Razonamiento Automtico
Machine Learning

lvaro Jover lvarez (aja10@alu.ua.es)
Jordi Amoros Moreno (jam80@alu.ua.es)
Cristian Garca Romero (cgr71@alu.ua.es)
Universidad de Alicante

January 3, 2019

ndice

1 Tecnologas implementadas y estado operativo.

1.1 Anlisis de RAM

Uno de los puntos ms importantes a la hora de implementar un bot con IA para un juego de Atari, es entender cmo est hecho. Utilizaremos el entorno *Arcade Learning Environment* (ALE) para extraer caractersticas de los juegos, el cual cuenta con una API que nos permite extraer informacin 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.

Adems, 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 ejecucin) como se puede ver en la figura 1.

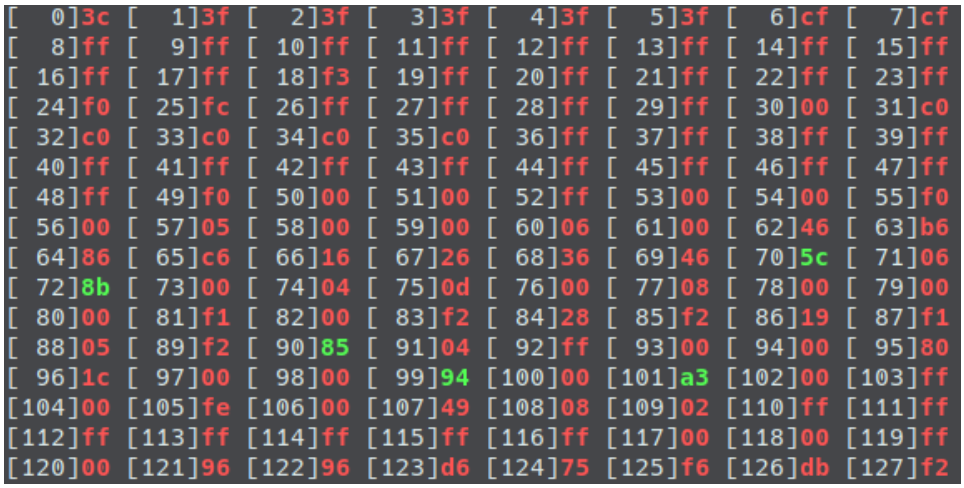


Figure 1: El color verde indica que el valor ha cambiado en este step.

Una de las caractersticas de este lector es que acompaa la ejecucin con un volcado de analytics para ver las posiciones de RAM que ms han cambiado en una ejecucin determinada.

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

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 - Anlisis de las primeras posiciones de RAM

Como se puede observar en la figura 2, para obtener la informacin correcta no solo basta con extraer las posiciones relevantes, en algunos casos ser necesario procesar esta informacin. 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 estn invertidos, adems, el primer nibble requiere una operacin extra, una resta (7 - valor del nibble).

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

1.1.1 Modificando ALE

La primera aproximacin que se hizo a la hora de analizar los datos de los diferentes juegos fue bsica y til, pero presentaba varios problemas. El problema ms serio que nos encontramos era la cantidad de datos que tenamos ante nuestros ojos, cosa que puede dificultar bastante el encontrar la informacin importante que necesitamos y, adems, que dependemos totalmente de nuestra "intuicin" para saber que significa cierta posicin de la RAM. Intentando solventar un poco este problema, se nos ocurri aislar las posiciones que hay en la RAM y, adems, poder modificar esos valores aislados para no depender totalmente de nuestra intuicin, sino saber ms a ciencia cierta, a base de prueba y error, saber qu hace cierta posicin de RAM, lo cual nos dara la ventaja de profundizar ms en aquellas posiciones de RAM que por alguna razn, en el paso anterior, hemos visto que podan resultar relevantes.

A la hora de intentar poner esta idea en prctica 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 mtodo *getRAM()* que hemos ido utilizando en todo momento para el anlisis de los datos (y para posteriormente el tratamiento de los mismos) y que est en el archivo *src/ale_interface.cpp*. La implementacin de dicho mtodo es la siguiente:

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

Figure 4: Mtodo *getRAM()*

Como observamos, el mtodo *getRAM()* nos devuelve una referencia de un objeto de tipo **ALERAM**, por lo que vamos a ver que mtodo nos pueden servir para modificar la RAM. Cuando investigamos el archivo *src/environment/ale_ram.hpp* encontramos la definicin del objeto ALERAM y la implementacin de sus mtodos. Contiene 2 mtodos que pueden servirnos para nuestro propsito (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: Mtodos de acceso a la RAM de ALEInteface

Una vez ya encontramos estos mtodos, intentamos hacer una pequea prueba, la cual falla, ya que al intentar hacer uso del mtodo *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 posicin 72). Lo que estaba sucediendo, despus de indagar un poco por el cdigo de ALE era que el mtodo *getRAM()* de *ale_interface.cpp* era un mtodo que a su vez llamaba a otro mtodo 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 ncleo 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 mtodo *getRAM()* desde *ale_interface.hpp*, se hace otra llamada al mtodo *getRAM()* de *src/environment/stella_environment.cpp* y este devuelve su copia interna, no la del ncleo de la consola. Cada vez que el objeto *StellaEnvironment* actualiza su copia hace uso del mtodo *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: Mtodo *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 mtodo que

lo que haga es actualizar la RAM del objeto *OSystem* del mismo mtodo que que podemos observar que se puede obtener una copia de la RAM con su mtodo *console().system().peek()*, pero tenemos que comprobar si existe el mtodo que nosotros necesitamos, y es que en vez de *peek()* necesitamos un mtodo que nos permita actualizar. De nuevo, buscando en el cdigo, 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 ltimo encontramos el mtodo *poke(uInt16, uInt8)*, el cual nos indica a travs de su descripcin que modifica un valor de la RAM de la consola Atari 2600. Parece ser que ya lo tenemos todo, pero nos queda un ltimo problema, y es que este mtodo no es accesible desde el objeto *StellaEnvironment*, ya que su referencia del objeto *OSystem*, a diferencia del objeto *ALEInterface*, est en su parte privada y no nos da ningn mtodo pblico para obtener una copia de la referencia. El ltimo paso que nos queda para solucionar este ltimo problema es modificar el cdigo y aadir un mtodo que nos permita actualizar la RAM de la Atari 2600 a partir de la copia que tiene internamente el objeto *StellaEnvironment*, ya que esta ltima s 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: Mtodo *poke(uInt16, uInt8)*

El mtodo que hemos implementado es muy sencillo. Es igual que el mtodo *processRAM()* pero utilizando el mtodo *poke()* en vez de *peek()*. Una vez ya implementado el mtodo, simplemente implementamos otro mtodo en el objeto *ALEInterface* que simplemente haga una llamada a este mtodo que est en el objeto *StellaEnvironment* y ya habremos actualizado la RAM de la Atari 2600. Por ltimo solo quedara recompilar ALE. Este proceso que hemos realizado a sido con el nico objetivo de comprender mejor el significado de las posiciones de RAM de los diferentes juegos a analizar durante este trabajo. Se adjuntarn para cada juego un archivo denominado **RAM.txt** en el cual hemos analizado cada posicin 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: Mtodo que actualiza la RAM de la Atari 2600

1.2 Bots bsicos

Se han desarrollado bots semi-deterministas empleando tcnicas bsicas 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 opt por implementar IA bsica para cada uno de los juegos.

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

Adems, sobre esta IA bsica, se pueden hacer iteraciones de mejora, teniendo en cuenta ms datos o mas informacin en pantalla, como se ha comentado anteriormente en la subseccin 1.1.

Este scripting bsico ayudar mas adelante a la implementacin utilizando machine learning, ya que los datos extrados y procesados para la implementacin bsica sern utilizados por el algoritmo de machine learning.

A continuacin, describiremos cada uno de los bots bsicos y su funcionamiento al igual que algunos detalles de implementacin.

1.2.1 Plantilla comn

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

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

```
/**
* 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 ejecucin

```
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 ejecucin que podemos ver a continuacin en la figura 11. Este bucle est 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 ejecucin.

En este bucle observamos nuestra opcin **printRam** que llama al mtodo encargado de imprimir la RAM, el cual convierte a hexadecimal cada uno de los valores de RAM obtenidos mediante el mtodo **getRAM().get(i)** del entorno ALE, adems 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 opcin que encontramos en el bucle principal de ejecucin es **display_media**, pero en este caso es usado para llamar al mtodo **checkKeys()**, esto se hace porque no tiene sentido trackear el input si no existe contenido de video. Este mtodo de input es propio, ya que el mtodo 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** decidir que funcin se llama para calcular **totalReward**. **manualMode()** mapea el teclado a diferentes acciones del juego, mientras que **agentStep()** es el bot autnomo especfico 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 seccin, pues el nmero de inputs que tiene es mas bien pequeo. 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 funcin a los ladrillos restantes que quedan, lo cual provocaba que perdisemos siempre en este punto.

Simplemente proporcionando la posicin 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 tamao de la pala, al igual que en vez de tener en cuenta solo la posicin actual de la pelota, aadir a los datos la posicin anterior. Con todas estas mejoras la puntuacin se maximiz hasta el punto en el cual el agente fue capaz de pasarse el juego completo. A continuacin se muestra en el algoritmo 1 el pseudocdigo de la

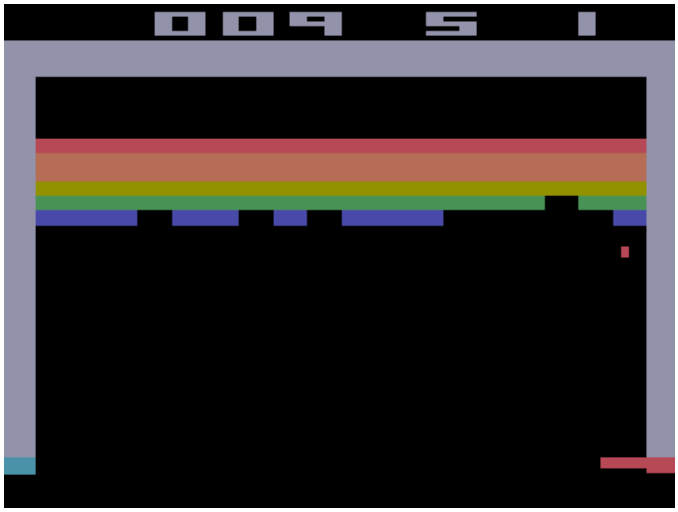


Figure 13: Juego Breakout.

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 pseudocdigo 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, adems de **playerX** y **ballX**.

La tercera parte calcula la direccin de la bola, esto se saca comprobando la posicin anterior de la bola con la actual en el eje X, una vez sabemos la direccin aplicamos una suma con un poco de aleatoriedad (para evitar ejecuciones deterministas) en la direccin recogida. Una vez hecho eso nos guardamos la posicin de la pelota para la siguiente iteracin.

En la cuarta parte aplicaremos el input en funcin a la posicin 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 resuma en todo ventajas.

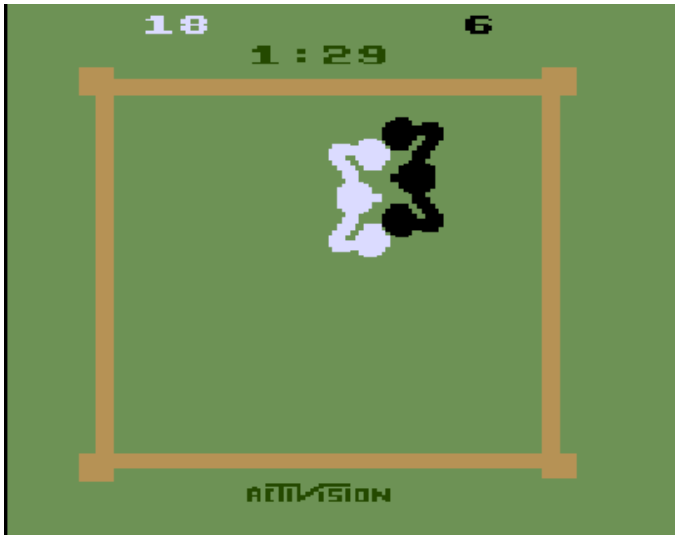


Figure 14: Juego Boxing.

En este agente hemos tomado una estrategia agresiva, si el rival lanza un puetazo, lo intentaremos bloquear con nuestros propios puos poniendo a nuestro boxeador exactamente en la misma posicin "Y" que el boxeador contrario (una de las features de Boxing es que la colisin de los puos bloquea los golpes). Adems, nuestro boxeador nunca se mover hacia la izquierda, solo se mueve hacia la derecha en posicin de ataque intentando posicionarse en la "Y" del rival, como hemos comentado antes. A su vez, basndonos en ciertas tolerancias, se atacar siempre que sea posible. Esta estrategia agresiva funciona una gran mayora de las veces, adems para evitar el determinismo se ha implementado una pequea 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.

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

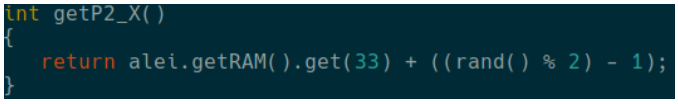


Figure 15: Uso de **rand()** para evitar el determinismo.

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 posicin del jugador uno y la del jugador dos, extrayndolos de RAM. Como ya hemos comentado anteriormente, estas posiciones implementan una aleatoriedad mnima para evitar ejecuciones deterministas. Adems, en la misma seccin, 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" vlido 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 patrn de movimiento similar al Space Invaders. Tendremos que disparar a las naves rivales para pasar de fase adems 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 mltiples 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 caracterstica de ALE que nos permite leer los pxeles de la pantalla.

Al analizar la RAM, observamos que haban demasiadas caractersticas a tener en cuenta, a pesar de tener un anlisis 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. Tambin 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 bsico (sin machine learning), era ms fcil hacer un anlisis de lo que estuviera pasando en la pantalla en un instante determinado, sin dejar de lado la RAM. Para esto, se dise un mtodo de visin en el cual la nave es capaz de ver y filtrar enemigos en un rea de visin y actuar concorde la situacin dependiendo de unas reglas determinadas. En la figura 17 se puede observar el rea de visin de la nave.

Figure 17: Vision de la nave.

La visin de la nave se divide principalmente en 3 partes. En la figura vemos una lnea roja representada con el nmero 1, esta lnea roja representa la visin de la nave en funcin a la anchura de la misma. El nmero 2 es un

rea parametrizable que permite ampliar esta visin de forma simtrica por los dos lados de la nave. Tanto el rectngulo rojo como el azul se extienden por casi toda la "Y" de la pantalla y lo nico 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 actuar en consecuencia.

El nmero 3 es un rea reducida dentro del rea completa que se encarga de detectar todas las posibles amenazas, no solo disparos, que estn cerca de la nave. Es decir, el objetivo de la visin es la deteccin de entidades peligrosas en un rea 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 nica funcionalidad.

Para analizar la parte prctica de ste algoritmo es necesario separarlo en distintas partes, deteccin, esquiva y seguimiento.

La parte de deteccin 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 pxel. Gracias a esto, podemos determinar el color de los disparos y de las naves enemigas, los que guardaremos en las siguientes variables para despus 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 visin 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. Adems podemos observar otras constantes como **LINE_WIDTH**, que representa el ancho en pxeles de la resolucin de la Atari, **SHIP_WIDTH**, el ancho en pxeles de la nave y **LEFT / RIGHT_THRESOLD**, que es la coordenada "X" mnima y mxima en la que se puede mover la nave aliada.

Adems de estas constantes, Demon Attack cuenta con cuatro constantes adicionales que nos permiten parametrizar algunas caractersticas 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 rea extra de visin 2 definida en la imagen 17, la segunda variable define en que lnea empieza el rea de visin 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 colisin y como se operan.

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

Una vez que ya sabemos qu 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 pxeles de la pantalla con la funcionalidad de ALE **getScreenGrayscale**, para luego emplearlos dentro del bucle en la funcin **isBlockingHit(int)**, que analizaremos mas adelante. Este bucle se encarga de recorrer 125 filas de la pantalla. En funcin a la respuesta de **isBlockingHit(int)**, marcar una variable a true, la cual representa que ha habido una colisin, adems dentro de este struct **DirtyState**, disponemos de otra variable de tipo EBlockingHit para marcar tambin la direccin contraria a la colisin dada por isBlockingHit. Se ha decidido este thresold ya que el resto de lneas de pantalla contienen informacin no relevante para el problema.

Para resumir esta parte del cdigo, marcamos el estado a Dirty y recogemos la direccin conveniente, siempre que isBlockingHit haya detectado una colisin. Hasta que no salimos del estado Dirty, no podremos recoger nuevas direcciones, pero de ello se encargar otra parte del cdigo. Es decir, tericamente se ha implementado una mini mquina de estados finita (FSM), ya que la nave posee dos estados principales (dirty y no dirty) y la forma de transicionar entre ellos es mediante el cdigo 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: Funcin isBlockingHit.

La funcin isBlockingHit tiene que ser capaz de recorrer el array obtenido por **getScreenGrayscale** de manera eficiente. Para ello lo primero que hace es resolver la posicin del array que ocupa el primer pxel de la lnea a analizar, esto es fcil, pues simplemente multiplicando el ancho de la lnea con el nmero de lnea ya tenemos este nmero (asumiendo que las lneas empiezan en 0).

La siguiente tarea es resolucionar el thresold, el cual consiste en el rea que ocupa la nave, en este caso 7, mas el thresold determinado, que en nuestro caso corresponde al nmero de pxeles extras que tendremos en cuenta en cada lado, por ejemplo, si el thresold es 2, el rea total ser 7+2+2 = 11.

La iteracin que haremos en el bucle ir desde 0 hasta el rea total que ocupa la visin de la nave, esto es, como ya hemos dicho anteriormente, el rea de la nave mas el thresold. Esto hace que solo iteremos los pxeles necesarios para nuestra situacin.

La primera lnea que encontramos en el bucle puede parecer confusa, pero tiene su explicacin:

Algorithm 4: Calculo del valor del pxel en un punto determinado de nuestra area de visin.

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

Necesitamos el valor del pxel en un punto determinado de nuestra rea de visin. Para ello, necesitamos pedirle al array grayscale esta informacin, pero antes necesitamos calcular qu posicin del array es la correcta. Al haber precalculado la posicin del array del primer pxel de la lnea lo tenemos mucho mas fcil, pues tenemos por donde empezar. A ese nmero le tenemos que

sumar la posicin del jugador uno, representado en **getP1_X(true)**, al cual le pasamos el parmetro true para que nos devuelva la posicin en pxeles, la cual es ms precisa para este problema. Una vez tenemos la posicin del jugador solo nos queda sumarle i para abordar todo el rea de visin. Pero como bien podemos observar en el algoritmo 4, estamos restando a la posicin del jugador el **VISION_THRESOLD**, esto se hace porque sino estaramos solo teniendo en cuenta el rea extra por la derecha, ya que la colisin estara desplazada a la izquierda como muestra la figura 22, por ello es necesario restar **VISION_THRESOLD**.,

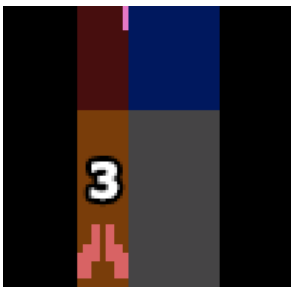


Figure 22: Area de sin restar VISION_THRESOLD al jugador.

Una vez hemos resolucionado el valor del pxel, podemos hacer el filtrado del mismo, el cual se realizar 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: Resolucin de la colisin

Como bien observamos en la figura 23, haremos una primera distincin para las moscas y enemigos mas cercanos a nuestra segunda rea de visin seguido de la visin normal. Priorizamos el segundo rea de visin porque est mas cerca del jugador, lo cual se traduce en ms peligro a perder una vida. En el primer if, miramos si nos encontramos en el rea de riesgo y filtramos por color principalmente si lo que hay en ese pxel es un enemigo. Si es as, es que ha habido una colisin. Si la i en la cual nos encontramos es menor (o est a la

izquierda) de la mitad del rea de visin, nos moveremos a la derecha (camino ms corto para evitar la colisin). Haremos lo mismo para el otro ladro. El segundo if es exactamente lo mismo solo que nos centramos nicamente en las balas enemigas.

Finalmente, si no se ha encontrado colisin 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 sta figura

podemos observar la parte del algoritmo que finalmente aplica movimiento a la nave. En este punto ya hemos solucionado la colisin y solo nos queda movernos en la direccin conveniente.

Esta parte del cdigo se divide en dos dependiendo respectivamente si hemos encontrado una colisin o no. En la primera parte ya tenemos en el **DirtyState** apuntado la direccin en la cual movernos, sin embargo si nos encontramos en los bordes de la pantalla, no nos podremos mover la la direccin del borde, es por esto por lo que cambiamos de direccin cuando llegamos a un borde. Si no estamos en ningn borde, simplemente aplicamos la direccin 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 est 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 colisin. **EnemeyHandler** se encargar de devolvernos el enemigo mas relevante en un instante determinado (usualmente el mas cercano al jugador). Esta funcin no funciona el 100% de los casos, ya que como hemos comentado anteriormente, la RAM no es especfica a casos concretos sino que va cambiando, por lo tanto habr, 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 lser. La solucin completa y real sera llevar un seguimiento mas especifico de todas las unidades en RAM, lo cual puede llevarnos a implementar demasiados casos especficos para un escenario mas simple de solucionar mediante un algoritmo genrico (mediante una buena funcin de fitness) o un neuroevolutivo en su defecto.

1.3 Perceptrn

El perceptrn es una de las tcnicas de *Machine Learning* ms sencillas de comprender, implementar y tiles. 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 perceptrn seguramente no sea la mejor opcin. El perceptrn lo que intenta hacer es separar los datos que le demos, asegurndonos que obtendr la mejor solucin posible si los datos de los cuales partimos son linealmente separables, y en el caso de no serlo, deberemos dar la mejor solucin posible (aquella con menor error).

El perceptrn es una tcnica 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 configuracin para acercarnos ms a los resultados correctos. Esta tcnica puede utilizarse para clasificacin o para regresin. El resultado que al final obtendremos ser, dependiendo del nmero de entradas que tengamos, una lnea, un plano o un hiperplano. Los elementos del perceptrn son las entradas, los pesos y la salida.

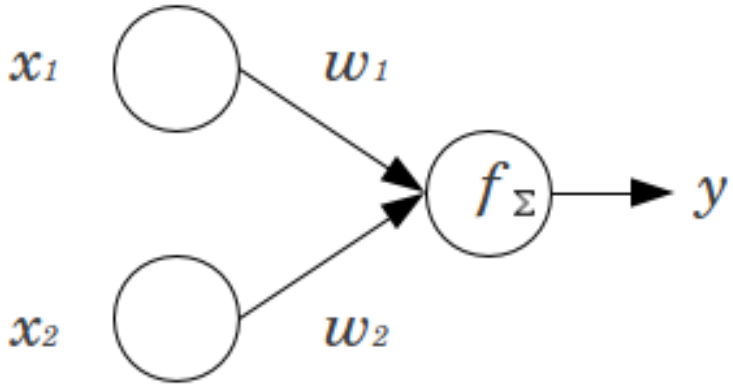


Figure 26: Estructura del perceptrn

La salida que tiene el perceptrn, en el caso del clasificador, es siempre de +1 o -1. Para obtener esta salida, la frmula 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 frmula que hemos descrito encontramos un problema, y es que no hay trmino independiente, todos los pesos estn asociados con alguna entrada, por lo que la linea, plano o hiperplano siempre pasar por el origen de coordenadas, cosa que seguramente no sea as para la mayora de los problemas. La solucin es, como hemos dicho, aadir un trmino independiente, al cual se le suele llamar *bias*. Una manera de hacerlo es aadir un peso w0 y una entrada x0 artificial que siempre tenga el valor 1, con lo que al final nos queda la siguiente frmula:

$$x_0 = 1$$

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

En nuestro trabajo hemos implementado el algoritmo de aprendizaje del perceptrn (PLA), el cual nos permite ir ajustando la configuracin del perceptrn, 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 perceptrn es el siguiente:

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 cdigo del PLA debera ejecutarse tantas veces como fuera necesario, y eventualmente llegara a una solucin, 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 iteracin de este bucle se le conoce con el concepto de **poca**).

Algorithm 6: PLA con pocas

```
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 ltimo, vemos el problema de que si no tenemos datos linealmente separables, las pocas no solucionan el problema totalmente (solucionan el problema de no quedarnos en un bucle infinito), ya que puede que antes de llegar a la ltima poca, ya hayamos encontrado una solucin bastante buena, pero que la hayamos perdido por el camino al actualizar los pesos, por lo que a cada poca que se est ejecutando el PLA, deberamos de estar guardando la mejor solucin hasta el momento. A esto se le conoce con el nombre del algoritmo *Pocket*.

Algorithm 7: PLA con pocas 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 cdigo es muy similar al descrito, ya que la parte ms importante es la que hemos explicado, la del PLA. La estructura que hemos empleado es la de una clase llamada **Perceptron** que contiene los mtodos necesarios para obtener una prediccin, la cual se obtiene haciendo uso del mtodo *double getPrediction(const vector<double>&inputs)* y el PLA se ejecuta haciendo uso del mtodo *void trainPerceptron(unsigned epochs, const vector<vector<double>>&inputs, const vector<double>&targets).*

1.4 Redes neuronales

1.4.1 Algoritmo Gentico (GANN)

2 Manual de utilizacin

2.1 Actividad principal de la empresa

2.2 Departamento del estudiante en prcticas

2.3 Infraestructura del centro, materiales y personal del lugar de trabajo

3 Experimentos realizados y resultados obtenidos

3.1 Experimento con el Perceptrn

El experimento que hemos realizado con el perceptrn ha sido sencillo, con el objetivo de comprobar que el funcionamiento era el esperado y, sobre todo, ver grficamente como el PLA iba evolucionando los pesos y cual iba siendo la evolucin. La funcin que hemos intentado aproximar con el perceptrn ha sido la funcin AND, la cual se describe en el algoritmo 8.

Algorithm 8: Funcin AND

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

La funcin AND es linealmente separable, por lo que el perceptrn debera de poder separar los puntos eventualmente. En la figura 27 se muestran los puntos que queremos separar (como subindice tienen el valor que se espera por el perceptrn).

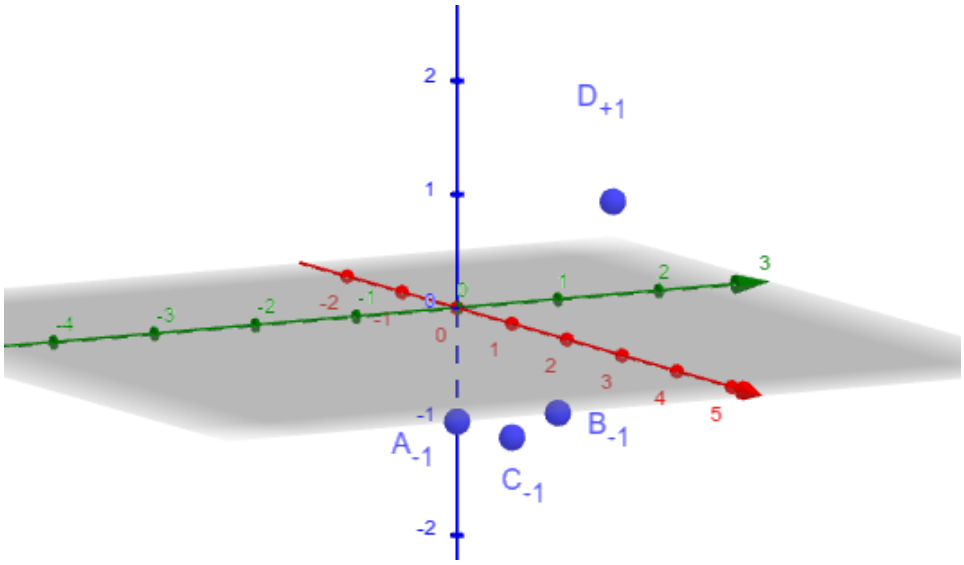
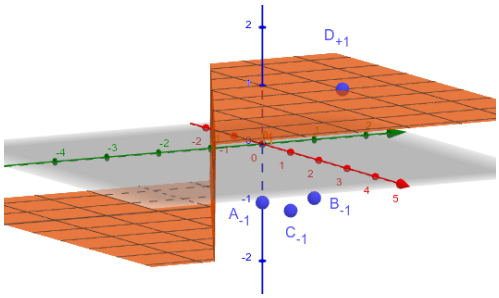
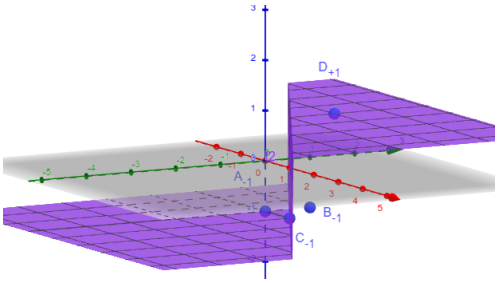


Figure 27: Puntos de la funcin AND

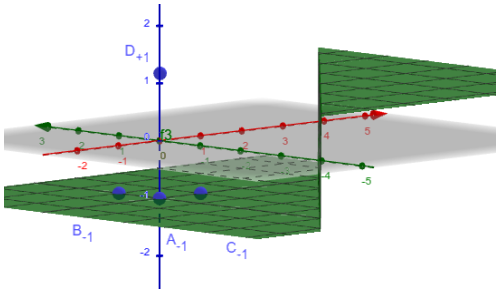
En la figura 28 podemos ver como el PLA ha ido ajustando los pesos para buscar el plano que separa correctamente estos puntos.



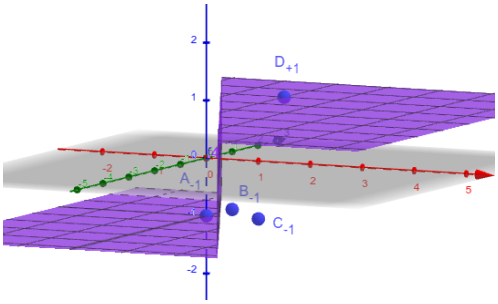
1a iteracin PLA (25% acierto)



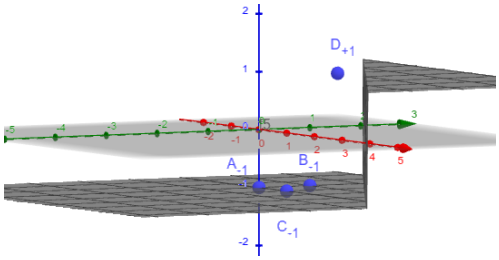
2a iteracin PLA (75% acierto)



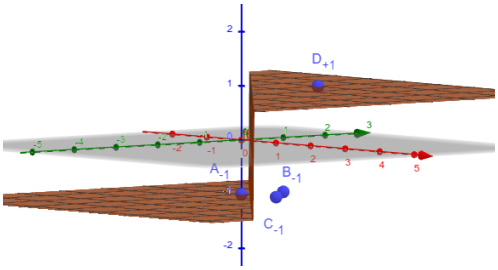
3a iteracin PLA (75% acierto)



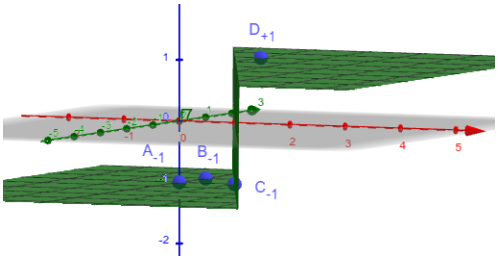
4a iteracin PLA (50% acierto)



5a iteracin PLA (75% acierto)



6a iteracin PLA (50% acierto)



7a iteracin PLA (100% acierto)

Figure 28: Iteraciones del PLA para la funcin AND

Algo interesante que podemos observar con este experimento es que el porcentaje de acierto no siempre aumenta (la razn ya se explic en la seccin 1.3), por lo que si en este experimento hubiramos configurado un nmero de pocas igual a 5 habramos obtenido como resultado final un 50% de aciertos, no un 75%, en el caso de que no hubiramos implementado el algoritmo del *Pocket*.

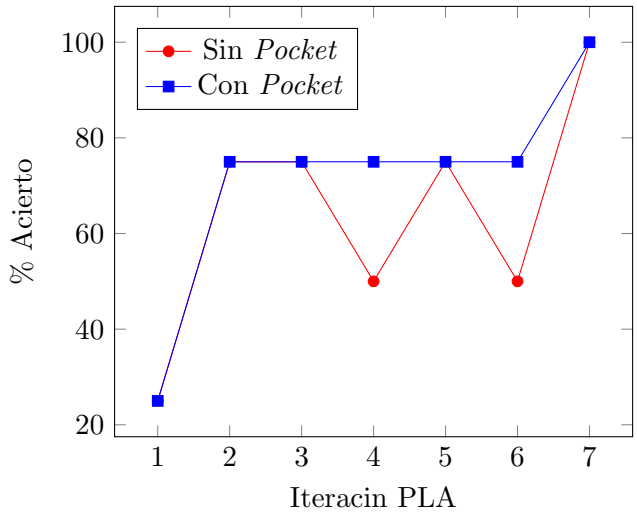


Figure 29: Resultados del PLA con y sin *Pocket*

Como ya hemos comentado, el algoritmo del *Pocket* tambin podemos verle la utilidad en este sencillo experimento. En la figura 29 se ve como a cada iteracin que se ejecuta del PLA (tambin se puede ver como una poca) 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 mayora de los problemas a los que se les puede aplicar el perceptrn y obtener unos resultados aceptables no sern linealmente separables.

3.2 Descripcin de las tareas y trabajos desarrollados

3.3 Descripcin de los conocimientos y competencias adquiridos

4 Conclusiones

4.1 Valoracin personal de las prcticas realizadas

4.2 Indicar qu ha echado de menos el alumno en la formacin recibida en la Universidad que considera le hubiera ayudado

4.3 Posibles sugerencias para mejorar las prcticas de empresa