Razonamiento Automático
Machine Learning

Álvaro Jover Álvarez (aja10@alu.ua.es)
Jordi Amoros Moreno (jam80@alu.ua.es)
Cristian García Romero (cgr71@alu.ua.es)
Universidad de Alicante

January 8, 2019

Índice

1 Tecnologías implementadas y estado operativo.

1.1 Análisis de RAM

Uno de los puntos más importantes a la hora de implementar un bot con IA para un juego de Atari, es entender cómo está hecho. Utilizaremos el entorno *Arcade Learning Environment* (ALE) para extraer características de los juegos, el cual cuenta con una API que nos permite extraer información de los mismos. Para ello, se ha desarrollado un lector de RAM que nos ayuda a visualizar los 128 bytes de memoria de la Atari mientras se ejecuta un juego.

Además, dicho lector implementa colores, lo cual permite que se puedan distinguir las posiciones de RAM que cambian de las que no en un step determinado (paso de ejecución) como se puede ver en la figura 1.

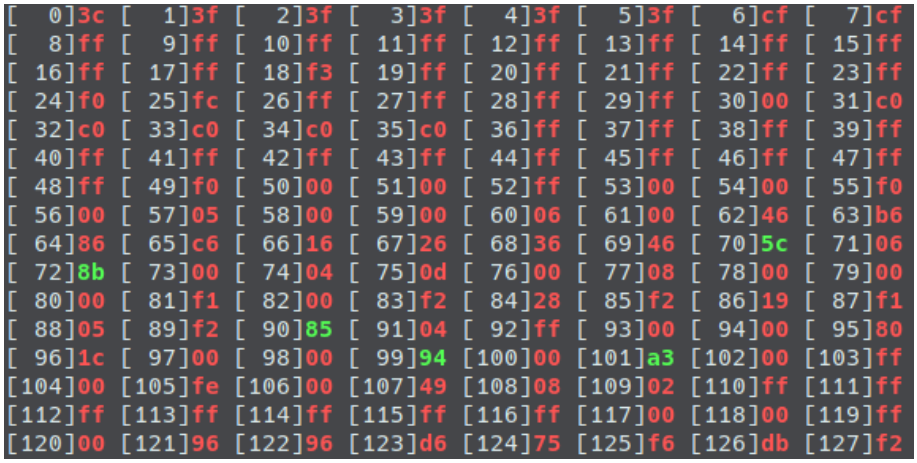


Figure 1: El color verde indica que el valor ha cambiado en este step.

Una de las características de este lector es que acompaña la ejecución con un volcado de analytics para ver las posiciones de RAM que más han cambiado en una ejecución determinada.

Para extraer los datos mas interesantes de un juego en concreto, simplemente hay que observar las posiciones de RAM mas alteradas según nuestro analytics. Una vez hecho esto, se pondrá el juego en cámara lenta gracias a una feature del entorno ALE, lo cual nos permitirá ver con qué sentido cambian estos valores. Como punto a destacar, no todos los valores que cambian mucho serán relevantes a la hora de sacar datos importantes del juego (un contador podría no ser relevante para un caso específico).

Una vez hecho esto se puede desglosar la RAM de manera bastante precisa, sacando datos como los siguientes.

```
Para las coordenadas X hay que aplicar la formula que obtiene cada nibble por separado y, ademas,
invierte el primer nibble

Las coordenadas X de los enemigos van desde:
  Cuando las 2 moscas estan unidas:
    Parte izquierda: desde 16 hasta 147
    Parte derecha: desde 24 hasta 155

  Cuando el enemigo muere, la coordenada X vale 0 y va aumentando hasta alcanzar el valor de donde
reaparece
  Cuando el enemigo muere definitivamente en la ronda actual, la coordenada X se queda congelada con
el ultimo valor que tuvo cuando el enemigo estaba vivo

La coordenada X del jugador va desde 21 hasta 138

0: El valor indica la ronda actual que se esta jugando. Se pasa de una ronda a otra cuando se
eliminan todos los enemigos en pantalla. Al terminar la ronda, el jugador se le transporta a la
posicion del medio de la pantalla (donde se empieza a jugar al principio de la partida). El valor
inicial es 0 y va aumentando
1: Modifica el contador de la puntuacion (valor * 10000)
2: * (parece tener el valor 0 todo el rato)
3: Modifica el contador de la puntuacion (valor * 100)
4: * (parece tener el valor 0 todo el rato)
5: Modifica el contador de la puntuacion (valor * 1)
6: * (parece tener el valor 0 todo el rato)
7: Parece seguir algun patron para el enemigo mas lejano. Cuando el enemigo mas lejano deja de
aparecer en la ronda actual, deja de cambiar de valor y se queda en un valor fijo
8: Igual que 7 pero para el enemigo de en medio
9: Igual que 7 pero para el enemigo mas cercano
10: Parece seguir algun patron para el enemigo mas lejano. Cuando el enemigo mas lejano reaparece en
la ronda actual, el valor empieza a cambiar, y cuando ya ha reaparecido, se congela
```

Figure 2: Demon Attack - Análisis de las primeras posiciones de RAM

Como se puede observar en la figura 2, para obtener la información correcta no solo basta con extraer las posiciones relevantes, en algunos casos será necesario procesar esta información. Por ejemplo, en Demon Attack, las coordenadas X de las entidades aparecen ofuscadas de la siguiente manera:

```
Valor original coordenada X en RAM:
5A -> Primera conversión -> A5 -> Segunda conversión -> A2
```

Figure 3: Demon Attack - Coordenadas X de las entidades

Como podemos ver en la figura 3, los nibbles de las coordenadas están invertidos, además, el primer nibble requiere una operación extra, una resta (7 - valor del nibble).

Una vez tenemos la información recogida y procesada, la podremos utilizar para crear una IA capaz de jugar al juego en concreto. Además de eso, el entorno ALE cuenta con diversas funcionalidades que nos permiten recoger la información en pantalla en el caso que fuese necesario.

1.1.1 Modificando ALE

La primera aproximación que se hizo a la hora de analizar los datos de los diferentes juegos fue básica y útil, pero presentaba varios problemas. El problema más serio que nos encontramos era la cantidad de datos que teníamos ante nuestros ojos, cosa que puede dificultar bastante el encontrar la información importante que necesitamos y, además, que dependemos totalmente de nuestra "intuición" para saber que significa cierta posición de la RAM. Intentando solventar un poco este problema, se nos ocurrió aislar las posiciones que hay en la RAM y, además, poder modificar esos valores aislados para no depender totalmente de nuestra intuición, sino saber más a ciencia cierta, a base de prueba y error, saber qué hace cierta posición de RAM, lo cual nos daría la ventaja de profundizar más en aquellas posiciones de RAM que por alguna razón, en el paso anterior, hemos visto que podían resultar relevantes.

A la hora de intentar poner esta idea en práctica lo primero que intentamos es obtener una referencia de la RAM desde el objeto que hemos ido empleando en todo momento: **ALEInterface**. Este objeto nos dota del método *getRAM()* que hemos ido utilizando en todo momento para el análisis de los datos (y para posteriormente el tratamiento de los mismos) y que está en el archivo *src/ale_interface.cpp*. La implementación de dicho método es la siguiente:

```
// Returns the current RAM content
const ALERAM& ALEInterface::getRAM() {
  return environment->getRAM();
}
```

Figure 4: Método *getRAM()*

Como observamos, el método *getRAM()* nos devuelve una referencia de un objeto de tipo **ALERAM**, por lo que vamos a ver que método nos pueden servir para modificar la RAM. Cuando investigamos el archivo *src/environment/ale_ram.hpp* encontramos la definición del objeto ALERAM y la implementación de sus métodos. Contiene 2 métodos que pueden servirnos para nuestro propósito (modificar la RAM), los cuales se muestran en la figura 5.

```
/** Returns a reference to a concrete byte. */
byte_t *byte(unsigned int x);

/** Returns the whole array (equivalent to byte(0)). */
byte_t *array() const { return (byte_t*)(m_ram); }
```

Figure 5: Métodos de acceso a la RAM de ALEInteface

Una vez ya encontramos estos métodos, intentamos hacer una pequeña prueba, la cual falla, ya que al intentar hacer uso del método *array()* vemos que modificamos un valor y que, efectivamente, al volver a comprobar el valor, se ha modificado, pero nada sucede en el juego (la prueba la realizamos sobre el videojuego *breakout*, el cual tiene la coordenada X de la barra en la posición 72). Lo que estaba sucediendo, después de indagar un poco por el código de ALE era que el método *getRAM()* de *ale_interface.cpp* era un método que a su vez llamaba a otro método llamado *getRAM()* pero del objeto **StellaEnvironment**, objecto que es el encargado de manejar todo el entorno de ALE. Este objeto, de nuevo, devuelve otra vez un objeto de tipo ALERAM por referencia, por lo que de nuevo parece que no hay problema, pero este objeto, *StellaEnvironment*, contiene una copia de la RAM del objeto **OSystem**, objeto que es el núcleo de la Atari 2600 (del emulador). El problema consiste en que este objeto ALERAM con el que constantemente trabajamos es una copia del que el objeto *OSystem* contiene, es decir, cada vez que llamados al método *getRAM()* desde *ale_interface.hpp*, se hace otra llamada al método *getRAM()* de *src/environment/stella_environment.cpp* y este devuelve su copia interna, no la del núcleo de la consola. Cada vez que el objeto *StellaEnvironment* actualiza su copia hace uso del método *processRAM()*.

```
void StellaEnvironment::processRAM() {
  // Copy RAM over
  for (size_t i = 0; i < m_ram.size(); i++)
    *m_ram.byte(i) = m_osystem->console().system().peek(i + 0x80);
}
```

Figure 6: Método *processRAM()* de StellaEnviroment

Una vez llegado a este punto, lo que se puede hacer si se quiere modificar la RAM que contiene el objeto *OSystem* es implementar un método que lo que haga es actualizar la RAM del objeto *OSystem* del mismo método que que podemos observar que se puede obtener una copia de la RAM con su método *console().system().peek()*, pero tenemos que comprobar si existe el método que nosotros necesitamos, y es que en vez de *peek()* necesitamos un método que nos permita actualizar. De nuevo, buscando en el código, encontramos el archivo *src/emucore/OSystem.hxx*, el cual nos lleva al archivo *src/emucore/Console.hxx*, el cual nos lleva, finalmente, al archivo *src/emucore/System.hxx*. En este último encontramos el método *poke(uInt16, uInt8)*, el cual nos indica a través de su descripción que modifica un valor de la RAM de la consola Atari 2600. Parece ser que ya lo tenemos todo, pero nos queda un último problema, y es que este método no es accesible desde el objeto *StellaEnvironment*, ya que su referencia del objeto *OSystem*, a diferencia del objeto *ALEInterface*, está en su parte privada y no nos da ningún método público para obtener una copia de la referencia. El último paso que nos queda para solucionar este último problema es modificar el código y añadir un método que nos permita actualizar la RAM de la Atari 2600 a partir de la copia que tiene internamente el objeto *StellaEnvironment*, ya que esta última sí que podemos modificarla.

```
/**
  Change the byte at the specified address to the given value.
  No masking of the address occurs before it's sent to the device
  mapped at the address.

  @param address The address where the value should be stored
  @param value The value to be stored at the address
*/
void poke(uInt16 address, uInt8 value);
```

Figure 7: Método *poke(uInt16, uInt8)*

El método que hemos implementado es muy sencillo. Es igual que el método *processRAM()* pero utilizando el método *poke()* en vez de *peek()*. Una vez ya implementado el método, simplemente implementamos otro método en el objeto *ALEInterface* que simplemente haga una llamada a este método que está en el objeto *StellaEnvironment* y ya habremos actualizado la RAM de la Atari 2600. Por último solo quedaría recompilar ALE. Este proceso que hemos realizado a sido con el único objetivo de compren-

der mejor el significado de las posiciones de RAM de los diferentes juegos a analizar durante este trabajo. Se adjuntarán para cada juego un archivo denominado **RAM.txt** en el cual hemos analizado cada posición de RAM de manera aislada.

```
void StellaEnvironment::processBackRAM()
{
  // Copy RAM back
  for (size_t i = 0; i < m_ram.size(); i++)
    m_osystem->console().system().poke(i + 0x80, *m_ram.byte(i));
}
```

Figure 8: Método que actualiza la RAM de la Atari 2600

1.2 Bots básicos

Se han desarrollado bots semi-deterministas empleando técnicas básicas de inteligencia artificial para poder extraer datos de gameplay. Los primeros volcados de datos se hicieron con operarios humanos jugando a los juegos, pero al ver que nuestras scores eran mas bien bajas, se optó por implementar IA básica para cada uno de los juegos.

Estas implementaciones básicas mejoraron mucho las scores obtenidas, por lo que los datos extraídos de los bots eran mas afines a obtener mayores puntuaciones que los nuestros.

Además, sobre esta IA básica, se pueden hacer iteraciones de mejora, teniendo en cuenta más datos o mas información en pantalla, como se ha comentado anteriormente en la subsección 1.1.

Este scripting básico ayudará mas adelante a la implementación utilizando machine learning, ya que los datos extraídos y procesados para la implementación básica serán utilizados por el algoritmo de machine learning.

A continuación, describiremos cada uno de los bots básicos y su funcionamiento al igual que algunos detalles de implementación.

1.2.1 Plantilla común

Todos los bots comparten una serie de utils que analizaremos a continuación.

```
/**
* argv[1] : rom
* argv[2] : media? true/>false<
* argv[3] : print_ram? true/>false<
* argv[4] : to csv? true/>false<
**/
const bool display_media(argc >= 3 ? atoi(argv[2])==1 : false);
const bool printRam(argc >= 4 ? atoi(argv[3])==1 : false);
toCSV = argc == 5 ? atoi(argv[4])==1 : false;
```

Figure 9: Opciones de ejecución

argv 1 es la ROM que utilizará nuestro ejecutable, argv 2 se corresponde con el contenido multimedia (video y audio), argv 3 escribirá la RAM en consola y argv 4 exportará los datos de gameplay a un archivo *valores separados por comas* (CSV) si así se requiere. Se han parametrizado estas opciones porque las ejecuciones son mucho mas lentas conforme mas información requiramos, esto se nota sobre todo a la hora de desactivar el contenido multimedia.

```
alei.setBool("sound", display_media);
alei.setBool("display_screen", display_media);
alei.loadROM(argv[1]);
```

Figure 10: La ROM y el contenido multimedia corren a cargo de ALE.

Algunas de estas opciones corren a cargo del entorno (Figura 10), mientras que las otras han sido implementadas por nosotros.

Otra de las partes comunes a todos los bots es el bucle principal de ejecución que podemos ver a continuación en la figura 11. Este bucle está situado en **main()** y es el encargado de analizar y ejecutar las acciones requeridas en cada step del juego en activo.

```
for (step = 0; !alei.game_over() && step < maxSteps; ++step)
{
    // Debug mode **********************************
    if(printRam) printRAM();
    if(display_media) checkKeys();
    // *********************************************

    // Total reward summation
    totalReward += manualInput ? manualMode() : agentStep();
}
```

Figure 11: Bucle principal de ejecución.

En este bucle observamos nuestra opción **printRam** que llama al método encargado de imprimir la RAM, el cual convierte a hexadecimal cada uno de los valores de RAM obtenidos mediante el método **getRAM().get(i)** del entorno ALE, además hace un seguimiento de los valores de RAM del step anterior para comprobar si dichos valores han cambiado como hemos visto anteriormente en la figura 1.

Otra opción que encontramos en el bucle principal de ejecución es **display_media**, pero en este caso es usado para llamar al método **checkKeys()**, esto se hace porque no tiene sentido trackear el input si no existe contenido de vídeo. Este método de input es propio, ya que el método de input de ALE ”pausa” el agente, lo cual no nos interesa para extraer datos. **checkKeys()** simplemente activa o desactiva el modo manual propio con la tecla ”E”, reflejado en la variable **manualInput**.

La variable **manualInput** decidirá que función se llama para calcular **totalReward**. **manualMode()** mapea el teclado a diferentes acciones del juego, mientras que **agentStep()** es el bot autónomo específico a cada juego.

Otra de las partes comunes a todos los agentes es la parte de volcado de datos, para ello se han implementado dos funciones de escritura que imprimen strings o dobles en el archivo CSV anteriormente comentado, como bien podemos ver en la figura .

```
void write(double d)
{
    if (toCSV)
    {
        csv << to_string(d);
    }
}
```

Figure 12: Una de las funciones de escritura en CSV.

1.2.2 Breakout

El Breakout es el juego mas simple de todos los que analizaremos en esta parte de la sección, pues el número de inputs que tiene es mas bien pequeño. En el Breakout contamos con un total de 5 vidas para pasarnos los 2 niveles de los que dispone. El Breakout fue el primer juego en el que vimos la necesidad de automatizar al jugador, ya que la velocidad de la pelota va incrementando en función a los ladrillos restantes que quedan, lo cual provocaba que perdiésemos siempre en este punto.

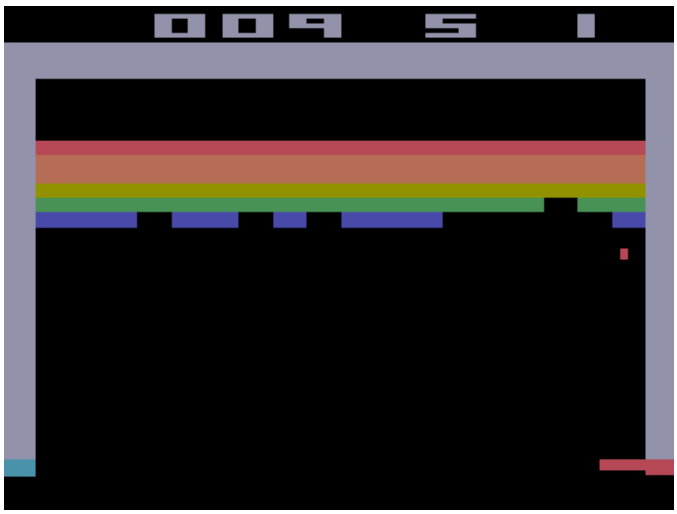


Figure 13: Juego Breakout.

Simplemente proporcionando la posición X de la pelota y de la pala al bot y moviendo la pala hacia la pelota, el agente ya jugaba bastante bien. Una mejora que se hizo al algoritmo es hacer un control del tamaño de la pala, al igual que en vez de tener en cuenta solo la posición actual de la pelota, añadir a los datos la posición anterior. Con todas estas mejoras la puntuación se maximizó hasta el punto en el cual el agente fue capaz de pasarse el juego completo. A continuación se muestra en el algoritmo 1 el pseudocódigo de la IA del Breakout.

Algorithm 1: Breakout agent

```
if lives() != lastLives then
    --lastLives;
    act(FIRE);
end
wide := getRAM.get(108);
playerX := getPlayerX();
ballX := getBallX();
if BallX_LastTick < ballX then
    ballX += ((rand()%2) + 2);
end
if BallX_LastTick > ballX then
    ballX -= ((rand()%2) + 2);
end
ballX_LastTick := getBallX();
if ballX < playerX + wide then
    reward += act(LEFT);
else
    if (ballX > playerX + wide)&&(playerX + wide < 188) then
        reward += act(RIGHT);
    end
end
```

Si analizamos el pseudocódigo anterior, podemos ver cuatro partes. En la primera parte, constituida por el primer if, vemos como el agente presiona la tecla **FIRE** cuando pierde una vida, esto se debe a que en Breakout cuando pierdes una vida tienes que sacar la pelota pulsando esa tecla.

En la segunda parte recogemos los datos principales, en este caso **wide**, que corresponde al ancho de la pala, además de **playerX** y **ballX**.

La tercera parte calcula la dirección de la bola, esto se saca comprobando la posición anterior de la bola con la actual en el eje X, una vez sabemos

la dirección aplicamos una suma con un poco de aleatoriedad (para evitar ejecuciones deterministas) en la dirección recogida. Una vez hecho eso nos guardamos la posición de la pelota para la siguiente iteración.

En la cuarta parte aplicaremos el input en función a la posición del jugador respecto a la pelota, teniendo en cuenta el ancho de la pala recogido anteriormente.

1.2.3 Boxing

Boxing es un juego en el cual controlamos a un boxeador y tenemos que asestar mas golpes que el rival para ganar la ronda. En este juego tuvimos los mismos problemas que con el Breakout, por lo que decidimos implementar otro bot, el cual jugaba mejor que nosotros, lo que se resumía en todo ventajas.

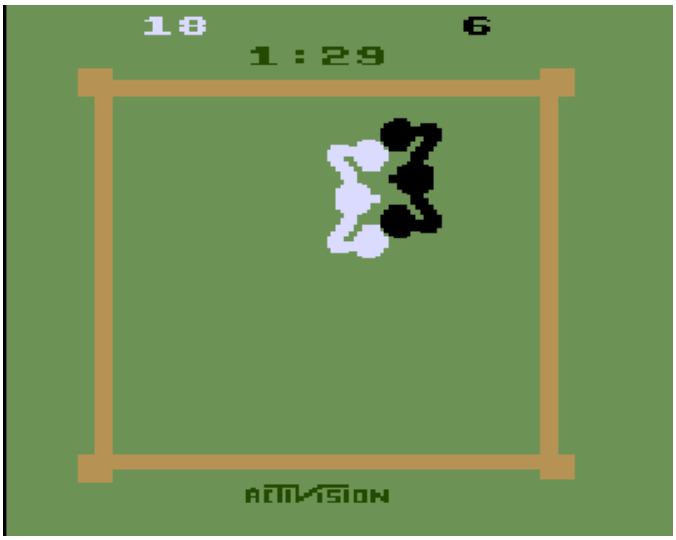


Figure 14: Juego Boxing.

En este agente hemos tomado una estrategia agresiva, si el rival lanza un puñetazo, lo intentaremos bloquear con nuestros propios puños poniendo a nuestro boxeador exactamente en la misma posición "Y" que el boxeador contrario (una de las features de Boxing es que la colisión de los puños bloquea los golpes). Además, nuestro boxeador nunca se moverá hacia la izquierda, solo se mueve hacia la derecha en posición de ataque intentando posicionarse en la "Y" del rival, como hemos comentado antes. A su vez, basándonos en ciertas tolerancias, se atacará siempre que sea posible. Esta

estrategia agresiva funciona una gran mayoría de las veces, además para evitar el determinismo se ha implementado una pequeña aleatoriedad cada vez que recogemos las posiciones del jugador 1 y del jugador 2 de RAM, como bien se puede ver en el ejemplo de la figura 15.

```
int getP2_X()
{
   return alei.getRAM().get(33) + ((rand() % 2) - 1);
}
```

Figure 15: Uso de **rand()** para evitar el determinismo.

A continuación se muestra en el algoritmo 2, la IA implementada. Un detalle a comentar antes de entrar a analizar el algoritmo es que **player_pos** es un tipo de dato struct.

Algorithm 2: Boxing agent

```
player_pos p1(getP1_X(), getP1_Y());
player_pos p2(getP2_X(), getP2_Y());
absp1p2X := abs(p1.x - p2.x);
absp1p2Y := abs(p1.y - p2.y);
if absp1p2Y > 3 && absp1p2Y < 20 then
    reward += act(FIRE);
else
    if absp1p2X > 25 && absp1p2X < 40 then
        reward += act(RIGHT);
    else
        reward += (p1.y > p2.y) ? act(UP) : act(DOWN);
    end
end
```

En la primera parte de recogida de datos, encapsulamos en un struct la posición del jugador uno y la del jugador dos, extrayéndolos de RAM. Como ya hemos comentado anteriormente, estas posiciones implementan una aleatoriedad mínima para evitar ejecuciones deterministas. Además, en la misma sección, calcularemos la distancia entre ambos jugadores en "X" y en "Y", representadas en **absp1p2X** y **absp1p2Y** respectivamente, para luego utilizarlas mas adelante.

Una vez tenemos todos los datos procesados, observaremos si nos encontramos en un rango de tolerancia "Y" válido para atacar, si lo estamos, atacaremos (como punto a destacar, este rango es bastante amplio para enfatizar esta estrategia ofensiva). Si no podemos atacar, nos moveremos hacia la derecha con ciertas tolerancias, o nos situamos en la "Y" del enemigo.

1.2.4 Demon Attack

El Demon Attack es un juego en el cual controlamos a una nave con un patrón de movimiento similar al Space Invaders. Tendremos que disparar a las naves rivales para pasar de fase además de evitar todo contacto enemigo. El jugador cuenta con una serie de vidas para pasarse los niveles, si estas vidas se acaban el juego termina. De nuevo, vimos la necesidad de crear un bot debido a las bajas puntuaciones obtenidas.

Figure 16: Juego Demon Attack.

El Demon Attack es, sin duda, uno de los juegos mas complejos con los que hemos tenido que lidiar, debido a la cantidad de entidades en pantalla, desde disparos hasta múltiples enemigos. La estrategia de IA seguida en el Demon Attack es compleja debido al factor de la gran cantidad de entidades, pues no solo leemos valores en RAM, sino que aprovechamos la característica de ALE que nos permite leer los píxeles de la pantalla.

Al analizar la RAM, observamos que habían demasiadas características a tener en cuenta, a pesar de tener un análisis detallado de la misma. Hay posiciones de RAM que representan varias cosas en diferentes situaciones, un ejemplo es el valor representado por *RAM[20]*:

"20: Coordenada "X" de las balas del enemigo. También es la coordenada

"X" de la mosca que se acerca para intentar matar al jugador 1 (se utiliza como una bala). Va desde 29 hasta 147 (incluso cuando dispara la mosca). El valor es todo el rato el mismo hasta que el enemigo dispara y actualiza la coordenada. Si hay mas de 2 cambios consecutivos en 2 frames significa que no es una bala y que es la mosca que sigue al jugador".

Es por ello por lo que se dedujo que para implementar un bot básico (sin machine learning), era más fácil hacer un análisis de lo que estuviera pasando en la pantalla en un instante determinado, sin dejar de lado la RAM. Para esto, se diseñó un método de visión en el cual la nave es capaz de ver y filtrar enemigos en un área de visión y actuar concorde la situación dependiendo de unas reglas determinadas. En la figura 17 se puede observar el área de visión de la nave.

Figure 17: Visión de la nave.

La visión de la nave se divide principalmente en 3 partes. En la figura vemos una línea roja representada con el número 1, esta línea roja representa la visión de la nave en función a la anchura de la misma. El número 2 es un

área parametrizable que permite ampliar esta visión de forma simétrica por los dos lados de la nave. Tanto el rectángulo rojo como el azul se extienden por casi toda la "Y" de la pantalla y lo único que filtran son los disparos enemigos, como los disparos enemigos tienen el mismo color es muy simple filtrarlos. Por lo tanto 1 y 2 se encargan de detectar disparos enemigos, una vez detectados se actuará en consecuencia.

El número 3 es un área reducida dentro del área completa que se encarga de detectar todas las posibles amenazas, no solo disparos, que estén cerca de la nave. Es decir, el objetivo de la visión es la detección de entidades peligrosas en un área de riesgo para posteriormente evitarlas.

El seguimiento de enemigos para atacarles y matarlos corre a cargo de la RAM, aunque no es del todo exacto debido al problema comentado anteriormente de posiciones de RAM no especificas a una única funcionalidad.

Para analizar la parte práctica de éste algoritmo es necesario separarlo en distintas partes, detección, esquiva y seguimiento.

La parte de detección funciona recogiendo la pantalla entera en grayscale, esto lo hacemos gracias a una funcionalidad de ALE, una vez tenemos la pantalla guardada, podemos ver cual es el valor de cada píxel. Gracias a esto, podemos determinar el color de los disparos y de las naves enemigas, los que guardaremos en las siguientes variables para después su posterior filtrado:

```
// non modificable
const int LINE_WIDTH(160);          // Amount of pixels in a line
const int SHIP_WIDTH(7);            // Ship width in pixels
const int LEFT_THRESOLD(21);        // MIN X non-pixel coordinate the ship can move 25
const int RIGHT_THRESOLD(138);      // MAX X non-pixel coordinate the ship can move 135
const int P1_BULLETS_COLOR(174);
const int P1_SHIP_COLOR(115);
const int EN_BULLETS_COLOR(176);
```

Figure 18: Constantes fijas relativas a la visión de la nave.

En la figura 18 podemos observar diversas constantes, entre ellas el color de las balas del jugador 1, **P1_BULLETS_COLOR**, el color de la nave del jugador 1, **P1_SHIP_COLOR** y el color de las balas enemigas, **EN_BULLETS_COLOR**. Gracias a estar en grayscale, con tener un solo valor es suficiente. Además podemos observar otras constantes como **LINE_WIDTH**, que representa el ancho en píxeles de la resolución de la Atari, **SHIP_WIDTH**, el ancho en píxeles de la nave y **LEFT / RIGHT_-THRESOLD**, que es la coordenada "X" mínima y máxima en la que se puede mover la nave aliada.

Además de estas constantes, Demon Attack cuenta con cuatro constantes adicionales que nos permiten parametrizar algunas características de nuestro algoritmo, como se puede ver en la figura 19 .

```
// modifiable params
const int VISION_THRESOLD(10);
const int SECOND_VISION_LINE_THRESOLD(168);
const bool bRandomisePlayerGPos(false);
const bool bRandomiseEnemyGPos(false);
```

Figure 19: Constantes modificables (no runtime).

La primera variable de la figura representa el área extra de visión 2 definida en la imagen 17, la segunda variable define en que línea empieza el área de visión 3. Las dos variables restantes es aleatoriedad opcional a la hora de recoger las posiciones, como se ha hecho antes en el *Boxing*. En este caso, debido a la gran cantidad de entidades en pantalla se ha optado por deshabilitar la aleatoriedad y hacer las batallas mas deterministas.

Otro dato a remarcar antes de empezar con el algoritmo es el Enum utilizado para las colisiones:

Algorithm 3: Enum empleado para las colisiones.

enum BlockingHit { $EMoveRight$, $EMoveLeft$, $ENotBlocking$ };

Para entender el Enum del algoritmo 3 tenemos primero que comprender las respuestas que puede dar una colisión y como se operan.

- **EMoveRight**: La amenaza se ha detectado en la parte izquierda de nuestra área de visión, para contrarrestarla nos moveremos a la derecha si podemos.
- **EMoveLeft**: La amenaza se ha detectado en la parte derecha de nuestra área de visión, para contrarrestarla nos moveremos a la izquierda si podemos.
- **ENotBlocking**: No se ha detectado amenaza ninguna.

Una vez que ya sabemos qué indicadores utilizaremos para las colisiones, podemos proceder a despiezar el algoritmo.

```
DirtyState ds(false, ENotBlocking);
float agentStep()
{
    // get screen information
    alei.getScreenGrayscale(grayscale);

    // Reseting variables
    float reward = 0;
    BlockingHit eBH = ENotBlocking;


    // Iterating from bottom line to top line which defines a vision rectangle (see is BlockingHit)
    for(int line = 185; line > 60; --line) {
        eBH = isBlockingHit(line);
        if (eBH != ENotBlocking && !ds.Dirty)
        {
            /**
            * Dirtying the warning state, now we have to avoid every enemy and undesired object
            **/
            ds.Dirty = true; ds.Direction = eBH;
            break;
        } else if (eBH != ENotBlocking) {
            /**
            * Theorically at this point the direction is already set so we don't have to re-set it here
            * we just mark the dirty state to true
            **/
            ds.Dirty = true;
            break;
        }
    }
```

Figure 20: Primera parte del algoritmo de IA para el Demon Attack.

En esta primera parte recogemos los píxeles de la pantalla con la funcionalidad de ALE **getScreenGrayscale**, para luego emplearlos dentro del bucle en la función **isBlockingHit(int)**, que analizaremos mas adelante. Este bucle se encarga de recorrer 125 filas de la pantalla. En función a la respuesta de **isBlockingHit(int)**, marcará una variable a true, la cual representa que ha habido una colisión, además dentro de este struct **DirtyState**, disponemos de otra variable de tipo EBlockingHit para marcar también la dirección contraria a la colisión dada por isBlockingHit. Se ha decidido este thresold ya que el resto de líneas de pantalla contienen información no relevante para el problema.

Para resumir esta parte del código, marcamos el estado a Dirty y recogemos la dirección conveniente, siempre que isBlockingHit haya detectado una colisión. Hasta que no salimos del estado Dirty, no podremos recoger nuevas direcciones, pero de ello se encargará otra parte del código. Es decir, teóricamente se ha implementado una pequeña máquina de estados finita (FSM), ya que la nave posee dos estados principales (dirty y no dirty) y la forma de cambiar de estado entre ellos es mediante el código de colisiones.

```
BlockingHit isBlockingHit(int line) {
    int const FILTER_LINE(LINE_WIDTH*line);
    int const VISION_X_AREA(SHIP_WIDTH + (VISION_THRESOLD*2));
    for(int i = 0; i < VISION_X_AREA; ++i) {
        const int impact_pixel_val(grayscale[FILTER_LINE + (getP1_X(true)-VISION_THRESOLD) + i]);
        // Avoid flies
        if (line > SECOND_VISION_LINE_THRESOLD && ImpactValIsAnEnemy(impact_pixel_val)) {
            if(i < (VISION_X_AREA/2)) return EMoveRight;
            else if(i >= (VISION_X_AREA/2)) return EMoveLeft;
        }
        // Avoid enemy shooting
        if(impact_pixel_val == EN_BULLETS_COLOR) {
            // Blocking hit, now we have to determine the direction we want to  based on the impact point
            if(i < (VISION_X_AREA/2)) return EMoveRight;
            else if(i >= (VISION_X_AREA/2)) return EMoveLeft;
        }
    }
    return ENotBlocking;
}
```

Figure 21: Función isBlockingHit.

La función isBlockingHit tiene que ser capaz de recorrer el array obtenido por **getScreenGrayscale** de manera eficiente. Para ello lo primero que hace es resolver la posición del array que ocupa el primer píxel de la línea a analizar, esto es fácil, pues simplemente multiplicando el ancho de la línea con el número de línea ya tenemos este número (asumiendo que las líneas empiezan en 0).

La siguiente tarea es resolver el thresold, el cual consiste en el área que ocupa la nave, en este caso 7, mas el thresold determinado, que en nuestro caso corresponde al número de píxeles extras que tendremos en cuenta en cada lado, por ejemplo, si el thresold es 2, el área total será 7+2+2 = 11.

La iteración que haremos en el bucle irá desde 0 hasta el área total que ocupa la visión de la nave, esto es, como ya hemos dicho anteriormente, el área de la nave mas el thresold. Esto hace que solo iteremos los píxeles necesarios para nuestra situación.

La primera línea que encontramos en el bucle puede parecer confusa, pero tiene su explicación:

Algorithm 4: Calculo del valor del píxel en un punto determinado de nuestra area de visión.

```
const int impact_pixel_val(grayscale[FILTER_LINE +
  (getP1_X(true)-VISION_THRESOLD) + i]);
```

Necesitamos el valor del píxel en un punto determinado de nuestra área de visión. Para ello, necesitamos pedirle al array grayscale esta información, pero antes necesitamos calcular qué posición del array es la correcta. Al haber precalculado la posición del array del primer píxel de la línea

lo tenemos mucho mas fácil, pues tenemos por donde empezar. A ese número le tenemos que sumar la posición del jugador uno, representado en **getP1_X(true)**, al cual le pasamos el parámetro true para que nos devuelva la posición en píxeles, la cual es más precisa para este problema. Una vez tenemos la posición del jugador solo nos queda sumarle i para abordar todo el área de visión. Pero como bien podemos observar en el algoritmo 4, estamos restando a la posición del jugador el **VISION_THRESOLD**, esto se hace porque sino estaríamos solo teniendo en cuenta el área extra por la derecha, ya que la colisión estaría desplazada a la izquierda como muestra la figura 22, por ello es necesario restar **VISION_THRESOLD**.,

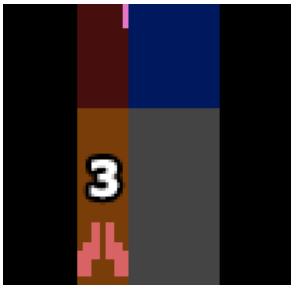


Figure 22: Área de sin restar VISION_THRESOLD al jugador.

Una vez hemos resuelto el valor del píxel, podemos hacer el filtrado del mismo, el cual se realizará en el cuerpo del bucle.

```
// Avoid flies
if (line > SECOND_VISION_LINE_THRESOLD && ImpactValIsAnEnemy(impact_pixel_val)) {
    if(i < (VISION_X_AREA/2)) return EMoveRight;
    else if(i >= (VISION_X_AREA/2)) return EMoveLeft;
}
// Avoid enemy shooting
if(impact_pixel_val == EN_BULLETS_COLOR) {
    // Blocking hit, now we have to determine the direction we want to  based on the impact
    if(i < (VISION_X_AREA/2)) return EMoveRight;
    else if(i >= (VISION_X_AREA/2)) return EMoveLeft;
}
```

Figure 23: Resolución de la colisión

Como bien observamos en la figura 23, haremos una primera distinción para las moscas y enemigos mas cercanos a nuestra segunda área de visión seguido de la visión normal. Priorizamos el segundo área de visión porque está mas cerca del jugador, lo cual se traduce en más peligro a perder una vida. En el primer if, miramos si nos encontramos en el área de riesgo y

filtramos por color principalmente si lo que hay en ese píxel es un enemigo. Si es así, es que ha habido una colisión. Si la i en la cual nos encontramos es menor (o está a la izquierda) de la mitad del área de visión, nos moveremos a la derecha (camino más corto para evitar la colisión). Haremos lo mismo para el otro ladro. El segundo if es exactamente lo mismo solo que nos centramos únicamente en las balas enemigas.

Finalmente, si no se ha encontrado colisión ninguna, como hemos visto en la figura 21, devolveremos *ENotBlocking*.

```
/************************************************
* Blocking hit detected at this point
************************************************/
if(eBH != ENotBlocking)
{
    // Left thresold
    if(getP1_X() == LEFT_THRESOLD) {
        // If we cannot move more to the left we change direction
        ds.Direction = EMoveRight;
    }
    // Right thresold
    else if(getP1_X() == RIGHT_THRESOLD) {
        // If we cannot move more to the right we change direction
        ds.Direction = EMoveLeft;
    }
    switch(ds.Direction){
        case EMoveLeft:
            reward+= alei.act(PLAYER_A_LEFT);
            break;
        case EMoveRight:
            reward+= alei.act(PLAYER_A_RIGHT);
            break;
        default:
            reward+= reward+=alei.act(PLAYER_A_FIRE);
    }

} else {
    // Here we are safe, don't move but keep shooting
    reward+=alei.act(PLAYER_A_FIRE);
    ds.Dirty = false;
    const int mypos = getP1_X();
    const int en_pos = EnemyHandler();
    if(mypos < en_pos) {
        reward+= alei.act(PLAYER_A_RIGHT);
    } else if (mypos > en_pos) {
        reward+= alei.act(PLAYER_A_LEFT);
    } else {
        reward+=alei.act(PLAYER_A_FIRE);
    }
}
return (reward + alei.act(PLAYER_A_NOOP));
```

Figure 24: Aplicando movimiento a la nave

La parte que extiende a la figura 20, es la figura 24. En ésta figura podemos observar la parte del algoritmo que finalmente aplica movimiento a la nave. En este punto ya hemos solucionado la colisión y solo nos queda movernos en la dirección conveniente.

Esta parte del código se divide en dos dependiendo respectivamente si hemos encontrado una colisión o no. En la primera parte ya tenemos en el **DirtyState** apuntado la dirección en la cual movernos, sin embargo si nos encontramos en los bordes de la pantalla, no nos podremos mover la la dirección del borde, es por esto por lo que cambiamos de dirección cuando llegamos a un borde. Si no estamos en ningún borde, simplemente aplicamos la dirección anotada por **ds.Direction**.

```
struct DirtyState {
    bool Dirty;
    BlockingHit Direction;
    DirtyState(bool bDirty, BlockingHit bDirection)
    : Dirty(bDirty), Direction(bDirection) { }
};
```

Figure 25: Struct Dirty State

En la segunda parte, podemos considerar que el jugador uno está a salvo de posibles ataques enemigos, esto viene determinado, como ya hemos comentado anteriormente, por el resultado de isBlockingHit. Por lo tanto, lo que haremos en este modo es disparar siempre que podamos a la vez que intentar traquear a los enemigos en RAM para dispararles dentro de su colisión. **EnemeyHandler** se encargará de devolvernos el enemigo mas relevante en un instante determinado (usualmente el mas cercano al jugador). Esta función no funciona el 100% de los casos, ya que como hemos comentado anteriormente, la RAM no es específica a casos concretos sino que va cambiando, por lo tanto habrá, veces que el jugador 1 se quede disparando a la nada porque no tiene nada que trackear. Esto se resuelve parcialmente gracias a la aleatoriedad y al movimiento de las entidades enemigas, que acabaremos matando si se acercan a nuestro láser. La solución completa y real sería llevar un seguimiento mas especifico de todas las unidades en RAM, lo cual puede llevarnos a implementar demasiados casos específicos para un escenario mas simple de solucionar mediante un algoritmo genérico (mediante una buena función de fitness) o un neuroevolutivo en su defecto.

1.3 Perceptrón

El perceptrón es una de las técnicas de *Machine Learning* más sencillas de comprender, implementar y útiles. En su sencillez radica los buenos resultados que suele dar, siempre que el problema a resolver no sea muy complejo, ya que pasado cierto nivel de complejidad, el perceptrón seguramente no sea la mejor opción. El perceptrón lo que intenta hacer es separar los datos que le demos, asegurándonos que obtendrá la mejor solución posible si los datos de los cuales partimos son linealmente separables, y en el caso de no serlo, deberemos dar la mejor solución posible (aquella con menor error).

El perceptrón es una técnica de aprendizaje supervisado en la que le damos datos etiquetados y, a partir del error obtenido al intentar etiquetar los datos, se hacen ajustes en la configuración para acercarnos más a los resultados correctos. Esta técnica puede utilizarse para clasificación o para regresión. El resultado que al final obtendremos será, dependiendo del número de entradas que tengamos, una línea, un plano o un hiperplano. Los elementos del perceptrón son las entradas, los pesos y la salida.

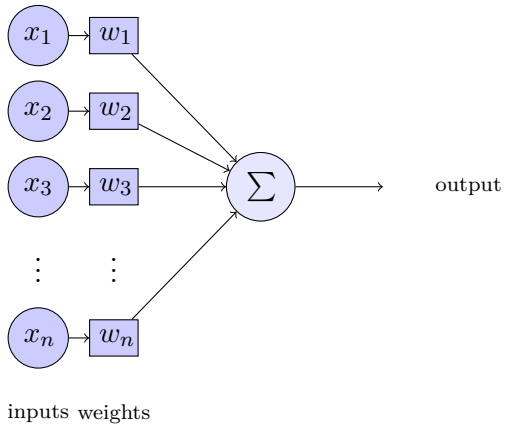


Figure 26: Estructura básica del perceptrón

La salida que tiene el perceptrón, en el caso del clasificador, es siempre de +1 o -1. Para obtener esta salida, la fórmula que se aplica es la siguiente cuando tenemos N entradas:

$$y = signo(\sum_{i=1}^{N} x_i * w_i)$$

Ahora bien, como hemos dicho, lo que el perceptron hace es separar

datos, es decir, encontrar la linea, plano o hiperplano que los separa, pero si nos fijamos en la fórmula que hemos descrito encontramos un problema, y es que no hay término independiente, todos los pesos están asociados con alguna entrada, por lo que la linea, plano o hiperplano siempre pasará por el origen de coordenadas, cosa que seguramente no sea así para la mayoría de los problemas. La solución es, como hemos dicho, añadir un término independiente, al cual se le suele llamar *bias*. Una manera de hacerlo es añadir un peso w0 y una entrada x0 artificial que siempre tenga el valor 1, con lo que al final nos queda la siguiente fórmula:

$$x_0 = 1$$

$$y = signo(\sum_{i=0}^{N} x_i * w_i)$$

Finalmente, la figura 27 muestra la estructura final del perceptrón.

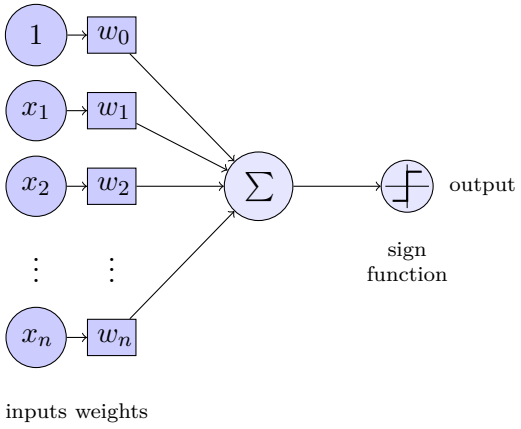


Figure 27: Estructura del perceptrón

En nuestro trabajo hemos implementado el algoritmo de aprendizaje del perceptrón (PLA), el cual nos permite ir ajustando la configuración del perceptrón, lo que quiere decir que vamos ajustando los pesos con el fin de que obtengamos el resultado que buscamos para las entradas. El algoritmo de aprendizaje del perceptrón se muestra en el algoritmo 5.

Algorithm 5: PLA

```
predictions = getPredictions(inputs);
mPoint = getRandomMisclassfiedPoint(predictions, targets);
if mPoint.empty() then
    return;
else
    for i ← 0 to N do
        weights[i] += mPoint.target*mPoint.prediction
    end
end
```

El código del PLA debería ejecutarse tantas veces como fuera necesario, y eventualmente llegaría a una solución, siempre que los datos fueran linealmente separables. Ya que no siempre van a ser linealmente separables, podemos poner un bucle y hacer tantas iteraciones como fuera necesario (a cada iteración de este bucle se le conoce con el concepto de **época**).

Algorithm 6: PLA con épocas

```
for epoch ← 1 to epochs do
    predictions = getPredictions(inputs);
    mPoint = getRandomMisclassfiedPoint(predictions, targets);
    if mPoint.empty() then
        return;
    else
        for i ← 0 to N do
            weights[i] += mPoint.target*mPoint.prediction
        end
    end
end
```

Por último, vemos el problema de que si no tenemos datos linealmente separables, las épocas no solucionan el problema totalmente (solucionan el problema de no quedarnos en un bucle infinito), ya que puede que antes de llegar a la última época, ya hayamos encontrado una solución bastante buena, pero que la hayamos perdido por el camino al actualizar los pesos, por lo que a cada época que se esté ejecutando el PLA, deberíamos de estar guardando la mejor solución hasta el momento. A esto se le conoce con el nombre del algoritmo *Pocket*.

Algorithm 7: PLA con épocas y Pocket

```
pocket = []
for epoch ← 1 to epochs do
    predictions = getPredictions(inputs);
    mPoint = getRandomMisclassfiedPoint(predictions, targets);
    if numberOfMisclassifiedPoints(pocket, targets) >
     numberOfMisclassifiedPoints(weights, targets) then
        pocket = weights
    end
    if mPoint.empty() then
        return;
    else
        for i ← 0 to N do
            weights[i] += mPoint.target*mPoint.prediction
        end
    end
end
if !pocket.empty() then
    weights = pocket
end
```

Nuestro código es muy similar al descrito, ya que la parte más importante es la que hemos explicado, la del PLA. La estructura que hemos empleado es la de una clase llamada **Perceptron** que contiene los métodos necesarios para obtener una predicción, la cual se obtiene haciendo uso del método *double getPrediction(const vector<double>&inputs)* y el PLA se ejecuta haciendo uso del método *void trainPerceptron(unsigned epochs, const vector<vector<double»&inputs, const vector<double>&targets).*

1.4 Redes neuronales

En esta sección describiremos los detalles de implementación y las bases a partir de las cuales hemos construido nuestras redes neuronales, así como sus algoritmos de aprendizaje. En concreto hemos utilizado dos métodos.

- **Aprendizaje supervisado:** Utilizando el algoritmo de **Back Propagation**.
- **Aprendizaje por refuerzo:** Utilizando un algoritmo genético.

Podemos ver la estructura básica de la red en la figura 28.

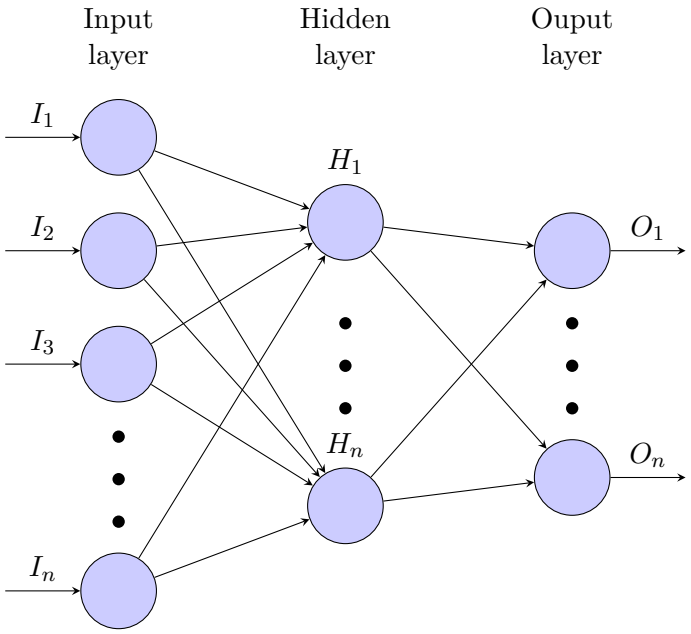


Figure 28: Estructura de una Red Neuronal

Cada nodo representa una neurona, en la figura 29 se representa el funcionamiento de cada una de ellas y se puede computar de la siguiente forma.

$$y = \sigma(\sum_{i=0}^{N} x_i * w_i)$$

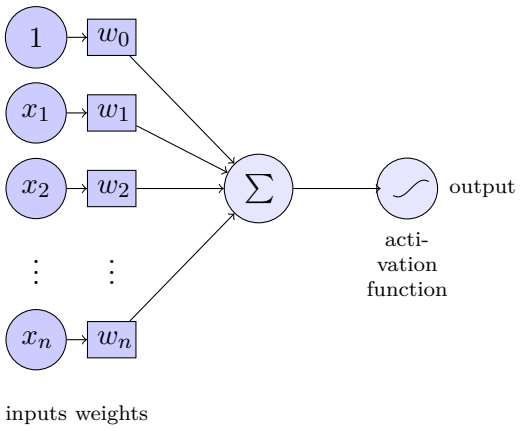


Figure 29: Estructura de la Neurona

Donde sigma representa la función de activación, x_i los inputs de la neurona y w_i los pesos correspondientes.

Su estructura es muy similar a la del perceptrón con la diferencia de que en lugar de una función de signo, su salida pasa por una función de activación (tanh, sigmoid, ReLU...) que modifica la salida en lugar de simplemente tomar una salida de +1 o -1.

Por ejemplo, la función ReLU(x) tomará 0 como salida para cualquier x con valor negativo mientras que tomará valor x para cualquier valor positivo de x.

$$ReLU = \begin{cases} 0 & \text{for } x < 0 \\ x & \text{otherwise} \end{cases}$$

A diferencia del perceptrón, una red neuronal es capaz de clasificar datos que no sean linealmente separables,combinando varias neuronas la red es capaz de aprender funciones más complejas. Debido a esta mayor expresividad, se nos presenta un nuevo problema que no existía con los clasificadores lineales, y es que ahora nuestro algoritmo es capaz de hacer **overfitting**.

El overfit es un problema común en los algoritmos de aprendizaje basado en que el algoritmo memoriza los ejemplos de entrenamiento en lugar de encontrar una solución general para otros casos que no ha visto nunca. Para evitar el overfit existen numerosas técnicas, algunas de ellas han sido implementadas y las veremos más adelante, aunque la forma más simple de reducir este fenómeno es encontrar una red con la mínima dimensión posible que sea capaz de encontrar una solución general. Es decir, es mucho más sencillo que

una recta sea una solución genérica a un problema que un polinomio de grado 8, sencillamente porque el polinomio de grado 8 tiene muchas más variables y es capaz de ser representado de muchas formas distintas. Es por esto que a la hora de utilizar una red neuronal para aprendizaje busquemos adecuar el tamaño de la red (numero de capas ocultas, numero de neuronas...) a la complejidad del problema.

1.4.1 Back Propagation

Uno de los algoritmos más comunes y utilizados para entrenar redes neuronales es **Back Propagation**. Se basa en buscar una minimización de la función de error utilizando el método de descenso por gradiente, considerando la combinación de los distintos pesos que minimizan este error una posible solución al problema.

Algorithm 8: Back Propagation

```
Inicializar los pesos con valores aleatorios normalizados entre -1 y
 +1 Feed forward pass
for layer ← 0 to layers do
    for neuron ← 0 to layerSize do
        neuron.computeOutput();
    end
end
Gradient for output layer
for neuron ← 0 to outputLayerSize do
    node.activationPrimeError(outputs, expectedOutputs);
end
Gradient for hidden layers for layer ← layers[layersSize − 1] to 0
 do
    for neuron ← 0 to layerSize do
        nextLayer = layer + 1; delta = nextLayer.getDelta(neuron)
        for nextLayerNeuron ← 0 to nextLayer do
            delta∗ = nextLayerNeuron.getWeight() ∗
             nextLayerNeuron.getDelta()
        end
    end
end
Weight update
for layer ← 0 to layers do
    for neuron ← 0 to layerSize do
        neuron.setWeight(neuron.getWeight() − learningRate ∗
         neuron.getOutput() ∗ neuron.getDelta()
    end
end
```

El algoritmo se basa en la minimización de la función de error (computada en base a los resultados obtenidos y a los resultados esperados). Obteniendo la derivada del gradiente de la función de error podemos minimizarla, haciendo así más pequeño el valor de error global de la red.

$$\nabla E = \left(\frac{\delta E}{w_1}, \dots \frac{\delta E}{w_n}, \frac{\delta E}{b} \right)$$

A la hora de actualizar los pesos introducimos el **learning rate** o ratio de aprendizaje (α), este se encarga de controlar el tamaño de los saltos en el gradiente. Si es muy elevado, dará pasos muy grandes y es posible que se pase una solución óptima, si es muy bajo, dará pasos muy pequeños y tardará mucho en alcanzar una solución. En forma matemática la actualización de pesos es la siguiente:

$$w\prime_i \longleftarrow w_i - \alpha[-(y - \hat{y})x_i]$$

A continuación, se describirán métodos de regularización que han sido implementados en esta práctica.

1.4.2 Dropout

El dropout es una técnica de regularización en la cual se seleccionan varias neuronas de forma aleatoria con una probabilidad p y se deshabilitan temporalmente junto con sus inputs y outputs. Esto implica que dichas neuronas no tendrán ningún efecto en la activación durante el **forward pass** ni tendrán efecto en la actualización de los pesos durante el **backward pass**.

Como hemos comentado con anterioridad, la gran expresividad de las redes neuronales profundas las hace propensas al overfitting, con una cantidad de datos limitada, es fácil que las neuronas aprendan patrones y hayan neuronas que dependan de sus vecinas o simplemente no tengan importancia en la red. Para evitar esto aplicamos dropout en cada época de entrenamiento.

Algorithm 9: Dropout

```
for epoch ← 0 to totalEpochs do
    for layer ← 0 to layers do
        for neuron ← 0 to layerSize do
            if dropout[neuron] < RandomDouble(0.0, 1.0) then
                neuron.disable()
            end
        end
    end
    network.train(trainingSamples)
end
```

Donde *dropout* es un vector que contiene los valores de probabilidad uniformes de que una neurona sea deshabilitada. Nuestra implementación varía ligeramente del algoritmo mostrado, ya que en lugar de deshabilitar la neurona como tal, tenemos una matriz con la misma shape que la capa que contiene valores 0 ó 1, de forma que multiplicando el output de la neurona por esta matriz la neurona queda deshabilitada.

1.4.3 Cross Validation

Cross-fold validation es una técnica para evaluar modelos predictivos mediante el particionado de el espacio de ejemplos de entrenamiento en un set de entrenamiento y otro de validación.

Este procedimiento que cuenta un único parámetro, el número k es el número de grupos en el que vamos a dividir los datos de entrenamiento. Cross validation es comunmente utilizado en modelos de machine learning para estimar la habilidad del modelo para rendir en datos que no ha visto nunca cuando contamos con datasets limitados.

Algorithm 10: Cross-Fold Validation

```
perFoldSamples = totalDataInputs/folds
for k ← 0 to folds do
    currentSampleIndex = k * perFoldSamples
    for input ← 0 to inputsSize do
        if input.belongsToK(currentSampleIndex, input, k) then
            trainingSamples.push_back(inputs[input])
        else
            validationSamples.push_back(inputs[input])
        end
    end
    NeuralNetwork net()
    net.train(trainingSamples, epochs)
    foldScore += net.getTotalError(validationSamples)
    trainingSamples.clear()
    validationSamples.clear()
end
return (foldScore/folds)
```

El algoritmo es muy simple, vamos a hacer k redes que van a entrenar sobre particiones distintas del conjunto de entrenamiento, para ello, en cada iteración dividimos los datos en dos grupos, uno de entrenamiento y otro de validación, una vez hecho esto, entrenamos la red con el grupo de entrenamiento y obtenemos el error con el grupo de validación. Una vez acabado obtenemos la media de las K iteraciones.

1.4.4 Algoritmo Genético (GANN)

Los algoritmos genéticos pertenecen a la categoría de los algoritmos evolutivos, los cuales intentan optimizar algún problema basándose en la biología, basándose en como la naturaleza ha evolucionado a lo largo de miles de años e intentando utilizar la misma técnica. Concretamente, los algoritmos genéticos intentan imitar la estructura genética de los individuos y como esta estructura va evolucionando de una generación de individuos a la siguiente, esa es la idea principal de dichos algoritmos. Antes de pasar a explicar la combinación de los algoritmos genéticos con las redes neuronales vamos a explicar las ideas principales de los algoritmos genéticos.

Como hemos dicho, los algoritmos genéticos se basan en la evolución de la estructura genética de los individuos y como se transmite esta herencia genética de una generación a la siguiente. Hablando sobre esta estructura, todo **individuo** vivo tiene una serie de **células** y cada una de estas células contiene el mismo conjunto de **cadenas de ADN**, lo que se conoce como **cromosomas**. El ADN que contiene un cromosoma es una doble cadena en forma de hebra, y las hebras están conectadas entre sí en una **doble espiral**. Finalmente, esta doble espiral está formada por pequeños bloques básicos, llamados **genes**, que a su vez están formados por sustancias llamadas **nucleótidos**. Solo hay 4 tipos de nucleótidos: timina, adenina, guanina y citosina. En la figura 30 se puede apreciar lo que hemos comentado de una manera más gráfica. Esta es la estructura básica de la información genética de los seres vivos.

Ahora sabemos un poco más cuál es la estructura básica, pero aún no hemos explicado qué es lo que contienen estas partes, lo cual va a ser lo importante para poder comprender correctamente el funcionamiento biológico y así poder hacer una buena aproximación con los algoritmos genéticos a este enfoque de la evolución respecto a la herencia genética. Lo importante es comprender que la información que hay dentro de una célula (conjunto de cadenas de ADN, los cromosomas) es lo que contiene toda la información que define a un individuo y, por lo tanto, tiene toda la información que podría pasarse al individuo de la siguiente generación (normalmente de forma parcial, no total). Esta colección de cromosomas (información genética dentro de la célula) es lo que se conoce como **genoma**, que no es más que la colección de genes que definen las características del individuo (e.g. ojos marrones, pelo liso, color del pelo, etc.). Volviendo a los **genes**, la configuración que cada gen puede tener es lo que da rasgo a alguna característica concreta, y a esto se le llama **alelo** (característica concreta del individuo), y la posición física donde está el gen a lo largo de la cadena del cromosoma

se le llama **locus**.

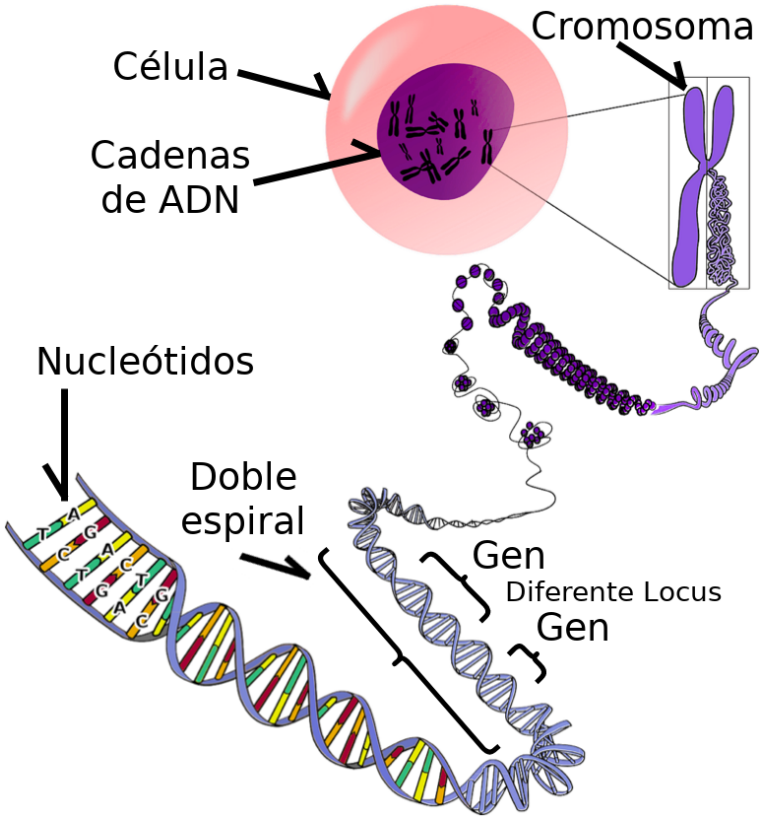


Figure 30: Estructura básica del ADN

Ahora bien, el estado de los alelos en un genoma particular se conoce como **genotipo** (información genética que da lugar a las características del individuo). El genotipo es una información que una vez está definida no cambia. Al organismo en sí, al individuo, a la expresión de ese genotipo en el mundo real, se le conoce como **fenotipo**. Por poner un ejemplo, el ADN que tiene cada ser humano sería el genotipo propio de cada uno, mientras que la persona en sí, el como se expresa en el mundo físico, sería su fenotipo, el cual, además, está influido por el ambiente. En la figura 31 se ve esta diferencia claramente, que es lo que se conoce como genotipo vs fenotipo.

Observando la figura 31, podemos darnos cuenta de algo, y es que aunque tengamos 2 copias exactas a nivel genético de un individuo (i.e. clones), lo que significaría que tienen el mismo genotipo, podrían no tener el mismo fenotipo, ya que el ambiente importa, aunque respecto a los algoritmos

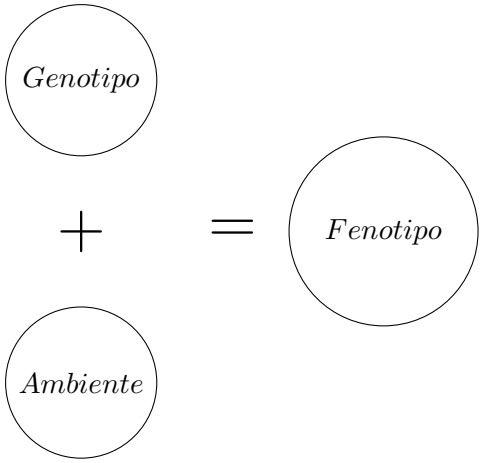


Figure 31: Genotipo vs Fenotipo

genéticos esto se obvía normalmente, al no ser que de alguna forma el individuo evolucione mientras está en vida (nos referimos a algoritmos, no a seres vivos). Si tuviéramos un algoritmo genético que combinado con otra técnica, el individuo evolucionase a la vez que se ejecuta entonces podríamos hablar de que el entorno importa (e.g. algoritmo genético que resuelve un camino en un laberinto y aprende a la vez que lo recorre, pero además el laberinto varía, por lo que un individuo podría aprender muy bien a resolverlo, pero otro individuo con el mismo genotipo tiene un laberinto complicado, por lo que no aprende bien, así que el fenotipo varía).

Una vez hablado de todo lo anterior, lo que es la parte biológica, necesario para comprender correctamente el funcionamiento de un algoritmo genético, podemos pasar a explicar como funciona el algoritmo en sí. Los algoritmos genéticos lo que intentan es aproximar el enfoque de la evolución desde el punto de vista de la herencia genética: de algún modo codifican el genotipo del individuo, de algún modo se evalúa lo bien que hace su tarea para sobrevivir, para tener más posibilidades de pasar parte de su información genética a la siguiente generación, y se hace evaluando su fenotipo. En definitiva, los algoritmos genéticos se basan en los principios de Charles Darwin de su teoría de la selección natural. Resumidamente es la supervivencia del más apto.

Las fases principales de los algoritmos genéticos son:

1. Crear una **población inicial** de individuos. Esta población suele generarse de forma aleatoria y, aunque muchos individuos harán mal

su tarea, otros lo harán medio bien o incluso bien. También se pueden inicializar siguiendo una distribución de probabilidad que se sabe que sigue el patrón de los individuos que hacen bien su tarea.

2. Calcular su ***fitness***. Este valor se calcula de la función de *fitness*, la cual nos indica como de bien un individuo hace la tarea para la que se supone que ha sido concebido. El objetivo suele ser maximizar esta función.

3. **Selección**. Seleccionar los individuos más aptos para pasar su información a la siguiente generación. Esto es una cuestión de probabilidad, no de directamente hacer que los mejores individuos sean los que pasen su información a la siguiente generación. Los mejores individuos tendrán más posibilidades, pero los peores también tendrán una pequeña posibilidad de pasar su información a la siguiente generación.

4. ***Crossover* (emparejamiento)**. Unir la información genética de los individuos seleccionados para pasar a la siguiente generación.

5. **Mutación**. Las mutaciones son algo que sucede cuando hablamos de la genética. A veces son buenas y a veces son malas, pero está claro que son necesarias. La mutación puede aportarnos algo que puede que ningún individuo de la población inicial tuviera en su información genética, y esa mutación podría hacer que un nuevo gen surgiera y predominara por encima de todos.

La estructura principal de los algoritmos genéticos se muestra en el algoritmo 11.

Algorithm 11: Estructura general de los algoritmos genéticos

```
population = createNewPopulation();
while true do
    fitness = evaluatePopulation(population);
    newPopulation = [];
    for individual ← 1 to population do
        parents = doSelection(population, fitness);
        newIndividual = doCrossover(parents);
        newIndividual = doMutation(newIndividual);
        newPopulation[individual] = newIndividual;
    end
    population = newPopulation;
end
```

Conforme se vaya pasando de generación en generación, los nuevos individuos irán mejorando, ya que han ido obteniendo los mejores genes de sus antepasados o al menos esa es la idea principal de estos algoritmos. Dentro de estos algoritmos luego hay muchos aspectos que controlar para que estos funcione como es debido, aunque lo básico ya está expuesto. Ahora pasaremos a hablar por qué pueden ser útiles a la hora de combinarlos con las redes neuronales.

Las redes neuronales forman parte de los algoritmos de aprendizaje supervisado, lo que quiere decir que necesita de datos que previamente estén etiquetados. Con estos datos utiliza el algoritmo de aprendizaje *back propagation* (propagación hacia atrás), el cual es el mejor algoritmo de aprendizaje hasta la fecha para las redes neuronales para el aprendizaje supervisado, pero no el único. La idea de juntar los algoritmos genéticos con las redes neuronales (GANN) es la de no utilizar el algoritmo de aprendizaje *back propagation*, sino que en su lugar utilizar el propio algoritmo genético como algoritmo de aprendizaje para la red neuronal. Como hemos dicho, el algoritmo de aprendizaje *back propagation* es el mejor que se conoce a día de hoy para el aprendizaje supervisado en las redes neuronales, así que ¿cuál es el objetivo de utilizar un algoritmo genético como algoritmo de aprendizaje? Hay varias razones, y una de las principales razones es que al utilizar un algoritmo genético como algoritmo de aprendizaje para las redes neuronales dejamos de estar utilizando aprendizaje supervisado y pasamos a estar utilizando **aprendizaje por refuerzo**.

El aprendizaje por refuerzo no necesita de datos etiquetados como sí lo necesita el aprendizaje supervisado. Este en su lugar necesita una medida de como de bien está comportándose el algoritmo, lo cual encaja a la perfección con la función de *fitness*, que es justamente lo que hace. La idea es codificar de alguna manera las redes neuronales de manera que un algoritmo genético pueda tratarlas, modificarlas, etc. Tenemos que conseguir el genotipo de la red neuronal, y la forma de hacerlo es con la codificación de la red. Hay muchos tipos de codificaciones, siendo la más típica la **codificación binaria**, la cual coge N bits para representar un peso de la red neuronal y entonces ya ejecuta todas las fases que hemos comentado. Esta codificación es la más típica, pero también es muy restrictiva y nos aporta poca flexibilidad. En nuestro caso, la codificación que hemos hecho de la red neuronal es una codificación de **números reales**, en la cual cada peso de la red se trata como lo que es, un número real. Esta codificación también trae problemas, siendo uno de ellos el límite superior e inferior que se le da a los pesos, ya que en función de esos límites podremos generar unos pesos en un rango o en otro.

Por la forma en la que tenemos nuestra implementación de la red neu-

ronal, en la cual hemos implementado el perceptrón multicapa, la manera más sencilla para representar la red, para codificarla, era coger nuestras matrices de pesos y aplanarlas para obtener un vector. Respecto a los *biases* hicimos lo mismo, coger la matriz que contiene todos los *biases*, aplanarla, y añadir todos los pesos al final del vector que contiene los pesos. Como luego es necesario decodificar y volver a obtener nuestras matrices de pesos y *biases*, con el vector de topología, que es el vector que nos indica cual es la topología de esa codificación en concreto, se puede volver a desaplanar sin ningún problema. El aplanamiento y unión de matrices de pesos y *biases* es sencillo, pero el desaplanar puede no serlo tanto, así que se puede observar el código en el algoritmo 12.

Algorithm 12: Desaplanamiento de las matrices de pesos y *biases*

```
Data: const vector<int>&topology, const Mat &flattened
Result: vector<vector<Mat»weightsAndBiases
vector<vector<Mat»res;
vector<Mat>weights;
vector<Mat>bias;
int flattenedCol = 0;
for (size_t i = 0; i < topology.size() − 1; i + +) do
    Mat matWeights(topology[i], topology[i + 1]);
    for (int row = 0; row < matWeights.rows(); row + +) do
        for (int col = 0; col < matWeights.cols(); col + +) do
            matWeights.set(row, col, flattened.get(0, flattenedCol));
            flattenedCol + +;
        end
    end
    weights.push_back(matWeights);
end
for (size_t i = 0; i < topology.size() − 1; i + +) do
    Mat matBias(1, topology[i + 1]);
    for (int col = 0; col < matBias.cols(); col + +) do
        matBias.set(0, col, flattened.get(0, flattenedCol));
        flattenedCol + +;
    end
    bias.push_back(matBias);
end
res.push_back(weights); res.push_back(bias);
return res;
```

Una vez ya desaplanados los pesos y *biases* ya solo nos quedaba crear

una nueva red neuronal y utilizar *setWeights()* y *setBiases()*.

Ahora vamos a pasar a la parte más importante de nuestra implementación de la GANN, que son los operadores genéticos que hemos empleado. Los operadores genéticos son las operaciones que realizamos para obtener la selección, el emparejamiento y la mutación de los individuos.

Para la selección de los individuos hay varios operadores genéticos, siendo los más famosos los siguientes:

1. Selección proporcional del *fitness* (Fitness proportionate selection) más conocido con el nombre de Roulette Wheel Selection. Este operador de selección consiste en asignarle a cada individuo una probabilidad basada en su *fitness* respecto a la suma total del *fitness* de toda la población. De esta manera, luego se simula la rotación de una ruleta, donde los que mayor probabilidad tienen son los que mayor sector tienen en la ruleta. Es el operador de selección más utilizado y es el que hemos utilizado en nuestra implementación. La fórmula que utiliza es la siguiente:
$$p_i = \frac{f_i}{\sum_{j=1}^{N} f_j}$$

2. Selección por torneo. Este método intenta simular un torneo, donde la probabilidad de selección varía a cada iteración. Cuando se han ejecutado los torneos que se haya configurado, es cuando se escoge a los mejores individuos (donde K es el número de individuos que participan en el torneo). La fórmula que utiliza es la siguiente:
$$p_0 = p$$
$$p_{iteration} = p * (1 - p)^{iteration}$$

3. Selección basada en recompensas. Este método recomensa a los individuos que han hecho algo de forma correcta. Hay muchas fórmulas que pueden ser utilizadas como recomensa, una básica es la siguiente:
$$individual^{(generation+1)} selected \Rightarrow reward^{(generation)} = 1$$

Respecto al emparejamiento de individuos, hay muchas maneras de hacerlo. Nosotros utilizamos 2 enfoques diferentes, y en ambos era para el emparejamiento de 2 individuos, es decir, de 2 redes neuronales. En el primero de ellos lo que hacíamos era alternar entre coger un peso de una red y luego otro de la otra, con lo que acabábamos con una nueva red con

cerca del 50% del genotipo de las 2 redes que se utilizaban como padres. El segundo enfoque lo que hacíamos era coger los 2 pesos de la misma posición de los padres y hacer una media ponderada, lo cual nos daba más seguridad de que los resultados obtenidos en la función de *fitness* de la nueva red neuronal iban a ser más próximos a los de las redes padre, ya que si ambas redes neuronales padre obtenían buenos valores de *fitness* la probabilidad de que sus pesos se parecieran es muy alta, como veremos que sucede en los experimentos realizados. Esta último enfoque es el que se ha quedado en la

última versión del GANN. Podemos ver ambos enfoques en el algoritmo 13.

Algorithm 13: Operadores genéticos de emparejamiento empleados

```
Data: const DNA &a, double ownFitness, double aFitness
Result: DNA newIndividual
DNA res(a.genes.rows(), a.genes.cols());
double random = UtilG :: getRandomDouble(0.0, 1.0);
if random <= this- > crossoverRate then
    res.setMutationRate(this- > mutationRate);
    res.setCrossoverRate(this- > crossoverRate);
    if a.genes.rows() == this- >
     genes.rows()&&a.genes.cols() == this- > genes.cols() then
        for (int row = 0; row < a.genes.rows(); row + +) do
            for (int col = 0; col < a.cols(); col + +) do
                // First approach
                if (row + col)%2 == 0 then
                    res.genes.set(row, col, this- >
                     genes.get(row, col));
                else
                    res.genes.set(row, col, a.genes.get(row, col));
                end
                // Second and definitive approach. Average
                res.genes.set(row, col, (a.genes.get(row, col) +
                 this- > genes.get(row, col))/2.0);
            end
        end
    end
else
    if ownFitness > aFitness then
        res = *this;
    else
        res = a;
    end
end
return res;
```

Por último, al igual que con los operadores genéticos de emparejamiento, hay muchas posibilidades para los operadores genéticos de mutación. En nuestra implementación aplicamos 2 tipos de mutaciones. La primera mutación que aplicamos es cambiar un número real por otro, por defecto en el rango [-1.0, 1.0], aunque es posible multiplicar dicho rango por factor,

dando lugar a [-1.0*factor,1.0*factor]. El otro operador genético de mutación que hemos empleado ha sido la permutación. La implementación de ambos operadores se muestra en el algoritmo 14.

Algorithm 14: Operadores genéticos de mutación empleados

```
// First mutation. Replace a real − number
for (int row = 0; row < this− > genes.rows(); row + +) do
    for (int col = 0; col < this− > genes.cols(); col + +) do
        double random = UtilG :: getRandomDouble(0.0, 1.0);
        if random <= this− > mutationRate then
            this− > genes.set(row, col, UtilG ::
              getRandomDouble((double)factor ∗
              −1.0, (double)factor ∗ 1.0));
        end
    end
end
// Second mutation. Permute 2 weights/biases
for (int row = 0; row < this− > genes.rows(); row + +) do
    for (int col = 0; col < this− > genes.cols(); col + +) do
        double random = UtilG :: getRandomDouble(0.0, 1.0);
        if random <= this− > mutationRate then
            int randomRow1 = rand()%this− > genes.rows(); int
              randomCol1 = rand()%this− > genes.cols(); int
              randomRow2 = rand()%this− > genes.rows(); int
              randomCol2 = rand()%this− > genes.cols(); double
              value1 = this− >
              genes.get(randomRow1, randomCol1); double value2 =
              this− > genes.get(randomRow2, randomCol2);
            this− > genes.set(randomRow1, randomCol1, value2);
              this− > genes.set(randomRow2, randomCol2, value1);
        end
    end
end
```

Una vez ya hemos visto los operadores genéticos que hemos empleado solo nos queda por ver dos cosas más: la función de *fitness* y otras técnicas que hemos implementado para evitar la rápida convergencia o para evitar la pérdida de los individuos más aptos.

Vamos a empezar por las funciones de *fitness* que hemos empleado, las cuales van ligadas al problema que queremos resolver, y en este caso el problema que queremos resolver es enseñar a un algoritmo a jugar lo mejor posible a los 4 videojuegos para la videoconsola Atari 2600 que comentamos al inicio (i.e. breakout, boxing, demon attack y stargunner) de esta memoria. Las funciones de *fitness* que hemos empleado han ido cambiando mucho, ya que cuando utilizábamos una u otra, veíamos que los resultados variaban, y mucho. A continuación, mostramos las funciones de *fitness* que hemos empleado para cada uno de los videojuegos.

1. Breakout:
   (a) $fitness = score$; // Ha funcionado a la perfección desde el principio.
2. Boxing:
   (a) $fitness = score$; // Se quedaba quieto todo el rato y aprendia a dar golpes de vez en cuando.
   (b) $fitness = score + (double)scoreP1 * ((double)scoreP1/(double)(scoreP2+1))$; // Intentábamos recomensar a P1 cuando la diferencia de resultados no fuera exagerada a favor de P2 y que, cuando mejor lo hiciera P1, más se le recompensaba. Obtuvo mejores resultados que el anterior, llegando a obtener puntuaciones igualadas, pero sin muy buenos resultados.
   (c) Algoritmo 15. Se mueve hacia la derecha y acorrala todo el rato al oponente, consiguiendo doble de puntos. Los resultados suelen ser de 100 a 10 ganando P1. A pesar de obtener unas puntuaciones muy buenas no parece que haya aprendido muy bien a jugar: solo avanza y golpea.
3. Demon Attack:
   (a) $fitness = score$; // Obtenía unas puntuaciones de 3000, las mejores que hemos obtenido, pero la jugabilidad por parte del bot era pésima: se quedaba en el centro y disparaba.

(b) $fitness = score * ((moveLeft/100.0) * (moveRight/100.0))$; // Se le recompensaba al bot cuando más se moviera. El problema era que aprendia a matar a algún enemigo muy esporádicamente a moverse todo el rato.

(c) $fitness = ((double)(score^2)/(500.0^2)) * ((moveLeft/1000.0) * (moveRight/1000.0))$; // Intentamos arreglar el problema anterior penalizando un poco el moverse y aumentando más la recompensa por disparar y matar a los enemigos. Factores de multiplicación muy desbalanceados. Muy malos resultados.

(d) $fitness = ((double)(score^2)/(500.0^2)) * ((moveLeft/1000) * (moveRight/1000))*((score/1000)+1)$; // Intentamos que la puntuación no fuera un factor extremadamente determinante como en la anterior función. Resultados aceptables en cuanto a jugabilidad.

(e) $fitness = (score/10) * ((moveLeft/1000) * (moveRight/1000))$; // Intento de simplificar todo lo anterior. Sin éxito.

(f) $fitness = (score/10)*((moveLeft/(int)((steps*500.0)/15000.0)) *(moveRight/(int)((steps * 500.0)/15000.0)) * (shooting/(int)((steps*1000.0)/15000.0)))$; // Intento de volver a la compleja función de antes pero esta vez recompensando el que bot disparara, sin importar si mataba. El bot aprendía a disparar y moverse todo el rato en el punto ciego del juego (lago izquierdo normalmente). Los movimiento que hacía eran izquierda - derecha constantemente, con lo que maximizaba todos los valores y de vez en cuando mataba a algún enemigo. Se terminaba el tiempo y obtenía un *fitness* muy elevado y una puntuación muy baja. Además, intentamos que el *fitness* obtenido fuera en función de lo que había tardado en jugar el bot.

(g) $fitness = (score/100)^2 * ((moveLeft + moveRight)/1000) * (shooting/2000) * (min(moveLeft, moveRight)/100)$; // Intento de obtener un mínimo en los disparos y en el movimiento y a la vez simplificar la función. Malos resultados.

(h) $fitness = (score/10)*(shooting/500)*(min(moveLeft, moveRight)/100)$; // Intento de mejorar la función anterior restándole importancia a la puntuación e intentnado obtener un valor mínimo en el movimiento y en los disparos. Resultados decentes: cerca de los 1000 puntos.

(i) $fitness = (score/10)^2 * (min(moveLeft, moveRight)/100)$; //

Simplificación de la fórmula. Al elevar al cuadrado la puntuación, forzábamos a que el bot disparara sin tener que poner explícitamente los disparos en la función. Mejores resultados que la fórmula anterior.

(j) Algoritmo 16. Con esta función de *fitness* le pasamos un nuevo valor que calculamos gracias a las coordenadas X de los enemigos y hacemos que cuando siga al enemigo que se le indica, se le recompense. Muy buenos resultados: sigue al enemigo más cerca vivo en todo momento (si ya no aparece el más cercano, el siguiente objetivo será el de en medio, y luego el final. Cuando empieza la fase de las moscas, va buscando cual es la mosca más cercana viva, teniendo en cuenta la mosca que se acerca al jugador como si fuera una bala), intentando eliminar al enemigo más cercano, ya que es el más fácil de dispararle y acertar. El problema es que no aprende muy bien a evitar los disparos del enemigo más cercano.

4. StarGunner:

   (a) $fitness = score$; // No muy buenos resultados, ya que la puntuación al ser de 100 en 100 al eliminar un objetivo, cuando un individuo eliminaba a un objetivo más que otro, su probabilidad de ser seleccionado era muy elevada aunque el bot fuera mucho más malo.

   (b) $fitness = score/100$; // Intento de mejorar el problema anterior. El bot juega bastante bien: en algunas ocasiones se queda quieto pero en otras empieza a seguir a los objetivos, consiguiendo unas puntuaciones de 3000.

Algorithm 15: Mejor función de *fitness* para boxing encontrada

```
if score < 0 then
    fitness = score * (1 -
      (double)moveLeft/(double)steps) * (1 -
      (double)moveUp/(double)steps) * (1 -
      (double)moveRight/(double)steps) * (1 -
      (double)moveDown/(double)steps) + (double)scoreP1 *
      ((double)scoreP1/(double)(scoreP2 + 1));
else
    fitness = score * (
      (double)moveLeft/(double)steps) *
      ((double)moveUp/(double)steps) *
      ((double)moveRight/(double)steps) *
      ((double)moveDown/(double)steps) + (double)scoreP1 *
      ((double)scoreP1/(double)(scoreP2 + 1));
end
```

Algorithm 16: Función de *fitness* del Demon Attack que sigue al enemigo

```
if min(moveLeft, moveRight)! = 0 then
    double punch = (double)inRangeOfEnemy/(double)steps;
    double dscore = (double)score;
    punch* = punch;
    dscore/ = 10;
    fitness = dscore * punch;
else
    fitness = 0.0;
end
```

Una vez ya comentadas las funciones de *fitness* que hemos probado nos falta comentar las otras técnicas que hemos implementado. Otras técnicas que hemos implementado en pro del GANN han sido las siguientes:

- Elitismo. El elitismo es una técnica muy sencilla la cual nos permite no perder los mejores individuos de una determinada generación. Consiste en coger el mejor individuo de una generación y pasarlo directamente a la siguiente generación N veces. Un elitismo de 0 significa que no hay elitismo.

- Inicialización de Elitismo. El elitismo se aplica a partir de la segunda generación, pero no en la primera. Un intento de una rápida convergencia, cosa que casi nunca es buena en los genéticos, ha sido nuestra implementación de utilizar elitismo en la creación de la población. Por sí solo no funciona muy bien, ya que para funcionar bien debería de generarse individuos con los que pudiera hacer un emparejamiento lo suficientemente bueno y para ello normalmente los individuos deben parecerse. Ahora bien, cuando se utiliza junto con una distribución de probabilidad obtiene buenos resultados, pero solo debería de aplicarse a problemas sencillos, ya que los complejos necesitan de una convergencia más lenta, donde haya más diversidad.

- Técnicas de nicho. Estas técnicas intentan que la diversidad de la población aumente y lo hace castigando a aquellos individuos que se parezcan mucho y premiando a aquellos individuos que aporten novedad a la población. La fórmula que utiliza para ello es la siguiente:
$$fitness_i = \frac{fitness_i}{\sum_{j=1}^{N} individual_i \simeq individual_j ? 1 : 0}$$

- Creación de población por rango de valores (distribución de probabilidad). Inicialmente se generan pesos y *biases* aleatorios entre un rango, pero si supiéramos la distribución de probabilidad que sigue cierto problema, una buena inicialización podría hacer que obtuviéramos mejores resultados antes. El problema es que hay que ir obteniendo buenos resultados poco a poco y, entonces, ir utilizando esa distribución para obtener buenos resultados. En los experimentos vemos un ejemplo de esto.

Tanto el elitismo, la inicialización con elitismo y la inicialización con rangos son sencillos de entender, por lo que no vamos a mostrar la implementación aquí en la documentación. En cambio, vamos a explicar qué criterio hemos utilizado para decir que dos individuos son similares. El criterio que hemos utilizado ha sido el de asignar a cada individuo un id de nicho, que es lo que se suele hacer con estas técnicas. El id del nicho que se asigna a cada individuo depende de la puntuación y de los pasos realizados al jugar alguno de los videojuegos. Dependiendo de esos dos valores, y del valor máximo de puntuación de toda la población de una generación concreta y el valor máximo de pasos de toda la población de la misma generación y ponemos los puntos obtenidos en el plano. Entonces dividimos el eje X en 10 regiones, y hacemos los hacemos lo mismo con el eje Y. Cada región de las 100 obtenidas es un id de nicho diferente. En la figura 32 se muestra lo que estamos explicando poniendo como ejemplo el videojuego Breakout (suponiendo que los pasos son 7500 y están en el eje Y y la puntuación máxima obtenida por algún individuo ha sido la máxima, la cual es 864).

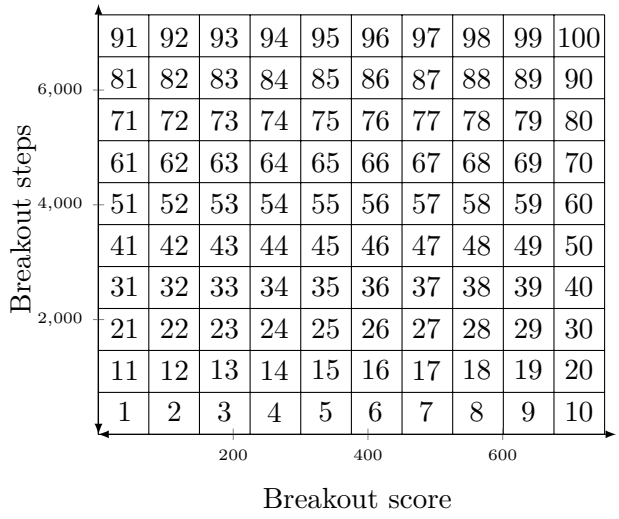


Figure 32: ID de nicho en Breakout

2 Manual de utilización

Antes de empezar con el manual de utilización para las tecnologías concretas desarrolladas hay que tener en cuenta dos cuestiones fundamentales:

1. La tecnología que hemos desarrollado se implementado bajo entorno Linux, concretamente todo está probado en la distribución *Manjaro Linux*. No se garantiza el correcto funcionamiento de la tecnología en otros sistemas.

2. La tecnología que hemos desarrollado utiliza el entorno ALE (*Arcade Learning Environment*), por lo que lo que es necesario para el funcionamiento de los experimentos y de la tecnología en sí.

Respecto a ALE, es necesario saber dónde está instalado, por lo que hemos utilizado la variable de entorno del sistema **ALEPath** para localizar el entorno y poder compilar y utilizar la tecnología sin problema. La manera de indicar donde está ALE, si por ejemplo estuviera en la ruta /home/UDLearn/Escritorio/ALE, sería con el siguiente comando desde la terminal: *export ALEPath=/home/UDLearn/Escritorio/ALE*. De esta manera ya estaría todo listo para poder compilar la tecnología desarrollada y utilizarla.

Para poder utilizar la tecnología solo es necesario compilar, lo cual es sencillo ya que solo habrá que ejecutar *make* en la terminal, ya que cada una de las tecnologías desarrolladas viene con su correspondiente fichero *makefile*.

Una vez ya se ha indicado donde está ALE y compilada la tecnología que se quiera utilizar, en los siguientes apartados se indica el uso concreto de cada una de ellas.

2.1 Bots y bots naive

Una vez que la variable de entorno está aplicada correctamente ya se pueden compilar y utilizar los bots. Dentro del directorio principal de UDLearn encontraremos varias carpetas con los nombres de los juegos. Dentro de cada una de éstas carpetas (excluyendo el strgunner), podemos encontrar un subdirectorio llamado "naive". Este subdirectorio representa la versión de Inteligencia Artificial simple realizada para ese juego en concreto, el directorio raíz contendrá diversos archivos específicos a la implementación mediante redes neuronales.

Para compilar los bots naive, haremos lo siguiente:

1. Abrir la consola en el directorio del bot naive deseado (ejemplo dattack/naive).
2. Dentro del directorio, escribir en la consola "make" y pulsar intro, esto compilará los ficheros involucrados.

Una vez hemos compilado el fichero, solo queda ejecutarlo. Para esto, desde la misma consola que hemos compilado el bot, podremos escribir lo siguiente:

- *./nombre_ejecutable nombre_rom.* Este modo ejecutará el bot sin ningun tipo de información multimedia disponible.
- *./nombre_ejecutable nombre_rom 1.* Si queremos habilitar o deshabilitar explícitamente el contenido multimedia (audio y vídeo), tendremos que escribir a continuación un 1 o un 0, respectivamente.
- *./nombre_ejecutable nombre_rom 1 1.* Si queremos habilitar o deshabilitar además la visualización de RAM, tendremos que escribir a continuación un 1 o un 0 respectivamente.
- Cualquier opción extra no común no reflejada en la documentación aparecerá en el ejecutable siempre y cuando no se introduzca el input correctamente.

2.2 Perceptrón

Para el perceptrón tenemos la posibilidad de ejecutar los experimentos que hemos implementado y están documentados en la sección 3.1. El volver a ejecutarlos actualizará los pesos con otros diferentes (se utilizan datos aleatorios para la generación de los puntos).

Hay un total de 5 experimentos, los cuales se codifican con un número. Tanto la codificación como la explicación de cada uno se describe:

1. Experimento 1 (cod. = 1): se intenta que el perceptrón aprenda la función AND.
2. Experimento 2 (cod. = 2): se intenta que el perceptrón aprenda la función y = 0'5.
3. Experimento 3 (cod. = 3): se intenta que el perceptrón aprenda la función y = 0'5 * x.
4. Experimento 4 (cod. = 4): se intenta que el perceptrón aprenda la función y = -0'5 * x + 0.5.
5. Experimento 5 (cod. = 5): se intenta que el perceptrón aprenda la función y = 1/(10*x) la cual **no** es lineal.

Al igual que el resto de tecnologías, solo habrá que ejecutar *make* para obtener el fichero con el que se pueden ejecutar la tecnología (en caso de querer agregar nuevos experimentos habrá que entrar al cógido y modificarlo). Los posibles comandos se describen a continuación:

1. De forma general:
   (a) ./main experimento [puntos épocas mensajes]
2. Ejemplo 1: experimento 1 (función AND):
   (a) ./main 1
3. Ejemplo 2: experimento 2 (y = 0.5) con 200 puntos, 100 épocas y activando mensajes:
   (a) ./main 2 200 100 1

Por defecto, el experimento 1 tiene 20 épocas asignadas y no se pueden cambiar. Esta decisión es debido a que la función AND suele obtener un acierto del 100% en menos de 10 épocas. El resto de experimentos tienen asignado por defecto 40 puntos y 200 épocas y se puede modificar como se ha visto en los ejemplos. Para el experimento 1 se muestran todos los mensajes por defecto, ya que la cantidad no es excesiva y ayuda. Por el contrario, para el resto de experimentos por defecto los mensajes están desactivados, pero pueden activarse como se ha visto en los ejemplos.

2.3 Red Neuronal

La red neuronal se encuentra en el directorio NeuralNetwork y contiene un makefile para compilar y un fichero *main.cpp* con 4 experimentos. Para construir y entrenar una red neuronal simplemente hay que crear un vector con la topología (numero de neuronas en cada capa) por ejemplo, $\{8, 4, 2\}$ si queremos una red neuronal con 8 neuronas de input, una capa oculta con 4 neuronas y 2 neuronas de output. El constructor por defecto recibe este vector de topología y nos devuelve una red. Existe un segundo constructor que además de el vector de topología recibe un vector de dropout, este tiene que tener el mismo tamaño que el de topología y valores de probabilidad de dropout para cada capa, comprendidos entre 0.0 y 1.0. Por ejemplo siguiendo el caso anterior, $\{0.0, 0.2, 0.0\}$ añadiría una probabilidad del 20% de que las neuronas de la capa oculta queden deshabilitadas.

Una vez creada la red, podemos entrenarla con el método de clase $train()$ que recibe dos vectores de Matrices (clase $Mat.cpp$) uno contiene los ejemplos de entrada para el entrenamiento y el otro los outputs esperados. Cada matriz del vector es un ejemplo de entrenamiento que debe coincidir con la shape de la red neuronal.

Podemos inicializar la red con pesos con el método $setWeights()$ que recibe como argumento un vector de matrices ($Mat.h$) con la misma shape que la red en la que cada matriz contiene los pesos para cada neurona de la capa correspondiente.

El fichero *main.cpp* contiene 5 ejemplos de ejecución.

1. Experimento 1: Función AND
2. Experimento 2: Función XOR
3. Experimento 3: CrossValidation
4. Experimento 4: Dropout

Para reproducirlos sencillamente hay que compilar con *make* y ejecutar *./main experimento* donde *experimento* es un entero de con valor de 1 a 4.

2.4 Red Neuronal y Genético: *GANN*

La utilización del GANN es muy sencilla. Lo primero que habría que hacer es compilar, lo cual haciendo *make* desde la terminal ya lo tendríamos (consultar sección 2 para más detalles). A continuación, se detallan los modos de uso.

Con el GANN tenemos 2 modos de funcionamiento:

1. Modo de **entrenamiento**: en este modo lo que se hace es utilizar el GANN para obtener resultados entrenando a cualquiera de los 4 juegos disponibles. Mientras se ejecuta se va mostrando la información del progreso y se va guardando un registro con la información más importante. Cuando termina de ejecutarse, guarda los pesos de la mejor red encontrada.

2. Modo de **ejecución**: en este modo lo único que se hace es ejecutar la mejor red que hemos encontrado conforme hemos ido haciendo pruebas. Los pesos están directamente escritos en el código, así que lo que se verá será directamente el juego elegido y el mejor bot que hemos encontrado jugando.

Para seleccionar el juego, la codificación que hemos escogido es un número para cada juego. La codificación es la siguiente:

- Breakout = 1
- Boxing = 2
- Demon Attack = 3
- StarGunner = 4

La ejecución de ambos modos es muy sencilla, y se describe a continuación:

1. Modo de Entrenamiento:
   (a) De modo general:
      i. *./main juego generaciones población nombreFicheroRegistros*
   (b) Ejemplo: Breakout con 5 generaciónes, 100 individuos en la población y archivo llamado "breakoutRecords.txt" (registro) y archivo "breakoutRecords.weights" (pesos).

i. *./main 1 5 100 breakoutRecords*

2. Modo de ejecución:

   (a) De modo general:

      i. *./main juego visualización*

   (b) Ejemplo: ver el bot del Breakout sin visualización.

      i. *./main 1 0*

   (c) Ejemplo: ver el bot del Boxing con visualización.

      i. *./main 2 1*

   (d) Ejemplo: ver el bot del Demon Attack sin visualización.

      i. *./main 3 0*

   (e) Ejemplo: ver el bot del StarGunner sin visualización.

      i. *./main 4 0*

3 Experimentos realizados y resultados obtenidos

3.1 Experimentos con el Perceptrón

El primer experimento que hemos realizado con el perceptrón ha sido sencillo, con el objetivo de comprobar que el funcionamiento era el esperado y, sobre todo, ver gráficamente como el PLA iba evolucionando los pesos y cual iba siendo la evolución. La función que hemos intentado aproximar con el perceptrón ha sido la función AND, la cual se describe en el algoritmo 17.

Algorithm 17: Función AND

```
if a * b == 1 then
    return 1;
else
    return 0;
end
```

La función AND es linealmente separable, por lo que el perceptrón debería de poder separar los puntos eventualmente. En la figura 33 se muestran los puntos que queremos separar (como subíndice tienen el valor que se espera por el perceptrón).

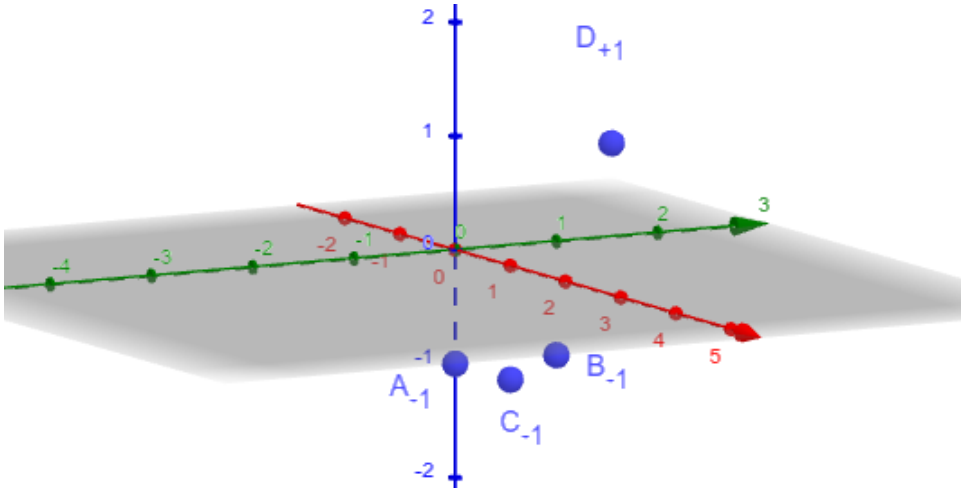
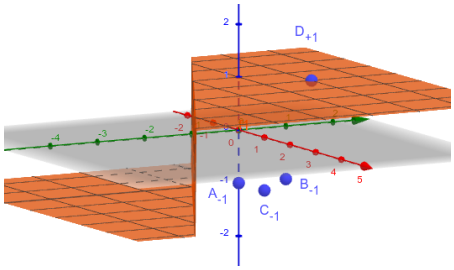
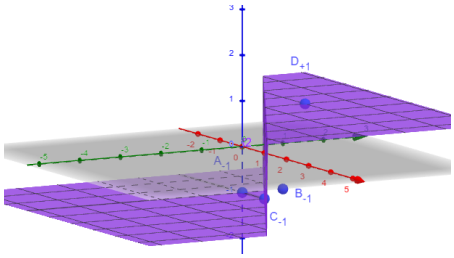


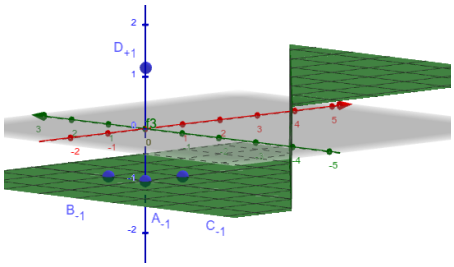
Figure 33: Puntos de la función AND



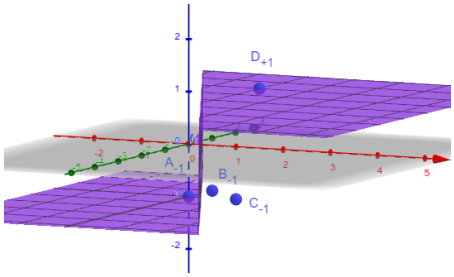
1a iteración PLA (25% acierto)



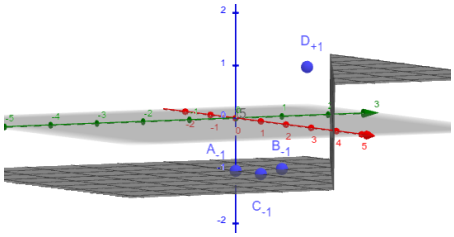
2a iteración PLA (75% acierto)



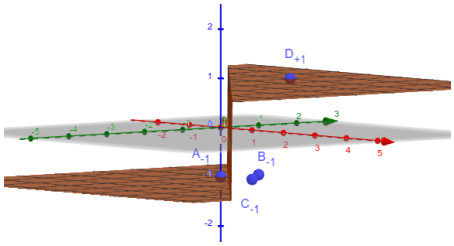
3a iteración PLA (75% acierto)



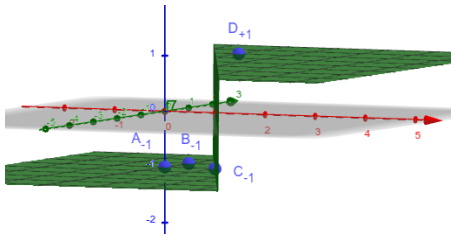
4a iteración PLA (50% acierto)



5a iteración PLA (75% acierto)



6a iteración PLA (50% acierto)



7a iteración PLA (100% acierto)

Figure 34: Iteraciones del PLA para la función AND

En la figura 34 podemos ver como el PLA ha ido ajustando los pesos para buscar el plano que separa correctamente estos puntos.

Una vez observamos los resultados nos damos cuenta de algo, y es que al aplicar la función signo hay una recta que corta con el plano XY, y por lo tanto podemos reducir los resultados en 1 dimensión y dejar los datos mucho más claros (pasar los resultados de las 3 dimensiones al plano XY en 2 dimensiones). Lo que hay que hacer para esto es igualar la ecuación obtenida del plano (no es necesario que sea con la función signo aplicada) y resolver el sistema de ecuaciones que se forma con el plano XY.

$$PlanoXY : (x, y, z) = (x, y, 0) \equiv z = 0$$

$$PlanoFuncinAND : z = 1.35065 * x + 0.657562 * y - 1.35867$$

Al resolver estas ecuaciones nos queda la siguiente ecuación:

$$y = -\frac{1.35065}{0.657562} * x + \frac{1.35867}{0.657562}$$

La ecuación anterior es el resultado de la intersección, que se puede observar en la figura 34 (7a iteración), con el plano XY. El resultado lo podemos observar en la figura 35.

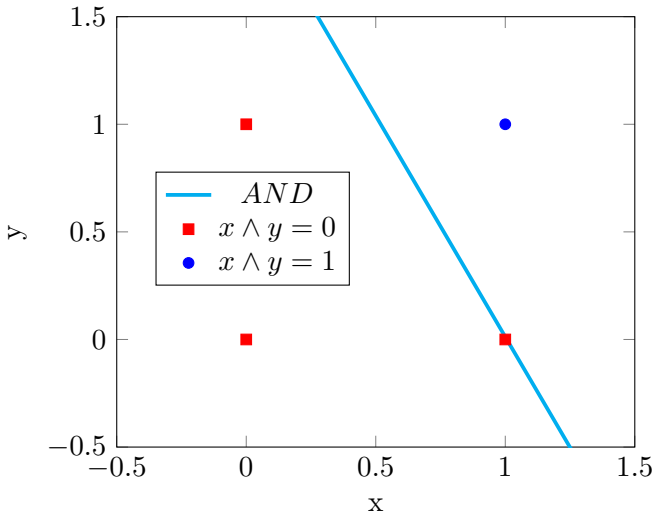


Figure 35: Ecuación resultante de la función AND

El resultado final ha sido ir desde el plano resultante que hemos obtenido por el perceptrón, que podemos observar en la figura 36, utilizar la función signo que podemos observar en la figura 36, y por último obtener la recta que ya hemos observado en la figura 35.

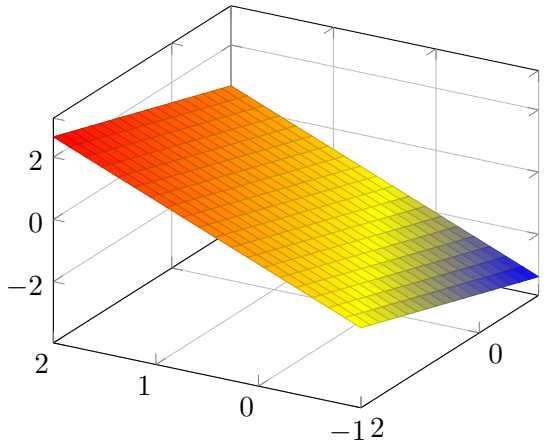


Figure 36: Plano de la función AND resultante

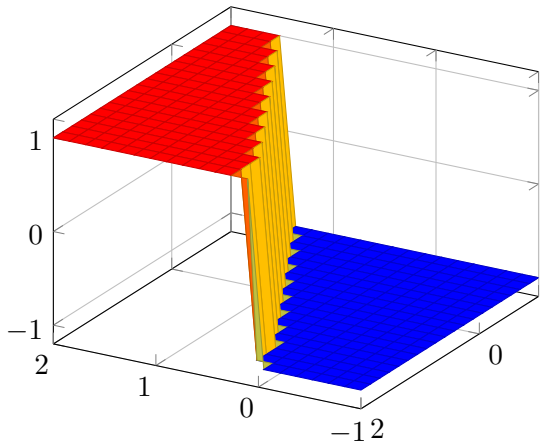


Figure 37: Plano de la función AND resultante aplicándo la función signo

Algo interesante que podemos observar con este experimento es que el porcentaje de acierto no siempre aumenta (la razón ya se explicó en la sección 1.3), por lo que si en este experimento hubiéramos configurado un número de épocas igual a 5 habríamos obtenido como resultado final un 50% de aciertos, no un 75%, en el caso de que no hubiéramos implementado el algoritmo del *Pocket*.

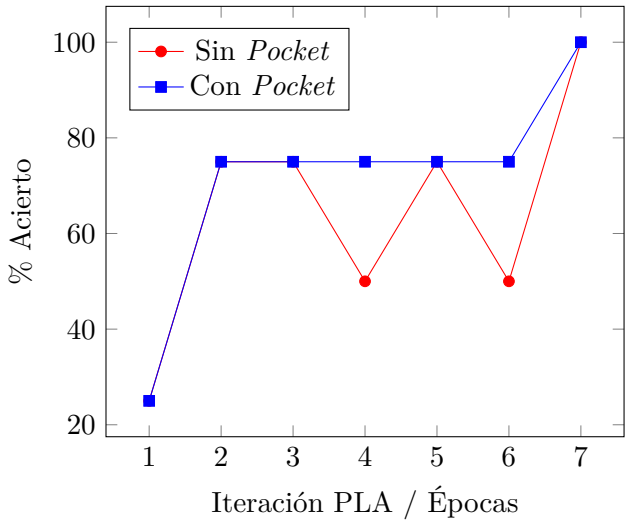


Figure 38: Resultados del PLA con y sin *Pocket*

Como ya hemos comentado, el algoritmo del *Pocket* también podemos verle la utilidad en este sencillo experimento. En la figura 38 se ve como a cada iteración que se ejecuta del PLA (también se puede ver como una época) dependiendo de si tenemos el algoritmo del *Pocket* o no, al final obtendremos un resultado u otro. Esto es esencial cuando nos enfrentamos a un problema que no es linealmente separable y la mayoría de los problemas a los que se les puede aplicar el perceptrón y obtener unos resultados aceptables no serán linealmente separables.

Finalmente, hemos realizado 3 experimentos más, donde cada experimento intenta aproximar una función diferente. Las funciones que intentan aproximar son las siguientes;

$$y = 0.5$$

$$y = 0.5 * x$$

$$y = -0.5 * x + 0.5$$

Los resultados han sido muy favorables ya que con pocas épocas hemos conseguido una muy buena aproximación, como podemos ver en las figuras 39, 40 y 41. En los todos los experimentos realizados se consiguió el 100% de aciertos. Además, en las figuras también está la función que intentaba aproximar y en color rojo marcado el error cometido.

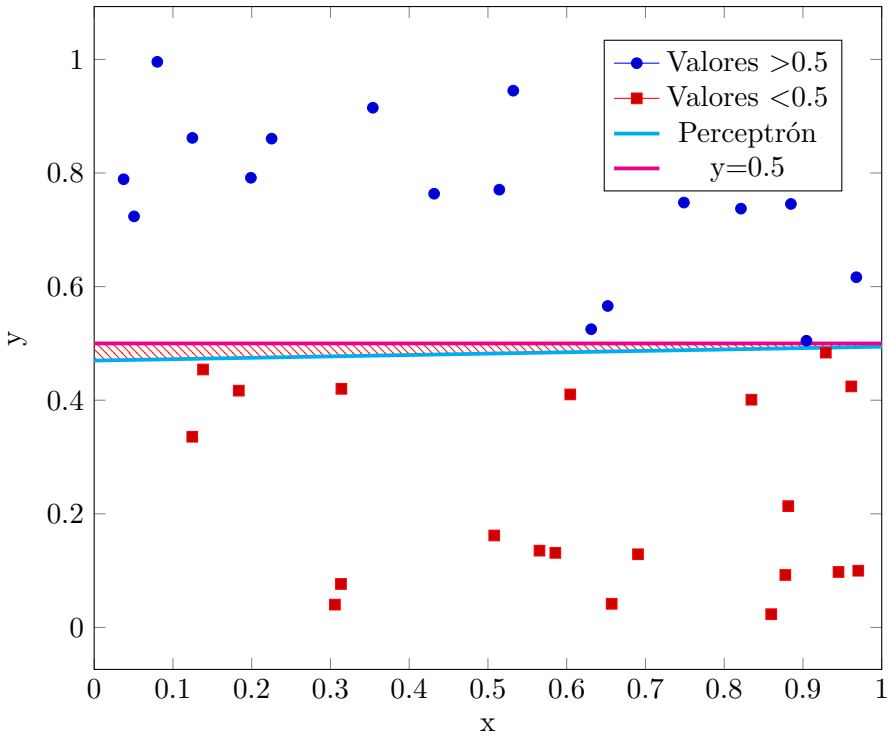


Figure 39: Función $y = 0.5$

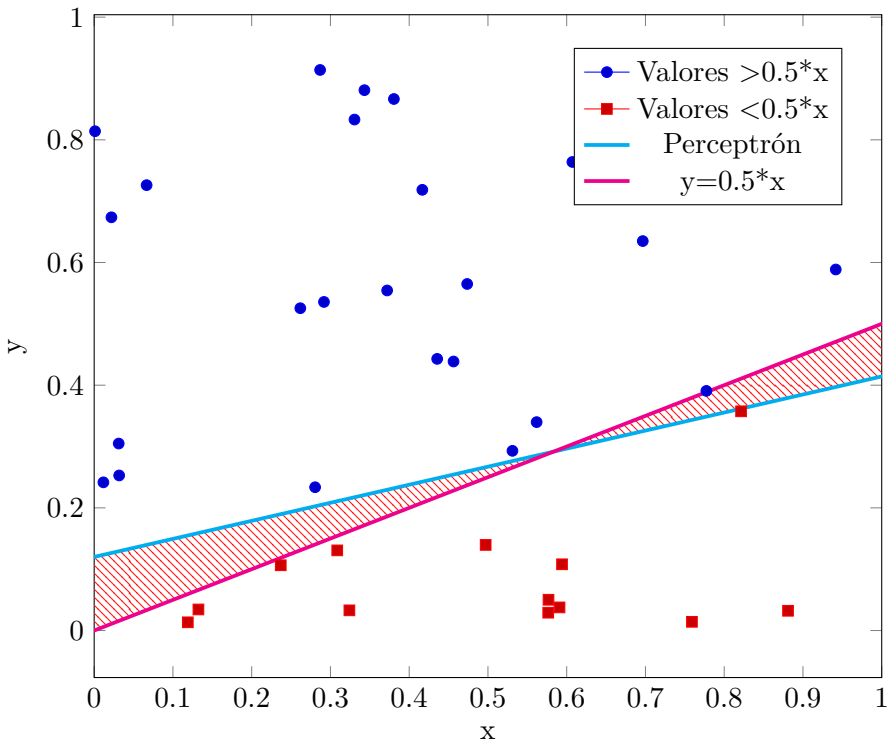


Figure 40: Función $y = 0.5 * x$

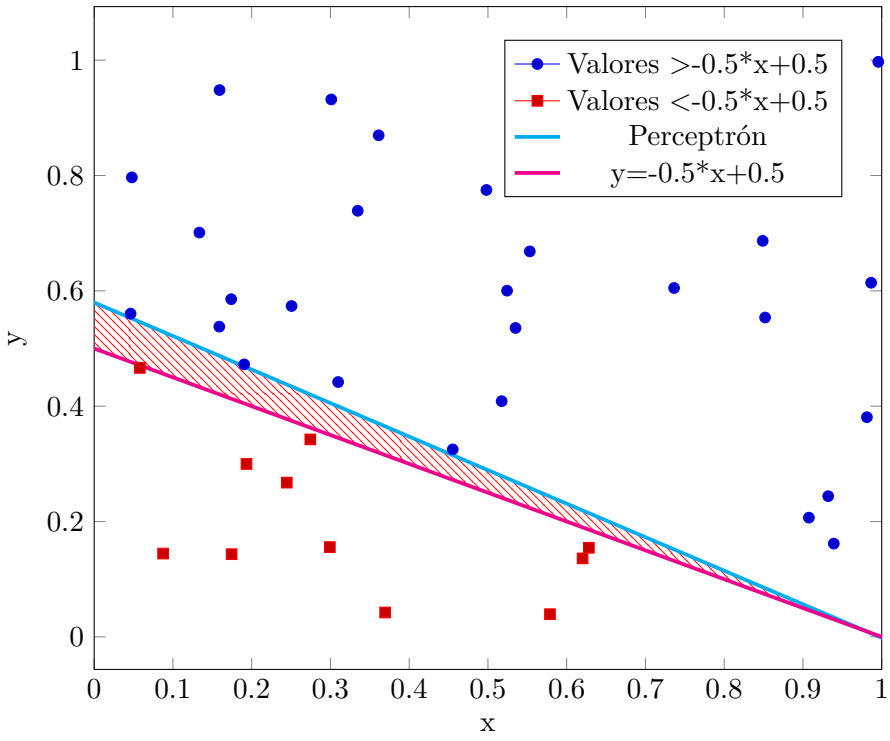


Figure 41: Función $y = -0.5 * x + 0.5$

Hemos realizado un último experimento donde el objetivo es ver la limitación del perceptrón. Como se comentó en la sección 1.3, el perceptrón nos asegura que convergerá, eventualmente, si los datos son linealmente separables, pero no nos asegura que convergerá si no lo son y de echo no converge en su totalidad, pero aún puede darnos buenos resultados e incluso podríamos recurrir a las transformaciones no lineales para convertir los datos que no son linealmente separables en linealmente separables.

El experimento trata de aproximar la función

$$y = \frac{1}{10 * x}$$

la cual no es lineal. El resultado se puede observar en la figura 42, donde el acierto ha sido de un 88%.

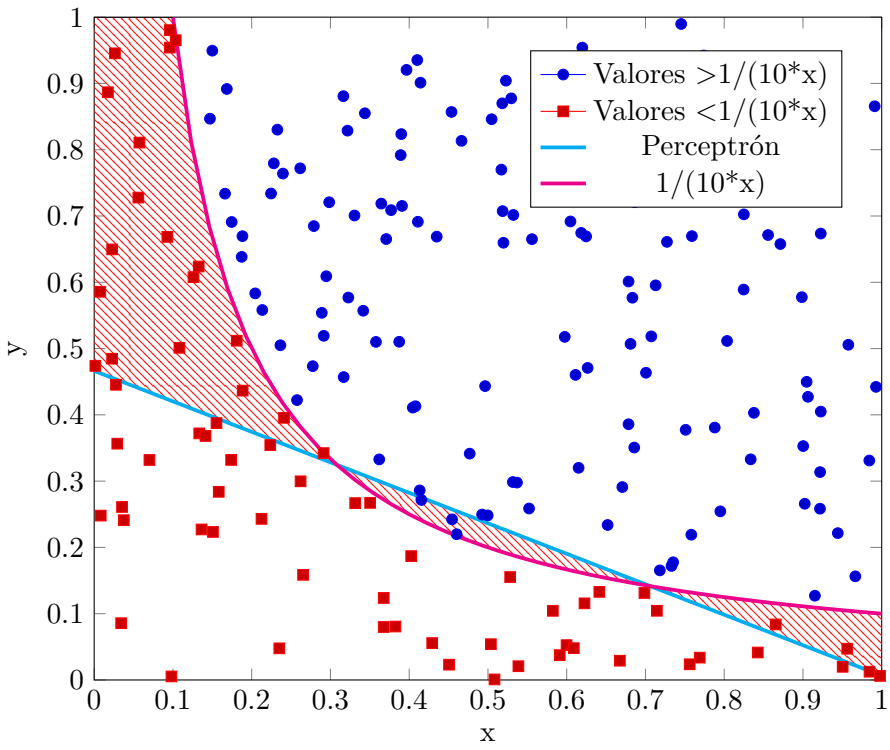


Figure 42: Función $y = \frac{1}{10 * x}$

En el último experimento de la figura 42 utilizamos 200 puntos en vez de los 40 que hemos utilizado en los otros experimentos para que el patrón de la función se notara más visualmente y con el objetivo de que hubieran más puntos que el perceptrón no pudiera aproximar.

3.2 Experimentos con la Red Neuronal

El primer experimento realizado ha sido con la función AND.

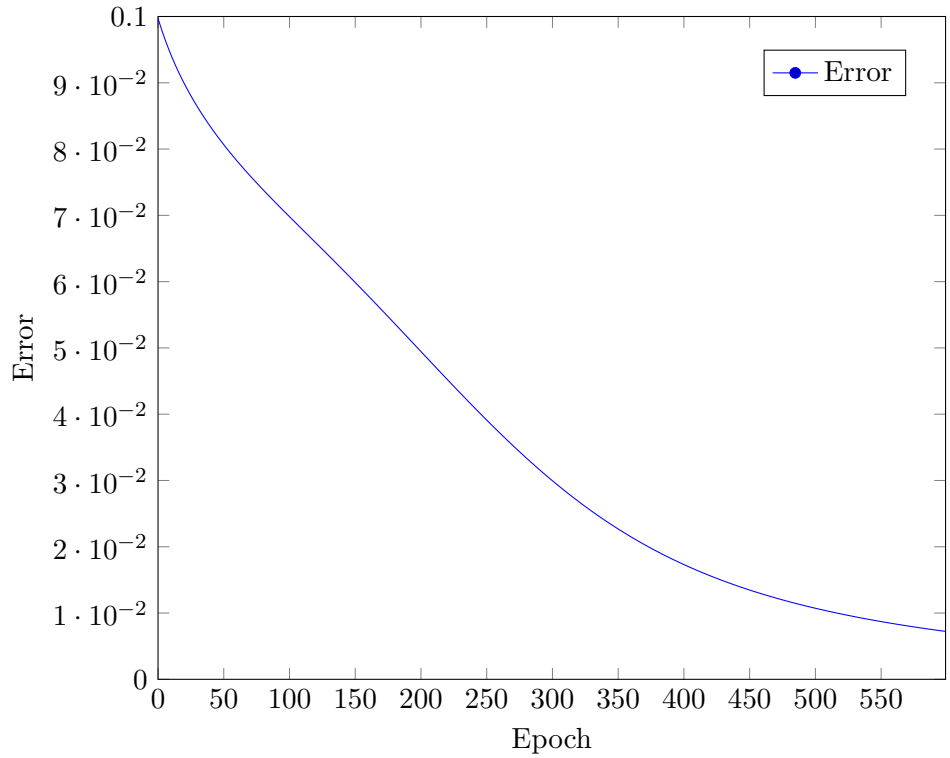


Figure 43: Error para la función AND con topología {2, 10, 1}

A pesar de que la función AND es linealmente separable se ha utilizado una capa oculta para observar el comportamiento de la red. Como se puede observar el error decrece a buen ritmo hasta la época 500 donde empieza a decaer. Se ha utilizado un α de 0.1.

También se han realizado experimentos con la función XOR ya que no es linealmente separable y es un buen punto de partida para probar la red.

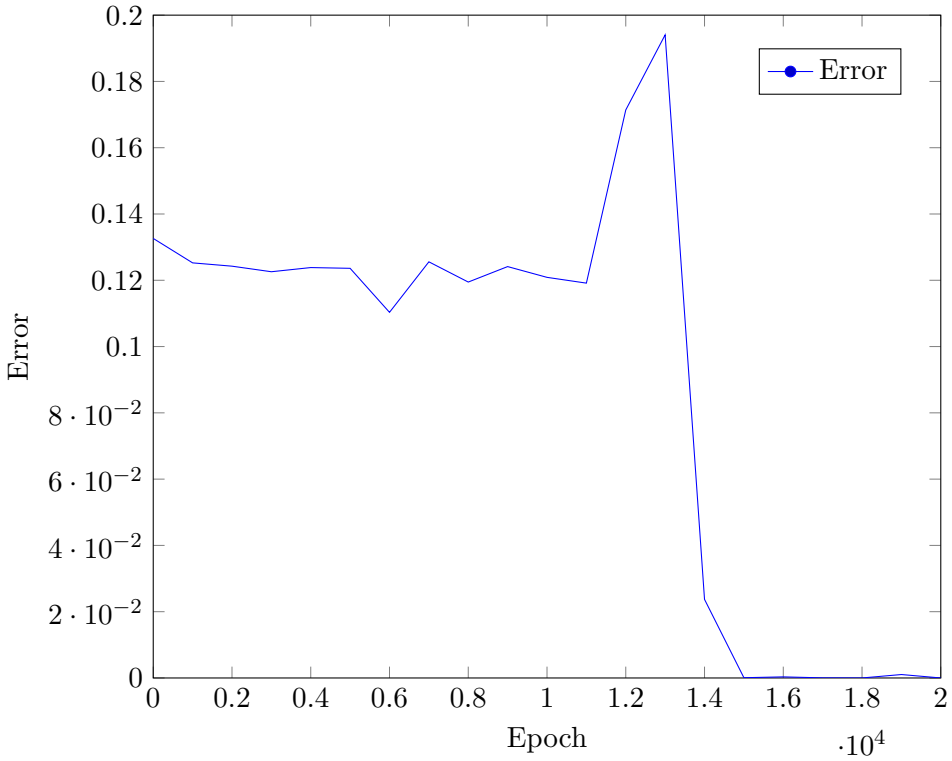


Figure 44: Error para la función XOR con topología {2, 40, 20, 10, 1}

Se ha utilizado una red profunda con una cantidad grande de neuronas para ver el comportamiento. Como se observa una vez la red encuentra una configuración buena el error cae en picado.

3.3 Experimentos con la Red Neuronal: *Dropout*

Se ha entrenado una red neuronal para que aprenda la función $y = \frac{1}{10 * x}$ con y sin dropout, se han medido los valores de error en cada época.

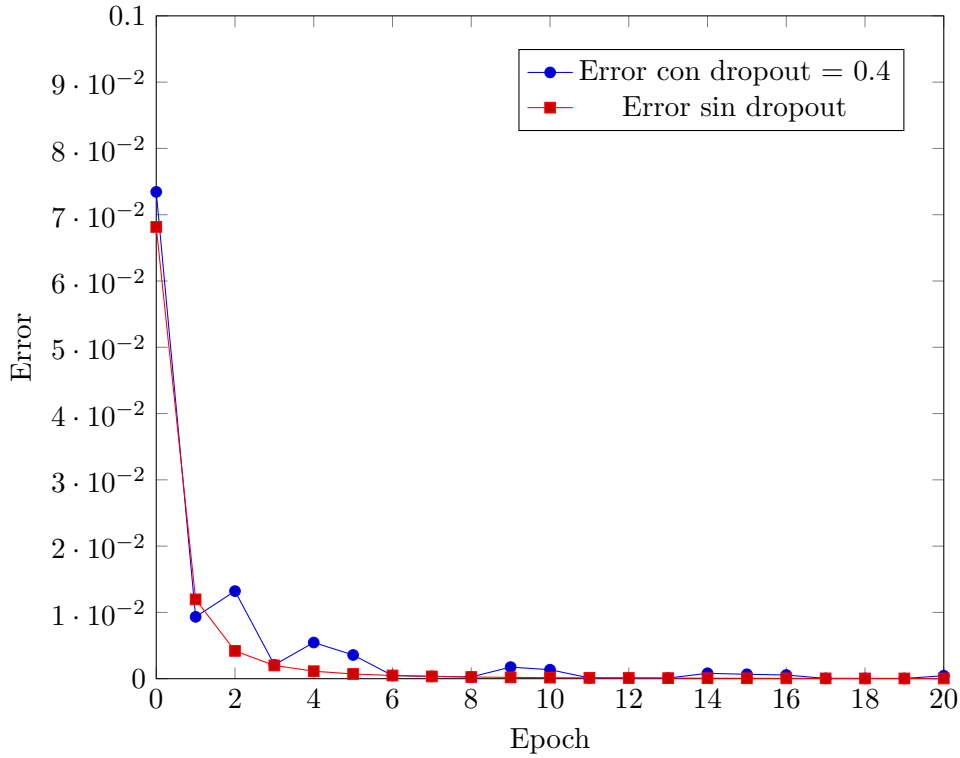


Figure 45: Error para $y = \frac{1}{10 * x}$ con topología {2, 4, 1} y dropout.

En este caso dropout no es realmente útil ya que tenemos una función sencilla de aprender con una topología bastante básica, se puede observar que la desactivación de algunas neuronas en las primeras épocas produce ciertos bumps en el error pero se normaliza hacia el final junto con la versión sin dropout.

3.4 Experimentos con la Red Neuronal: *Cross Validation* 10

Se han realizado experimentos con validación cruzada. En este caso hemos utilizado un dataset real (datos extraidos de RAM de el videojuego Breakout)

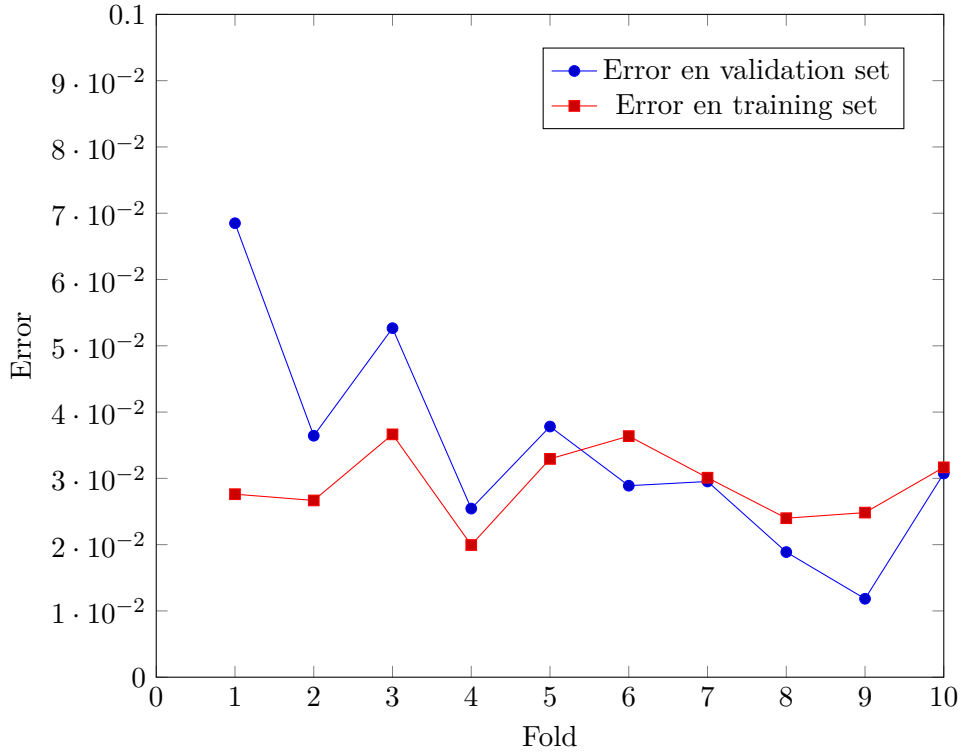


Figure 46: Error en train y validation $k = 10$

Con una validación cruzada de 10 folds en los que hemos separado el dataset en 10 particiones y hemos entrenado por separado, validando con ejemplos que la red no ha visto nunca. En el gráfico podemos ver como el error en validación es bastante más variable en comparación con el error en entrenamiento que se mantiene más regular.

3.5 Experimentos con la Red Neuronal y Genético: *GANN*

Los experimentos realizados con el GANN han consistido en la variación de diferentes parámetros como pueden ser la activación del elitismo, cambiar la probabilidad de mutación, etc. y ver como se convergía antes, o como se alcanzaban mejores puntuaciones, etc.

El primer experimento vamos a ver los mejores resultados obtenidos en el videojuego Breakout cuando tenemos elitismo y cuando no lo tenemos. Como podemos observar en la figura 47, el elitismo hace que no perdamos el mejor individuo de una generación, como sí que sucede cuando no lo utilizamos. El elitismo es parecido al algoritmo del *Pocket* que vimos en la sección 1.3 con la teoría y en la sección 3.1 con los experimentos.

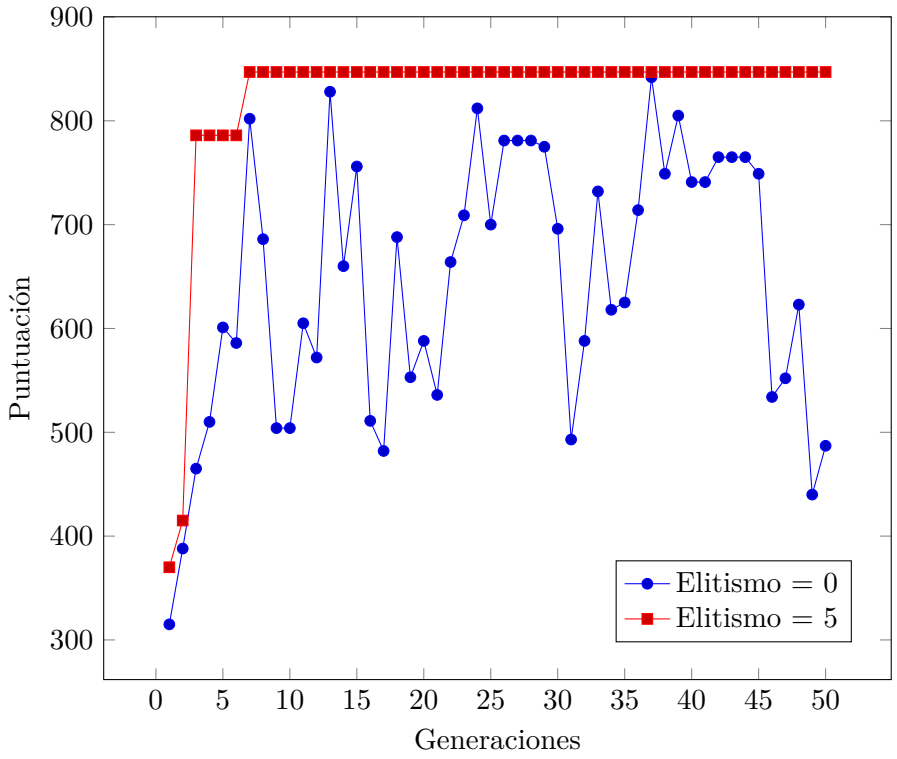


Figure 47: Breakout con y sin Elitismo

Otro de los experimentos que realizamos fue inicializar los pesos en el rango [-2.0, 2.0] en vez de [-1.0, 1.0]. Este cambio depende mucho del problema, ya que al aumentar el rango al doble de posibles valores debería de ser

más difícil de obtener una mejor inicialización, pero eventualmente, como ahora tenemos pesos que no están solo limitados al rango [-1.0, 1.0] sino que ahora pueden tener un rango mayor de valores, puede que ahora tengan la posibilidad de obtener mejores resultados que antes. Esto es justo lo que sucede en la figura 48. Hay que destacar que en ambos hay un elitismo de 5 y el resto de parámetros son iguales para poder comparar los resultados. Lo único que cambia es el factor por el que se multiplica el rango de los números reales. El resultado al final es que converge a una puntuación mayor a pesar de tener que buscar unos pesos y *biases* en un rango de valores más amplios, por lo que quizá en tener ese rango de valores le ha permitido aprender mejor a como jugar.

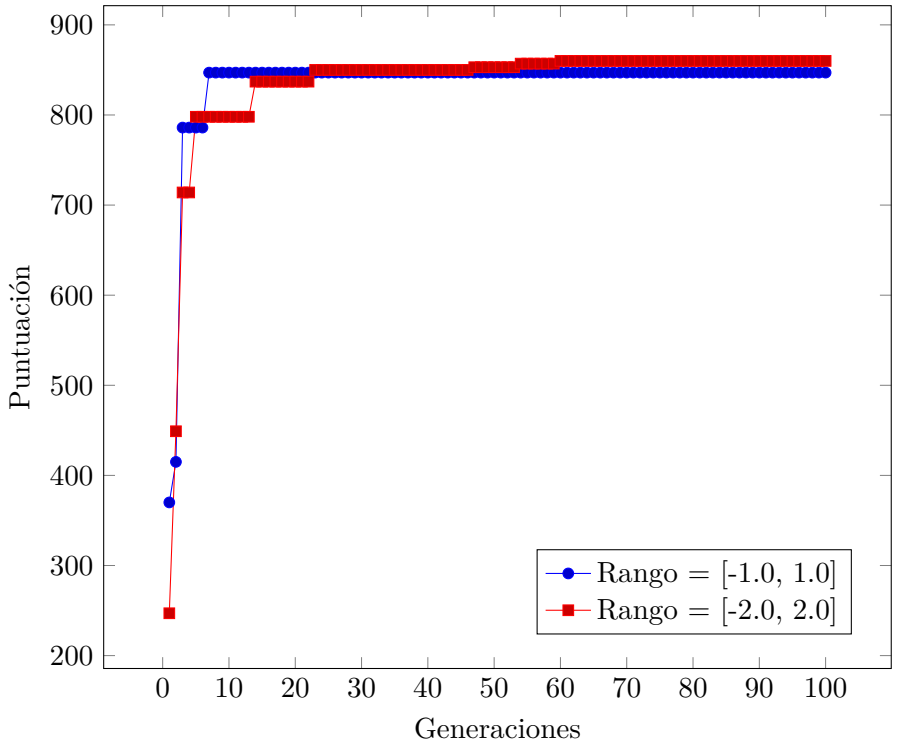


Figure 48: Breakout con diferente rango de valores reales

Una vez tenemos muchos individuos que han jugado bien al videojuego podemos plantearnos lo que ya se comentó en la sección 1.4.4, en la que se habló de la inicialización con una distribución de probabilidad. El modo en el que nosotros hemos realizado este experimento es cogiendo los 10 pesos

de los 7 mejores individuos que hemos encontrado, y hemos unido los puntos para comprobar si se seguía un patrón en los pesos, algo que pudiéramos utilizar como distribución de probabilidad. En la figura 49 podemos ver que los 10 pesos de los mejores individuos claramente siguen un patrón: la mayoría están próximos y en casi todos los casos se cumple que yendo de un punto a otro, o todos tienen pendiente ascendente o todos tienen pendiente descendente.

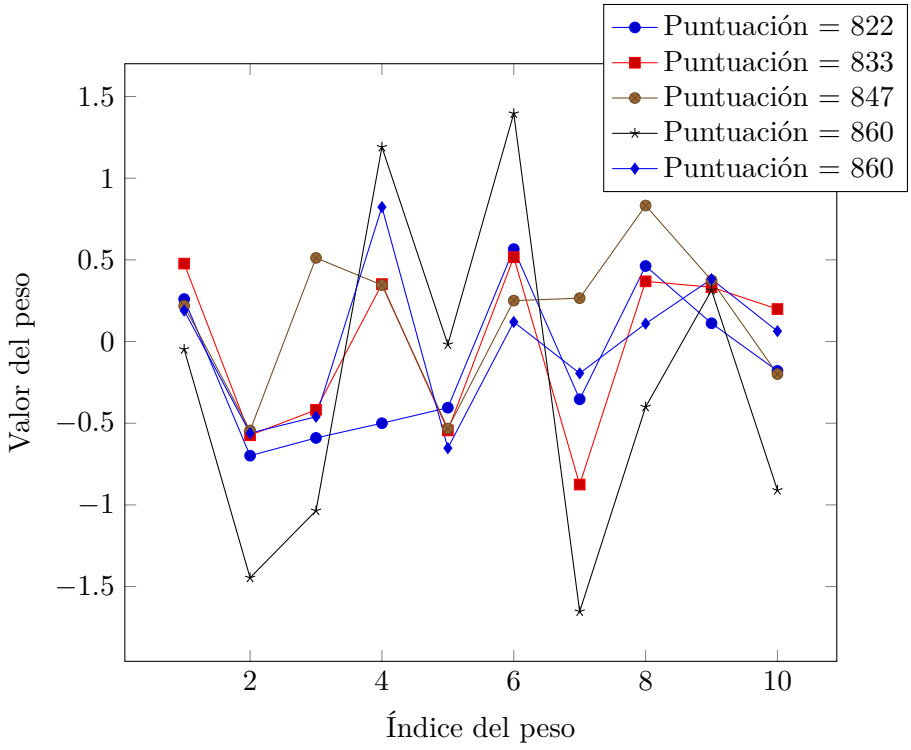


Figure 49: Distribución de probabilidad del Breakout

Si nos fijamos en la figura 50 todavía es más claro el patrón, ya que al interpolar los puntos, gráficamente, queda más claro el patrón que se sigue.

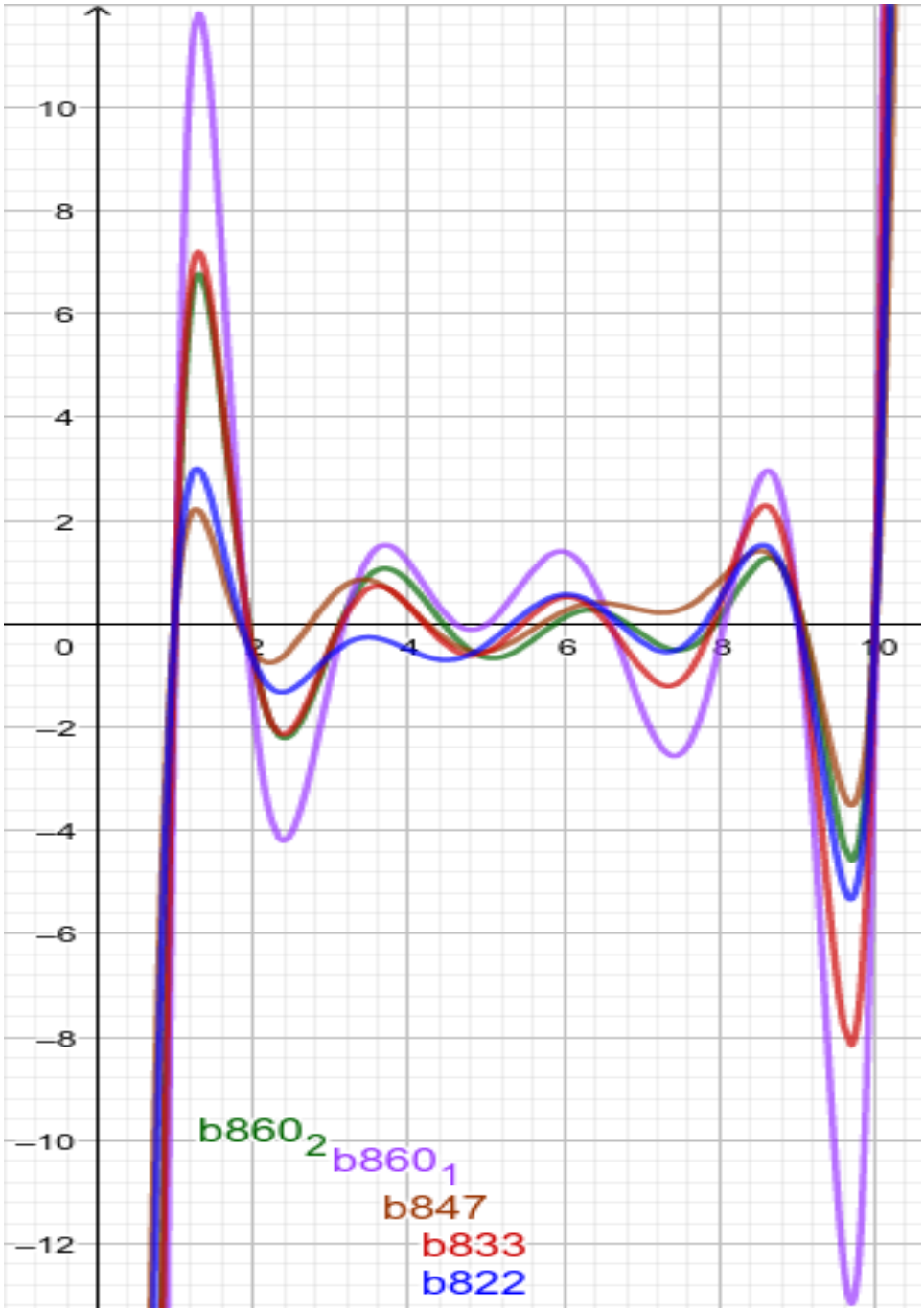


Figure 50: Distribución de probabilidad del Breakout interpolada

Finalmente, para obtener la distribución de probabilidad que vamos a emplear cogemos los valores máximos y mínimos de cada uno de los pesos y ya tenemos los valores que podemos darle al GANN para utilizar con la población inicial. En la figura 51 observamos el resultado.

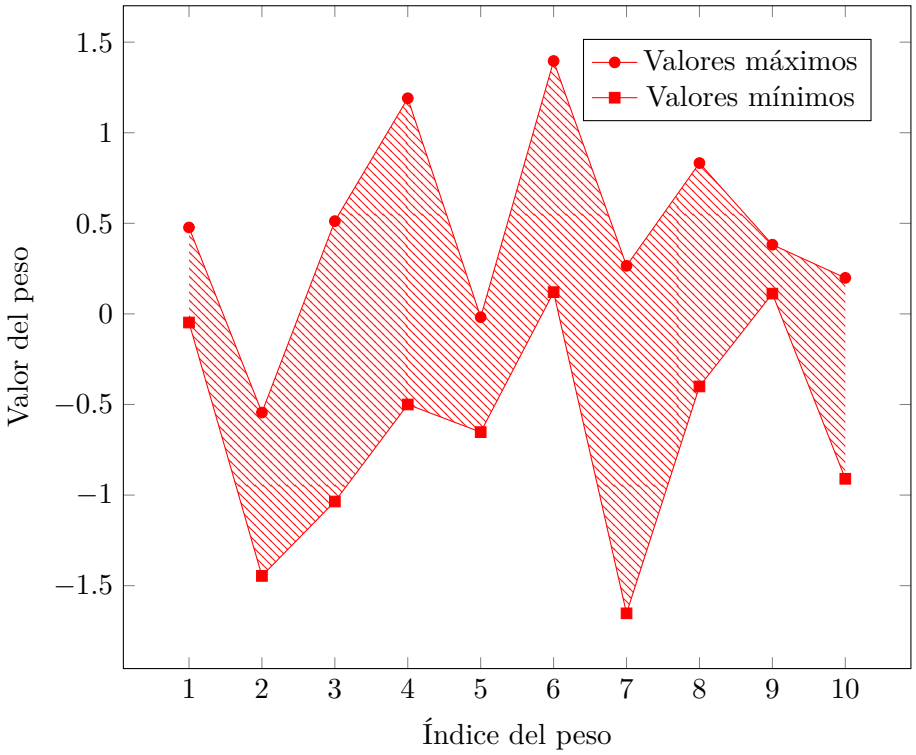


Figure 51: Distribución de probabilidad del Breakout (final)

Una vez ya tenemos estos valores, podemos aplicar la distribución de probabilidad que se observa en la figura 51 y así poder valorar si se obtienen mejores resultados o no. Como se puede apreciar en la figura 52 sí que se obtienen buenos resultados, por lo que obtendremos una convergencia más rápida, lo cual no siempre es bueno, pero para problemas sencillos como el Breakout ahorra mucho tiempo.

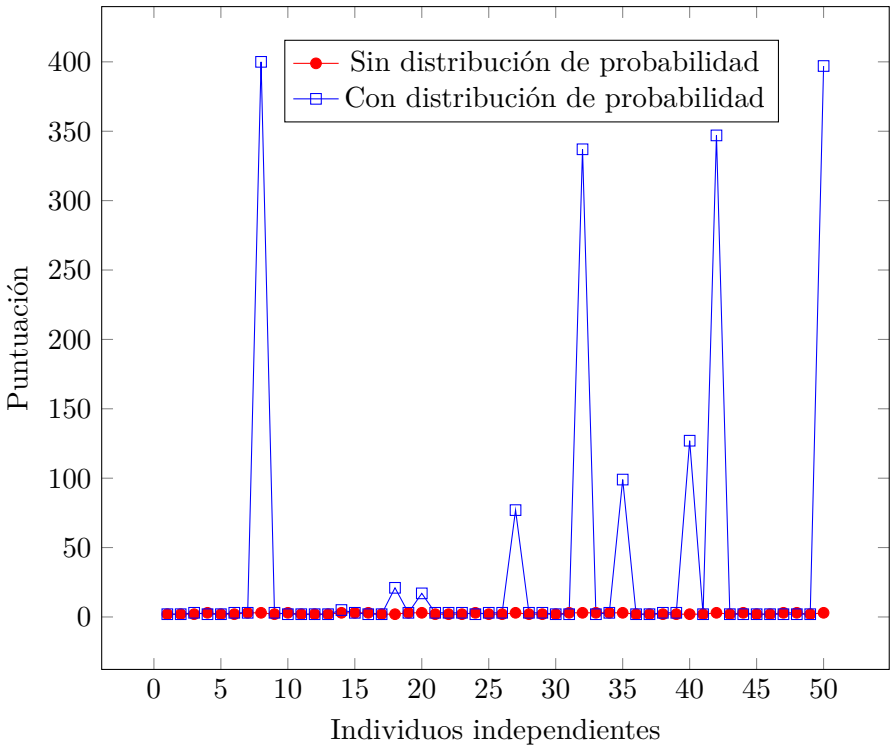


Figure 52: Experimento: Distribución de probabilidad del Breakout

4 Conclusiones

En general estamos muy satisfechos con la práctica. Machine learing es un tema que nos interesa mucho a los 3, y consideramos que los ejercicios propuestos nos han ayudado a ampliar nuestro conocimiento en este campo.

Con el objetivo de conseguir mejores resultados, hemos hecho diferentes iteraciones, probado diferentes tecnologías y realizado numerosos experimentos. Inicialmente desarrollamos el perceptrón simple y la red neuronal, en esta primera aproximación pudimos obtener los primeros resultados a partir de los inputs obtenidos en cada juego.

Tras haber realizado esta primera aproximación y viendo que teníamos una red que funcionaba y tiempo de sobra, nos animamos a implementar el algoritmo genético sobre la red neuronal, tratar los datos para 4 juegos distintos y encontrar topologías y conjuntos de entrenamiento que funcionaran bien era una tarea costosa, el algoritmo genético resolvía en parte este problema.

Por otro lado el genético añade una complejidad extra como es encontrar una función de fitness ideal para cada juego, cuya dificultad variará en función la complejidad del juego. Sin embargo, esta tarea es menos tediosa que sanitizar y tratar todos los datos del juego.

Además, realizamos un intento de implementación de *Neuro Evolution of Augmented Topologies* (NEAT) que no se finalizó debido a la falta de tiempo y complejidad de la tarea. A pesar de esto, nos parece que es la tecnología más interesante de todas y no descartamos finalizarla en un futuro.

En conclusión,el algoritmo genético ha sido el que finalmente hemos utilizado para resolver los problemas propuestos debido a que nos ha parecido una tecnología mucho más interesante y que se ajusta más a este tipo de problemas.