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 2, 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.

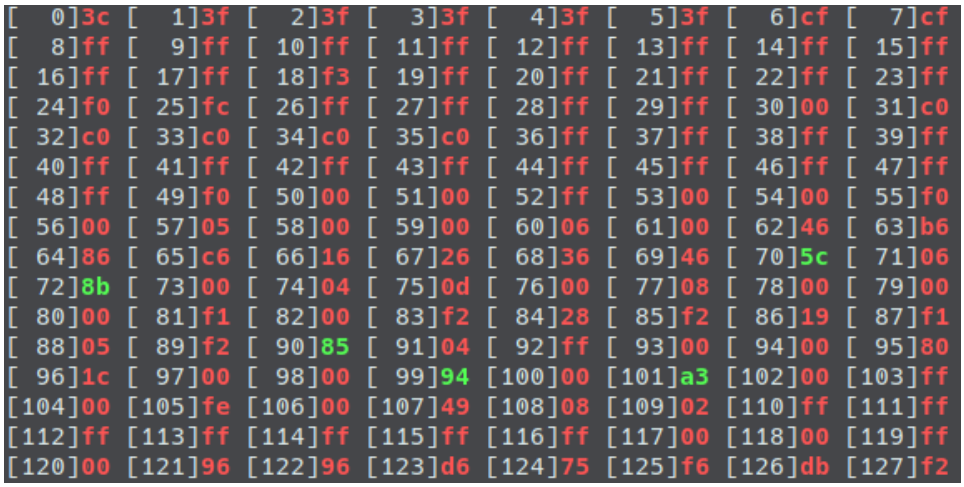

Figure 1: El color verde indica que el valor ha cambiado en este step.

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

Lorem ipsum dolor sit amet, consectetur adipiscing elit. Curabitur diam quam, placerat sit amet ornare ut, dignissim non orci. Suspendisse et pellentesque sem, id facilisis magna. Morbi consequat blandit ipsum, sed accumsan ex vehicula quis. Maecenas interdum malesuada neque, non laoreet magna dignissim a. Vivamus rhoncus enim eget neque pretium, condimentum hendrerit leo venenatis. Vestibulum elementum semper sem a ultricies. Pellentesque habitant morbi tristique senectus et netus et malesuada fames ac turpis egestas. Nullam pulvinar pellentesque quam viverra consectetur. Donec sit amet enim eu ante porttitor commodo elementum eget orci. Duis lobortis mattis nibh, sodales feugiat sapien efficitur sed. Duis ultrices turpis eu nisl maximus, id luctus justo elementum. Curabitur porttitor nec nibh at ullamcorper.

Aliquam egestas elit vitae faucibus tincidunt. Donec dolor nulla, congue quis lobortis quis, euismod at elit. Phasellus viverra auctor augue nec luctus. Aenean elementum facilisis sapien, a imperdiet lacus maximus varius. Mauris in maximus urna. Mauris quis varius mi, id commodo nibh. Vivamus et quam ultrices, sollicitudin justo a, egestas mauris.

Aenean ut egestas ipsum. Praesent placerat ex sit amet mauris eleifend auctor sit amet tincidunt nibh. Fusce in varius justo, eget euismod tortor. Pellentesque sagittis bibendum mi, vel mollis dolor congue a. Nulla vitae pharetra ex, quis commodo justo. Cras a volutpat nisi, vitae lobortis sem. Aenean ullamcorper, erat eget vehicula tempor, magna nunc rhoncus turpis, ac commodo dolor risus id sem. Morbi efficitur vel felis at finibus. Nullam tristique erat id est ornare, sed finibus libero bibendum. Cras at tellus ligula. Ut eget pretium ante, condimentum tristique nisi.

Quisque condimentum nunc id nisi sollicitudin elementum et at diam. Etiam facilisis cursus urna, mollis euismod magna efficitur at. Praesent ullamcorper varius accumsan. Sed quis est id orci sollicitudin sollicitudin. Donec consectetur, ante quis eleifend ultrices, risus turpis efficitur justo, eget fermentum ipsum nisi ac ipsum. Integer accumsan facilisis viverra. Aliquam id nisl faucibus, pharetra dui vel, venenatis elit. Morbi auctor, magna ac pellentesque fringilla, urna ex aliquam velit, ut lacinia enim diam nec augue. Nulla posuere auctor imperdiet. Praesent cursus cursus erat, ac varius odio vulputate in. Morbi ultricies consequat ex, eu convallis purus pretium non. Duis tincidunt nibh eget fermentum auctor.

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 4: 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 5: La ROM y el contenido multimedia corren a cargo de ALE.

Algunas de estas opciones corren a cargo del entorno (Figura 5), 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 6. 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 6: 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 video. 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 7: 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.

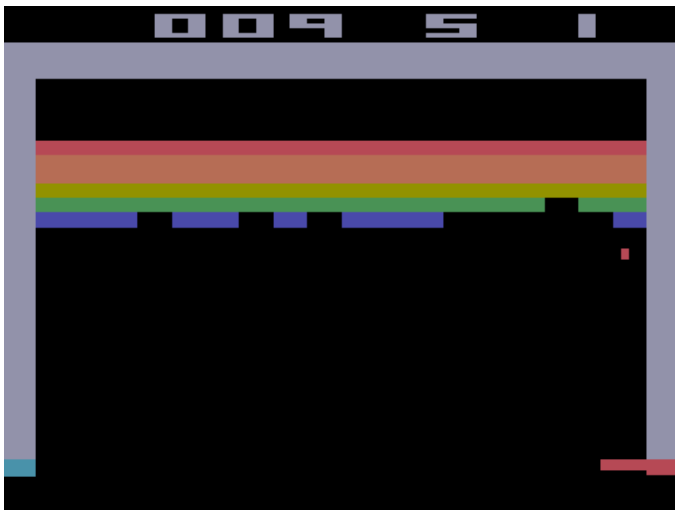

Figure 8: Juego Breakout.

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.

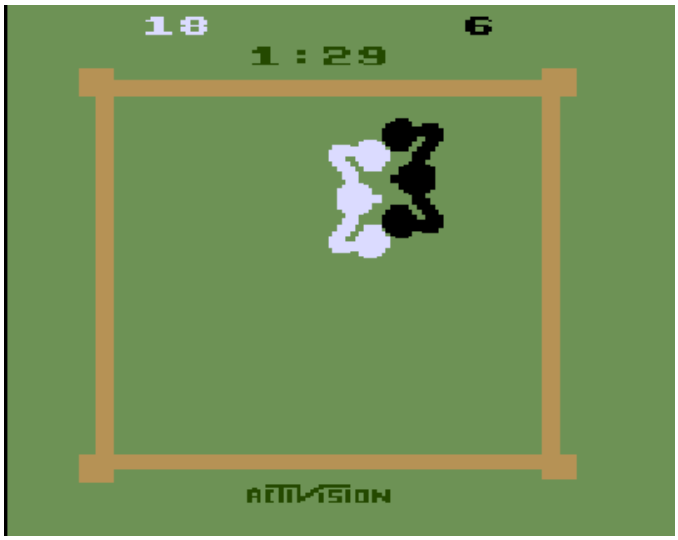

Figure 9: Juego Boxing.

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 10.

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

Figure 10: 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 11: 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 12 se puede observar el área de visión de la nave.

Figure 12: Vision 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

area 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 13: Constantes fijas relativas a la visión de la nave.

En la figura 13 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 14 .

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

Figure 14: Constantes modificables (no runtime).

La primera variable de la figura representa el área extra de visión 2 definida en la imagen 12, 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 15: 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 funcion **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 mini máquina 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 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 16: 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 resolucionar 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 pixeles 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 pixel 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 pre-calculado 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 17, por ello es necesario restar **VISION_THRESOLD**.,

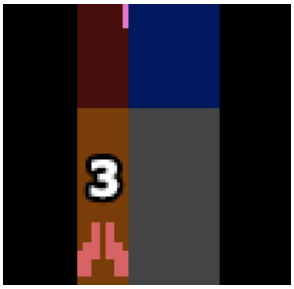

Figure 17: Area de sin restar VISION_THRESOLD al jugador.

Una vez hemos resolucionado el valor del pixel, 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 18: Resolución de la colisión

Como bien observamos en la figura 18, 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 16, 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 19: Aplicando movimiento a la nave

La parte que extiende a la figura 15, es la figura 19. 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 20: 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 laser. La solución completa y real sería llevar un seguimiento mas espécifico 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 Redes neuronales

2 Manual de utilización

2.1 Actividad principal de la empresa

2.2 Departamento del estudiante en prácticas

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

3 Experimentos realizados y resultados obtenidos

3.1 Objetivos planteados por la empresa al estudiante

3.2 Descripción de las tareas y trabajos desarrollados

3.3 Descripción de los conocimientos y competencias adquiridos

4 Conclusiones

4.1 Valoración personal de las prácticas realizadas

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

4.3 Posibles sugerencias para mejorar las prácticas de empresa