras necesidades.

Componentes D-Bus

El funcionamiento interno de D-Bus cuenta con diferentes componentes que en algunos casos debemos utilizar para una buena implementación, y en otras ocasiones podremos prescindir de ellos.

El *modelo D-Bus* es la estructura que crea el bus de comunicación en el demonio D-Bus. Los nombres de los buses se suelen separar por puntos, como por ejemplo, un bus de sesión puede ser el siguiente: `org.freedesktop.secrets`.

Los nombres de los buses deben ser únicos, independientemente de si son de sistema o de sesión.

Los *objetos* pertenecen a buses creados previamente y tienen parecido a los objetos que se crean en lenguajes de programación orientados a objetos. Los objetos poseen *métodos* y *señales*, que se denominan *miembros* del objeto. Los miembros de los objetos se declaran en las *interfaces* y éstas se encuentran en *rutas* del objeto.
Los métodos se pueden ejecutar desde un cliente, teniendo en cuenta que el cliente tiene que establecer una conexión al bus, y también pueden recibir las señales que envíe el objeto.

Las *interfaces* contienen la declaración de métodos y señales dentro de un objeto D-Bus. Las interfaces las crea D-Bus de forma automática teniendo en cuenta, en el caso de C++, el nombre de la clase.

Las *rutas* se definen para cada objeto, y en ellas se encuentran las interfaces, que a su vez contienen métodos y señales.

Los *métodos* son las partes de código que se pueden invocar desde otro proceso que esté conectado al bus definido previamente, y estos constan de argumentos y de un valor de retorno.

Las *señales* son difusiones de mensajes desde un objeto al bus al que está conectado, para que cualquier proceso interesado en dicho objeto capture la señal y la utilice.

En la Figura 1.10 se puede ver de forma gráfica como se organizan jerárquicamente los componentes dentro de D-Bus.

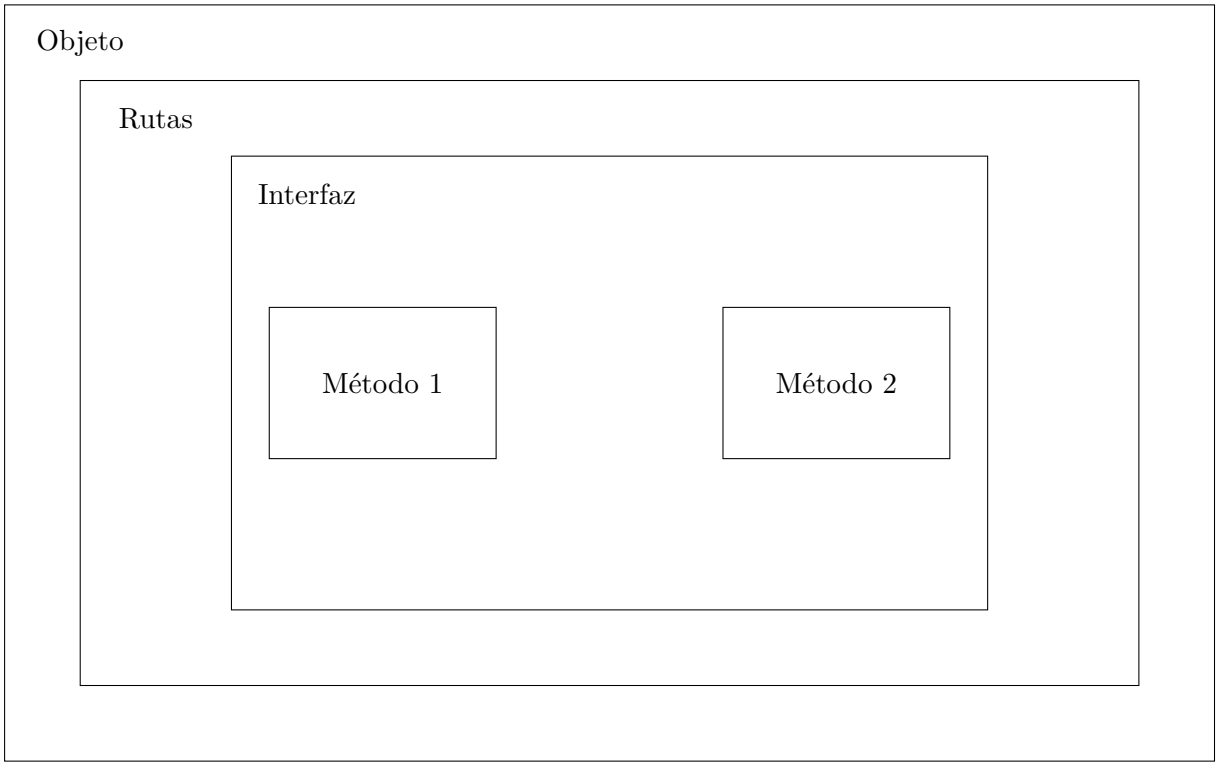


Figura 1.10: Jerarquía de los componentes D-Bus

Implementación

En nuestro caso la implementación de D-Bus ha sido de la siguiente forma:

- *Conexión:* en C++ creamos un bus D-Bus con un nombre único, como se puede ver en el Fragmento de Código 1.13 (Anexo: 2.28.
  Se puede apreciar que al crear una conexión se especifica el tipo de bus que es, en este caso un bus de sesión.

```
1 QDBusConnection::sessionBus().registerService("nombre.del.dbus");
```

Fragmento de Código 1.13: Crea un bus D-Bus

- *Objeto:* registramos un objeto, en el caso de Qt, debe ser un objeto de la clase `QObject`. En nuestro caso la clase `Controller` hereda de la clase `QMainWindow`, que ésta a su vez hereda de `QWidget` y esta última hereda de `QObject`, por lo tanto, podemos hacer uso de la clase `Controller` como servidor D-Bus.
  En el Fragmento de Código 1.14 se observa como se registra el objeto actual, que es un puntero a este mismo objeto (`Controller`), con una ruta personalizada y con los métodos declarados como `public slots`.

```
QDBusConnection::sessionBus()
                        .registerObject(
                                  "/path",
                                  this,
                                  QDBusConnection::ExportAllSlots);
```

Fragmento de Código 1.14: Registra un objeto en el D-Bus junto a la ruta y métodos

- *Ruta del objeto:* por defecto cualquier objeto tiene una raíz, pero también se podrán definir las rutas propias, como se puede observar en el Fragmento de Código 1.14, la ruta definida al registrar el objeto es `/path`.
- *Interfaz:* como se ha explicado previamente, D-Bus nos crea una interfaz de forma automática basándose en la clase donde se implementa.
- *Métodos:* se han creado varios métodos para los datos de entrada a la red neuronal, para los datos esperados de la misma, métodos de información, de control y de movimiento para la serpiente.
  En el Fragmento de Código 1.15 tenemos los métodos declarados como `public slots`, accesibles desde otro proceso mediante D-Bus.

```
void newDirection(int direction);
int getTargetSize();
int getAtTarget(int position);
int getInputsSize();
int getAtInputs(int position);
int getAtDistance(int in);
int getThreshold();
int isPredict();
int getEpochs();
void setLabelError(double err, int epoch);
```

Fragmento de Código 1.15: Métodos accesibles mediante el bus de D-Bus

Podemos observar la implementación que se ha realizado en C++ en la Figura 1.11 de forma gráfica.

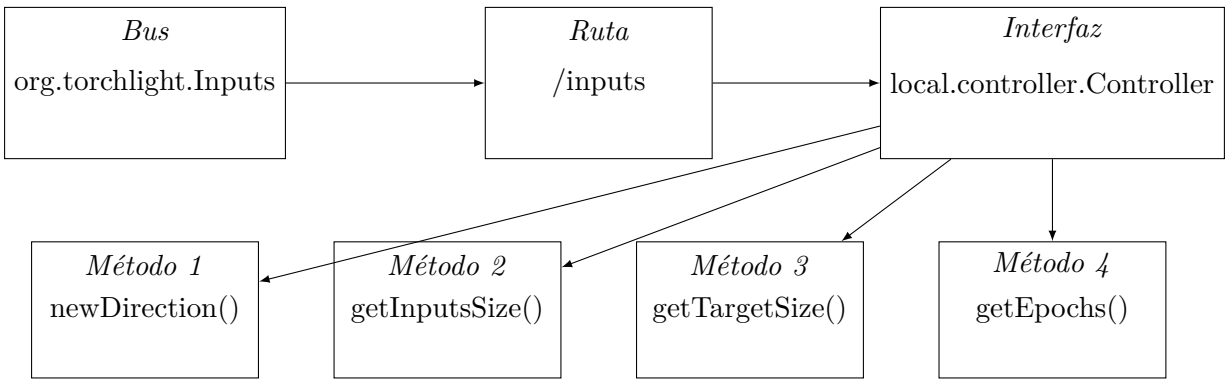


Figura 1.11: Implementación D-Bus C++

- `org.torchlight.Inputs`: nombre del bus en D-Bus

- `/inputs`: ruta propia del objeto
- `local.controller.Controller`: nombre de la interfaz
- Varios métodos como ejemplo
  - `newDirection`: desde la red neuronal se llama a este método cuando esta se encuentra en modo predicción. Recibe como parámetro un número entero, en función del número modifica la dirección de la serpiente (Anexo: 2.37).
  - `getInputsSize`: este método al ejecutarse devuelve un número entero con el tamaño del vector de los datos de entrada a la red neuronal (Anexo: 2.31).
  - `getTargetSize`: este método retorna el tamaño del vector correspondiente a los datos esperados (Anexo: 2.29).
  - `getEpochs`: este método obtiene el valor introducido en el campo de texto de la interfaz gráfica que hace referencia al número de épocas para entrenar la red neuronal, si no se introduce un número o se introduce un dato incorrecto la cantidad de épocas es 10.000 (Anexo: 2.33).

El objeto contiene una ruta raíz y la ruta propia que hemos definido.

La ruta contiene distintas interfaces, que una de ellas es la que nos crea D-Bus.

Dentro de la interfaz propia se encuentran nuestros métodos.

Lo que se ha realizado en el juego en es un servidor que crea la conexión al demonio D-Bus y a su vez define el nombre del bus. Una vez realizado este proceso se define una ruta para el objeto, la interfaz y dentro de esta interfaz se exportan los métodos.

Desde la red neuronal en Python no debemos crear un nuevo bus, simplemente nos conectamos al bus generado e invocamos los métodos. Desde la red neuronal debemos conectarnos al bus que se ha creado en C++, para ello utilizamos el método que vemos en el Fragmento de Código 1.16, como se puede observar el tipo de bus al que nos conectamos es de tipo sesión.

```
1     bus = dbus.SessionBus()
```

Fragmento de Código 1.16: Conexión a un bus de sesión desde python

Para poder acceder a los métodos se tendrá que especificar el nombre del bus y la ruta de la cual queremos obtener el objeto, para ello utilizamos el método del Fragmento de Código 1.17, donde se especifica el nombre del bus y la ruta.

```
1     self.dbus_obj = bus.get_object('org.torchlight.Inputs', '/inputs')
```

Fragmento de Código 1.17: Obtención del objeto D-Bus especificando el nombre y la ruta

Para ejecutar los métodos desde python se utilizará el método `get_dbus_method`, como se puede observar en el Fragmento de código 1.18. Éste está disponible para el objeto que ha retornado el anterior método 1.17. El primer parámetro es el nombre del método a ejecutar, el segundo el nombre de la interfaz donde se encuentra el método y para los métodos que reciben parametros se pasan en los paréntesis externos.

```
1 self.dbus_obj.get_dbus_method('metodo', dbus_interface='interfaz')(parámetro)
```

Fragmento de Código 1.18: Ejecución de un método en C++ desde Python

El gráfico de la implementación entre el juego y la red neuronal corresponde a la Figura 1.12 y los pasos que sigue son los siguientes

1. Ejecución del juego
2. Se crea el D-Bus
3. En el juego se generan los datos (*inputs y target*) para red neuronal
4. Desde el juego se ejecuta un nuevo proceso, el de la red neuronal
5. La red neuronal se conecta al D-Bus y obtiene los datos
6. Entrena con los datos obtenidos
7. Predice una nueva salida
8. En el modo predicción envía los movimientos al D-Bus y este al juego

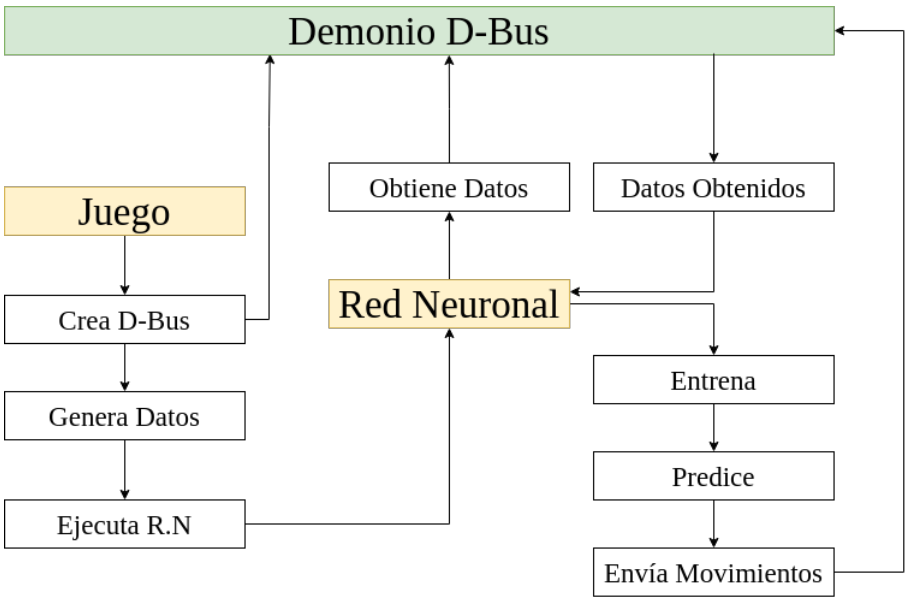


Figura 1.12: Implementación D-Bus entre el juego y la red neuronal

Depuración

Para comprobar el correcto funcionamiento del bus creado en D-Bus tenemos varias opciones, una sería programar un script en algún lenguaje interpretado como Python o JavaScript o programar un compilado en C o C++.

Otro método es hacer uso de herramientas ya existentes y fiables, que pueden ser de terminal o de forma gráfica, unos ejemplos de estas herramientas son los siguientes:

- *CLI:* `dbus-monitor`, `dbus-send`
- *GUI:* `d-feet`, `bustle`

En nuestro caso, hemos usado la herramienta gráfica `d-feet`, que nos muestra todos los buses existentes en el demonio D-Bus, tanto los de tipo sesión como de tipo sistema, como se aprecia en la Figura 1.13

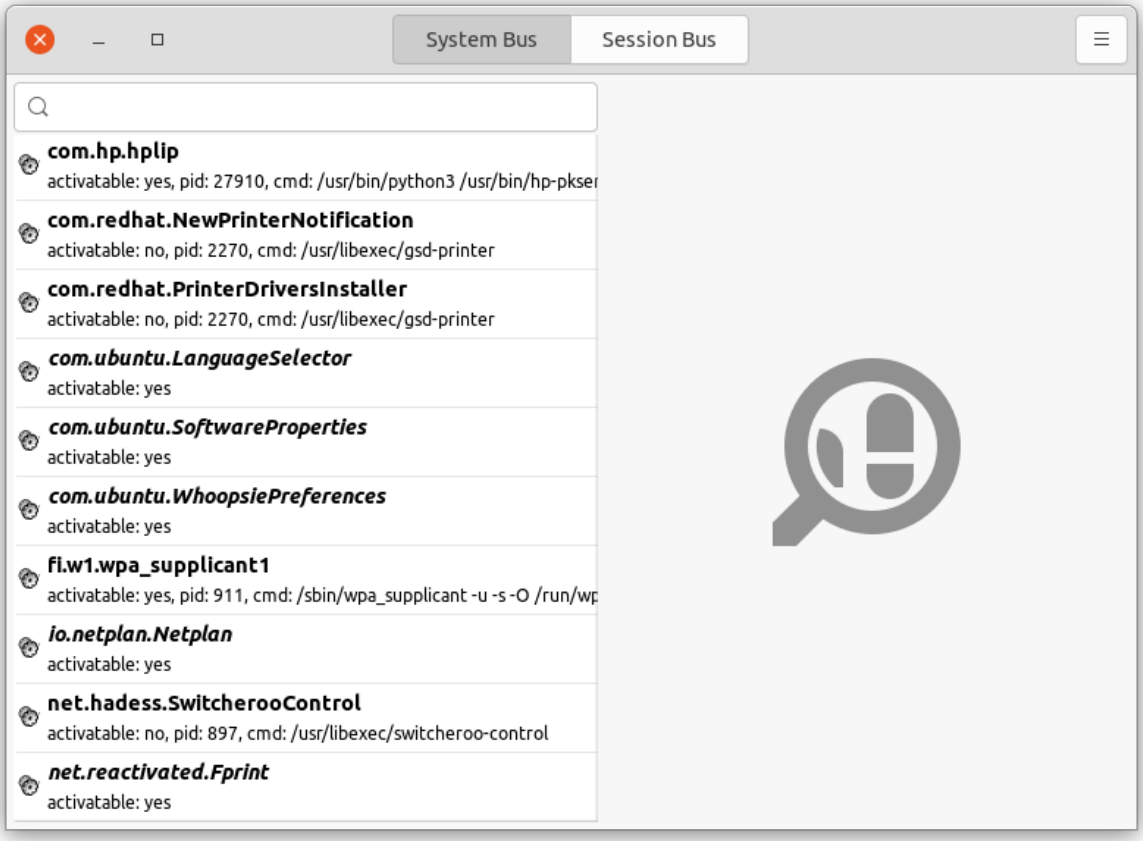


Figura 1.13: Herramienta D-Feet

Al crear el bus de conexión en D-Bus se puede hacer una búsqueda mediante el nombre que le hemos indicado al programarlo y vemos que D-Feet nos muestra las rutas del objeto, las interfaces dentro de cada ruta y por último los métodos de las interfaces como se observa en la Figura 1.14.

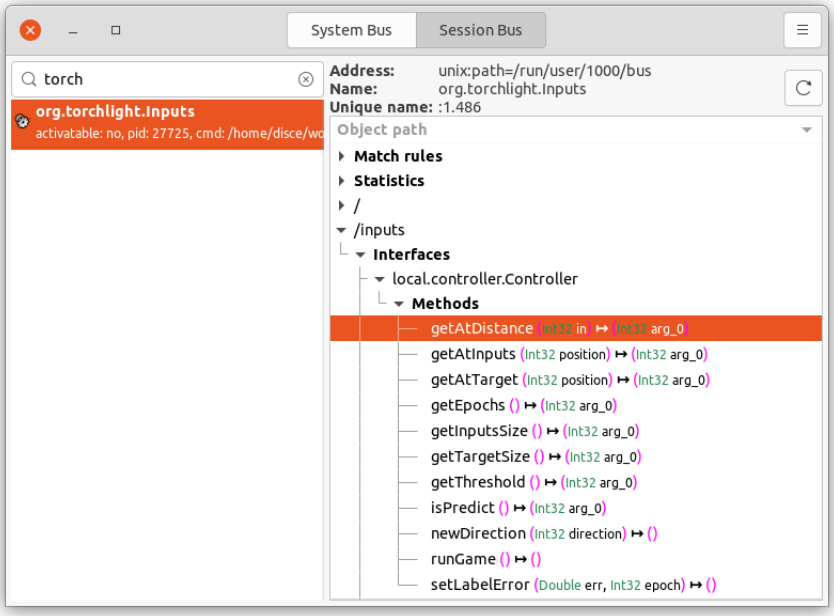


Figura 1.14: Inspección del bus mediante D-Feet

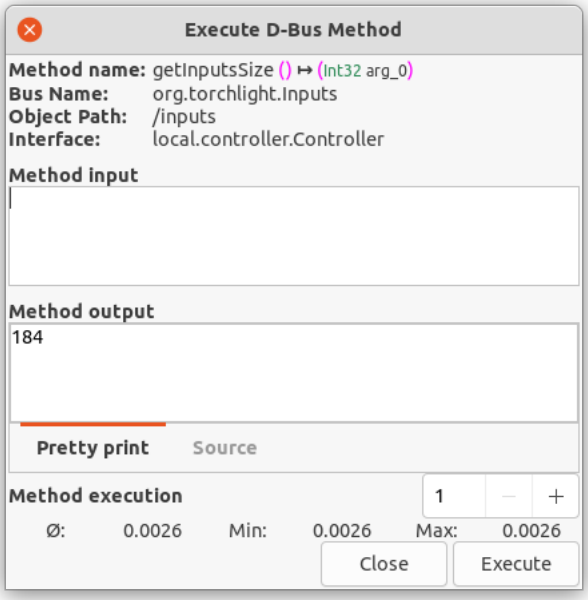


Figura 1.15: Un método concreto en D-Feet

Por último, desde D-Feet existe la posibilidad de ejecutar un método en concreto.

Para ello abrimos un dicho método haciendo clic sobre éste y en la nueva ventana podremos ejecutar método. En la nueva ventana se muestra información acerca de éste (*nombre, bus, ruta, interfaz*) como se observa en la Figura 1.15.

Red Neuronal

Elecciones y sus motivos

Para definir la red neuronal que se encarga de aprender a jugar, se ha decidido utiliza la librería *PyTorch* [Glosario: 2.2]. El motivo de la elección de dicha librería es la simplicidad del entendimiento del código, teniendo así una curva de aprendizaje más suave, ideal para empezar a desarrollar redes neuronales.

Para el modelo de la red neuronal se ha decidido utilizar un *Perceptrón Multicapa* explicado más adelante. El motivo de esta elección es el resultado de un análisis de los diferentes tipos de modelos según su complejidad, determinando así, que para el problema planteado en este proyecto contando con la variable del tiempo, la mejor solución es el tipo de modelo *Perceptrón Multicapa*. Por último, como tipo de aprendizaje del modelo neuronal se ha decidido utilizar *aprendizaje supervisado*. El motivo de la elección es la sencillez del algoritmo y la rapidez de sus resultados.

Descripción

Desde que el hombre dio el paso de dibujar un engranaje a fabricarlo, soñó con la idea de crear una inteligencia capaz de simular el comportamiento humano, dando pie a definir como inteligencia artificial el intento de imitar el comportamiento y la inteligencia humana. Para ello, se idearon a lo largo del tiempo varios conjuntos de algoritmos, de los cuales los más prometedores se sitúan en el grupo del machine learning o aprendizaje automático. Existen varios algoritmos aplicables, uno de ellos es el *Perceptrón*.

Dentro del grupo de algoritmos de machine learning, nos encontramos con el más avanzado y el que más futuro asegura; *deep learning*, utilizado en este proyecto.
El *deep learning* es un subgrupo de *machine learning*, el cual utiliza redes neuronales como algoritmo de aprendizaje. Las redes neuronales con mayor capacidad de cálculo se definen como deep learning, por lo que una de sus características es el uso de más de una capa oculta en su arquitectura, mientras que una red neuronal densa, es aquella en donde todas las neuronas de una capa se conectan con todas las neuronas de la siguiente.

Fase de aprendizaje

Definición

Una red neuronal aprende en la denominada fase de aprendizaje, donde cada neurona está conectada con las demás de la siguiente capa, y cada conexión tiene un peso (weight) asociado a ella.

El peso es un número, y representa la importancia de ese valor o característica por encima de las demás, es decir, de los datos entregados a la red, hay algunos que representan una mayor importancia o que son claves para dar una u otra salida, por lo que el peso de estas últimas será mayor.

Funcionamiento

A priori se desconocen los pesos ideales de cada conexión [Glosario: 2.2], por lo que se asignan los pesos de forma aleatoria, y se calcula la media del error comparando la salida de la red con la salida esperada mediante las denominadas funciones de pérdida. Después, mediante algoritmo de *Descenso de Gradiente*, se obtienen los valores de esa iteración o época con los que ajustar

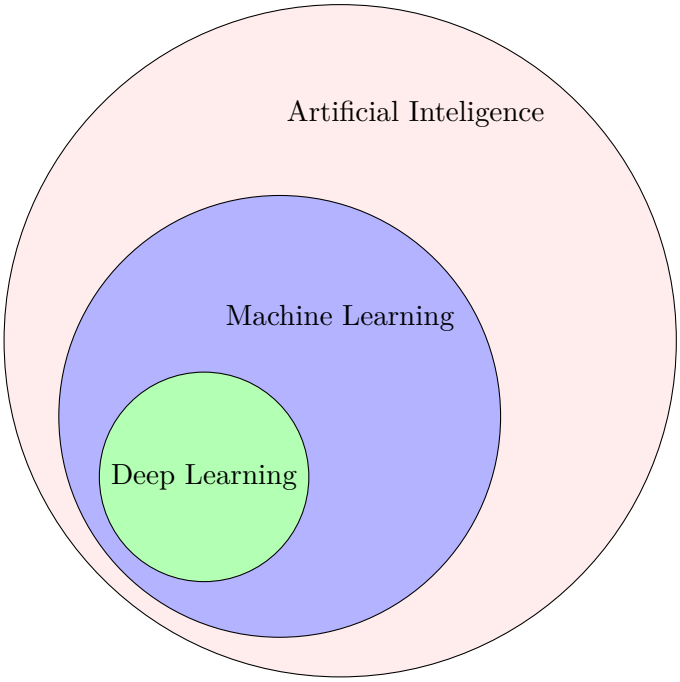


Figura 1.16: Esquema de diferentes algoritmos IA

los pesos, incrementado o decrementando su valor con el objetivo de obtener una salida lo más fiable posible.

Repetimos este proceso hasta obtener un valor de error lo más cercano a 0.

Por lo tanto, el aprendizaje se define como la fase donde la red neuronal ajusta los pesos con el objetivo de predecir la salida correcta según unos datos de entrada.

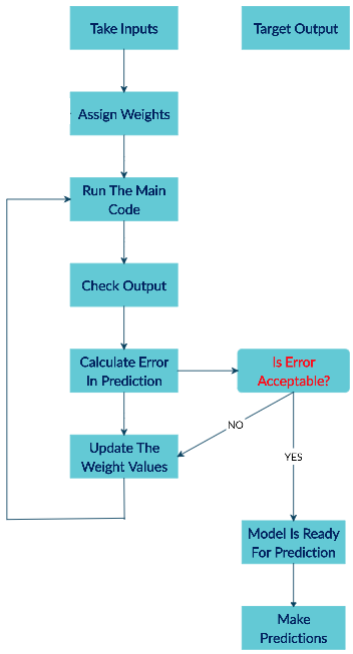


Figura 1.17: Esquema Fase de entrenamiento IA

1. Obtener inputs/target

2. Asignar valores aleatorios a los pesos de los inputs o características de entrada
3. Ejecutar el modelo neuronal en fase de entrenamiento
4. Obtener la media del valor de error en la predicción
5. Actualizar los valores de los pesos de los inputs mediante el Descenso de Gradiente
6. Repetir la fase de entrenamiento con los nuevos pesos hasta encontrar el mínimo error posible
7. El modelo neuronal está listo para realizar predicciones

Aprendizaje supervisado

Se ha decidido utilizar el tipo de aprendizaje supervisado. La diferenciación de este tipo de aprendizaje es la utilización de dos conjuntos de datos, a partir de ahora conjuntos de datos o datasets, para la fase de entrenamiento.

Disponemos de un dataset para los inputs, que almacena todos los ejemplos con los que deseamos entrenar a nuestro modelo, y un dataset de outputs denominado target, que almacena el objetivo de la predicción por cada conjunto de inputs. Veamos un ejemplo de como quedarían los datasets del operador lógico *OR*:

Inputs	Target
0 0	0
0 1	1
1 0	1
1 1	1

Algoritmo

Perceptrón

El perceptrón es un tipo de algoritmo de machine learning y es la unidad mínima funcional de una red neuronal. Su funcionamiento está basado en el de una neurona biológica, donde ésta recibe información por medio de nuestros sentidos, la procesa y generan una respuesta que propaga a las demás neuronas conectadas a ella.

De la misma manera que una neurona biológica, una neurona artificial tiene como objetivo descubrir patrones en los datos recibidos como entrada, de los cuáles dar una respuesta.

Por lo tanto, se dispone de unas neuronas de entrada, que en conjunto se denominan capa de entrada, a la que se pondrá a su disposición unos datos denominados inputs, y unas neuronas de salida en la capa de salida con unos valores denominados outputs.

En el perceptrón multicapa, dispone de unas neuronas ocultas en las capas ocultas, el cometido de estas neuronas es abstraer más información del problema planteado con el fin de mejorar el valor en la predicción. Se denominan ocultas porque sobre estas neuronas no se dispone de ningún poder más allá de conocer sus valores.

Lo más importante de una red neuronal son sus pesos, y éstos se representan mediante matrices.

Funciones Matemáticas

Para el desarrollo de una red neuronal artificial es necesario conocer las funciones matemáticas que le dan forma:

Función de propagación

A diferencia de la capa de entrada, que es el programador quien decide la entrada de sus neuronas, para el resto de neuronas de la red, las entradas se calculan de manera automática. La manera de hacerlo, es calculando el sumatorio de la salida de cada neurona de la capa anterior por los pesos correspondientes a las conexiones entrantes de la neurona que queremos calcular.
Este cálculo, se consigue mediante la llamada *Función de Propagación.*

A continuación se muestra un esquema de dicha función y su expresión:

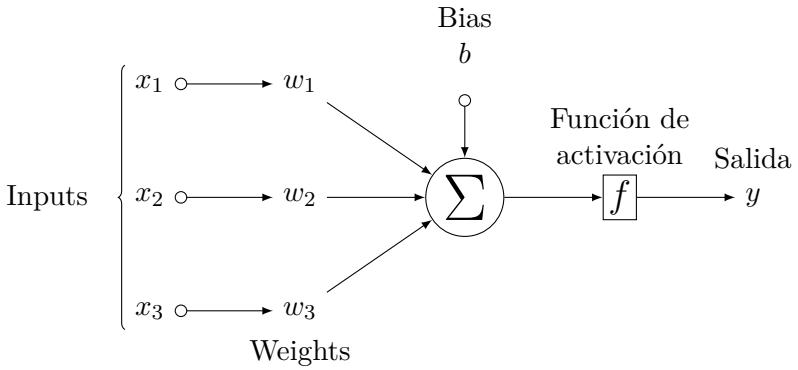


Figura 1.18: Esquema función de propagación

Para el ejemplo de la Figura 1.18, con 3 entradas y, por lo tanto, 3 pesos a cada neurona de la siguiente capa, la manera de calcular la entrada de la neurona correspondiente sería la siguiente:

$$x_1 \cdot w_1 + x_2 \cdot w_2 + x_3 \cdot w_3 = f(y)$$

O lo que es lo mismo:

$$\sum_{i=1}^{3} x_i w_i = f(y)$$

$$\underset{\text{Inputs}}{\begin{pmatrix} 2 & 1 & 4 \end{pmatrix}} \quad \underset{\text{Weights}}{\begin{pmatrix} 0{,}2 & 0{,}3 & 0{,}8 & 0{,}5 \\ 0{,}1 & 0{,}5 & 0{,}1 & 0{,}7 \\ 0{,}3 & 0{,}3 & 0{,}4 & 0{,}8 \end{pmatrix}} \quad \underset{\text{Output}}{\begin{pmatrix} 1{,}7 & 2{,}3 & 3{,}3 & 4{,}9 \end{pmatrix}}$$

Función de activación

Teniendo calculada la salida de la anterior capa mediante la función de propagación, interesa que la entrada a la siguiente capa se acote en un rango para la entrada *(inputs)* de las siguientes neuronas. Esto se consigue haciendo uso de las funciones de activación, ya que éstas transforman la entrada global a un valor de activación para la siguiente capa.
Los rangos que se suelen utilizar están entre 0 y 1, -1 y 1 o 0 y el valor de x de la función.

El valor 0 se utiliza para que una neurona esté inactiva, y los valores -1, 1 o x para que una neurona esté activa en la siguiente capa.

Cada capa de neuronas debe tener una salida acotada hacia la siguiente capa de neuronas o a la salida de la red neuronal. En este caso se han utilizado dos tipos de funciones de activación: para las entradas hacia las siguientes capas, es decir, las capas ocultas, hemos utilizado la función ReLU y para la entrada de la capa de salida hemos utilizado la función Sigmoide.

La función *ReLU* (*función de activación lineal rectificada*) devuelve 0 si x es menor que 0 y para cualquier valor positivo en x devuelve este mismo, como se puede ver en la Figura 1.19.

$$f(x) = max(0, x)$$

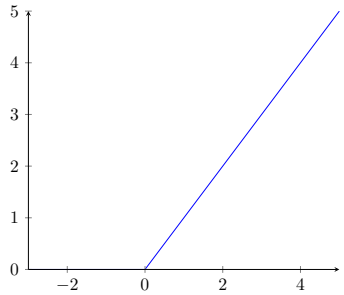


Figura 1.19: Gráfico función de activación ReLU

En la capa de salida hemos utilizado la función Sigmoide que por cada valor en x nos da un resultado acotado entre 0 y 1 como vemos en la Figura 1.20.

$$f(x) = \frac{1}{1 + e^{-x}}$$

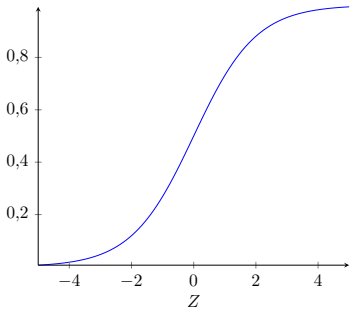


Figura 1.20: Gráfico función de activación Sigmoide

Descenso de gradiente

Una red neuronal hace uso de una función para ajustar los valores de los pesos, una función de la cual desconocemos tanto su forma como sus mínimos y máximos locales y globales. En cambio, si podemos mediante algoritmos de optimización como el *Descenso de Gradiente*, utilizado en este proyecto, llegar a los mínimos locales y a los mínimos globales.

Por lo tanto, el descenso de gradiente nos permite conocer el mínimo valor en cualquier función $f(x)$.

Para obtener dicho valor, calculamos la derivada (pendiente) de $f(x)$ que nos proporciona la rapidez con la que cambia una función, es decir, cuando crece o decrece la función.

Si continuamos en sentido contrario a esta pendiente, es decir en negativo, disminuiremos hasta un punto valor mínimo, de esta manera calculamos el mínimo local de $f(x)$.

Al desconocer la forma de la función, generalmente se determina el mínimo local (punto rojo), existiendo otros mínimos como el global (punto verde), según se muestra en la Figura 1.21.

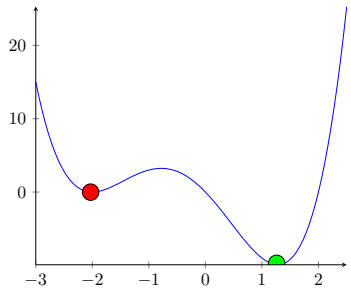


Figura 1.21: Función con mínimos locales y globales

Para calcular los nuevos valores o gradientes de los pesos, se realizan dos pasos:

- *Backpropagation*: En primera instancia mediante el algoritmo de *Backpropagation* o *Retropropagación*, se calcula el valor de la contribución de cada neurona al error total.
- *Autograd*: A continuación, se calcula todas las derivadas del error con respecto a los pesos. De esta manera, al averiguar lo que cambia el error respecto al valor de los pesos, se puede obtener un valor que reduzca este error al mínimo.

PyTorch mediante *Autograd* [Glosario: 2.2] realiza estos cálculos automáticamente. Dichos cálculos son posibles gracias a que*Autograd* construye un grafo computacional de los Tensores [Glosario: 2.2] de forma transparente al programador.

Un *Grafo computacional* es una forma abstracta de representar información. Pueden ser tanto estructuras matemáticas como funciones o algoritmos, o problemas de la vida cotidiana. Se representan gráficamente por medio de unos nodos y vértices que definen la relación entre ellos.

Por ejemplo, dada la siguiente función matemática:

$$f(x) = 2 * \sin(x)$$

Su representación en un *Grafo computacional* se muestra en la Figura 1.22.

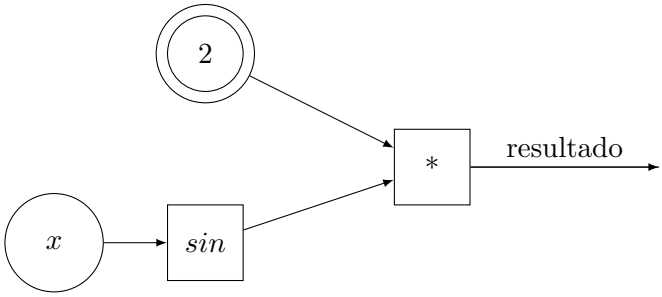


Figura 1.22: Ejemplo grafo computacional

- Los círculos representan valores fijos, datos que no se modifican al realizar el cálculo
- Los círculos dobles representan variables
- Los cuadrados representan operaciones o funciones

Por último, mediante el algoritmo de *Descenso de Gradiente*, se actualizan los pesos con los nuevos Gradientes, resultado de aplicar derivadas sobre el *Grafo computacional*, fruto de la función de *Backpropagation*.

Función de pérdida

Para calcular el error en el valor de la predicción de una red neuronal, se realizan cálculos mediante las funciones de pérdida, las cuáles comparando el valor de salida de la red neuronal (la predicción), y el valor real de dicha predicción (target), se obtiene el error de la salida.

En este proyecto se ha utilizado como función de pérdida el *Error Cuadrático Medio* (MSE)

$$MSE = \frac{1}{N}\sum_{i=1}^{N}(f_i - y_i)^2$$

Donde:

- N es el número total de ejemplos
- f es el valor de la predicción del modelo neuronal
- y valor real del ejemplo N

Calculamos el error al cuadrado para que el error siempre sea positivo, de esta forma sabemos que el error perfecto es 0. Por último, para calcular el promedio, se realiza el sumatorio de todos los errores y se divide entre el número total de datos.

Implementación de nuestra red neuronal

A continuación, se explicará la implementación práctica del código de la red neuronal encargada de controlar la serpiente.

Arquitectura de la red neuronal

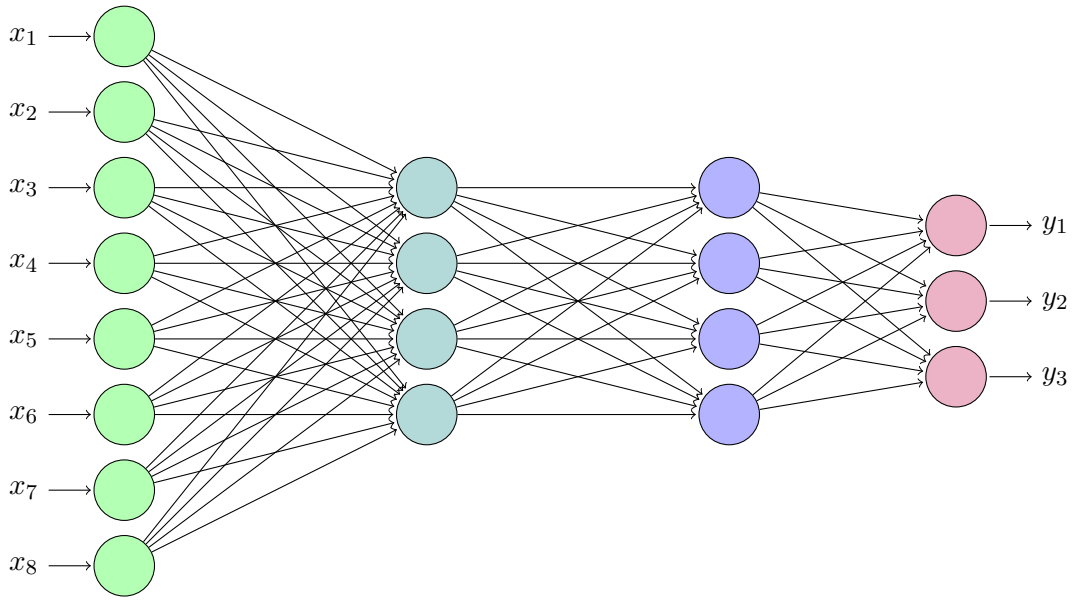


Figura 1.23: Arquitectura de la Red Neuronal

El planteamiento del modelo es el siguiente:

- En la capa de entrada se dispone de 8 neuronas, de las cuáles 4 están destinadas a recibir la información de las distancias de la cabeza de la serpiente a las cuatro paredes en todo momento, y las 4 restantes están destinadas a recibir la información de las distancias de la cabeza de la serpiente a la fruta, siempre y cuando entre en su ángulo de visión.
- En la capa de salida se dispone de 3 neuronas que indican la dirección a la cual se moverá la serpiente: izquierda, derecha, o seguir de frente.

A continuación, se explicarán los métodos y el funcionamiento de las clases que forman el modelo neuronal.

UML Red Neuronal

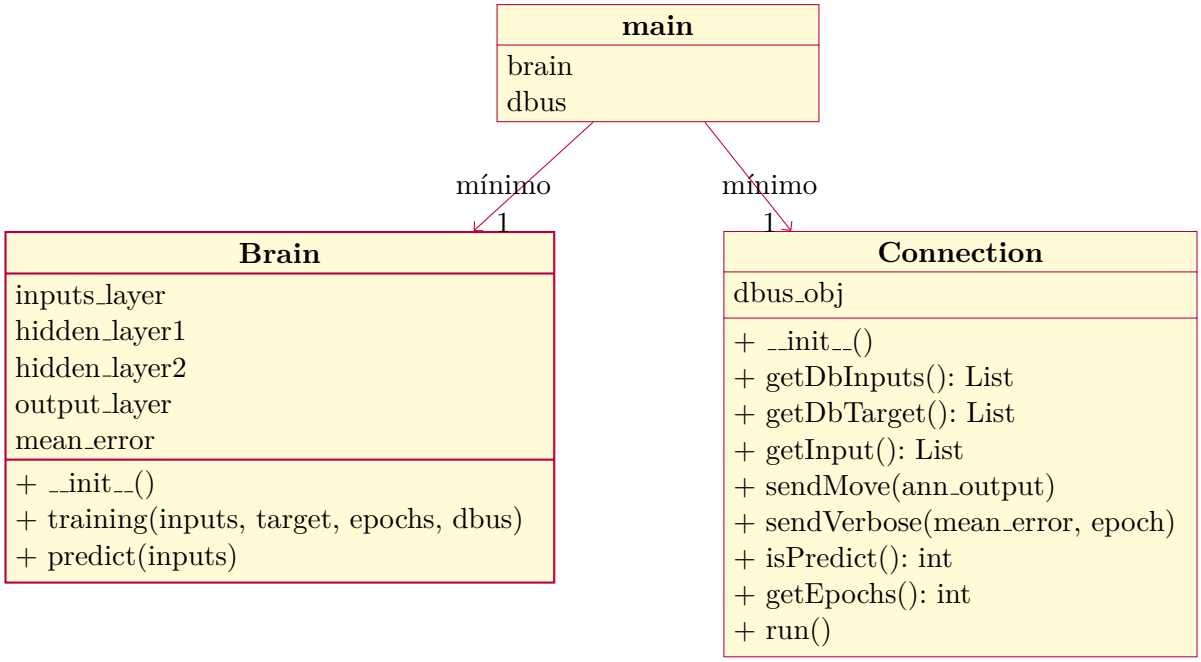


Figura 1.24: UML de la Red Neuronal

Brain

```
1 inputs_layer, hidden_layer1, hidden_layer2, output_layer = 8, 4, 4, 3
2 mean_error = []
```

Fragmento de Código 1.19: Variables

Como variables disponemos del número de neuronas de cada capa y una lista donde almacenaremos la media del error.

```
def __init__(self):
    # Definir arquitectura del modelo neuronal
    self.model = torch.nn.Sequential(
        torch.nn.Linear(self.inputs_layer, self.hidden_layer1, bias=True),
        torch.nn.ReLU(),
        torch.nn.Linear(self.hidden_layer1, self.hidden_layer2, bias=True),
        torch.nn.ReLU(),
        torch.nn.Linear(self.hidden_layer2, self.output_layer, bias=True),
        torch.nn.Sigmoid(),
    )

    # Definir función de pérdida (Error Cuadrático Medio)
    self.loss_function = torch.nn.MSELoss()

    # Definir algoritmo de optimización (Descenso de Gradiente Estocástico)
    self.optimizer = torch.optim.SGD(self.model.parameters(), lr=0.1)
```

Fragmento de Código 1.20: Constructor

En el constructor se declara e inicializa el modelo. Mediante *Sequential* [Glosario: 2.2] y el uso de *Linear* [Glosario: 2.2] aplicamos una transformación lineal, es decir, la función de propagación de dicha capa.

Además, definimos la función de pérdida y el algoritmo de optimización a utilizar.

```
def train(self, inputs, target, epochs, dbus):
    # Convertir datasets (listas) a tensores
    inputs = torch.from_numpy(np.array(inputs)).float()
    target = torch.from_numpy(np.array(target)).float()

    for epoch in range(epochs):
        output = self.model(inputs) # Función de propagación

        error = self.loss_function(output, target) # Cálcula el error

        self.mean_error.append(error.item()) # Guarda el error

        self.optimizer.zero_grad() # Gradientes a cero

        error.backward() # Cálcula gradientes usando Backpropagation

        # Actualizar los pesos mediante el algoritmo de optimización
        self.optimizer.step()

        # Enviar media del error a snake
        dbus.sendVerbose(np.mean(self.mean_error), epoch)
```

Fragmento de Código 1.21: Método de entrenamiento

Mediante este método se realiza la fase de entrenamiento del modelo neuronal.

1. En primera instancia se crean los tensores a partir de los datasets, ya que éstos en un principio son listas.

2. A continuación, se iteran el número total de épocas, en cada una se aplica:

   a) La función de propagación.

   b) Se calcula el error en el valor de la predicción con la función de pérdida, en este caso *MSE*.

   c) Se ponen los gradientes a 0 para evitar posibles interferencias con los anteriores calculados.

   d) Se calculan los nuevos gradientes haciendo uso de *Autograd*.

   e) Se actualizan los pesos mediante el algoritmo de optimización, en este caso Descenso de Gradiente.

   f) Finalmente, se envía el valor del error al proceso del juego por medio de D-Bus.

```
1  def predict(self, inputs):
2      return self.model(torch.from_numpy(np.array(inputs)).float())
3                        .detach()
4                        .numpy()
```

Fragmento de Código 1.22: Método encargado de realizar predicciones

Este método se encarga de realizar predicciones con el modelo neuronal ya entrenado. Recibe los inputs como parámetro y devuelve los outputs correspondientes.

1. Se realiza una llamada a *model*, al cual se le entrega como argumento los inputs convertidos a Tensores `float`.

2. Al resultado se le aplica un `detach` [Glosario: 2.2] para desactivar *Autograd*.

3. Por último, se convierte los Tensores resultado a un array.

Connection

```
1  def __init__(self):
2      bus = dbus.SessionBus()
3      self.dbus_obj = bus.get_object('org.torchlight.Inputs', '/inputs')
```

Fragmento de Código 1.23: Constructor

En el constructor se inicializa un bus de sesión y un objeto D-Bus con la conexión al proceso del juego.

```
def getDbInputs(self):
    inputs_size = self.dbus_obj
                      .get_dbus_method
                          ('getInputsSize',
                            dbus_interface='local.controller.Controller')
                          ()
    inputs = []
    buffer = []

    for index in range(inputs_size):
        input = self.dbus_obj
                    .get_dbus_method
                        ('getAtInputs',
                          dbus_interface='local.controller.Controller')
                        (index)
        buffer.append(input.conjugate())

        if not index % n_inputs:
            inputs.append(buffer.copy())
            buffer.clear()

    return inputs
```

Fragmento de Código 1.24: Método encargado de crear el dataset de inputs

1. Se establece la conexión mediante *D-Bus* al método `getInputsSize` en el proceso de *Snake*.

2. Finalmente, se crea una lista con todos los valores devueltos del método `getAtTarget` del proceso *Snake*.

```
def getDbTarget(self):
    target_size = self.dbus_obj
                      .get_dbus_method
                          ('getTargetSize',
                            dbus_interface='local.controller.Controller')
                          ()
    target = []
    buffer = []

    for index in range(target_size):
        item_target = self.dbus_obj
                          .get_dbus_method
                              ('getAtTarget',
                                dbus_interface='local.controller.Controller')
                              (index)
        buffer.append(item_target.conjugate())

       if not index % n_outputs:
            target.append(buffer.copy())
            buffer.clear()

    return target
```

Fragmento de Código 1.25: Método encargado de crear el dataset Target

Método encargado de crear el dataset de Target.

1. Se establece la conexión mediante D-Bus al método `getTargetSize` en el proceso de *Snake*
2. Por último, se crea una lista con todos los valores devueltos por el método `getAtTarget` del proceso *Snake*

```
def getInput(self):
    # Derecha, izquierda, abajo, arriba
    input = []

    for index in range(4):
        buffer = self.dbus_obj
                      .get_dbus_method
                          ('getAtDistance',
                            dbus_interface='local.controller.Controller')
                          (index)
        input.append(buffer.conjugate())

    return input
```

Fragmento de Código 1.26: Método encargado de trasformar los inputs

Método encargado recibir los inputs del proceso del juego y transformarlos a una lista que pueda manejar el modelo neuronal.

1. Realiza una conexión al método `getAtDistance` del proceso del juego.
2. Finalmente, devuelve una lista con los inputs que serán entregados al modelo neuronal.

```
1  def sendMove(self, ann_output):
2      threshold = self.dbus_obj
3                       .get_dbus_method
4                           ('getThreshold',
5                             dbus_interface='local.controller.Controller')
6                           ()
7
8      if ann_output.item(0) > ann_output.item(1) and
9         ann_output.item(0) > threshold:
10         direction = 1
11     elif ann_output.item(1) > ann_output.item(0) and
12          ann_output.item(1) > threshold:
13         direction = 2
14     else:
15         direction = 3
16
17     self.dbus_obj
18         .get_dbus_method
19              ('newDirection',
20                dbus_interface='local.controller.Controller')
21              (direction)
```

Fragmento de Código 1.27: Método encargado de enviar la predicción al proceso del juego

Método encargado de enviar la predicción del modelo neuronal al proceso del juego.

1. Se realiza una conexión al método `getThreshold` del proceso del juego para recibir el umbral que se aplicará al valor de la predicción
2. Por último, se envía la predicción filtrada por el umbral al método `newDirection` del proceso del juego.

1.3. Conclusiones

Grado de Consecución de los Objetivos

Se ha conseguido el desarrollo del juego Snake con el algoritmo de movimiento que estaba planteado en un principio.

Otro punto logrado del objetivo, es la implementación de una red neuronal capaz de controlar el juego mediante tres posibles movimientos, izquierda, derecha y de frente. El algoritmo de machine learning utilizado para llevarlo a cabo es el *Perceptrón Multicapa* y el tipo de aprendizaje supervisado.

Por último, como comuinicación entre los dos procesos se ha desarrollado D-Bus como sistema IPC.

Uno de los objetivos no alcanzados es cambiar el tipo de aprendizaje de nuestro modelo neuronal a con el método de aprendizaje por refuerzo.

-
- Conseguir que la red neuronal no se encierre en el juego
- Conseguir que la red neuronal complete la máxima puntuación

Análisis del Alcance

Nuestros mayores desafíos sin duda han sido las matemáticas. Dónde el problema arraigó principalmente en el entendimiento del álgebra y su posterior implementación, además de la impetuosa necesidad de aclarar varios conceptos técnicos como *bias*, *umbral* etc y conceptos matemáticos como, multiplicación y transposición de matrices etc

En cuanto al sistema de IPC *D-Bus*, el problema surgió al encontrarnos con una documentación muy extensa a nivel teoría en nuestras mentes habitaba el principal objetivo de aprender inteligencia artificial, llevábamos mucho tiempo escuchando y viendo noticias, inclusive mitos y rumores, por lo que cada vez lo teníamos más claro, queríamos aprender cómo funciona aquello que todo el mundo teme. A día de hoy, podemos afirmar con creces que nuestro objetivo ha sido cumplido, durante este tiempo hemos sido capaces de allanar el camino para avanzar con mayor rapidez,co y escasa a nivel práctico, lo que conllevó a la realización de numerosas pruebas prácticas y el uso de herramientas

Por último, en cuanto al juego snake. El mayor problema planteado ha sido su movimiento, al decidir implementar un sistema de movimiento circular y no fijo, surgió conflicto con las posiciones de la cabeza y la cola en el array de vectores.

Aprendizajes

El día que se presentó la idea de proyecto, en nuestras mentes habitaba el principal objetivo de aprender inteligencia artificial.

Para llevarla a cabo nos hemos visto obligados a ampliar nuestros conocimientos en matemáticas y lógica, necesarios para entender fundamentos prácticos de las redes neuronales, poniendo el foco en el deep learning.

A medida que el proyecto avanzaba, nos hemos enfrentado a diferentes algoritmos de optimización para mejorar el desempeño de la red, cálculos matemáticos para afianzar conocimientos y librerías a más alto nivel.

Nos hemos dado cuenta que en un proyecto de mayor o menor magnitud, el rigor en la organización es fundamental.

Reflexión/Posibles mejoras

A día de hoy, el Machine Learning define lo que leemos, lo que vemos, lo que opinamos, incluso lo que necesitamos. En definitiva, nos conoce mejor de lo que nos podríamos imaginar.

Más allá de la ética del uso de estas prácticas, la inteligencia artificial ha llegado para quedarse, para cambiar el mundo tal y como lo conocemos. Es una tecnología que nos brinda la oportunidad predecir errores, errores de los cuales no sabemos su existencia. De nosotros como especie y sociedad será la obligación de corregirlos y dejar un mundo mejor para las generaciones venideras.

El futuro es incierto, pero podemos decir que la inteligencia artificial convivirá con nosotros, mejorando nuestras vidas y dándonos la oportunidad de seguir avanzando.

Nos aventuramos a decir que la inteligencia artificial y la robótica son el siguiente paso evolutivo de la humanidad.

Torchlight abre las puertas a este inmenso mundo, del cual animamos a navegar y aprender, para así seguir descubriendo el funcionamiento del mundo que nos rodea.

Mejoras: Snake modular Aprendizaje supervisado y posterior implementación de la neuroevolucion Dbus con trasferencia de vectores

Capítulo 2

Anexos

2.1. Diagramas UML

Arquitectura del Juego

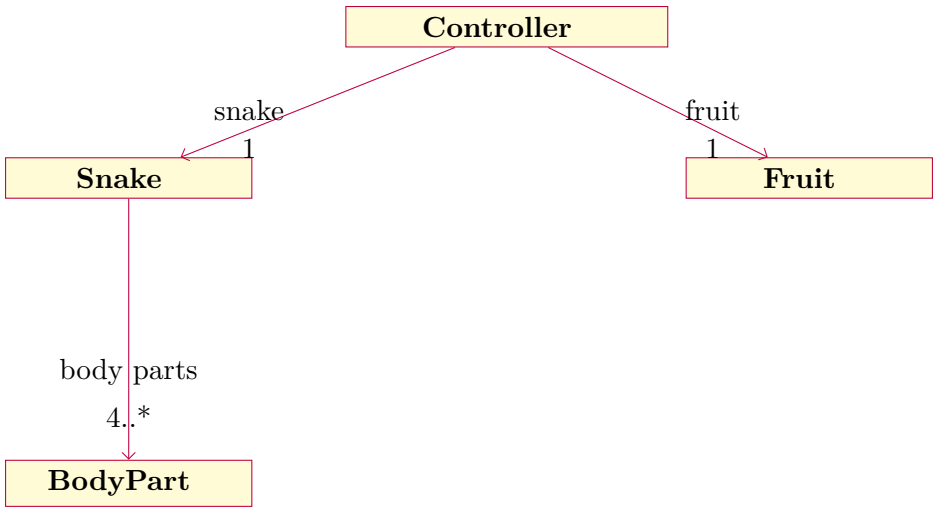


Figura 2.1: Arquitectura del juego

UML Juego

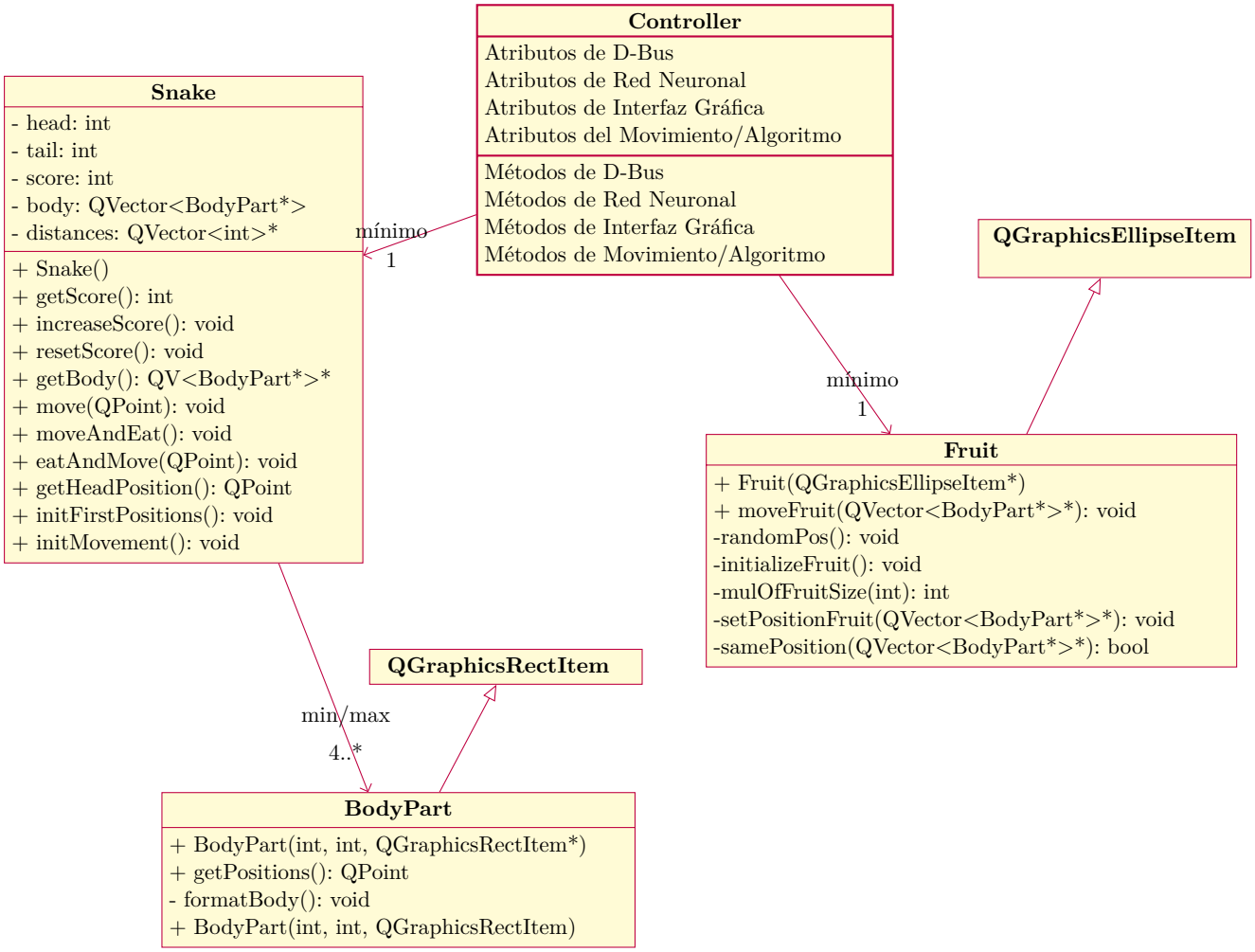


Figura 2.2: UML de la arquitectura del juego

UML Red Neuronal

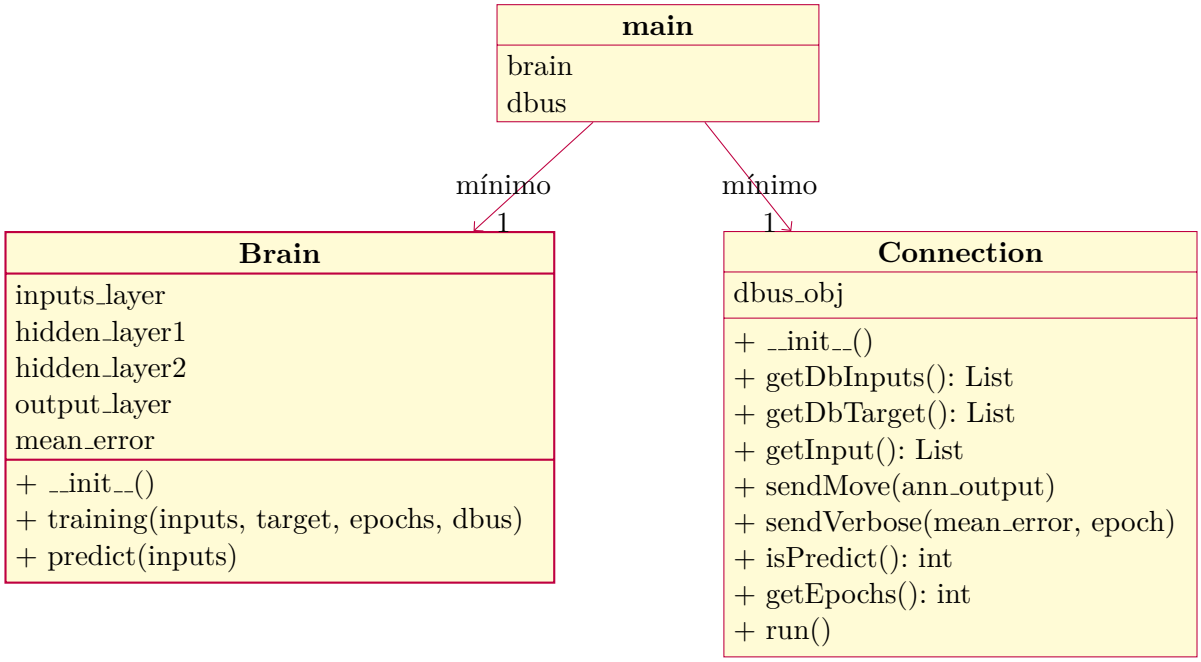


Figura 2.3: UML de la Red Neuronal

2.2. Código

Clase `main`

```
1  #include "controller.h"
2
3  #include <QApplication>
4  #include <QStyle>
5  #include <QDesktopWidget>
6
7  int main(int argc, char *argv[])
8  {
9      QApplication a(argc, argv);
10     Controller w;
11
12     w.show();
13     return a.exec();
14 }
```

Fragmento de Código 2.1: Método main

Cabecera `common`

```
1  //#define __DEBUG__
2  //#define __DEBUG_SNAKE__
3  //#define __DEBUG_FRUIT__
4
```

```
5  #define BODY_SIZE  25
6  #define FRUIT_SIZE BODY_SIZE
7  #define ADVANCE    BODY_SIZE
8
9  #define SCENE_WIDTH  800
10 #define SCENE_HEIGHT 475
11
12 #define TIMER_SPEED 100
13
14
15 /* COLORS */
16 #define SNAKE_COLOR       0x60c18a
17 #define FRUIT_COLOR       0xf76c6f
18 #define BOARD_COLOR       0xefebe1
19 #define BOARD_LINES_COLOR 0xc6a365
20
21
22 #define DBUS_INPUTS  "org.torchlight.Inputs"
23 #define DBUS_PATH    "/inputs"
24 #define THRESHOLD    0.53
25 #define TRAIN_EPOCHS 10000
26
27 enum TDirection {
28     RIGHT,
29     LEFT,
30     UP,
31     DOWN,
32     TOTAL
33 };
```

Fragmento de Código 2.2: Datos comunes entre las clases

Clase `Controller`

```
1  /**
2   * @brief Controller::Controller constructor del controlador,
3   * inicializa los gráficos, las direcciones, los vectores de
4   * inputs y target para la red neuronal y crea el DBus.
5   * @param parent
6   */
7  Controller::Controller(QWidget *parent) :
8                              QMainWindow(parent) ,
9                              ui(new Ui::Controller)
10 {
11     ui->setupUi(this);
12
13     /* Centrar Ventana */
14     move(QGuiApplication::screens().at(0)->
15                                geometry().center() -
16                                frameGeometry().center());
17
```

```
18      initializeGraphics();
19      setDirections();
20
21      /* Vectores para entrenar la red */
22      inputs    = new QVector<int>();
23      targetKey = new QVector<char>();
24
25      createDBus();
26  }
```

Fragmento de Código 2.3: Constructor de la clase Controller

```
1   /**
2    * @brief Controller::drawBackground, dibuja las lineas
3    * verticales y horizontales en el QGraphicsScene.
4    */
5   void
6   Controller::drawBackground()
7   {
8       /* Lineas Horizontales */
9       for(int j=0; j<SCENE_HEIGHT; j+=BODY_SIZE)
10          graphicScene->addLine(0,
11                                j,
12                                SCENE_WIDTH,
13                                j,
14                                QPen(BOARD_LINES_COLOR));
15
16      /* Lineas Verticales */
17      for(int j=0; j<SCENE_WIDTH; j+=BODY_SIZE)
18          graphicScene->addLine(j,
19                                0,
20                                j,
21                                SCENE_HEIGHT,
22                                QPen(BOARD_LINES_COLOR));
23  }
```

Fragmento de Código 2.4: Dibuja las lineas horizontales y verticales

```
1   /**
2    * @brief Controller::initializeMovement en función de donde se genera
3    * la serpiente se configura una dirección u otra.
4    */
5   void
6   Controller::initializeMovement()
7   {
8       /* Posiciones dentro del tablero */
9       bool upLeft, upRight, downLeft, downRight;
10      /*
11        +----------+----------+
12        |          |          |   El tablero del juego.
```

```
        |  upLeft   |  upRight  |
        |           |           |
        +-----------+-----------+
        |           |           |
        | downLeft  | downRight |
        |           |           |
        +-----------+-----------+
       */
    snake->initMovement(&upLeft, &upRight, &downLeft, &downRight);

    if (upLeft)
        setMoveDirection(false, false, false, true, DOWN);

    else if (upRight)
        setMoveDirection(true, false, false, false, LEFT);


    else if (downLeft)
        setMoveDirection(false, true, false, false, UP);

    else if (downRight)
        setMoveDirection(true, false, false, false, LEFT);
}
```

Fragmento de Código 2.5: Cambia la dirección inicial del movimiento de la serpiente en función de la posición inicial de la serpiente

```
/**
 * @brief Controller::initializeGraphics, inicializa los elementos
 * gráficos del QGrpahicsScene y de QGraphicsView.
 */
void
Controller::initializeGraphics()
{
    ui->graphicsView->setMaximumSize(SCENE_WIDTH, SCENE_HEIGHT);
    ui->graphicsView->setMinimumSize(SCENE_WIDTH, SCENE_HEIGHT);
    ui->graphicsView->setBackgroundBrush(QBrush(BOARD_COLOR));
    ui->graphicsView->setCursor(Qt::BlankCursor);

    graphicScene = new QGraphicsScene(this);  // Manejador de items 2D.
    graphicScene->setSceneRect(0, 0, SCENE_WIDTH, SCENE_HEIGHT);

    // Añadimos el Scene al Widget de Gráficos del UI.
    ui->graphicsView->setScene(graphicScene);
    // Deshabilita el arrastre con el ratón
    ui->graphicsView->setDragMode(QGraphicsView::NoDrag);
    drawBackground();
}
```

Fragmento de Código 2.6: Inicializa los elementos gráficos

```
/**
 * @brief Controller::addObject, añade cualquier
 * QGraphicsItem al QGraphicsScene.
 * @param el QGraphicsItem para añadir a la
 * escena gráfica.
 */
void
Controller::addObject(QGraphicsItem *object)
{
    graphicScene->addItem(object);
}
```

Fragmento de Código 2.7: Añade un QGraphicsItem al QGraphicsScene

```
/**
 * @brief Controller::setDirections Configura las
 * direcciones.
 */
void
Controller::setDirections()
{
    directions[0].setX(1),  directions[0].setY(0);  // Derecha
    directions[1].setX(-1), directions[1].setY(0);  // Izquierda
    directions[2].setX(0),  directions[2].setY(-1); // Arriba
    directions[3].setX(0),  directions[3].setY(1);  // Abajo
}
```

Fragmento de Código 2.8: Configura las direcciones

```
/**
 * @brief Controller::updateSnake Muestra en la interfaz
 * gráfica la posición de la serpiente, de la fruta y
 * comprueba se se ha chocado con la pared/fruta y mueve
 * la serpiente.
 */
void
Controller::updateSnake()
{
    debugPosition();
    debugFruit();
    snake->calcWallDistance();
    insertedInput = true;

    /* Comparaciones para la creación del target */
    if (!keyPressed)
        addFrontTarget();

    if (keyPressed)
```

```
20          keyPressed = false;
21
22      /* Si choca con la fruta */
23      if ( snakeCollapseFruit() ) {
24          snake->eatAndMove(direction);
25          addObject(snake->getBody()->at(snake->getHead()));
26          changeScoreValue();
27          fruit->moveFruit(snake->getBody());
28
29      } else {
30          snake->move(direction);
31      }
32
33      lastDistance = snake->getLastDistance();
34
35      /* Si se choca con ella misma o la pared */
36      if ( cannibalism() || hintWall() ) {
37          resetGame();
38      }
39  }
```

Fragmento de Código 2.9: Método que ejecuta el Timer y actualiza la serpiente

```
1   /**
2    * @brief Controller::addTurnTarget Añade un nuevo dato
3    * al target si se ha pulsado una de las teclas 'A' o 'D'.
4    * @param key: La tecla pulsada.
5    */
6   void
7   Controller::addTurnTarget(int key)
8   {
9       if ( runningGame && (key == Qt::Key_A || key == Qt::Key_D) ) {
10          keyPressed = true;
11
12          if (key == Qt::Key_A) {
13              targetKey->append(1);
14              targetKey->append(0);
15              targetKey->append(0);
16          }
17
18          else {
19              targetKey->append(0);
20              targetKey->append(1);
21              targetKey->append(0);
22          }
23      }
24  }
```

Fragmento de Código 2.10: Añade al vector de target si se ha girado la serpiente

```
/**
 * @brief Controller::addFrontTarget añade un nuevo dato al
 * vector 'target' si no se ha pulsado la 'A' o 'D'.
 */
void
Controller::addFrontTarget()
{
    if (runningGame) {
        targetKey->append(0);
        targetKey->append(0);
        targetKey->append(1);
    }
}
```

Fragmento de Código 2.11: Añade al vector de target la dirección de frente

```
/**
 * @brief Controller::setMoveDirection cambia la dirección
 * actual de la serpiente
 * @param left
 * @param up
 * @param right
 * @param down
 * @param moveDirection
 */
void
Controller::setMoveDirection(bool left,
                             bool up,
                             bool right,
                             bool down,
                             const int moveDirection)
{
    leftDirection  = left;
    upDirection    = up;
    rightDirection = right;
    downDirection  = down;
    direction = directions[moveDirection];
}
```

Fragmento de Código 2.12: Modifica los atributos de la dirección actual de la serpiente

```
/**
 * @brief Controller::snakeCollapseFruit verifica si la serpiente
 * se ha chocado con la fruta.
 * Si se ha chocado incrementa el atributo 'score' y retorna true,
 * si no retorna false
 * @return bool
 */
```

```
8  bool
9  Controller::snakeCollapseFruit()
10 {
11     QPoint fruitPos  = fruit->pos().toPoint();
12     QPoint snakeHead = snake->getHeadPosition();
13
14     if (snakeHead == fruitPos)
15         return true;
16
17     else return false;
18 }
```

Fragmento de Código 2.13: Retorna verdadero o falso si la serpiente ha chocado o no con la fruta

```
1  /**
2   * @brief Controller::hintWall comprueba si la cabeza de la
3   * serpiente se ha chocado contra la pared y devuelve true si
4   * se ha chocado, si no, false.
5   * @return bool
6   */
7  bool
8  Controller::hintWall()
9  {
10     int headX = snake->getHeadPosition().x();
11     int headY = snake->getHeadPosition().y();
12
13     if( (headX >= SCENE_WIDTH) || (headX < 0) )
14         return true;
15
16     else if ((headY >= SCENE_HEIGHT) || (headY < 0 ))
17         return true;
18
19     else return false;
20
21 }
```

Fragmento de Código 2.14: Retorna verdadero o falso si la serpiente ha chocado con alguna pared

```
1  /**
2   * @brief Controller::bodyToScene añade al QGraphicsScene cada
3   * elemento del cuerpo.
4   */
5  void
6  Controller::bodyToScene()
7  {
8      for(int i=0; i<snakePositions->size(); i++)
9          graphicScene->addItem(snakePositions->at(i));
10 }
```

Fragmento de Código 2.15: Añade el cuerpo de la serpiente al QGrpahicsScene

```
/**
 * @brief Controller::cannibalism verifica si la cabeza se ha chocado
 * con alguna parte del cuerpo, si es así retorna true, si no retorna
 * false.
 * @return bool
 */
bool
Controller::cannibalism()
{
    QPoint headPosition = snake->getBody()
                            ->at(snake->getHead())
                            ->getPositions();

    for (int i=0; i<snake->getSize(); i++)
        if (i != snake->getHead())
            if (headPosition.x() == snake->getBody()->at(i)->x() &&
                    headPosition.y() == snake->getBody()->at(i)->y() )
                return true;
    return false;
}
```

Fragmento de Código 2.16: Retorna verdadero o falso si la serpiente choca con alguna parte de su cuerpo

```
/**
 * @brief Controller::gameOver llama al método resetGame()
 */
void
Controller::gameOver()
{
    resetGame();
}
```

Fragmento de Código 2.17: Reinicia el juego

```
/**
 * @brief Controller::runGame comprueba si el juego está parado
 * y si es asi lo inicia, iniciando el timer.
 */
void
Controller::runGame() {
    if (!runningGame){
        initializeGameObjects();
        timer->start(TIMER_SPEED);
        runningGame = true;
        predict = 0;
    }
}
```

Fragmento de Código 2.18: Inicia el juego

```
/**
 * @brief Controller::initializeGameObjects inicializa los objetos
 * snake, fruit y timer.
 * Configura el método que ejecutará el timer.
 * Añade la fruta y la serpiente al QGraphicsScene
 */
void
Controller::initializeGameObjects()
{
    snake     = new Snake();
    fruit     = new Fruit();
    timer     = new QTimer(this);

    runningGame = false;
    keyPressed  = false;

    /* Cuando el timer mande la señal timeout, lo conecto con el metodo
    *  de esta misma clase, llamado updateSnake()
    */
    connect(timer, SIGNAL(timeout()), this, SLOT(updateSnake()));

    snake->initFirstPositions();

    /* Guardamos el cuerpo de la serpiente para poder
       modificar sus datos en controller.cpp
    */
    snakePositions = snake->getBody();
    fruit->moveFruit(snakePositions);

    addObject(fruit);
    bodyToScene();

    /* Si no lanzamos el evento, la serpiente muere nada más empezar
     * a no ser que nosotros pulsemos una tecla
```

```
35          */
36          initializeMovement();
37      }
```

Fragmento de Código 2.19: Inicializa los objetos del juego

```
/**
 * @brief Controller::keyPressEvent se ejecuta cuando se lanza
 * un evento de QKeyEvent y captura la tecla pulsada.
 * Si la tecla pulsada es A o D cambia la dirección de la serpiente
 * basandose en la dirección actual de la misma.
 * @param QKeyEvent event
 */
void
Controller::keyPressEvent(QKeyEvent *event)
{
    int key = event->key();

    if (!event->isAutoRepeat()) {
        /* Movimientos actuales */
        if (leftDirection) {     // Se mueve hacia la Izquierda.
            if  (key == Qt::Key_A)
                setMoveDirection(false, false, false, true, DOWN);

            else if (key == Qt::Key_D)
                setMoveDirection(false, true, false, false, UP);
        }

        else if (upDirection) {  // Se mueve hacia Arriba.
            if  (key == Qt::Key_A)
                setMoveDirection(true, false, false, false, LEFT);

            else if (key == Qt::Key_D)
                setMoveDirection(false, false, true, false, RIGHT);
        }

        else if (rightDirection) { // Se mueve hacia la Derecha.
            if  (key == Qt::Key_A)
                setMoveDirection(false, true, false, false, UP);

            else if (key == Qt::Key_D)
                setMoveDirection(false, false, false, true, DOWN);
        }

        else if (downDirection) { // Se mueve hacia Abajo.
            if  (key== Qt::Key_A)
                setMoveDirection(false, false, true, false, RIGHT);

            else if (key == Qt::Key_D)
                setMoveDirection(true, false, false, false, LEFT);
        }
        if (insertedInput) {
            addTurnTarget(key);
            insertedInput = false;
        }
    }
}
```

Fragmento de Código 2.20: Método que se ejecuta al pulsar una tecla

```
/**
 * @brief Controller::resetGame si el juego está iniciado para el timer,
 * elimina los objetos snake, fruit y timer, limpia el tablero, dibuja el
 * tablero y cambia labels del ui gráfico.
 */
void
Controller::resetGame()
{
    if (runningGame){
        getInputs();
        timer->stop();

        delete snake;
        delete fruit;
        delete timer;

        graphicScene->clear();
        drawBackground();

        ui->lblSnakeX->setText("SNAKE X: 0");
        ui->lblSnakeY->setText("SNAKE Y: 0");
        ui->lblFruitX->setText("FRUIT X: 0");
        ui->lblFruitY->setText("FRUIT Y: 0");
        ui->scoreLabelNum->setText("0");

        runningGame = false;
    }
#ifdef __DEBUG__
    qDebug("AUX->X = %f\nAUX->Y = %f\n",aux->x(), aux->y() );

    int inpt = inputs->length();
    int tar  = targetKey->length();

    qDebug("Target: %i - Regla: %i ",tar, (3 * inpt) / 4);

    for (int i=0; i<inputs->length(); i++)
        if(!inputs->at(i))
            qDebug("Hay un cero");
    qDebug("Position 0: %i", inputs->at(0));
#endif
}
```

Fragmento de Código 2.21: Reinicia el juego

```
/**
 * @brief Controller::runBrain ejecuta la red neuronal en
 * un proceso distinto al actual.
 */
```

```
5  void
6  Controller::runBrain()
7  {
8      QString program = "python3";
9      QStringList arguments;
10     arguments << "../snake_brain/main.py";
11
12     QProcess *myProcess = new QProcess(this);
13     myProcess->start(program, arguments);
14 }
```

Fragmento de Código 2.22: Ejecuta la red neuronal

Métodos de los botones

```
1  /**
2   * @brief Controller::on_btnInit_clicked botón de inicio
3   * que llama al método runGame().
4   */
5  void
6  Controller::on_btnInit_clicked()
7  {
8      runGame();
9  }
```

Fragmento de Código 2.23: Método botón `Play`

```
1  /**
2   * @brief Controller::on_btnReset_clicked botón de reinicio
3   * del juego, llama al método resetGame().
4   */
5  void
6  Controller::on_btnReset_clicked()
7  {
8      resetGame();
9  }
```

Fragmento de Código 2.24: Método botón `Reset`

```
1  /**
2   * @brief Controller::on_btnTrain_clicked botón de entrenamiento
3   * que resetea el juego y llama al método runBrain().
4   */
5  void
6  Controller::on_btnTrain_clicked()
7  {
8      resetGame();
9      runBrain();
10 }
```

Fragmento de Código 2.25: Método botón `Train`

```
/**
 * @brief Controller::on_btnPlayIA_clicked botón de Play IA que configura
 * el flag de predicciones y ejecuta el método runBrain().
 */
void
Controller::on_btnPlayIA_clicked()
{
    predict = 1;
    runBrain();
}
```

Fragmento de Código 2.26: Método botón `Play IA`

```
/**
 * @brief Controller::on_btnExit_clicked botón salir, finaliza
 * el programa.
 */
void
Controller::on_btnExit_clicked()
{
    delete inputs;
    delete targetKey;
    exit(0);
}
```

Fragmento de Código 2.27: Método botón `Exit`

Métodos utilizados en D-Bus

```
/**
 * @brief Controller::createDBus crea un bus en D-Bus con el
 * nombre de la constante DBUS_INPUTS y registra este objeto (Controller)
 * con los métodos declarados como public slots dentro de la ruta DBUS_PATH.
 */
void
Controller::createDBus()
{
    QDBusConnection::sessionBus().registerService(DBUS_INPUTS);
    QDBusConnection::sessionBus().
                            registerObject(DBUS_PATH,
                                           this,
                                           QDBusConnection::ExportAllSlots);
}
```

Fragmento de Código 2.28: Creación de un bus en Qt

```
/**
 * @brief Controller::getTargetSize retorna el tamaño del vector
 * de targets para que la red neuronal sepa el tamaño y pueda iterar
 * según su tamaño.
 * @return tamaño del vector targetKey.
 */
int
Controller::getTargetSize()
{
#ifdef __DEBUG__
    qDebug("TARGE SIZE: %i", targetKey->length());
#endif
    return targetKey->length();
}
```

Fragmento de Código 2.29: Devuelve el tamaño del vector que contiene los target

```
/**
 * @brief Controller::getAtTarget retorna el dato concreto de una
 * posición dentro del vector.
 * @param position, la posición a retornar.
 * @return el dato con indice position.
 */
int
Controller::getAtTarget(int position)
{
    return targetKey->at(position);
}
```

Fragmento de Código 2.30: Devuelve un target en concreto del array

```
/**
 * @brief Controller::getInputsSize retorna el tamaño del vector
 * de inputs para que la red neuronal conozca el tamaño y lo pueda
 * iterar.
 * @return tamaño del vector inputs
 */
int
Controller::getInputsSize()
{
    return inputs->length();
}
```

Fragmento de Código 2.31: Devuelve el tamaño del vector de datos de entrada

```
/**
 * @brief Controller::getAtInputs devuelve el dato con indice position
 * dentro del vector de inputs.
```

```
 * @param position, indice del vector.
 * @return dato dentro del vector inputs con indice position.
 */
int
Controller::getAtInputs(int position)
{
    return inputs->at(position);
}
```

Fragmento de Código 2.32: Devuelve un un dato de una posición concreta de los datos de entrada

```
/**
 * @brief Controller::getEpochs obtiene el número introducido en el
 * QTextEdit del GUI y lo retorna, para que la red entrene según las
 * epocas que se han introducido.
 * Si no se introduce un número o algo distinto de un número, por defecto
 * son TRAIN_EPOCHS epocas de entrenamiento.
 * @return el total de epocas
 */
int
Controller::getEpochs()
{
    int inputEpochs;
    QString epochs (ui->inputEpoch->toPlainText());
    inputEpochs = epochs.toInt();

    if (!inputEpochs)
        inputEpochs = TRAIN_EPOCHS;

    return inputEpochs;
}
```

Fragmento de Código 2.33: Configura las épocas de aprendizaje, por defecto 10.000

```
/**
 * @brief Controller::setLabelError añade la interfaz gráfica
 * el error medio de la epoca actual de entrenamiento y las
 * epocas que lleva la red neuronal entrenadas.
 * @param err
 * @param epoch
 */
void
Controller::setLabelError(double err, int epoch)
{
    ui->lblMeanError->setText(QString::number(err));
    ui->lblActEpoch->setText(QString::number(epoch));
}
```

Fragmento de Código 2.34: Añade al UI el error de la época y la época actual

```
/**
 * @brief Controller::getThreshold retorna el valor de THRESHOLD para que
 * los datos de salida de la red estén por debajo del THRESHOLD.
 * @return THRESHOLD
 */
int
Controller::getThreshold()
{
    return THRESHOLD;
}
```

Fragmento de Código 2.35: Retorna el umbral para la capa de salida

```
/**
 * @brief Controller::isPredict retorna el flag 'predict' para que la
 * red sepa si entrenar o predecir resultados sin entrenar.
 * @return predict
 */
int
Controller::isPredict()
{
    return predict;
}
```

Fragmento de Código 2.36: Método para que la red sepa si entrenar o predecir

```
/**
 * @brief Controller::newDirection lanza un evento de tecla pulsada para
 * cambiar la dirección actual de la serpiente según la predicción de la
 * red neuronal.
 * @param direction
 */
void
Controller::newDirection(int direction)
{
    /* 1 - Izquierda
     * 2 - Derecha
     * 3 - Frente;    No se comprueba, siempre va adelante.
     */
    QKeyEvent *e;
    setActualDirection(direction);

    if (direction == 1)
        e = new QKeyEvent(QEvent::KeyPress,Qt::Key_A,Qt::NoModifier,"a");

    else if (direction == 2)
        e = new QKeyEvent(QEvent::KeyPress,Qt::Key_D,Qt::NoModifier,"d");

```

```
23      if (direction == 1 || direction == 2)
24          QApplication::sendEvent(this, e);
25      delete e;
26  }
```

Fragmento de Código 2.37: Cambia la dirección de la serpiente

Clase Snake

```
1   /**
2    * @brief Snake::Snake constructor, inicia el score a 0 y
3    * las posiciones de head y tail dentro del vector del cuerpo.
4    */
5   Snake::Snake()
6   {
7       score = 0;
8       // Índice del vector correspondiente a la cabeza del snake
9       head  = SNAKE_INIT_LENGHT - 1;
10      // Índice del vector correspondiente a la cola del snake
11      tail  = 0;
12      distances = new QVector<int>();
13  }
```

Fragmento de Código 2.38: Constructor de Snake

```
1   /**
2    * @brief Snake::initFirstPositions añade tantas BodyParts como
3    * el valor de la constante SNAKE_INIT_LENGHT, y ejecuta el método
4    * randomPosition().
5    */
6   void
7   Snake::initFirstPositions()
8   {
9       for (int i=0; i<SNAKE_INIT_LENGHT; i++)
10          body.push_back(new BodyPart(0, 0));
11
12      randomPosition();
13  }
```

Fragmento de Código 2.39: Inicializa las primeras posiciones

```
1   /**
2    * @brief Snake::randomPosition cambia la posición de la serpiente.
3    */
4   void
5   Snake::randomPosition()
6   {
7       timespec timeSeed;
8       QRandomGenerator *rand;
```

```
9       int snakeX, snakeY;
10
11      timespec_get(&timeSeed, TIME_UTC);
12      rand = new QRandomGenerator(timeSeed.tv_nsec);
13
14      snakeX = (int) rand->bounded(SCENE_WIDTH  - BODY_SIZE);
15      snakeY = (int) rand->bounded(SCENE_HEIGHT - BODY_SIZE);
16
17      for(int i=0; i<SNAKE_INIT_LENGHT; i++)
18      {
19          body.at(i)->setX(mulOfBodySize(snakeX)+BODY_SIZE);
20          body.at(i)->setY(mulOfBodySize(snakeY));
21      }
22  }
```

Fragmento de Código 2.40: Posiciona la serpiente de forma aleatoria

```
1   /**
2    * @brief Snake::initMovement Inicializa el movimiento de la serpiente
3    * dependiendo de la zona donde se haya generado la cabeza.
4    * @param upLeft
5    * @param upRight
6    * @param downLeft
7    * @param downRight
8    */
9   void
10  Snake::initMovement(bool *upLeft,
11                      bool *upRight,
12                      bool *downLeft,
13                      bool *downRight )
14  {
15      int headX, headY;
16      headX = body.at(head)->x();
17      headY = body.at(head)->y();
18
19      // La mitad izquierda del graphicsview
20      if ( headX < (SCENE_WIDTH / 2) )
21      {
22          if (headY <(SCENE_HEIGHT / 2)) // La mitad supierior de la izquierda
23              *upLeft = true, *downRight = *upRight = *downLeft = false;
24          else                           // La mitad inferior de la izquierda
25              *downLeft = true, *upLeft = *upRight = *downRight = false;
26      }
27      else // La mitad derecha del graphicsview
28      {
29          if (headY <(SCENE_HEIGHT / 2)) // La mitad superior de la derecha
30              *upLeft = true, *downRight = *upRight = *downLeft = false;
31          else                           // La mitad inferior de la derecha
32              *downRight = true, *upLeft = *upRight = *downLeft = false;
```

```
33      }
34  }
```

Fragmento de Código 2.41: Inicializa la dirección de la serpiente en función de su posición en el tablero

```
1   /**
2    * @brief Snake::move actualiza el valor de la cola a la posición
3    * de la nueva cabeza.
4    * Incrementa la cabeza actual y dirección recibida la almacena
5    * en la cola.
6    * @param direction
7    */
8   void
9   Snake::move(QPoint direction)
10  {
11      QPoint headPosition = body.at(head)->getPositions();  // Comprobado
12
13      // Cola = Posición cabeza + Dirección
14      body.at(tail)->setX( headPosition.x() + direction.x()*ADVANCE );
15      body.at(tail)->setY( headPosition.y() + direction.y()*ADVANCE );
16
17      head = tail;  // Cabeza = Antigua cola
18      tail++;
19
20      if ( head == body.size() ) head = 0;
21      if ( tail == body.size() ) tail = 0;
22
23  #ifdef __DEBUG_SNAKE__
24      qDebug("\nHead: %i\nTail: %i", head, tail);
25      for (int i=0; i<body.size(); i++)
26          qDebug("snake.at(%i) = [%f, %f]", i,
27                                            body.at(i)->scenePos().x(),
28                                            body.at(i)->scenePos().y());
29  #endif
30  }
```

Fragmento de Código 2.42: Mueve la serpiente

```
1   /**
2    * @brief Snake::eatAndMove incrementa el tamaño de snake
3    * insertando una nueva parte del cuerpo al vector body
4    * @param direction
5    */
6   void
7   Snake::eatAndMove(QPoint direction)
8   {
9       BodyPart *aux = new BodyPart(0, 0);
10
11      int x = body.at(head)->x() + direction.x()*BODY_SIZE;
```

```
12      int y = body.at(head)->y() + direction.y()*BODY_SIZE;
13
14      aux->setPos(x, y);
15
16  #ifdef __DEBUG_SNAKE__
17      qDebug("AUX->X = %f\nAUX->Y = %f\n",aux->x(), aux->y() );
18  #endif
19
20      body.insert(body.begin() + (head+1), aux);
21      head++;
22      tail = head+1;
23
24      if ( head == body.size() ) head = 0;
25      if ( tail == body.size() ) tail = 0;
26
27      increaseScore();
28
29  #ifdef __DEBUG_SNAKE__
30      qDebug("\nComido");
31      qDebug("\nHead: %i\nTail: %i", head, tail);
32      for (int i=0; i<body.size(); i++)
33          qDebug("snake.at(%i) = [%f, %f]", i,
34                                              body.at(i)->x(),
35                                              body.at(i)->y());
36  #endif
37  }
38  }
```

Fragmento de Código 2.43: Come una fruta y mueve la serpiente

```
1   /**
2    * @brief Snake::calcWallDistance calcula el tamaño de la
3    * cabeza de la serpiente hacia las 4 paredes y lo añade
4    * al vector 'distances'
5    */
6   void
7   Snake::calcWallDistance()
8   {
9       /*DERECHA, IZQUIERDA, ABAJO, ARRIBA*/
10      QPoint snakeHead = getHeadPosition();
11
12      // Distancia hacia la derecha
13      distances->append((SCENE_WIDTH - snakeHead.x()) / BODY_SIZE);
14      // Distancia hacia la izquierda
15      distances->append((snakeHead.x() + BODY_SIZE) / BODY_SIZE);
16
17      // Distancia hacia abajo
18      distances->append((SCENE_HEIGHT - snakeHead.y()) / BODY_SIZE);
19      // Distancia hacia arriba
```

```
20      distances->append((snakeHead.y() + BODY_SIZE) / BODY_SIZE);
21  }
```

Fragmento de Código 2.44: Añade las distancias hacia la pared en vector de distancias

```
1   /**
2    * @brief Snake::getInputs retorna un puntero al vector de distancias,
3    *  que para la red neuronal son los datos de entrada.
4    * @return QVector<int>*
5    */
6   QVector<int>*
7   Snake::getInputs()
8   {
9       return distances;
10  }
```

Fragmento de Código 2.45: Retorna un puntero al vector de distancias

```
1   /**
2    * @brief Snake::getLastDistance recorre desde el final hacia delante
3    * el vector distances y retorna las 4 ultimas en un vector.
4    * @return QVector<int>
5    */
6   QVector<int>
7   Snake::getLastDistance()
8   {
9       QVector<int> dist;
10
11      for (int i=4; i>0; i--) {
12          int temp = distances->at( distances->length() - i);
13          dist.append(temp);
14      }
15      return dist;
16  }
```

Fragmento de Código 2.46: Retorna un vector con la ultima distancia de la serpiente

```
1   /**
2    * @brief Snake::getBody retorna todo el cuerpo de la serpiente.
3    * @return QVector<BodyPart*>*
4    */
5   QVector<BodyPart*>*
6   Snake::getBody()
7   {
8       return &body;
9   }
```

Fragmento de Código 2.47: Retorna un puntero al vector del cuerpo de la serpiente

```
/**
 * @brief Snake::getHeadPosition retorna la posición, x e y, de la
 * cabeza de la serpiente.
 * @return QPoint
 */
QPoint
Snake::getHeadPosition()
{
    return QPoint(body.at(head)->x(),
                  body.at(head)->y());
}
```

Fragmento de Código 2.48: Devuelve la posición de la cabeza en un QPoint

Clase `BodyPart`

```
/**
 * @brief BodyPart::BodyPart constructor de cada parte del cuerpo
 * recibe posiciones x e y.
 * @param x
 * @param y
 * @param parent
 */
BodyPart::BodyPart(int x, int y, QGraphicsRectItem *parent) :
    QGraphicsRectItem(0, 0, BODY_SIZE, BODY_SIZE, parent)
{
    setPos(x, y);
    formatBody();
}
```

Fragmento de Código 2.49: Constructor de cada parte del cuerpo

```
/**
 * @brief BodyPart::formatBody configura el estilo de la parte
 * del cuerpo.
 */
void
BodyPart::formatBody()
{
    setBrush(QBrush(SNAKE_COLOR, Qt::SolidPattern));
    setPen(QPen(Qt::lightGray));
}
```

Fragmento de Código 2.50: Añade color a cada parte del cuerpo (relleno y trazado)

```
/**
 * @brief BodyPart::getPositions retorna la posición de
 * la parte del cuerpo actual de la serpiente.
```

```
 * @return
 */
QPoint
BodyPart::getPositions()
{
    QPoint positions;

    positions.setX(x());
    positions.setY(y());

    return positions;
}
```

Fragmento de Código 2.51: Retorna un QPoint que contiene la posición de cada parte del cuerpo

Clase `Fruit`

```
/**
 * @brief Fruit::Fruit constructor de la fruta
 * @param parent
 */
Fruit::Fruit(QGraphicsEllipseItem *parent) : QGraphicsEllipseItem(parent)
{
    initializeFruit();
}
```

Fragmento de Código 2.52: Constructor de la fruta

```
/**
 * @brief Fruit::initializeFruit inicializa la fruta con un color de fondo
 * y un trazado en la posición 0 0.
 */
void
Fruit::initializeFruit()
{
    QPen   *pen   = new QPen(Qt::white);
    QBrush *brush = new QBrush(FRUIT_COLOR, Qt::SolidPattern);

    setBrush(*brush);
    setPen(*pen);
    setRect(0, 0, FRUIT_SIZE, FRUIT_SIZE);
#ifdef __DEBUG_FRUIT__
    fr = 0;
#endif
    delete pen;
    delete brush;

}
```

Fragmento de Código 2.53: Inicializa la fruta, color y posición

```
/**
 * @brief Fruit::moveFruit mueve la fruta a una nueva posición y comprueba si
 * es distinta a la de alguna parte del cuerpo de la serpiente, si coincide
 * se vuelve a generar.
 * @param snake
 */
void
Fruit::moveFruit(QVector<BodyPart*> *snake)
{
    QPoint *pos;
    do
    {
        pos = new QPoint();
        randomPos(pos);
        setPos(*pos);

#ifdef __DEBUG_FRUIT__
        if (fr == 5)
        {
            pos->setX(snake->at(0)->x());
            pos->setY(snake->at(0)->y());
            setPos(*pos);
            fr = 0;
        }
        fr++;
#endif
        delete pos;
    }
    while( samePosition(snake) );
}
```

Fragmento de Código 2.54: Mueve la fruta a una nueva posición

```
/**
 * @brief Fruit::randomPos genera una posición aleatoria para la fruta.
 * @param position
 */
void
Fruit::randomPos(QPoint *position)
{
    timespec timeSeed;
    QRandomGenerator *rand;
    int newFruitX, newFruitY;

    /* Seed para el random (se puede usar .tv_sec o .tv_nsec)*/
    timespec_get(&timeSeed, TIME_UTC);
    rand = new QRandomGenerator(timeSeed.tv_nsec);

    newFruitX = rand->bounded(SCENE_WIDTH  - FRUIT_SIZE);
```

```
17      newFruitY = rand->bounded(SCENE_HEIGHT - FRUIT_SIZE);
18
19      position->setX(mulOfFruitSize(newFruitX));
20      position->setY(mulOfFruitSize(newFruitY));
21
22      delete rand;
23  }
```

Fragmento de Código 2.55: Genera una posición aleatoria para la fruta

```
1   /**
2    * @brief Fruit::samePosition verifica si la posición de la fruta coincide
3    * con alguna posición del cuerpo de la serpiente, si es asi retorna true, si
4    * no retorna false.
5    * @param snake
6    * @return
7    */
8   bool
9   Fruit::samePosition(QVector<BodyPart*> *snake)
10  {
11      int fruitX, fruitY, snakeX, snakeY;
12      bool equals = false;
13
14      fruitX = x();
15      fruitY = y();
16
17      for (int i=0; i<snake->size(); i++)
18      {
19          snakeX = (int) snake->at(i)->x();
20          snakeY = (int) snake->at(i)->y();
21
22          if ((snakeX == fruitX) && (snakeY == fruitY))
23          {
24              equals = true;
25
26  #ifdef __DEBUG_FRUIT__
27              qDebug("COINCIDE\n");
28              qDebug("F-X: %d\
29                     \rF-Y: %d\
30                     \rS-X: %d\
31                     \rS-Y: %d\n",
32                     fruitX, fruitY,
33                     snakeX, snakeY);
34
35              qDebug("FIN\n");
36  #endif
37
38              return equals;
39          }
40      }
```

```
41      return equals;
42  }
```

Fragmento de Código 2.56: Retorna verdadero o falso si la fruta se ha generado en la misma posición que la serpiente

Glosario

- **Autograd:** Paquete de PyTorch encargado de realizar operaciones con tensores
- **Conexión:** Una conexión hace referencia al peso que unen dos neuronas.
- **Detach:** Método que excluye el seguimiento de Autograd a un Tensor*Detach*.
- **PyTorch:** Librería de aprendizaje automático basada en la biblioteca *Torch*.
- **Linear:** Define una capa del modelo de la red
- **Sequential:** Actúa como un contenedor secuencial, al realizarse la llamada se ejecutan los módulos en el orden establecido por los argumentos
- **Target:** Representa los valores objetivo en la fase de entrenamiento de una red neuronal y se utilizan
- **Tensor:** Estructura de datos principal que utilizan las redes neuronales, es equivalente a un array multidimensional o matriz
- **Torch:** Biblioteca de código abierto utilizada principalmente en el ámbito científico.

Índice de figuras

Bibliografía

[1] Havoc Pennington, Anders Carlsson, Alexander Larsson, Sven Herzberg, Simon McVittie, David Zeuthen (2020, Abril) D-Bus Specification *[freedesktop]* `https://dbus.freedesktop.org/doc/dbus-specification.html`

[2] Havoc Pennington, Anders Carlsson, Alexander Larsson, Sven Herzberg, Simon McVittie, David Zeuthen (2018) dbus-python: Python bindings for D-Bus *[freedesktop]* `https://dbus.freedesktop.org/doc/dbus-python/index.html`

[3] The Qt Company (2018, Diciembre) Qt Reference Pages *[Qt Documentation]* `https://doc.qt.io/qt-5.15/reference-overview.html`

[4] Adam Paszke, Sam Gross, Soumith Chintala, Gregory Chanan (2020) Pytorch Documentation *[Pytorch]* `https://pytorch.org/docs/stable/index.html`

[5] Damián Jorge Matich (2001, Marzo) Redes Neuronales: Conceptos Básicos y Aplicaciones *[Informática Aplicada a la Ingeniería de Procesos, Universidad Tecnológica Nacional]* `https://www.frro.utn.edu.ar/repositorio/catedras/quimica/5_anio/orientadora1/monograias/matich-redesneuronales.pdf`

[6] Pedro Larranaga, Inaki Inza, Abdelmalik Moujahid Redes Neuronales *[Departamento de Ciencias de la Computación e Inteligencia Artificial, Universidad del País Vasco]* `http://www.sc.ehu.es/ccwbayes/docencia/mmcc/docs/t8neuronales.pdf`

[7] Pratik Shukla, Roberto Iriondo (2020, Junio) Neural Networks from Scratch with Python Code and Math in Detail I *[Towards AI]* `https://towardsai.net/neural-networks-with-python`

[8] Pratik Shukla, Roberto Iriondo (2020, Junio) Building Neural Networks with Python Code and Math in Detail II *[Towards AI]* `https://towardsai.net/building-neural-nets-with-python`

[9] Pratik Shukla, Roberto Iriondo (2021, Marzo) Main Types of Neural Networks and its Applications Tutorial *[Towards AI]* `https://towardsai.net/building-neural-nets-with-python`